Copyrighted Material

# DIGITAL SYSTEMS DESIGN AND PROTOTYPING

## Using Field Programmable Logic and Hardware Description Languages

Zoran Salcic
Asim Smailagic

## SECOND EDITION

CD Included

Kluwer Academic Publishers

Copyrighted Material

# DIGITAL SYSTEMS DESIGN AND PROTOTYPING

## Using Field Programmable Logic and Hardware Description Languages

*Second Edition*

***Digital Systems Design and Prototyping: Using Field Programmable Logic and Hardware Description Languages, Second Edition*** includes a CD-ROM that contains Altera's MAX+PLUS II Student Edition programmable logic development software. MAX+PLUS II is a fully integrated design environment that offers unmatched flexibility and performance. The intuitive graphical interface is complemented by complete and instantly accessible on-line documentation, which makes learning and using MAX+PLUS II quick and easy. MAX+PLUS II version 9.23 Student Edition offers the following features:

- ✓ Operates on PCs running Windows 95/098, or Windows NT 4.0
- ✓ Graphical and text-based design entry, including the Altera Hardware Description Language (AHDL), VHDL and Verilog
- ✓ Design compilation for product-term (MAX 7000S) and look-up table (FLEX 10K) device architectures
- ✓ Design verification with functional and full timing simulation

The MAX+PLUS II Student Edition software is for students who are learning digital logic design. By entering the designs presented in the book or creating custom logic designs, students develop skills for prototyping digital systems using programmable logic devices.

**Registration and Additional Information**

To register and obtain an authorization code to use the MAX+PLUS II software, go to: ***http://www.altera.com/maxplus2-student.*** For complete installation instructions, refer to the ***read.me*** file on the CD-ROM or to the ***MAX+PLUS II Getting Started Manual*****,** available on the Altera world-wide web site **(http://www.altera.com).**

This CD-ROM is distributed by Kluwer Academic Publishers with *ABSOLUTELY NO SUPPORT* and *NO WARRANTY* from Kluwer Academic Publishers.

Kluwer Academic Publishers shall not be liable for damages in connection with, or arising out of, the furnishing, performance or use of this CD-ROM.

# DIGITAL SYSTEMS DESIGN AND PROTOTYPING

## Using Field Programmable Logic and Hardware Description Languages

*Second Edition*

**Zoran Salcic**
*The University of Auckland*

**Asim Smailagic**
*Carnegie Mellon University*

KLUWER ACADEMIC PUBLISHERS
NEW YORK, BOSTON, DORDRECHT, LONDON, MOSCOW

eBook ISBN: 0-306-47030-6
Print ISBN: 0-792-37920-9

©2002 Kluwer Academic Publishers
New York, Boston, Dordrecht, London, Mosh

The CD-ROM is only available in the print edition. Print ©2000 Kluwer Academic Publishers

All rights reserved

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without written consent from the Publisher

Created in the United States of America

Visit Kluwer Online at: http://kluweronline.com
and Kluwer's eBookstore at: http://ebooks.kluweronline.com

# Table of Contents

**PREFACE TO THE SECOND EDITION** **XVI**

**1 INTRODUCTION TO FIELD PROGRAMMABLE LOGIC DEVICES** **1**

**1.1. Introduction** **1**
1.1.1 Speed 4
1.1.2 Density 4
1.1.3 Development Time 5
1.1.4 Prototyping and Simulation Time 5
1.1.5 Manufacturing Time 6
1.1.6 Future Modifications 6
1.1.7 Inventory Risk 6
1.1.8 Cost 7

**1.2 Types of FPLDs** **7**
1.2.1 CPLDs 10
1.2.2 Static RAM FPGAs 11
1.2.3 Antifuse FPGAs 12

**1.3 Programming Technologies** **13**
1.3.1 SRAM Programming Technology 13
1.3.2 Floating Gate Programming Technology 15
1.3.3 Antifuse Programming Technology 16
1.3.4 Summary of Programming Technologies 17

**1.4. Logic Cell Architecture** **17**

**1.5 Routing Architecture** **26**

**1.6 Design Process** **30**

**1.7 FPLD Applications** **33**
1.7.1 Glue Random Logic Replacement 34
1.7.2 Hardware Accelerators 35
1.7.3 Non-standard Data Path/Control Unit Oriented Systems 36
1.7.4 Virtual Hardware 37
1.7.5 Custom-Computing Machines 38

**1.8 Questions and Problems** **39**

## 2 EXAMPLES OF MAJOR FPLD FAMILIES 43

**2.1 Altera MAX 7000 Devices** **43**
2.1.1 MAX 7000 devices general concepts 44
2.1.2 Macrocell 46
2.1.3 I/O Control Block 49
2.1.4 Logic Array Blocks 50
2.1.5 Programmable Interconnect Array 52
2.1.6 Programming 53

**2.2 Altera FLEX 8000** **54**
2.2.1 Logic Element 55
2.2.2 Logic Array Block 61
2.2.3 FastTrack Interconnect 62
2.2.4 Dedicated I/O Pins 65
2.2.5 Input/Output Element 65
2.2.6 Configuring FLEX 8000 Devices 67
2.2.7 Designing with FLEX 8000 Devices 72

**2.3 Altera FLEX 10K Devices** **75**
2.3.1 Embedded Array Block 75
2.3.2 Implementing Logic with EABs 77

**2.4 Altera APEX 20K Devices** **80**
2.4.1 General Organization 82
2.4.2 LUT-based Cores and Logic 82
2.4.3 Product-Term Cores and Logic 84
2.4.4 Memory Functions 87

**2.5 Xilinx XC4000 FPGAs** **91**
2.3.1 Configurable Logic Block 92
2.5.2 Input/Output Blocks 95
2.5.3 Programmable Interconnection Mechanism 97
2.5.4 Device Configuration 100
2.3.5 Designing with XC4000 Devices 101

**2.6 Xilinx Virtex FPGAs** **102**
2.6.1 General Organization 103
2.6.2 Configurable Logic Block 103
2.6.3 Input/Output Block 105
2.6.4 Memory Function 106

**2.7 Atmel AT40K Family** **107**
2.7.1 General Organization 107
2.7.2 Logic Cell 109
2.7.3 Memory Function 111
2.6.4 Dynamic Reconfiguration 112

**2.7 Problems and Questions** **112**

**3 DESIGN TOOLS AND LOGIC DESIGN WITH FPLDS** **115**

**3.1 Design Framework** **115**
3.1.1 Design Steps and Design Framework 116
3.1.2 Compiling and Netlisting 116

**3.2 Design Entry and High Level Modeling** **120**
3.2.1 Schematic Entry 121
3.2.2 Hardware Description Languages 122
3.2.3 Hierarchy of Design Units - Design Example 125

**3.3 Design Verification and Simulation** **129**

**3.4 Integrated Design Environment Example: Altera's Max+Plus II** **131**
3.4.1 Design Entry 133
3.4.2 Design Processing 134
3.4.3 Design Verification 137
3.4.4 Device Programming 139

**3.5 System prototyping: Altera UP1 Prototyping Board** **139**

**3.6 Questions and Problems** **142**

**4 INTRODUCTION TO DESIGN USING AHDL** **143**

**4.1. AHDL Design Entry** **143**
4.1.1 AHDL Design Structure 144
4.1.2 Describing Designs with AHDL 145

**4.2. AHDL Basics** **146**
4.2.1 Using Numbers and Constants 146
4.2.2 Combinational Logic 149
4.2.3 Declaring Nodes 150
4.2.4 Defining Groups 151

4.2.5 Conditional Logic 152
4.2.6 Decoders 154
4.2.7 Implementing Active-Low Logic 156
4.2.8 Implementing Bidirectional Pins 157

**4.3 Designing Sequential logic 159**
4.3.1 Declaring Registers and Registered Outputs 159
4.3.2 Creating Counters 162
4.3.3 Finite State Machines 163
4.3.4 State Machines with Synchronous Outputs – Moore Machines 168
4.3.5 State Machines with Asynchronous Outputs – Mealy Machines 171
4.3.6 More Hints for State Machine Description 172

**4.4 Problems and Questions 177**

**5 ADVANCED AHDL 185**

**5.1 Names and Reserved Keywords and Symbols 185**

**5.2 Boolean Expressions 188**

**5.3 Primitive Functions 191**
5.3.1 Buffer Primitives 191
5.3..2 Flip-flop and Latch Primitives 194
5.3.3 Macrofunctions 195
5.3.4 Logic Parameterized Modules 197
5.3.5 Ports 198

**5.4 Implementing a Hierarchical Project Using Altera-provided Functions 199**

**5.5 Creating and Using Custom Functions in AHDL 204**
5.5.1 Creation of Custom Functions 205
5.5.2 In-line References to Custom Functions 207
5.5.3 Using Instances of Custom Function 209

**5.6 Using Standard Parameterized Designs 210**
5.6.1 Using LPMs 210
5.6.2 Implementing RAM and ROM 212

**5.7 User-defined Parameterized Functions 213**

**5.8 Conditionally and Iteratively Generated Logic** **217**

**5.9 Problems and Questions** **220**

**6 DESIGN EXAMPLES** **223**

**6.1 Electronic Lock** **223**
6.1.1 Keypad encoder 224
6.1.2 Input Sequence Recognizer 229
6.1.3 Piezo Buzzer Driver 234
6.1.4 Integrated Electronic Lock 235

**6.2 Temperature Control System** **236**
6.2.1 Temperature Sensing and Measurement Circuitry 238
6.2.2 Keypad Control Circuitry 240
6.2.3 Display Circuitry 241
6.2.4 Fan and Lamp Control Circuitry 243
6.2.5 Control Unit 244
6.2.6 Temperature Control System Design 248

**6.3 Problems and Questions** **253**

**7 SIMP - A SIMPLE CUSTOMIZABLE MICROPROCESSOR 255**

**7.1 Basic Features** **255**
7.1.1 Instruction Formats and Instruction Set 256
7.1.2 Register Set 259

**7.2 Processor Data Path** **259**

**7.3 Instruction Execution** **262**

**7.4 SimP Implementation** **267**
7.4.1 Data Path Implementation 267
7.4.2 Control Unit Implementation 276
7.4.3 Synthesis Results 290

**7.5 Questions and Problems** **291**

**8 RAPID PROTOTYPING USING FPLDS - VUMAN CASE STUDY 295**

**8.1 System Overview 295**

**8.2 Memory Interface Logic 298**

**8.3 Private Eye Controller 305**

**8.4 Secondary Logic 311**

**8.5 Questions and Problems 311**

**9 INTRODUCTION TO VHDL 313**

**9.1 What is VHDL for? 314**

**9.2 VHDL Designs 317**

**9.3 Library 320**

**9.4 Package 321**

**9.5 Entity 322**

**9.6 Architecture 324**
9.6.1 Behavioral Style Architecture 326
9.6.2 Dataflow Style Architecture 327
9.6.3 Structural Style Architecture 328

**9.7 Configuration 329**

**9.8 Questions and Problems 330**

**10 OBJECTS, DATA TYPES AND PROCESSES 333**

**10.1 Literals 334**
10.1.1 Character and String Literals 334
10.1.2 Bit, Bit String and Boolean Literals 335
10.1.3 Numeric Literals 336
10.1.4 Physical literals 336

10.1.5 Range Constraint 337
10.1.6 Comments 337

**10.2 Objects in VHDL 337**
10.2.1 Names and Named Objects 337
10.2.2 Indexed names 338
10.2.3 Constants 339
10.2.4 Variables 339
10.2.5 Signals 340

**10.3 Expressions 340**

**10.4 Basic Data Types 341**
10.4.1 Bit Type 344
10.4.2 Character Type 345
10.4.3 Boolean Type 346
10.4.4 Integer Type 346
10.4.5 Real Types 347
10.4.6 Severity_Level Type 347
10.4.7 Time Type 348

**10.5 Extended Types 348**
10.5.1 Enumerated Types 348
10.5.2 Qualified Expressions 350
10.5.3 Physical Types 351

**10.6 Composite Types - Arrays 351**
10.6.1 Aggregates 352
10.6.2 Array Type Declaration 353

**10.7 Records and Aliases 354**

**10.8 Symbolic Attributes 355**

**10.9 Standard Logic 360**
10.9.1 IEEE Standard 1164 360
10.9.2 Standard Logic Data Types 362
10.9.3 Standard Logic Operators and Functions 365
10.9.4 IEEE Standard 1076.3 (The Numeric Standard) 367
10.9.5 Numeric Standard Operators and Functions 368

**10.10 Type Conversions 371**

**10.11 Process Statement and Processes** **374**

**10.12 Sequential Statements** **376**
10.12.1 Variable Assignment Statement 376
10.12.2 If Statement 377
10.12.3 Case Statement 377
10.12.4 Loop Statement 379
10.12.5 Next Statement 380
10.12.6 Exit Statement 380
10.12.7 Null statement 381
10.12.8 Assert Statement 381

**10.13 Wait Statement** **382**

**10.14 Subprograms** **384**
10.14.1 Functions 385
10.14.2 Procedures 386

**10.15 Questions and Problems** **387**

**11 VHDL AND LOGIC SYNTHESIS** **391**

**11.1 Specifics of Altera's VHDL** **391**

**11.2 Combinational Logic Implementation** **392**
11.2.1 Logic and Arithmetic Expressions 393
11.2.2 Conditional Logic 398
11.2.3 Three-State Logic 401
11.2.4 Combinational Logic Replication 405
11.2.5 Examples of Standard Combinational Blocks 408

**11.3 Sequential Logic Synthesis** **415**
11.3.1 Describing Behavior of Basic Sequential Elements 416
11.3.2 Latches 418
11.3.3 Registers and Counters Synthesis 421
11.3.4 Examples of Standard Sequential Blocks 425

**11.4 Finite State Machines Synthesis** **431**
11.4.1 State assignments 433
11.4.2 Using Feedback Mechanisms 436
11.4.3 Moore Machines 440
11.4.4 Mealy Machines 441

**11.5 Hierarchical Projects** **442**
11.5.1 Max+Plus II Primitives 443
11.5.2 Max+Plus II Macrofunctions 443

**11.6 Using Parameterized Modules and Megafunctions** **449**

**11.8 Questions and Problems** **455**

**12 EXAMPLE DESIGNS AND PROBLEMS** **459**

**12.1 Sequence Recognizer and Classifier** **459**
12.1.1 Input Code Classifier 461
12.1.2 Sequence Recognizer 462
12.1.3 BCD Counter 466
12.1.4 Display Controller 468
12.1.5 Circuit Integration 470

**12.2 SART – A Simple Asynchronous Receiver-Transmitter** **475**
12.2.1 SART Global Organization 476
12.2.2 Baud Rate Generator 477
12.2.3 SART Transmitter 480
12.2.4 SART Receiver 484

**12.3 Questions and Problems** **490**

**13 INTRODUCTION TO VERILOG HDL** **493**

**13.1 What is Verilog HDL?** **493**

**13.2 Basic Data Types and Objects** **494**
13.2.1 Nets 495
13.2.2 Registers 496
13.2.3 Parameters 497
13.2.4 Literals 497

**13.3 Complex Data Types** **499**
13.3.1 Vectors 499
13.3.2 Arrays 499
13.3.3 Memories 500
13.3.4 Tri-state 500

**13.4 Operators** **501**

13.4.1 Arithmetic operators 501
13.4.2 Logical Operators 502
13.4.3 Relational Operators 503
13.4.4 Equality operators 503
13.4.5 Bitwise Operators 504
13.4.6 Reduction Operators 505
**13.4.7 Shift Operators** 505
13.4.8 Concatenation operator 506
13.4.9 Replication operator 507

**13.5 Design Blocks and Ports** **507**
13.5.1 .Modules 507
13.5.2 Ports 511

**13.6 Procedural Statements** **513**
13.6.1 Selection - if and case Statements 513
13.6.2 Repetition - for, while, repeat and forever Statements 514

**13.7 Simulation Using Verilog** **523**
13.7.1 Writing to Standard Output 524
13.7.2 Monitoring and Ending Simulation 525

**13.8 Questions and Problems** **527**

**14 VERILOG AND LOGIC SYNTHESIS BY EXAMPLES 529**

**14.1 Specifics of Altera's Verilog HDL** **529**

**14.2 Combinational Logic Implementation** **530**
14.2.1 Logic and Arithmetic Expressions 530
14.2.2 Conditional Logic 533
14.2.3 Three-State Logic 534
14.2.4 Examples of Standard Combinational Blocks 535

**14.3 Sequential Logic Synthesis** **540**
14.3.1 Describing Behavior of Basic Sequential Elements 540
14.3.2 Latches 541
14.3.3 Registers and Counters Synthesis 542
14.3.4 Examples of Standard Sequential Blocks 545

**14.4 Finite State Machines Synthesis** **548**
14.4.1 Verilog FSM Example 548
14.4.2 Moore Machines 550

14.4.3 Mealy Machines 553

**14.5 Hierarchical Projects 555**
14.5.1 User Defined Functions 555
14.5.2 Using Parameterized Modules and Megafunctions 556

**14.6 Questions and Problems 557**

**15 A VERILOG EXAMPLE: PIPELINED SIMP 559**

**15.1 SimP Pipelined Architecture 559**
15.1.1 Memory conflicts 560
15.1.2 Stage Registers 561
15.1.3 Branching Instructions 562

**15.2 Pipelined SimP Design 562**
15.2.1 Data Path 563
15.2.2 Control Unit 565

**15.3 Pipelined SimP Implementation 570**
15.3.1 Data Path Design 571
15.3.2 Control Unit Design 578

**15.4 Questions and Problems 595**

**GLOSSARY 597**

**SELECTED READING 609**

**WEB RESOURCES 613**

**INDEX 617**

# PREFACE TO THE SECOND EDITION

As the response to the first edition of the book has been positive, we felt it was our obligation to respond with this second edition. The task of writing has never been easy, because at the moment you think and believe the manuscript has been finished, and is ready for printing, you realize that many things could be better and get ideas for further improvements and modifications. The digital systems design field is such an area in which there is no end. Our belief is that with this second edition we have succeeded to improve the book and performed all those modifications we found necessary, or our numerous colleagues suggested to do. This edition comprises a number of changes in an attempt to make it more readable and useful for teaching purposes, but also to numerous engineers who are entering the field of digital systems design and field-programmable logic devices (FPLDs). In that context, the second edition contains seven additional chapters, updated information on the current developments in the area of FPLDs and the examples of the most recent developments that lead towards very complex system-on-chip solutions on FPLDs. Some of the new design examples and suggested problems are just pointing to the direction of system-on-chip. Number of examples is further increased as we think that the best learning is by examples. Besides further emphasis on AHDL, as the main language for design specification, a further extension of presentation of two other hardware description languages, VHDL and Verilog, is introduced. However, in order to preserve complementarity with another book "VHDL and FPLDs in Digital Systems Design, prototyping and Customization" (Zoran Salcic, Kluwer Academic Publishers, 1998) presentation of VHDL is oriented mostly towards synthesizable designs in FPLDs.

This book focuses on digital systems design and FPLDs combining them into an entity useful for designers in the areas of digital systems and rapid system prototyping. It is also useful for the growing community of engineers and researchers dealing with the exciting field of FPLDs, reconfigurable, and programmable logic. Our goal is to bring these areas to the students studying digital system design, computer design, and related topics, as to show how very complex circuits can be implemented at the desk. Hardware and software designers are

getting closer every day by the emerging technologies of in-circuit reconfigurable and in-system programmable logic of very high complexity.

Field-programmable logic has been available for a number of years. The role of FPLDs has evolved from simply implementing the system "glue-logic" to the ability to implement very complex system functions, such as microprocessors and microcomputers. The speed with which these devices can be programmed makes them ideal for prototyping and education. Low production cost makes them competitive for small to medium volume productions. These devices make possible new sophisticated applications and bring-up new hardware/software trade-offs and diminish the traditional hardware/software demarcation line. Advanced design tools are being developed for automatic compilation of complex designs and routing to custom circuits.

To our knowledge, this book makes a pioneering effort to present rapid prototyping and generation of computer systems using FPLDs. Rapid prototyping systems composed of programmable components show great potential for full implementation of microelectronics designs. Prototyping systems based on FPLDs present many technical challenges affecting system utilization and performance.

The book contains fifteen chapters. Chapter 1 represents an introduction into the field-programmable logic. Main types of FPLDs are introduced, including programming technologies, logic cell architectures, and routing architectures used to interconnect logic cells. Architectural features are discussed to allow the reader to compare different devices appearing on the market, sometimes using confusing terminology and hiding the real nature of the devices. Also, the main characteristics of the design process using FPLDs are discussed and the differences to the design for custom integrated circuits underlined. The necessity to introduce and use new advanced tools when designing complex digital systems is also emphasized. New section on typical applications is introduced to show in the very beginning where FPLDs and complex system design are directed to.

Chapter 2 describes the field-programmable devices of the three major manufacturers in the market, Altera, Xilinx and Atmel. It does not mean that devices from other manufacturers are inferior to presented ones. The purpose of this book is not to compare different devices, but to emphasize the most important features found in the majority of FPLDs, and their use in complex digital system prototyping and design. Altera and Xilinx invented some of the concepts found in major types of field-programmable logic and also produce devices which employ all major programming technologies. Complex Programmable Logic Devices (CPLDs) and Field-Programmable Gate Arrays (FPGAs) are presented in Chapter 2, along with their main architectural and application-oriented features. Although sometimes we use different names to distinguish CPLDs and FPGAs, usually with the term FPLD we will refer to both types of devices. Atmel's devices, on the other hand,

give an option of partial reconfiguration, which makes them potential candidates for a range of new applications.

Chapter 3 covers aspects of the design methodology and design tools used to design with FPLDs. The need for tightly coupled design frameworks, or environments, is discussed and the hierarchical nature of digital systems design. All major design description (entry) tools are briefly introduced including schematic entry tools and hardware description languages. The complete design procedure, which includes design entry, processing, and verification, is shown in an example of a simple digital system. An integrated design environment for FPLD-based designs, the Altera's Max+Plus II environment, is introduced. It includes various design entry, processing, and verification tools. Also, a typical prototyping system, Altera's UP1 board is described as it will be used by many who will try designs presented in the book or make their own designs.

Chapter 4 is devoted to the design using Altera's Hardware Description Language (AHDL). First, the basic features of AHDL are introduced without a formal presentation of the language. Small examples are used to illustrate its features and how they are used. The readers can intuitively understand language and its syntax by examples. The methods for design of combinatorial logic in AHDL, including the implementation of bidirectional pins, standard sequential circuits such as registers and counters, and state machines is presented.

Chapter 5 introduces more advanced features of AHDL. Vendor supplied and user defined macrofunctions appear as library entities. The implementation of user designs as hierarchical projects consisting of a number of subdesigns is also shown. AHDL, as a lower level hardware description language, allows user control of resource assignments and very effective control of the design fit to target either speed or size optimization. Still, the designs specified in AHDL can be of behavioral or structural type and easily retargeted, without change, to another device without the need for the change of the design specification. New AHDL features that enable parameterized designs, as well as conditional generation of logic, are introduced. They provide mechanisms for design of more general digital circuits and systems that are customized at the time of use and compilation of the design.

Chapter 6 shows how designs can be handled using primarily AHDL, but also in the combination with the more convenient schematic entry tools. Two relatively simple design case studies, which include a number of combinational and sequential circuit designs are shown in this chapter. The first example is an electronic lock which consists of a hexadecimal keypad as the basic input device and a number of LEDs as the output indicators of different states. The lock activates an unlock signal after recognizing the input of a sequence of five digits acting as a kind of password. The second example is a temperature control system, which enables temperature

control in a small chamber (incubator). The temperature controller continuously scans the current temperature and activates one of two actuators, a lamp for heating or a fan for cooling. The controller allows set up of a low and high temperature limit range where the current temperature should be maintained. It also provides the basic interface with the operator in the form of hexadecimal keypad as input and 7-segment display and couple of LEDs as output. Both designs fit into the standard Altera's devices.

Chapter 7 includes a more complex example of a simple custom configurable microprocessor called SimP. The microprocessor contains a fixed core that implements a set of instructions and addressing modes, which serve as the base for more complex microprocessors with additional instructions and processing capabilities as needed by a user and/or application. It provides the mechanisms to be extended by the designers in various directions and with some further modifications it can be converted to become a sort of dynamically reconfigurable processor. Most of the design is specified in AHDL to demonstrate the power of the language.

Chapter 8 is used to present a case study of a digital system based on the combination of a standard microprocessor and FPLD implemented logic. The VuMan wearable computer, developed at Carnegie Mellon University (CMU), is presented in this chapter. Examples of the VuMan include the design of memory interfacing logic and a peripheral controller for the Private Eye head-on display are shown. FPLDs are used as the most appropriate prototyping and implementation technology.

Although AHDL represents an ideal vehicle for learning design with hardware description languages (HDLs), it is Altera proprietary language and as such can not be used for other target technologies. That is the reason to expand VHDL presentation in the second part of the book. Chapter 9 provides an introduction to VHDL as a more abstract and powerful hardware description language, which is also adopted as an IEEE standard. The goal of this chapter is to demonstrate how VHDL can be used in digital system design. A subset of the language features is used to provide designs that can almost always be synthesized. The features of sequential and concurrent statements, objects, entities, architectures, and configurations, allow very abstract approaches to system design, at the same time controlling design in terms of versions, reusability, or exchangeability of the portions of design. Combined with the flexibility and potential reconfigurability of FPLDs, VHDL represents a tool which will be more and more in use in digital system prototyping and design. This chapter also makes a bridge between a proprietary and a standard HDLs.

Chapter 10 introduces all major mechanisms of VHDL used in description and design of digital systems. It emphasizes those feature not found in AHDL, such as objects and data types. As VHDL is object oriented language, it provides the use of

a much higher level of abstraction in describing digital systems. The use of basic objects, such as constants, signals and variables is introduced. Mechanisms that allow user own data types enable simpler modeling and much more designer friendly descriptions of designs. Finally, behavioral modeling enabled by processes as the basic mechanism for describing concurrency is presented.

Chapter 11 goes a step further to explain how synthesis from VHDL descriptions is made. This becomes important especially for those who are not interested for VHDL as description, documentation or simulation tool, but whose goal is synthesized design. Numerous examples are used to show how synthesizable combinational and standard sequential circuits are described. Also, finite state machines and typical models for Moore and Mealy machine descriptions are shown.

In Chapter 12 we introduce two full examples. The first example of an input sequence classifier and recognizer is used to demonstrate the use of VHDL in digital systems design that are easily implemented in FPLDs. As the system contains a hierarchy of subsystems, it is also used to demonstrate a typical approach in digital systems design when using VHDL. The second example is of a simple asynchronous receiver/transmitter (SART) for serial data transfers. This example is used to further demonstrate decomposition of a digital system into its parts and integration at a higher level and the use of behavioral modeling and processes. It also opens addition of further user options to make as sophisticated serial receiver/transmitter as required.

Chapter 13 presents the third hardware description language with wide spread use in industry - Verilog HDL. Presentation of Verilog is mostly restricted to a subset useful for synthesis of digital systems. Basic features of the language are presented and their utilization shown.

Chapter 14 is oriented only towards synthesizable models in Verilog. A number of standard combinational and sequential circuits is described by synthesizable models. Those examples provide a clear parallel with modeling the same circuits using other HDLs and demonstrate power and simplicity of Verilog. They also show why many hardware designers prefer Verilog over VHDL as the language that is primarily suited for digital hardware design.

Final Chapter 15 is dedicated to the design of a more complex digital system. The SimP microprocessor, introduced in Chapter 7 as an example of a simple general purpose processor, is redesigned introducing pipelining. Advantages of Verilog as the language suitable for both behavioral and structural modeling are clearly demonstrated. The pipelined SimP model represents a good base for further experiments with the SimP open architecture and its customization in any desired direction.

The problems given at the end of each chapter are usually linked to and require extension to examples presented within that or other chapters. By solving them, the reader will have opportunity to further develop his own skills and feel the real power of both HDLs and FPLDs as implementation technology. By going through the whole design process from its description and entry simulation and real implementation, the reader will get his own ideas how to use all these technologies in the best way.

The book is based on lectures we have taught in different courses at Auckland University and CMU, various projects carried out in the course of different degrees, and the courses for professional engineers who are entering the field of FPLDs and CAD tools for complex digital systems design. As with any book, it is still open and can be improved and enriched with new materials, especially due to the fact that the subject area is rapidly changing. The complete Chapter 8 represents a portion of the VuMan project carried out at Carnegie Mellon University. Some of the original VuMan designs are modified for the purpose of this book at Auckland University.

A special gratitude is directed to the Altera Corporation for enabling us to try many of the concepts using their tools and devices in the course of its University Program Grant and for providing design software on CD ROM included with this book. Also Altera made possible the opportunity for numerous students at Auckland University to take part in various courses designing digital systems using these new technologies. The thank also goes to a number of reviewers and colleagues who gave valuable suggestions. We believe that the book will meet their expectations. This book would not be possible without the supportive environment at Auckland University and Carnegie Mellon University as well as early support from Cambridge University, Czech Technical University, University of Edinburgh, and Sarajevo University where we spent memorable years teaching and conducting research.

At the end, when we analyze the final manuscript as it will be printed, the book looks more as a completely new one than as the second edition of original one. Still, as it owes to its predecessor, we preserved the main title. However, the subtitle reflects its shift of the ballance to hardware description languages as we explained in this preface.

Z. A. Salcic
Auckland, New Zealand

A. Smailagic
Pittsburgh, USA

# 1 INTRODUCTION TO FIELD PROGRAMMABLE LOGIC DEVICES

Programmable logic design is beginning the same paradigm shift that drove the success of logic synthesis within ASIC design, namely the move from schematics to HDL based design tools and methodologies. Technology advancements, such as 0.25 micron five level metal processing and architectural innovations such as large amount of on-chip memory, have significantly broadened the applications for Field-Programmable Logic Devices (FPLDs).

This chapter represents an introduction to the Field-Programmable Logic. The main types of FPLDs are introduced, including programming technologies, logic cell architectures, and routing architectures used to interconnect logic cells. Architectural features are discussed to allow the reader to compare different devices appearing on the market. The main characteristics of the design process using FPLDs are also discussed and the differences to the design for custom integrated circuits underlined. In addition, the necessity to introduce and use new advanced tools when designing complex digital systems is emphasized.

## 1.1. Introduction

FPLDs represent a relatively new development in the field of VLSI circuits. They implement thousands of logic gates in multilevel structures. The architecture of an FPLD, similar to that of a Mask-Programmable Logic Device (MPLD), consists of an array of logic cells that can be interconnected by programming to implement different designs. The major difference between an FPLD and an MPLD is that an MPLD is programmed using integrated circuit fabrication to form metal interconnections while an FPLD is programmed using electrically programmable switches similar to ones in traditional Programmable Logic Devices (PLDs). FPLDs can achieve much higher levels of integration than traditional PLDs due to their more complex routing architectures and logic implementation. The first PLD developed for implementing logic circuits was the field-Programmable Logic Array (PLA). A PLA is implemented using AND-OR logic with wide input programmable AND gates followed by a programmable OR gate plane. PLA routing architectures

are very simple with inefficient crossbar like structures in which every output is connectable to every input through one switch. As such, PLAs are suitable for implementing logic in two-level sum-of-products form. The next step in PLDs development was introduction of Programmable Array Logic (PLA) devices with a single level of programmability - programmable AND gates followed by fixed OR gates. In order to allow implementation of sequential circuits, OR gates are usually followed by flip-flops. A variant of the basic PLD architectures appears in several today's FPLDs. FPLD combines multiple simple PLDs on a single chip using programmable interconnect structures. Today such combinations are known as Complex PLDs (or CPLDs) with the capacities equivalent to tens of simple FPLDs. FPLD routing architectures provide a more efficient MPLD-like routing where each connection typically passes through several switches. FPLD logic is implemented using multiple levels of lower fan-in gates which is often more compact than two-level implementations. Building FPLDs with very high capacity requires a different approach, more similar to Mask-Programmable Gate Arrays (MPGAs) that are the highest capacity general-purpose logic chips. As a MPGA consists of an array of prefabricated transistors, that are customized for user logic by means of wire connections, customization during chip fabrication is required. An FPLD which is the field-programmable equivalent of an MPGA is very often known as an FPGA. The end user configures an FPGA through programming. In this text we use the FPLD as a term that covers all field-programmable logic devices including CPLDs and FPGAs.

An FPLD manufacturer makes a single, standard device that users program to carry out desired functions. Field programmability comes at a cost in logic density and performance. FPLD capacity trails MPLD capacity by about a factor of 10 and FPLD performance trails MPLD performance by about a factor of three. Why then FPLDs? FPLDs can be programmed in seconds rather than weeks, minutes rather than the months required for production of mask-programmed parts. Programming is done by end users at their site with no IC masking steps. FPLDs are currently available in densities over 100,000 gates in a single device. This size is large enough to implement many digital systems on a single chip and larger systems can be implemented on multiple FPLDs on the standard PCB or in the form of Multi-Chip Modules (MCM). Although the unit costs of an FPLD is higher than an MPLD of the same density, there is no up-front engineering charges to use an FPLD, so they are more cost-effective for many applications. The result is a low-risk design style, where the price of logic error is small, both in money and project delay.

FPLDs are useful for rapid product development and prototyping. They provide very fast design cycles, and, in the case that the major value of the product is in algorithms or fast time-to-market they prove to be even cost-effective as the final deliverable product. Since FPLDs are fully tested after manufacture, user designs do not require test program generation, automatic test pattern generation, and design for testability. Some FPLDs have found a suitable place in designs that require

reconfiguration of the hardware structure during system operation, functionality can change "on the fly."

An illustration of device options ratings, that include standard discrete logic, FPLDs, and custom logic is given in Figure 1.1. Although not quantitative, the figure demonstrates many advantages of FPLDs over other types of available logic.

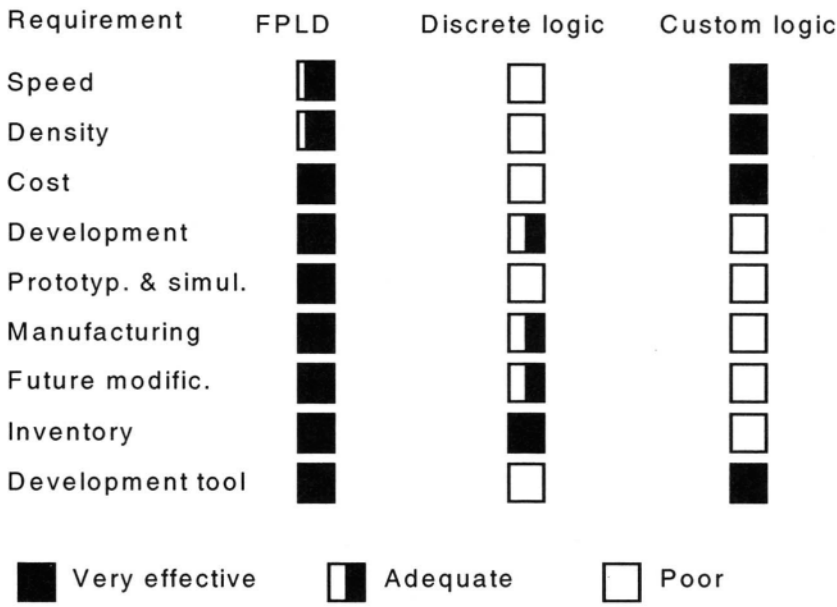



Figure 1.1 Device options ratings for different device technologies

The purpose of Figure 1.1 and this discussion is to point out some of the major features of currently used options for digital system design, and show why we consider FPLDs as the most promising technology for implementation of a very large number of digital systems.

Until recently only two major options were available to digital system designers.

- First, they could use Small-Scale Integrated (SSI) and Medium-Scale Integrated (MSI) circuits to implement a relatively small amount of logic with a large number of devices.
- Second, they could use a Masked-Programmed Gate Array (MPGA) or simply gate array to implement tens or hundreds of thousands of logic gates on a single integrated circuit in multi-level logic with wiring between logic

levels. The wiring of logic is built during the manufacturing process requiring a custom mask for the wiring. The low volume MPGAs have been expensive due to high mask-making charges.

As intermediate solutions for the period during the 1980s and early 1990s various kinds of simple PLDsm(PLAs, PALs) were available. A simple PLD is a general purpose logic device capable implementing the logic of tens or hundreds of SSI circuits and customize logic functions in the field using inexpensive programming hardware. Large designs require a multi-level logic implementation introducing high power consumption and large delays.

FPLDs offer the benefits of both PLDs and MPLDs. They allow the implementation of thousands of logic gates in a single circuit and can be programmed by designers on the site not requiring expensive manufacturing processes. The discussion below is largely targeted to a comparison of FPLDs and MPLDs as the technologies suitable for complex digital system design and implementation.

### *1.1.1 Speed*

FPLDs offer devices that operate at speeds exceeding 200 MHz in many applications. Obviously, speeds are higher than in systems implemented by SSI circuits, but lower than the speeds of MPLDs. The main reason for this comes from the FPLD programmability. Programmable interconnect points add resistance to the internal path, while programming points in the interconnect mechanism add capacitance to the internal path. Despite these disadvantages when compared to MPLDs, FPLD speed is adequate for most applications. Also, some dedicated architectural features of FPLDs can eliminate unneeded programmability in speed critical paths.

By moving FPLDs to faster processes, application speed can be increased by simply buying and using a faster device without design modification. The situation with MPLDs is quite different; new processes require new mask-making and increase the overall product cost.

### *1.1.2 Density*

FPLD programmability introduces on-chip programming overhead circuitry requiring area that cannot be used by designers. As a result, the same amount of logic for FPLDs will always be larger and more expensive than MPLDs. However, a large area of the die cannot be used for core functions in MPLDs due to the I/O

pad limitations. The use of this wasted area for field programmability does not result in an increase of area for the resulting FPLD. Thus, for a given number of gates, the size of an MPLD and FPLD is dictated by the I/O count so the FPLD and MPLD capacity will be the same. This is especially true with the migration of FPLDs to submicron processes. MPLD manufacturers have already shifted to high-density products leaving designs with less than 20,000 gates to FPLDs.

### *1.1.3 Development Time*

FPLD development is followed by the development of tools for system designs. All those tools belong to high-level tools affordable even to very small design houses. The development time primarily includes prototyping and simulation while the other phases, including time-consuming test pattern generation, mask-making, wafer fabrication, packaging, and testing are completely avoided. This leads to the typical development times for FPLD designs measured in days or weeks, in contrast to MPLD development times in several weeks or months.

### *1.1.4 Prototyping and Simulation Time*

While the MPLD manufacturing process takes weeks or months from design completion to the delivery of finished parts, FPLDs require only design completion. Modifications to correct a design flaw are quickly and easily done providing a short turn around time that leads to faster product development and shorter time-to-market for new FPLD-based products.

Proper verification requires MPLD users to verify their designs by extensive simulation before manufacture introducing all of the drawbacks of the speed/accuracy trade-off connected with any simulation. In contrast, FPLDs simulations are much simpler due to the fact that timing characteristics and models are known in advance. Also, many designers avoid simulation completely and choose in-circuit verification. They implement the design and use a functioning part as a prototype that operates at full speed and absolute time accuracy. A prototype can be easily changed and reinserted into the system within minutes or hours.

FPLDs provide low-cost prototyping, while MPLDs provide low-cost volume production. This leads to prototyping on an FPLD and then switching to an MPLD for volume production. Usually there is no need for design modification when retargeting to an MPLD, except sometimes when timing path verification fails. Some FPLD vendors offer mask-programmed versions of their FPLDs giving users flexibility and advantages of both implementation methods.

### *1.1.5 Manufacturing Time*

All integrated circuits must be tested to verify manufacturing and packaging. The test is different for each design. MPLDs typically incur three types of costs associated with testing.

- on-chip logic to enable easier testing
- generation of test programs for each design
- testing the parts when manufacturing is complete

Because they have a simple and repeatable structure, the test program for one FPLD device is same for all designs and all users of that part. It further justifies all reasonable efforts and investments to produce extensive and high quality test programs that will be used during the lifetime of the FPLD. Users are not required to write design specific tests because manufacturer testing verifies that every FPLD will function for all possible designs implemented. The consequences of manufacturing chips from both categories are obvious. Once verified, FPLDs can be manufactured in any quantity and delivered as fully tested parts ready for design implementation while MPLDs require separate production preparation for each new design.

### *1.1.6 Future Modifications*

Instead of customizing the part in the manufacturing process as for MPLDs, FPLDs are customized by electrical modifications. The electrical customization takes milliseconds or minutes and can even be performed without special devices, or with low cost programming devices. Even more, it can usually be performed in-system, meaning that the part can already be on the printed circuit board reducing the danger of the damage due to uncareful handling. On the other hand, every modified design to be implemented in an MPLD requires a custom mask that costs several thousands dollars that can only be amortized over the total number of units manufactured.

### *1.1.7 Inventory Risk*

An important feature of FPLDs is low inventory risk, similar to SSI and MSI parts. Since actual manufacturing is done at the time of programming a device, the same part can be used for different functionality and different designs. This is not found in an MPLD since the functionality and application is fixed forever once it is produced. Also, the decision on the volume of MPLDs must be made well in

advance of the delivery date, requiring concern with the probability that too many or not enough parts are ordered to manufacture. Generally, FPLDs are connected with very low risk design in terms of both money and delays. Rapid and easy prototyping enables all errors to be corrected with short delays, but also gives designers the chance to try more risky logic designs in the early stages of product development. Development tools used for FPLD designs usually integrate the whole range of design entry, processing, and simulation tools which enable easy reusability of all parts of a correct design.

FPLD designs can be made with the same design entry tools used in traditional MPLDs and Application Specific Integrated Circuits (ASICs) development. The resulting netlist is further manipulated by FPLD specific fitting, placement, and routing algorithms that are available either from FPLD manufacturers or CAE vendors. However, FPLDs also allow designing on the very low device dependent level providing the best device utilization, if needed.

#### *1.1.8 Cost*

Finally, the above-introduced options reflect on the costs. The major benefit of an MPLD-based design is low cost in large quantities. The actual volume of the products determines which technology is more appropriate to be used. FPLDs have much lower costs of design development and modification, including initial Non-Recurring Engineering (NRE) charges, tooling, and testing costs. However, larger die area and lower circuit density result in higher manufacturing costs per unit. The break-even point depends on the application and volume, and is usually at between ten and twenty thousand units for large capacity FPLDs. This limit is even higher when an integrated volume production approach is applied, using a combination of FPLDs and their corresponding masked-programmed counterparts. Integrated volume production also introduces further flexibility, satisfying short term needs with FPLDs and long term needs at the volume level with masked-programmed devices.

## 1.2 Types of FPLDs

The general architecture of an FPLD is shown in Figure 1.2. A typical FPLD consists of a number of logic cells that are used for implementation of logic functions. Logic cells are arranged in a form of a matrix. Interconnection resources connect logic cell outputs and inputs, as well as input/output blocks used to connect FPLD with the outer world.

Despite the same general structure, concrete implementations of FPLDs differ among the major competitors. There is a difference in approach to circuit programmability, internal logic cell structure, input/output blocks and routing mechanisms.

An FPLD logic cell can be a simple transistor or a complex microprocessor. Typically, it is capable of implementing combinational and sequential logic functions of different complexities.

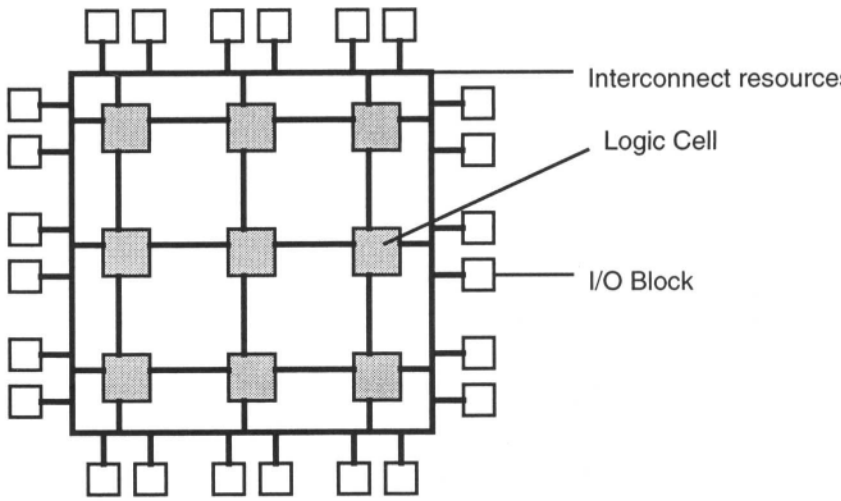



Figure 1.2 FPLD architecture

Current commercial FPLDs employ logic cells that are based on one or more of the following:

- Transistor pairs
- Basic small gates, such as two-input NANDs or XORs
- Multiplexers
- Look-up tables (LUTs)
- Wide-fan-in AND-OR structures

Three major programming technologies, each associated with area and performance costs, are commonly used to implement the programmable switch for FPLDs. These are:

- Static Random Access Memory (SRAM), where the switch is a pass transistor controlled by the state of a SRAM bit
- EPROM, where the switch is a floating-gate transistor that can be turned off by injecting charge onto its floating gate, and
- Antifuse, which, when electrically programmed, forms a low resistance path.

In all cases, a programmable switch occupies a larger area and exhibits much higher parasitic resistance and capacitance than a typical contact used in a custom MPLDs. Additional area is also required for programming circuitry, resulting in higher density and lower speed of FPLDs compared to MPLDs.

An FPLD routing architecture incorporates wire segments of varying lengths which can be interconnected with electrically programmable switches. The density achieved by an FPLD depends on the number of wires incorporated. If the number of wire segments is insufficient, only a small fraction of the logic cells can be utilized. An excessive number of wire segments wastes area. The distribution of wire segments greatly affects both density and performance of an FPLD. For example, if all segments stretch over the entire length of the device (so called long segments), implementing local interconnections costs area and time. On the other hand, employment of only short segments requires long interconnections to be implemented using many switches in series, resulting in unacceptably large delays.

Both density and performance can be optimized by choosing the appropriate granularity and functionality of logic cell, as well as designing the routing architecture to achieve a high degree of routability while minimizing the number of switches. Various combinations of programming technology, logic cell architecture, and routing mechanisms lead to various designs suitable for specific applications. A more detailed presentation of all major components of FPLD architectures is given in the sections and chapters that follow.

If programming technology and device architecture are combined, three major categories of FPLDs are distinguished:

- Complex Programmable Logic Device CPLDs,
- Static RAM Field Programmable Logic Arrays, or simply FPGAs,
- Antifuse FPGAs

In this section we present the major features of these three categories of FPLDs.

### *1.2.1 CPLDs*

A typical CPLD architecture is shown in Figure 1.3. The user creates logic interconnections by programming EPROM or EEPROM transistors to form wide fan-in gates.

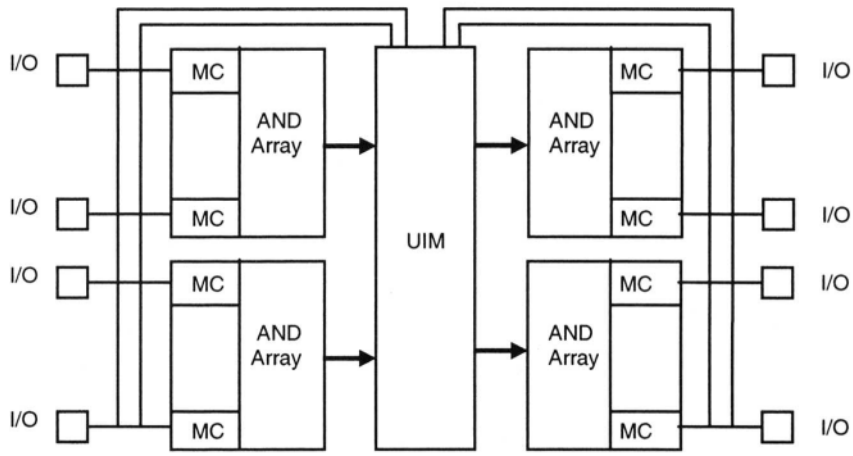



Figure 1.3 Typical CPLD architecture

Function Blocks (FBs) are similar to a simple two-level PLD. Each FB contains a PLD AND-array that feeds its macrocells (MC). The AND-array consists of a number of product terms. The user programs the AND-array by turning on EPROM transistors that allow selected inputs to be included in a product term.

A macrocell includes an OR gate to complete AND-OR logic and may also include registers and an I/O pad. It can also contain additional EPROM cells to control multiplexers that select a registered or non-registered output and decide whether or not the macrocell result is output on the I/O pad at that location. Macrocell outputs are connected as additional FB inputs or as the inputs to a global universal interconnect mechanism (UIM) that reaches all FBs on the chip. FBs, macrocells, and interconnect mechanisms vary from one product to another, giving a range of device capacities and speeds

### *1.2.2 Static RAM FPGAs*

In SRAM FPGAs, static memory cells hold the program that represents the user design. SRAM FPGAs implement logic as lookup tables (LUTs) made from memory cells with function inputs controlling the address lines. Each LUT of $2^n$ memory cells implements any function of n inputs. One or more LUTs, combined with flip-flops, form a logic block (LB). LBs are arranged in a two-dimensional array with interconnect segments in channels as shown in Figure 1.4.

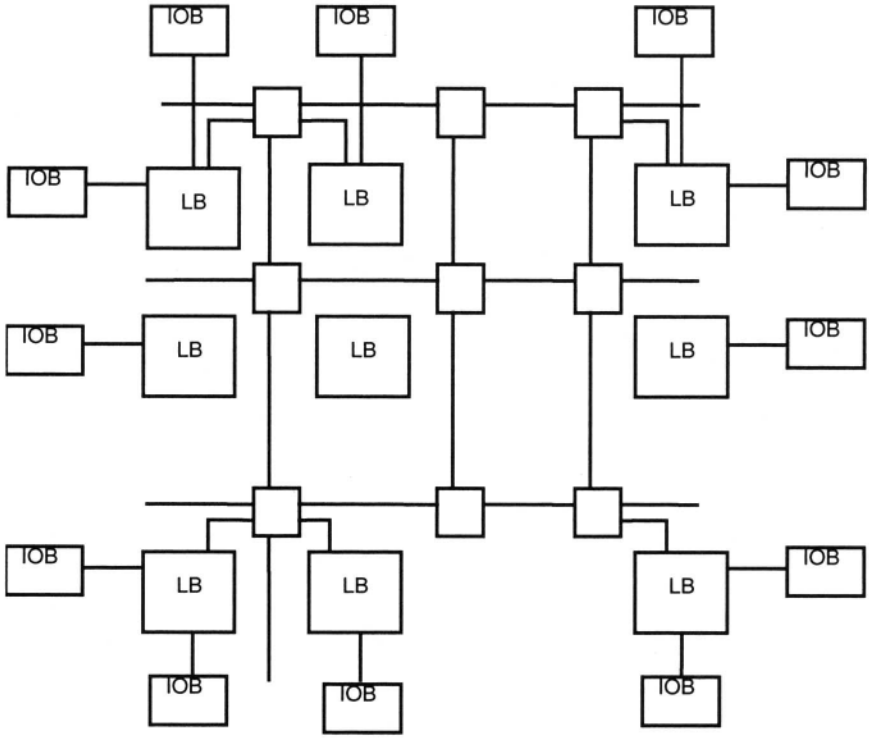



Figure 1.4 Typical SRAM FPGA architecture

Interconnect segments connect to LB pins in the channels and to the other segments in the switch boxes through pass transistors controlled by configuration memory cells. The switch boxes, because of their high complexity, are not full crossbar switches.

An SRAM FPGA program consists of a single long program word. On-chip circuitry loads this word, reading it serially out of an external memory every time

power is applied to the chip. The program bits set the values of all configuration memory cells on the chip, thus setting the lookup table values and selecting which segments connect each to the other. SRAM FPGAs are inherently reprogrammable. They can be easily updated providing designers with new capabilities such as reconfigurability.

### *1.2.3 Antifuse FPGAs*

An antifuse is a two-terminal device that, when exposed to a very high voltage, forms a permanent short circuit (opposite to a fuse) between the nodes on either side. Individual antifuses are small, enabling an antifuse-based architecture to have thousands or millions of antifuses. Antifuse FPGA, as illustrated in Figure 1.5, usually consists of rows of configurable logic elements with interconnect channels between them, much like traditional gate arrays.

The pins on logic blocks (LBs) extend into the channel. An LB is usually a simple gate-level network, which the user programs by connecting its input pins to fixed values or to interconnect nets. There are antifuses at every wire-to-pin intersection point in the channel and at all wire-to-wire intersection points where channels intersect.

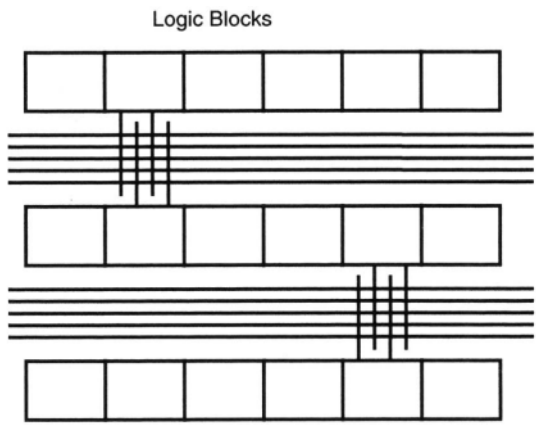



Figure 1.5 Antifuse FPGA architecture

Commercial FPLDs use different programming technologies, different logic cell architectures, and different structures of their routing architectures. A survey of

major commercial architectures is given in the rest of this part, and a more detailed presentation of FPLD families from two major manufacturers, Xilinx, and Altera, is given in Part 2. The majority of design examples introduced in later chapters are illustrated using Altera's FPLDs.

## 1.3 Programming Technologies

An FPLD is programmed using electrically programmable switches. The first user-programmable switch was the fuse used in simple PLDs. For higher density devices, especially the dominant CMOS IC industry, different approaches are used to achieve programmable switches. The properties of these programmable switches, such as size, volatility, process technology, on-resistance, and capacitance determine the major features of an FPLD architecture. In this section we introduce the most commonly used programmable switch technologies in commercial FPLDs.

### *1.3.1 SRAM Programming Technology*

SRAM programming technology uses static RAM cells to configure logic and control intersections and paths for signal routing. The configuration is done by controlling pass gates or multiplexers as it is illustrated in Figure 1.6. When a "1" is stored in the SRAM cell in Figure 1.6(a), the pass gate acts as a closed switch and can be used to make a connection between two wire segments. For the multiplexer, the state of the SRAM cells connected to the select lines controls which one of the multiplexers inputs are connected to the output, as shown in Figure 1.6(b). Reprogrammability allows the circuit manufacturer to test all paths in the FPGA by reprogramming it on the tester. The users get well tested parts and 100% "programming yield" with no design specific test patterns and no "design for testability." Since on-chip programming is done with memory cells, the programming of the part can be done an unlimited number of times. This allows prototyping to proceed iteratively, re-using the same chip for new design iterations. Reprogrammability has advantages in systems as well. In cases where parts of the logic in a system are not needed simultaneously, they can be implemented in the same reprogrammable FPGA and FPGA logic can be switched between applications.

Besides volatility, a major disadvantage of SRAM programming technology is its large area. At least five transistors are needed to implement an SRAM cell, plus at least one transistor to implement a programmable switch. A typical five-transistor memory cell is illustrated in Figure 1.7. There is no separate RAM area on the chip. The memory cells are distributed among the logic elements they control. Since FPGA memories do not change during normal operation, they are built for stability

and density rather than speed. However, SRAM programming technology has two further major advantages; fast-reprogrammability and that it requires only standard integrated circuit process technology.

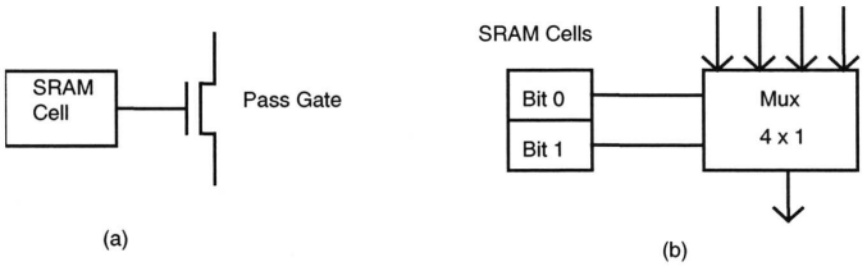



Figure 1.6 SRAM Programming Technology

Since SRAM is volatile, the FPGA must be loaded and configured at the time of chip power-up. This requires external permanent memory to provide the programming bitstream such as PROM, EPROM, EEPROM or magnetic disk. This is the reason that SRAM-programmable FPGAs include logic to sense power-on and to initialize themselves automatically, provided the application can wait the tens of milliseconds required to program the device.

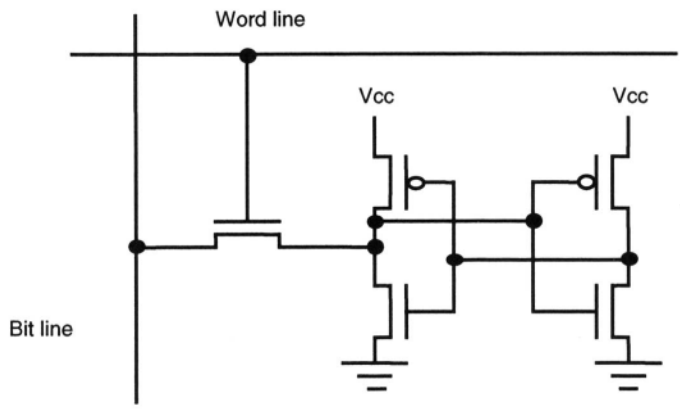



Figure 1.7 Five-transistor Memory Cell

### *1.3.2 Floating Gate Programming Technology*

Floating gate programming technology uses the technology of ultraviolet-erasable EPROM and electrically erasable EEPROM devices. The programmable switch, as shown in Figure 1.8, is a transistor that can be permanently "disabled." To disable the transistor, a charge is injected on the floating polysilicon gate using a high voltage between the control gate and the drain of the transistor. This charge increases the threshold voltage of the transistor so it turns off. The charge is removed by exposing the floating gate to ultraviolet light. This lowers the threshold voltage of the transistor and makes the transistor function normally. Rather than using an EPROM transistor directly as a programmable switch, the unprogrammed transistor is used to pull down a "bit line" when the "word line" is set to high. While this approach can be simply used to provide connection between word and bit lines, it can also be used to implement a wired-AND style of logic, in that way providing both logic and routing.

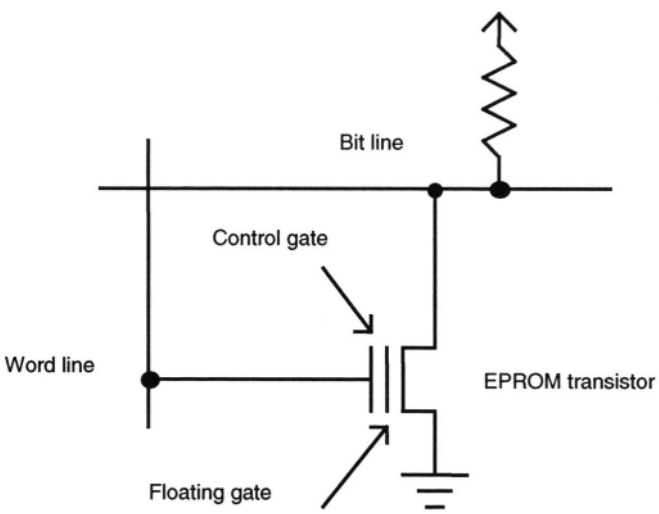



Figure 1.8 Floating gate programming technology

The major advantage of the EPROM programming technology is its reprogrammability. An advantage over SRAM is that no external permanent memory is needed to program a chip on power-on. On the other hand, reconfiguration itself can not be done as fast as in SRAM technology devices. Additional disadvantages are that EPROM technology requires three more processing steps over an ordinary CMOS process, the high on-resistance of an EPROM transistor, and the high static power consumption due to the pull-up resistor used.

EEPROM technology used in some devices is similar to the EPROM approach, except that removal of the gate charge can be done electrically, in-circuit, without ultraviolet light. This gives an advantage of easy reprogrammability, but requires more space due to the fact that EEPROM cell is roughly twice the size of an EPROM cell.

### *1.3.3 Antifuse Programming Technology*

An antifuse is an electrically programmable two-terminal device. It irreversibly changes from high resistance to low resistance when a programming voltage (in excess of normal signal levels) is applied across its terminals. Antifuses offer several unique features for FPGAs , most notably a relatively low on-resistance of 100-600 Ohms and a small size. The layout area of an antifuse cell is generally smaller than the pitch of the metal lines it connects; it is about the same size as a via connecting metal lines in an MPLD. When high voltage (11 to 20 Volts) is applied across its terminals, the antifuse will "blow" and create a low resistance link. This link is permanent. Antifuses are built either using an Oxygen-Nitrogen-Oxygen (ONO) dielectric between an N+ diffusion and polysilicon, or amorphous silicon between metal layers or between polysilicon and the first layer of metal.

Programming an antifuse requires extra circuitry to deliver the high programming voltage and a relatively high current of 5 mA or more. This is done through large transistors to provide addressing to each antifuse.

Antifuses are normally "off" devices. Only a small fraction of the total that need to be turned on must be programmed (about 2% for a typical application). So, other things being equal, programming is faster with antifuses than with "normally on" devices.

Antifuse reliability must be considered for both the unprogrammed and programmed states. Time dependent dielectric breakdown (TDDB) reliability over 40 years is an important consideration. It is equally important that the resistance of a programmed antifuse remains low during the life of the part. Analysis of ONO dielectrics shows that they do not increase the resistance with time. Additionally, the parasitic capacitance of an unprogrammed amorphous antifuse is significantly lower than for other programming technologies.

### *1.3.4 Summary of Programming Technologies*

Major properties of each of above presented programming technologies are shown in Table 1.1. All data assumes a 1.2 μm CMOS process technology and is used only for comparison purposes. The most recent devices use much higher density devices and many of them are implemented in 0.5 or even 0.22 μm CMOS process technology with the tendency to reduce it even further (0.18 μm and 0.15 μm).

Table 1.1 Comparison of Programming technologies

| Technology and Process | Volatile | Reprogram mability ? | Area | R (ohm) (on switch) | C (fF) (parasitic) | # Extra fabric. steps |
|---|---|---|---|---|---|---|
| SRAM Mux Pass Trans. 1.2 □m CMOS | Yes | Yes In-circuit | Large | 0.5-2K | 10-20 | 0 |
| ONO Antifuse 1.2 □m CMOS | No | No | Fuse small Prog. trans. large | 300-600 | 5 | 3 |
| Amorphous Antifuse 1.2 □m CMOS | No | No | Fuse small Prog. trans. large | 50-100 | 1.1-1.3 | 3 |
| EPROM 1.2 □m CMOS | No | Yes Out of circuit | Small in array | 2-4K | 10-20 | 3 |
| EEPROM 1.2 □m CMOS | No | Yes In- circuit | 2 x EPRO M | 2-4K | 10-20 | >5 |

## 1.4. Logic Cell Architecture

In this section we present a survey of commercial FPLD logic cell architectures in use today, including their combinational and sequential portions. FPLD logic cells differ both in size and implementation capability. A two transistor logic cell can only implement a small size inverter, while the look-up table logic cells can implement any logic function of several input variables and is significantly larger. To capture these differences we usually classify logic blocks by their granularity.

Since granularity can be defined in various ways (as the number of Boolean functions that the logic block can implement, the number of two-input AND gates, total number of transistors, etc.), we choose to classify commercial blocks into just two categories: fine-grain and coarse-grain.

Fine-grain logic cells resemble MPLD basic cells. The most fine grain logic cell would be identical to a basic cell of an MPLD and would consist of few transistors that can be programmably interconnected.

The FPGA from Crosspoint Solutions uses a single transistor pair in the logic cell. In addition to the transistor pair tiles, as depicted in Figure 1.9, the cross-point FPGA has a second type of logic cell, called a RAM logic tile, that is tuned for the implementation of random access memory, but can also be used to build other logic functions.

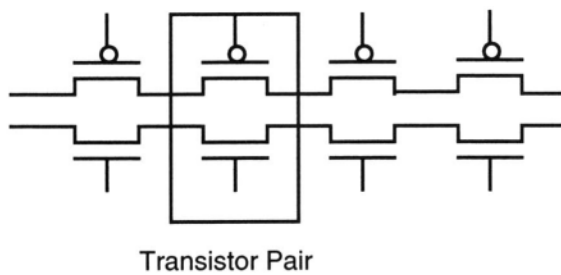



Figure 1.9 Transistor pair tiles in cross-point FPGA

A second example of a fine-grain FPGA architecture is the FPGA from Plessey. Here the basic cell is a two-input NAND gate as illustrated in Figure 1.10. Logic is formed in the usual way by connecting the NAND gates to implement the desired function. If the latch is not needed, then the configuration memory is set to make the latch permanently transparent.

Several other commercial FPGAs employ fine-grain logic cells. The main advantage of using fine-grain logic cells is that the usable cells are fully utilized. This is because it is easier to use small logic gates efficiently and the logic synthesis techniques for such cells are very similar to those for conventional MPGAs (Mask-Programmable Gate Arrays) and standard cells.

The main disadvantage of fine-grain cells is that they require a relatively large number of wire segments and programmable switches. Such routing resources are costly in both delay and area. If a function that could be packed into a few complex

cells must instead be distributed among many simple cells, more connections must be made through the programmable routing network. As a result, FPLDs with fine-grain logic cells are in general slower and achieve lower densities than those using coarse-grain logic cells.

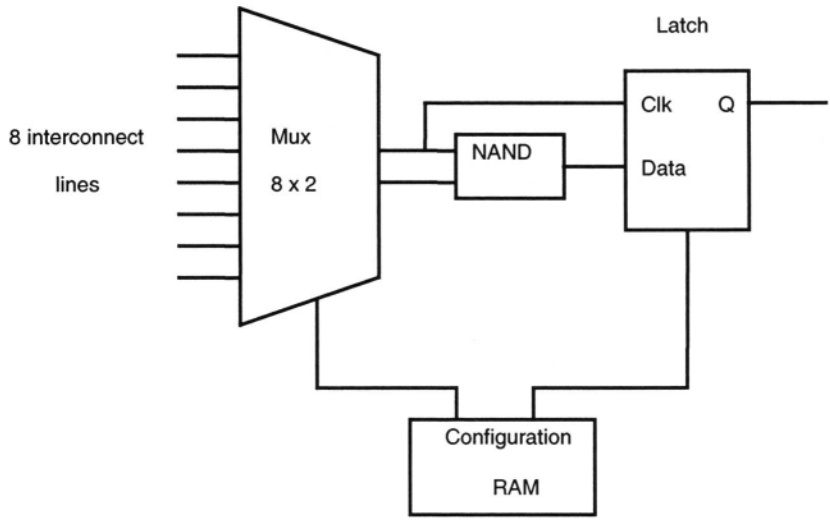



Figure 1.10 The Plessey Logic Cell

As a rule of thumb, an FPLD should be as fine-grained as possible while maintaining good routability and routing delay for the given switch technology. The cell should be chosen to implement a wide variety of functions efficiently, yet have minimum layout area and delay.

Actel's logic cells have been designed on the base of usage analysis of various logic functions in actual gate array applications. The Act-1 family uses one general-purpose logic cell as shown in Figure 1.11. The cell is composed of three 2-to-l multiplexers, one OR gate, 8 inputs, and one output. Various macrofunctions (AND, NOR, flip-flops, etc.) can be implemented by applying each input signal to the appropriate cell inputs and tying other cell inputs to 0 or 1. The cell can implement all combinational functions of two inputs, all functions of three inputs with at least one positive input, many functions of four inputs, and some ranging up to eight inputs. Any sequential macro can be implemented from one or more cells using appropriate feedback routings.

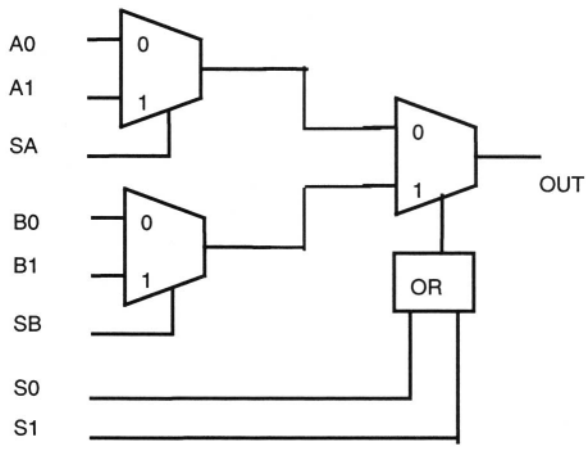



Figure 1.11 Act-1 logic cell

Further analysis of macros indicate that a significant proportion of the nets driving the data input of flip-flop have no other fan-out. This motivated the use of a mixture of two specialized cells for Act-2 and Act-3 families. The "C" cell and its equivalent shown in Figure 1.12 are modified versions of the Act-1 cell re-optimized to better accommodate high fan-in combinational macros. It actually represents a 4-to-l multiplexer and two gates, implementing a total of 766 distinct combinational functions.

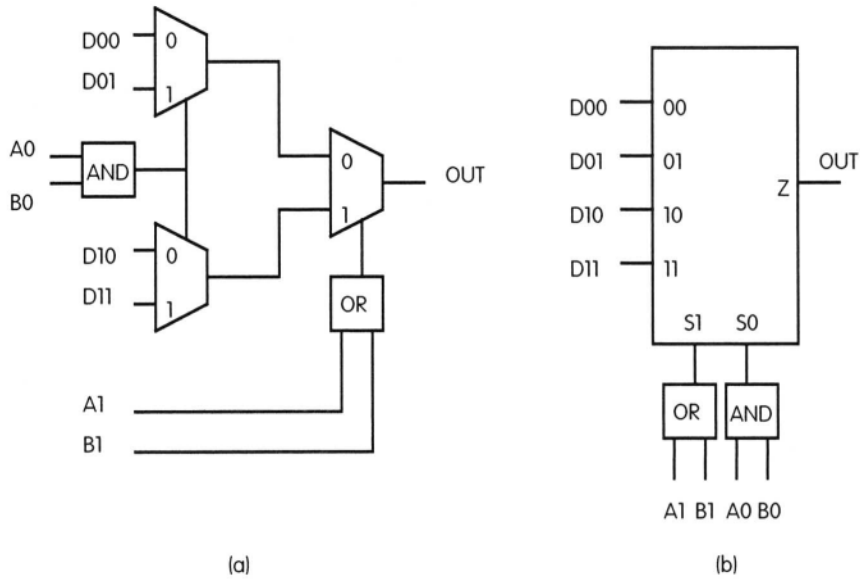



Figure 1.12 Act-2 "C" cell

The "S" cell, shown in Figure 1.13, consists of a front end equivalent to "C" cell followed by sequential block built around two latches. The sequential block can be used as a rising- or falling-edge D flip-flop or a transparent-high or transparent-low latch, by tying the C1 and C2 inputs to a clock signal, logical zero or logical one in various combinations. For example, tying Cl to 0 and clocking C2 implements a rising-edge D flip-flop. Toggle or enabled flip-flops can be built using combinational front end in addition to the D flip-flop. JK or SR flip-flops can be configured from one or more "C" or "S" cells using external feedback connections. A chip with an equal mixture of "C" and "S" cells provides sufficient flip-flops for most designs plus extra flexibility in placement. Over a range of designs, the Act-2 mixture provides about 40-100% greater logic capacity per cell than the Act-1 cell.

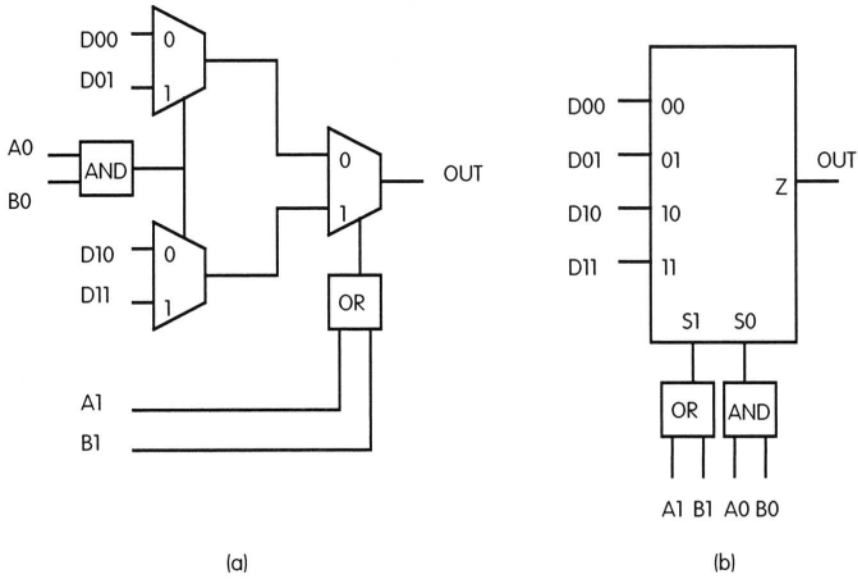



Figure 1.13 Actel-2 "S" cell

The logic cell in the FPLD from QuickLogic is similar to the Actel logic cell in that it uses a 4-to-1 multiplexer. Each input to the multiplexer is fed by an AND gate, as shown in Figure 1.14. Alternating inputs to the AND gates are inverted allowing input signals to be passed in true or complement form, therefore eliminating the need to use extra logic cells to perform simple inversions.

Multiplexer-based logic cells provide a large degree of functionality for a relatively small number of transistors. However, this is achieved at the expense of a large number of inputs placing high demands on the routing resources. They are best suited to FPLDs that use small size programmable switches such as antifuses.

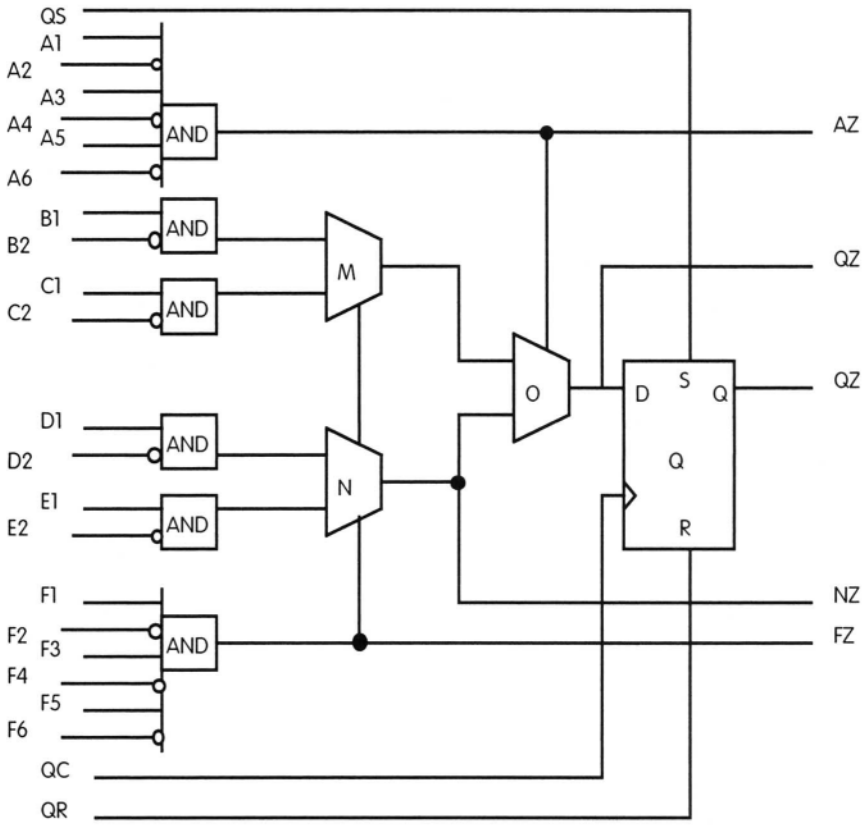



Figure 1.14 The QuickLogic logic cell

Xilinx logic cells are based on the use of SRAM as a look-up table. The truth table for a K-input logic function is stored in a $2^K$ x 1 SRAM as it is illustrated in Figure 1.15.

The address lines of the SRAM function as inputs and the output (data) line of the SRAM provides the value of the logic function. The major advantage of K-input look-up table is that it can implement any function of K inputs. The disadvantage is that it becomes unacceptably large for more than five inputs since the number of memory cells needed for a K-input look-up table is $2^K$. Since many of the logic functions are not commonly used, a large look-up table will be largely underutilized.

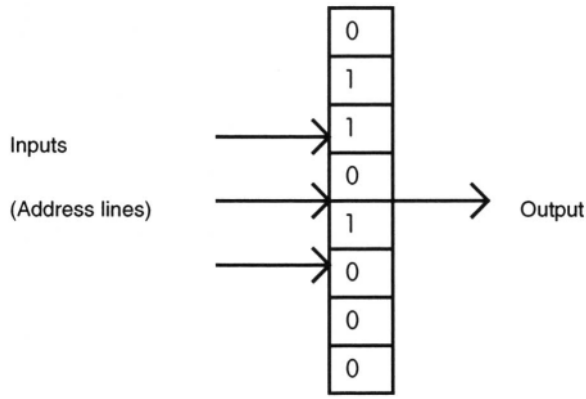



Figure 1.15 Look-up table

The Xilinx 3000 series logic cell contains a five input one output look-up table. This block can be configured as two four-input LUTs if the total number of distinct inputs is not greater than five. The logic cell also contains sequential logic (two D flip-flops) and several multiplexers that connect combinational inputs and outputs to the flip-flops or outputs.

The Xilinx 4000 series logic cell contains two four input look-up tables feeding into a three input LUT. All of the inputs are distinct and available external to the logic cell. The other difference from the 3000 series cell is the use of two nonprogrammable connections from the two four input LUTs to the three input LUT. These connections are much faster since no programmable switches are used in series.

A detailed explanation of Xilinx 3000 and 4000 series logic cells is given in Chapter 2, since they represent two of the most popular and widely used FPGAs.

Other popular families of FPLDs with the coarse-grain logic cells are Altera's EPLDs and CPLDs. The architecture of Altera 5000 and 7000 series EPLDs has evolved from a PLA-based architecture with logic cells consisting of wide fan-in (20 to over 100 inputs) AND gates feeding into an OR gate with three to eight inputs. They employ a floating gate transistor based programmable switch that enables an input wire to be connected to an input to the gate as shown in Figure 1.16. The three product terms are then OR-ed together and can be programmable inverted by an XOR gate, which can also be used to produce other arithmetic functions. Each signal is provided in both truth and complement form with two

separate wires. The programmable inversion significantly increases the functional capability of the block.

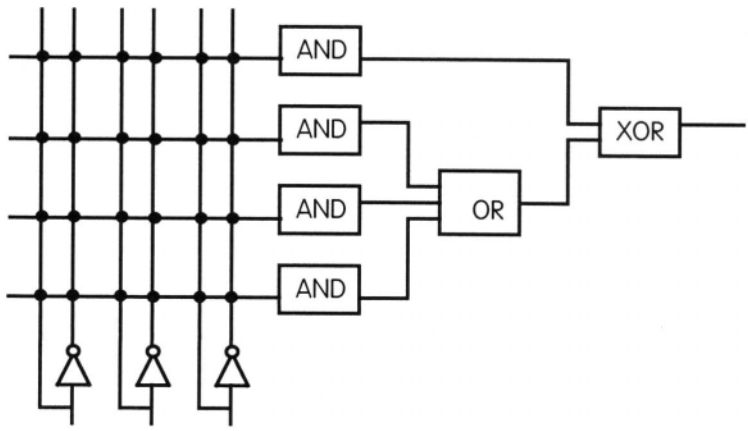



Figure 1.16 The Altera 5000 series logic block

The advantage of this type of block is that the wide AND gate can be used to form logic functions with few levels of logic cells reducing the need for programmable interconnect resources. However, it is difficult to make efficient use of the inputs to all of the gates. This loss is compensated by the high packing density of the wired AND gates. Some shortcomings of the 5000 series devices are overcome in the 7000 series, most notably it provides two more product terms and has more flexibility because neighboring blocks can "borrow" product terms from each other.

The Altera Flex 8000 and 10K series CPLDs are the SRAM based devices providing low stand-by power and in-circuit reconfigurability. A logic cell contains 4-input LUT that provides combinational logic capability and a programmable register that offers sequential logic capability. High system performance is provided by a fast, continuous network of routing resources. The detailed description of both major Altera's series of CPLDs is given in Chapter 2.

Most of the logic cells described above include some form of sequential logic. The Xilinx devices have two D flip-flops, while the Altera devices have one D flip-flop per logic cell. Some devices such as Act-1 do not explicitly include sequential logic, forming it using programmable routing and combinational logic cells.

## 1.5 Routing Architecture

The routing architecture of an FPLD determines a way in which the programmable switches and wiring segments are positioned to allow the programmable interconnection of logic cells. A routing architecture for an FPLD must meet two criteria: routability and speed. Routability refers to the capability of an FPLD to accommodate all the nets of a typical application, despite the fact that the wiring segments must be defined at the time the blank FPLD is made. Only switches connecting wiring segments can be programmed (customized) for a specific application, not the numbers, lengths or locations of the wiring segments themselves. The goal is to provide a sufficient number of wiring segments while not wasting chip area. It is also important that the routing of an application can be determined by an automated algorithm with minimal intervention.

Propagation delay through the routing is a major factor in FPLD performance. After routing an FPLD, the exact segments and switches used to establish the net are known and the delay from the driving output to each input can be computed. Any programmable switch (EPROM, pass-transistor, or antifuse) has a significant resistance and capacitance. Each time a signal passes through a programmable switch, another RC stage is added to the propagation delay. For a fixed R and C, the propagation delay mounts quadratically with the number of series RC stages. The use of a low resistance switch, such as antifuse, keeps the delay low and its distribution tight. Of equal significance is optimization of the routing architecture. Routing architectures of some commercial FPLD families are presented in this section.

In order to present commercial routing architectures, we will use the routing architecture model shown in Figure 1.17. First, a few definitions are introduced in order to form a unified viewpoint when considering routing architectures.

A wire segment is a wire unbroken by programmable switches. One or more switches may attach to a wire segment. Typically, one switch is attached to the each end of a wire segment. A track is a sequence of one or more wire segments in a line. A routing channel is a group of parallel tracks.

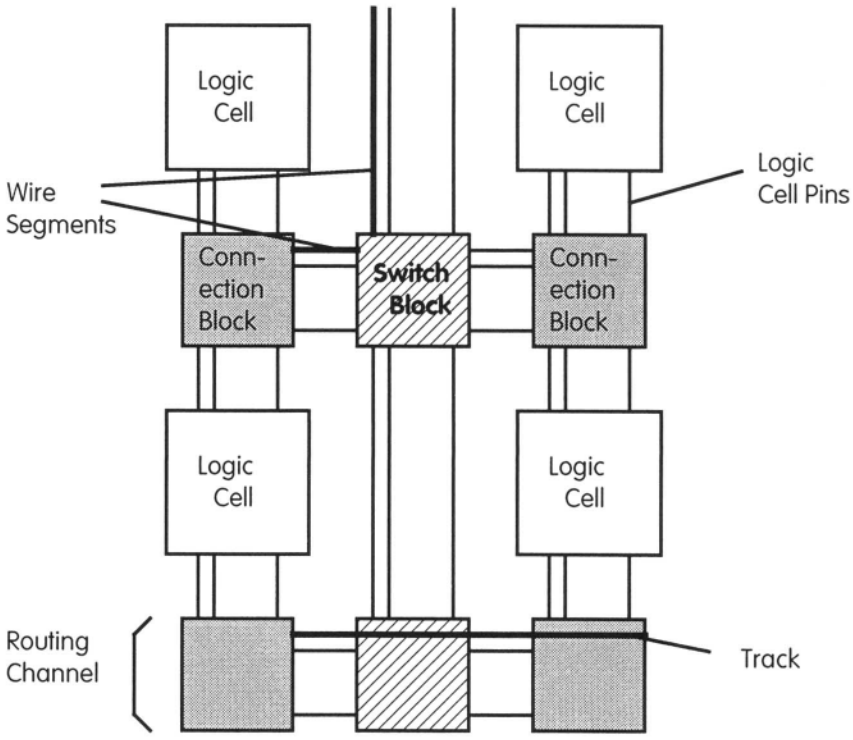



Figure 1.17 General FPLD routing architecture model

As shown in Figure 1.17, the model contains two basic structures. The first is a connection block which appears in all architectures. The connection block provides connectivity from the inputs and outputs of a logic block to the wire segments in the channels and can be both vertical or horizontal. The second structure is the switch block which provides connectivity between the horizontal as well as vertical wire segments. The switch block in Figure 1.17 provides connectivity among wire segments on all four of its sides.

Trade-offs in routing architectures are illustrated in Figure 1.18. Figure 1.18(a) represents a set of nets routed in a conventional channel. Freedom to configure the wiring of an MPLD allows us to customize the lengths of horizontal wires.

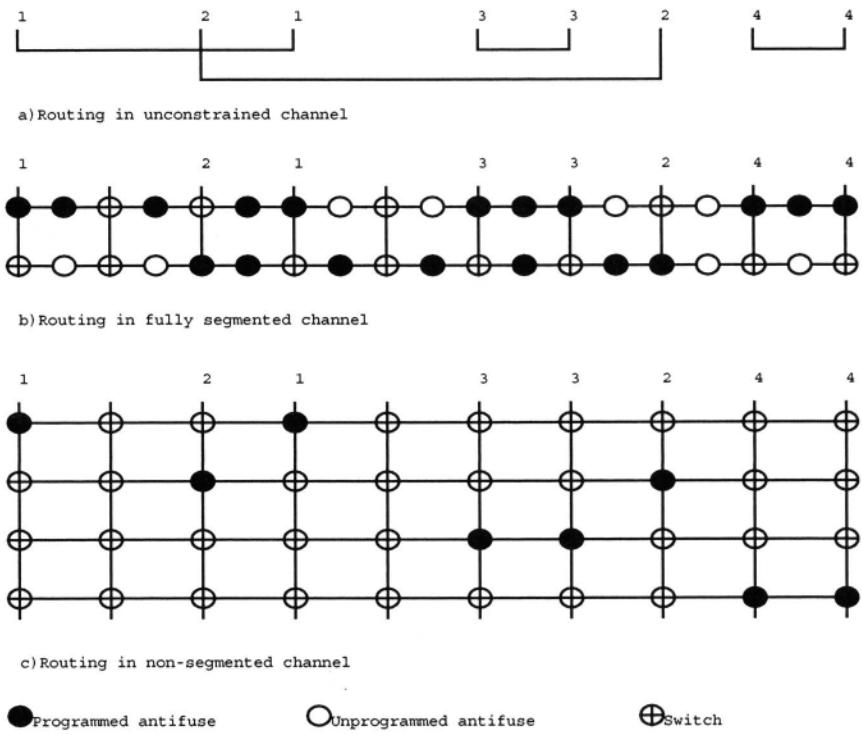



Figure 1.18 Types of routing architecture

In order to have complete freedom of routing, a switch is required at every cross point. More switches are required between two cross points along a track to allow the track to be subdivided into segments of arbitrary length, as shown in Figure 1.18(b). In FPLDs, each signal enters or leaves the channel on its own vertical segment.

An alternative is to provide continuous tracks in sufficient number to accommodate all nets, as shown in Figure 1.18(c). This approach is used in many types of programmable logic arrays and in the interconnect portion of certain programmable devices. Advantages are that two RC stages are encountered and that the delay of each net is identical and predictable. However, full length tracks are used for all, even short nets. Furthermore, the area is excessive, growing quadratically with the number of nets. This is the reason to employ some intermediate approaches, usually based on segmentation of tracks into varying

(appropriate) sizes. A well-designed segmented channel does not require many more tracks than would be needed in a conventional channel. Although surprising, this finding has been supported both experimentally and analytically.

In the Xilinx 3000 series FPGAs, the routing architecture connections are made from the logic cell to the channel through a connection block. Since the connection site is large, because of the SRAM programming technology, the Xilinx connection block typically connects each pin to only two or three out of five tracks passing a cell. Connection blocks connect on all four sides of the cell. The connections are implemented by pass transistors for the output pins and multiplexers for input pins. The use of multiplexers reduces the number of SRAM cells needed per pin.

The switch block makes a connection between segments in intersecting horizontal and vertical channels. Each wire segment can connect to a subset of the wire segments on opposing sides of the switch block (typically to 5 or 6 out of 15 possible wire segments). This number is limited by the large size and capacitance of the SRAM programmable switches.

There are four types of wire segments provided in the Xilinx 3000 architecture and five types in the Xilinx 4000 architecture. The additional wire segment consists of so called double-length lines that essentially represent the wire segments of the double length that are connected to every second switch block. In the Xilinx 4000 devices the connectivity between the logic cell pins and tracks is much higher because each logic pin connects to almost all of the tracks. The detailed presentation of the Xilinx routing architectures is given in Chapter 2.

The routing architecture of the Altera 5000 and 7000 series EPLDs uses a two-level hierarchy. At the first level hierarchy, 16 or 32 of the logic cells are grouped into a Logic Array Block (LAB) providing a structure very similar to the traditional PLD. There are four types of tracks passing each LAB. In the connection block every such track can connect into every logic cell pin making routing very simple. Using fewer connection points results in better density and performance, but yields more complex routing. The internal LAB routing structure could be considered as segmented channel, where the segments are as long as possible. Since connections also perform wire ANDing, the transistors have two purposes.

Connections among different LABs are made using a global interconnect structure called a Programmable Interconnect Array (PIA). It connects outputs from each LAB to inputs of other LABs, and acts as one large switch block. There is full connectivity among the logic cell outputs and LAB inputs within a PIA. The advantage of this scheme is that it makes routing easy, but requires many switches adding more to the capacitive load than necessary. Another advantage is the delay through the PIA is the same regardless of which track is used. This further helps

predict system performance. However, the circuits can be much slower than with segmented tracks.

A similar approach is found in the Altera 8000 series CPLDs. Connections among LABs are implemented using FastTrack Interconnect continuous channels that run the length of the device. A detailed presentation of both of Altera's interconnect and routing mechanisms is given in Chapter 2.

## 1.6 Design Process

The complexity of FPLDs has surpassed the point where manual design is desirable or feasible. The utility of an FPLD architecture becomes more and more dependent on automated logic and layout synthesis tools.

The design process with FPLDs is similar to other programmable logic design. Input can come from a schematic netlist, a hardware description language, or a logic synthesis system. After defining what has to be designed, the next step is design implementation. It consists of fitting the logic into the FPLD structures. This step is called "logic partitioning" by some FPGA manufacturers and "logic fitting" in reference to CPLDs.

After partitioning, the design software assigns the logic, now described in terms of functional units on the FPLD, to a particular physical locations on the device and chooses the routing paths. This is similar to placement and routing traditional gate arrays.

One of the main advantages of FPLDs is their short development cycle compared to full- or semi-custom integrated circuits. Circuit design consists of three main tasks:

- design definition
- design implementation
- design modification

From the designer's point of view, the following are important features of design tools:

- enable that the design process evolves towards behavioral level specification and synthesis

- provide design freedom from details of mapping to specific chip architecture
- provide an easy way to change or correct design

A variety of design tools are used to perform all or some of the above tasks. Chapter 3 is devoted to the high level design tools with an emphasis on those that enable behavioral level specification and synthesis, primarily high-level hardware description languages. Examples of designs using two of such languages, the Altera Hardware description Language (AHDL) and VHSIC Hardware description Language (VHDL), are given together with the introduction to these specification tools.

An application targeted to an FPLD can be designed on any one of several logic or ASIC design systems, including schematic capture and hardware description languages. To target an FPLD, the design is passed to FPLD specific implementation software. The interface between design entry and design implementation is a netlist that contains the desired nets, gates, and references to specific vendor provided macros. Manual and automatic tools can be used interchangeably or an implementation can be done fully automatically.

A combination of moderate density, reprogrammability and powerful prototyping tools to a hardware designer resembles a software-like iterative-implementation methodology. Figure 1.19 is presented to compare a typical ASIC and typical FPLD design cycle.

In a typical ASIC design cycle, the design is verified by simulation at each stage of refinement. Accurate simulators are slow. ASIC designers use the whole range of simulators in the speed/accuracy spectrum in an attempt to verify their design. Although simulation can be used in designing for FPLDs, simulation can be replaced with in-circuit verification by simulating the circuitry in real time with a prototype. The path from design to prototype is short allowing verification of the operation over a wide range of conditions at high speed and high accuracy.

A fast design-place-route-load loop is similar to the software edit-compile-run loop and provides similar benefits, a design can be verified by the trial and error method. A designer can also verify that a design works in a real system, not merely in a potentially erroneous simulation.

Design by prototype does not verify proper operation with worst case timing, but rather that a design works on the typical prototype part. To verify worst case timing, designers can check speed margins in actual voltage and temperature comers with a scope and logic analyzer, speeding up marginal signals. They also may use a

software timing analyzer or simulator after debugging to verify worst case paths or simply use faster speed grade parts in production to ensure sufficient speed margins over the complete temperature and voltage range.

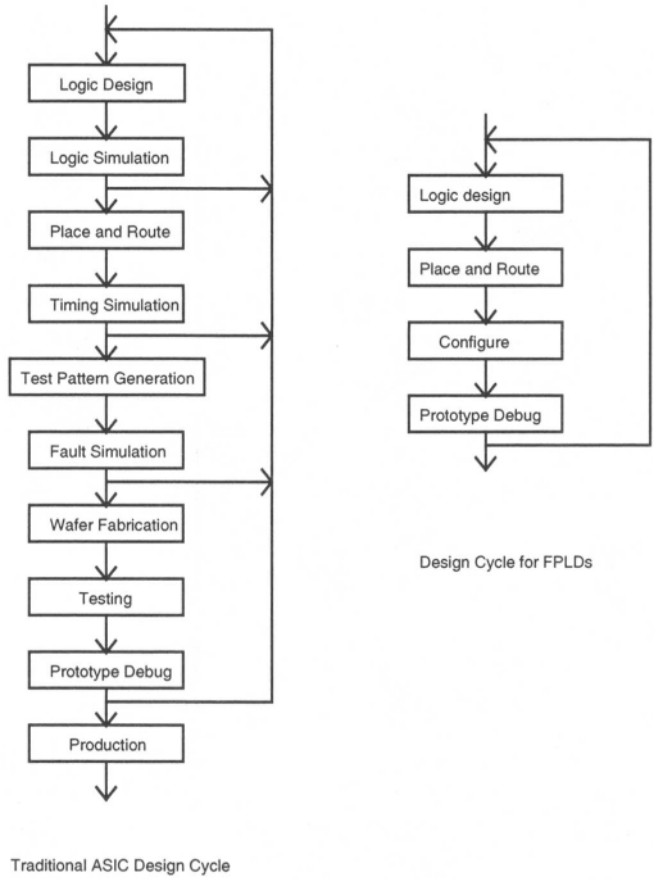



Figure 1.19 Comparing Design Cycles

As with software development, a reprogrammable FPLD removes the dividing line between prototyping and production. A working prototype may qualify as a production part if it meets performance and cost goals. Rather than redesign, a designer may choose to substitute a faster FPLD and use the same programming bitstream, or choose a smaller, cheaper FPLD (with more manual work to squeeze the design into a smaller device). A third choice is to substitute a mask-programmed version of the logic array for the field-programmable array. All three options are

much simpler than a system redesign, which must be done for traditional MPLDs or ASICs.

The design process usually begins with the capture of the design. Most users enter their designs as schematics built of macros from a library. An alternative is to enter designs in terms of Boolean equations, state machine descriptions, or functional specifications. Different portions of a design can be described in different ways, compiled separately, and merged at some higher hierarchical level in the schematic.

Several guidelines are suggested for reliable design with FPLDs, mostly the same as those for users of MPLDs. The major goal is to make the circuit function properly independent of shifts in timing from one part to the next. Guidelines will be discussed in Chapter 3.

Rapid system prototyping is most effective when it becomes rapid product development. Reprogrammability allows a system designer another option, to modify the design in an FPLD by changing the programming bitstream after the design is in the hands of the customer. The bitstream can be stored in a dedicated (E)PROM or elsewhere in the system. In some existing systems, manufacturers send modified versions of hardware on a floppy disk or as a file sent over modem.

## 1.7 FPLD Applications

FPLDs have been used in a large number of applications, ranging from the simple ones replacing glue logic to those implementing new computing paradigms, that are not possible using other technologies. In this section we will list some of them, as to make a classification into some typical groups, and emphasize most important features of each group.

CPLDs are used in applications that can efficiently use wide fan-in of AND/OR gates and do not need a large number of flip-flops. Examples of such circuits are various kinds of finite state machines. On the other hand, FPGAs with a large number of flip-flops are better suited for the applications that need memory functions and complex data paths. Also, due to their easy reprogrammability they become an important element of prototyping digital systems designs. As such they enable emulation of entire complex systems, and in many cases also their final implementation. Finally, all FPGAs as the static RAM based circuits allow at least a minimum level of dynamic reconfigurability. While all of them allow full device reconfiguration by downloading another bitstream (configuration file), some of them also allow partial reconfiguration. The partial reconfiguration provides change of the function of a part of the device, while the remaining part operates without disruption of the system function.

In order to visualize the range of current and potential applications, we have to mention typical features of FPLDs in terms of their capacity and speed. Today the leading suppliers of FPLDs offer devices containing up to 500,000 equivalent (two-input NAND) gates, with a perspective to quadruple this figure in the next two to three years. These devices are delivered in a number of configurations so that application designers have the choice to fit their designs into a device with minimal capacity. They also come in a range of speed grades and different packages with different number of input/output pins. The number of pins sometimes exceeds 600. The speed of circuits implemented in FPLDs varies depending primarily on application and design approach. As an illustration, all major manufacturers offer devices that provide full compliance with 64-bit 66MHz PCI-bus requirements.

### *1.7.1 Glue Random Logic Replacement*

Initial applications of FPLDs were influenced by earlier use of simple PLDs. Having larger capacity than simple PLDs, FPLDs have been used to replace random glue logic. In an application of this type FPLDs obviously provide lower chip count, more compact and reliable designs and higher speeds as the entire circuit can usually be implemented within a single device. The overall system design can be placed on a smaller PCB. With good planning of external pin assignment, the design fitting into an FPLD can be later modified without re-designing the entire PCB. In particular, this applies when using FPLDs that are in-circuit programmable. In this case there is no need for removal and insertion of FPLDs on a PCB, as the circuit can be programmed while being on a PCB. In the case of in-system programmable FPLDs, it is possible to reconfigure hardware and partially change a function implemented in an FPLD, without powering it down.

A typical example of replacing glue logic is in interfacing standard microprocessor or microcontroller based systems. Glue logic is used to provide interfaces to subsystems like external memories and specific peripherals. It usually requires a timing and logic adjustment that is achieved using different combinational and sequential circuits (decoders, multiplexers, registers, and finite state machines). An example is given in Chapter 8, where interfacing requirements for connecting memory and head-on display to VuMan wearable computer, which is based on a standard 386SX processor, are implemented in a single FPLD. A number of other small examples of circuits that are easily customized and replace a number of standard SSI and MSI circuits are given throughout the book. Even simpler standard VLSI circuits can often fit into an FPLD, as illustrated in Figure 1.20.

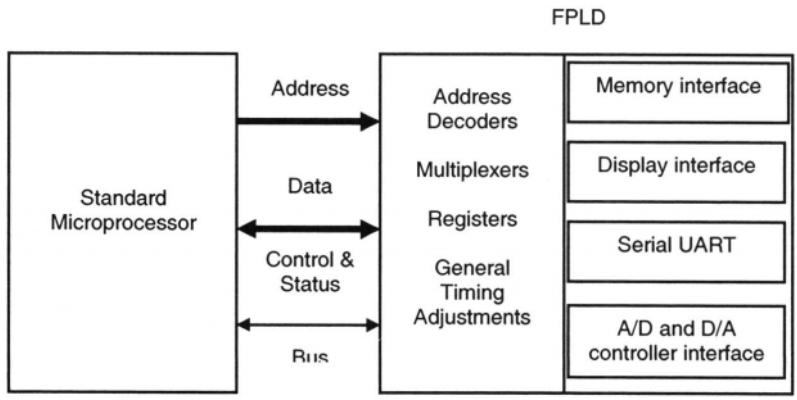



Figure 1.20 Glue logic replacement

### *1.7.2 Hardware Accelerators*

For many applications FPLDs have a performance which is superior to a more traditional microprocessor or digital signal processor. This is especially true for tasks that can be parallelized and executed several orders of magnitude faster on an FPLD than on a microprocessor. The idea of using hardware accelerators that supply a single, dedicated service to the microprocessor is employed in such applications as graphic acceleration, sound, and video processing. However, if the algorithms require processing of data with non-standard formats and repetitive execution of relatively simple operations, the FPLDs represent an obvious choice. Furthermore, having a system with reconfigurable hardware, several advantages can be achieved, such as:

- reduced number of components and space as the FPLD can implement different functions/systems at different times
- new versions of design are implemented by simple downloading configuration bitstream
- new functions can be added as required

The acceleration functions implemented in the FPLD can be in a form of functional unit, co-processor, attached processing unit or stand-alone processing unit, connected by an input/output interface to the main microprocessor-based

system. The further away is the function from the microprocessor, the slower is the exchange of data between the microprocessor and the function. An example of image enhancement co-processor hardware accelerator is shown in Figure 1.21.

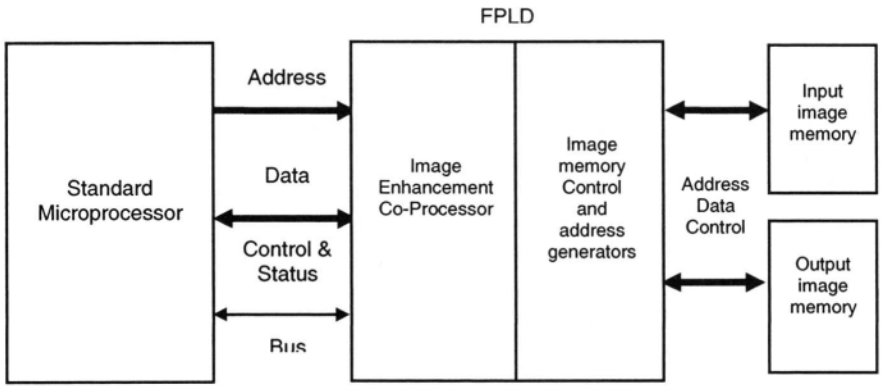



Figure 1.21 Using FPLD to implement a hardware accelerator

### *1.7.3 Non-standard Data Path/Control Unit Oriented Systems*

Oftentimes complex computational systems and algorithms can be described in a form of dataflow oriented description and implemented as data path controlled by its own control unit. Such non-standard dedicated systems, especially in the case of low volumes, are the best candidates for implementation in FPLDs. Typical applications include digital signal and image processing, neural networks and other complex, computationally demanding algorithms. By using high-level integrated design tools, capable to easily capture such an application (hardware description languages), the design process of dedicated hardware systems in its complexity becomes comparable to software-only solutions. An illustration of non-standard data path/control-unit system is given in Figure 1.22.

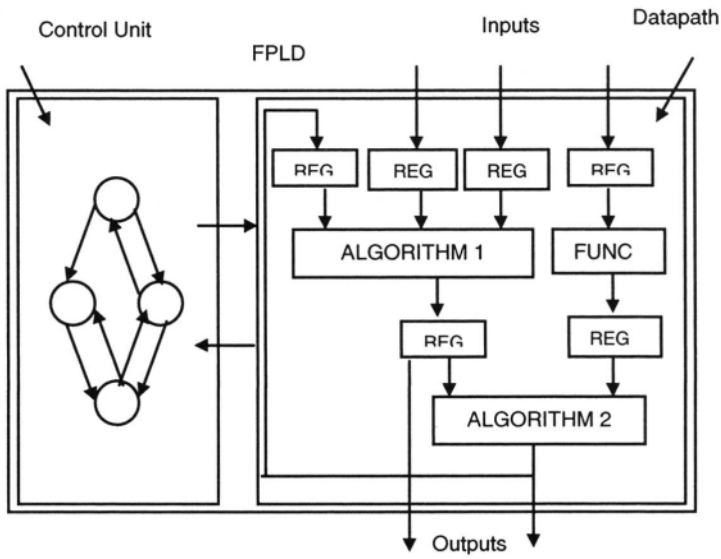



Figure 1.22 Complex non-standard datapath/control unit systems

### *1.7.4 Virtual Hardware*

The reconfigurable hardware can be viewed in two completely new ways. First, as a hardware resource that can perform different tasks on demand, executing them one at the time. The user perceives a hardware resource to be "larger" than it actually is. An illustration of such a system is given in Figure 1.23. Different hardware configurations are stored in a configuration memory and executed one at the time, as the overall application requires. Another view is to consider it as a hardware cache where the most recently used hardware elements are stored and accessed by an application. The implementation of a hardware cache, or virtual hardware system, requires some form of management to control the process and ensure it runs effectively. There are several options available for this management task, including standalone hardware (can be another FPLD), custom software routines, or integrated operating system support.

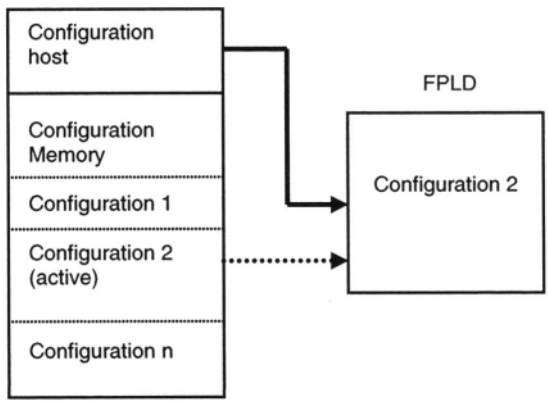



Figure 1.23 FPLD used to implement virtual hardware

### *1.7.5 Custom-Computing Machines*

In recent years, several computing systems have been developed implementing a custom processor within an FPLD. In this type of system, the goal is not to compete with performances of dedicated processors, but rather to provide a platform with an optimal partitioning of functions between hardware and software components. This approach allows the attributes of the custom processor, such as the architecture of its core and instruction set, to be modified as the application requires. The FPLD can implement not only processor core but also hardware acceleration units in the form of functional units as illustrated in Figure 1.24. This type of system can utilize the flexibility of software and the speed of hardware in single device to achieve optimal performance. The advantage of this approach is that hardware functional units are located in the closest position to the processor core, and therefore the communication interface can be very fast. However, the entire system is compile (synthesis) time configurable, and in order to be run-time reconfigurable it requires truly dynamically reconfigurable FPLDs. As such, custom-computing machines represent an ideal platform for achieving the goal of hardware/software co-design and to partition a task into software and hardware components to satisfy design criteria. The design criteria may not necessarily be to develop a system with highest speed performance. Taking into account the cost and other constraints, a trade-off between a fully hardware and fully software solution is required. Almost all digital systems designers have been aware of that fact, especially those designing embedded systems. Traditionally, hardware and software parts have been designed

independently, however, with little effort to pursue a concurrent design. The goal of hardware/software co-design is to design both hardware and software parts from a single specification and make the partitioning decisions based on design criteria. In Chapter 7 we present a simple processor core, called SimP, that can be easily modified and customized at the compile time as the application requires, and was used as an initial vehicle in our hardware/software co-design research.

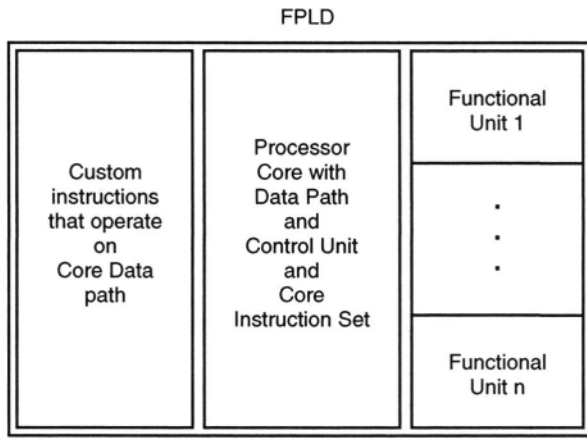



Figure 1.24 Custom-computing machine based on a fixed processor core

## 1.8 Questions and Problems

1.1 Describe the major differences between discrete logic, field-programmable logic and custom logic.

1.2 What are the major ways of classifying FPLDs?

1.3 How would you describe the impact of complexity and granularity of a logic element on the design of more complex logic?

1.4 What is the role of multiplexers in programmable logic circuits. Explain it using examples of implementation of different (alternative) data paths depending on the value of select inputs.

1.5 How do look-up tables (LUTs) implement logic functions? What are advantages of using LUTs for this purpose? What is the role of read and write operation on the look-up table?

1.6 Given five-input/single-output look-up table. How many memory locations it contains? How many different logic (Boolean) functions can be implemented in it? Implement the following Boolean functions using this table:

a) F(A, B, C, D, E) = ABC'D'E' + A'BCDE' + A'BCDE' + A'B'C'DE
b) F(A, B, C, D, E) = ABC + AB'CDE' + DE
c) F(A, B, C, D, E) = (A+B'+C)(A+B'+C'+D+E)(A'+B+E')

1.7 Four-input/single-output LUT is given as the basic building block for combinational logic. Implement the following logic functions

a) F(A, B, C, D, E) = AB'CDE' + ABC'D'E
b) F(A, B, C, D, E) = (A+B+C+E)(A+B'+D')(B'+C'+D+E)

using only LUTs of the given type. How many LUTs you need for this implementation? Show the interconnection of all LUTs and list contents of each of them.

1.8 "Design" your own FPLD circuits that contains only four-input/single-output LUTs based logic elements, which can fit the designs from the previous problem. Show partitioning and fitting of the design to your FPLD. Draw all connections assuming that sufficient number of long interconnect lines are available. Your FPLD should be organized as a matrix of LUTs.

1.9 List at least three advantages and disadvantages of segmented and non-segmented interconnection mechanism used in FPLDs.

1.10 Analyze a typical microprocessor-based embedded system that requires external RAM and ROM and address decoding to access other external chips. How would you implement address decoding using standard SSI/MSI components? How FPLD-based solution can reduce the number of components?

1.11 Give a few examples of hardware accelerators that can significantly improve performance of a microprocessor/DSP-based solution. Explain the advantages of implementing the accelerator in an FPLD.

1.12 What is the difference between reconfigurability and dynamic reconfigurability. Illustrate this with examples of use of each of them.

1.13 What are in-circuit and in-system programmability of FPLDs? What are their advantages for implementation of digital systems over other technologies?

1.14 What are the obstacles in implementing virtual hardware? Explain it on examples of currently available FPLD architectures.

1.15 Analyze a typical 8- or 16-bit microprocessor and its instruction set. How would you minimize the instruction set and processor architecture and still be able to implement practically any application?

# 2 EXAMPLES OF MAJOR FPLD FAMILIES

In this Chapter we will concentrate on a more detailed description of the two major FPLD families from Altera and Xilinx as two major manufacturers of FPLDs and companies providing very wide range of devices in terms of features and capacities. We will also give a brief description of the Atmel's family of FPLDs, which have some interesting architectural features and provide technology for full or partial dynamic reconfiguration. Their popularity comes from the high flexibility of individual devices, high circuit densities, flexible programming technologies, reconfigurability, as well as the range of design tools. We will emphasize the most important features found in FPLDs and their use in complex digital system design and prototyping.

## 2.1 Altera MAX 7000 Devices

As mentioned in Chapter 1, Altera has two different types of FPLDs: general purpose devices based on floating gate programming technology used in the MAX 5000, 7000 and 9000 series, and SRAM based Flexible Logic Element matriX (FLEX) 6000, 8000, 10K and 20K series. All Altera devices use CMOS process technology, which provides lower power dissipation and greater reliability than bipolar technology. Currently, Altera's devices are built on an advanced 0.5 and 0.35 μm technology, and the newest devices use 0.22 μm technology. MAX FPLDs from the 5000, 7000 and 9000 series are targeted for combinatorially intensive logic designs and complex finite state machines. In the following sections we will concentrate on the MAX 7000 series. This family provides densities ranging from 150 to 5,000 equivalent logic gates and pin counts ranging from 44 to 208 pins. The FLEX 8000 family provides logic density from 2,500 to 24,000 equivalent logic gates and pin counts from 84 to 304 pins. Predictable interconnect delays combined with the high register counts, low standby power, and in-circuit reconfigurability of FLEX 8000 make these devices suitable for high density, register intensive designs. The FLEX 10K family provides logic densities of up to 250,000 equivalent logic gates, and APEX 20K family densities that go up to 1,500,000 equivalent logic gates.

### *2.1.1 MAX 7000 devices general concepts*

The MAX 7000 family of high density, high performance MAX FPLDs provides dedicated input pins, user configurable I/O pins, programmable flip-flops, and clock options that ensure flexibility for integrating random logic functions. The MAX 7000 architecture supports emulation of standard TTL circuits and integration of SSI, MSI, and LSI logic functions. It also easily integrates multiple programmable logic devices from standard PALs, GALs, and MAX FPLDs. MAX 7000 MAX FPLDs use CMOS EEPROM cells as the programming technology to implement logic functions and contain from 32 to 256 logic cells, called macrocells, in groups of 16 macrocells, also called Logic Array Blocks (LABs). Each macrocell has a programmable-AND/fixed-OR array and a configurable register with independently programmable Clock, Clock Enable, Clear, and Preset functions.

Each MAX FPLD contains an AND array that provides product terms, which are essentially n-input AND gates. MAX FPLD schematics use a shorthand AND-array notation to represent several large AND gates with common inputs. An example of the same function in different representations is shown in Figure 2.1. In Figures 2.1(a), (b), and (c), a classic, sum-of-product and AND array notation are shown, respectively. A dot represents a connection between an input (vertical line) and one of the inputs to an n-input AND gate. If there is no connection to the AND gate, AND gate input is unused and floats to a logic 1.

The AND array circuit of Figure 2.1(c), with two 8-input AND gates, can produce any Boolean function of four variables (provided that only two product terms or simply p-terms are required) when expressed in sum-of-products form. The outputs of the product terms are tied to the inputs of an OR gate to compute the sum.

Product terms can also be used to generate complex control signals for use with programmable registers (Clock, Clock Enable, Clear, and Preset) or Output Enable signals for the I/O pins. These signals are called array control signals.

As discussed in Chapter 1, the Altera MAX FPLDs support programmable inversion allowing software to generate inversions, wherever necessary, without wasting macrocells for simple functions. Software also automatically applies De Morgan's inversion and other logic synthesis techniques to optimize the use of available resources.

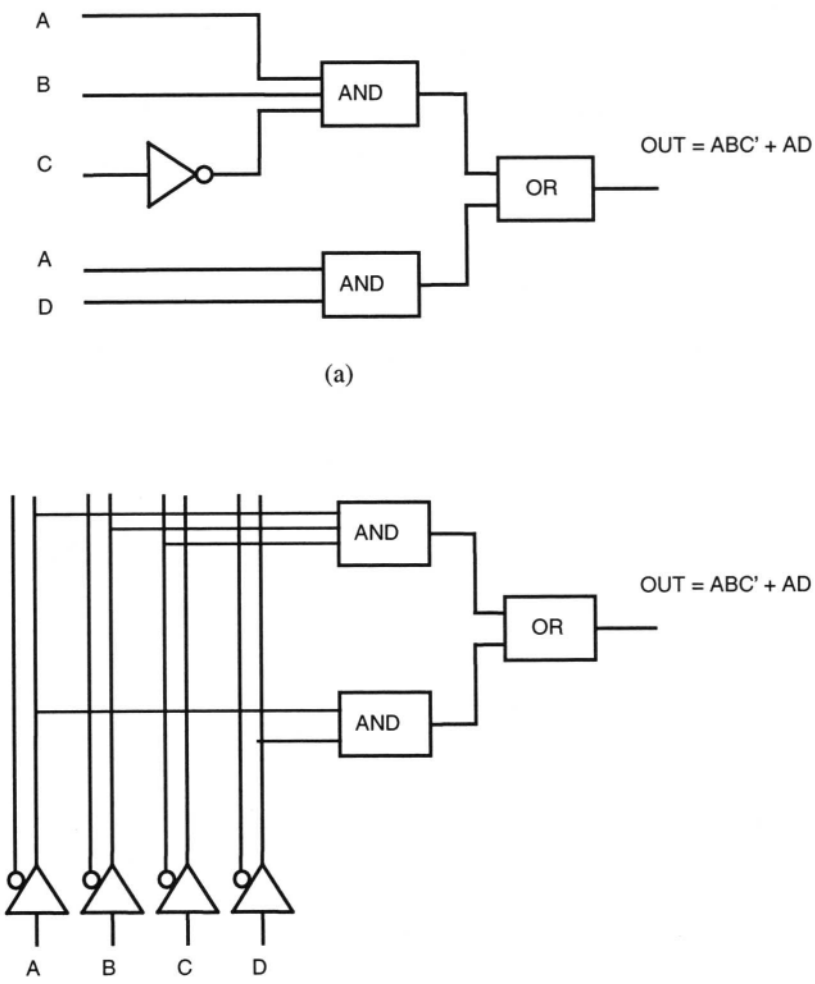



Figure 2.1 Different representations of logic function

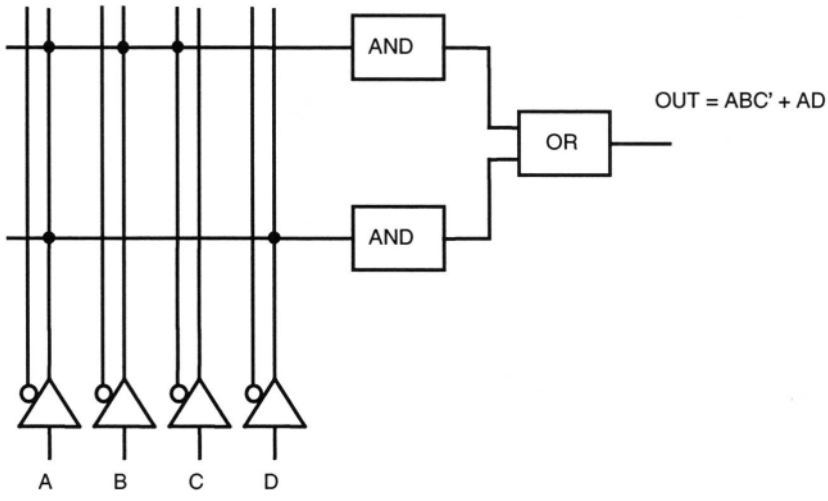



Figure 2.1 Different representations of logic function (cont.)

In the remaining sections of this Chapter we will present the functional units of the Altera MAX FPLD in enough detail to understand their operation and application potential.

### *2.1.2 Macrocell*

The fundamental building block of an Altera MAX FPLD is the macrocell. A MAX 7000 macrocell can be individually configured for both combinational and sequential operation. Each macrocell consists of three parts:

- A logic array that implements combinational logic functions
- A product term select matrix that selects product terms which take part in implementation of logic function
- A programmable register that provides D, T, JK, or SR options that can be bypassed

One typical macrocell architecture of MAX 7000 series is shown in Figure 2.2.

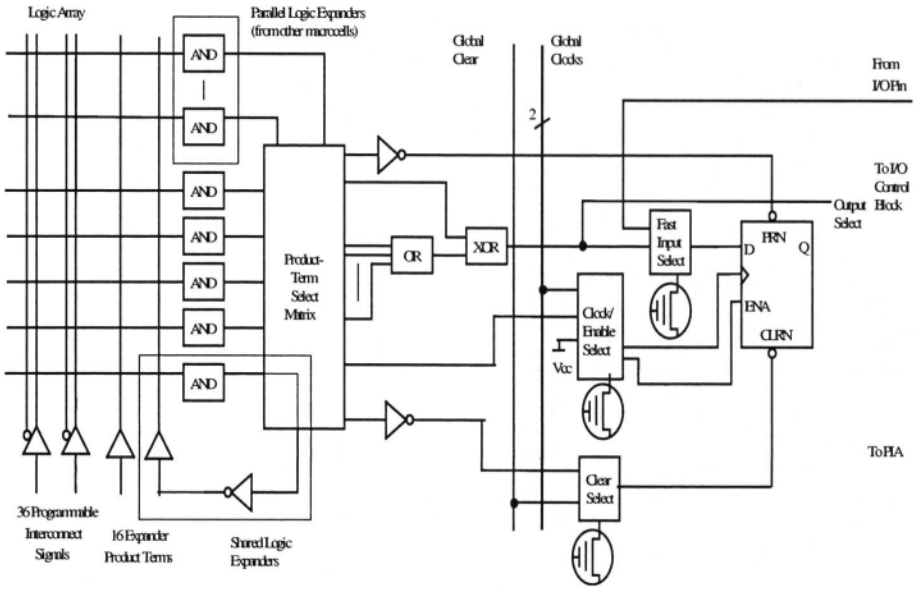



Figure 2.2 Macrocell architecture

A logic array consists of a programmable AND/ fixed OR array, known as PLA. Inputs to the AND array come from the true and complement of the dedicated input and clock pins from macrocell paths and I/O feedback paths. A typical logic array contains 5 p-terms that are distributed among the combinational and sequential resources. Connections are opened during the programming process. Any p-term may be connected to the true and complement of any array input signal. The p-term select matrix allocates these p-terms for use as either primary logic inputs (to the OR and XOR gates) to implement logic functions or as secondary inputs to the macrocell's register Clear, Preset, Clock, and Clock Enable functions. One p-term per macrocell can be inverted and fed back into the logic array. This "shareable" p-term can be connected to any p-term within the LAB.

Each macrocell flip-flop can be programmed to emulate D, T, JK, or SR operations with a programmable Clock control. If necessary, a flip-flop can be bypassed for combinational (non-registered) operation and can be clocked in three different modes:

- The first is by a global clock signal. This mode results with the fastest Clock to output performance.

- The second is by a global Clock signal and enabled by an active high Clock Enable. This mode provides an Enable on each flip-flop while still resulting in the fast Clock to output performance of the global Clock.
- Finally, the third is by an array Clock implemented with a p-term. In this mode, the flip-flop can be clocked by signals from buried macrocells or I/O pins.

Each register also supports asynchronous Preset and Clear functions by the p-terms selected by the p-term select matrix. Although the signals are active high, active low control can be obtained by inverting signals within the logic array. In addition, the Clear function can be driven by the active low, dedicated global Clear pin.

The flip-flops in macrocells also have a direct input path from the I/O pin, which bypasses PIA and combinational logic. This input path allows the flip-flop to be used as an input register with a fast input set up time (3 ns).

The more complex logic functions, those requiring more than five p-terms, can be implemented using shareable and parallel expander p-terms instead of additional macrocells. These expanders provide outputs directly to any macrocell in the same LAB.

Each LAB has up to 16 shareable expanders that can be viewed as a pool of uncommitted single p-terms (one from each macrocell) with inverted outputs that feed back into the logic array. Each shareable expander can be used and shared by any macrocell in the LAB to build complex logic functions.

Parallel expanders are unused p-terms from macrocells that can be allocated to a neighboring macrocells to implement fast, complex logic functions. Parallel expanders allow up to 20 p-terms to directly feed the macrocell OR logic five p-terms are provided by the macrocell itself and 15 parallel expanders are provided by neighboring macrocells in the LAB.

When both the true and complement of any signal are connected intact, a logic low 0 results on the output of the p-term. If both the true and complement are open, a logical "don't care" results for that input. If all inputs for the p-term are programmed opened, a logic high (1) results on the output of the p-term.

Several p-terms are input to a fixed OR whose output connects to an exclusive OR (XOR) gate. The second input to the XOR gate is controlled by a programmable resource (usually a p-term) that allows the logic array output to be inverted. In this

way active low or active high logic can be implemented, as well as the number of p-terms can be reduced (by applying De Morgan's inversion).

### *2.1.3 I/O Control Block*

The MAX FPLD I/O control block contains a tri state buffer controlled by one of the global Output Enable signals or directly connected to GND or Vcc, as shown in Figure 2.3. When the tri state buffer control is connected to GND, the output is in high impedance and the I/O pin can be used as a dedicated input. When the tri-state buffer control is connected to Vcc, the output is enabled.

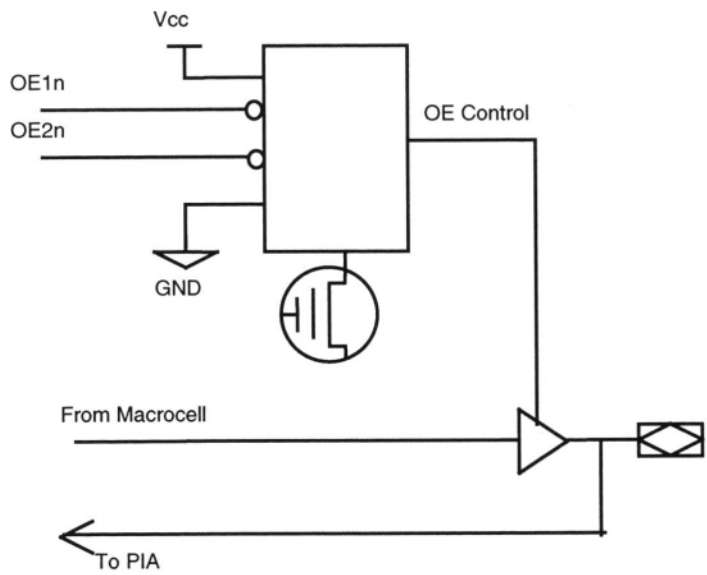



Figure 2.3 I/O control block

I/O pins may be configured as dedicated outputs, bi-directional lines, or as additional dedicated inputs. Most MAX FPLDs have dual feedback, with macrocell feedback being decoupled from the I/O pin feedback.

In the high end devices from the MAX 7000 family the I/O control block has six global Output Enable signals that are driven by the true or complement of two Output Enable signals (a subset of the I/O pins) or a subset of the I/O macrocells. This is shown in Figure 2.4. Macrocell and pin feedbacks are independent. When an I/O pin is configured as an input, the associated macrocell can be used for buried logic.

Additional features are found in the MAX 7000 series. Each macrocell can be programmed for either high speed or low power operation. The output buffer for each I/O pin has an adjustable output slew rate that can be configured for low noise or high speed operation. The fast slew rate should be used for speed critical outputs in systems that are adequately protected against noise.

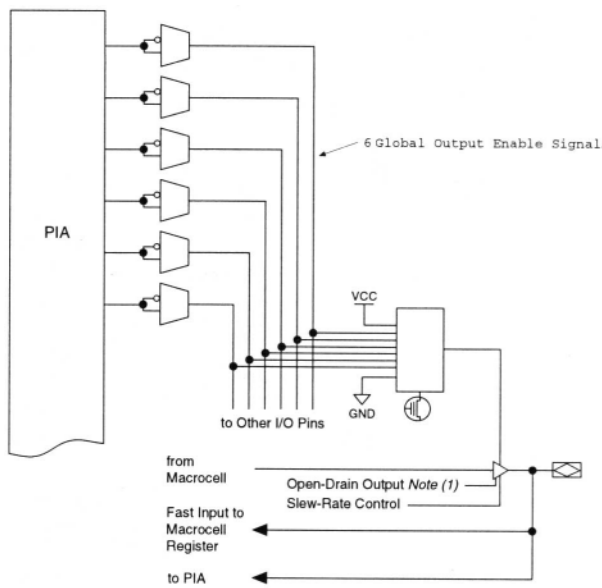



Figure 2.4 I/O control block in high end MAX 7000 devices

### *2.1.4 Logic Array Blocks*

Programmable logic in MAX FPLDs is organized into Logic Array Blocks (LABs). Each LAB contains a macrocell array, an expander product term array, and an I/O control block. The number of macrocells and expanders varies with each device. The general structure of the LAB is presented in Figure 2.5. Each LAB is accessible through Programmable Interconnect Array (PIA) lines and input lines. Macrocells are the primary resource for logic implementation, but expanders can be used to

supplement the capabilities of any macrocell. The outputs of a macrocell feed the decoupled I/O block, which consists of a group of programmable 3-state buffers and I/O pins. Macrocells that drive an output pin may use the Output Enable p-term to control the active high 3-state buffer in the I/O control block. This allows complete and exact emulation of 7400 series TTL family.

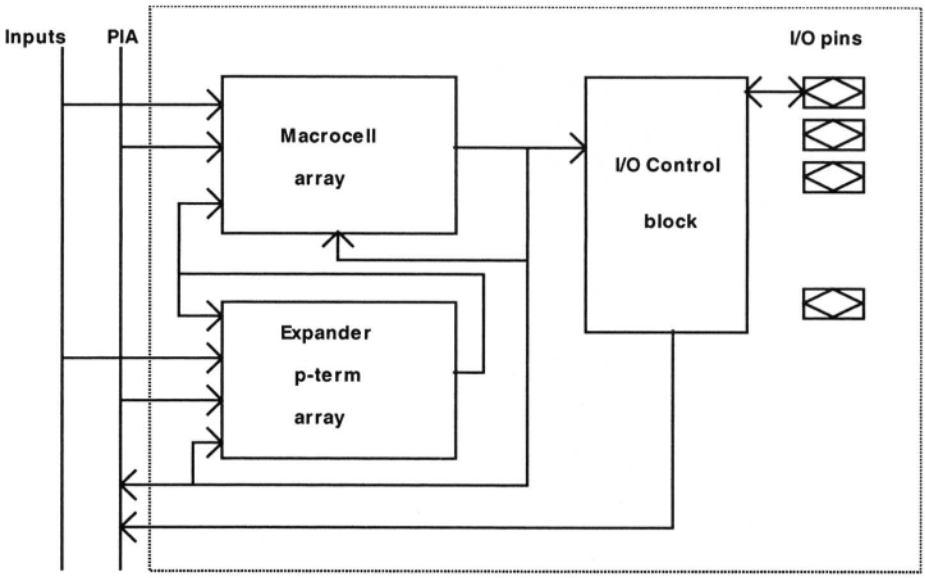



Figure 2.5 Logic Array Block architecture

Each LAB has two clocking modes: asynchronous and synchronous. During asynchronous clocking, each flip-flop is clocked by a p-term allowing that any input or internal logic to be used as a clock. Moreover, each flip-flop can be configured for positive or negative edge triggered operation.

Synchronous clocking is provided by a dedicated system clock (CLK). Since each LAB has one synchronous clock, all flip-flop clocks within it are positive edge triggered from the CLK pin.

Altera MAX FPLDs have an expandable, modular architecture allowing several hundreds to tens of thousands of gates in one package. They are based on a logic matrix architecture consisting of a matrix of LABs connected with a PIA, shown in

Figure 2.6. The PIA provides a connection path with a small fixed delay between all internal signal sources and logic destinations.

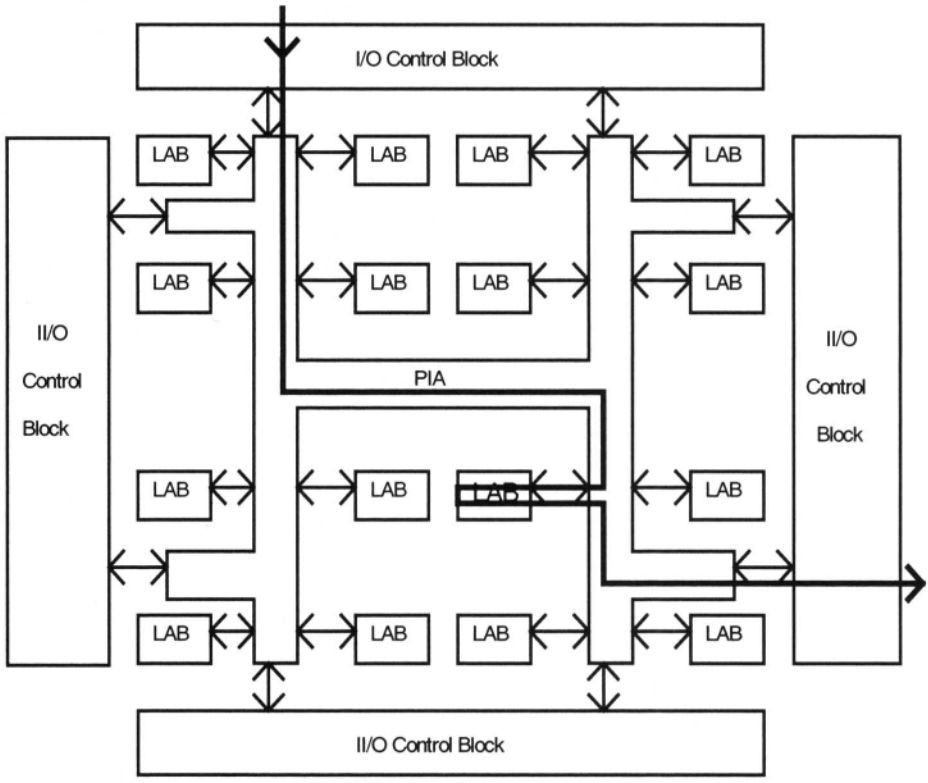



Figure 2.6 Altera MAX FPLD entire block diagram

### *2.1.5 Programmable Interconnect Array*

Logic is routed between LABs on the Programmable Interconnect Array (PIA), This global bus is programmable and enables connection of any signal source to any destination on the device. All dedicated inputs, I/O pins, and macrocell outputs feed the PIA, which makes them available throughout the entire device. An EEPROM cell controls one input of a 2-input AND gate which selects a PIA signal to drive into the LAB, as shown in Figure 2.7. Only signals required by each LAB are actually routed into the LAB.

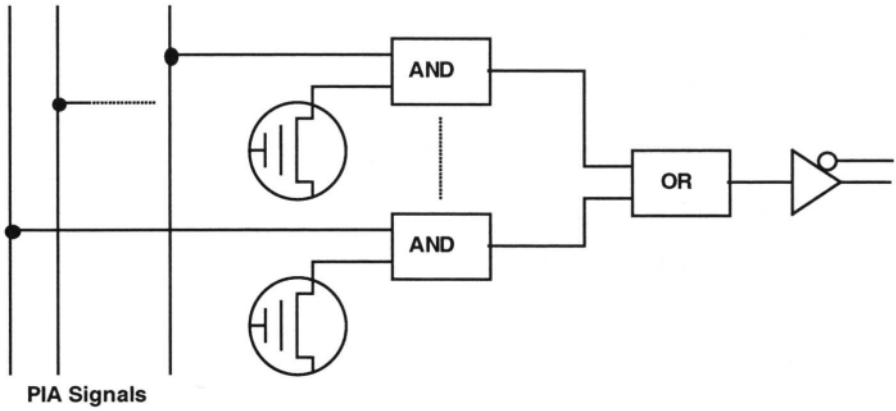



Figure 2.7 PIA routing

While routing delays in channel based routing schemes in MPGAs and FPGAs are cumulative, variable, and path dependent, the MAX PIA has a fixed delay. Therefore, it eliminates skew between signals and makes timing performance easy to predict. MAX 7000 devices have fixed internal delays allowing the user to determine the worst case timing for any design.

### *2.1.6 Programming*

Programming of MAX 7000 devices consists of configuring EEPROM transistors as required by design. The normal programming procedure consists of the following steps:

1. The programming pin ($V_{pp}$) is raised to the super high input level (usually 12.5V).
2. Row and column address are placed on the designated address lines (pins).
3. Programming data is placed on the designated data lines (pins).
4. The programming algorithm is executed with a sequence of 100 microsecond programming pulses separated by program verify cycles.
5. Overprogram or margin pulses may be applied to double ensure EPLD programming.

The programming operation is typically performed eight bits at a time on specialized hardware. The security bit can be set to ensure EPLD design security.

Some of the devices from the MAX 7000 family have special features such as 3.3 V operation or power management. The 3.3 V operation offers power savings of 30% to 50% over 5.0 V operation. The power saving features include a programmable power saving mode and power down mode. Power down mode allows the device to consume near zero power (typically 50 **μA).** This mode of operation is controlled externally by the dedicated power down pin. When this signal is asserted, the power down sequence latches all input pins, internal logic, and output pins preserving their present state.

## 2.2 Altera FLEX 8000

Altera's Flexible Logic Element Matrix (FLEX) programmable logic combines the high register counts of CPLDs and the fast predictable interconnects of EPLDs. It is SRAM based providing low stand-by power and in circuit reconfigurability. Logic is implemented with 4-input look-up tables (LUTs) and programmable registers. High performance is provided by a fast, continuous network of routing resources. FLEX 8000 devices are configured at system power up, with data stored in a serial configuration EPROM device or provided by a system controller. Configuration data can also be stored in an industry standard EPROM or downloaded from system RAM. Since reconfiguration requires less than 100 ms, real-time changes can be made during system operation.

The FLEX architecture incorporates a large matrix of compact logic cells called logic elements (LEs). Each LE contains a 4-input LUT that provides combinatorial logic capability and also contains a programmable register that offers sequential logic capability. LEs are grouped into sets of eight to create Logic Array Blocks (LABs). Each LAB is an independent structure with common inputs, interconnections, and control signals. LABs are arranged into rows and columns. The I/O pins are supported by I/O elements (IOEs) located at the ends of rows and columns. Each IOE contains a bi-directional I/O buffer and a flip-flop that can be used as either an input or output register. Signal interconnections within FLEX 8000 devices are provided by FastTrack Interconnect continuous channels that run the entire length and width of the device. The architecture of FLEX 8000 device is illustrated in Figure 2.8.

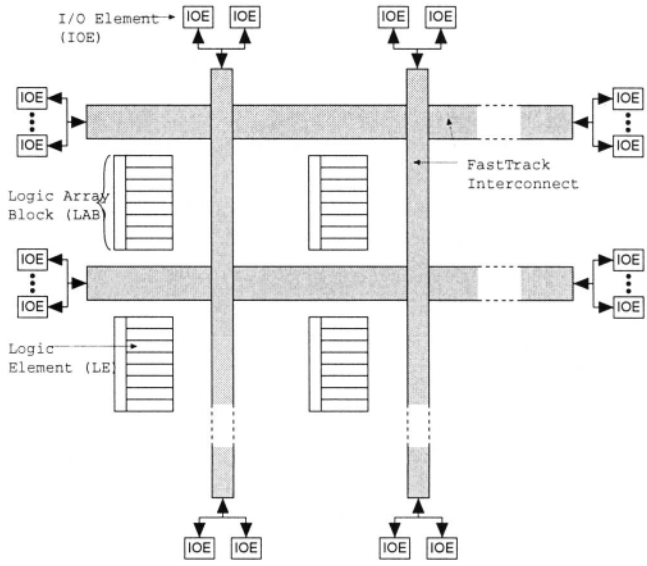



Figure 2.8 FLEX 8000 device architecture

### *2.2.1 Logic Element*

A logic element (LE) is the basic logic unit in the FLEX 8000 architecture. Each LE contains a 4-input LUT, a programmable flip-flop, a carry chain, and a cascade chain as shown in Figure 2.9.

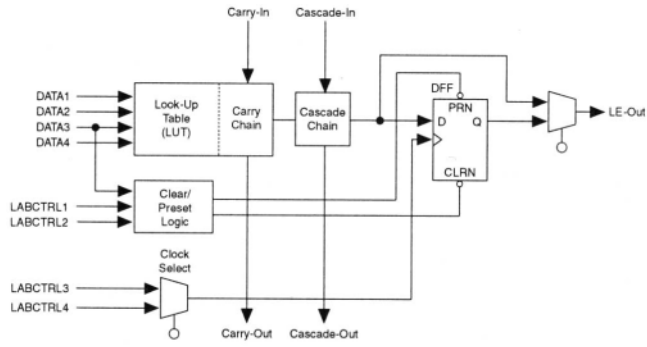



Figure 2.9 Logic element

The LUT quickly computes any Boolean function of four input variables. The programmable flip-flop can be configured for D, T, JK, or SR operation. The Clock, Clear, and Preset control signals can be driven by dedicated input pins, general purpose I/O pins, or any internal logic. For combinational logic, the flip-flop is bypassed and the output of the LUT goes directly to the output of the LE.

Two dedicated high speed paths are provided in the FLEX 8000 architecture; thecarry chain and cascade chain both connect adjacent LEs without using general purpose interconnect paths. The carry chain supports high speed adders and counters. The cascade chain implements wide input functions with minimal delay. Carry and cascade chains connect all LEs in a LAB and all LABs of the same row.

The carry chain provides a very fast (less than 1 ns) carry forward function between LEs. The carry-in signal from a lower order bit moves towards the higher order bit by way of the carry chain and also feeds both the LUT and a portion of the carry chain of the next LE. This feature allows implementation of high speed counters and adders of practically arbitrary width. A 4-bit parallel full adder can be implemented in 4+1=5 LEs by using the carry chain as shown in Figure 2.10. The LEs look-up table is divided into two portions. The first portion generates the sum of two bits using input signals and the carry-in signal. The other generates the carry-out signal, which is routed directly to the carry-in input of the next higher order bit. The final carry-out signal is routed to an additional LE, and can be used for any purpose.

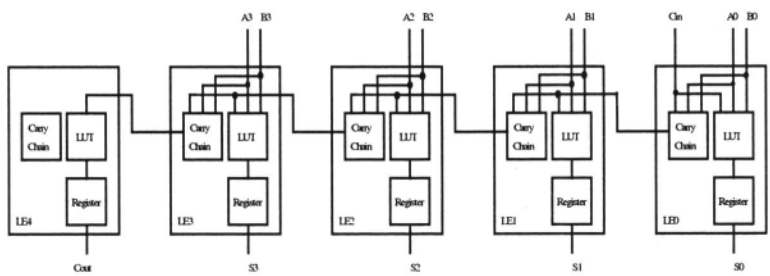



Figure 2.10 Carry chain illustration

With the cascade chain, the FLEX 8000 architecture can implement functions with a very wide fan-in. Adjacent LUTs can be used to compute portions of the function in parallel, while the cascade chain serially connects the intermediate values. The cascade chain can use a logical AND or logical OR to connect the outputs of adjacent LEs. Each additional LE provides four more inputs to the effective width of a function adding a delay of approximately 1 ns per LE. Figure

2.11 illustrates how the cascade function can connect adjacent LEs to form functions with wide fan-in.

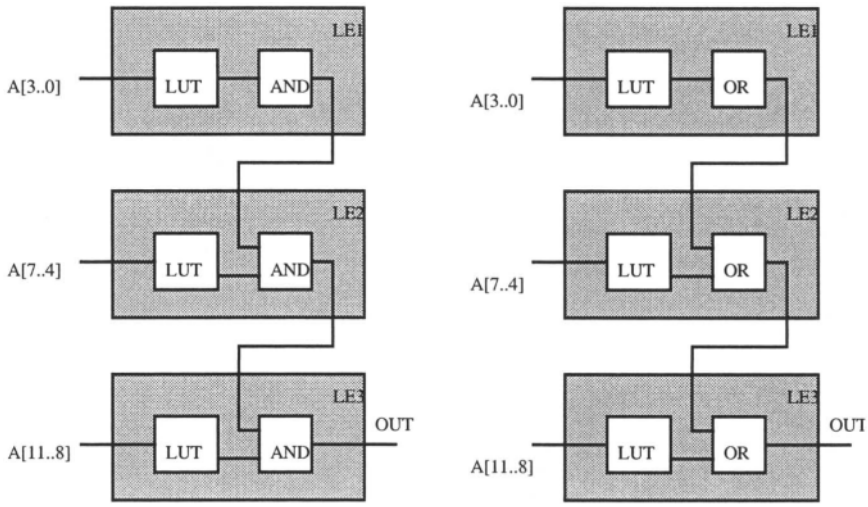



Figure 2.11 Cascade chain illustration

The LE can operate in the four different modes (shown in Figure 2.12). In each mode, seven of the ten available inputs to the LE - the four data inputs from the LAB, local interconnect, the feedback from the programmable register, and the carry-in from the previous LE are directed to different destinations to implement the desired logic function. The remaining inputs provide control for the register.

The normal mode is suitable for general logic applications and wide decode functions that can take advantage of a cascade chain.

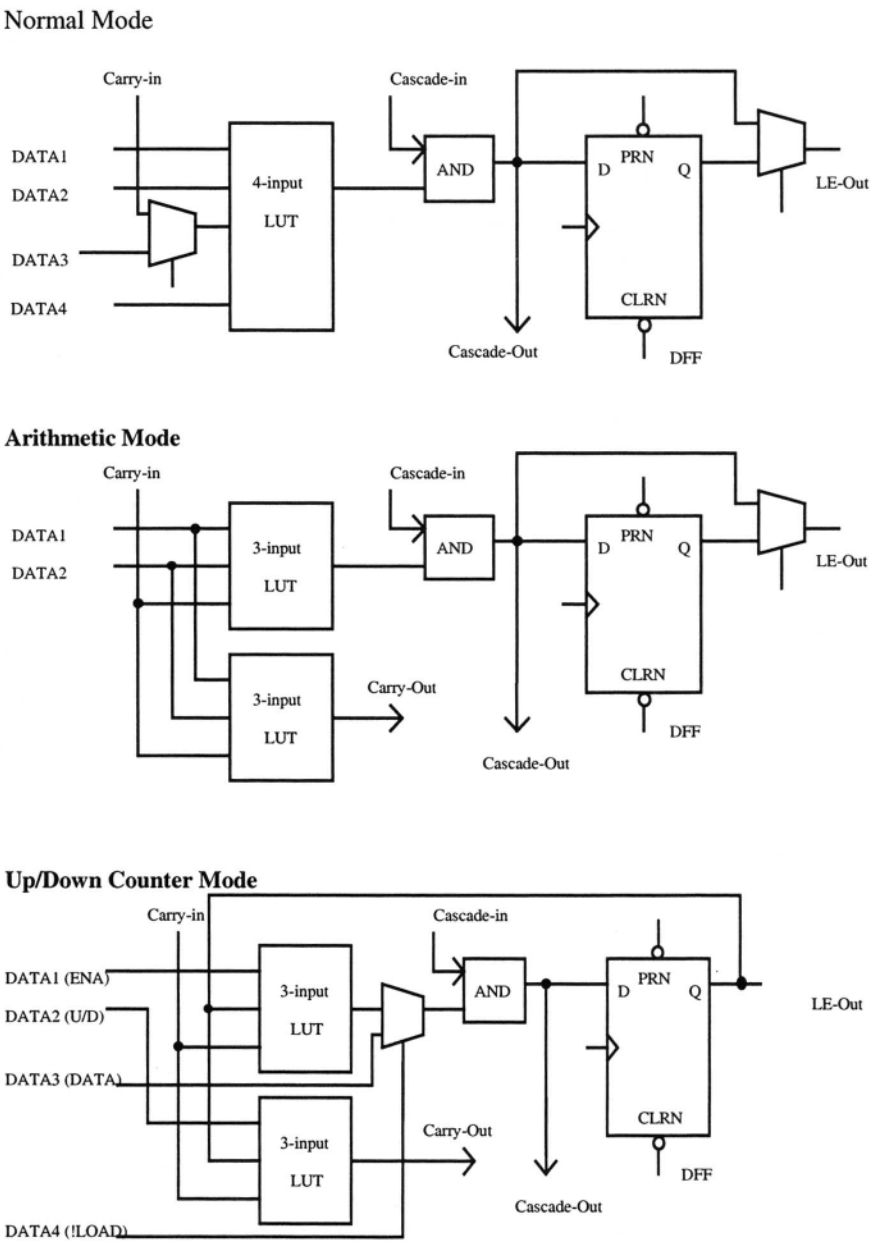



Figure 2.12 LE operating modes

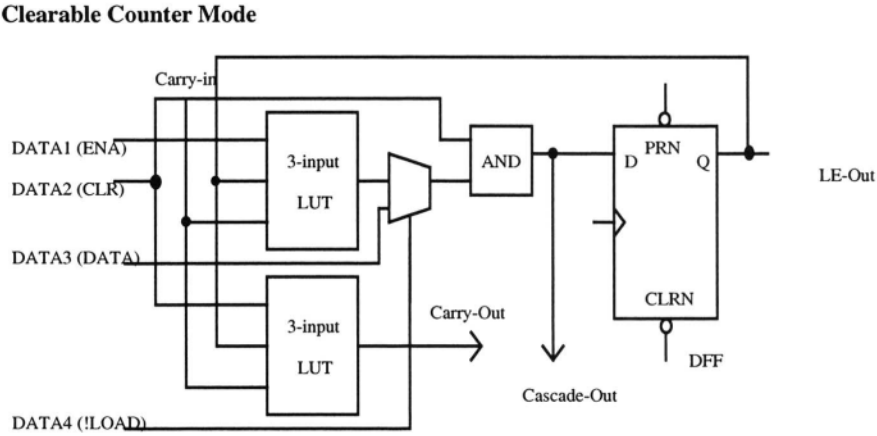



Figure 2.12 LE operating modes (continued)

The arithmetic mode offers two 3-input LUTs that are ideal for implementing adders, accumulators, and comparators. One LUT provides a 3-bit Boolean function, and the other generates a carry bit. The arithmetic mode also supports a cascade chain.

The Up/Down counter mode offers counter enable, synchronous up/down control, and data loading options. Two 3-input LUTs are used: one generates the counter data, the other generates the fast carry bit. A 2-to-1 multiplexer provides synchronous loading. Data can also be loaded asynchronously with the Clear and Preset register control signals.

The clearable counter mode is similar to the Up/Down counter mode, but supports a synchronous Clear instead of the up/down control. The Clear function is substituted for Cascade-in signal in Up/Down Counter mode. Two 3-input LUTs are used: one generates the counter data, the other generates the fast carry bit.

The Logic controlling a register's Clear and Preset functions is controlled by the DATA3, LABCTRL1, and LABCTRL2 inputs to LE, as shown in Figure 2.13. Default values for the Clear and Preset signals, if unused, are logic highs.

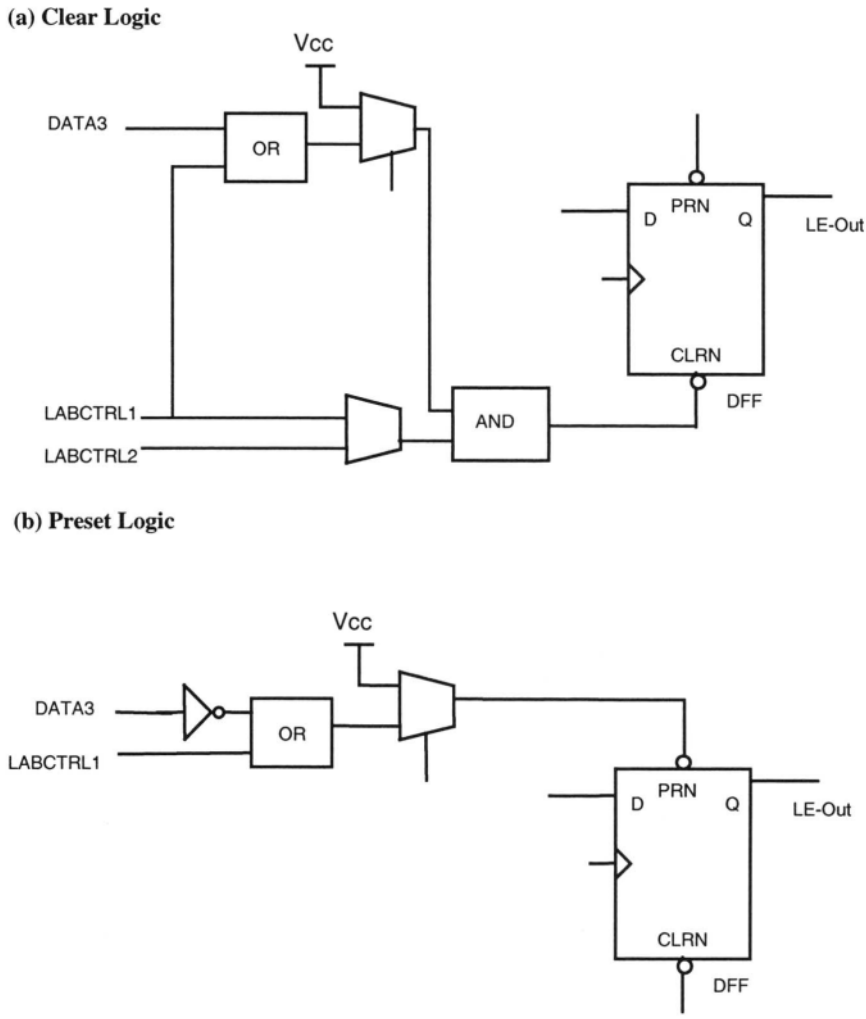



Figure 2.13 LE Clear and Preset Logic

If the flip-flop is cleared by only one of two LABCTRL signals, the DATA3 input is not required and can be used for one of the logic element operating modes.

### *2.2.2 Logic Array Block*

A Logic Array Block (LAB) consists of eight LEs, their associated carry chains, cascade chains, LAB control signals, and the LAB local interconnect. The LAB structure is illustrated in Figure 2.14. Each LAB provides four control signals that can be used in all eight LEs.

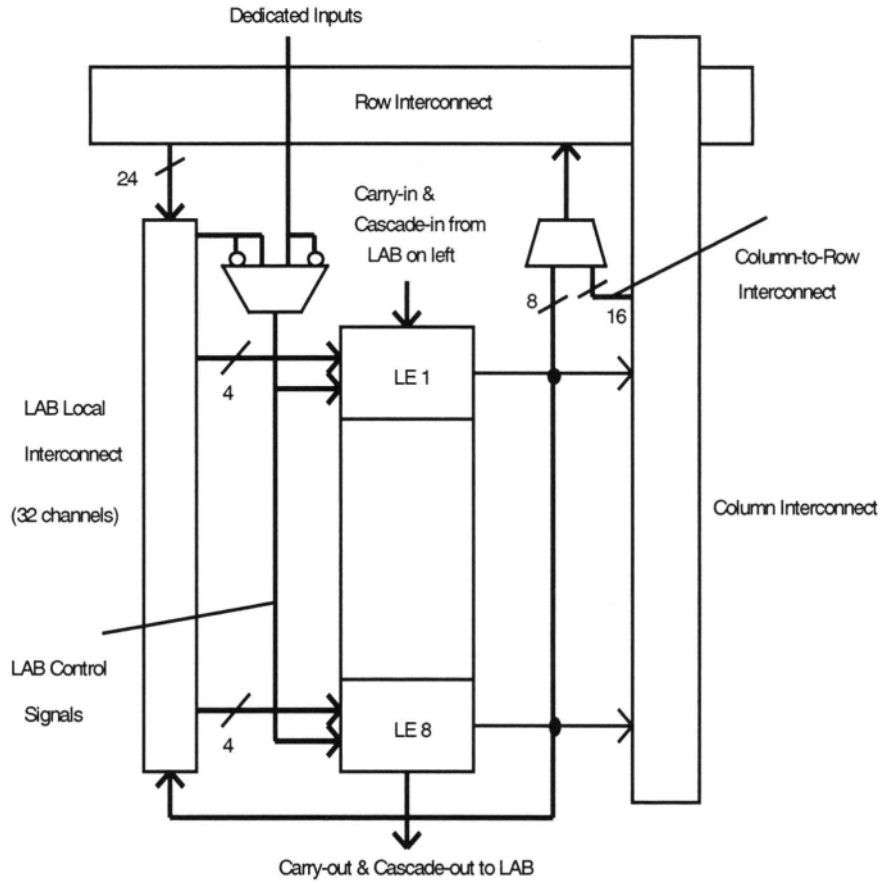



Figure 2.14 LAB Internal Architecture

### *2.2.3 FastTrack Interconnect*

Connections between LEs and device I/O pins are provided by the FastTrack Interconnect mechanism represented by a series of continuous horizontal and vertical routing channels that traverse the entire device. The LABs within the device are arranged into a matrix of columns and rows. Each row has a dedicated interconnect that routes signals into and out of the LABs in the row. The row interconnect can then drive I/O pins or feed other LABs in the device. Figure 2.15 shows how an LE drives the row and column interconnect.

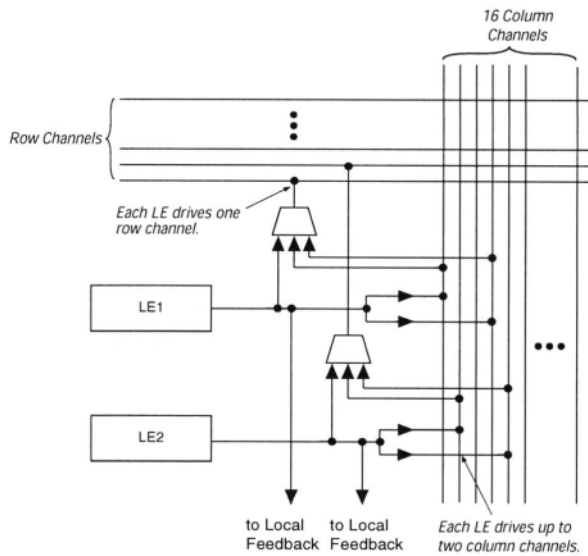



Figure 2.15 LAB Connections to Row and Column Interconnect

Each LE in a LAB can drive up to two separate column interconnect channels. Therefore, all 16 available column channels can be driven by a LAB. The column channels run vertically across the entire device and LABs in different rows share access to them by way of partially populated multiplexers. A row interconnect channel can be fed by the output of the LE or by two column channels. These three signals feed a multiplexer that connects to a specific row channel. Each LE is connected to one 3-to-1 multiplexer. In a LAB, the multiplexers provide all 16 column channels with access to the row channels.

Each column of LABs has a dedicated column interconnect that routes signals out of the LABs in that column. The column interconnect can drive I/O pins or feed into the row interconnect to route the signals to other LABs in the device. A signal from the column interconnect, which can be either the output from an LE or an

input from an I/O pin, must transfer to the row interconnect before it can enter a LAB. Figure 2.16 shows the interconnection of four adjacent LABs with row, column, and local interconnects, as well as associated cascade and carry chains.

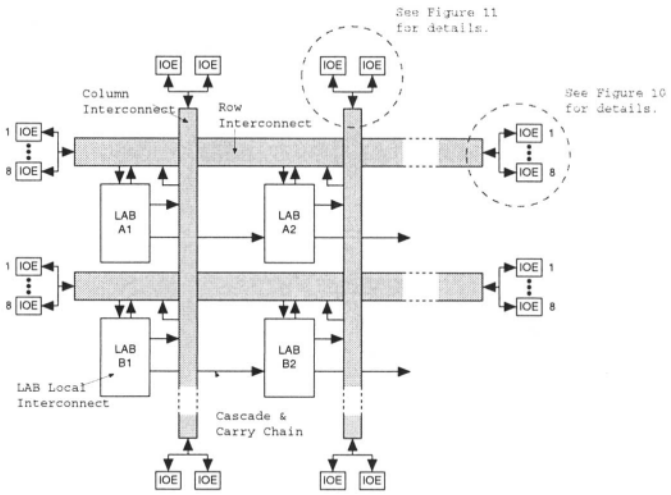



Figure 2.16 Device Interconnect Resources

The Interconnection between row interconnect channels and IOEs is illustrated in Figure 2.17. An input signal from an IOE can drive two row channels. When an IOE is used as an output, the signal is driven by an n-to-1 multiplexer that selects the row channels. The size of the multiplexer depends on the number of columns in the device. Eight IOEs are connected to each side of the row channels.

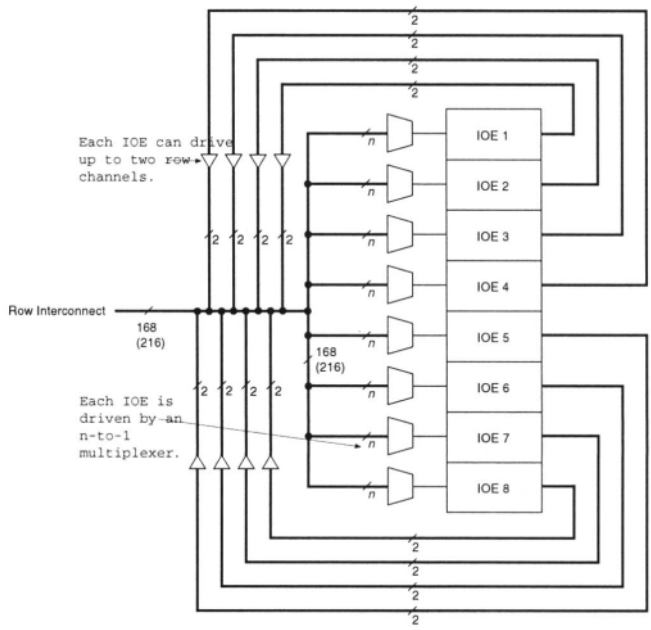



Figure 2.17 Row to IOE Connection

On the top and bottom of the column channels are two IOEs, as shown in Figure 2.18. When an IOE is used as an input, it can drive up to 2 column channels. The output signal to an IOE can choose from 8 column channels through an 8-to-1 multiplexer.

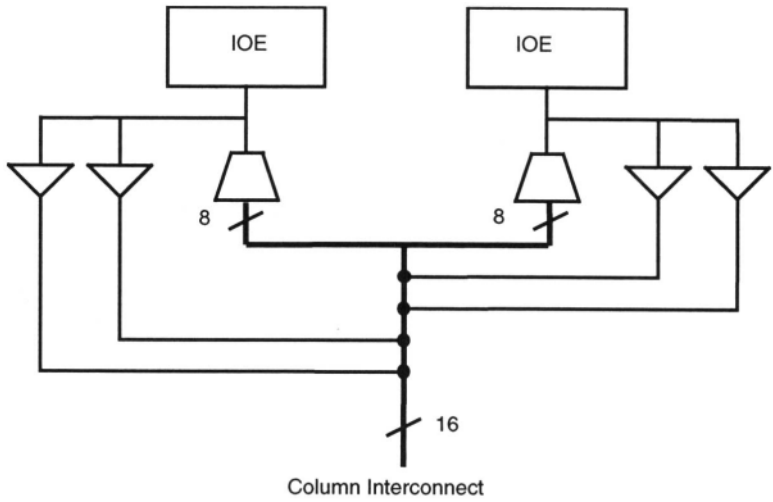



Figure 2.18 Column to IOE Connection

### *2.2.4 Dedicated I/O Pins*

In addition to general purpose I/O pins, four dedicated input pins provide low skew and device wide signal distribution. Typically, they are used for global Clock, Clear, and Preset control signals. These signals are available for all LABs and IOEs in the device. The dedicated inputs can be used as general purpose data inputs for nets with large fan-outs because they feed the local interconnect.

### *2.2.5 Input/Output Element*

Input/Output Element (IOE) architecture is presented in Figure 2.19. IOEs are located at the ends of the row and column interconnect channels. I/O pins can be used as input, output, or bi-directional pins. Each I/O pin has a register that can be used either as an input or output register in operations requiring high performance (fast set up time or fast Clock to output time). The output buffer in each IOE has an adjustable slew rate.

A fast slew rate should be used for speed critical outputs in systems protected against noise. Clock, Clear, and Output Enable controls for the IOE are provided by a network of I/O control signals. These signals are supplied by either the dedicated input pins or internal logic. All control signal sources are buffered onto high speed drivers that drive the signals around the periphery of the device. This "peripheral

bus" can be configured to provide up to four Output Enable signals and up to two Clock or Clear signals.

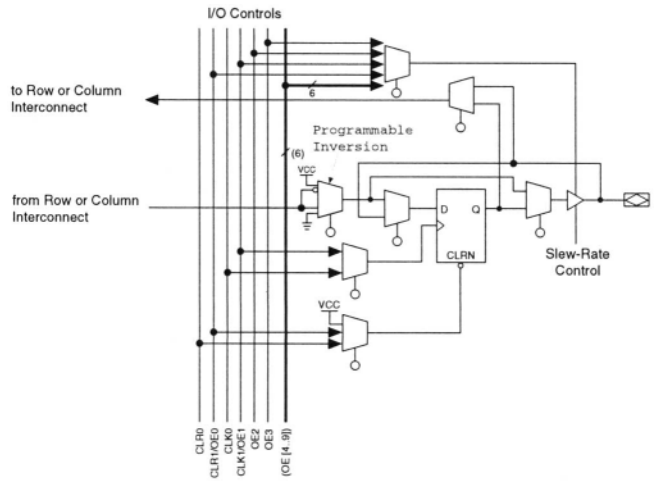



Figure 2.19 IOE Architecture

The signals for the peripheral bus are generated by any of the four dedicated inputs or signals on the row interconnect channels, as shown in Figure 2.20.

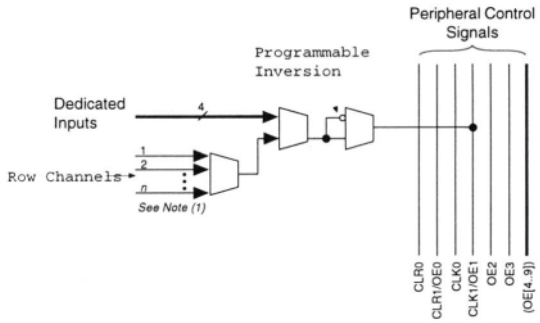



Figure 2.20 Peripheral Bus

The number of row channels used depends on the number of columns in the device. The six peripheral control signals can be accessed by every I/O element.

### *2.2.6 Configuring FLEX 8000 Devices*

The FLEX 8000 family supports several configuration schemes for loading the design into a chip on the circuit board. The FLEX 8000 architecture uses SRAM cells to store configuration data for the device. These SRAM cells must be loaded each time the circuit powers up and begins operation. The process of physically loading the SRAM with programming data is called configuration. After configuration, the FLEX 8000 device resets its registers, enables I/O pins, and begins operation as a logic device. This reset operation is called initialization. Together, the configuration and initialization processes are called the command mode. Normal in-circuit device operation is called the user mode.

The entire command mode requires less than 100 ms and can be used to dynamically reconfigure the device even during system operation. Device configuration can occur either automatically at system power up or under control of external logic. The configuration data can be loaded into FLEX 8000 device with one of six configuration schemes, which is chosen on the basis of the target application.

There are two basic types of configuration schemes: active, and passive. In an active configuration scheme, the device controls the entire configuration process and generates the synchronization and control signals necessary to configure and initialize itself from external memory. In a passive configuration scheme, the device is incorporated into a system with an intelligent host that controls the configuration process. The host selects either a serial or parallel data source and the data is transferred to the device on a common data bus. The best configuration scheme depends primarily on the particular application and on factors such as the need to reconfigure in real time, the need to periodically install new configuration data, as well as other factors.

Generally, an active configuration scheme provides faster time to market because it requires no external intelligence. The device is typically configured at system power up, and reconfigured automatically if the device senses power failure. A passive configuration scheme is generally more suitable for fast prototyping and development (for example from development Max+PLUS II software) or in applications requiring real-time device reconfiguration. Reconfigurability allows reuse of logic resources instead of designing redundant or duplicate circuitry in a system. Short descriptions of several configuration schemes are presented in the following sections.

**Active Serial Configuration**

This scheme, with a typical circuit shown in Figure 2.21, uses Altera's serial configuration EPROM as a data source for FLEX 8000 devices. The nCONFIG pin is connected to Vcc, so the device automatically configures itself at system power up. Immediately after power up, the device pulls the nSTATUS pin low and releases it within 100 ms. The DCLK signal clocks serial data bits from the configuration EPROM. When the configuration is completed, the CONF_DONE signal is released causing the nCS to activate and bring the configuration EPROM data output into a high impedance state. After CONF_DONE goes high, the FLEX 8000 completes the initialization process and enters user mode. In the circuit shown in Figure 2.21, the nCONFIG signal is tied up to the Output Enable (OE) input of the configuration EPROM. External circuitry is necessary to monitor nSTATUS of the FLEX device in order to undertake appropriate action if configuration fails.

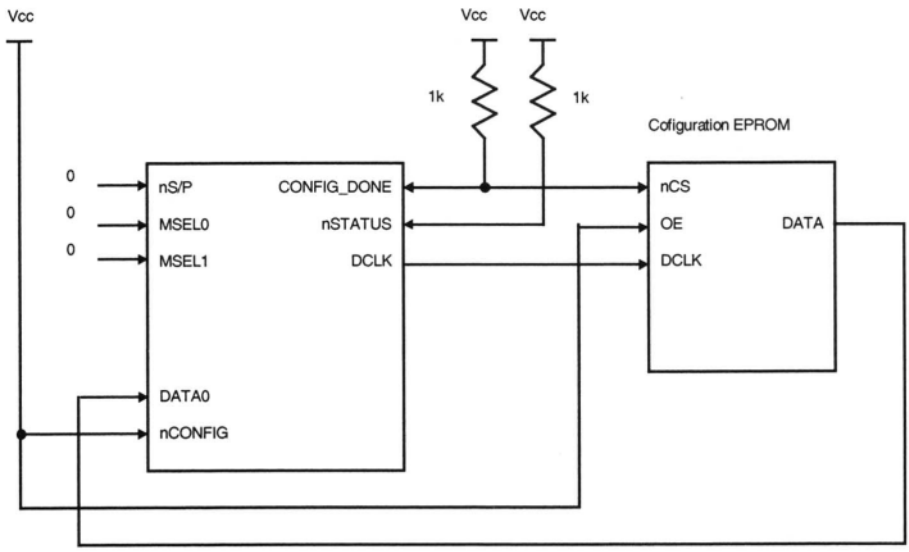



Figure 2.21 Active Serial Device Configuration

**Active Parallel Up (APU) and Active Parallel Down (APD) Configuration**

In Active Parallel Up and Active Parallel Down configuration schemes, the FLEX 8000 device generates sequential addresses that drive the address inputs to an external EPROM. The EPROM then returns the appropriate byte of data on the data lines DATA[7..0]. Sequential addresses are generated until the device has been

completely loaded. The CONF_DONE pin is then released and pulled high externally indicating that configuration has been completed. The counting sequence is ascending (00000H to 3FFFFH) for APU or descending (3FFFFH to 00000H) for APD configuration. A typical circuit for parallel configuration is shown in Figure 2.22.

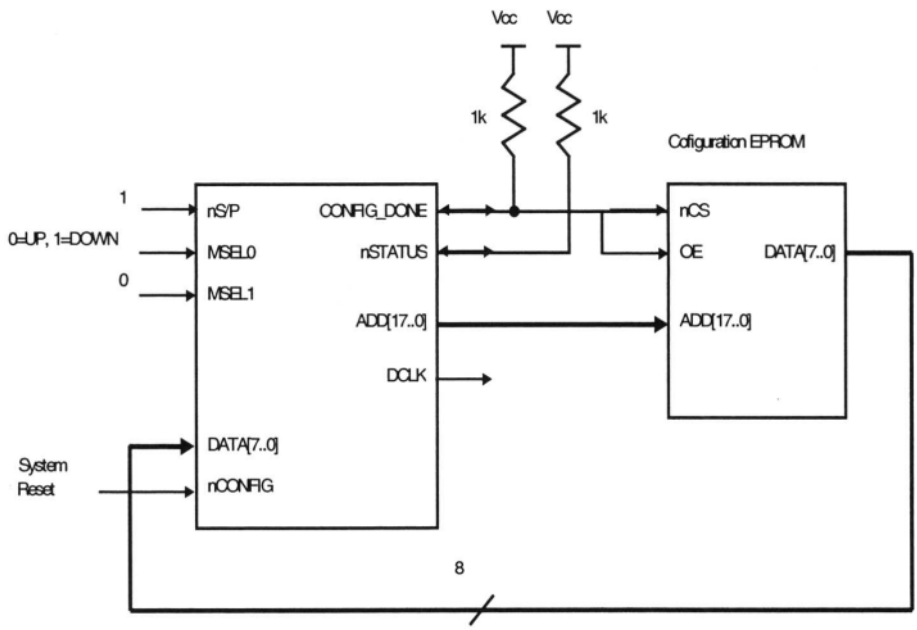



Figure 2.22 APU and APD Configuration with a 256 Kbyte EPROM

On each pulse of the RDCLK signal (generated by dividing DCLK by eight), the device latches an 8-bit value into a serial data stream. A new address is presented on the ADD[17..0] lines a short time after a rising edge on RDCLK. External parallel EPROM must present valid data before the subsequent rising edge of RDCLK, which is used to latch data based on address generated by the previous clock cycle.

Both active parallel configuration schemes can generate addresses in either an ascending or descending order. Counting up is appropriate if the configuration data is stored at the beginning of an EPROM or at some known offset in an EPROM larger of 256 Kbytes. Counting down is appropriate if the low addresses are not available, for example if they are used by the CPU for some other purpose.

**Passive Parallel Synchronous Configuration**

In this scheme the FLEX 8000 device is tied to an intelligent host. The DCLK, CONF_DONE, nCONFIG, and nSTATUS signals are connected to a port on the host, and the data can be driven directly onto a common data bus between the host and the FLEX 8000 device. New byte of data is latched on every eighth rising edge of DCLK signal, and serialized on every eight falling edge of this signal, until the device is completely configured.

A typical circuit for passive serial configuration is shown in Figure 2.23. The CPU generates a byte of configuration data. Data is usually supplied from a microcomputer 8-bit port. Dedicated data register can be implemented with an octal latch. The CPU generates clock cycles and data; eight DCLK cycles are required to latch and serialize each 8-bit data word and a new data word must be present at the DATA[7..0] inputs upon every eight DCLK cycles.

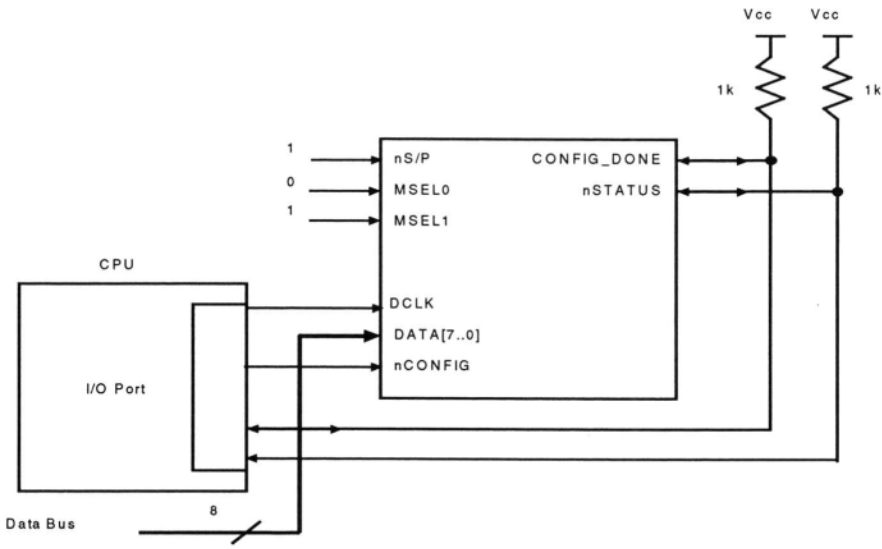



Figure 2.23 Parallel Passive Synchronous Configuration

**Passive Parallel Asynchronous Configuration**

In this configuration, a FLEX 8000 device can be used in parallel with the rest of the board. The device accepts a parallel byte of input data, then serializes the data with its internal synchronization clock. The device is selected with nCS and CS chip

select input pins. A typical circuit with a microcontroller as an intelligent host is shown in Figure 2.24. Dedicated I/O ports are used to drive all control signals and the data bus to the FLEX 8000 device. The CPU performs handshaking with a device by sensing the RDYnBUSY signal to establish when the device is ready to receive more data. The RDYnBUSY signal falls immediately after the rising edge of the nWS signal that latches data, indicating that the device is busy. On the eighth falling edge of DCLK, RDYnBUSY returns to Vcc, indicating that another byte of data can be latched.

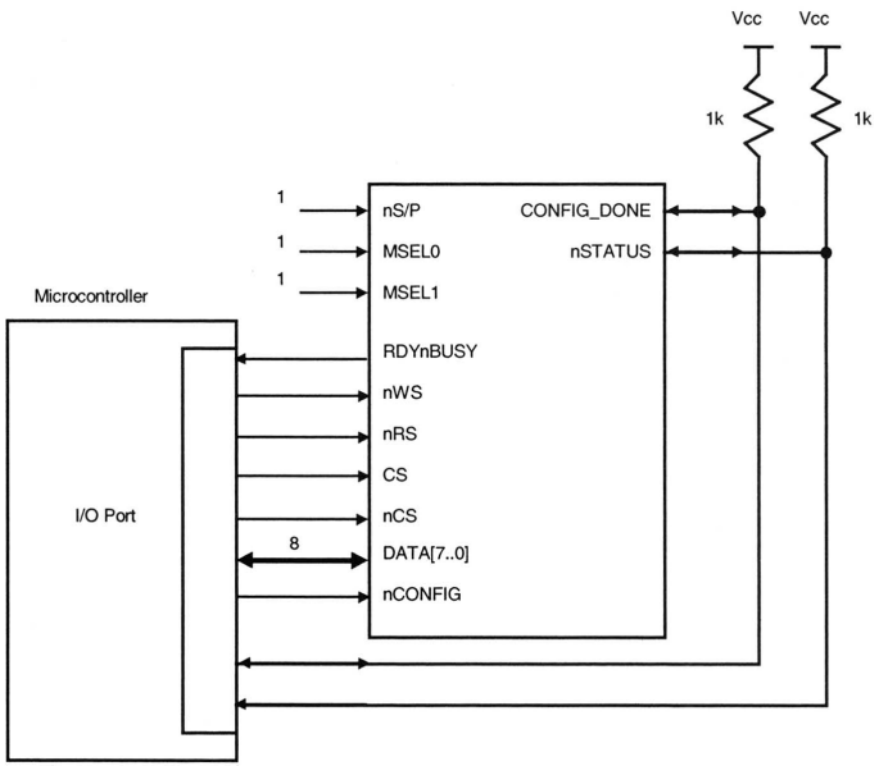



Figure 2.24 Passive Parallel Asynchronous Configuration

## Passive Serial Configuration

The passive serial configuration scheme uses an external controller to configure the FLEX 8000 device with a serial bit stream. The FLEX device is treated as a slave and no handshaking is provided. Figure 2.25 shows how a bit-wide passive

configuration is implemented. Data bits are presented at the DATA0 input with the least significant bit of each byte of data presented first. The DCLK is strobed with a high pulse to latch the data. The serial data loading continues until the CONF_DONE goes high indicating that the device is fully configured. The data source can be any source that the host can address.

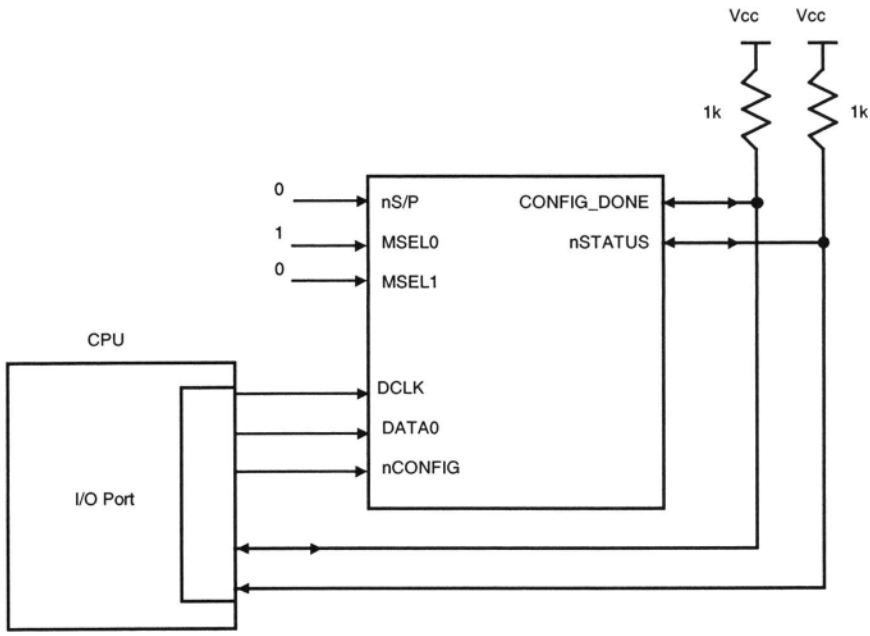



Figure 2.25 Bit-Wide Passive Serial Configuration

### *2.2.7 Designing with FLEX 8000 Devices*

In both types of the Altera's FPLD architectures, trade-offs are made to optimize designs for either speed or density. The FLEX 8000 architecture allows control of the speed/density trade-offs. In addition, Altera's Max+PLUS II software can automatically optimize all or part of a circuit for speed or density.

The Altera FLEX 8000 architecture is supported by design methods that offer a full spectrum of low to high level control over actual design implementation. If a fast design cycle is the primary goal, the design can be described with high level constructs in a hardware description language such as VHDL, Verilog, or AHDL.

If the designer wants to optimize performance and density, the design can be described with primitive gates and registers ("gate level" design) using hardware description language or schematics. Family specific macrofunctions are also available.

Different logic options and synthesis styles can be used (set up) to optimize a design for a particular design family. Also different options can be used in portions of the design to improve the overall design. The following design guidelines yield maximum speed, reliability, and device resource utilization, while minimizing fitting problems.

1. Reserve Resources for Future Expansions. Because designs are modified and extended, we recommend leaving 20% of a device's logic cells and I/O pins unused.
2. Allow the Compiler to Select Pin & Logic Cell Assignment. Pin & logic cell assignments, if poorly or arbitrarily selected, can limit the Max+PLUS II compiler's ability to arrange signals efficiently, reducing the probability of a successful fit. We recommend the designer allow the compiler to choose all pin and logic cell locations automatically.
3. Balance Ripple Carry & Carry Look Ahead Usage. The dedicated carry chain in the FLEX 8000 architecture propagate a ripple carry for short and medium length counters and adders with minimum delay. Long carry chains, however, restrict the compiler's ability to fit a design because the LEs in the chain must be contiguous. On the other hand, look ahead counters do not require the use of adjacent logic cells. This allows the compiler to arrange and permute the LEs to map the design into the device more efficiently.
4. Use Global Clock & Clear Signals. Sequential logic is most reliable if it is fully synchronous, that is if every register in the design is clocked by the same global clock signal and reset by the same global clear signal. Four dedicated high speed, low skew global signals are available throughout the device, independent of FastTrack interconnect, for this purpose. Figure 2.13 shows the register control signals in the FLEX 8000 device. The Preset and Clear functions of the register can be functions of LABCTRL1, LABCTRL2, and DATA3. The asynchronous load and Preset are implemented within a single device. Figure 2.26 shows an asynchronous load with a Clear input signal. Since the Clear signal has priority over the load signal, it does not need to feed the Preset circuitry. An asynchronous load without the Clear Input Signal is shown on Figure 2.27.
5. Use One Hot Encoding of State Machines. One Hot Encoding (OHE) of states in state machines is a technique that uses one register per state and allows one

state bit to be active at any time. Although this technique increases the number of registers, it also reduces the average fan-in to the state bits. In this way, the number of LEs required to implement the state decoding logic is minimized and OHE designs run faster and use less interconnect.

6. Use Pipelining for Complex Combinatorial Logic. One of the major goals in circuit design is to maintain the clock speed at or above a certain frequency. This means that the longest delay path from the output of any register to the input(s) of the register(s) it feeds must be less than a certain value. If the delay path is too long, we recommend the pipelining of complex blocks of combinatorial logic by inserting flip-flops between them. This can increase device usage, but at the same time it lowers the propagation delay between registers and allows high system clock speeds. Pipelining is very effective especially with register intensive devices, such as FLEX 8000 devices.

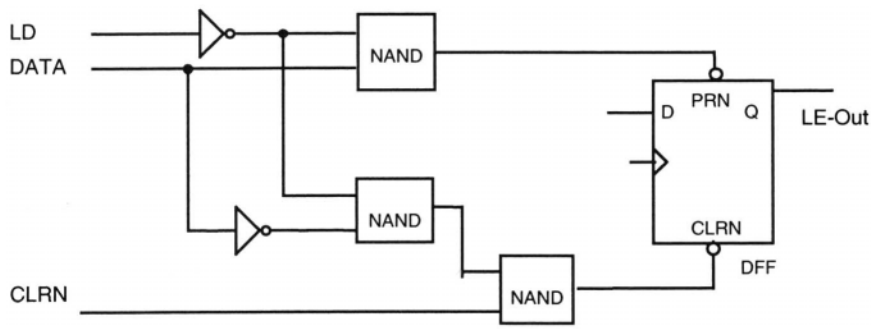



Figure 2.26 Asynchronous Load with Clear Input Signal

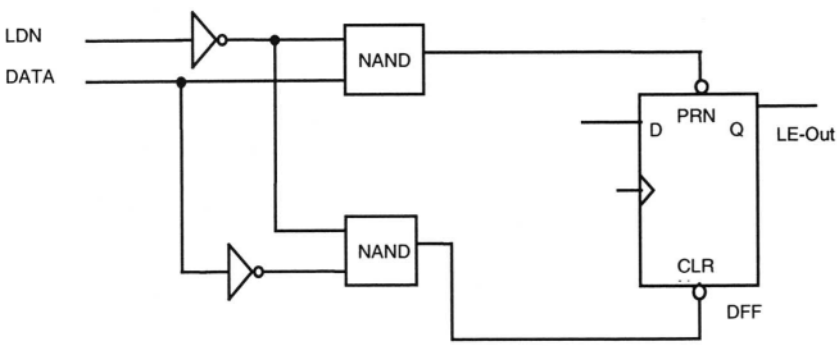



Figure 2.27 Asynchronous Load without a Clear Input Signal

An asynchronous Preset signal, which actually represents the load of a "1" into a register, is shown in Figure 2.28.

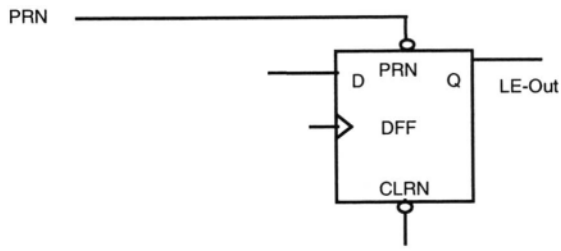



Figure 2.28 Asynchronous Preset

## 2.3 Altera FLEX 10K Devices

The aim of this Section is to make a brief introduction to basic features of Altera's FLEX 10K devices which offer quite new design alternatives and solutions to existing problems than the other CPLDs and FPGAs. Altera's FLEX 10K devices are currently industry's most complex and most advanced CPLDs. Besides logic array blocks and their logic elements, which are with the same architecture as those in FLEX8000 devices, FLEX 10K devices incorporate dedicated die areas of embedded array blocks (EABs) for implementing large specialized functions providing at the same time programmability and easy design changes. The architecture of FLEX10K device family is illustrated in Figure 2.29. The EAB consists of memory array and surrounding programmable logic which can easily be configured to implement required function. Typical functions which can be implemented in EABs are memory functions or complex logic functions, such as microcontrollers, digital signal processing functions, data-transformations functions, and wide data path functions. The LABs are used to implement general logic.

### *2.3.1 Embedded Array Block*

If the EAB is used to implement memory functions, it provides 2,048 bits, which are used to create single- or dual-port RAM, ROM or FIFO functions. When implementing logic, each EAB is equivalent to 100 to 600 gates for implementation of complex functions, such as multipliers, state machines, or DSP functions. One FLEX 10K device can contain up to 12 EABs. EABs can be used independently, or multiple EABs can be combined to implement more complex functions.

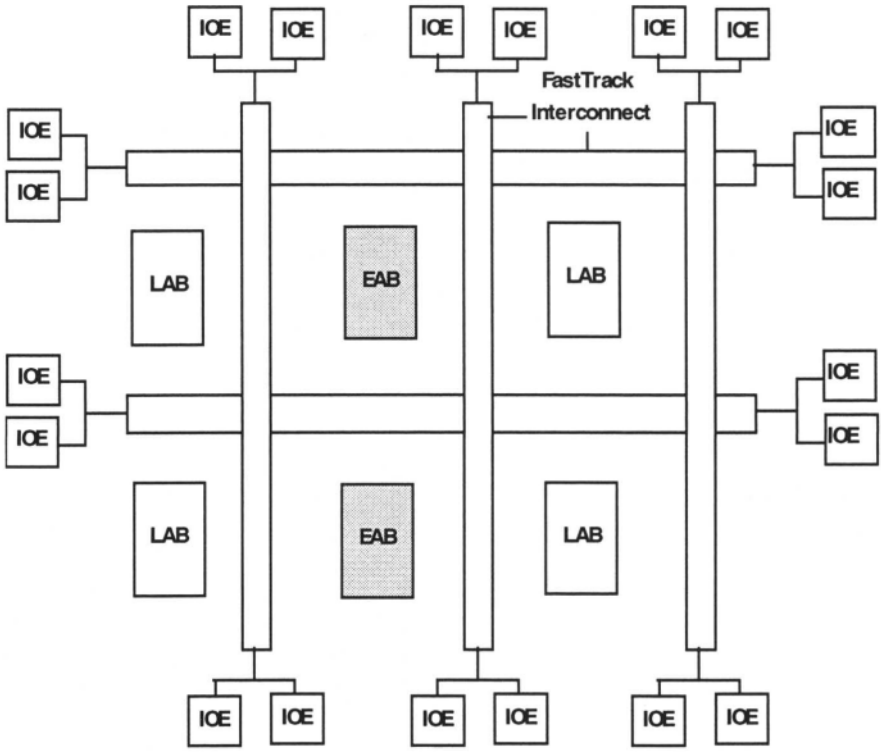



Figure 2.29 FLEX10K device architecture

The EAB is a flexible block of RAM with registers on the input and output ports. Its flexibility provides implementation of memory of the following sizes: 2,048 x 1, 1,024 x 2, 512 x 4, or 2,048 x 1 as it is shown in Figure 2.30. This flexibility makes it suitable for more than memory, for example by using words of various size as look-up tables and implementing functions such as multipliers, error correction circuits, or other complex arithmetic operations. For example, a single EAB can implement a 4 x 4 multiplier with eight inputs and eight outputs providing high performance by fast and predictable access time of the memory block. Dedicated EABs are easy to use, eliminate timing and routing concerns, and provide predictable delays. The EAB can be used to implement both synchronous and asynchronous RAM. In the case of synchronous RAM, the EAB generates its own "write enable" signal and is self-timed with respect to global clock. Larger blocks of RAM are created by combining multiple EABs in "serial" or "parallel" fashion. The global FLEX 10K signals, dedicated clock pins, and EAB local interconnect can

drive the EAB clock signals. Because the LEs drive the EAB local interconnect, they can control the "write enable" signal or the EAB clock signal. The EAB architecture is illustrated in Figure 2.31.

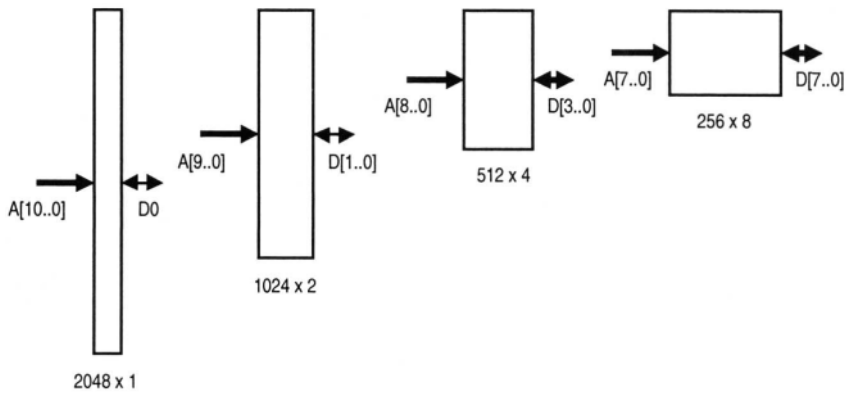



Figure 2.30 EAB Memory Configurations

In contrast to logic elements which implement very simple logic functions in a single element, and more complex functions in multi-level structures, the EAB implements complex functions, including wide fan-in functions, in a single logic level, resulting in more efficient device utilization and higher performance. The same function implemented in the EAB will often occupy less area on a device, have a shorter delay, and operate faster than functions implemented in logic elements. Depending on its configuration, an EAB can have 8 to 10 inputs and 1 to 8 outputs, all of which can be registered for pipelined designs. Maximum number of outputs depends on the number of inputs. For example, an EAB with 10 inputs can have only 1 output, and an EAB with 8 inputs can have 8 outputs.

### *2.3.2 Implementing Logic with EABs*

Logic functions are implemented by programming the EAB during configuration process with a read-only pattern, creating a large look-up table (LUT). The pattern can be changed and reconfigured during device operation to change the logic function. When a logic function is implemented in an EAB, the input data is driven

on the address input of the EAB. The result is looked up in the LUT and driven out on the output port. Using the LUT to find the result of a function is faster than using algorithms implemented in general logic and LEs.

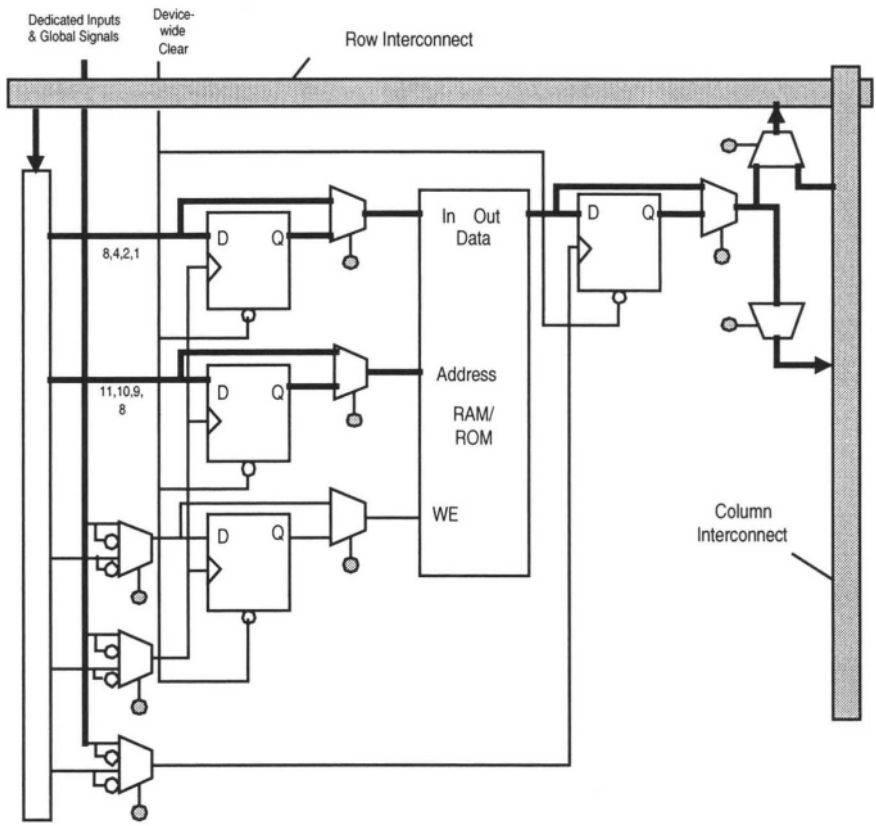



Figure 2.31 EAB architecture

EABs make FLEX 10K devices suitable for a variety of specialized logic applications such as complex multipliers, digital filters, state machines, transcendental functions, waveform generators, wide input/wide output encoders, but also various complex combinatorial functions.

For example, in a 4-bit x 4-bit multiplier, which requires two 4-bit inputs and one 8-bit output, two data inputs drive address lines of the EAB, and the output of the EAB drives out the product. The contents of the EAB memory locations is product of input data (multiplicands) presented on address lines. Higher order multipliers can be implemented using multiple 4 x 4 multipliers and parallel adders.

Another interesting application is a constant multiplier which is found often in digital signal processing and control systems. The value of constant determines the pattern that is stored in the EAB. If the constant is to be changed at run-time, it can be easily done by changing the pattern stored in the EAB. The accuracy of the result of multiplication can be adjusted by varying width of the output data bus. This can be easily done by adjusting the EAB's configuration or connecting multiple EABs in parallel if the accuracy grater than 8 bits is required.

General multipliers, constant multipliers, and adders, in addition to delay liners implemented by D-type registers, are most frequently used in various data path applications such as digital filters. The EAB, configured as a LUT, can implement a FIR filter by coefficient multiplication for all taps. The required precision on the output determines the EAB configuration used to implement the FIR filter.

Another example is implementation of transcendental function such as sine, cosine and logarithms which are difficult to compute using algorithms. It is more efficient to implement transcendental functions using LUTs. The argument of the function drives the address lines to the EAB, and the output appears at the data output lines. After implementing functions such as sine and cosine, it is easy to use them in implementing waveform generators. The EAB is used to store and generate waveforms that repeat over time. Several examples of using EABs are given in the following chapters.

Similarly, large input full encoders can be implemented using LUTs stored in EABs. Number of input address lines determines the number of combinations that can be stored in the EAB, and the number of data lines determines how many EABs are needed to implement encoder. For example, the EAB with eight address lines can store 256 different output combinations. Using two EABs connected in parallel enables encoding of input 8-bit numbers into up to 16-bit output numbers.

The contents of an EAB can be changed at any time without reconfiguring the entire FLEX 10K device. This enables the change of portion of design while the rest of device and design continues to operate. The external data source used to change the current configuration can be a RAM, ROM, or CPU. For example, while the EAB operates, a CPU can calculate a new pattern for the EAB and reconfigure the EAB at any time. The external data source then downloads the new pattern in the EAB. After this partial reconfiguration process, the EAB is ready to implement logic functions again. If we apply such design approach that some of the EABs are active and some dormant at the same time, on-the-fly reconfiguration can be

performed on the dormant EABs and they can be switched into the working system. This can be accomplished using internal multiplexers to switch-out and switch-in EABs as it is illustrated in Figure 2.32.

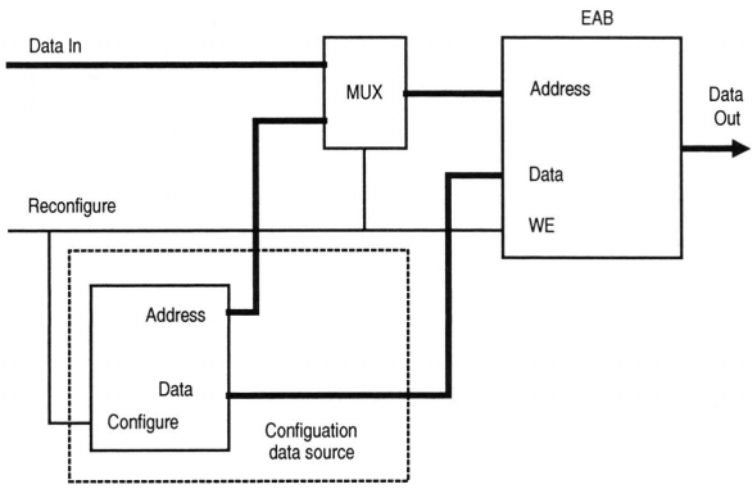



Figure 2.32 Implementation of reconfigurable logic in the EAB

If the new configuration is stored in an external RAM, it does not have to be defined in advance. It can be calculated and stored into RAM, and downloaded into the EAB when needed. For example, if the coefficients in an active filter are stored in an EAB, the characteristics of the filter can be changed dynamically by modifying the coefficients. The coefficients are modified by writing in the RAM.

## 2.4 Altera APEX 20K Devices

APEX 20K FPLDs represent the most recent development in Altera's FPLDs, combining product-term-based devices from MAX devices and LUT-based devices from FLEX devices with embedded memory blocks. As such, they enable effective integration of various types of logic into a single chip and system-on-chip designs. LUT-based logic provides efficient implementation of data-paths, register intensive operations, digital signal processing (DSP) and designs that implement algorithms involving arithmetic operations. Product-term logic efficiently implements wide fan-in combinatorial functions as well as complex finite state machines. It is implemented in embedded system blocks (ESBs). ESBs are also used to implement memory functions. As such they are suitable for multiple-input/multiple-output

look-up tables that implement logic functions, arithmetic operations and transcendental functions. They can also implement temporary storage in digital designs where they are used as ordinary read/write memory, read-only memory, and specialized memories such as dual-port memory or FIFO memory. The complexity of APEX devices ranges from typical 60,000 to more than million equivalent logic gates with different number of LUT-based logic elements, macrocells and ESBs. As an illustration, Table 2.1 shows features of a few selected APEX 20K devices. As building blocks of APEX 20K devices have many similarities with MAX and FLEX devices, in this section we will concentrate on those features that differentiate between them.

APEX 20KE devices (with suffix E) include additional features such as advanced standard I/O support, content addressable memory (CAM), additional global clocks, and enhanced ClockLock circuitry. Their capacity goes beyond one million equivalent logic gates. All APEX 20K devices are reconfigurable, and as such can be configured on board for the specific functionality, and tested before delivery. As a result, the designer only has to focus on simulation and design verification. The devices can be configured in-system via a serial data stream in passive or active configuration scheme. The external microprocessors can treat an APEX 20K device as memory and configure it by writing to a virtual memory location. Input/output (I/O) pins are fed by I/O elements (I/OEs) located at the end of each and column FastTrack interconnect, as in FLEX 8000 devices, can be used as inputs, outputs or bi-directional signals. I/Os provide features such as 3.3V operation, 64-bit 66MHz PCI compliance, JTAG boundary scan test (BST) support, slew-rate control, and tri-state buffers. Summary of APEX device features is presented in Table 2.1.

Table 2.1 APEX 20K Device Features

| Feature | 20K60E | 20K100 | 20K400 | 20K600E | 20K1000E |
|---|---|---|---|---|---|
| Maximum system gates | 162,000 | 263,000 | 1,052,000 | 1,537,000 | 1,771,520 |
| Typical gates | 60,000 | 100,000 | 400,000 | 600,000 | 1,000,000 |
| LEs | 2,560 | 4,160 | 16,640 | 24,320 | 38,400 |
| ESBs | 16 | 26 | 104 | 152 | 160 |
| Maximum macrocells | 256 | 416 | 1,664 | 2,432 | 2,560 |
| Maximum user I/O pins | 204 | 252 | 502 | 624 | 716 |

### 2.4.1 General Organization

The APEX 20K general organization block diagram is shown in Figure 2.33. They are constructed from a series of MegaLAB structures as shown in Figure 2.34. Each MegaLAB structure contains 16 logic array blocks (LABs), one ESB, and a MegaLAB interconnect, which enables routing of signals within the MegaLAB. Each LAB consists of 10 LEs and carry and cascade chains.

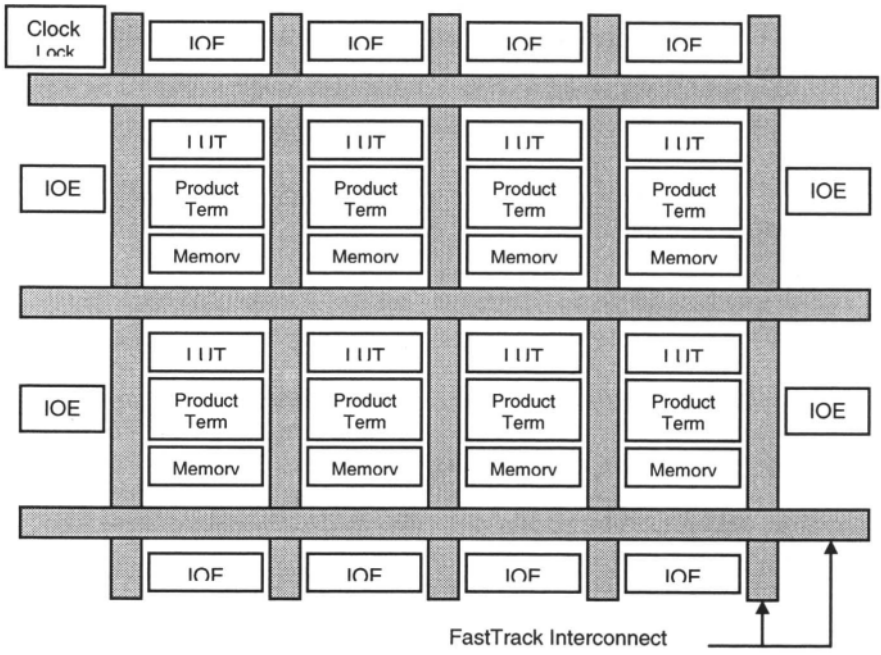



Figure 2.33 APEX 20K general organization

### 2.4.2 LUT-based Cores and Logic

The logic element (LE) is the smallest unit for logic implementation in the APEX 20K architecture. It is similar to the LEs in FLEX devices containing a four-input LUT for logic function implementation, and a number of additional control lines to specify the way it is used. The LEs architecture is presented in Figure 2.35.

The LE has two outputs that drive the local, MegaLAB, or FastTrack interconnect routing scheme. Each output can be driven independently by the LUT's or register's output. This enables better device utilization as the register and

the LUT can be used for unrelated functions. The LE can also drive out registered and unregistered versions of the LUT output.

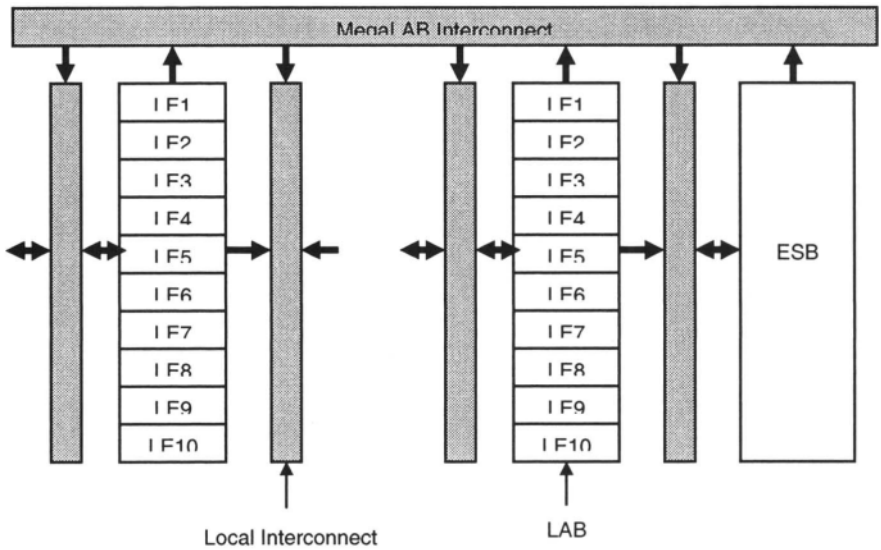



Figure 2.34 MegaLAB structure

Two types of dedicated high-speed data paths, carry chains and cascade chains, connect adjacent LEs without using the local interconnect. A carry chain supports high-speed arithmetic functions such as adders and counters, while a cascade chain implements wide-input functions such as equality comparators with minimum delay. The LE can be used in three different modes of operation: normal, arithmetic, and counter mode.

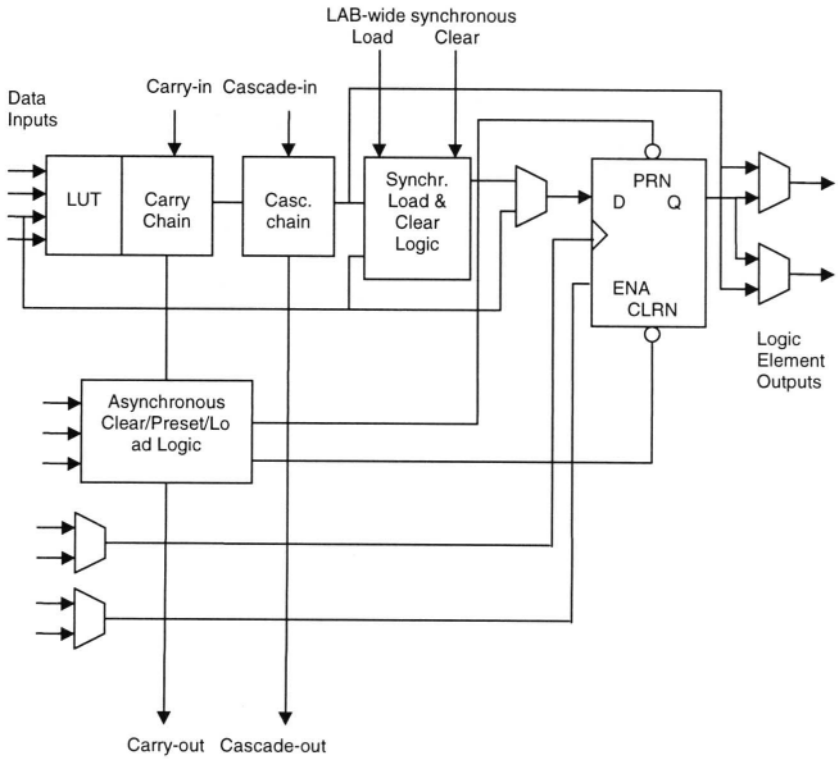



Figure 2.35 APEX 20K logic element

### *2.4.3 Product-Term Cores and Logic*

The product-term portion of the APEX 20K architecture is implemented with ESBs. Each ESB can be configured to act as a block of macrocells, as shown in Figure 2.36, that are used for the implementation of logic. In product-term mode each ESB contains 16 macrocells. It is fed by 32 inputs from the adjacent local interconnect. Also, nine ESB macrocells feed back into the ESB through the local interconnect for higher performance.

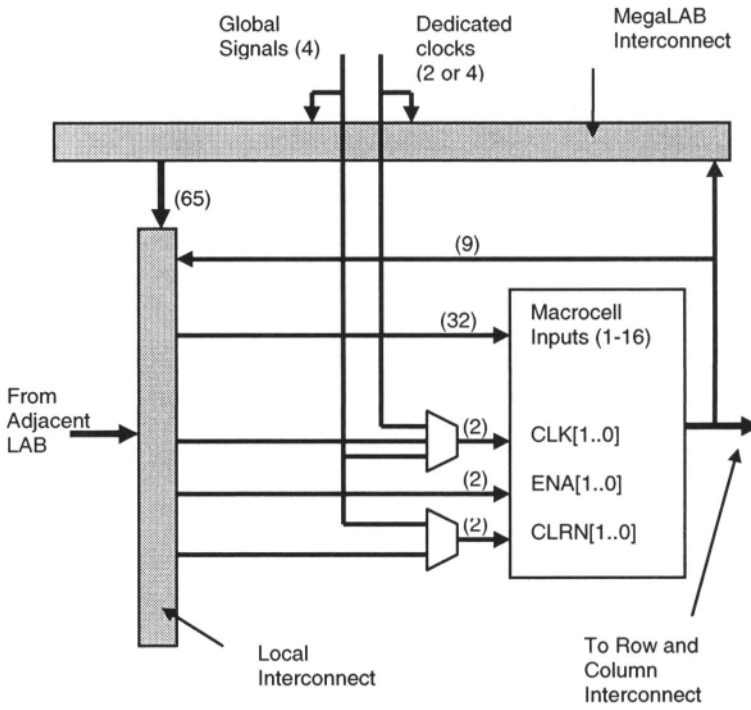



Figure 2.36 13 ESB used for product-term logic

The macrocells can be configured individually for either sequential or combinational logic operation. The APEX 20K macrocell consists of three functional parts: the logic array, the product-term select matrix, and the programmable register, similar to MAX devices, as shown in Figure 2.37.

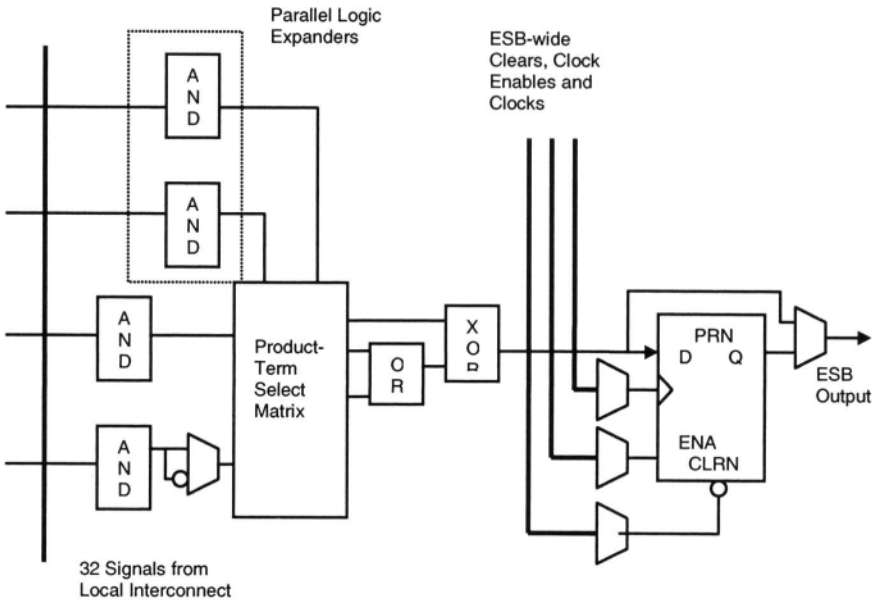



Figure 2.37 14 APEX 20K macrocell

Parallel expanders are unused product terms that can be allocated to a neighboring macrocell to implement fast, complex logic functions. Parallel expanders allow up to 32 product terms to feed the macrocell OR logic directly, with two product terms provided by the macrocell and 30 parallel expanders provided by the neighboring macrocells in the ESB. An illustration of the use of parallel expanders is given in Figure 2.38.

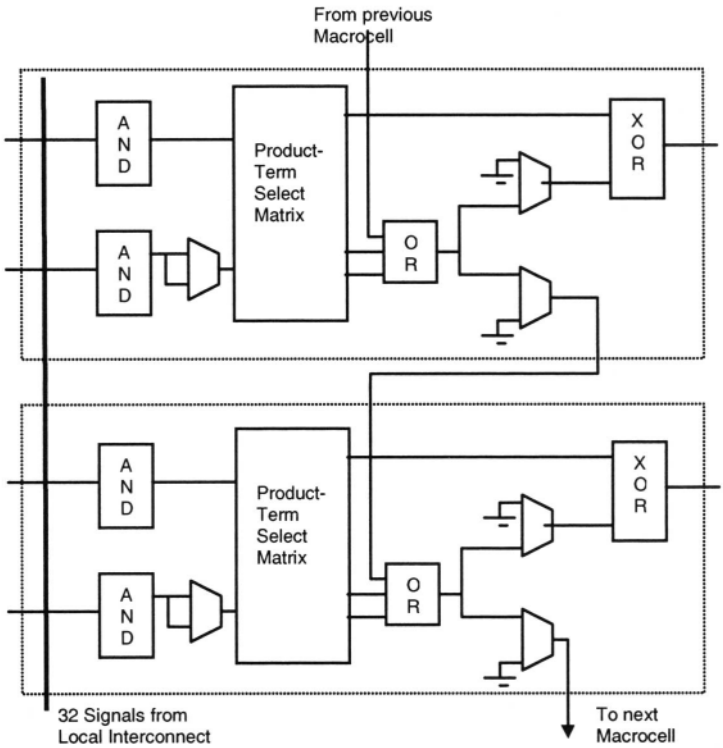



Figure 2.38 Using parallel expanders

### *2.4.4 Memory Functions*

The ESB can implement various memory functions, such as single-port and dual-port RAM, ROM, FIFO and content-addressable memory (CAM). Its capacity is 2,048 bits. The ESB has input registers to synchronize write operations, and output registers to enable pipelined designs. The dual-port mode supports simultaneous reads and writes at two different clock frequencies. The general block diagram representing the ESB block is shown in Figure 2.39.

When implementing memory, each ESB can be configured in any of the following memory configurations: 128×16, 256×8, 512×4, 1,024×2 and 2,048×1.

Larger memory blocks can be implemented by combining multiple ESBs in serial or parallel configurations. Memory performance does not degrade for memory blocks deep up to 2,048 words. ESBs ca be used in parallel, eliminating the need for any external decoding logic and its associated delays. To create a high-speed memory of capacity larger than 2,048 locations, EABs drive tri-state lines that connect all EABs in a column of MegaLAB structures. Each ESB incorporates a programmable decoder to activate tri-state driver as required. For example, a 8K memory block can be formed by serially connecting four 2K memory blocks as shown in Figure 2.40. Eleven address lines are used by each ESB, and two additional address lines drive the tri-state decoder. The internal tri-state logic is designed to avoid internal contention and floating lines.

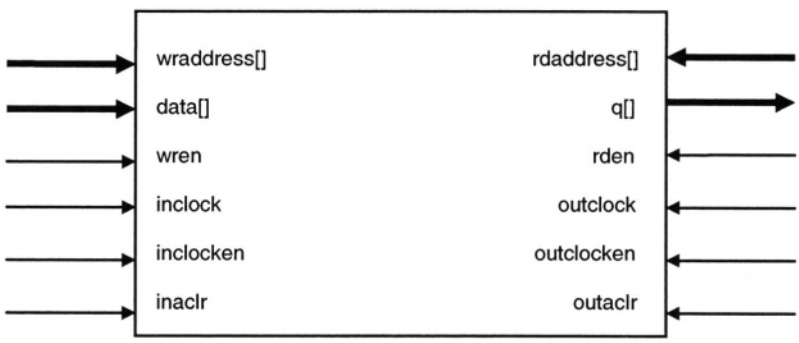



Figure 2.39 ESB ports

The ESB can be used to implement dual-port RAM applications where both ports can read or write, as shown in Figure 2.39. It implements two forms of dual-port memory:

- read/write clock mode memory. Two clocks are required. One clock controls all registers associated with writing, while the other clock controls all registers associated with reading. Also, input and output registers enable and asynchronous clear signals are supported. This mode is commonly used for applications where read and write occur at different clock frequencies.

- input/output clock mode memory. Similar set of clock and control lines is provided. This mode is commonly used for applications where reads and writes occur at the same clock frequency, but require different clock enable signals for the input and output registers.

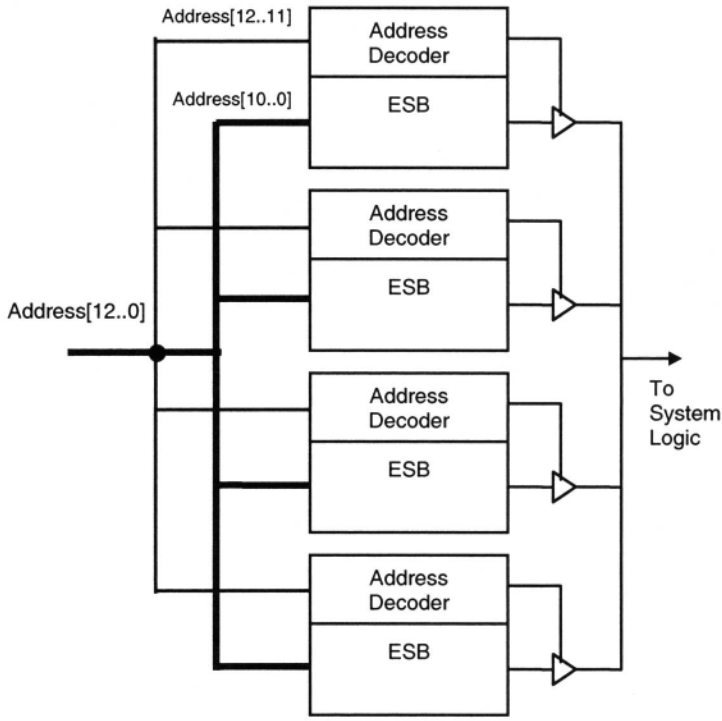



Figure 2.40 Using ESBs to implement larger memory blocks

In case of using ESB for bi-directional dual-port memory applications, when both ports can be read or written simultaneously, two EABs must be used to support simultaneous accesses.

The ESB can implement content-addressable memory (CAM) that can be considered as the inverse of RAM and illustrated in Figure 2.41. When read, CAM outputs an address for given data word. This sort of operation is very suitable for high-speed search algorithms often found in networking, communications, data compression, and cache management applications. CAM searches all addresses in parallel and outputs the address storing a particular data word. When a match is found, a match-found flag is set high.

When in CAM mode, the ESB implements a 32 word, 32-bit CAM. Wider or deeper CAMs can be implemented by combining multiple CAMs with additional

logic implemented in LEs. CAM supports writing "don't-care" bits into words of the memory and they can be used as a mask for CAM comparisons; any bit set to don't-care has no effect on matches. The output of the CAM can be encoded or unencoded. If the output is unencoded, two clock cycles are required to show the status of 32 words, because a 16-bit output bus can show only 16 status lines at the same time. In the case of encoded output, encoded address is output in a single clock cycle. If duplicate data is written into two locations, the CAM output will not be correct when using the encoded output. If the CAM contains duplicate data, the unencoded output is better solution, because all CAM locations containing duplicate data will be indicated. CAM can be pre-loaded with data during FPLD configuration, or it can be written during system operation. In most cases two clock cycles are required to write each word into CAM, and when don't-care bits are used an additional clock cycle is required.

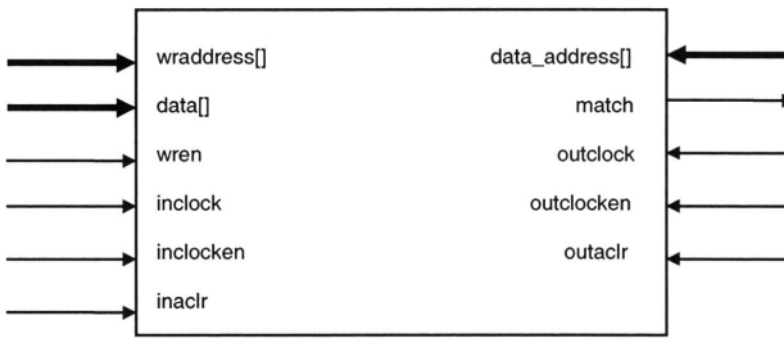



Figure 2.41 ESB in CAM mode

ESBs provide a number of options for driving control signals. Different clocks can be used for the ESB inputs and outputs. Registers can be inserted independently on the data input, data output, read address, write address, WE and RE signals. The global signals and the local interconnect can drive the WE and RE signals. The global signals, dedicated clock pins, and local interconnect can drive the ESB clock signals. As the LEs drive the local interconnect, they can control the WE and RE signals and the ESB clock, clock enable, and asynchronous clear signals. The ESB, on the other hand, can drive the local, MegaLAB, or FastTrack interconnect routing structure to drive LEs and IOEs in the same MegaLAB or anywhere in the device.

## 2.5 Xilinx XC4000 FPGAs

The Xilinx XC4000 family of FPGAs provides a regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs) interconnected by a hierarchy of versatile routing resources and surrounded by a perimeter of programmable Input/Output Blocks (IOBs). The devices are customized by loading configuration data into the internal static memory cells (SRAMs).

The basic building blocks used in the Xilinx XC4000 family include:

- Look-up tables for implementation of logic functions. A designer can use a function generator to implement any Boolean function of a given number of inputs by preloading the memory with the bit pattern corresponding to the truth table of the function. All functions of a function generator have the timing: the time to look up results in the memory. Therefore, the inputs to the function generator are fully interchangeable by simple rearrangement of the bits in the look-up table.
- A Programmable Interconnect Point (PIP) is a pass transistor controlled by a memory cell. The PIP is the basic unit of configurable interconnect mechanisms. The wire segments on each side of the transistor are connected depending on the value in the memory cell. The pass transistor introduces resistance into the interconnect paths and hence delay.

A multiplexer is a special case one-directional routing structure controlled by a memory cell. Multiplexers can be of any width, with more configuration bits (memory cells) for wider multiplexers.

The FPGA can either actively read its configuration data out of external serial or byte parallel PROM (master modes) or the configuration data can be written into the FPGA (slave and peripheral modes). FPGAs can be reprogrammed an unlimited number of times allowing the design to change and allowing the hardware to adapt to different user applications.

CLBs provide functional elements for constructing user's logic. IOBs provide the interface between the package pins and internal signal lines. The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs to the appropriate networks. Customized configuration is provided by programming internal static memory cells that determine the logic functions and interconnections in the Logic Cell Array (LCA) device. The Xilinx family of FPGAs consists of different circuits with different complexities. Here we present the most advanced type, the Xilinx XC4000. The XC4000 can be used in designs

where hardware is changed dynamically, or where hardware must be adapted to different user applications.

### *2.3.1 Configurable Logic Block*

The CLB architecture, shown in Figure 2.42, contains a pair of flip-flops and two independent 4-input function generators. Four independent inputs are provided to each of two function generators which can implement any arbitrarily defined Boolean function of their four inputs. Function generators are labeled F and G.

Function generators are implemented as memory look-up tables (LTUs). A third function generator (labeled H) can implement any Boolean function of its three inputs, two of them being outputs from F and G function generators, and the third input from outside of the CLB. Outputs from function generators are available at the output of the CLB enabling the generation of different combinations of four or five variables Boolean functions. They can even be used to implement some nine variable Boolean functions such as nine-input AND, OR, XOR (parity) or decode in one CLB. The CLB contains two edge-triggered D-type flip-flops with common clock (K) and clock enable (EC) inputs. A Third common input (S/R) can be programmed as either an asynchronous set or reset signal independently for each of the two registers This input can be disabled for either flip-flop. A separate global Set/Reset line (not shown in Figure) is provided to set or reset each register during power up, reconfiguration, or when a dedicated Reset network is driven active. The source of the flip-flop data input can be functions F', G', and H', or the direct input (DIN). The flip-flops drive the XQ and YQ CLB outputs.

Each CLB includes high speed carry logic that can be activated by configuration. As shown in Figure 2.43, two 4-input function generators can be configured as 2-bit adder with built in hidden carry circuitry that is so fast and efficient that conventional speed up methods are meaningless even at the 16-bit level. The fast carry logic opens the door to many new applications involving arithmetic operations, (such as address offset calculations in microprocessors and graphics systems or high speed addition in digital signal processing).

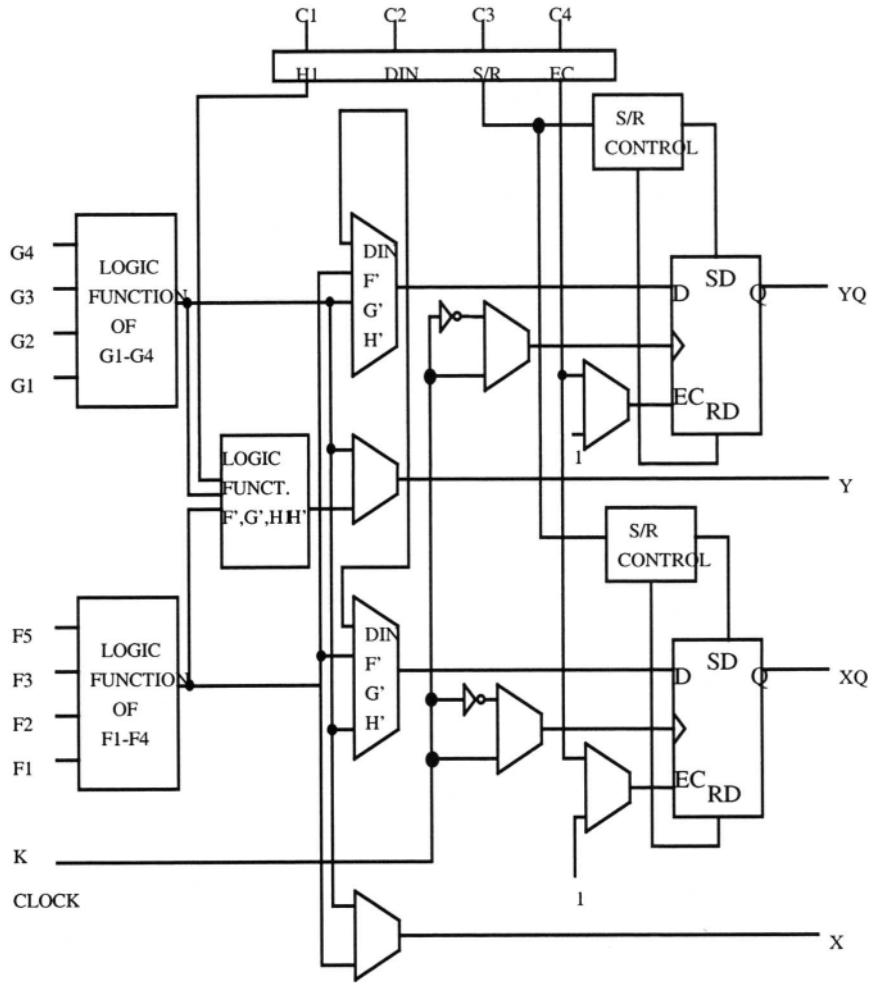



Figure 2.42 CLB Architecture

The Xilinx XC4000 family LCAs include on-chip static memory resources. An optional mode for each CLB makes the memory look-up tables in the function generators usable as either a 16 × 2 or 32 × 1 bit array of Read/Write memory cells, as shown in Figure 2.44. The function generator inputs are used as address bits and additional inputs to the CLB for Write, Enable, and Data-In. Reading memory is the same as using it to implement a function.

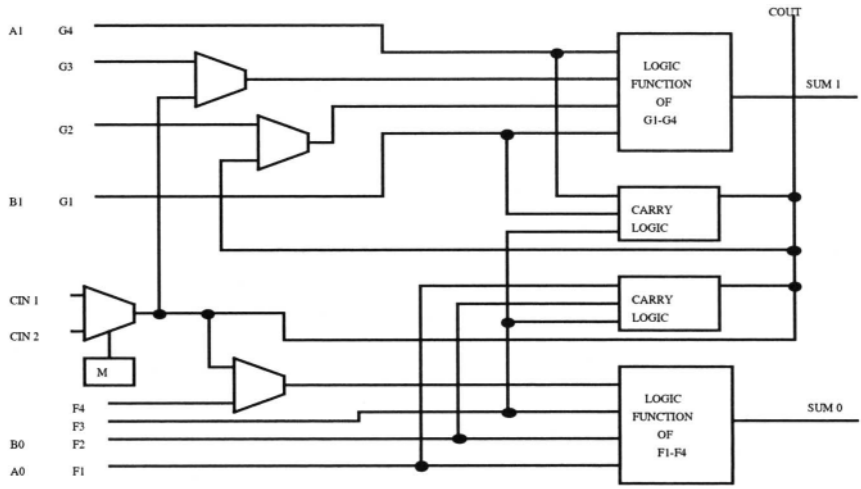



Figure 2.43 Fast Carry Logic in CLB

The F1-F4 and G1-G4 inputs act as address lines selecting a particular memory cell in each LUT. The functionality of CLB control signals change in this configuration. The H1, DIN, and S/R lines become the two data inputs and Write enable (WE) input for 16 × 2 memory. When the 32 x 1 configuration is selected, D1 acts as the fifth address bit and D0 is the data input. The contents of the memory cell being addressed is available at F' and G' function generator outputs. They can exit through X and Y CLB outputs or can be pipelined using the CLB flip-flops. Configuring the CLB function generators as R/W memory does not affect functionality of the other portions of the CLB, with the exception of the redefinition of control signals.

The RAMs are very fast with read access time being about 5 ns and write time about 6 ns. Both are several times faster than off chip solutions. This opens new possibilities in system design such as registered arrays of multiple accumulators, status registers, DMA counters, LIFO stacks, FIFO buffers, and others.

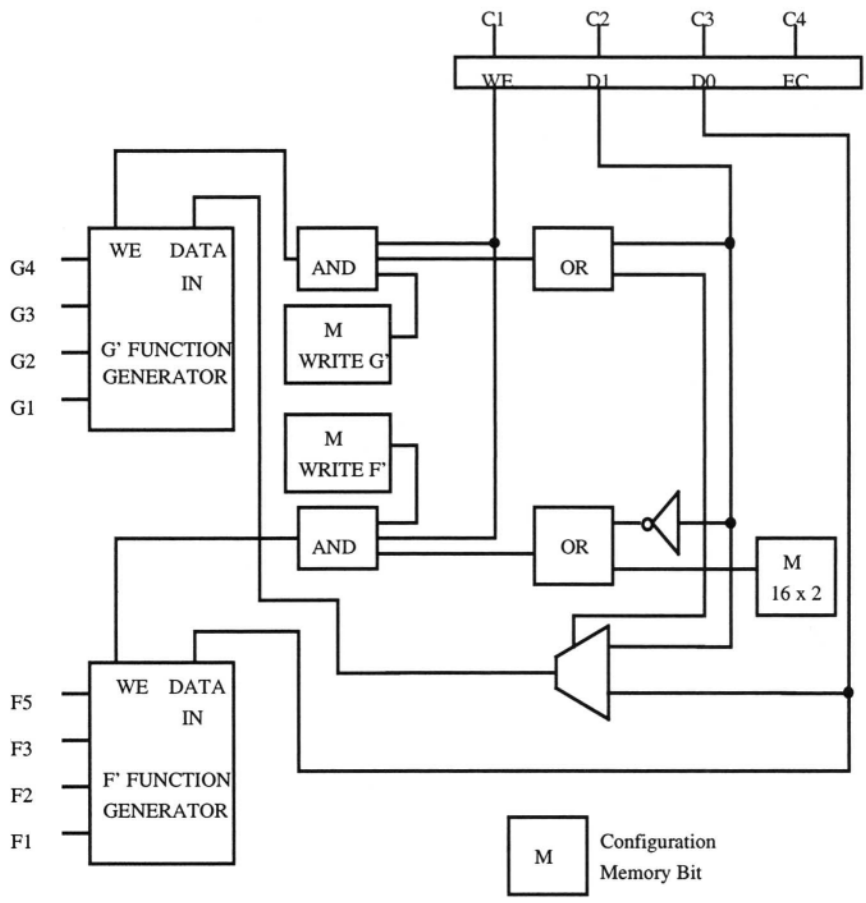



Figure 2.44 Usage of CLB function generators as Read/Write memory cells

### *2.5.2 Input/Output Blocks*

User programmable Input/Output Blocks (IOBs) provide the interface between internal logic and external package pins as shown in Figure 2.45. Each IOB controls one package pin. Two lines, labeled I1 and I2 bring input signals into the array. Inputs are routed to an input register that can be programmed as either an edge triggered flip-flop or a level sensitive transparent latch. Each I1 and I2 signals can carry either a direct or registered input signal. By allowing both, the IOB can de-multiplex external signals such as address/data buses, store the address in the flip-

flop, and feed the data directly into the wiring. To further facilitate bus interfaces, inputs can drive wide decoders built into the wiring for fast recognition of addresses. Output signals can be inverted or not inverted and can pass directly to the pad or be stored in an edge triggered flip-flop. Optionally, an output enable signal can be used to place the output buffer in a high impedance state, implementing s-state outputs or bi-directional I/O.

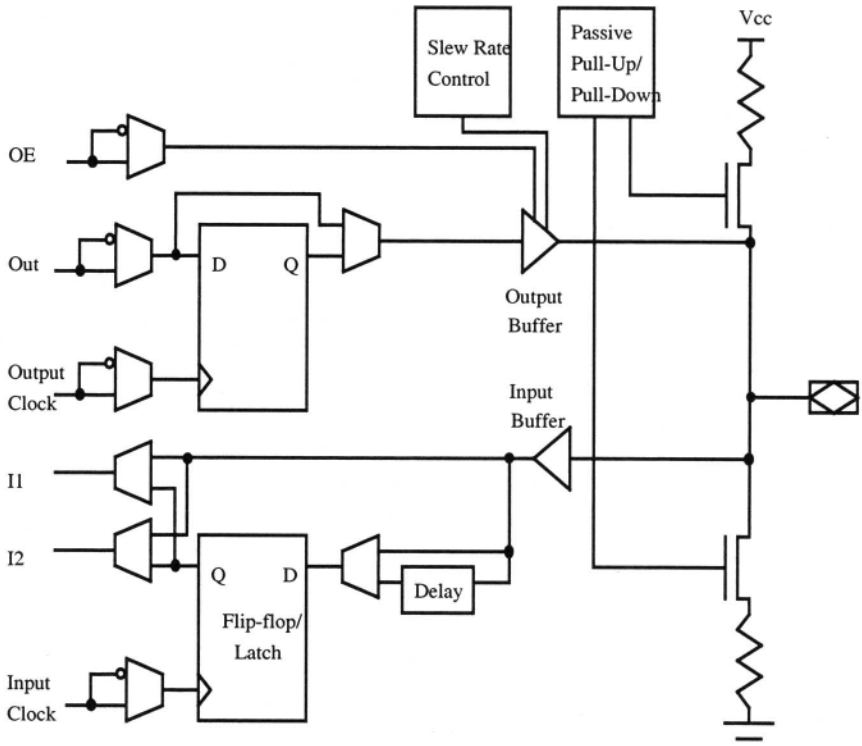



Figure 2.45 IOB architecture

There are a number of other programmable options in the IOB such as programmable pull-up and pull-down resistors, separate input and output clock signals, and global Set/Reset signals as in the case of the CLB.

### *2.5.3 Programmable Interconnection Mechanism*

All internal connections are composed of metal segments with programmable switching points to implement the desired routing. The number of the routing channels is scaled to the size of the array and increases with array size. CLB inputs and outputs are distributed on all four sides of the block as shown in Figure 2.46.

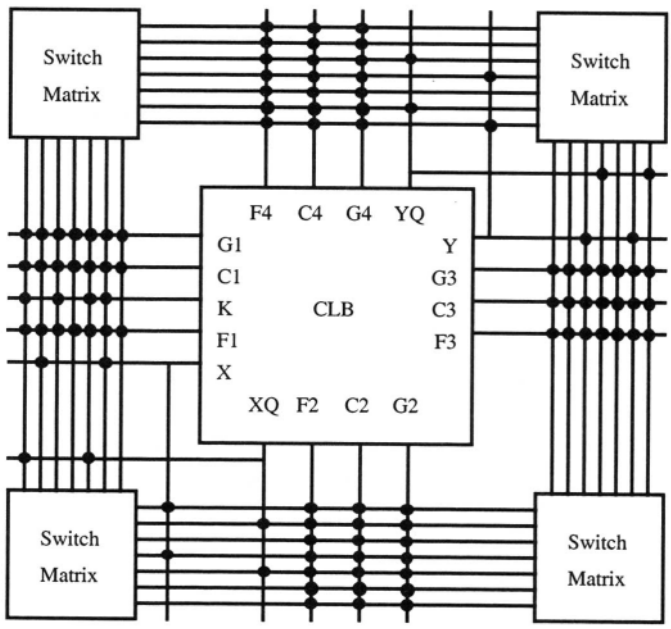



Figure 2.46 Typical CLB connection to adjacent single length lines

There are three types of interconnects distinguished by the relative length of their segments: single length lines, double length lines, and longlines.

The single length lines are a grid of horizontal and vertical lines that intersect at a Switch Matrix between each block. Each Switch Matrix consists of programmable n-channel pass transistors used to establish connections between the single length lines as shown in Figure 2.47. For example, a signal entering on the right side of the Switch matrix can be routed to a single length line on the top, left, or bottom sides, or any combination if multiple branches are required. Single length lines are normally used to conduct signals within localized areas and to provide branching for nets with fanout greater than one.

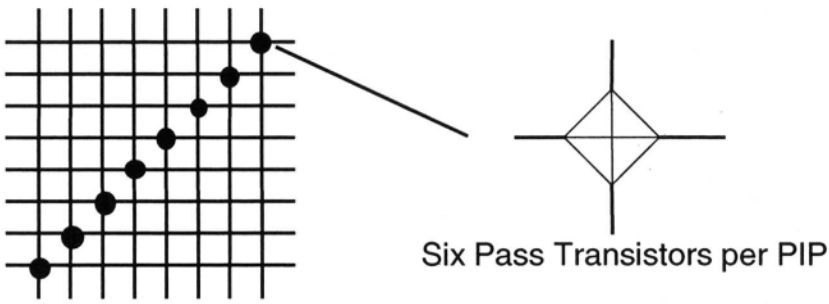



Figure 2.47 Switch Matrix

The double length lines, as shown in Figure 2.48 consist of a grid of metal segments twice as long as the single length lines. They are grouped in pairs with the Switch Matrix staggered so each line goes through the Switch Matrix at every other CLB location in that row or column. Longlines form a grid of metal interconnect segments that run the entire length or width of the array (Figure 2.49). Additional long lines can be driven by special global buffers designed to distribute clocks and other high fanout control signals throughout the array with minimal skew. Six of the longlines in each channel are general purpose for high fanout, high speed wiring. CLB inputs can be driven from a subset of the adjacent longlines. CLB outputs are routed to the longlines by way of 3-state buffers or the single interconnect length lines. Communication between longlines and single length lines is controlled by programmable interconnect points at the line intersections, while double length lines can not be connected to the other lines.

A pair of 3-state buffers, associated with each CLB in the array, can be used to drive signals onto the nearest horizontal longlines above and below of the block. The 3-state buffer input can be driven from any X, Y, XQ, or YQ output of the neighboring CLB, or from nearby single length lines with the buffer enable coming from nearby vertical single length lines or longlines. Another 3-state buffer is located near each IOB along the right and left edges of the array. These buffers can be used to implement multiplexed or bi-directional buses on the horizontal longlines. Programmable pull-up resistors attached to both ends of these longlines help to implement a wide wired-AND function.

The XC4000 family has members with different amounts of wiring for different size ranges. The amount of wire and distribution among different wire lengths is dictated by routability requirements of the FPGAs in the target size range. For the CLB array from 14 × 14 to 20 × 20, each wiring channel includes eight single length

lines, four double length lines, six longlines and four global lines. The distribution was derived from an analysis of wiring needs of a large number of existing designs.

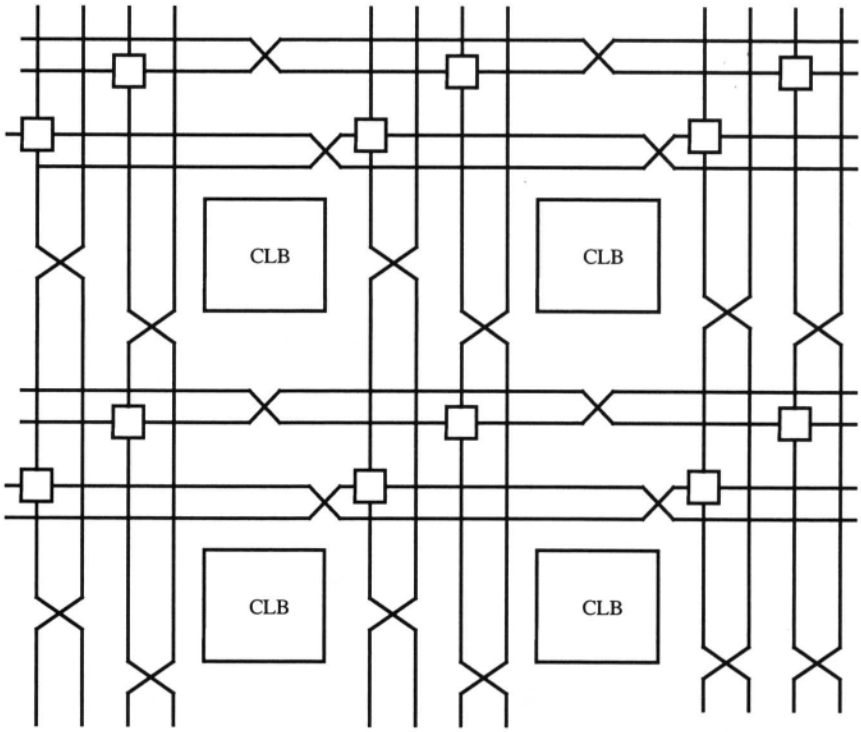



Figure 2.48 Double length lines

All members of the Xilinx family of LCA devices allow reconfiguration to change logic functions while resident in the system. Hardware can be changed as easily as software. Even dynamic reconfiguration is possible, enabling different functions at different times.

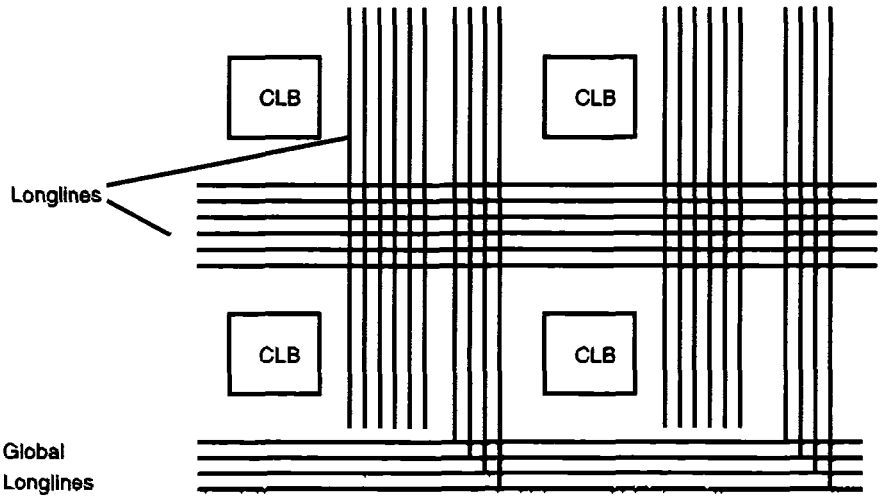



Figure 2.49 Longlines

### *2.5.4 Device Configuration*

Configuration is a process of loading design specific programming data into LCA devices to define the functional operation and the interconnections of internal blocks. This is, to some extent, similar to loading the control registers of a programmable chip. The XC4000 uses about 350 bits of configuration data per CLB and its associated interconnections. Each bit defines the state of a SRAM cell that controls either look-up table bit, multiplexer input, or interconnection pass transistor.

The XC4000 has six configuration modes selected by a 3-bit input code. They are similar to the modes in Altera's family: three are self-loading Master modes, two Peripheral modes, and one is a Serial Slave mode.

The master modes use an internal oscillator to generate the clock for driving potential slave devices and to generate address and timing for external PROM(s) containing configuration data. Data can be loaded either from parallel or serial PROM. In the former case, data is internally serialized into the appropriate format.

Peripheral modes accept byte wide data from a bus. A READY/BUSY status is available as a handshake signal. An externally supplied clock serializes the data. In

the Serial Slave mode, the device receives serial configuration data from external source.

LCAs can configure themselves when they sense power up or they can be reconfigured on command while residing in the circuit. A designer can create a system in which the FPGA's program changes during operation.

The LCA can also read back its programming along with the contents of internal flip-flops, latches, and memories. A working part can be stopped and its state recovered. The read-back facility is especially valuable during verification and debugging of prototypes and is also used in manufacturing test.

### *2.3.5 Designing with XC4000 Devices*

As with other FPGAs families, Xilinx FPGAs allow the use of existing design tools for logic or ASIC systems, including schematic entry and hardware description languages. To target an FPGA, a design is passed to FPGA specific implementation software. The interface between design entry and design implementation is a netlist that contains the desired nets, gates, and reference to hard macros.

Although many designs are still done manually, because of the special density and performance requirements, manual designs can be combined with automatic design procedures and can be done completely automatically. Automatic design implementation, the most common method of implementing logic on FPGAs, consists of three major steps: partitioning, placement, and routing.

Partitioning is the separation of the logic into CLBs. It has both a logical and physical component. The connections within a CLB are constrained by the limited intra-block paths and by the limited number of block outputs. The quality of partitioning depends on how well the subsequent placement can be done, so physically related logic should be partitioned into the same block. Placement starts with CLBs, IOBs, hard macros, and other structures in the partitioned netlist. A decision is then made as to which corresponding blocks on the chip should contain those structures. Routing is not as flexible as mask programmed gate arrays. FPGA routing shows very little connectivity between vertical and horizontal segments, requiring many constraints to be taken into account including those for the optimization of the length of nets as well as their delays.

Interactive tools allow constraints on the already known automated algorithms used for MPGAs, postroute improvements on the design, and quick design iterations. The manual editing capability allows users to modify the configuration of any CLB or routing path. In support of an iterative design methodology, Xilinx's

automatic place and route system has built-in incremental design facilities. Small changes in a design are incorporated without changing unaffected parts of a design. Large, complex CLBs facilitate incremental changes because a small change can more easily be isolated to a change in a single CLB or a single new routing connection. The incremental change may take only a few minutes, where the original placement and routing may take hours.

## 2.6 Xilinx Virtex FPGAs

This section briefly presents Virtex FPGAs, that represent the most recent and advanced Xilinx FPGAs implemented in 0.22 μm CMOS process. They include an array of configurable logic blocks (CLBs) surrounded by input/output blocks (IOBs) and interconnected by a hierarchy of fast, versatile routing resources. The configuration data is loaded into internal SRAM memory cells either from an external PROM (active configuration) or it is written into the FPGA using passive configuration schemes. Virtex devices accommodate large designs with clock rates up to 200MHz. Many designs operate internally at speeds in excess of 100MHz. Virtex family consists of a number of devices with different capacities as shown in Table 2.2.

Table 2.2 Virtex FPGA Features

| Feature | XCV50 | XCV100 | XCV400 | XCV600 | XCV1000 |
|---|---|---|---|---|---|
| System gates | 57,906 | 108,904 | 468,252 | 661,111 | 1,124,022 |
| CLB array | 16x24 | 20x30 | 40x60 | 48x72 | 64x96 |
| Logic cells | 1,728 | 2,700 | 10,800 | 15,552 | 27,648 |
| Maximum available I/O | 180 | 180 | 404 | 512 | 512 |
| BlockRAM Bits | 32,768 | 40,960 | 81,920 | 98,304 | 131,072 |
| Max select RAM bits | 24,576 | 38,400 | 153,600 | 221,184 | 393,216 |

### *2.6.1 General Organization*

The Virtex FPGA includes two major building elements:

- CLBs, which provide the functional elements to implement logic.
- IOBs, which provide the interface between the device pins and CLBs.

CLBs interconnect through a general routing matrix (GRM) that comprises an array of routing switches located at the intersections of horizontal and vertical channels. The local routing resources (VersaBlock) are provided for connecting CLBs to GRMs.

The Virtex architecture also includes additional building blocks for digital systems implementation:

- Dedicated block memories (Block RAM) of 4096 bits each
- Clock delay-locked loops (DLLs) for clock-distribution delay compensation and clock domain control
- 3-state buffers associated with each CLB that drive dedicated horizontal routing resources

### *2.6.2 Configurable Logic Block*

The basic building block for implementation of logic within CLB is the logic cell (LC). An LC includes 4-input function generator, implemented as 4-bit look-up tables (LUTs), carry logic and a storage element. Each Virtex CLB contains four LCs organized in two almost identical slices. The organization of a single slice is illustrated in Figure 2.50. In addition, each CLB contains logic that combines function generators to provide logic functions of five and six input variables.

The two LUTs within a slice can be combined to create a 16x2 or 32x1 synchronous RAM, or a 16x1 dual-port RAM. The LUT can also be used as a 16-bit shift register. The storage elements in the Virtex slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D inputs can be driven either by the function generators within the slice or directly from slice inputs, bypassing the function generators.

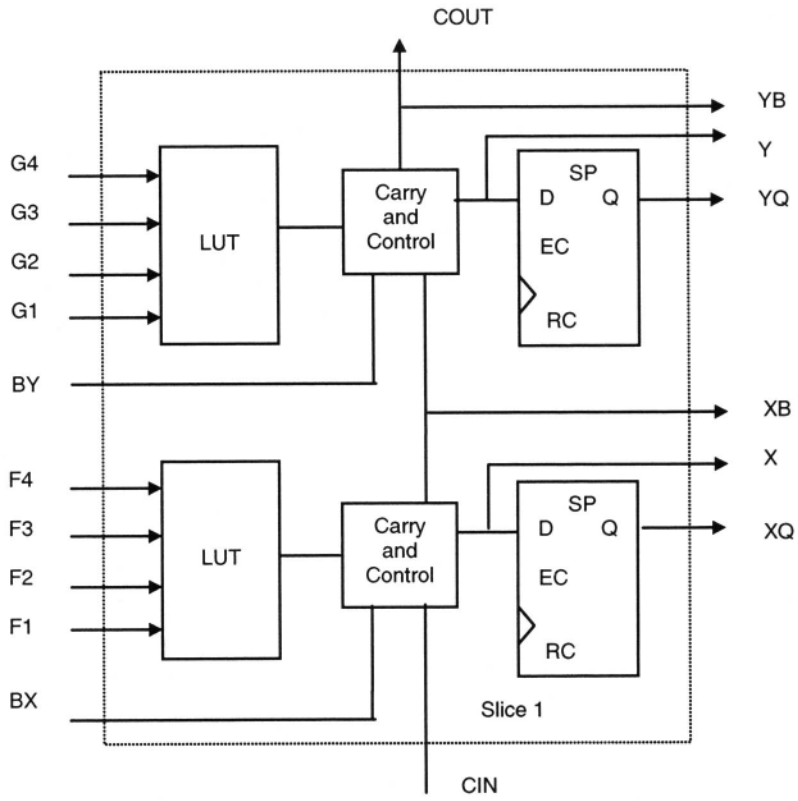



Figure 2.50 CLB organization

Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions. The Virtex CLB supports two separate carry chains, one per a Slice. The height of the carry chains is two bits per CLB. The arithmetic logic includes a XOR gate that allows a 1-bit full adder to be implemented within a LC. In addition, a dedicated AND gate improves the efficiency of multiplier implementation. The dedicated carry path can also be used to cascade function generators for implementing wide logic functions.

### *2.6.3 Input/Output Block*

The Virtex Input/Output Blocks (IOBs) support a variety of I/O signaling standards. The organization of the IOB is presented in Figure 2.51. They contain three storage elements that can be used as either D-type flip-flops or as level sensitive latches with individual configuration capabilities. Input and output paths are controlled by separate buffers with separate enable signals. The output is driven by a 3-state buffer either from internal logic (CLB) or from the IOB flip-flop..

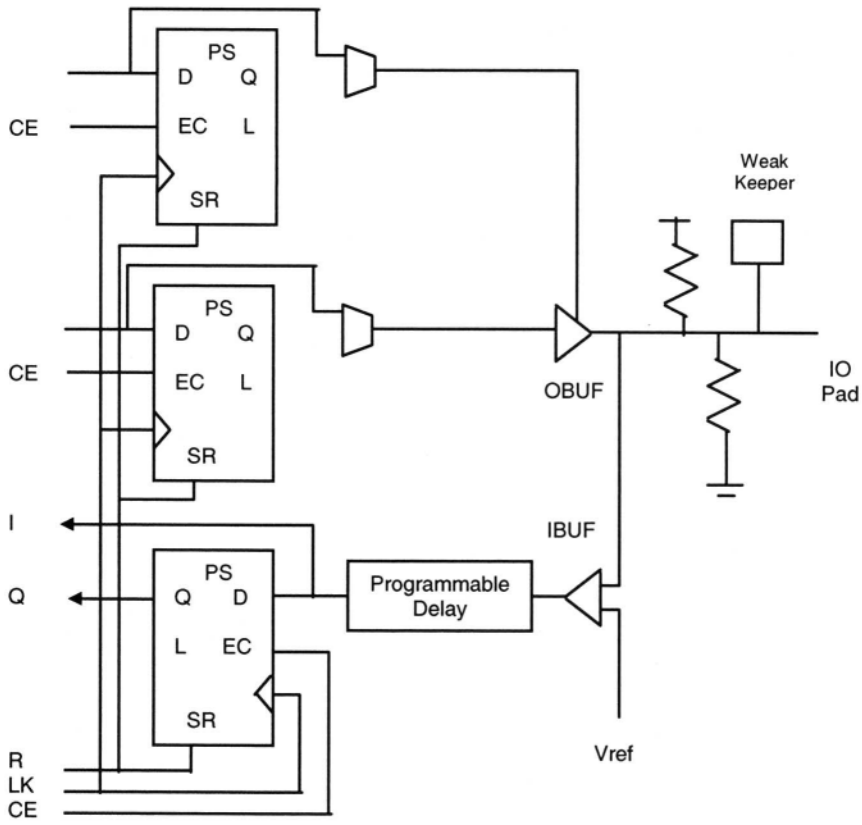



Figure 2.51 Virtex Input/Output Block (IOB)

### *2.6.4 Memory Function*

SRAM memory blocks, called SelectRAMs, are organized in columns. All Virtex devices contain two such columns, one along each vertical edge. Each memory block is four CLBs high, and consequently, a Virtex device 64 CLBs high contains 16 memory blocks per column, and a total of 32 blocks. Each Block SelectRAM cell, as illustrated in Figure 2.52, is a fully synchronous dual-ported 4096-bit RAM with independent control signals for each port. The data widths of the two ports can be configured independently, providing built-in bus-width conversion and configuration of the blocks as the application requires. Configurations of the RAM blocks can be 256×16, 512×8, 1,024×4, 2,048×2 and 4,096×1.

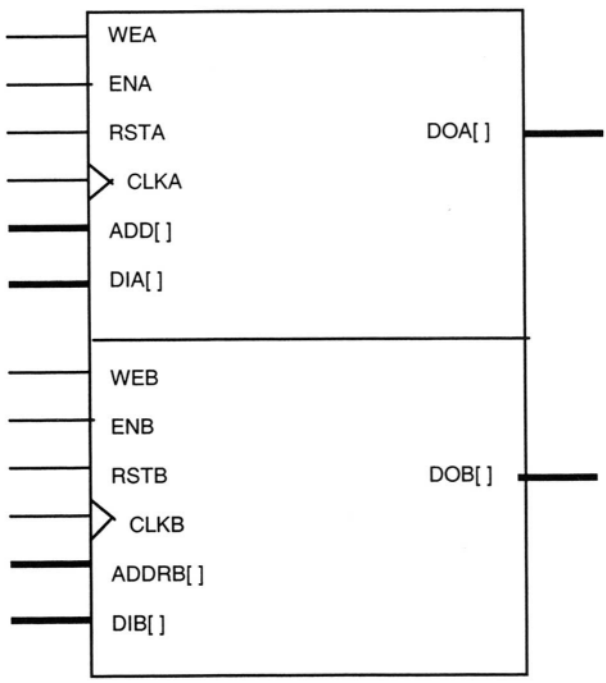



Figure 2.52 Dual-port SelectRAM

## 2.7 Atmel AT40K Family

The AT40K is a family of SRAM-based FPGAs with distributed 10ns programmable synchronous/asynchronous, dual port/single port SRAM and dynamic full or partial reconfigurability. The devices range in size from 5,000 to 50,000 equivalent logic gates. They support 3V and 5V designs. The AT40K is designed to quickly implement high performance, large gate count designs through the use of synthesis and schematic-based tools. Some of the unique features are system speeds to 100MHz, array multipliers faster than 50MHz, high-speed flexible SRAM, and internal 3-state capability in each logic cell.

The AT40K device can be used as a coprocessor for high speed (DSP/Processor-based) designs by implementing a variety of compute-intensive, arithmetic functions. These include adaptive finite impulse response (FIR) filters, fast Fourier transforms (FFT), convolvers, interpolators and discrete-cosine transforms (DCT) that are required for video compression and decompression, encryption, convolution and other multimedia applications. Table 2.3 presents the features of some of the AT40K family devices. As we can notice, those devices have lower capacity than corresponding Altera and Xilinx devices, but also some features not found in those two families.

Table 2.3 Atmel AT40K features

| Features | AT40K05 | AT40K10 | AT40K20 | AT40K40 |
|---|---|---|---|---|
| Equivalent Gates | 5K-10K | 10K-20K | 20K-30K | 40K-50K |
| Rows x Columns | 16x16 | 24x24 | 32x32 | 48x48 |
| Logic cells | 256 | 576 | 1,024 | 2,304 |
| RAM bits | 2,048 | 4,608 | 8,192 | 18,432 |
| Max I/O pins | 128 | 192 | 256 | 384 |

### *2.7.1 General Organization*

The AT40K FPGAs are organized in a symmetrical array (matrix) of identical cells as illustrated in Figure 2.53. The array is continuous from one edge to the other, except for bus repeaters spaced every four cells that divide up the busing resources on the device into 4×4 cell areas, referred to as sectors. At the lower right corner of each sector is a 32×4 SRAM block accessible by adjacent buses. The SRAM can be

configured as either a single-ported or dual-ported RAM, with either synchronous or asynchronous operation.

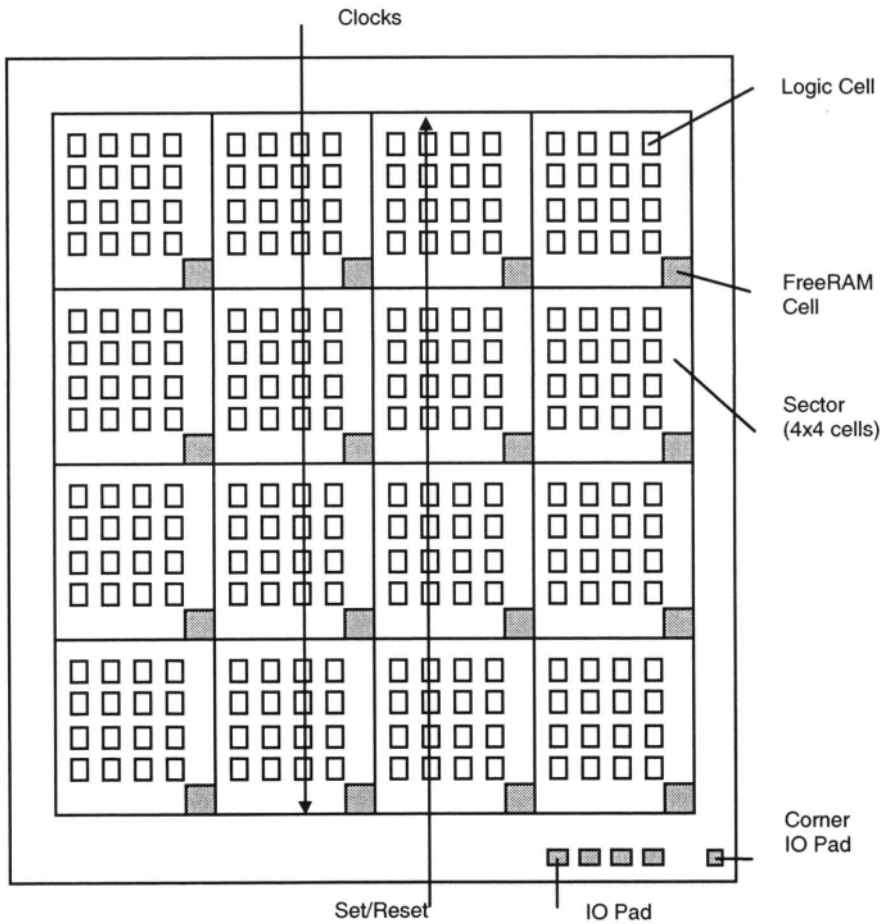



Figure 2.53 Atmel AT40K General Organization

### *2.7.2 Logic Cell*

The logic cell has two 3-input LUTs that can implement any 3-input logic function. The outputs of both LUTs can be connected to the neighboring cells directly or registered using a D-type flip-flop. The organization of a logic cell is shown in Figure 2.54. Each logic cell contains a single flip-flop. The logic cells can be configured in several specialized modes of operation found in most digital systems application areas:

- Synthesis mode combines both LUTs into a single 16×1 LUT and enables implementation of four-input logic functions. Output from the LUT can be registered.

- Tri-state/Mux mode enables implementation of multiplexers by combining one LUT with the tri-state buffer.

- Arithmetic mode in which two LUTs are used to implement three-input logic functions (for example sum and carry) often found in adders, subtractors and accumulators, where one of the LUT outputs can be registered.

- DSP/Multiplier mode in which, with the addition of one upstream AND gate, two LUTs can efficiently calculate both product and carry bits of an multiplication. It can be efficiently used to implement elements of FIR filters or multiply-and-accumulate units.

- Counter mode in which a logic cell can completely implement a single stage of a ripple-carry counter by using internal feedback path and a flip-flop.

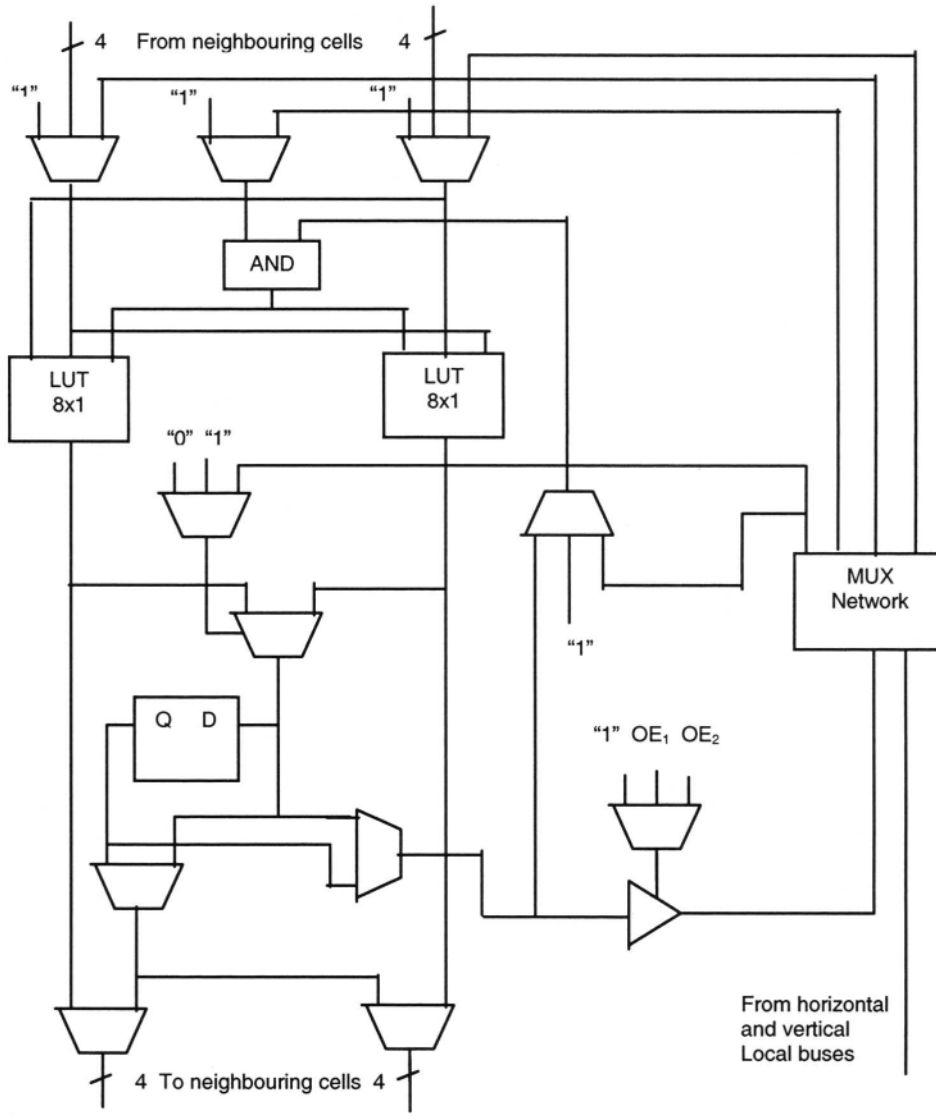



Figure 2.54 Atmel AT40K logic cell architecture

### *2.7.3 Memory Function*

The AT40K includes SRAM blocks that can be used without losing logic resources. It allows creation of multiple independent, synchronous or asynchronous, single-port or dual-port RAM functions. They are constructed using the AT40K SRAM cells, called FreeRAM cells, presented in Figure 2.55.

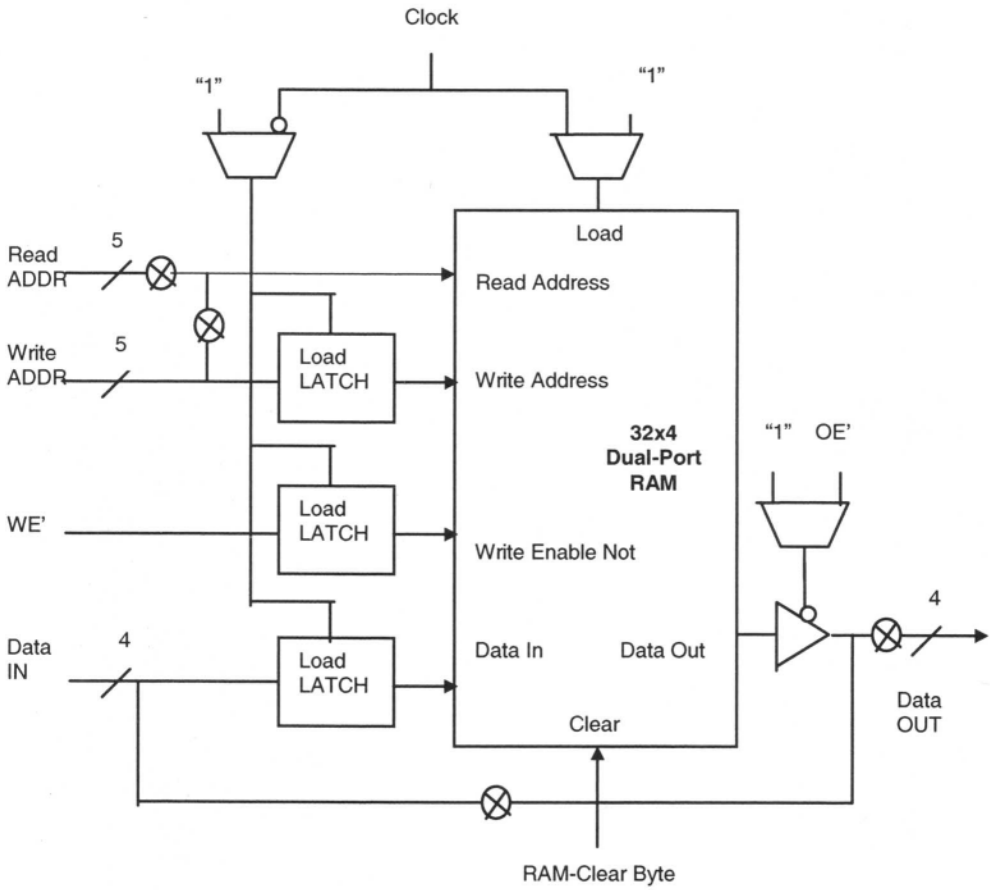



Figure 2.55 Atmel FreeRAM cell

### *2.6.4 Dynamic Reconfiguration*

The AT40K family is capable of implementing dynamic full/partial logic reconfiguration, without loss of data (on-the-fly) for building adaptive logic and systems. Only those portions of the system that are active at a given time are implemented in the FPGA, while inactive portions of the system are stored externally in the configuration memory in the form of configuration streams. As new logic functions are required, they can be downloaded into the logic cache without losing the data already there or disrupting the operation of the rest of the chip, replacing or complementing the active logic. The AT40K can thus act as a reconfigurable co-processor. By time-multiplexing a design, a single Atmel FPGA can implement large designs, which require much larger FPGAs when implemented in other FPLD families.

## 2.7 Problems and Questions

2.1 Implement an 8-to-1 multiplexer using an Altera MAX 7000 device. Write the logic function and determine the number of macrocells required if parallel expanders are used. How would the solution look like if only shareable expanders are used?

2.2 Implement an 8-bit parallel adder using an Altera MAX 7000 device. Write the logic functions and determine the number of macrocells you need for this implementation.

2.3 Implement an 8-bit counter using an Altera MAX 7000 device. Write the logic functions and determine the number of macrocells you need for this implementation.

2.4 Implement a 2-to-l multiplexer using an Altera FLEX 10K device. Write the logic functions and determine the number of logic elements you need for this implementation. Specify the content of the LUT(s) used in this implementation.

2.5 Extend this design from the preceding example to the 4-to-l, 8-to-l and 16-to-l multiplexers. Specify the content of each LUT used in LEs for each of these implementations. How many LEs you need for each implementation?

2.6 Implement an 8-bit parallel adder using LEs from an Altera FLEX 10K device. Write the logic functions and determine the number of logic elements required for implementation.

2.7 Extend the design from the previous example to an 16-bit parallel adder by using the 8-bit parallel adder as the building block. Estimate the speed of addition of two 16-bit numbers assuming that propagation delay through an LUT is 5ns and carry chain delay is 1ns. If you divide the adder to calculate the sum for upper and lower byte at the same time, you have to use the assumption that the value of the carry bit from the lower to the upper byte can be either 0 or 1. Design 16-bit parallel adder that calculates both sums of the upper byte simultaneously and then makes decision which sum to take in final result depending on the value of the carry bit from the lower byte. Estimate the number of LEs required for this implementation. What is the speed of the modified parallel adder?

2.8 Design a 16-bit ripple carry free-running counter using an Altera 10K device LEs. Draw a diagram that shows all interconnection. How many LEs are needed for this implementation?

2.9 Embedded array blocks (EABs) from Altera FLEX 10K device are used to implement 8×8-bit unsigned multiplier. Design this multiplier using 4×4-bit multipliers implemented in a single EAB. Use as many 4×4-bit multipliers to achieve the maximum speed for multiplication. Show all details of the design.

2.10 Repeat the preceding example to design an 8-bit×8-bit multiplier for signed magnitude numbers.

2.11 The multiplier from example 2.9 should be implemented by serial multiplication and addition of 4-bit constituent parts of 8-bit numbers (consider them as hexadecimal digits, each 8-bit number consisting of two hexadecimal digits). The implementation should use a single EAB and other logic as required. How many additional LEs you need for this type of multiplier? How many multiplication and addition steps are needed to perform the task? If those steps are converted into clock cycles, what is the speed at which multiplication can be performed?

2.12 Design a multiplier that multiplies a 12-bit number with constants. For the implementation of this multiplier use Altera FLEX 10K EABs. Show the schematic diagram of your design, content of each EAB (at least a few entries) if the constant is equal

a) $5_{10}$
b) $13_{10}$

2.13 Design a code converter that converts a 6-bit input code into a 10-bit output code using EABs and no additional logic.

2.14 Repeat the preceding example by designing a code converter that a 10-bit input into 16-bit output code.

2.15 Design a data path and control unit (if required) that implement the following requirements. The system has to perform calculation of the following expression on the 8-bit unsigned input numbers A, B and C:

$$R = AxB + AxC \text{ if } A \geq B$$

$$AxB + BxC \text{ if } A < B$$

The above calculation has to be performed by a datapath with the following criteria

a) minimizing the amount of hardware by allowing longer time for calculation
b) minimizing time and by allowing to use all necessary hardware resources

2.16 Extend the preceding example to calculate the same expressions for the stream (array) of 1000 sets of input data. Try to use pipelining to improve the overall performance of the circuit. Estimate the required resources if you use Altera FLEX 10K devices.

2.17 Design a 10-tap finite impulse response filter (FIR) that calculates the following output based on input stream (samples) of 8-bit numbers x(i) and filter coefficients h(i). Assuming that the coefficients h(i) are symmetric, that is h(i)=h(10-i) for every i. The design should be shown at the level of schematic diagram using basic building blocks such as parallel multipliers, adders, and registers that contain input data stream (and implement the delay line).

# 3 DESIGN TOOLS AND LOGIC DESIGN WITH FPLDS

This chapter covers aspects of the tools and methodologies used to design with FPLDs. The need for tightly coupled design frameworks, or environments, is discussed and the hierarchical nature of digital systems design is emphasized. The major design description (entry) tools are introduced including schematic entry tools and hardware description languages. The complete design procedure, which includes design entry, processing, and verification, is shown in an example of a simple digital system. An integrated design environment for FPLD-based designs, the Altera's Max+Plus II environment, is introduced. It includes a variety of design entry, design processing, and verification tools.

## 3.1 Design Framework

FPLD architectures provide identical logic cells (or some of their variations) and interconnection mechanisms as a basis for the implementation of digital systems. These architectures can be used directly for the design of digital circuits. However, the available resources and complexity of designs to be placed in a device require tools that are capable of translating the designer's functions into the exact cells and interconnections needed to form the final design. It is desirable to have design software that will automatically translate designs for different FPLD architectures.

The complexity of FPLDs requires sophisticated design tools that can efficiently handle complex designs. These tools usually integrate several different design steps into a uniform design environment enabling the designer to work with different tools from within the same design framework. They enable design to be performed at a relatively high abstract level, but at the same time allowing the designer to see a physical relationship inside an FPLD device and even change design details at the lowest, physical level.

### *3.1.1 Design Steps and Design Framework*

Design software must perform the following primary functions, as to enable:

Design Entry in some of the commonly used and widely accepted formats. Design entry software should provide an architecture independent design environment that easily adapts to specific designer's needs. The most common design entries belong to the categories of graphic (schematic) design entry, hardware description languages, waveform editors, or some other appropriate tools to transfer designer's needs to a translator.

Translation of design entered by any of the design entry tools or their combinations into the standard internal form that can be further translated for different FPLD architectures. Translation software performs functions such as logic synthesis, timing driven compilation, partitioning, and fitting of design to a target FPLD architecture. Translation mechanisms also provide the information needed for other design tools used in the subsequent design phases.

Verification of a design using functional and timing simulation. In this way many design errors are discovered before actually programming the devices and can be easily corrected using the design entry tools. Usually vendor provided translators produce designs in the forms accepted by industry standard CAE tools that provide extensive verification models and procedures.

Device Programming consisting of downloading design control information into a target FPLD device.

Reusability by providing the libraries of vendor and user designed units that have been proven to operate correctly.

All of the primary functions above are usually integrated into complex design environments or frameworks with a unified user interface. A common element of all these tools is some common circuit representation, most often in the form of so-called netlists.

### *3.1.2 Compiling and Netlisting*

The first step of compiling is the transformation of a design entered in user provided form into the internal form which will be manipulated by the compiler and other tools. A compiler is faced with several issues, the first being will the design fit into the target FPLD architecture at all. Obviously, it depends on the number of input and output pins, but also on the number of internal circuits needed to implement the

desired functions. If the design is entered using a graphic editor and the usual schematic notation, a compiler must analyze the possible implementation of all logic elements in existing logic cells of the targeted FPLD. The design is dissected into known three or four input patterns that can be implemented in standard logic cells, and the pieces are subsequently added up. Initially, the compiler has to provide substitutions for the target design gates into equivalent FPLD cells and make the best use of substitution rules. Once substitution patterns are found, a sophisticated compiler eliminates redundant circuitry. This increases the probability that the design will fit into the targeted FPLD device. Compilers translate the design from its abstract form (schematic, equations, waveforms) to a concrete version, a bitmap forming functions and interconnections. An intermediate design form that unifies various design tools is a netlist. After the process of translating a design into the available cells provided by the FPLD (sometimes called the technology mapping phase), the cells are assigned specific locations within the FPLD. This is called cell placement. Once the cells are assigned to a specific locations the signals are assigned to specific interconnection lines. The portion of the compiler that performs placement and routing is usually called a fitter.

A netlist is a text file representing logic functions and their input/output connections. A netlist can describe small functions like flip-flops, gates, inverters, switches, or even transistors. Also it can describe large units (building blocks) like multiplexers, decoders, counters, adders or even microprocessors. They are very flexible because the same format can be used at different levels of description. For example, a netlist with an embedded multiplexer can be rewritten to have the component gates comprising the multiplexer, as an equivalent representation. This is called netlist expansion. One example of the netlist for an 8-to-l multiplexer is given in the Table 3.1. It simply specifies all gates with their input and output connections, including the inputs and outputs of the entire circuit.

A compiler uses traditional methods to simplify logic designs, but it also uses netlist optimization, which represents design minimization after transformation to a netlist. Today's compilers include a large number of substitution rules and strategies in order to provide netlist optimization. One example of a possible netlist optimization of a multiplexer is shown in Figure 3.1. Although simplified, the example shows the basic ideas behind the netlist optimization. Note that five out of eight inputs to a multiplexer are used. The optimizer scans the multiplexer netlist first finding unused inputs, then eliminates gates driven by the unused inputs. In this way a new netlist, without unneeded inputs and gates, is created.

Table 3.1 Example netlist

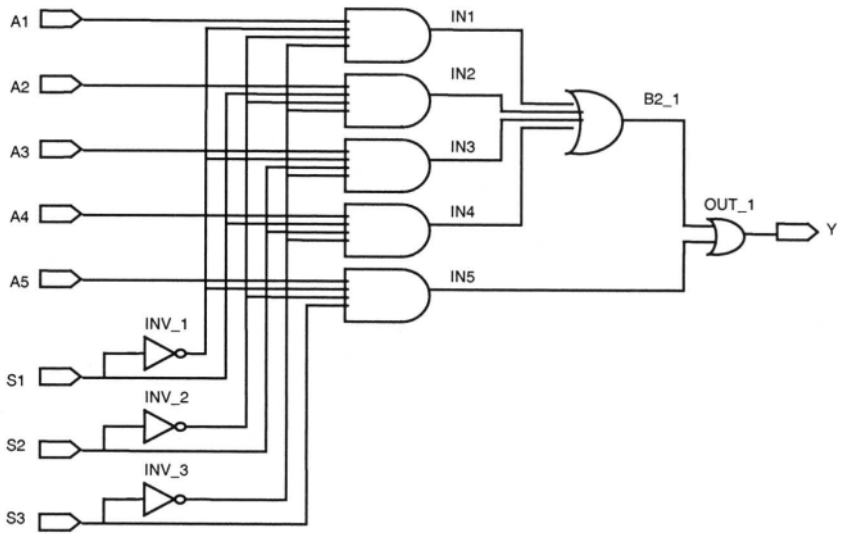
```
NETSTART
*
IN_1     AN4          I(A1, INV_1,INV_2,INV_3 )  O(IN_1 )
IN_2     AN4          I(A2, INV_1,INV_2, S3)  O(IN_2)
IN_3     AN4          I(A3, INV_1, S2, INV_3)  O(IN_3)
IN_4     AN4          I(A4, INV_1, S2, S3)  O(IN_4)
IN_5     AN4          I(A5, S1, INV_2,INV_3)  O(IN_5)
IN_6     AN4          I(A6, S1, INV_2, S3)  O(IN_5)
IN_7     AN4          I(A7, S1, S2, INV_3)  O(IN_6)
IN_8     AN4          I(A8, S1, S2, S3)  O(IN_7)
INV_1     NOT         I(S1)  O(INV_1)
INV_2     NOT         I(S2)  O(INV_2)
INV_3     NOT         I(S3)  O(INV_3)
B2_1     OR4  I(IN_1, IN_2, IN_3, IN_4)  O(B2_1)
B2_2     OR4  I(IN_5, IN_6, IN_7, IN_8)  O(B2_2)
OUT_1     OR2         I(B2_1, B2_2)  O(Y)

*
NETEND
*
NETIN     A1, A2, A3, A4, A5,A6,A7,A8, S1,S2,S3
NETOUT       Y
*
```

Even this simple example shows potential payoffs when larger designs are optimized. Optimization procedures are repeated as long as there are gates and flip-flops that can be eliminated or there are logic gates performing identical functions that can be combined and duplication avoided. Even though complex rules are applied during optimization, they save FPLD resources.

After netlist optimization, logic functions are translated into available logic cells with an attempt to map (as much as possible) elementary gates into corresponding logic cells. The next step is to assign logic functions to specific locations within the device. The compiler usually attempts to place them into the simplest possible device if it is not specified in advance.

Cell placement requires iteration and sometimes, if the compiler produces unsatisfactory results, manual placement may be necessary. The critical criteria for cell placement is that interconnections of the cells must be made in order to implement the required logic functions. Additional requirements may be minimum skew time paths or minimum time delays between input and output circuit pins. Usually, several attempts are necessary to meet all constraints and requirements




```
NETSTART

IN_1     AN4   I(A1, INV_1,INV_2,INV_3 )  O(IN_1 )
IN_2     AN4   I(A2, INV_1,INV_2, S3)  O(IN_2)
IN_3     AN4   I(A3, INV_1, S2, INV_3)  O(IN_3)
IN_4     AN4   I(A4, INV_1, S2, S3)   O(IN_4)
IN_5     AN4   I(A5, S1, INV_2,INV_3)  O(IN_5)
INV_1      NOT  I(S1)  O(INV_1)
INV_2      NOT  I(S2)  O(INV_2)
INV_3      NOT  I(S3)  O(INV_3)
B2_1     OR4   I(IN_1, IN_2, IN_3, IN_4)  O(B2_1)
OUT_1      OR2   I(B2_1, B2_2)  O(Y)
NETEND
NETIN      A1, A2, A3, A4, A5,A6,A7,A8, S1,S2,S3
NETOUT   Y
```

Figure 3.1 Netlist optimization

Some compilers allow the designer to implement portions of a design manually. Any resource of the FPLD, such as an input/output pin (cell) or logic cell can perform a specific user defined task. In this way; some logic functions can be placed together in specific portions of the device; specific functions can be placed in specific devices (if the projects cannot fit into one device), and inputs or outputs of a logic function can be assigned to specific pins, logic cells, or specific portions of the device. These assignments are taken as fixed by the compiler and it then produces placement for the rest of the design.

Assuming the appropriate placement of cells and other resources, the next step is to connect all resources. This step is called routing. Routing starts with the examination of the netlist that provides all interconnection information, and from inspection of the placement. The routing software assigns signals from resource outputs to destination resource inputs. As the connection proceeds, the interconnect lines become used, and congestion appears. In this case the routing software can fail to continue routing. At this point, the software must replace resource placement into another arrangement and repeat routing again.

As the result of placement and routing design, a file describing the original design is obtained. The design file is then translated to a bitmap that is passed to a device programmer to configure the FPLD. The type of device programmer depends on the type of FPLD programming method (RAM or (E)EPROM based devices).

Good interconnection architectures increase the probability that the placement and routing software will perform the desired task. However, bad routing software can waste a good connection architecture. Even in the case of total interconnectivity, when any cell could be placed at any site and connected to any other site, the software task is very complex. This complexity is increased when constraints are added. Such constraints are, for instance, a timing relationship or requirement that the flip-flops of some register or counter must be placed into adjacent logic cells within the FPLD. These requirements must be met first and then the rest of the circuit is connected. In some cases, placement and routing become impossible. This is the reason to keep the number of such requirements at the minimum.

## 3.2 Design Entry and High Level Modeling

Design entry can be performed at different levels of abstraction and in different forms. It represents different ways of design modeling, some of them being suitable for behavioral simulation of the system under the design and some being suitable for circuit synthesis. Usually, the two major design entry methods belong to schematic entry systems or textual entry systems.

Schematic entry systems enable a design to be described using primitives in the form of standard SSI and MSI blocks or more complex blocks provided by the FPLD vendor or designer. Textual entry systems use hardware description languages to describe system behavior or structures and their interconnections.

Advanced design entry systems allow combinations of both design methods and the design of subsystems that will be interconnected with other subsystems at a higher level of design hierarchy. Usually, the highest level of design hierarchy is

called the project level. Current projects can use and contain designs done in previous projects as its low level design units.

In order to illustrate all design entry methods, we will use an example of a pulse distributor circuit that has Clock as an input and produces five non-overlapping periodic waveforms (clock phases) at the output as shown in Figure 3.2. The circuit has asynchronous Clear, Load initial parallel data, and Enable input which must be active when the circuit generates output waveforms.

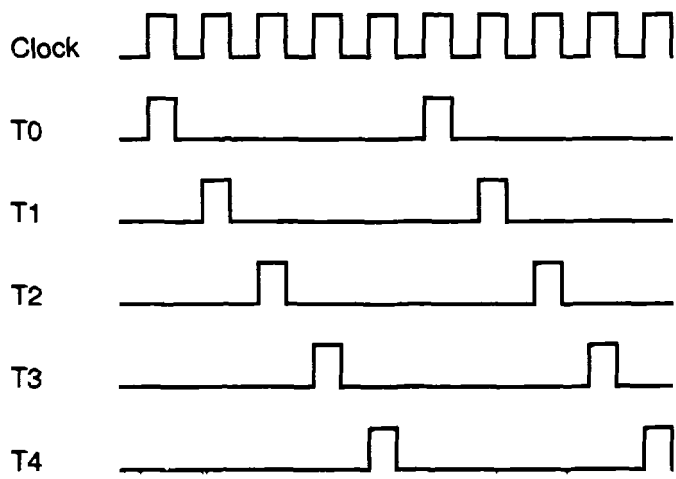



Figure 3.2 Waveforms produced by a pulse distributed circuit

### *3.2.1 Schematic Entry*

Schematic entry is a traditional way to specify a digital system design. A graphics editor is a schematic entry capture program that allows relatively complex designs to be entered quickly and easily. Built-in and extensible primitive and macrofunction libraries provide basic building blocks for constructing a design, while the symbol generation capability enables users to build libraries of custom functions. A graphic editor usually provides a WYSIWYG (What You See Is What You Get) environment. Typical provided primitives include input and output pins, elementary logic gates, buffers, and standard flip-flops. Vendor provided libraries contain macrofunctions equivalent to standard 74- series digital circuits (SSI and MSI), with standard input and output facilities. In the translation process these

circuits are stripped off the unused portions such as unused pins, gates, and flip-flops.

A graphic editor enables easy connection of desired output and input pins, editing of new design, duplication of portions or complete design, etc. Symbols can be assigned to new designs and used in subsequent designs. Common features of a graphic editor are:

- Symbols are connected with single lines or with bus lines. When the name is assigned to a line or bus, it can be connected to another line or bus either graphically or by name only.
- Multiple objects can be selected and edited at the same time.
- Complete areas containing symbols and lines can be moved around the worksheet while preserving signal connectivity. Any selected symbol or area can be rotated.
- Resources can be viewed and edited in the graphic editor, such as probes, pins, logic cells, blocks of logic cells, logic and timing assignments.

The example pulse distribution circuit represented by a schematic diagram is shown in Figure 3.3. Some standard 74- series components are used in its implementation.

### *3.2.2 Hardware Description Languages*

Hardware description languages (HDLs) represent another tool for the description of digital system behavior and structure at different abstraction levels. HDLs belong either to a category of vendor designed languages or general languages that are independent of the vendor.

An example of a vendor provided HDL is Altera's HDL (AHDL). It is a high level, modular language that is integrated into the design environment. AHDL consists of a variety of elements and behavioral statements that describe logic systems. AHDL is a very convenient tool for describing functions such as state machines, truth tables, Boolean functions, conditional logic, and group operations.

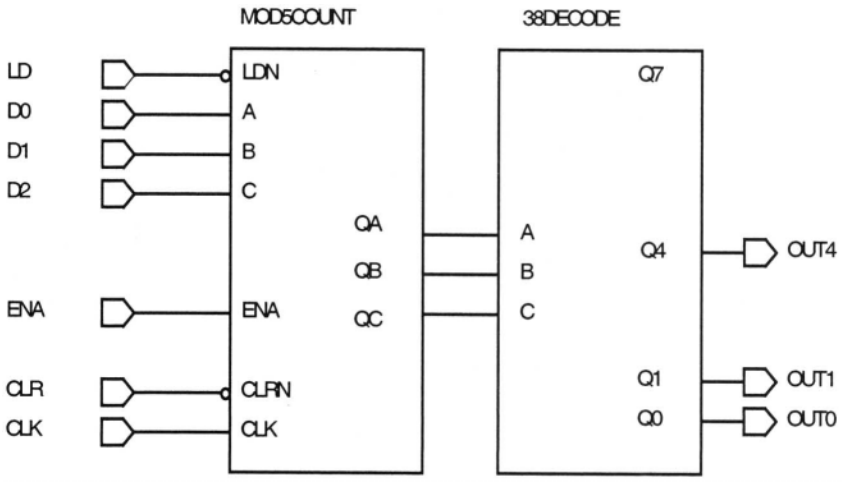



Figure 3.3 Pulse distribution circuit represented by a schematic diagram

It facilitates implementation of combinational logic, such as decoders, multiplexers, arithmetic logic circuits, using Boolean functions and equations, macrofunctions, and truth tables. It allows the creation of sequential logic circuits, such as various types of registers and counters, using Boolean functions and equations, macrofunctions, and truth tables. Frequently used constants and prototypes (of vendor provided or user defined macrofunctions) can be stored in libraries in include files and used where appropriate in new design (textual) files. State machines can be designed using user defined state assignments or by the Compiler.

A detailed introduction to AHDL and a presentation of its features and design mechanisms of digital circuits is given in Chapter 4 and subsequent chapters. For the purpose of providing the "flavor" of the tool, our example pulse distribution circuit is described in AHDL in Example 3.1.

Example 3.1 AHDL Pulse Distribution Circuit.

```
INCLUDE "modscount";
INCLUDE "38decode";

SUBDESIGN pulsdist
 (
 d[2..0]: INPUT
 clk,clr,ld,ena        : INPUT;
 out[4..0]  :OUTPUT;
)
```

```
VARIABLE
 counter  : mod5count;
 decoder  : 8dmux;

BEGIN
 counter.clk = clk;
 decoder.(c,b,a) = counter.(qc,qb,qa);
 out[4..0] = decoder.q[4..0];
END;
```

Include Statements are used to import function prototypes for two already provided user macrofunctions. In the variable section, a variable counter is declared as an instance of the `mod5count` macrofunction and the variable decoder is declared as an instance of the `38decode` macrofunction. They represent a binary modulo-5 counter and decoder of the type 3-to-8, respectively.

The example shows some of the most basic features and potential of hardware description languages. Once described, designs can be compiled and appropriate prototypes and symbols assigned, which can be used in subsequent designs. This approach is not merely leading to the library of designs, but is introducing reusability as a concept in rapid system prototyping.

Although AHDL allows efficient implementation of many combinational and sequential circuits, it can be considered a traditional hardware description language that can be applied mainly to structural and low level digital design.

As a result of the needs and developments in digital systems design methodologies, VHDL (Very High Speed Integrated Circuit Hardware Description Language) and Verilog HDL have emerged as the standard tool for description of digital systems at various levels of abstraction optimized for transportability among many computer design environments. Both these languages are described in more details in chapters 9 to 15. VHDL is a specification language that follows the philosophy of an object-oriented approach and stresses object-oriented specification and reusability concepts. It describes inputs, outputs, behavior, and functions of digital circuits. It is defined by the IEEE Standard 1076-1987 and revision 1076-1993. In order to compare different design tools on the example, our small pulse distribution circuit is described using VHDL in Example 3.2.

Example 3.2 VHDL Pulse Distribution Circuit.

```
use work.mycomp.all;

entity pulsdist is
 port(d: in integer range 0 to 7;
    clk, ld, ena, clr: in bit;
    q: out integer range  0 to 255);
```

```
end pulsdist;

architecture puls_5 of pulsdist is
 signal a: integer range 0 to 7;
begin
 cnt_5: mod_5_counter port map (d,clk,ena,clr,a);
 dec_1: decoder3_to_8 port map (a, q);
end puls_5;
```

The basic VHDL design units (entity and architecture) appear in this example. The architecture `puls_5` of the pulse contains instances of two components from the library `mycomp`, `cnt_5` of type `mod_5_counter`, and `dec_1` of type `decoder3_to_8`. The complete example of the pulse distributor is presented in the next section, where the hierarchy of design units is introduced. A more detailed introduction to VHDL is presented in Chapter 7. It must be noted that VHDL is a very complex language and as such can be used in a variety of ways. It allows a designer to build his/her own style of design, while still preserving its features of transportability to different design environments.

### *3.2.3 Hierarchy of Design Units - Design Example*

As mentioned earlier, design entry tools usually allow the use of design units specified in a single tool and also the mixing of design units specified in other tools. This leads to the concept of project as the design at the highest level of hierarchy. The project itself consists of all files in a design hierarchy including some ancillary files produced during the design process. The top level design file can be a schematic entry or textual design files, defining how previously designed units, together with their design files are used.

Consider the design hierarchy of our pulse distribution circuit. Suppose the circuit consists of a hierarchy of subcircuits as shown in the Figure 3.4. Figure 3.4 also shows us how portions of the pulse distribution circuit are implemented. At the top of the hierarchy we have the VHDL file (design) that represents the pulse distributor. It consists of modulo-5 counter circuit, denoted MOD-5-COUNTER which is also designed in VHDL. Decoder 3-to-8, denoted DECODER 3-TO-8 is designed using schematic editor. This decoder is designed, in turn, using two 2-to-4 decoders, denoted DECODER 2-TO-4, with enable inputs designed in VHDL. Any of the above circuits could be designed in AHDL, as well. This example just opens the window to a powerful integrated design environment which provides even greater flexibility.

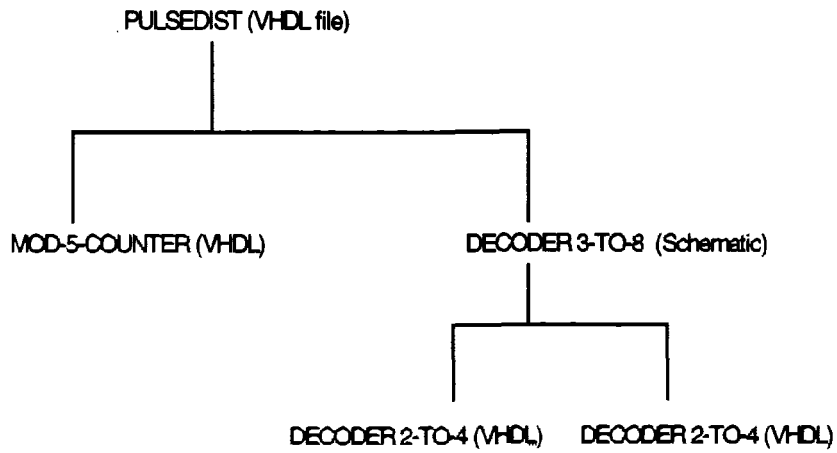



Figure 3.4 Design hierarchy of pulse distribution circuit project

The topmost VHDL file specifying our pulse distributor was given in the preceding section. As was mentioned, it represents the structural representation of the circuit that uses two components, `mod_5_counter` and `decoder3_to_8`. VHDL specification of the modulo-5 counter is given in Example 3.3.

Example 3.3 VHDL Modulo-5 Counter5

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mod_5_counter is
 port(d: in integer range 0 to 4;
    clk, ld, ena, clr: in bit;
    q: inout integer range 0 to 4);
end mod_5_counter;

architecture cnt_5 of mod_5_counter is
begin
 process (clk)
    variable cnt: integer range 0 to 4;
 begin
    if(clk'event and clk='1' then
       if (clr = '0' or q = 4) then
         cnt := 0;
```

```
        else
           if ld = '0' then
            cnt := d;
            else
               if ena = '1' then
                      cnt := cnt +1;
               end if;
            end if;
           end if;
        end if;
           q <= cnt;

    end process;
   end cnt_4;
```

This counter uses a behavioral style architecture that describes the modulo-5 counter behavior rather than structure. Most VHDL compilers are capable of synthesizing a circuit that carries out the desired function.

Decoder 3-to-8 is designed using a schematic entry with type 2-to-4 decoders, and one standard inverter as its basic components. The schematic diagram of this decoder is shown in Figure 3.5.

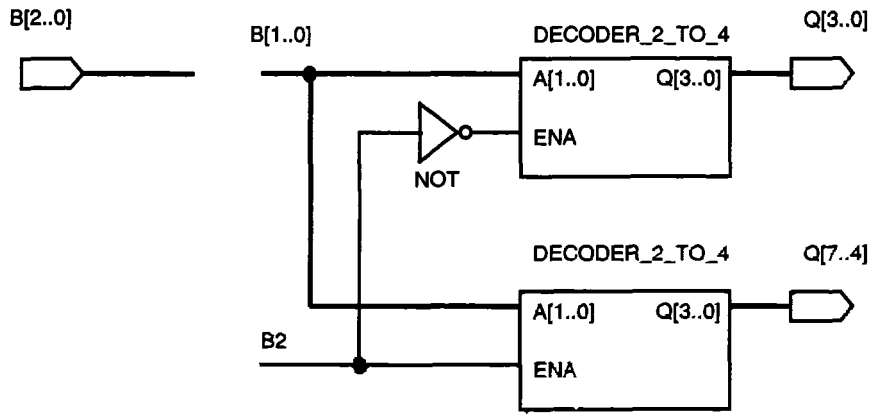



Figure 3.5 Schematic diagram of decoder 3-to-8

Type 2-to-4 decoder is designed using VHDL as shown in Example 3.4.

Example 3.4 VHDL 2-to-4 Decoder.

```
library ieee;
use ieee.std_logic_1164.all;

entity decoder_2_to_4 is
 port (a: in integer range 0 to 3;
    en: in bit;
    q: out integer range 0 to 15;
end decoder_2_to_4;

architecture dec_behav of decoder_2_to_4 is

begin
 q <= 1 when (en = '1' and a = 0) else
    2 when (en = '1' and a = 1) else
    4 when (en = '1' and a = 2) else
    8 when (en = '1' and a = 3) else
    0;
end dec_behav;
```

The architecture of the decoder is given again in a behavioral style demonstrating some of the powerful features of VHDL.

Another important component in the hierarchical design of projects is the availability of libraries of primitives, standard 74-series, and application specific macrofunctions, including macrofunctions that are optimized for the architecture of a particular FPLD device or family. Primitives are basic function blocks such as buffers, flip-flops, latches, input/output pins, and elementary logic functions. They are available in both graphical and textual form and can be used in schematic diagrams and textual files. Macrofunctions are high level building blocks that can be used with primitives and other macrofunctions to create new logic designs. Unused inputs, logic gates, and flip-flops are automatically removed by a compiler, ensuring optimum design implementation. Macrofunctions are usually given together with their detailed implementation, enabling the designer to copy them into then- own library and edit them according to specific requirements.

The common denominator of all design entry tools is the netlist level at which all designs finally appear. If standard netlist descriptions are used, then further tools that produce actual programming data or perform simulation can be specified by another design specification tool, provided the compiler produces standard netlist formats.

### 3.3 Design Verification and Simulation

Design verification is necessary because there are bugs and errors in the translation, placement and routing processes, as well as errors made by a designer. Most verification tools are incorporated into design tools and examine netlists and analyze properties of the final design. For instance, a design checker can easily identify the number of logic cells driven by any logic cell and determine how it contributes to a cumulative load resulting in a time delay attached to the driving cell's output. If the delay is unacceptable, the designer must split the load among several identical logic cells. Similarly, a design checker can identify unconnected inputs of logic cells which float and produce noise problems.

While some checks can be performed during design compilation, many checks can only be done during a simulation that enables assessing functionality and the timing relationship and performance of an FPLD based design.

Regardless of the type of logic simulation, a model of the system is created and driven by a model of inputs called stimuli or input vectors, that generate a model of output signals called responses or output vectors. Simulation is useful not only for observing global behavior of the system under design, but also because it permits observation of internal logic at various levels of abstraction, which is not possible in actual systems.

Two types of simulation used in digital systems design are functional simulation and timing simulation. Functional simulation enables observation of design units at the functional level, by combining models of logic cells with models of inputs to generate response models that takes into account only relative relationships among signals and neglecting circuit delays. This type of simulation is useful for a quick analysis of system behavior, but produces inaccurate results because propagation delays are not taken into account.

Timing simulation takes into account additional element in association with each cell model output, a time delay variable. Time delay enables more realistic modeling of logic cells and the system as the whole. They consist of several components that can or cannot be taken into account, such as time delay of logic cells, without considering external connections, time delays associated with the routing capacitance of the metal connecting outputs with the inputs of logic cells, and time delay which is the function of the driven cell input impedances.

Timing simulation is the major part of verification process of the FPLD design. Timing simulator uses two basic information to produce output response, Input Vectors and Netlists. Input vectors are given in either tabular or graphical form.

Input timing diagrams represent a convenient form to specify stimuli of simulated design.

Netlists represent an intermediate form of the system modeled. Besides connectivity information, netlists contain information about delay models of individual circuits and logic cells, as well as logic models that describe imperfections in the behavior of logic systems. These models increase complexity of used logic, but at the same time improve quality of the model and system under design.

The simulator applies the input vectors to the system model under design and after processing according to the input netlists and models of individual circuits, produces the resulting output vectors. Usually, outputs are presented in the form of timing diagrams. Later, both input and output timing diagrams can be used by electronic testers to compare simulated and real behavior of the design.

In order to perform simulation, the simulator has to maintain several internal data structures that easily and quickly help find the next event requiring the simulation cycle to start. A simulation event is the occurrence of a netlist node (gate, cell, output, etc.) making a binary change from one value to another. The scheduler is a part of the simulator that keeps a list of times and events, and dispatches events when needed. The process initiates every simulated time unit regardless of an event existence (in that case we say that simulation is time driven) or only at the time units in which there are some events (event driven simulation).

The second type of simulation is more popular and more efficient in today's simulators. The most important data structure is the list of events that must be ordered according to increased time of occurrence.

In the ideal case, these changes are simply binary (from zero to one and vice versa), but more realistic models take into account imperfect or sometimes unspecified values of signals. The part of a simulator called the evaluation module is activated at each event and uses models of functions that describe behavior of the subsystems of design. These models take into account more realistic electrical conditions of circuit behavior, such as three-state outputs, unknown states, and time persistence, essentially introducing multi-valued instead of common binary logic. Some simulators use models with up to twelve values of the signals. This leads to complex truth tables and complex and time consuming simulation even for simple logic gates, but also produces a more accurate simulation results.

## 3.4 Integrated Design Environment Example: Altera's Max+Plus II

An integrated design environment for EPLD/CPLD design represents a complete framework for all phases of the design process, starting with design entry and ending with device programming. Altera's Max+Plus II is an integrated software package for designing with Altera programmable devices. The same design can be retargeted to various devices without changes to the design itself. Max+Plus II consists of a spectrum of logic design tools and capabilities, such as a variety of design entry tools for hierarchical projects, logic synthesis algorithms, timing-driven compilation, partitioning, functional and timing simulation, linked multi device simulation, timing analysis, automatic error location, and device programming and verification. It is also capable of reading netlist files produced by other vendor systems or producing netlist files for other industry standard CAE software.

The Max+Plus II design environment is shown in Figure 3.6. The heart of the environment is a compiler capable of accepting design specifications in various design entry tools, and producing files for two major purposes, design verification and device programming. Design verification is performed using functional or timing simulation, or timing analysis and device programming, is performed by Altera's or other industry standard programmers. Output files produced by the Max+Plus II compiler can be used by other CAE software tools.

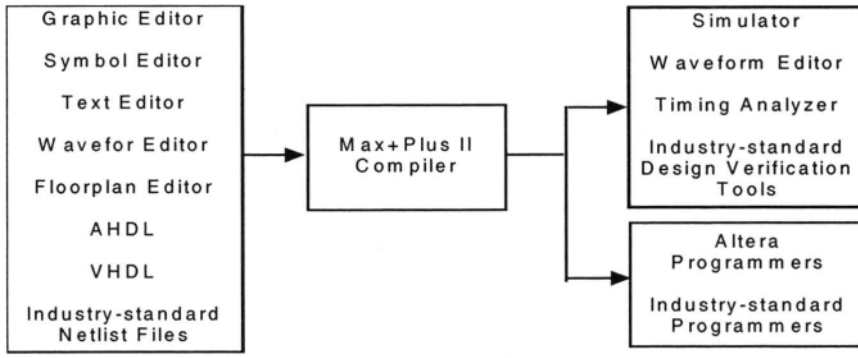



Figure 3.6 Max+Plus II Design Environment.

Once the logic design is created, the entity is called a project. A project can include one or more subdesigns (previously designed projects). It combines different types of subdesigns (files) into a hierarchical project, choosing the design entry format that best suits each functional block. In addition, large libraries of

Altera provided macrofunctions simplify design entry. Macrofunctions are available in different forms and can be used in all design entry tools.

A project consists of all files in a design hierarchy. If needed, this hierarchy can be displayed at any moment, and the designer sees all files that make the project, including design and ancillary files. Design files represent a graphic, text, or waveform file created with a corresponding editor, or with another industry standard schematic text editor or a netlist writer. The Max+Plus II compiler can process the following files:

- Graphic design files (.gdf)
- Text design files (.tdf)
- Waveform files (.wdf)
- VHDL files (.vhd)
- Verilog files (.v)
- OrCAD schematic files (.sch)
- EDIF input files (.edf)
- Xilinx netlist format files (.xnf)
- Altera design files (.adf)
- State machine files (.smf)

Ancillary files are associated with a project, but are not part of a project hierarchy tree. Most of them are generated by different Max+Plus II functions and some of them can be entered or edited by a designer. Examples of ancillary files are assignment and configuration files (.acf) and report files (.rpt).

The Max+Plus II compiler provides powerful project processing and customization to achieve the best or desired silicon implementation of a project. Besides fully automated procedures, it allows designer to perform some of the assignments or functions manually to control a design. A designer can enter, edit, and delete resource and device assignments that control project compilation, including logic synthesis, partitioning, and fitting.

The Max+Plus II design environment is Windows based. This means that all functions can be invoked using menus or simply by clicking on different buttons with the icons describing corresponding functions.

### *3.4.1 Design Entry*

Max+Plus II provides three design entry editors: the graphic, text, and waveform editors. Two additional editors are included to help facilitate design entry, the floorplan and symbol editor.

Design entry methods supported by Max+Plus II are:

- Schematic, designs are entered in schematic form using Graphic editor.
- Textual, AHDL, VHDL or Verilog designs are entered using Altera's or any other standard text editor
- Waveform, designs are specified wit Altera's waveform editor
- Netlist, designs in the form of netlist files or designs generated by other industry standard CAE tools can be imported into Max+Plus II design environment
- Pin, logic cell, and chip assignments for any type of design file in the current project can be entered in a graphical environment with the floorplan editor.
- Graphic symbols that represent any type of design file can be generated automatically in any design editor. Symbol editor can be used to edit symbols or create own customized symbols.

The Assign menu, accessed in any Max+Plus II application, allows the user to enter, edit, and delete the types of resource and device assignments that control project compilation. This information is saved in assignment and configuration files (.acf) for the project. Assignment of device resources can be controlled by the following types of assignments:

- Clique assignments specify which logic functions must remain together in the same logic array block, row, or device
- Chip assignments specify which logic must remain together in a particular device when a project is partitioned into multiple devices
- Pin assignments assign the input or output of a single logic function to a specific pin, row, or column within a chip
- Logic cell assignments assign a single logic function to a specific location within a chip (to a logic cell, I/O cell, LAB, row, or column)
- Probe assignments assign a specific name to an input or output of a logic function

- Connected pin assignments specify how two or more pins are connected externally on the printed circuit board.
- Device assignments assign project logic to a device (for example, maps chip assignments to specific devices in multi-device project)
- Logic option assignments that specify the logic synthesis style in logic synthesis (synthesis style can be one of three Altera provided or specified by designer)
- Timing assignments guides logic synthesis tools to the desired performance for input to non-registered output delays ($t_{PD}$), clock to output delays ($t_{CO}$), clock setup time ($t_{SU}$), and clock frequency ($f_{MAX}$).

Max+Plus II allows preservation of the resource assignments the compiler made during the most recent compilation so that we can produce the same fit with subsequent compilation. This feature is called back annotation. It becomes essential because after compiling all time delays are known and the design software can calculate a precise annotated netlist for the circuit by altering the original netlist. The subsequent simulation using this altered netlist is very accurate and can show trouble spots in the design that are not otherwise observable.

Some global device options can be specified before compilation such as the reservation of device capacity for future use or some global settings such as an automatic selection of a global control signal like the Clock, Clear, Preset, and Output Enable. The compiler can be directed to automatically implement logic in I/O cell registers.

### *3.4.2 Design Processing*

Once a design is entered, it is processed by the Max+Plus II compiler producing the various files used for verification or programming. The Max+Plus II compiler consists of a series of modules that check a design for errors, synthesize the logic, fit the design into the needed number of Altera devices, and generate files for simulation, timing analysis, and device programming. It also provides a visual presentation of the compilation process, showing which of the modules is currently active and allowing this process to be stopped.

Besides design entry files, the inputs to the compiler are the assignment and configuration files of the project (.acf), symbol files (.sym) created with Symbol editor, include files (.inc) imported into text design files containing function prototypes and constants declarations, and library mapping files (.lmf) used to map

EDIF and OrCAD files to corresponding Altera provided primitives and macrofunctions.

The compiler netlist extractor first extracts information that defines hierarchical connections between a project's design files and checks the project for basic design entry errors. It converts each design file in the project into a binary Compiler Netlist File (.cnf) and creates one or more Hierarchy Interconnect Files (.hif), a Symbol File (.sym) for each design file in a project, and a single Node Database File (.ndb) that contains project node names for assignment node database.

If there are no errors, all design files are combined into a flattened database for further processing. Each Compiler Netlist File is inserted into the database as many times as it is used in the original hierarchical project. The database preserves the electrical connectivity of the project.

The compiler applies a variety of techniques to implement the project efficiently in one or more devices. The logic synthesizer minimizes logic functions, removes redundant logic, and implements user specified timing requirements.

If a project does not fit into a single device, the partitioner divides the database into the minimal number of devices from the same device family. A project is partitioned along logic cell boundaries and the number of pins used for inter-device communication is minimized.

The fitter matches project requirements with known resources of one or more device. It assigns each logic function to a specific logic cell location and tries to match specific resource assignments with available resources. If it does not fit, the fitter issues a message with the options of ignoring some or all of the required assignments.

Regardless if a fit is achieved or not, a report file (.rpt) is created showing how a project will be implemented. It contains information on project partitioning, input and output names, project timing, and unused resources for each device in the project.

At the same time, the compiler creates a functional or timing simulation netlist file (.snf) and one or more programming files that are used to program the devices. The programming image can be in the form of one or more programmer object files (.pof), or SRAM object files (.sof files). For some devices, JEDEC files (.jed) can be generated.

As an example, our pulse distributor circuit is compiled by the Max+Plus II compiler without constraints or user required assignments of resources. The tables below show some of the results of compilation. The compiler has placed the pulsdist circuit into the EPF8282LC84 device with the logic cell utilization of 5%.

Other important information about the utilization of resources is available in the tables 3.2 through 3.5 given below.

Table 3.2 Device summary for pulsdist circuit

| Chip | Device | Input pins | Output pins | Bidir pins | Logic cells | % utilized |
|---|---|---|---|---|---|---|
| pulsdist | EPF8282 LC84 | 7 | 8 | 0 | 12 | 5% |
| User pins | | 7 | 8 | | | |

Table 3.3 Logic cell utilization

| Column / Row | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| B | 4 | 3 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 12 |
| Total | 4 | 3 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 12 |

Table 3.4 Interconnect mechanism usage

| LAB | Logic cells | Column Interconn. Driven | Row Interconn. Driven | Clocks | Clear/ Preset | External Interconn. |
|---|---|---|---|---|---|---|
| B1 | 4/8 (50%) | 0/8 (0%) | 3/8 (37%) | 1/2 | 1/2 | 6/24 (25%) |
| B2 | 3/8 (37%) | 0/8 (0%) | 3/8 (37%) | 0/2 | 0/2 | 3/24 (12%) |
| B3 | 1/8 (12%) | 0/8 (0%) | 1/8 (12%) | 0/2 | 0/2 | 3/24 (12%) |
| B4 | 1/8 (12%) | 0/8 (0%) | 1/8 (12%) | 0/2 | 0/2 | 3/24 (12%) |
| B5 | 1/8 (12%) | 0/8 (0%) | 1/8 (12%) | 0/2 | 0/2 | 3/24 (12%) |
| B6 | 1/8 (12%) | 0/8 (0%) | 1/8 (12%) | 0/2 | 0/2 | 3/24 (12%) |
| B7 | 1/8 (12%) | 0/8 (0%) | 1/8 (12%) | 0/2 | 0/2 | 3/24 (12%) |

Table 3.5 Other resources utilization

| | |
|---|---|
| Total dedicated input pins used | 1/4 (25%) |
| Total I/O pins used | 16/64 (25%) |
| Total logic cells used | 12/208 ( 5%) |
| Total input pins required | 7 |
| Total input registers required | 0 |
| Total output pins required | 8 |
| Total output registers required | 0 |
| Total buried I/O cell registers required | 0 |
| Total bidirectional pins required | 0 |
| Total reserved pins required | 2 |
| Total logic cells required | 12 |
| Total flip-flops required | 3 |
| Total logic cells in carry chains | 3 |
| Total number of carry chains | 1 |
| Total logic in cascade chains | 0 |
| Total number of cascade chains | 0 |
| Synthesized logic cells | 0/208 ( 0%) |

### *3.4.3 Design Verification*

The process of project verification is aided with two major tools: the simulator, and the timing analyzer. The simulator tests the logical operation and internal timing of a project. To simulate a project, a Simulator Netlist File (.snf) must be produced by the compiler. An appropriate SNF file (for functional, timing, or linked multi-project simulation) is automatically loaded when the simulator is invoked.

The input vectors are in the form of a graphical waveform Simulator Channel File (.scf) or an ASCII Vector File (.vec). The Waveform editor creates a default SCF file. The simulator allows the designer to check the outputs of the simulation against any outputs in SCF, such as user defined outputs or outputs from a previous simulation. It can also be used to monitor glitches, oscillations, and setup and hold time violations.

An example of the simulator operation is given for our pulse distributor circuit in Figure 3.7. Input vectors are denote by capital letter I, and output vectors by capital letter O. A total of 800 ns was simulated. In Figure 3.7 a 280 ns interval of the simulation is shown.

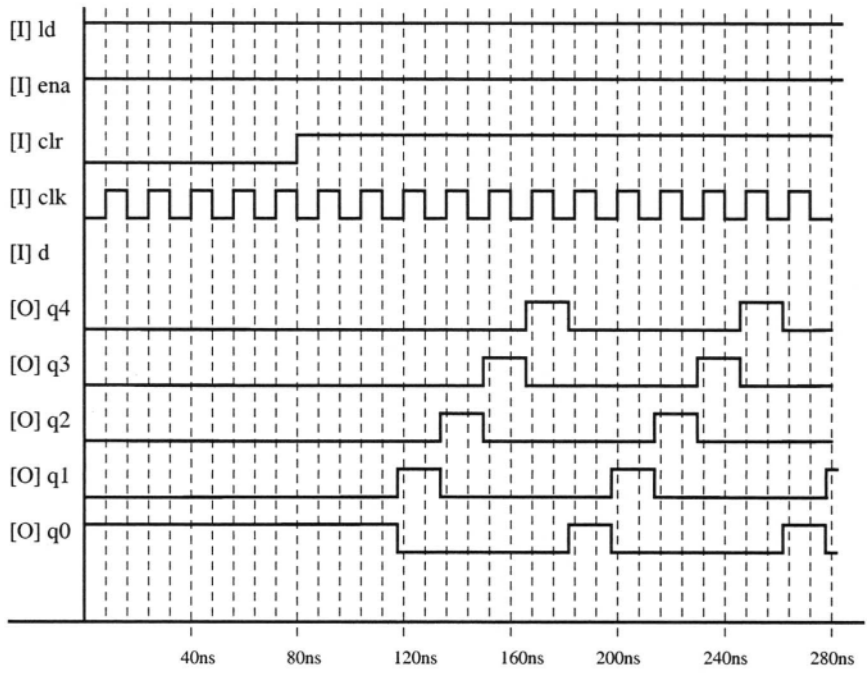



Figure 3.7 Simulation results for pulse distributor circuit

The input clock period is 16 ns. Further timing analysis has shown that the circuit can safely run to the minimum clock period of 15.7 ns or frequency of 63.69 MHz.

The Max+Plus II Timing analyzer allows the designer to analyze timing performance of a project after it has been optimized by the compiler. All signal paths in the project can be traced, determining critical speed paths and paths that limit the project's performance. The timing analyzer uses the network and timing information from a timing Simulator Netlist File (.snf) generated by the compiler. It generates three types of analyses, the delay matrix, the set up/ hold matrix, and the registered performance display.

The delay matrix shows the shortest and longest propagation delay paths between multiple source and destination nodes. The setup/hold matrix shows the minimum required setup and hold times from input pins to the D, Clock, and latch Enable inputs to flip-flops and latches. The registered performance display shows the results of a registered performance analysis, including the performance limited delay, minimum Clock period, and maximum circuit frequency.

After the timing analyzer completes an analysis, it is possible to select a source or destination node and list its associated delay paths. Using the message processor it is easy to open and list the paths for the selected node and locate a specific path in the original design file.

### *3.4.4 Device Programming*

The last portion of Altera's integrated design environment is the hardware and software necessary for programming and verifying Altera devices. The software part is called the Max+Plus II programmer. For EPROM base devices Altera provides an add-on Logic Programmer card (for PC-AT compatible computers) that drives the Altera Master Programming Unit (MPU). The MPU performs continuity checks to ensure adequate electrical contact between the programming adapter and the device. With the appropriate programming adapter, the MPU also supports functional testing. It allows the application of simulation input vectors to verify its functionality.

For the FLEX 8000 family, Altera provides the FLEX download cable and the BitBlaster. The FLEX download cable can connect any configuration EPROM programming adapter, which is installed on the MPU, to a single target FLEX 8000 device. The BitBlaster serial download cable is a hardware interface to a standard RS-232 port that provides configuration data to FLEX 8000 devices. The BitBlaster allows the designer to configure the FLEX 8000 device independently from the MPU or any other programming hardware.

## 3.5 System prototyping: Altera UP1 Prototyping Board

Altera UP1 prototyping board and package have been designed specifically to meet the needs of educational purposes of digital design at the university level. The package includes prototyping board, ByteBlaster download device, that enables downloading FPLDs from the PC computer, and the Max+Plus II design environment. These three components provide all of the necessary tools for creating and implementing digital logic designs. The entire UP prototyping environment is illustrated in Figure 3.8.

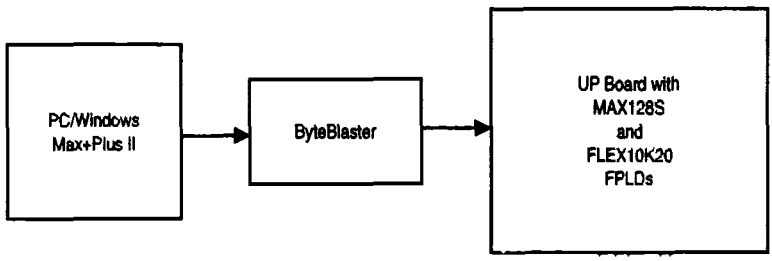



Figure 3.8 Overall UP Prototyping Environment

The Max+Plus II design environment has already been introduced in preceding section. The ByteBlaster represents an interface with download cable that plugs into PC parallel port and enables downloading of configuration bitstreams into two types of FPLDs present on the UP prototyping board: one is a product-based MAX EPM7128S device with built-in in-system programmability feature, and the other one is look-up-table based FLEX10K20 device that uses SRAM for programming purposes. An external EPROM can also be used for an on-board configuration. In this section we provide description of the UP1 board, which has been used in verification of many of examples presented in this book.

The UP board functional organization is presented in Figure 3.9. It contains two parts dedicated to two types of FPLDs present on the board:

- MAX7000S part. This part contains the EPM128S device with 128 macrocells and an equivalent of 2500 gates for designs of medium complexity. The device is suitable for introductory designs which include larger combinatorial and sequential functions. It is mounted on 84-pin socket, and all pins are accessible via on-board connectors. Associated with this device are 2 octal DIP switches, 16 LEDs, dual digit 7-segment display, two momentary push buttons, on-board oscillator with 25.175MHz crystal, and an expansion port with 42 I/O pins and dedicated Global CLR, OE1 and OE2 pins. The switches and LEDs are not pre-wired to the device pins, but they are broken out to female connectors providing flexibility of connections using hook-up wires.

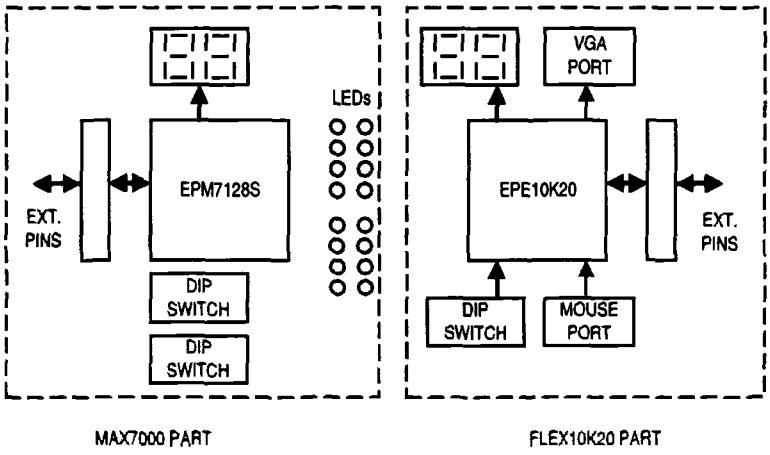



Figure 3.9 UP1 Prototyping board structure

- FLEX10K part. This part contains the EPF10K20-240 device with 1152 logic elements, six embedded array blocks of 2048 bits of SRAM each, and total of 240 pins. With a typical gate count of 20,000 gates, this device is suitable for advanced designs, including more complex computational, communication, and DSP systems. This part contains a socket for an EPC1 serial configuration EPROM, an octal DIP switch, two momentary push buttons, dual digit 7-segment display, on-board oscillator 25.175MHz crystal, a VGA port, a mouse port, and 3 expansion ports each with 42 I/O pins and 7 global pins. The VGA interface allows the FLEX10K device to control an external monitor according to the VGA standard. The FLEX device can send signals to an external monitor through the diode-resistor network and a D-sub connector that are designed to generate voltages for the VGA standard. Information about the color of the screen, and the row and column indexing of the screen are sent from the FLEX device to the monitor via 5 signals (3 signals for red, green and blue, and 2 signals for horizontal and vertical synchronization). With the proper usage of these signals, images can be written to the monitor's screen. The mouse interface allows the FLEX10K device to receive data from a PS/2 mouse or PS/2 keyboard. The FLEX10K device outputs the data clock signal to the external device and receives data signal from the device.

Configuration of the devices on the UP prototyping board is performed simply by selecting menus and options in the Max+Plus II design environment.

## 3.6 Questions and Problems

3.1 What are the major steps in a digital system design process? Describe the role of each of the tools used in the design process.

3.2 What is a netlist? Generate a netlist for a 4-to-1 multiplexer described using

a) elementary two-input logic gates
b) logic elements that contain only 3-input/single-output look-up tables

3.3 Repeat preceding example for 8-to-1 multiplexer.

3.4 Repeat preceding example for BCD to seven-segment encoder.

3.5 A hexadecimal keypad encoder is a circuit that drives rows and scans columns of a mechanical hexadecimal keypad. Decompose the encoder circuit as a hierarchy of other lower complexity circuits and represent it in the form of hierarchy tree.

3.6 Describe basic features of functional and timing simulation.

3.7 Analyze the operation of Altera UP1 prototyping board from its data sheet (available on Altera's web site www.altera.com). Design a circuit that enables demonstration of input and output functions using switches, push buttons and 7-segment displays.

3.8 Analyze the operation of VGA interface on Altera UP1 prototyping board. Design a circuit that enables the access to a VGA monitor.

3.9 Analyze the operation of mouse interface on Altera UP1 prototyping board. Design a circuit that provides data input from a mouse.

3.10 Using circuits from 3.8 and 3.9 design a circuit that will enable interaction between the user and VGA monitor using mouse.

# 4 INTRODUCTION TO DESIGN USING AHDL

This chapter presents the basics of the design using Altera's Hardware Description Language (AHDL). The basic features of AHDL are introduced without a formal presentation of the language. Small examples are given to illustrate its features and usage. The design of combinatorial logic in AHDL including the implementation of bidirectional pins, standard sequential circuits such as registers and counters, and state machines is presented. The implementation of user designs as hierarchical projects consisting of a number of subdesigns is also shown. The more advanced features of AHDL are presented in Chapter 5.

## 4.1. AHDL Design Entry

The Altera Hardware Description Language (AHDL) is a high level, modular language especially suited for complex combinatorial logic, group operations, state machines, and truth tables. AHDL Text Design Files (TDFs with extension .tdf) can be entered using any text editor, and subsequently compiled and simulated, and are used to program Altera FPLDs. However, the text editor within the Max+Plus II environment provides AHDL templates and helps the designer especially in the early stage of learning the language.

AHDL allows a designer to create hierarchical designs (projects) which also incorporate other types of design files. A symbolic representation of a TDF entered design is automatically created upon compilation and synthesis and can be incorporated into a Graphic Design File (.gdf). Also, user custom functions, as well as Altera provided macrofunctions and megafunctions can be incorporated into any TDF. Altera provides Include Files (.inc) with function prototypes for all provided functions in the macrofunction and megafunction library. A hierarchical project can contain TDFs, GDFs, and EDIF Input Files (.edf) at any level of the project hierarchy. Waveform Design Files (.wdf), Altera Design Files (.adf), and State Machine Files (.smf), which provide compatibility with earlier Altera design tools, can be used only at the lower level of a project hierarchy. Any new design can be an AHDL design and is treated as a new project, which may include hierarchy of other

designs transformed into components after their compilation. Figure 4.1 illustrates typical project hierarchy.

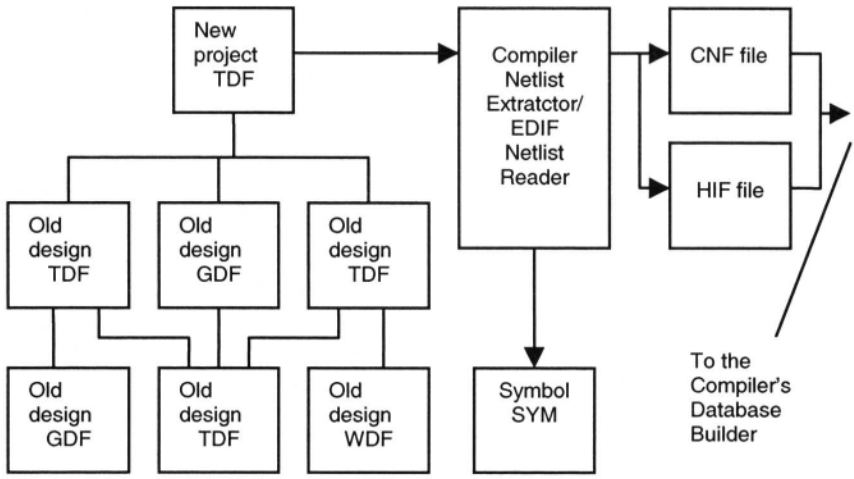



Figure 4.1 AHDL Design Entry and Relationship to Max+Plus II Design Environment

### *4.1.1 AHDL Design Structure*

A new TDF design contains one or more separate parts called sections and statements. A TDF must contain either a Design Section or a Subdesign Section/Logic Section combination, or both, while the other sections and statements are optional. The sections and statements that appear in TDF (in the order of appearance) are:

- **Title Statements** (Optional) provide comments for the Report File (.rpt) generated by the Max+Plus II Compiler.
- **Constant Statements** (optional) specify a symbolic name that can be substituted for a constant.
- **Function Prototype Statements** (optional) declare the ports of a macrofunction or primitive and the order in which these ports must be declared in any in-line reference.
- **Define Statement** (optional) defines an evaluated function, which is a mathematical function that returns a value that is based on optional argument.

- **Parameters Statement** (optional) declares one or more parameters that control the implementation of a parameterized functions. A default value can be specified for each parameter.
- **Include Statements** (optional) specify an Include File that replaces the Include Statement in the TDF.
- **Options Statements** (optional) set the Turbo and Security Bits of Altera devices and specifies logic options and logic synthesis styles. This statement can be placed before the Design Section, inside the Design Section, or inside the Device Specification. In the newer versions of the Max+Plus II environment, various options are not specified in TDF, but rather they are set using specialized menus and windows for that purpose.
- **Design Sections** (required) specifies pin, buried logic cell, chip, clique, logic option, and device assignments, as well as the placement of design logic. It also describes which pins are wired together on the board. The design section is required if it is the only section in the TDF.
- **Assert Statement** (optional) allows the designer to test validity of an arbitrary expression and report the results.
- **Subdesign Sections** (required) declare the input, output, and bidirectional ports of an AHDL TDF. This section is required unless the TDF consists of a Design Section only.
- **Variable Sections** (optional) declare variables that represent and hold internal information.
- **Logic Sections** (required) define the logical operations of the file. This section is required unless the TDF consists of a Design Section only.

### *4.1.2 Describing Designs with AHDL*

Although AHDL looks in its appearance and syntax like a programming language, there are many features that differentiate it significantly from a programming language. The most important one is that AHDL is a concurrent language. All behavior specified in the Logic Section of a TDF is evaluated at the same time. Equations that assign multiple values to the same AHDL node or variable are logically connected (ORed if the node or variable is active high, and ANDed if it is active low). The Design Section contains an architectural description of the TDF. The last entries in the TDF, the Subdesign Section, Variable Section (optional), and Logic Section, collectively contain the behavioral description of the TDF.

If used, already present macrofunctions and megafunctions are connected through their input and output ports to the design file at the next higher level of the hierarchy. The contents of the Include File, an ASCII file, are substituted wherever an include statement is found in the TDF. It is recommended to include only constants or function prototype statements in the Include File.

When the TDF is entered using a text editor, its syntax can be checked with the Save & Check command, without full compilation that includes synthesis for the assigned FPLD device, or all files can be compiled in a project with the Save & Compile command. The Max+Plus II compiler automatically generates a symbol for the current file, which can be used in GDF. Optionally, Include File corresponding to the new design (prototype function of the new design) can be generated and used as the user defined function. After the project has compiled successfully, you can perform optional design verification using simulation and timing analysis, and then program one or more devices.

## 4.2. AHDL Basics

The Altera Hardware Description Language is a text entry language for describing logic designs. It is incorporated into the Max+Plus II design environment and can be used as a sole design entry tool or together (in combination) with the other design entry tools including other hardware description languages (VHDL and Verilog). AHDL consists of a variety of elements that are used in behavioral statements to describe logic. The following sections introduce the basic features of AHDL by examples of small designs.

### *4.2.1 Using Numbers and Constants*

Numbers are used to specify constant values in Boolean expressions and equations. AHDL supports all combinations of decimal, binary, octal, and hexadecimal numbers.

Example 4.1 present an address decoder that generates an active-high chip enable when the address is FF30 (Hex) or FF50 (Hex) present on the input.

**Example 4.1 Address Decoder**

```
SUBDESIGN decode
(
   address[15..0]  :INPUT;
```

```
    chip_enable1,chip_enable2      :OUTPUT;
)

BEGIN
    chip_enable1 = (address[15..0] == H"FF30"); chip_enable2 =
(address[15..0    ] == H"PF50");
END;
```

The address decoder TDF description consists of two sections:

- Subdesign Section, which describes the input and output ports and specifies their names that can be later referred to when using this address decoder. The subdesign also has its name that can be any identifier following the naming conventions of AHDL. In subsequent designs the current design can be referred to by its name and the names of the inputs and outputs.

- Logic Section, which describes the operation of the address decoder using assignment statements and relational expressions (comparison of the input address with the specified value).

The decimal numbers 15 and 0 are used to specify bits of the address bus. The hexadecimal numbers H"FF30" and H"FF50" specify the addresses that are decoded. Similarly, address can be specified using binary number, such as B"1111111100110000". Obviously, the designer can decide when to use specific number representations depending of what is described with it.

Example design can be stored in the TDF. The equivalent GDF file of this address decoder is shown in Figure 4.2 and it represents the symbol that will be generated after design compilation. As such, this symbol can be used subsequently in GDF design entry.

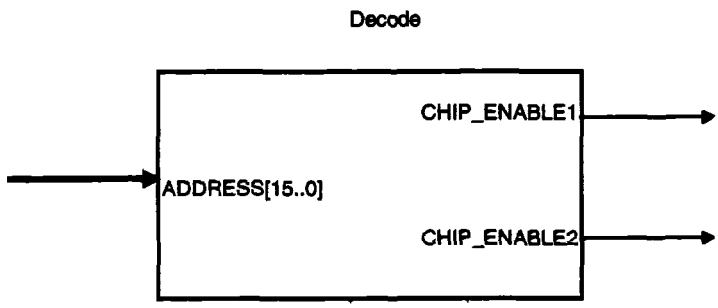



Figure 4.2 GDF equivalent of the decoder circuit

Constants can be used to give a descriptive name to a number. This name can be used throughout a design description. In the case that the change of the value of a constant is needed, it is done at only one place, where constant is declared.

In Example 4.1 above, we can introduce the constants IO_ADDRESS1 and IO_ADRESS2 to describe the addresses that are to be decoded. The new TDF description is shown in Example 4.2.

Example 4.2 TDF of Modified Decoder.

```
CONSTANT IO_ADDRESS1 = H"FF30";
CONSTANT IO_ADDRESS2 = H"FF50";

SUBDESIGN decode1
 (
   a[15..0]   :INPUT;
   cel, ce2   :OUTPUT;
)

BEGIN
   cel = (a[15..0] == IO_ADDRESS1);
   ce2 = (a[15..0] == IO_ADDRESS2);
END;
```

Constants can be declared using arithmetic expressions, which include the other already declared constants. The Compiler evaluates arithmetic expressions and replaces them by numerical values before further analysis of the description. For example, we can declare constant:

```
CONSTANT  IO_ADDRESS1 = H"FF30";
```

and then

```
CONSTANT  IO_ADDRESS2 = IO_ADDRESS1 + H"0010";
CONSTANT  IO_ADDRESS3 = IO_ADDRESS1 + H"0020";
```

using address H"FF30" as a base address, and generating other addresses relative to the base address.

Another example of using constants is to substitute the values for parameters during compilation. The design can contain one or more parameters whose values are replaced with actual values at the compilation time. For example, the target device family can be a parameter, and the actual value of parameter can be substituted by a constant:

```
PARAMETERS
 (
   DEVICE_FAMILY
);

CONSTANT FAMILY1 = "MAX7000";
CONSTANT FAMILY2 = "FLEX8000";
```

whic h is further used within the Subdesign Section to compile the design for a specific device family depending on the value of DEVICE_FAMILY parameter (which can be FAMILY1 or FAMILY2). The use of parameters helps to make more general component designs that can be customized at the time of use of specific component. The use of parameters will be discussed in more details in Chapter 5 in conjunction with some more advanced features of AHDL.

### *4.2.2 Combinational Logic*

Two types of circuits are designed in typical digital systems: combinational (or combinatorial) and sequential circuits. Current outputs of a combinational circuit depend only on the current values of the inputs, while in sequential circuits they depend also on the previous values of the inputs (history). As combinational circuits are found as a part of sequential circuits, we will first show how they are described in AHDL.

Combinatorial logic is implemented in AHDL with Boolean expressions and equations, truth tables, and a variety of macrofunctions. Boolean expressions are sets of nodes, numbers, constants, and other Boolean expressions separated by operators and/or comparators, and optionally grouped with parentheses. A Boolean equation sets a node or group equal to the value of a Boolean expression. Example 4.3 shows simple Boolean expressions that represent logic gates.

**Example 4.3 Boolean Expressions for Logic Gates.**

```
SUBDESIGN booll
 (
   a0, a1, b0, b1  :INPUT;
   s0, s1   :OUTPUT;
)

BEGIN
   s0 = a0 & a1 & !b1;
   s1 = s0 # b0;
END;
```

Since two logic equations used in Logic Section are evaluated concurrently, their order in the TDF description above is not important. This again emphasizes concurrent nature of AHDL and the need for departure from thinking that AHDL is a programming language. The GDF equivalent of the above TDF is shown in Figure 4.3.

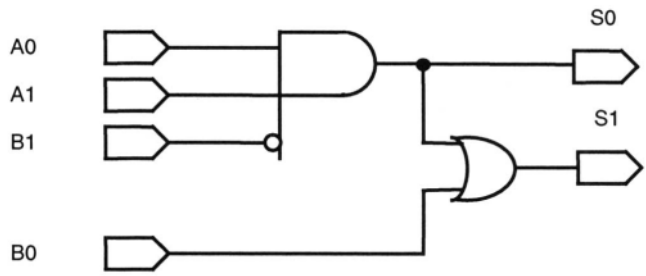



Figure 4.3 GDF Representation of the Circuit

### *4.2.3 Declaring Nodes*

Besides describing inputs and outputs through which the design communicates with external world in Subdesign Section, AHDL allows to declare internal signals within the design and use them to simplify design descriptions. The internal signals in AHDL are called nodes, and they are not accessible by other designs that will use the design being described as its component.

A node is declared with a node declaration in the Variable Section. It can be used to hold the value of an intermediate expression. Node declarations are useful when a Boolean expression is used repeatedly. The Boolean expression can be replaced with a more descriptive node name. Example 4.4 below performs the same function as the former one, but it uses a node declaration which saves device resources if repeatedly used.

Example 4.4 Boolean Expressions with Node Declarations.

```
SUBDESIGN bool2
 (
    a0, b0, a1, b1  :INPUT;
    s1   :OUTPUT;
)
VARIABLE
    Inter   :NODE;
BEGIN
```

```
    inter = a0 & a1 & !b1;
    s1 = inter # b0;
END;
```

GDF equivalent to this TDF is shown in Figure 4.4.

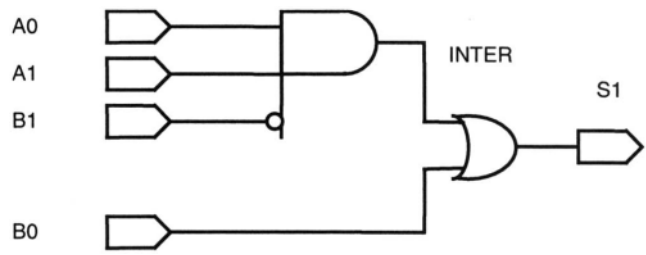



Figure 4.4 GDF Representation of the Node Declared Circuit

The other important use of nodes is when interconnecting already existing components in hierarchical TDF descriptions. This use of nodes will be shown in the following sections and chapters.

### *4.2.4 Defining Groups*

A group, which can include up to 256 members (bits), is treated as a collection of input or output signals or nodes and is acted upon as one unit. In Boolean equations, a group can be set equal to a Boolean expression, another group, a single node, Vcc, GND, 1, or 0. In each case, the value of the group is different. Once the group has been defined, [ ] is a shorthand way of specifying the entire range. A subgroup is represented using a subrange of the indices. The Example 4.5 shows a number of legal group values and assignments.

Example 4.5 Legal Group Values.

```
a[7..0] = b[15..8];        % a7 connected to b15, a6 to b14,...%
a[2..0] = inter;    % all bits connected to inter %
a[7..0] = Vcc;      % all bits connected to Vcc %
```

This example also introduces comments that are specified by using "%" characters to enclose the text that represents comment.

The Options Statement can be used to specify the most significant bit (MSB) or the least significant bit (LSB) of each group. For example

```
OPTIONS BIT0 = MSB;
```

or

```
OPTIONS BITO = LSB;
```

specify the lowest numbered bit (bit 0) to be either MSB or LSB, respectively.

### *4.2.5 Conditional Logic*

Conditional logic chooses from different behaviors depending on the values of the logic inputs. AHDL provides two statements for conditional logic implementation, the IF statement and the Case statement.

- **IF statements** evaluate one or more Boolean expressions, then describe the behavior for different values of the expression. IF statement can be in the simple IF THEN form or in any variant of IF THEN ELSIF... ELSE forms.
- **Case statements** list alternatives that are available for each value of an expression. They evaluate expression, then selects a course of action on the basis of the expression value.

Example 4.6 represents the use of IF Statement in a priority encoder. It is given together with the truth table describing the function of the encoder (Table 4.1). Don't care conditions are described by x. While the design of this encoder using traditional methods may represent a challenge for a higher number of inputs, its description in AHDL is straightforward.

Example 4.6 IF Statement Use.

The inputs `prior4`, `prior3`, `prior2`, and `prior1` are evaluated to determine whether they are driven by Vcc. The IF Statement activates the equations that follow the highest priority IF or ELSE clause that is active. Output priority code is represented by 3-bit value. If no input is active, then the output code 0 is generated.

```
SUBDESIGN priority
 (
    prior4,prior3,prior2,prior1    :INPUT;
    prior_code[2..0]               :OUTPUT;
)
BEGIN

IF prior4 THEN
    prior_code[] = 4;
ELSIF prior3 THEN
    prior_code[] = 3;
ELSIF prior2 THEN
    prior_code[] = 2;
ELSIF prior1 THEN
    prior_code[] = 1;
ELSE
    prior_code[] = 0;
ENDIF;
END;
```

Table 4.1 truth table for four input priority encoder

| prior | | | | prior_code |
|---|---|---|---|---|
| 4 | 3 | 2 | 1 | (decimal) |
| 1 | x | x | x | 4 |
| 0 | 1 | x | x | 3 |
| 0 | 0 | 1 | x | 2 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 |

While encoders compress individual information into corresponding codes, decoders have the opposite role. Example 4.7 shows the use of a Case Statement in specifying the 2-to-4 decoder that converts two bit code into "one hot" code. The expression (in this case just input code) is matched against a number of constant values and appropriate action activated.

Example 4.7 Case Statement Use

```
SUBDESIGN decoder_2_to_4
 (
    inpcode[1..0]     :INPUT;
    outcode[3..0]     :OUTPUT;
)
BEGIN

CASE inpcode[ ]  IS
    WHEN 0 => outcode[ ]  = B"0001";
    WHEN 1 => outcode[ ]  = B"0010";
    WHEN 2 => outcode[ ]  = B"0100";
    WHEN 3 => outcode[ ]  = B"1000";

END CASE;
END;
```

The input group inpcode[1..0] may have the value 0, 1, 2, or 3. The equation following the appropriate => symbol is activated.

It is important to note that besides similarities, there are also differences between IF and Case Statements. Any kind of Boolean expression can be used in an IF Statement, while in a Case Statement, only one Boolean expression is compared to a constant in each WHEN clause.

### *4.2.6 Decoders*

A decoder contains combinatorial logic that converts input patterns to output values or specifies output values for input patterns. Very often the easiest way to describe mapping of input to output patterns is by using truth tables. AHDL Truth Table Statements can be used to create a decoder. This is illustrated in Example 4.8.

Example 4.8 Truth Table Decoder.

```
SUBDESIGN decoder
 (
    inp[1..0]   :INPUT;
    a, b, C, d  :OUTPUT;
)
```

```
BEGIN

TABLE
     inp[l..0] => a, b, c, d;

     H"0" => 1, 0, 0, 0;
     H"1" => 0, 1, 0, 0;
     H"2" => 0, 0, 1, 0;
     H"3" => 0, 0, 0, 1;
END TABLE;
END;
```

The Truth Table Statement contains a header that describes which inputs are mapped to which outputs and in what order, and a number of rows that specify mapping of the input to output patterns. In Example 4.8, the output pattern for all four possible input patterns of inp[1..0] is described in Truth Table Statement.

In the case that decoder is partial one (not decoding all possible input combinations), the Default Statement can be used to specify the output of the decoder when not-specified values of input appear as shown in the following example:

```
SUBDESIGN partial_decoder
 (
    inpcode[3..0]:   INPUT;
    outcode[4..0]: OUTPUT;
)

BEGIN
DEFAULTS
    outcode[]= B"11111";     %value of output for%
                             %unspecified input codes%
END DEFAULTS;

TABLE
    inpcode [] => outcode[];

    B"0001"   => B"01000";
    B"0011"=> B"00100";
    B"0111"=> B"00010";
    B"1111"=> B"00001";
END TABLE;
END;
```

Example 4.9 represents an address decoder for a generalized microcomputer system with 16-bit address. The decoder decodes a number of specific, partially specified addresses to provide select signals for the parts of microcomputer system,

such as RAM, ROM and peripherals. The inputs to the decoder are address lines and m/io signal that specifies whether the access is to memory or input/output (peripheral) devices.

Example 4.9 Address Decoder for a 16-Bit Microprocessor.

```
SUBDESIGN decode2
(
   addr[15.. 0] , m/io  :INPUT;
   rom, ram, print, inp[2..1]   :OUTPUT;
)

BEGIN

TABLE
   m/io, addr[15..0]      => rom, ram, print, inp[ ];
   1, B"00xxxxxxxxxxxxxxxx" => 1, 0, 0, B"00";
   1, B"10xxxxxxxxxxxxxxxx" => 0, 1, 0, B"00";
   0, B"000001010000000000" => 0, 0, 1, B"00";
   0, B"000011010000010000" => 0, 0, 0, B"01";
END TABLE;
END;
```

Instead of specifying all possible combinations, we can use x for "don't care" to indicate that output does not depend on the input corresponding to that position of x.

### *4.2.7 Implementing Active-Low Logic*

An active-low control signal becomes active when its value is GND. An example of the circuit that uses active-low control signals is given in Example 4.10. The circuit performs daisy-chain arbitration. This module requests the bus access of the preceding module in the daisy chain. It receives requests for bus access from itself and from the next module in the chain. Bus access is granted to the highest-priority module that requests it.

Example 4.10 Active Low Daisy-Chain Arbitrator.

```
SUBDESIGN daisy-chain
(
   /local_request    :INPUT;
   /local_grant      :OUTPUT;
   /request_in       :INPUT;  %from lower prior%
   /request_out      :OUTPUT; %to higher prior%
   /grant_in         :INPUT;  %from higher prior%
   /grant_out        :OUTPUT;  %to lower prior%
)
```

```
BEGIN

DEFAULTS
    /local_grant = Vcc; %active-low output%
    /request_out = Vcc; %signals should%
    /grant_out = Vcc;   %default to Vcc%
END DEFAULTS;

IF /request_in == GND # /local_request == GND THEN
    /request_out = GND;
END IF;

IF /grant_in == GND THEN
    IF /local_request == GND THEN
     /local_grant = GND;
    ELSIF /request_in == GND THEN
     /grant_out = GND;
    END IF;
END IF;
END;
```

All signals in Example 4.10 are active-low. It is recommended to indicate the signal is active low by using some indication, in this case a slash ("/"), as part of the signal name. The Defaults Statements in the example specify that a signal is assigned to Vcc when it is not active.

### *4.2.8 Implementing Bidirectional Pins*

AHDL allows I/O pins in Altera devices to be configured as bidirectional pins. Bidirectional pins can be specified with a BIDIR port type that is connected to the output of a TRI primitive. The signal between the pin and tri-state buffer is a bidirectional signal that can be used to drive other logic in the project. It should be noted that the bidirectional ports can be implemented only on external pins of Altera FPLDs.

Example 4.11 below shows an implementation of a register that samples the value found on a tri-state bus. It can also drive the stored value back to the bus. The bidirectional I/O signal, driven by TRI, is used as the D input to a D flip-flop (DFF). Commas are used to separate inputs to the D flip-flop, including placeholders for the CLRN and PRN signals that default to the inactive state.

Example 4.11 Bus Register.

```
SUBDESIGN bus_reg
(
   clk : INPUT;
   oe : INPUT;
   io  : BIDIR;
)

BEGIN
   io = TRI(DFF(io, clk, , ), oe);
END;
```

A GDF equivalent to bus_reg example is shown in Figure 4.5.

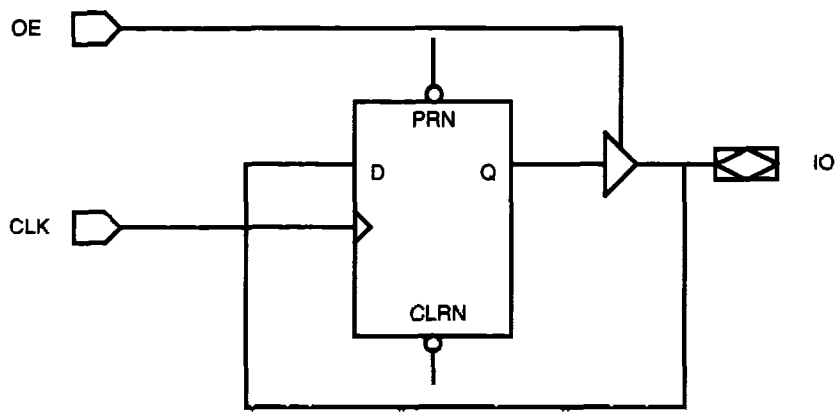



Figure 4.5 Example of a Bidirectional Pin

It is also possible to connect a bidirectional pin from a lower-level TDF to a top-level pin. The bidirectional port of the macrofunction should be assigned to a bidirectional pin. Example 4.12 shows the use of four instances of the bus_reg macrofunction.

Example 4.12 Bidirectional 4-Bit Port.

```
TITLE "bidirectional 4-bit port";
FUNCTION bus_reg (clk, oe) RETURNS (io) ;

SUBDESIGN bidir
(
   clk, oe  :INPUT;
   io[3..0]   :BIDIR;
)
```

```
BEGIN
    io0 = bus_reg(clk, oe) ;
    io1 = bus_reg(clk, oe);
    io2 = bus_reg(clk, oe) ;
    io3 = bus_reg(clk, oe) ;
END;
```

The instances of `bus_reg` are used in-line in the corresponding AHDL statements and this resembles to functional calls in programming languages. Actually, it describes mapping of corresponding inputs to the outputs of the component as specified by function of the component. More details on in-line referencing to already available components will be presented in the following sections.

With this section we have covered basics of combinational circuits design. In the examples and case studies in the following sections we will show how to describe frequently used standard combinational circuits, but also those that are customized for specific applications.

## 4.3 Designing Sequential logic

Sequential logic is usually implemented in AHDL with standard circuits such as registers, latches, and counters, or with non-standard ones represented by finite state machines. As their outputs depend not only on current values of inputs but also on their past values, they must include one or more memory elements. Those are usually flip-flops, but the other types of memory can be used.

### *4.3.1 Declaring Registers and Registered Outputs*

Registers are used to store data values, hold count values, and synchronize data with a clock signal. Registers can be declared with a register declaration in the Variable Section of TDF description. A port (input or output) of an instance of already available component can be used to connect an instance of a primitive, macrofunction, or state machine to other logic in a TDF description. A port of an instance uses the format:

```
Instance_name.Port_name
```

The Port_name is an input or output of a primitive, macrofunction, or state machine, and is synonymous with a pin name in the GDF. Example 4.13 contains a byte register that latches values of the `d` inputs onto the `q` outputs on the rising edge of the Clock when the load input is high.

Example 4.13 Byte Register Design.

```
SUBDESIGN register
 (
    clock, load, d[7..0]   : INPUT;
    q[7..0]   : OUTPUT;
)

VARIABLE
    ff [7..0]   : DFFE;

BEGIN
    ff [ ].clk = clock;
    ff [ ].ena = load;
    ff [ ].d = d[ ];
    q[ ] = ff[ ].q;
END;
```

All four statements in the Logic Section of the subdesign are evaluated concurrently (at the same time). The variable Section contains declaration (and instantiation) of eight flip-flops, ff, which are of the DFFE type. The DFFE flip-flop is a standard AHDL primitive representing D flip-flop with an enable input as those used in a GDF equivalent to the above TDF. It is shown in Figure 4.6.

Instead of D flip-flops, other types of flip-flops can be declared in the Variable Section. Various types of flip-flops are supported in AHDL and referred to as primitive components. The whole list of primitives is shown in Table 5.7 of Chapter 5.

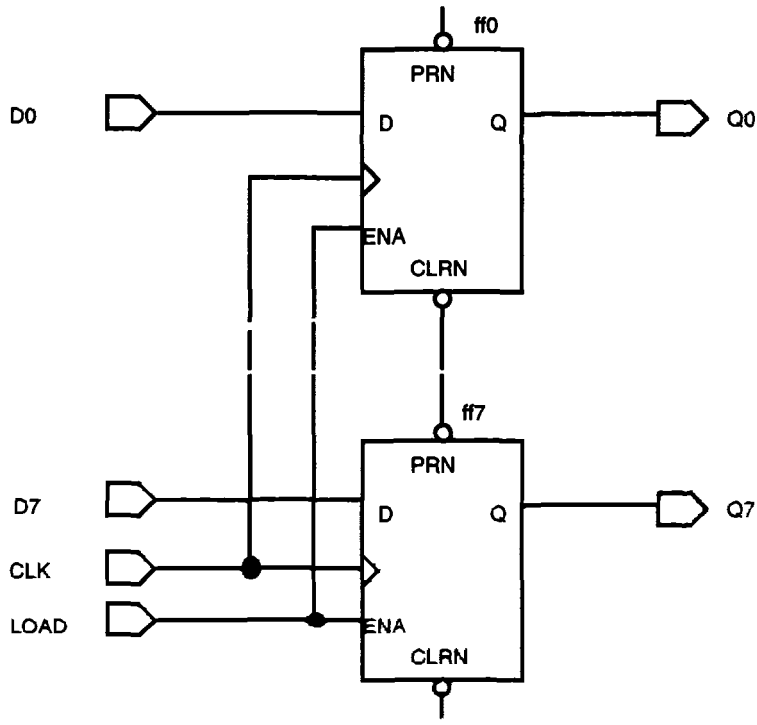



Figure 4.6 GDF of the Example 4.13 Register

Registered outputs of a subdesign can be declared as D flip-flops in the Variable Section. Example 4.14 is similar to the previous one, but has registered outputs.

Example 4.14 Registered Output Byte Register.

```
SUBDESIGN reg_out
 (
    clk, load, d[7..0]       :INPUT;
    q[7..0]   :OUTPUT;
)
VARIABLE
    q[7..0]   :DFFE; %also decl.as output%
BEGIN
    q[].clk = clk;
    q[].ena = load;
    q[] = d[];
END;
```

Each Enable D flip-flop declared in the Variable Section feeds an output with the same name, so it is possible to refer to the q outputs of the declared flip-flops without using the q port of the flip-flops. The register's output does not change until the rising edge of the Clock. The Clock of the register is defined using

```
<output_pin_name>.clk
```

for the register input in the Logic Section. A global Clock can be defined with the GLOBAL primitive.

### *4.3.2 Creating Counters*

Counters use sequential logic circuits to count Clock or other pulses. Counters are usually defined with D flip-flops and IF Statements. Example 4.15 shows a 16-bit loadable up counter that can be cleared to zero.

Example 4.15 16-Bit Loadable Up Counter.

```
SUBDESIGN loadable_counter
(
   clock, load, enable, reset, d[15..0]   :INPUT;
   q[15..0]   :OUTPUT;
)

VARIABLE
   count[15..0]   :DFF;

BEGIN
   count[].clk = clock;
   count[].clrn = !reset; %low signal active%

IF load THEN
   count[].d = d[];
ELSIF enable THEN
   count [] .d = count [] .q + 1;
ELSE
   count [] .d = count [] .q;
END IF;

    q[] = count [] ;

END;
```

In this example, 16 D flip-flops are declared in the Variable Section and assigned the names `count0` through `count15`. The IF Statement determines whether the value present on the data input lines is loaded into the flip-flops on the rising Clock edge, or counter increment its value. In the case that neither load nor enable inputs are active, counter stays in its previous state. Reset signal is asynchronous to the clock and initializes counter when activated.

### *4.3.3 Finite State Machines*

Finite State Machines (FSMs) represent an important part of design of almost any more complex digital system. They are used to sequence specific operations, control other logic circuits, and provide synchronization of different parts of more complex circuit. FSM is a circuit that is designed to sequence through specific patterns of states in a predetermined manner. Sequences of states through which an FSM passes depend on the current state of the FSM, and previous history of the FSM. A state is represented by the binary value held on the current state register. FSM is clocked from a free running clock source. The general FSM model is presented in Figure 4.7.

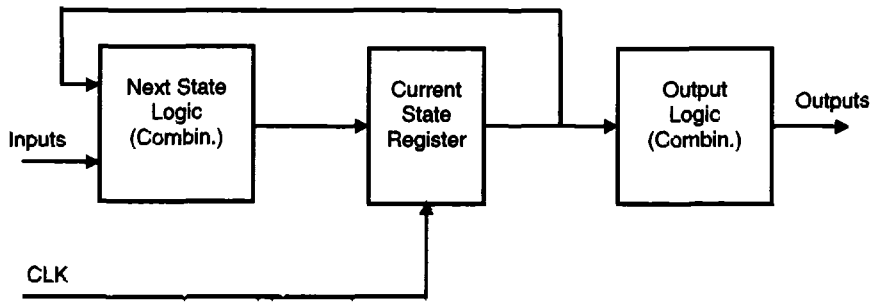



Figure 4.7 General model of FSM

It contains three main parts:

1. Current State Register. It is a register of n flip-flops used to hold the current state of the FSM. The current state is represented by the binary value contained in this register. State changes are specified by sequences of states through which the FSM passes after changes of inputs to the FSM.

2. Next State Logic. It is a combinational logic used to generate the transition to the next state from the current state. The next state is a

function of the current state and the external inputs to the FSM. The fact that the current state is used to generate transition to the next state means that feedback mechanism within the FSM must be used to achieve desired behavior.

3. Output Logic. It is a combinational circuit used to generate output signals from the FSM. Outputs are a function of the current state and possibly FSM's inputs. If the output is a function of only the current state, then we classify the FSM as Moore FSM. If the outputs depend also on the inputs to the FSM, then we classify the FSM as Mealy FSM. Both these types of FSMs are discussed in more details in the following sections. Sometimes, combined Mealy/Moore models are suitable to describe specific behavior.

The behavior of an FSM is usually described either in the form of a state transition table or a state transition diagram. AHDL enables easy implementation of FSMs. The language is structured so that designers can either assign state bits by themselves, or allow the compiler to do the work. If the compiler performs the task, state assignment is done by minimizing the required logic resources. In case of MAX FPLDs the Compiler assigns by default a minimal number of state variables, and therefore flip-flops, to the states. For FLEX FPLDs the Compiler assigns state variables and values to the states using one hot encoding, as the number of flip-flops in these devices is high enough for most applications. The designer first has to specify state machine behavior, draw the state diagram and construct a next-state table. The Compiler then performs the following functions automatically:

- Assigns bits, select a T or D flip-flops to the bits
- Assigns state values
- Applies logic synthesis techniques to derive the excitation equations

The designer is allowed to specify state machine transitions in a TDF description using Truth Table Statement as well. In that case, the following items must be included in the TDF:

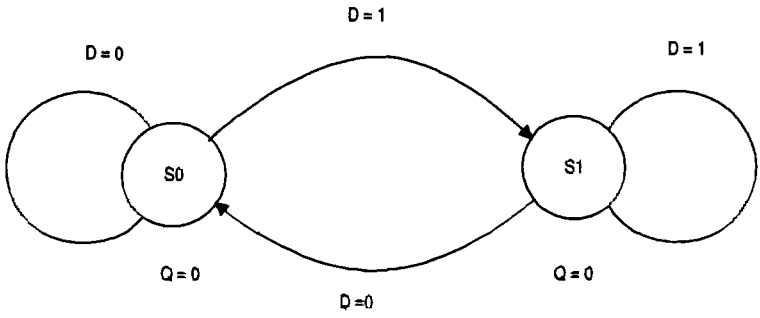
- State Machine Declaration (Variable Section)
- Boolean Control Equations (Logic Section)
- State Transitions (Logic Section)

AHDL machines, once compiled and synthesized, can be exported or imported between TDFs and GDFs, or TDFs and WDFs by specifying an input or output signal as a machine port in the Subdesign Section.

A state machine can be created by declaring the name of the state machine, its states, and, optionally, the state machine bits in the State Machine Declaration of the Variable Section. Example 4.16 represents the state machine with the functionality of a D flip-flop and a state transition diagram as shown below.

The states of the machine are defined as `s0` and `s1`, (can be any valid names) and no state bits are declared. The GDF equivalent to this state machine is shown in Figure 4.8. A number of signals are used to control the flip-flops in the state machine. In more general case, the states are represented by a number of flip-flops that form a state register. In the example above, external clock and reset signals control directly clk and reset inputs of the state machine flip-flop. Obviously, the expression that specifies creation of these signals (on the right hand side of the assignment statement) can be any Boolean expressions. From this example we see that a single Case Statement describes the state transitions.

Example 4.16 State Machine D Flip-Flop.




```
SUBDESIGN d_flip_flop
(
   clock, reset, d  :INPUT;
   q       :OUTPUT;
)
VARIABLE
   ss: MACHINE WITH STATES (s0, s1);
```

```
BEGIN
    ss.clk = clock;
    ss.reset = reset;

CASE 88 IS
     WHEN s0 =>
        q = GND;

        IF d THEN
           ss=s1;
        END IF;
     WHEN s1 =>
        q = Vcc;

        IF !d THEN
           ss=s0;
        END IF;
END CASE;
END;
```

Outputs are associated just with the states (synchronous with states) and depend only on the current state. An output value can be defined with an IF or Case Statement. In our example, output `q` is assigned to GND when state machine `ss` is in state `s0`, and to value Vcc when the machine is in state `s1`. These assignments are made in WHEN clauses of the Case Statement. Output values can also be defined in truth tables.

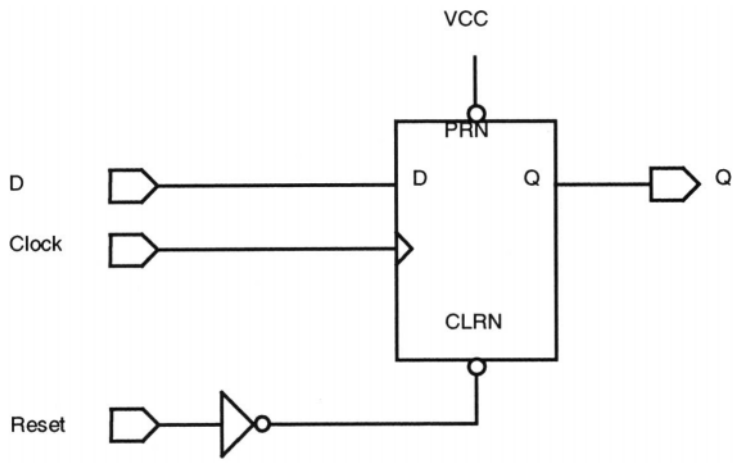



Figure 4.8 GDF Equivalent to the State Machine from Example 4.16

Clock, Reset, and Clock enable signals control the flip-flops of the state register. These signals are specified with Boolean equations in the Logic Section.

In the former example, the state machine Clock is driven by the input clock. The state machine's asynchronous Reset signal is driven by reset, which is active high. To connect the Clock Enable signal in the TDF, we would add the line

```
Enable   :INPUT;
```

to the Subdesign Section and the Boolean equation,

```
ss.ena = enable;
```

to the Logic Section.

State machine transitions define the conditions under which the state machine changes to a new state. The states must be assigned within a single behavioral construct to specify state machine transitions. For this purpose, it is recommended to use Case or Truth Table Statements. The transitions out of each state are defined in WHEN clauses of the Case Statement. However, IF statements can be used to describe transitions, too.

State bits, which represent outputs of the flip-flops used by a state machine, are usually assigned by the Max+Plus II Compiler. However, the designer is allowed to make these assignments explicitly in the State Machine declaration. An example of such an assignment is shown in Example 4.17.

Example 4.17 Direct State Bit Assignment.

```
SUBDESIGN manual_state_assignmnent
 (
   clock, reset, ccw, cw   :INPUT;
   phase[3..0]             :OUTPUT;
)

VARIABLE
   ss: STATE MACHINE OF BITS (phase[3..0])

WITH STATES (
         s0 - B"0001",
         s1 = B"0010",
         s2 = B"0100",
         s3 = B"1000");
```

```
BEGIN
    ss.clk = clock;
    ss.reset = reset;

TABLE
    ss, CCW, CW => ss;

    s0,   1,    x  =>  s3;
    s0,   x,    1  =>  s1;
    s1,   1,    x  =>  s0;
    s1,   x,    1  =>  s2;
    s2,   1,    x  =>  s1;
    s2,   x,    1  =>  s3;
    s3,   1,    x  =>  s2;
    s3,   x,    1  =>  s0;
END TABLE;
END;
```

In Example 4.17, the phase [3 . . 0] outputs declared in the Subdesign Section are also declared as bits of the state machine `ss`. The state assignments are performed manually using one hot codes. State transitions are described using a truth table.

An important issue is ability to bring an FSM to a known state regardless of its current state. This is usually achieved by using (implementing) a reset signal, which can be synchronous or asynchronous. An asynchronous reset ensures that the FSM is always brought to a known initial state, before the next active clock and normal operation resumes. Another way of bringing an FSM to an initial state is to use synchronous reset. This usually requires the decoding of unused codes in the next state logic, because the FSM can be stuck in an uncoded state. AHDL considers the first enumerated state within the State Machine Declaration as an initial state.

### *4.3.4 State Machines with Synchronous Outputs – Moore Machines*

AHDL allows to describe two kinds of state machines. The state machines with the present state depending only on its previous inputs and previous state, and the present output depending only on the present state, are called Moore State Machines. The general structure of Moore-type FSM is presented in Figure 4.9. It contains two functional blocks that can be implemented as combinational circuits:

- next state logic, which can be represented by function next_state_logic, and
- output logic, which can be represented by function output_logic

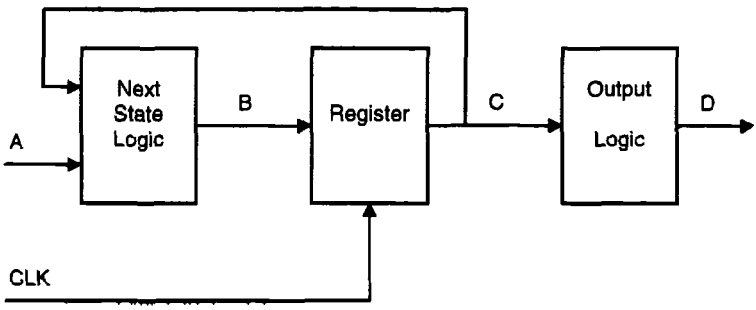



Figure 4.9 Moore-type FSM

Outputs of both of these functions are the functions of their respective current inputs. The third block is a register that holds the current state of the FSM.

Outputs of Moore State Machines can be specified in the `WITH STATES` clause of the State Machine Declaration. The following example implements the Moore State Machine.

Example 4.18 Moore State Machine.

```
SUBDESIGN moore
 (
    clock, reset, y  : INPUT;
    z       : OUTPUT;
)

VARIABLE

    ss: MACHINE OF BITS   (z)  WITH STATES
        (   s0 =       0,
            s1 =       1,
            s2 =       1,
            s3 =       0);

BEGIN
    ss.clk = clk;
    ss.reset = reset;

TABLE
    %current  current       next %
    %state    input         state %
       ss,    y      =>     ss;
```

```
      s0,   0          =>     s0;
      s0,   1          =>     s2;
      s1,   0          =>     s0;
      s1,   1          =>     s2;
      s2,   0          =>     s2;
      s2,   1          =>     s3;
      s3,   0          =>     s3;
      s3,   1          =>     s1;
END TABLE;
END;
```

The state machine is defined with a State Machine declaration. The state transitions are defined in a next-state table, which is implemented with a Truth Table Statement. In this example, machine `ss` has four states and only one state bit `z` is assigned in advance. The Compiler automatically adds another bit and makes appropriate assignments to produce a four-state machine. When state values are used as outputs, as in example above, the project may use fewer logic cells, but the logic cells may require more logic to drive their flip-flop inputs.

Another way to design state machines with synchronous outputs is to omit state value assignments and to explicitly declare output flip-flops. This method is illustrated in Example 4.19.

Example 4.19 Moore Machine with Explicit Output D Flip-Flops.

```
SUBDESIGN moore
 (
    clock, reset, y  : INPUT;
    z   : OUTPUT;
)

VARIABLE
    ss: MACHINE WITH STATES  (s0,s1,s2,S3);
    zd : NODE;

BEGIN
    ss.clk = clock;
    ss.reset = reset;

    z = DFF(zd, clk, Vcc, Vcc);
```

```
TABLE
    %current   current       next          next %
    %state     input         state         output %
       ss,     y      =>     ss,           zd;

       s0,     0      =>     s0,           0;
       s0,     1      =>     s2,           1;
       s1,     0      =>     s0,           0;
       s1,     1      =>     s2,           1;
       s2,     0      =>     s2,           1;
       s2,     1      =>     s3,           0;
       s3,     0      =>     s3,           0;
       s3,     1      =>     s1,           1;
END TABLE;
END;
```

This example includes a "next output" column after the "next state" column in the Truth Table Statement. This method uses a D flip-flop, called with an in-line reference, to synchronize the outputs with the Clock.

### *4.3.5 State Machines with Asynchronous Outputs – Mealy Machines*

A Mealy FSM has outputs that are a function of both the current state and primary system inputs. The general structure of the Mealy-type FSM is presented in Figure 4.10.

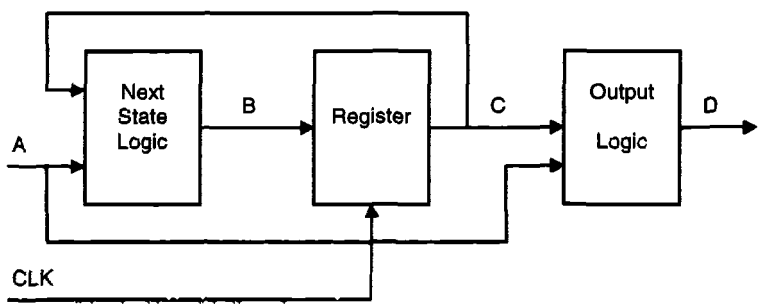



Figure 4.10 Mealy-type FSM

AHDL supports implementation of state machines with asynchronous outputs. Outputs of Mealy State Machines may change when inputs change, regardless of Clock transitions. Example 4.20 shows a state machine with asynchronous outputs.

Example 4.20 State Machine with Asynchronous Outputs.

```
SUBDESIGN mealy
(
   clock, reset, y  : INPUT;
   z       : OUTPUT;
)

VARIABLE
   ss: MACHINE WITH STATES (s0, s1, s2, s3) ;

BEGIN
   ss.clk = clock;
   ss.reset = reset;

TABLE
   %current   current       current       next%
   %state     input         output        state%
      ss,     y      =>     z,            ss;

      s0,     0      =>     0,            s0;
      s0,     1      =>     1,            s1;
      s1,     0      =>     0,            s1;
      s1,     1      =>     1,            s2;
      s2,     0      =>     0,            s2;
      s2,     1      =>     1,            s3;
      s3,     0      =>     0,            s3;
      s3,     1      =>     1,            s0;
END TABLE;
END;
```

### *4.3.6 More Hints for State Machine Description*

Let us consider a simple state machine that describes a modulo-4 counter that can count in two directions depending on the value of input control line up_down. The output from the counter equals its current state. The counter can be described by the AHDL description that implements Moore state machine given in Example 4.21.

Example 4.21 Up_down counter implemented as Moore FSM

```
SUBDESIGN moore_counter
(
   clock, reset, up_down  :INPUT ;
   out[1..0]   : OUTPUT;
)
```

```
VARIABLE
   moore  :MACHINE WITH STATES (S0,S1,S2,S3) ;

BEGIN
   moore.clk = clock;
   moore.reset = reset;

CASE moore IS
   WHEN S0 =>
      out[] = B"00";
    IF up_down THEN
      moore = S1;
    ELSE
      moore = S3;
   END IF;

   WHEN S1 =>
      OUt[] = B"01";
    IF up_down THEN
      moore = S2;
    ELSE
      moore = S0;
    END IF;

   WHEN S2 =>
      out[] = B"10";
    IF up_down THEN
      moore = S3;
    ELSE
      moore = S1;
    END IF;

   WHENS3=>
      out[] = B"11";
    IF up_down THEN
      moore = S0;
    ELSE
      moore = S2;
    END IF;

   WHEN OTHERS =>
      moore = S0;

END CASE;
END;
```

From this description we see that the counter FSM is described using a single case statement. WHEN clause is assigned to each state, and the output of the counter (binary coded state) depend only on the current state. State transitions are

described with IF-THEN-ELSE statements and they depend on the value of the input up_down. This model actually combines next state logic and output logic generation (see Figure 4.8) into a single statement.

The same counter can be described in a different form by grouping next logic and output generation into the separate CASE statements like below (Subdesign and Variable sections not shown):

```
BEGIN

CASE moore IS

    WHEN S0 =>
     IF up_down THEN
        moore = S1;
     ELSE
        moore = S3;
    END IF;

    WHEN S1 =>
     IF up_down THEN
        moore = S2;
     ELSE
        moore = S0;
     END IF;

    WHEN S2 =>
     IF up_down THEN
        moore = S3;
     ELSE
        moore = S1;
     END IF;

    WHEN S3 =>
     IF up_down THEN
        moore = S0;
     ELSE
        moore = S2;
     END IF;

    WHEN OTHERS =>
        moore = S0;

END CASE;

CASE moore IS

    WHEN S0 =>
```

```
        out[]  = B"00";
    WHEN S1 =>
        out[]  = B"01";
    WHEN S2 =>
        out[]  = B"10";
    WHEN S3 =>
        out[]  = B"11";
END CASE;
END;
```

This model exactly follows the one from Figure 4.8 and nicely separates next state logic and output generation into two concurrent conditional statements. WHEN OTHERS clause used in this example shows another good feature of CASE statement. If the machine comes into any state other than four states shown (don't care states), it will be returned automatically into the state s0 even after initialization failures.

When compiling a state machine description for a FLEX architecture the state machine will use one-hot encoding and consume exactly one flip-flop for each state. The WHEN OTHERS statement is not really necessary in this case, because there are no other possible legal states. When compiling the same machine for a MAX architecture, the state machine will use binary encoding. The number of flip-flops used for state representation will be such that they can be used to represent more states than necessary (for example, 5 states require 3 flip-flops and there will be a total of 8 states possible including the undefined states). This is where the WHEN OTHERS clause is really useful, to account for the unused states and make a recovery from these states possible. However, in this case the designer will need to define undefined states to be able to recover from them. This requires changing the declaration of the state machine and introducing the names of unused or undefined states in the state machine declaration.

Recovering from illegal states in one-hot encoded state machines is even more difficult as it is impossible to define illegal states and be able to recover from them. The illegal states occur if more than one state bit is active at any given time. This could be caused by inputs violating the setup/hold times, or using too fast clock.

Another important fact is that upon system start-up and reset state machine enters state s0 (or the first state in the state machine declaration). However, if all flip-flops are cleared to zero after reset, the question is how could the state machine possibly enter state S0, which is represented with one-hot code, after a system reset.

Altera's one-hot encoding is done a little differently by using a coding scheme with all 0s representing the initial state and coding the other states as shown in Table 4.2

Table 4.2 One-hot encoding scheme as used by Altera

| State | State bit assignment |
|---|---|
| S0 | 0 0 0 ....0 0 0 |
| S1 | 1 0 0 ....0 0 1 |
| S2 | 1 0 0 ....0 1 0 |
| . | . |
| . | . |
| . | . |
| Sn | 1 1 0 ....0 0 0 |

The state bit corresponding to the default/reset state (S0) is actually inverted, so that all state bits are 0s after system reset. This is still one-hot coding, and you still only have to check a single state bit to determine if the state machine is in a particular state or not.

Another question is how to convert Moore FSM into Mealy FSM and vice versa. As outputs in a Mealy FSM depend on inputs and current state, if we are using case statement to describe a state machine, then WHEN clauses will contain IF-THEN-ELSE statement in which not only transition but also the output will be generated. For example, if we have a counter as in Example 4, which has an additional Enable input, the output from the counter will depend on the value of the Enable input. The following kind of WHEN clause will appear in the description of the state machine:

```
WHEN s0 =>
   IF Enable = B"1" THEN
    IF Up_down THEN
       Mealy = s1;
       Out[]  = B"01";
    ELSE
       Mealy = s3;
       Out= B"11";
    END IF;
    Out[] =
   ELSE
    Mealy = s0;
    Out[]  = B"00";
   END IF;
```

From the examples presented above, it can be seen that a Mealy machine can generate more output combinations than a Moore machine, given the same number of states. The reason for this is that the output of a Moore machine only depends on

the current state, while in a Mealy machine the output depends on some input as well. Therefore, to achieve the same set of output values in the two types of state machines, a Moore machine will generally be consuming more states than a Mealy machine would do.

An advantage of a Moore machine is that the output is independent of changes to the inputs, so the behavior of a Moore machine in a complex system may be less critical than in the Mealy case. In a Mealy machine, if the inputs do not follow an appropriate pattern, then problems with glitches on the outputs may occur.

Finally, if the state machine is designed with the possibility to branch to several other states from a given state, depending on the values of input signals, a good practice is to register (sample) the external inputs to the state machine. In this way, the inputs to the state machine can be prevented to enter illegal states, e.g as the result of violating the setup/hold rquirements.

## 4.4 Problems and Questions

4.1 A single-bit full adder is the circuit that takes as its inputs two operand bits a and b and input carry bit $c_{in}$, and produces as its output sum bit and output carry cout. The circuit is described by the truth Table 4.3. Design the circuit using AHDL and at least two different descriptions (models) (eg, Boolean equations and truth table). Compile and simulate design using different combinations of inputs.

Table 4.3 Truth table of full adder

| Inputs | | | Outputs | |
|---|---|---|---|---|
| a | b | $c_{in}$ | sum | $c_{out}$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

4.2 Model a 8-to-1 single-bit multiplexer using at least two different AHDL descriptions. Synthesize the multiplexer using two different target devices

(from MAX 7000 and FLEX 10K family). Compare results in terms of logic cells used.

4.3 Extend the design from the preceding example to a 16-to-l multiplexer.

4.4 Repeat problem 4.2 by modeling 8-to-l 8-bit multiplexer (all inputs are 8-bit wide). Are there any limitations in implementation?

4.5 Using AHDL design an address decoder that uses as its inputs address lines from 16-bit address to select memory chips that have capacity 4K words each. Assume that the memory is implemented with the full capacity of 64K words.

4.6 Design a parallel 16-bit comparator that compares two unsigned 16-bit numbers and has three outputs, which indicate whether the first number is greater than, equal to, or smaller than the first number. Compile your design for two target devices, one from MAX 7000 family and the other one from FLEX 10K family. Compare those two results in terms of number of logic cells used for implementation. Extend the design to 32-bit numbers and perform the same analysis.

4.7 An 8-bit combinatorial shifter is represented by the function in Table 4.4.

Table 4.4 Combinational shifter

| Operation Select | Operation | Function |
|---|---|---|
| 0 | Y ← A | No shift |
| 1 | Y ← shl(A) | Logic shift left |
| 2 | Y← shr(A) | Logic shift right |
| 3 | Y ← 0 | Zero output |
| 4 | Y← ashl(A) | Arithmetic shift left |
| 5 | Y ← ashr(A) | Arithmetic shift right |

A represent the input to the shifter, and Y represents the output.

- Design a circuit that performs the required functions
- Express your design using AHDL

4.8 Using AHDL design a 16-bit register that can be loaded with external data d[15..0] when the load signal is activated (active high). Content of the register is available on the output lines q[15..0].

4.9 A 16-bit bus connects four registers ($R_i$, $i = 0, 1, 2, 3$)from the preceding problem enabling register transfers from any register as the source to any other register as the destination of data. Assume that all registers have an additional common control input called init that, when activated, initializes each register to a predefined constant (different for each register). Selection of the source register for a bus transfer is done using a 2-bit select code source[1..0], and selection of the destination register is done using individual load lines load[3..0] (one line for each register). Register transfers are described by $R_i \leftarrow R_j$ (i,j = 0, 1, 2, 3). Using AHDL design the described circuit and check its operation using simulation. The circuit is illustrated in Figure 4.10.

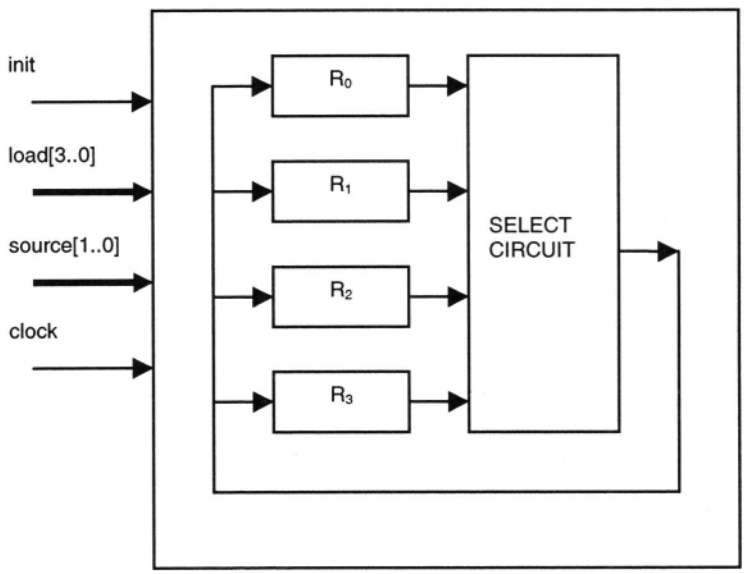



Figure 4.10 Simple bus for Problem 4.8

4.10Extend preceding example by introducing an external 16-bit data input and an external 16-bit data output that enable access to/from the bus from/to external world.

4.11 Using AHDL design a 16-bit register that can load new data and rotate or shift its content one bit left or right.

4.12 Repeat preceding problem by extending shift capabilities to shift or rotate its content for two bits left or right.

4.13 Using AHDL design a 16-bit register that can be initialized to H"8000" and rotate its content one bit left.

4.14 Using AHDL design a 16-bit register that can load a new data and swap the most-significant and least significant bytes.

4.15 Two 16-bit numbers are stored in two shift registers and compared using serial comparison (from the least-significant towards the most-significant bit). As the result three outputs are provided, which indicate whether the first number is greater than, equal to, or smaller than the first number. Compare this design with one from Problem 4.6.

4.16 Design a binary counter that can load a new initial data and counts up by increment (step) of 2, or counts down by decrement (step) of 1.

4.17 Design a 16-bit serial adder that loads (in parallel) two unsigned binary numbers in two shift registers A and B, and performs addition on the bit-by-bit basis using a single-bit full adder. The adder is illustrated in Figure 4.11.

Addition starts by activating start signal. After addition, result is stored in register B, and register A contains initially loaded data. Provide a proper propagation of carry bit from lower to higher significant bits. The design should also include control circuit that enables proper timing.

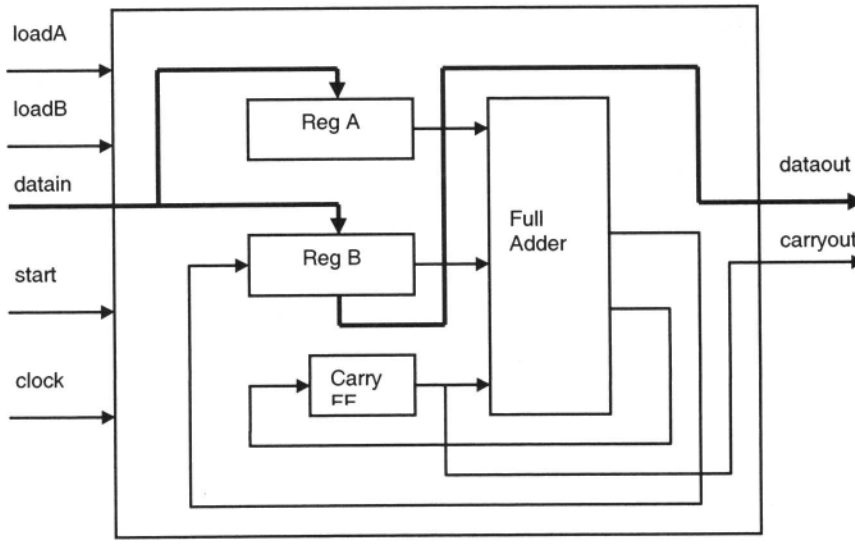



Figure 4.11 serial adder for Problem 4.16

4.18Design an 8x8-bit serial multiplier that multiplies two unsigned binary numbers in two shift registers A and B, and stores the result in register pair A,B (the most-significant byte in register A).

a) Draw a datapath similar to one in problem 4.16 and identify all data and control signals

b) Add a control unit that generates all internal signals
c) Using AHDL describe both data path and control unit and then integrate them into the serial multiplier

4.19Design a simple data path that can compute the following expression:

$$R = \sum_{i=1}^{n} a_i x_i$$

where $a_i$ and $x_I$ are two streams of 8-bit unsigned binary numbers and n constant that can fit in an 8-bit register. Add the corresponding control unit that controls calculation. Your design should be described in AHDL and synthesized for a FLEX10K device.

4.20Repeat the task from preceding problem for the following expression:

$$R = \sum_{i=1}^{n} a_i x_i \qquad if \; a_i \geq b_i$$

$$= \sum_{i=1}^{n} b_i x_i \qquad if \; a_i < b_i$$

Your design should be described in AHDL and synthesized for a FLEX10K

4.21 You are to implement in an FPLD a simple automatic gear controller that has a manual control to switch between PARK and DRIVE state. When in the DRIVE state the controller provides change between three gears depending on the status of accelerator and brake, and the current reading of the RPMs (rotations per minute). The change from the lower to the higher gear happens when the accelerator is pressed and RPM reading exceeds 2500 rpms, and the change to the lower gear happens when the accelerator is deactivated or brake pressed and RPM reading is lowers below 2000 rpms. Actual reading of the RPM meter is presented as a binary number representing hundreds of rpms (for example the reading for 2000 rpms is represented with binary value of 25). The only output from the controller is an indicator of the current gear displayed on one of the three LEDs. If the accelerator and the brake are pressed simultaneously, the brake has the higher priority and the accelerator function is overridden. Design the automatic gear controller using AHDL.

4.22A digital circuit is used to weigh small parcels and classifies them into four categories:

- less than 200 grams
- between 200 and 500 grams
- between 500 and 800 grams
- between 800 and 1000 grams

The input weight is obtained from a sensor as an 8-bit unsigned binary number. A linear dependency between the weight and input binary number in the whole range between 0 and 1023 grams is assumed. The weight is displayed on a four digit 7-segment display.

(a) Design a circuit that performs this task and activates a separate output signal whenever a new parcel is received. New parcel arrival is indicated by a single bit input signal. The weighing process starts at that point and lasts for exactly 100ms.

(b) Describe your design using AHDL

# 5 ADVANCED AHDL

This chapter presents the more advanced features of AHDL that are needed for the design of complex digital systems using FPLDs. As the language has been introduced informally, in this chapter we present it with a slightly more formal notion. The full syntax of the language can be found in Altera's documents and on CD-ROM included with this book. The features presented in Chapter 4 allow design of individual circuits of low to moderate complexity. However, as capacity and complexity of FPLDs grow, they require other mechanisms to enable managing complex designs that contain many interconnected parts.. In this chapter we concentrate on those features of the language that support the implementation of user designs as hierarchical projects consisting of a number of subdesigns, reusability of already designed circuits and mechanisms that enable generic designs by using parameters whose values are resolved before the actual compilation and synthesis take place. We also present techniques for digital systems design that fully utilize specific features found in FPLDs, such as memory blocks (EABs and ESBs in Altera FPLDs). Further examples using these features are given in chapters 6 and 7.

## 5.1 Names and Reserved Keywords and Symbols

Names in AHDL belong to either symbolic names or identifiers. They are formed as strings of only legal name characters (a-z, A-Z, 0-9, slash "/" and underscore "_") of the length of up to 32 characters. Reserved keywords are used for beginnings, endings, and transitions of AHDL statements and as predefined constant values such as GND and Vcc. Table 5.1 shows all AHDL reserved keywords and identifiers in alphabetical order. In the preceding chapter we used many of these reserved keywords and identifiers without formal introduction. Besides them, AHDL has uses a number of symbols with predefined meaning as shown in Table 5.2.

Table 5.1 AHDL reserved keywords

| AND | BEGIN | BIDIR | BITS | BURIED |
|---|---|---|---|---|
| CARRY | CASCADE | CASE | CLIQUE | CONNECTED_PINS |
| CONSTANT | DEFAULTS | DEFINE | DEVICE | DFF |
| DFFE | DIV | ELSE | ELSIF | END |
| EXP | FOR | FUNCTION | GENERATE | GLOBAL |
| GND | HELP_ID | IF | INCLUDE | INPUT |
| IS | JKFF | JKFFE | LATCH | LCELL |
| LOG2 | MACHINE | MEMORY | MOD | MCELL |
| NAND | NODE | NOR | NOT | OF |
| OPNDRN | OPTIONS | OR | OTHERS | OUTPUT |
| PARAMETERS | REPORT | RETURNS | SEGMENTS | SEVERITY |
| SOFT | SRFF | SRFFE | STATES | SUBDESIGN |
| TABLE | TFF | TFFE | THEN | TITLE |
| TO | TRI | TRI_STATE_NODE | VARIABLE | VCC |
| WHEN | WIRE | WITH | X | XNOR |
| XOR | | | | |

Table 5.2 AHDL symbols with predefined meanings

| Symbol | Function |
|---|---|
| _ (underscore)<br>- (dash) | User-defined identifiers used as legal characters in symbolic names |
| % (percent) | Encloses comments |
| ( ) (left & right parentheses) | Enclose and define sequential group names.<br>Enclose pin names in Subdesign Section and Function Prototype.<br>Optionally enclose inputs and outputs of truth tables in Truth Table Statements.<br>Enclose states of State Machine Declarations.<br>Enclose highest priority operations in Boolean expressions.<br>Enclose options in a Design Section (within Resource Assignment Statement). |
| [ ] (left and right brackets) | Enclose the number range of a decimal group name. |
| '...' (single quotation marks) | Enclose quoted symbolic names. |

| | |
|---|---|
| "..." (double quotation marks) | Enclose string in Title Statements.<br>Enclose pathname in Include Statements.<br>Enclose digits in non-decimal numbers.<br>Optionally enclose design name and device name in Design Section.<br>Optionally enclose clique name in Clique Assignment Section. |
| . (period) | Separates symbolic names of macrofunction or primitive variables from port names.<br>Separates extensions from filenames. |
| .. (ellipsis) | Separates most significant bit (MSB) from least significant bit (LSB) in ranges. |
| ; (semicolon) | Ends AHDL statements and sections. |
| , (comma) | Separates members of sequential groups and lists. |
| : (colon) | Separates symbolic names from types in declarations and resource assignments. |
| @ (at) | Assigns symbolic nodes to device pins and logic cells in Resource Assignments Statements. |
| => (arrow) | Separates inputs from outputs in Truth Table Statements.<br>Separates WHEN clauses from Boolean expressions in Case Statements. |
| = (equals) | Assigns default GND and VCC values to inputs in Subdesign Section.<br>Assigns settings to options.<br>Assigns values to state machine states.<br>Assigns values in Boolean equations. |

AHDL supports three types of names: symbolic names, subdesign names, and port names. These types of names are described below:

- Symbolic names are user-defined identifiers. They are used to name internal and external nodes, constants, state machine variables, state bits, states, and instances.
- Subdesign names are user-defined names for lower-level design files; they must be the same as the TDF filename.
- Port names are symbolic names that identify input or output of a primitive, macrofunction, megafunction or user defined function

The names can be used in quoted or unquoted notation. The quoted names are enclosed in single quotation marks. Quotes ate not included in pinstub names that are shown in the graphical symbol for a TDF.

## 5.2 Boolean Expressions

The result of every Boolean expression must be the same width as the node or group (on the left side of an equation) to which it is eventually assigned. The logical operators for Boolean expressions are shown in Table 5.3.

**Table 5.3 Logical Operators for Boolean expressions.**

| Operator | Description | Example |
|---|---|---|
| ! (NOT) | one's complement (prefix inverter) | !alpha<br>NOT alpha |
| & (AND) | AND | alpha & beta<br>alpha AND beta |
| !& (NAND) | AND inverter | a[3..0]!&b [5..2]<br>a[3..0]NAND b[5..2] |
| # (OR) | OR | all # all1<br>all OR all1 |
| !# (NOR) | OR inverter | a[3..1]!# b[4..2]<br>a[3..1]NOR b[4..2] |
| $ (XOR) | exclusive OR | ami $ bami<br>ami XOR bami |
| !$ (XNOR) | exclusive NOR | x1 !$ X2<br>x1 XNOR x2 |

Each operator represents a 2-input logic gate (binary operation), except the NOT operator (unary operation), which is a prefix inverter. Expressions that use these operators are interpreted differently depending on whether the operands are single nodes, groups, or numbers.

Three operand types are possible with the NOT operator. If the operand is a single node, GND, or Vcc, a single inversion operation is performed. If the operand is a group of nodes, every member of the group passes through an inverter. If the operand is a number, it is treated as a binary number with as many bits as the group context in which it is used and every bit is inverted. For example: !5 in a three-member group is interpreted as !B"101" = B"010".

Five operand combinations are possible with the binary operators and each of these combinations is interpreted differently:

- If both operands are single nodes or the constants GND or Vcc, the operator performs the logical operation on two elements.
- If both operands are groups of nodes, the operator produces a bit wise set of operations between the groups. The groups must be of the same size.
- If one operand is a single node (GND or Vcc) and the other operand is a group of nodes, the single node or constant is duplicated to form a group of the same size as the other operand. The expression is then treated as group operation.
- If both operands are numbers, the shorter number is sign extended to match the size of the other number. The expression is then treated as a group operation.
- If one operand is a numbers, the other is a node or group of nodes, the number is truncated or sign extended to match the size of the group.

Arithmetic operators are used to perform arithmetic addition and subtraction operations on groups and numbers. The Table 5.4 shows the arithmetic operators in AHDL.

Table 5.4 AHDL arithmetic operators.

| Operator | Description | Example |
|---|---|---|
| + (unary) | positive | +5 |
| - (unary) | negative | -b[3..0] |
| + | addition | a[3..0] + b[3..0] |
| - | subtraction | a_1[ ] - b_1[ ] |

The "+" unary operator does not effect the operand. The "-" unary operator interprets its operand as a binary representation of a number. It then performs a two's complement unary minus operation on the operand. In the case of arithmetic operators the following rules apply:

- The operands must be groups of nodes or numbers.
- If both operands are groups of nodes, the groups must be of the same size.
- If both operands are numbers, the shorter is sign-extended to match the size of the other operand.
- If one operand is a number and the other is a group ofnodes, the number is truncated or sign extended to match the size of the group. In the case of truncation of any significant bits, the compiler generates an error message.

Comparators are used to compare single nodes or groups. There are two types of comparators: logical and arithmetic. All types of comparators in AHDL are presented in Table 5.5.

**Table 5.5 AHDL Comparators**

| Comparator | Type | Description | Example |
|---|---|---|---|
| == | logical | equal to | addr[2..0] == B"101" |
| != | logical | not equal to | a_1 != a_2 |
| < | arithmetic | less than | a[ ] < b[ ] |
| <= | arithmetic | less than or equal | c[ ] <= d[ ] |
| > | arithmetic | greater than | a[ ] > b[ ] |
| >= | arithmetic | greater than or equal | c[ ] >= d[ ] |

The logical equal to operator (==) is used exclusively in Boolean expressions. Logical comparators can compare single nodes, groups (of the same size) or numbers. Comparison is performed on a bit-wise basis and returns Vcc when the comparison is true, and GND when the comparison is false. Arithmetic comparators can only compare groups of nodes or numbers. Each group is interpreted as a positive binary number and compared to the other group.

Priority of evaluation of logical and arithmetic operators and comparators is given in Table 5.6 (operations of equal priority are evaluated from left to right with the possibility to change the order using parentheses).

Table 5.6 AHDL Comparator and Operator Priority

| Priority | Operator/Comparator |
|---|---|
| 1 | - (negative) |
| 1 | ! (NOT) |
| 2 | + (addition) |
| 2 | - (subtraction) |
| 3 | == (equal to) |
| 3 | != (not equal to) |
| 3 | < (less than) |
| 3 | <= (less than or equal to) |
| 3 | > (greater than) |
| 3 | >= (greater than or equal to) |
| 4 | & (AND) |
| 4 | !& (NAND) |
| 5 | $ (XOR) |
| 5 | !$ (XNOR) |
| 6 | # (OR) |
| 6 | !# (NOR) |

## 5.3 Primitive Functions

AHDL TDFs use statements, operators, and keywords to replace some GDF primitives. Function Prototypes for these primitives are not required in TDFs. However, they can be used to redefine the calling order of the primitive inputs.

### *5.3.1 Buffer Primitives*

Buffer primitives allow control of the logic synthesis process. In most circumstances it is recommended to let the compiler indicate when and where to insert the buffers in order to support logic expansion.

1. CARRY

Function Prototype:
```
FUNCTION carry (in)
    RETURNS (out);
```

The carry buffer designates the carry out logic for a function and acts as the carry in to another function. It is supported only by the FLEX 8000 and 10K and APEX 20K family devices.

2. CASCADE

Function Prototype:
```
FUNCTION cascade (in)
    RETURNS (out);
```

The cascade buffer designates the cascade out function from an AND or an OR gate, and acts as an cascade-in to another AND or OR gate. It is supported only by the FLEX 8000 and 10K and APEX 20K family devices.

3. EXP

Function Prototype:
```
FUNCTION EXP (in)
    RETURNS (out);
```

The EXP expander buffer specifies that an expander product term is desired in the project. The expander product is inverted in the device. This feature is supported only for MAX devices. In other families it is treated as a NOT gate.

4. GLOBAL

Function Prototype:
```
FUNCTION GLOBAL (in)
    RETURNS (out);
```

The global buffer indicates that a signal must use a global (synchronous) Clock, Clear, Preset, or Output Enable signal, instead of signals generated by internal logic or driven by ordinary I/O pins. If an input port feeds directly to the input of GLOBAL, the output of GLOBAL can be used to feed a Clock, Clear, Preset, or Output Enable to a primitive. A direct connection must exist from the output of GLOBAL to the input of a register or a TRI buffer. A NOT gate may be required between the input pin and GLOBAL when the GLOBAL buffer feeds the Output Enable of a TRI buffer.

Global signals propagate more quickly than array signals, and are used to implement global clocking in a portion or all of the project.

5. LCELL

Function Prototype:
```
FUNCTION lcell (in)
    RETURNS (out);
```

The LCELL buffer allocates a logic cell for the project. It produces the true and complement of a logic function and makes both available to all logic in the device. An LCELL always consumes one logic cell.

6. OPNDRN

Function Prototype:
```
FUNCTION opndrn (in)
        RETURNS (out);
```

The OPNDRN primitive is equivalent to a TRI primitive whose Output Enable input can be any signal, but whose primary input is fed by GND primitive. If the input to the OPNDRN primitive is low, the output will be low. If the input is high, the output will be a high-impedance logic level.

The OPNDRN primitive is supported only for FLEX 10K device family. It is automatically converted to a TRI primitive for other devices.

7. SOFT

Function Prototype:
```
FUNCTION soft (in)
        RETURNS (out);
```

The SOFT buffer specifies that a logic cell may be needed at a specific location in the project. The Logic Synthesizer examines whether a logic cell is needed. If it is, the SOFT is converted into an LCELL; if not, the SOFT buffer is removed. If the Compiler indicates that the project is too complex, a SOFT buffer can be inserted to prevent logic expansion. For example, a SOFT buffer can be added at the combinational output of a macrofunction to decouple two combinational circuits.

8. TRI

Function Prototype:
```
FUNCTION TRI (in, oe)
        RETURNS (out);
```

The TRI is a tri-state buffer with an input, output, and Output Enable (oe) signal. If the oe signal is high, the output is driven by input. If oe is low, the output is placed into a high-impedance state that allows the I/O pin to be used as an input pin. The oe defaults to Vcc. If oe of a TRI buffer is connected to Vcc or a logic function that will minimize to true, a TRI buffer can be converted into a SOFT buffer during logic synthesis. When using a TRI buffer, the following must be considered:

- A TRI buffer may only drive one BIDIR port. A BIDIR port must be used if feedback is included after the TRI buffer.

- If a TRI buffer feeds logic, it must also feed a BIDIR port. If it feeds a BIDIR port, it may not feed any other outputs.
- When one is not tied to Vcc, the TRI buffer must feed an OUTPUT or BIDIR port. Internal signals may not be tri-stated.

### *5.3..2 Flip-flop and Latch Primitives*

Max+Plus II flip-flop and latch primitives are listed together with their Function Prototypes in Table 5.7. All flip-flops are positive edge triggered and latches are level-sensitive. When the Latch or Clock Enable (ena) input is high, the flip-flop or latch passes the signal from that data input to q output. When the ena input is low, the state q is maintained regardless of the data input.

Table 5.7 Flip-flop and latch primitives.

| Primitive | AHDL Function Prototype |
|---|---|
| LATCH | FUNCTION LATCH (d, ena)<br>RETURNS (q); |
| DFF | FUNCTION DFF (d, clk, clrn, prn)<br>RETURNS (q); |
| DFFE | FUNCTION DFFE (d, clk, clrn, prn, ena)<br>RETURNS (q); |
| JKFF | FUNCTION JKFF (j, k, clk, clrn, prn)<br>RETURNS (q); |
| JKFFE | FUNCTION JKFFE (j, k, clk, clrn, prn, ena)<br>RETURNS (q); |
| SRFF | FUNCTION SRFF (s, r, clk, clrn, prn)<br>RETURNS (q); |
| SRFFE | FUNCTION SRFFE (s, r, clk, clrn, prn, ena)<br>RETURNS (q); |
| TFF | FUNCTION TFF (t, clk, clrn, prn)<br>RETURNS (q); |
| TFFE | FUNCTION TFFE (t, clk, clrn, prn, ena)<br>RETURNS (q); |

Notes and definitions for Table 5.7:

clk = Register Clock Input

clrn = Clear Input

d, j, k, r, s, t = Input from Logic Array

ena = latch Enable or Clock Enable Input

prn = Preset Input

q = Output

### *5.3.3 Macrofunctions*

Max+Plus II provides a number of standard macrofunctions that represent high level building blocks that may be used in logic design. The macrofunctions are automatically installed in the \maxPlus2\max21ib directory and its subdirectories. The \maxPlus2\max2inc directory contains an Include File with a Function Prototype for each macrofunction. All unused gates and flip-flops are automatically removed by the compiler. The input ports also have default signal values, so the unused inputs can simply be left disconnected. Most of the macrofunctions have the same names as their 74-series TTL equivalents, but some additional macrofunctions are also available. Refer to the relevant directories for the most recent list of available macrofunctions. Examples of macrofunctions are given in Table 5.8.

**Table 4.8 Standard Max+Plus II Macrofunctions.**

| Macrofunction Type | Macrofunction Name | Description of Operation |
|---|---|---|
| Adder | 8fadd<br>7480<br>74283 | 8-bit full adder<br>Gated full adder<br>4-bit full adder with fast carry |
| Arithmetic Logic Unit | 74181<br>74182 | Arithmetic logic unit<br>Look-ahead carry generator |
| Application specific | pll | Rising- and falling-edge detector |
| Buffer | 74240<br>74541 | Octal inverting 3-state buffer<br>Octal 3-state buffer |
| Comparator | 8mcomp<br>7485<br>74688 | 8-bit magnitude comparator<br>4-bit magnitude comparator<br>8-bit identity comparator |
| Converter | 74184 | BCD-to-binary converter |

| | | |
|---|---|---|
| Counter | gray4<br>7468<br>7493<br>74191<br>74669 | Gray code counter<br>Dual decade counter<br>4-bit binary counter<br>4-bit up/down counter with asynch. load<br>Synchr. 4-bit up/down counter |
| Decoder | 16dmux<br>7446<br>74138 | 4-to-16 decoder<br>BCD-to-7-segment decoder<br>3-to-8 decoder |
| Digital Filter | 74297 | Digital phase-locked loop filter |
| EDAC | 74630 | 16-bit parallel error detection &correction circuit |
| Encoder | 74148<br>74348 | 8-to-3 encoder<br>8-to-3 priority encoder with 3-state outputs |
| Frequency divider | 7456 | Frequency divider |
| Latch | inpltch<br>7475<br>74259<br>74845 | Input latch<br>4-bit bistable latch<br>8-bit addressable latch with Clear<br>8-bit bus interface D latch with 3-state outputs |
| Multiplier | mult4<br>74261 | 4-bit parallel multiplier<br>2-bit parallel binary multiplier |
| Multiplexer | 21mux<br>74151<br>74157<br>74356 | 2-to-1 multiplexer<br>8-to-1 multiplexer<br>Quad 2-to-1 multiplexer<br>8-to-1data selector/multiplexer/register with 3-state outputs |
| Parity generator/ checker | 74180 | 9-bit odd/even parity generator/checker |
| Rate multiplier | 74167 | Synchronous decade rate multiplier |
| Register | 7470<br>7473<br>74171<br>74173<br>74396 | AND-gated JK flip-flop with Preset and Clear<br>Dual JK flip-flop with Clear<br>Quad D flip-flops with Clear<br>4-bit D register<br>Octal storage register |

| | | |
|---|---|---|
| Shift register | barrelst<br>7491<br>7495<br>74198<br>74674 | 8-bit barrel shifter<br>Serial-in serial-out shift register<br>4-bit parallel-access shift register<br>8-bit bidirectional shift register<br>16-bit shift register |
| Storage register | 7498 | 4-bit data selector/storage register |
| SSI Functions | inhb<br>7400<br>7421<br>7432<br>74386 | Inhibit gate<br>NAND2 gate<br>AND4 gate<br>OR2 gate<br>Quadruple XOR gate |
| True/ Complement I/O Element | 7487<br>74265 | 4-bit true/complement I/O element<br>Quadruple complementary output elements |

### *5.3.4 Logic Parameterized Modules*

AHDL allows the use of a number of megafunctions that are provided in the form of library of parameterized modules (LPMs). Those modules represent generic designs that are customized at the moment of their use by setting the values for parameters. A list of LPMs supported by Altera for use in AHDL and other design entry tools within Max+Plus II design environment is shown in Table 5.9.

Table 5.9 LPM modules

| Name | Description |
|---|---|
| **Gates** | |
| lpm_and | Multi-bit and gate for bit-wise and operation |
| lpm_bustri | Multi-bit three-state buffer for unidirectional and bidirectional buffer implementation |
| lpm_clshift | Combinatorial Logic Shifter or Barrel Shifter |
| lpm_constant | Constant Generator |
| lpm_decode | Decoder |
| lpm_or | Multi-bit or gate for bit-wise or operation |
| lpm_xor | Multi-bit xor gate for bit-wise xor operation |
| lpm_inv | Multi-bit inverter |
| busmux | Two-input multi-bit multiplexer (can be derived from lpm_mux) |

| | |
|---|---|
| lpm_mux | Multi-input multi-bit multilexer |
| mux | Single input multi-bit multiplexer (can be derived from lpm_mux) |
| **Arithmetic Components** | |
| lpm_abs | Absolute value |
| lpm_add_sub | Multi-bit Adder/Subtractor |
| lpm_compare | Two-input multi-bit comparator |
| lpm_counter | Multi-bit counter with various control options |
| lpm_mult | Multi-bit multiplier |

### *5.3.5 Ports*

A port is an input or output of a primitive, macrofunction, or state machine. A port can appear in three locations: the Subdesign Section, the Design Section, and the Logic Section.

A port that is an input or output of the current file is declared in the Subdesign Section. It appears in the following format:

Port_Name: Port_Type [=Default_Port_Value]

The following are port types available: INPUT, OUTPUT, BIDIR, MACHINE INPUT, and MACHINE OUTPUT.

When a TDF is the top-level file in hierarchy, the port name is synonymous with a pin name. The optional port value, GND or Vcc, can be specified for INPUT and BIDIR port types. It is used only if the port is left unconnected when an instance of the TDF is used in a higher- level design file.

A port that is an input or output of the current file can be assigned to a pin, logic cell, or chip in the Design Section. Examples are given in earlier sections.

A port that is an input or output of an instance of a primitive or lower-level design file is used in the Logic Section. To connect a primitive, macrofunction, or state machine to other portions of a TDF, insert an instance of the primitive or macrofunction with an in-line reference or Instance Declaration, or declare the state machine with a State Machine Declaration. Then use ports of the function in the Logic Section.Port names are used in the following format:

```
Instance_name.Port_name
```

The Instance_name is a user defined name for a function. The Port_name is identical to the port name that is declared in the Logic Section of a lower level TDF or to a pin name in another type of design file. All Altera-provided logic functions have predefined port names, which are shown in the Function Prototype. Commonly used names are shown in Table 5.10.

Table 5.10 Common Port Names

| Port Name | Definition |
|---|---|
| .q | Output of a flip-flop or latch |
| .d | Data input to a D flip-flop or latch |
| .t | Toggle input to a T flip-flop |
| .j | J input to a JK flip-flop |
| .k | K input to a JK flip-flop |
| .s | Set input to a SR flip-flop |
| .r | Reset input to a SR flip-flop |
| .clk | Clock input to a flip-flop |
| .ena | Clock Enable input to a flip-flop, Latch Enable input to a latch, or Enable input to a state machine |
| .prn | Active-low Preset input to a flip-flop |
| .clrn | Active-low Clear input to a flip-flop |
| .reset | Active-high Reset input to a state machine |
| .oe | Output Enable input to a flip-flop |
| .in | Primary input to CARRY, CASCADE, EXP, GLOBGAL, LCELL, OPNDRN, SOFT, and TRI primitives |
| .out | Output of CARRY, CASCADE, EXP, GLOBGAL, LCELL, OPNDRN, SOFT, and TRI primitives |

## 5.4 Implementing a Hierarchical Project Using Altera-provided Functions

AHDL TDFs can be mixed with GDFs, WDFs, EDIF Input Files, Altera Design Files, State Machine Files, and other TDFs in a project hierarchy. Lower level files in a project hierarchy can either be Altera-provided macrofunctions or user defined (custom) macrofunctions. This section shows how Altera-provided functions can be used in design hierarchy. In the following section we will show how users can write their own functions, including parameterized ones, and instantiate them into new designs.

Max+Plus II includes a large library of standard 74-series, bus, architecture optimized, and application-specific macrofunctions which can be used to create a hierarchical logic design. These macrofunctions are installed in the \maxPlus2\max21ib directory and its subdirectories.

There are two ways to call (insert an instance) a macrofunction in AHDL. One way is to declare a variable of type <macrofunction> in an Instance Declaration in the Variable Section and use ports of the instance of the macrofunction in the Logic Section. In this method, the names of the ports are important. The second way is to use a macrofunction reference in the Logic section of the TDF. In this method, the order of the ports is important.

The inputs and outputs of macrofunctions are listed in the Function Prototype Statement. A Function Prototype Statement can also be saved in an Include File and imported into a TDF with an Include Statement. Include Files for all macrofunctions are provided in the \maxPlus2\max2inc directory.

Example 5.1 shows the connection of a 4-bit counter in the free-running mode to a 4-to-16 decoder to make a multi-phase clock with 16 non-overlapping output phases. Macrofunctions are called with Instance Declarations in the Variable Section. The Function Prototypes for the two macrofunctions that are present in component library and are stored in the Include Files 4count.inc and 16dmux.inc, are shown below:

```
FUNCTION 4count  (clk,clrn,setn,ldn,cin,dnup,d,c,b,a)
    RETURNS (qd, qc, qb, qa, cout);

FUNCTION 16dmux (d, c, b, a)
    RETURNS  (q[15..0]);
```

The order of ports is important because there is an one-to-one correspondence between the order of the ports in the Function Prototype and the ports defined in the Logic Section.

### Example 5.1 Multiphase clock implementation

```
INCLUDE  "4count";
INCLUDE  "16dmux";

SUBDESIGN  multiphaes_clock
 (
    Clk    :  INPUT;
    out[15..0]   :OUTPUT;
)
```

```
VARIABLE
    counter: 4count;
    decoder: 16dmux;

BEGIN
    counter.clk = clk;
    counter.dnup = GND;
    decoder.(d,c,b,a) = counter.(qd,qc,qb,qa);
    out[15..0] = decoder.q[15..0];
END;
```

Include Statements are used to import Function Prototypes for the two Altera provided macrofunctions. In the Variable Section, the variable counter is declared as an instance of the `4count` macrofunction, and the variable decoder is declared as an instance of the `16dmux` macrofunction. The input ports for both macrofunctions, in the format <Instance_name>.Port_name, are defined on the left side of the Boolean equations in the Logic Section; the output ports are defined on the right. The order of the ports in the Function Prototypes is not important because the port names are explicitly listed in the Logic Section. A hierarchical dependency of the multiphase clock and its components and the GDF that is equivalent to the example above is shown in figures 5.1 and 5.2, respectively.

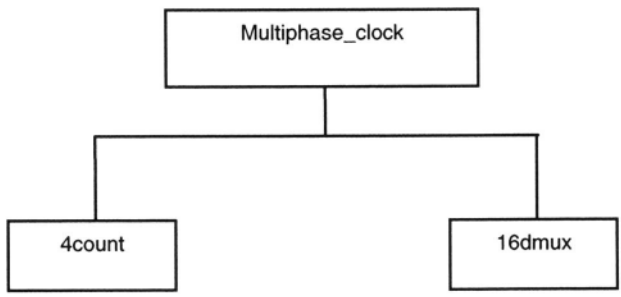



Figure 5.1 Hierarchical dependency in mutiphase clock design

The same functionality as in Example 5.1 can be implemented using in line references as shown in Example 5.2.

The in-line reference for the functions `4count` and `16dmux` appear on the right side of the Boolean equations in the Logic Section. In this case placeholders must be used instead of unused ports to the individual components.

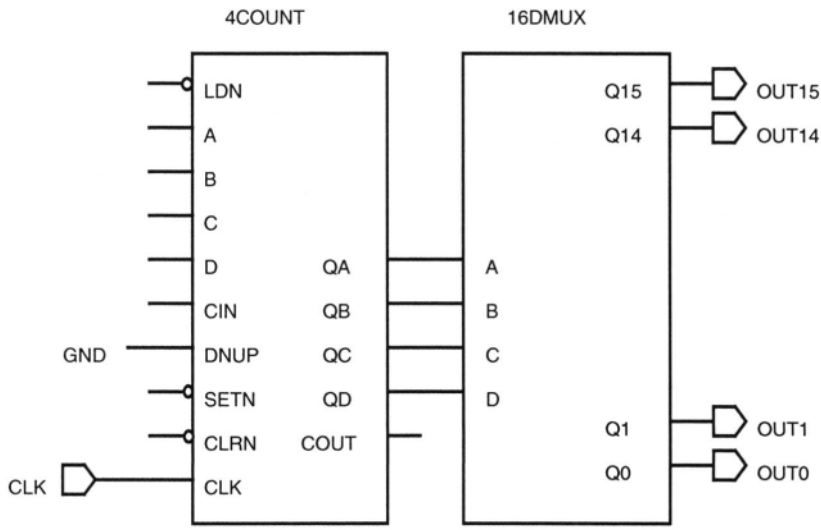



Figure 5.2 GDF Equivalent to Example 5.1 TDF

Example 5.2 In Line References for Counter and Decoder Connections.

```
INCLUDE "4count";
INCLUDE "16dmux";

SUBDESIGN multiphase_clock_2
 (
    clk    :INPUT;
    out [15..0] :OUTPUT;
)

VARIABLE
    q[3..0]:NODE;

BEGIN
     (q[3..0], )= 4count (clk,,,,, GND,,,,);
    out[15..0] = 16dmux (q[3..0]);
END;
```

To use a function, a Function Prototype must be included in the current TDF, or in the Include Statement used to include the information from an Include File's Function Prototype. Example 5.3 shows the implementation of a keypad encoder for a hexadecimal keypad. In this case the keypad encoder uses already present components from Altera library, some of them being functionally fully compatible

with the well known 74xxx TTL family (74151 multiplexer and 74154 2-to-4 decoder). The function of the hexadecimal keypad is to detect the key being pressed and generate its code on the output lines. It also generates a strobe signal that indicates that a valid key pressure has been detected and the code generated is valid. Relationship between the keypad and the encoder is illustrated in Figure 5.3. The hex keypad encoder is explained in more details in Chapter 6, where it is used in implementing an electronic lock and contains only user defined functions.

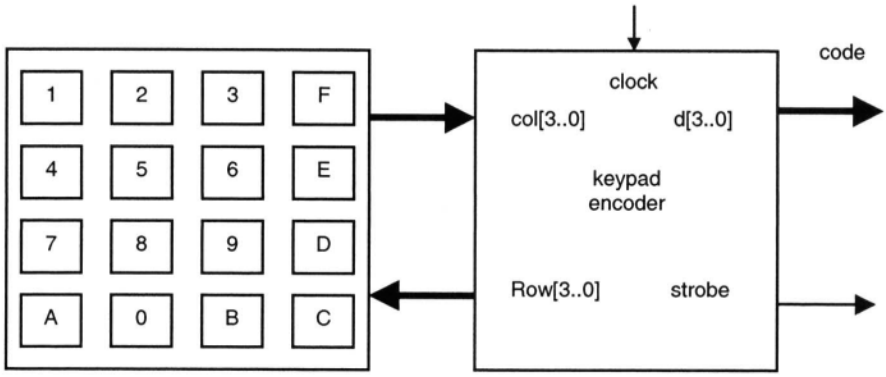



Figure 5.3 Hexadecimal keypad and its encoder

Example 5.3 Keypad Encoder

```
TITLE "Keypad encoder"

INCLUDE "74151";
INCLUDE "74154";
INCLUDE "4count";
FUNCTION debounce (clk, key_pressed);
   RETURNS (pulse);

SUBDESIGN keypad_encoder
 (
   clock  : INPUT; % 50 KHz clock %
   col[3..0]  : INPUT; % keypad columns %
   row[3..0], d[3..0]  : OUTPUT; % k-rows,key code%
   strobe  : OUTPUT; % key code is valid %
)

VARIABLE
   key_pressed : NODE; % Vcc when key d[3..0]  %
               % is pressed %
```

```
    mux   : 74151;
    decoder: 74154;
    counter : 4count;
    opencol[3..0] : TRI;

BEGIN
    % drive keyp.rows with a decoder and open %
    % collector outputs %
    row[ ] = opencol[ ].out;
    opencol[ ].in = GND;
    opencol [ ].oe = decoder.(o0n,o1n,o2n,o3n);
    decoder.(b,a) = counter.(qd,qc);

    % sense keyp.columns with a multiplexer %
    mux.d[3..0] = col[3..0];
    mux.(b,a) = counter.(qb,qa);
    key_pressed = !mux.y;

    % scan keyp. until a key is pressed &
    % drive key's code onto d[ ] outputs %

    counter.clk = clock;
    counter.cin = !key_pressed;
    d[ ] = counter.(qd,qc,qb,qa);

    % generate strobe when key has settled %

    strobe = debounce(clock, key_pressed);
END
```

Include Statements include Function Prototypes for the Altera provided macrofunctions `4count`, `74151`, and `74154`. A separate Function Prototype Statement specifies the ports of the custom function debounce, which is used to debounce keys, within the TDF rather than in an Include File. Instances of Altera provided macrofunctions are called with Instance Declarations in the Variable Section; an instance of the debounce function is called with an in-line reference in the Logic Section.

## 5.5 Creating and Using Custom Functions in AHDL

In this section we will discuss creation and use of custom functions and use of parameters to make more generic designs. Firstly, we consider creation of custom functions including parameterized and non-parameterized ones. Then we discuss the methods of using those functions in the designs on a higher hierarchical level.

### *5.5.1 Creation of Custom Functions*

Custom functions can be easily created and used in AHDL by performing the following tasks:

- Create the logic for the function in a design file and compile it.
- Specify the function's ports with a Function Prototype Statement. This, in turn, provides a shorthand description of a function, listing the name and its input, output, and bidirectional ports. Machine ports can also be used for functions that import or export state machines. The Function Prototype Statement can also be placed in an Include File and called with an Include Statement in the file.
- Insert an instance of the macrofunction with an Instance Declaration or an in-line reference.
- Use the macrofunction in the TDF description.

As we have seen in the preceding section there are two basic methods for using functions in AHDL:

- Instantiation of function in Variable Section and use of the instances in Logic Section of the design
- In-line reference of function by direct use of "function calls" in Logic Section with one of two types of port associations: named port association, or positional port association

Let us consider two example functions. The first function is non-parameterized 3-to-8 line decoder given in Example 5.4.

Example 5.4 Non-parameterized decoder

```
SUBDESIGN decoder
 (
    address  [2..0]   :NPUT = GND;
    decode   [7..0]   :OUTPUT;
)

BEGIN

TABLE
```

```
    address [] =>   decode [] ;

    0          =>    B"00000001";
    1          =>    B"00000010";
    2          =>    B"00000100";
    3          =>    B"00001000";
    4          =>    B"00010000";
    5          =>    B"00100000";
    6          =>    B"01000000";
    7          =>    B"10000000";
END TABLE;
END;
```

The second function is a parameterized **n-to-2$^n$** line decoder, as shown in Example 5.5. We introduce parameterized designs in this section. They will be discussed in more details in the following sections, as one of the most powerful features of AHDL for describing generic designs.

Example 5.5 Parameterized **n-to-2$^n$** decoder

```
PARAMETERS
(
    WIDTH = 3
);

CONSTANT TOTAL_BITS = 2^WIDTH;

SUBDESIGN param_decoder
(
    address  [(WIDTH-1)..0]   :INPUT = GND;
    decode  [(TOTAL_BITS-1)..0]   :OUTPUT;
)

BEGIN

FOR i IN 0 TO (TOTAL_BITS-1) GENERATE
   IF address[] == i THEN
      decode[i] = VCC;
    END IF;
END GENERATE;

END;
```

Note that the address[ ] input of both functions has a default value of GND. This means that if we don't connect these inputs externally, they will default to GND. The function prototypes for the above functions will be created by Max+Plus II and they have the following form:

```
FUNCTION decoder  (address[2..0])
   RETURNS  (decode[7..0]);

FUNCTION parm_decoder (address[(width) - (1)..0])
   WITH  (WIDTH)
   RETURNS  (decode[((2)  ^  (width)) - (1)..0]);
```

respectively.

### *5.5.2 In-line References to Custom Functions*

The designer must use the INCLUDE statement to include the function prototypes into a new TDF description before the use of these functions. Example 5.6 represents an AHDL design using in-line references of the above functions.

**Example 5.6 In-line references to custom functions**

```
INCLUDE "decoder";
INCLUDE "param_decoder";

SUBDESIGN complex_decoder
 (
   a[2..0]   :INPUT;
   b[1..0] [1..0]  :INPUT;
   alpha[7..0]   :OUTPUT;
   beta[7..0]   :OUTPUT;
   out   :OUTPUT;
   out1,  out2   :OUTPUT;
)

BEGIN

    (out1,  out2)  =  decoder  (a[])
   RETURNS  (.decode7,  .decode0);

   alpha[]  =  decoder  (  ,  a1,  a0);

   out = param_decoder (Vcc, b[1][])
   WITH  (WIDTH = 3)
   RETURNS  (.decode0);

   beta [3..0]  = param_decoder  (.address[2..1]  =  (VCC,
             b[1][1]),  .(address3, address0)  =  (a3,)  )
   WITH  (WIDTH = 4)
   RETURNS  (.decode7,  .decode[2..0]);
   beta[7..4]  =  decoder  (.address[2..1]  =  b[0]  [],
```

```
            .address0 = a1 )
   RETURNS (.decode[1..0] , .decode[1..0] ) ;

END;
```

This example shows a mixture of positional and named port associations. It is not allowed to mix positional and named port associations within the same statement. Here is a summary of the functions using different types of port associations:

| Positional port associations | Named port associations |
|---|---|
| (out1, out2)<br>alpha[]<br>out | beta[3..0]<br>beta[7..4] |

With positional port association, we use a string of bits placed in any order we want as the function input. The only thing we have to make sure is that the number of bits provided in the function input is the same as number of bits in the function prototype. If an input of a function has a default value, we don't have to specify a value for that input. It is indicated by the placeholders with the lonely comma sign (','). With named port association, we name the used port and assign the values of our choice, unless the ports have default values.

By default, the function we instantiate returns the same number of bits as defined in the function prototype. If we want to return a lower number of bits, or if we want to return the same bits twice (like in the beta[7..4] function), or if we want to return the bits in another order, we must use the RETURNS statement to name and state the order of the returned bits.

The lonely comma signs on the left side can be used if we want to discard return values. For example, the line

```
alpha[] = decoder( , a1, a0);
```

can be replaced with the line

```
alpha[7..4],,,,) = decoder( , a1, a0);
```

if we want to discard the 4 least significant bits returned from the decoder function. In this case - alpha[3..0] remains unassigned.

From the above examples we see that the only difference between using a parameterized function as compared to a non-parameterized function, is the use of the WITH ( ) clause.

### *5.5.3 Using Instances of Custom Function*

Example 5.7 uses instances of the decoder and param_decoder functions to create a new design. Instances together with their names are declared in Variable Section of TDF description. The syntax is straightforward and easily readable.

Example 5.7 Using Instances of Custom Functions

```
INCLUDE "decoder";
INCLUDE "paramm_decoder";

SUBDESIGN top_2
 (
    a[3..0]         : INPUT;
    b[1..0] [1. .0] : INPUT;
    alpha[15..0]    : OUTPUT;
    out1            : OUTPUT;
)

VARIABLE
    first, second   :decoder;
    param_first    :param_decoder WITH (WIDTH = 4);

BEGIN
    first.(address3, address0) = GND;
    first.address[2..1]    = (VCC, a1);
    alpha[3..0] = (first.decode[12..11].first.decode15, VCC);

    param_first.address[] = b[][];
    alpha[15..4] = param_first.(decode13,
    decode[15..5]);

    second.address[] = (VCC, GND);
    out1 = second.decode15;
END;
```

As we can see the left side of the line second.address[ ] = (VCC, GND) requires 4 bits. On the right side there are only 2 bits. What happens is that the 2 bits on the right side are replicated, so that the bit string assigned to second.address[ ] is (VCC, GND, VCC, GND).

## 5.6 Using Standard Parameterized Designs

In this section we first discuss the use of parameterized functions already provided in Altera Max+Plus II library. Those designs belong to the class of library parameterized modules (LPMs), which implement frequently used digital functions and algorithms. Number of such functions is growing as many companies make and sell designs that are parameterized and perform specialized tasks. Typical examples of such functions are various signal processing elements (digital FIR and IIR filters, image processing elements, microcontrollers etc).

### *5.6.1 Using LPMs*

A number of Altera Max+Plus II functions are parameterized, allowing declaration of the parameter value at the compilation time. Those functions belong to the library of parameterized modules (LPMs). For example, parameters are used to specify the width of ports which represent input operands and results of operation of a functional unit. LPMs are instantiated with an in-line logic function reference or an Instance Declaration in the same way as non-parameterized functions as shown in preceding section. However, a few additional steps are required to declare the values of parameters, providing essentially customization of the function to the designer's requirements. These steps include:

- Use of the `WITH` clause, that lists parameters used by the instance. If parameter values are not supplied within the instance, they must be provided somewhere else within the project.

- Specification of the values of unconnected pins. This requirement comes from the fact that the parameterized functions do not have default values for unconnected inputs.

The inputs, outputs, and parameters of the function are declared with Function Prototype Statement, or they can be provided from corresponding Include Files. For example, if we want to use Max+PlusII multiplier LPM lpm_mult, its Function Prototype is given as shown below:

```
FUNCTION lpm_mult  (dataa[(LPM_WIDTHA-1)..0],
          datab [ (LPM_WIDTHB-1)..0], sum[(LPM_WIDTHS-1)..0],
          aclr, clock)
WITH  (LPM_WIDTHA, LPM_WIDTHB, LPM_WIDTHP, LPM_WIDTHS,
          LPM_REPRESENTATION, LPM_PIPELINE, LATENCY,
          INPUT_A_IS_CONSTANT, INPUT_B_IS_CONSTANT,
          USE_EAB)
RETURNS  (result[LPM_WIDTHP-1..0]);
```

This function provides multiplication of input operands a and b and addition of the partial sum to provide the output result. As such it is unsuitable for multiply-and-accumulate type of operation. Clock and aclr control signals are used for pipelined operation of the multiplier to clock and clear intermediate registers.

A number of parameters using WITH clause are provided including those to specify the widths and nature of all operands and result (variable and constants), as well as the use of FPLD resources (implementation using EABs in the case of FLEX 10K devices). Only widths of inputs and result are required, while the other parameters are optional. Example 5.8 shows the use of lpm_mult function.

Example 5.8 Using LPM multiplier with in-line reference

```
INCLUDE "lpm_mult.inc";

SUBDESIGN mult8x8
 (
    a[7..0], b[7..0] : INPUT;
    c[15..0]         : OUTPUT;
)

BEGIN
    c[] = lpm_mult(a[], b[],0, , )
       WITH (lpm_widtha=8, lpm_widthb=8, lpm_widths=1,
             lpm_widthp=16);
END;
```

It should be noted that the width must be a positive number. Also, placeholders are used instead of clock and clear inputs which are not used in this case.

Another possibility is to use Instance Declaration of the multiplier as in Example 5.9.

Example 5.9 Using LPM multiplier with Instance Declaration

```
INCLUDE "lpm_mult.inc";

SUBDESIGN mult8x8
 (
    a[7..0] , b[7..0] : INPUT;
    c[15..0]          : OUTPUT;
)
```

```
VARIABLE
    8x8mult: lpm_mult WITH(lpm_widtha=8, lpm_widthb=8,
            lpm_widths=1, lpm_widthr=16);

BEGIN
    8x8mult.dataa[] = a[] ;
    8x8mult.datab[] = b[];
    c[]= 8x8mult.result [] ;

END;
```

A number of LPMs currently available in Max+PlusII library are listed in Table 5.9. The details of their use and parameters available are provided in corresponding Altera documents. Another group of LMPs that enable use of embedded memory blocks is described in the following section.

### *5.6.2 Implementing RAM and ROM*

Besides providing standard combinational and sequential parameterized logic modules, Max+PlusII provides a number of very useful memory functions that can be a part of user's designs. Their implementation is not equally efficient in all Altera's devices, but higher capacity is obviously achieved in FLEX 10K and APEX 20K devices, due to existence of embedded memory blocks. In other devices memory functions are implemented using logic elements or macrocells.

The following memory megafunctions can be used to implement RAM and ROM:

- lpm_ram_dq: Synchronous or asynchronous memory with separate input and output ports
- lpm_ram_io: Synchronous or asynchronous memory with single I/O port
- lpm_rom: Synchronous or asynchronous read-only memory
- csdpram: Cycle-shared dual port memory
- csfifo: Cycle-shared first-in first-out (FIFO) buffer

Parameters are used to determine the input and output data widths, the number of data words stored in memory, whether data inputs, address and control inputs and outputs are registered or not, whether an initial memory content file is to be included and so on.

As an example consider synchronous or asynchronous read-only (ROM) memory LPM (sometimes called megafunction, too) that is represented by the following prototype function:

```
FUNCTION lpm_rom (address[LPM_WIDTHAD-1..0], inclock,
          outclock, memenab)
WITH (LPM_WIDTH, LPM_WIDTHAD, LPM_NUMWORDS, LPM_FILE,
          LPM_ADDRESS_CONTROL, LPM_OUTDATA)
RETURNS (q[LPM_WIDTH-1..0]);
```

Input ports are address lines with the number of lines specified by lpm_widthad parameter, inclock and outclock that specify frequency for input and output registers, and memory enable, while the output port is presented by a number of data lines given with parameter 1pm_width. The ROM megafunction can be used either by in-line reference or by Instance Declaration, as it was shown in the previous example of parameterized multiplier. The examples of using memory LPMs will be shown in the later chapters.

## 5.7 User-defined Parameterized Functions

Parameterized functions represent a very powerful tool in AHDL as they enable designs that are easily customizable to the needs of a concrete application. The designers will take the advantages of using parameterized functions and making generic designs. The function is designed and specified once and then can be re-used in a number of designs by setting the values for parameters as required, without a need for re-design of the functionality implemented within the function. In this way, AHDL provides the tool similar to object-oriented tools in programming languages. As an example of a parameterized design and use of parameterized functions, we will design a parameterized frequency divider.

At the beginning, we will design a frequency divider with fixed, hard-coded, divisor. Its output out goes high for every 200th count of the input system clock. It will count synchronously with clock whenever enable is high. Counter[ ] starts at 0 after a system power-up. On the next count it is loaded with 199. On the next count again it will be decremented to 198, and will then continue to decrement on every count all the way down to 0, when the cycle starts all over again. The AHDL description of the divider is given in Example 5.10.

Example 5.10 Frequency divider by 200

```
SUBDESIGN divider_by_200
 (
    clock, enable     : INPUT;
    count[7..0], out  : OUTPUT;
)

VARIABLE
    counter[7..0]     : DFF;

BEGIN
    counter[].clk = SysClk;
    count[] = counter[];

IF enable THEN
    IF counter[] == 0 THEN
       counter[] = 199;
       out = Vcc;
     ELSE
       counter[] = counter[] - 1;
     END IF;
ELSE
     counter[] = counter[];
END IF;

END;
```

This design can be made more user friendly by replacing hard-coded values with constants that are defined at the top of the design description, as it is shown in Example 5.11. We introduce a constant DIVISOR, but also use another constant DIVISOR_LOAD that will have the value one less than the value of the DIVISOR and will be loaded into counter variable whenever a cycle of counting has been finished. Obviously, the only condition is that the number DIVISOR_LOAD has to have the value that can be represented by the binary number with WIDTH bits. Whenever we want to change the value of divisor, only two constants on the top of description have to be changed.

Example 5.11 Frequency divider by n

```
CONSTANT DIVISOR = 199;
CONSTANT WIDTH  = 8;
CONSTANT DIVISOR_LOAD = DIVISOR - 1;

SUBDESIGN divider_by_n
 (
    clock, enable  :INPUT;
```

```
    count[WIDTH-1..0], out : OUTPUT;
)

VARIABLE
    counter[WIDTH-1..0]     : DFF;

BEGIN
    counter[].clk = clock;
    count[] = counter[];

IF enable THEN
    IF counter[] == 0 THEN
        counter[] = DIVISOR_LOAD;
        out = Vcc;
    ELSE
        counter[] = counter[] - 1;
    END IF;

ELSE
    counter[] = counter[];

END IF;

END;
```

As the last step we can complete this frequency divider design by transforming it into a parameterized function.. All we have to do now is replace the two topmost constant definitions with a Parameter Section, and the resulting TDF description will look as shown in Example 5.12.

Example 5.12 Parameterized frequency divider

```
PARAMETERS
(
    DIVISOR = 200;
    WIDTH  = 8;
);

CONSTANT DIVISOR_LOAD = DIVISOR - 1;

SUBDESIGN divider_by_n
(
    clock, enable  :INPUT;
    count[WIDTH-1..0], out  :OUTPUT;
)

VARIABLE
    counter[WIDTH-1..0]  :DFF;
```

```
BEGIN
    counter[].clk = clock;
    count [] = counter[];

IF enable THEN
    IF counter[] == 0 THEN
        counter[] = DIVISOR_LOAD;
        out = Vcc;
    ELSE
        counter[] = counter[] - 1;
    END IF;
ELSE
    counter[] = counter[];

END IF;

END;
```

If an input port from the design would not be used in a subsequent design, it should be marked as unused. Similarly, for an output port that will not be used a default value must be given. AHDL provides the USED statement to check for unused ports and take appropriate actions.

As we can see from Parameter Section we have been using default values for our parameters. The default values of parameters will only be used when the file is compiled as a stand-alone top-level file. However, when using parameterized function in design hierarchy, the higher level design sets the values of parameters.

AHDL allows to use global project parameters to further parameterize the design. For instance, if we use a parameterized function with a parameter called WIDTH, we could specify WIDTH to be a number like 16, or we could specify it to be equal to a global project parameter, like GLOBAL_WIDTH. Using this method, we only need to specify the GLOBAL_WIDTH parameter once, and when changed, it is applied to all the WIDTH parameters for the entire project.

Two compile-time functions can be used to further enhance and simplify parameterized functions specification. The LOG2() (logarithm base 2) function can be used to determine number of bits required to represent specific numbers. A number of bits required to represent number N can be calculated using LOG2(N). As it can be a non-integer (for example LOG2(15) = 3.9069), the upwards rounding can be performed using CEIL compile-time function. As we have to include zero as one of the numbers, the required number of bits to represent all integers up to N will be:

```
CEIL(LOG2(N+1))
```

Having this in mind we can further simplify frequency divider from the previous example. The topmost TDF description lines will look as those shown in Example 5.13.

Example 5.13 Further parameterization of frequency divider

```
PARAMETERS
(
   DIVISOR = 200;
);

CONSTANT DIVISOR_LOAD = DIVISOR - 1;
CONSTANT WIDTH = CEIL(LOG2(DIVISOR1);
```

## 5.8 Conditionally and Iteratively Generated Logic

Logic can be generated conditionally using If Generate Statements. This shows to be useful if, for example, we want to implement different behavior based on the value of a parameter or an arithmetic expressions. An If Generate Statement lists a series of behavioral statements that are activated after the positive evaluation of one or more arithmetic expressions. Unlike If Then Statements, which can evaluate only Boolean expressions, If Generate Statements can evaluate the superset of arithmetic expressions. The essential difference between an If Then Statement and an If Generate Statement is that the former is evaluated in hardware (silicon), whereas the latter is evaluated when the design is compiled.

The frequency divider function as described in previous examples may produce glitches on the out output. We can add another parameter, specifying that we can optionally register the out output. By using If Generate Statement, we can optionally declare and use an extra register for the output, as it is shown in Example 5.14. Parameter GLITCH_FREE (string) is used with the default value “YES”, but if we do not want the output flip-flop, it will not be generated. The If Generate Statement can be used in the Variable Section of the TDF description as well as in the Logic Section.

Example 5.14 Using conditionally generated logic

```
PARAMETERS
(
   DIVISOR = 200,
   GLITCH_FREE = "YES"
);
CONSTANT DIV_TMP = (DIVIDE - 1);
CONSTANT WIDTH = CEIL ( LOG2(DIVIDE) );
```

```
SUBDESIGN frequency_divider
 (
    Clock   :INPUT;
    enable  :INPUT = VCC;
    count[WIDTH-1..0], out  : OUTPUT;
)

VARIABLE
    counter[WIDTH-1..0]      : DFF;

IF GLITCH_FREE == "YES" GENERATE
    Outreg  :DFF;
ELSE GENERATE
    Outnode  :NODE;
END GENERATE;

BEGIN
    counter[].clk = clock;
    count [] = counter[];

IF GLITCH_FREE == "YES" GENERATE
    out = outreg;
    outreg.clk = clock;
ELSE GENERATE
    out = outnode;
END GENERATE;

IF enable THEN
    IF counter[] == 0 THEN
        counter[] = DIVISOR_LOAD;

        IF GLITCH_FREE == "YES" GENERATE
           outreg = Vcc;
        ELSE GENERATE
           outnode = VCC;
        END GENERATE;

    ELSE
        counter[] = counter[] - 1;
    END IF;

ELSE
    counter[] = counter[];
END IF;

END;
```

It should be noted in this example that when GLITCH_FREE is set to "YES", the output will be delayed by one clock cycle because of the extra flip-flop inserted.

When we wish to use multiple blocks of logic that are the same or very similar, we can use the For Generate Statement to iteratively generate logic based on a numeric range delimited by arithmetic expressions. The For Generate Statement has the following parts:

- The keywords FOR and GENERATE enclose the following items:
- A temporary variable name, which consists of a symbolic name that is used only within the context of the For Generate Statement, i.e., the variable ceases to exist after the Compiler processes the statement. This variable name cannot be a constant, parameter, or node name that is used elsewhere in the project.
- The word IN, which is followed by a range delimited by two arithmetic expressions. The arithmetic expressions are separated by the TO keyword. The range endpoints can consist of expressions containing only constants and parameters; variables are not required.
- The GENERATE keyword is followed by one or more logic statements, each of which ends with a semicolon (;).
- The keywords END GENERATE and a semicolon (;) end the For Generate Statement.

In Example 5.15 the For Generate Statement is used to instantiate full adders that each perform one bit of the NUM_OF_ADDERS-bit (i.e., 8-bit) addition. The carryout of each bit is generated along with each full adder.

Example 5.15 Using iteratively generated logic

```
CONSTANT NO_OF_ADDERS = 8;
SUBDESIGN iteration_add
 (
    a[NO_OF_ADDERS..1]   :INPUT;
    b[NO_OF_ADDERS..1],  cin   :INPUT;
    C[NO_OF_ADDERS..1],  cout   :OUTPUT;
 )

VARIABLE
    sum[NO_OF_ADDERS..1],carryout[(NO_OF_ADDERS+1)..1]   :NODE;

BEGIN
    carryout[1] = cin;

FOR i IN 1 TO NO_OF_ADDERS GENERATE
```

```
    sum[i] = a[i] $ b[i] $ carryout[i]; % Full Adder%
    carryout[i+1] = a[i] & b[i] # carryout[i] & (a[i] $ b[i]);
END GENERATE;

    cout = carryout[NO_OF_ADDERS+1];
    c[] = sum[];

END;
```

## 5.9 Problems and Questions

5.1 Using parameterized design, describe in AHDL a loadable register of the size REG_WIDTH with optional enable input.

5.2 A timer is a circuit that can be initialized to the binary value INIT_VALUE, enabled and started to countdown when external events occur. When INTI_VALUE of events occur, the timer activates the timeout output for duration of one clock cycle and disables itself until start signal is activated again. Make an AHDL description of the timer.

5.3 The frequency divider presented in this chapter is to be modified to enable not only division of input clock frequency, but also to generate output clock with a desired duty cycle. Design a parameterized frequency divider that divides input clock frequency by N, and provides the duty cycle of the generated clock of duration M (M<N-1) cycles of the input clock.

5.4 Pulse-Width Modulation (PWM) generator takes an input value through the input data lines that specifies duty cycle of the output periodic waveform. The duty cycle is specified as a number of cycles of the system (fundamental) clock. The output clock has the frequency obtained by dividing system clock frequency by N. Describe the PWM generator using AHDL and comment on its performance after synthesis for an FLEX10K device.

5.5 Using memory LPM module design a small RAM memory systems of capacity:

a) 1024x8 words
b) 512x16 words

Memory decoding circuitry should be a part of the memory system. Memory system has W/R line (for writing/reading), MS (memory select line) for

integration with other circuits, separate input and output data lines and necessary number of address lines.

5.6 Using memory LPM modules implement a look-up table that performs squaring of 8-bit input. Use AHDL to describe the circuit, and specify at least 10 values of the look-up table that will be stored in the corresponding .mif file.

5.7 Design a data path that performs the following calculation:

$$Y = 15X_1 + 32X_2$$

where $X_1$ and $X_2$ are 8-bit positive numbers. For your design you can use LPM memory modules.

5.8 Repeat the preceding problem assuming that $X_1$ and $X_2$ are input sequences, and Y output sequence, that contain N numbers.

5.9 Using a ROM LPM design a multiplier that multiplies 4-bit unsigned binary numbers. Describe your design in AHDL.

5.10 Using the 4x4 multiplier from 5.9 design a maximum-parallel multiplier for multiplication of 8-bit unsigned numbers (Hint: consider 8-bit numbers as consisting of two hexadecimal digits $A=A_1A_0$ and $B=B_1B_0$ and express the product AxB as $A_1A_0 \times B_1B_0$). For the design you can use as many as you need parallel adders.

5.11 Design an 8x8 multiplier for multiplication of 8-bit unsigned binary numbers that uses only a single 4x4 multiplier designed in Problem 5.9. The design should employ serial-type architecture.

a) Draw the data path for the multiplier
b) Design the control unit
c) Compare design with one from Problem 5.10 in terms of used resources and speed of multiplication

5.12 Design an absolute value unit that accepts an N-bit two's complement number on its input and delivers its absolute value on the output.

5.13 Design a min/max unit that accepts two two's complement N-bit binary numbers on its input and delivers (as selected by a select bit) either lower or greater number.

5.14 Design an adder/subtracter unit that adds or subtracts two N-bit binary numbers.

5.15 Combine units from problems 5.12-5.14 to design a single unit that can perform a selected operation: absolute value, min/max, add or subtract.

5.16Design a N-tap finite impulse response filter (FIR) that calculates the following output based on input stream of 8-bit numbers x(i) and constants h(i)

$$y = \sum_{i=1}^{N} h(10 - i)x(i)$$

assuming that the coefficients h(i) are symmetric, that is h(i)=h(10-i) for every i. Also, assume that N is an even natural number. The design should be shown at the level of schematic diagram using basic building blocks such as parallel multipliers, adders, and registers that contain input data stream (and implement the delay line). Use AHDL to express your design.

5.17Design a LIFO (Last In First Out) stack with 512Kx8 of RAM that uses all 512 locations of the RAM. As RAM use RAM LPMs implemented in EABs of FLEX10K family. The stack pointer always points to the next free location on the stack.

5.18 Design a FIFO (First In First Out) queue containing 16 8-bit locations implemented in logic cells of a FLEX10K device. TOP and TAIL pointers always point to the location from which next data will be pulled from the FIFO and to which next data will be stored into FIFO. When they point to the same location the FIFO is empty, and when they differ for 1, the FIFO is full.

5.19 Design a parameterized FIFO that has a number of locations and location width as parameters. Assume that the number of locations can be expressed as $2^n$ (n integer in the range from 3 to 7). Analyze performance figures (resource utilization and speed) as a function of n.

5.20 Design a FIFO that has 512 8-bit locations implemented in RAM. For RAM use RAM LPMs implemented in EABs of FLEX 10K or APEX 20K family.

5.21 Design a circuit that implements a sinusoidal function. A full period of sinusoid is presented in the form of a look-up table as 256 samples stored in a single EAB. This table can be used to generate sinusoids of different frequencies depending on the speed at which samples are read from the look-up table. Design the whole circuit that generate samples of a sinusoid of desired frequency on its outputs.

# 6 DESIGN EXAMPLES

This chapter contains two design examples illustrating the use of FPLDs and AHDL in design of complete simple systems. The emphasis is on textual design entry and a hierarchical approach to digital systems design. The first design is an electronic lock that is used to enter a password, which consists of five decimal digits, and unlock if the right combination is entered. The second design is a temperature control system that controls temperature within a specified range in a small chamber by using a fan for cooling and a lamp for heating. Both examples are simplified version of projects that can be easily extended in various directions.

## 6.1 Electronic Lock

An electronic lock is a circuit that recognizes a 5-digit input sequence (password) and indicates that the sequence is recognized by activating the unlock signal. A sequence of five 4-bit digits is entered using a hexadecimal keypad. If any of the digits in the sequence is incorrect, the lock resets and indicates that the new sequence should be entered from the beginning. This indication appears at the end of the sequence entry to increase the number of possible combinations that can be entered, making the task more difficult for a potential intruder.

The electronic lock consists of three major parts, as illustrated in the block diagram of Figure 6.1. The input device is a hexadecimal keypad with a keypad encoder. The keypad accepts a keypress and the keypad encoder produces the corresponding 4-bit binary code. The input sequence receives five-digit sequence of codes produced by the keypad encoder, compares it with the correct password and activates an unlock signal if the sequence is correct. The output consists of two parts. First part has three LEDs that indicate that indicate status of the lock. The lock can be ready to accept a new sequence, or is accepting (active) a sequence of five digits, or the correct sequence has been entered (unlock). The second part of the output is a piezo buzzer that produces a sound signal whenever a key correctly pressed (after debouncing valid key pressure detected). When a key is correctly pressed, the buzzer is held high for a short time interval of about 100 ms, corresponding to 5000 clock cycles of a 50kHz clock input.

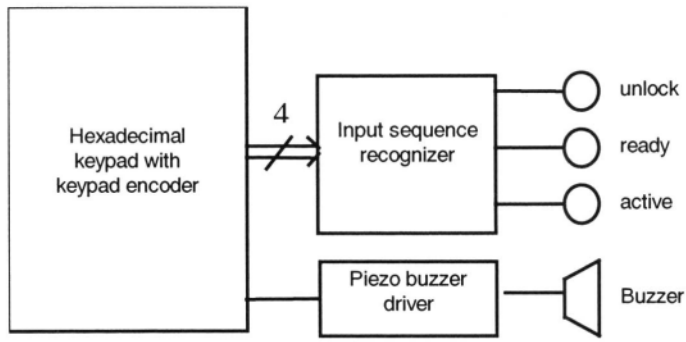



Figure 6.1 Electronic Lock

The lock is driven by an internal clock generator that can easily be implemented in MAX type FPLD as shown in Figure 6.2. Its frequency is determined by the proper selection of an external resistor R and capacitors C1 and C2.

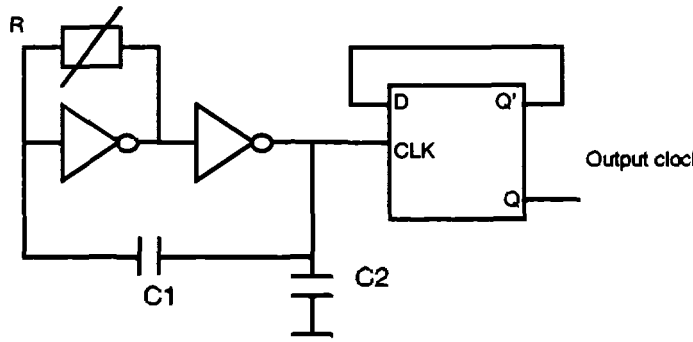



Figure 6.2 Internal clock implementation

### *6.1.1 Keypad encoder*

The keypad encoder controls the hexadecimal keypad to obtain the binary code of the key being pressed. Different hexadecimal keypad layouts are possible, with the one presented in Figure 6.3 being the most frequently used.

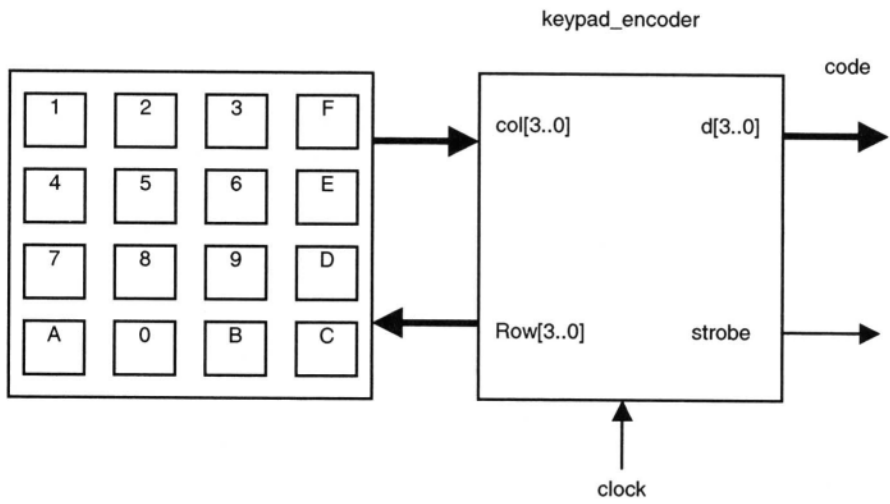



Figure 6.3 Hexadecimal keypad with encoder

The keypad encoder scans each row and senses each column. If a row and column are electrically connected, it produces the binary code for that key along with a strobe signal to indicate that the binary value is valid. The key press and produced binary code are considered valid if the debouncing circuitry, which is a part of the encoder, discovers the key has been held down for at least a specified time, in our example measured as 126 clock cycles. When the key is settled, a strobe output is activated to indicate to the external circuitry (input sequence recognizer and buzzer driver) that the key press is valid. The debounce circuitry also ensures that the key does not auto-repeat if it is held down for a longer time. Key debouncing circuit is described with the AHDL description given in Example 6.1.

Example 5.1 Key debouncing

```
SUBDESIGN debounce
(
    clk :INPUT;
    key_pressed :INPUT; % Key currently pressed %
    strobe :OUTPUT; % Key pressed for over 126 cycles %
)

VARIABLE
    count[6..0]  :DFF; % 7-bit counter for key_pressed %
                                % cycles% %
```

```
BEGIN
    count[].clk=clk;
    count[].clrn=key_pressed; % reset counter when no %
                        % key pressed %

IF (count[].q<=126) & key_pressed THEN
    count[].d=count[].q+1;
END IF;

IF count[].q==126 THEN
    strobe = Vcc;  %strobe produced when 126%
ELSE
    strobe = GND;
END IF;

END;
```

The keypad encoder is shown in Figure 6.4. The purpose of this diagram is to visualize what will be described in an AHDL text file.

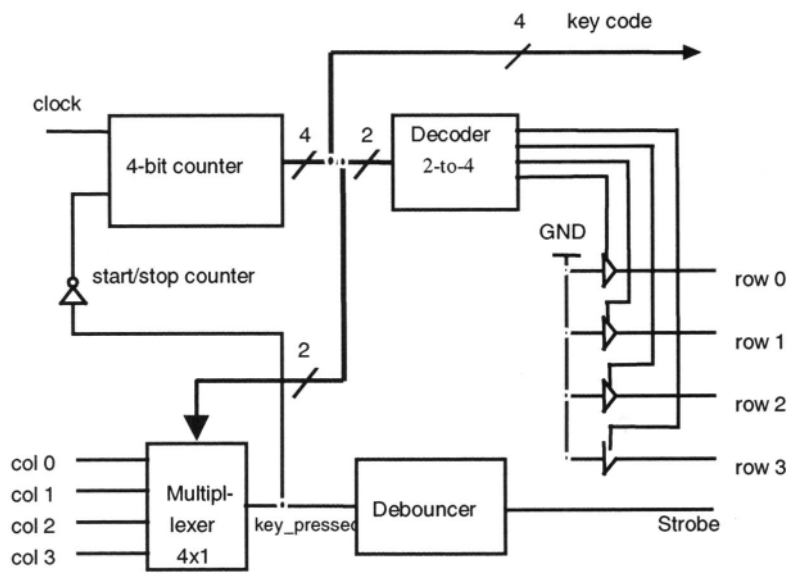



Figure 6.4 Hexadecimal keypad encoder

The 4-bit counter is used to scan and drives rows (two most significant bits) and select columns to be sensed (two low significant bits). When a key is pressed the counting process stops. The output of the 4-input multiplexer is used both as a

counter enable signal as well as the signal indicating that a key is pressed. The value at the counter outputs represents a binary code of the key pressed. Different mappings of this code are possible if needed. The debouncing circuit gets the information that a key is pressed, and checks if it is pressed long enough. Its strobe output is used to activate the buzzer driver and indicates the input sequence recognizer that a binary code is present and the keypad encoder output is valid.

The overall electronic lock is described in Figure 6.5 as a hierarchy of several simpler pans that will be designed first and then integrated into the final circuit. As the figure shows, the keypad encoder is in turn decomposed in a number of simpler circuits that can easily be described using AHDL.

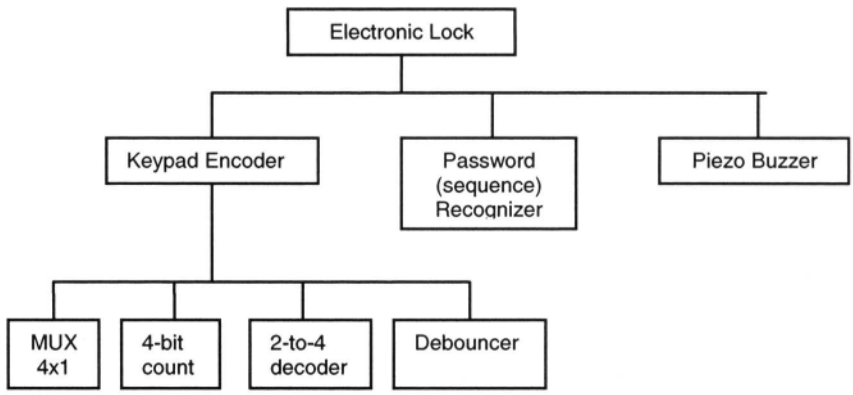



Figure 6.5 Electronic lock design hierarchy

We assume that the circuits used in the design of the keypad encoder, 4-to-1 multiplexer, 2-to-4 decoder, and the 4-bit counter, are already present in the library and will be included into the keypad encoder design. The keypad encoder design is shown in Example 6.2.

Example 6.2 Keypad Encoder.

```
TITLE "Keypad encoder";

INCLUDE  "4mux"; % Prototype for 4-to-l multiplexer %
INCLUDE  "24dec"; % Prototype for 2-to-4 decoder %
INCLUDE  "4count"; % Prorotype for 4-bit counter %
FUNCTION debounce(clk, key_pressed)
   RETURNS (strobe); % Prototype for debounce circuitry %
```

```
SUBDESIGN keyencode
(
   clk   :INPUT;
   col[3..0]   :INPUT; % Signals from keypad columns %
   row[3..0]   :OUTPUT; % Signals to keypad rows %
   key_code[3..0]   :OUTPUT; % Code of a key pressed %
   strobe    :OUTPUT; % Valid pressed key code %
)

VARIABLE

   key_pressed      :NODE; % Vcc when a key pressed %
   d[3..0]   :NODE; % Standard code for key %
   mux  :4mux; % Instance of 4mux %
   decoder  :24dec; % Instance of 24dec %
   counter  :4count; % Instance of 4count %
      opencol[3..0] :TRI;  % Tristated row outputs %

BEGIN
   row[] = opencol[].out;
   opencol[].in = GND; % Inputs connected to GND  %
   opencol[].oe = decoder.out[]; % Decoder drives keypad
                              rows %
   decoder.in[] = counter.(q3, q2) ;
   mux.d [] = col [] ;
   mux.sel  [] = counter.(q1,q0);
   key_pressed =  !mux.out;

% When a key is pressed its code appears on internal d[]lines%

   counter.clk = clk;
   counter.ena =  !key_pressed;
   d[] = counter.q[] ;

% Code conversion for different keypad topologies %

   TABLE
    d[]  =>  key_code[]; % Any code mapping %
    H"0" =>  H"1";
    H"1" =>  H"2";
    .
    .
    .
    H"F" =>  H"F";
   END TABLE;

   strobe = debounce(clk, keypressed);
END;
```

The above AHDL description contains one addition to the diagram shown in Figure 6.4. It is the truth table that implements conversion of the standard code, produced by the counter to any binary combination as the application requires. In Example 6.2 those two codes are identical, but obviously they can be easily changed to perform any required mapping. For that purpose, an internal signal group, d[ ], has been used to represent code generated by key pressure. Low level hierarchy components that are included into design are instantiated in Variable Section.

### *6.1.2 Input Sequence Recognizer*

An Input Sequence Recognizer (ISR) will be implemented as a sequential circuit that recognizes a predefined sequence of five hexadecimal (4-bit) digits. The presence of a digit at the input of the ISR is signified by a separate strobe input. When the five digit sequence has been correctly received, the unlock output line is activated until the circuit is reset. When the first incorrect digit is received, the ISR is reset to its initial state. Assuming the sequence of digits to be recognized is $D_1D_2D_3D_4D_5$, the state machine used to implement the ISR is described by the state transition diagram in Figure 6.6. It has six states, where SO is the initial state from which the ISR starts recognition of input digits.

After recognition of the required sequence, the circuit is returned to its initial state by the reset signal. The AHDL description of the ISR is shown in Example 6.3. Constants are used to define input digits, and can be easily changed as needed. In the case shown, the default input sequence is 98765.

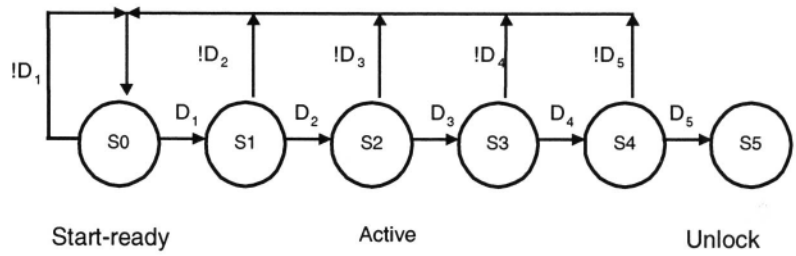



Figure 6.6 Input Sequence Recognizer State Transition Diagram

Example 6.3 Input Sequence Recognizer

```
TITLE "Input Sequence Recognizer";

% new password is set by changing following constants %
```

```
CONSTANT D1 = H"9"; % First digit in sequence %
CONSTANT D2 = H"8"; % Second digit in sequence %
CONSTANT D3 = H"7"; % Third digit in sequence %
CONSTANT D4 = H"6"; % Fourth digit in sequence %
CONSTANT D5 = H"5"; % Fifth digit in sequence %

SUBDESIGN isr
(
   clk  : INPUT; % Indicate that the input code is valid %
   strobe  :INPUT; %valid key pressed %
   i[3..0]  :INPUT; % 4-bit binary code of input digit %
   rst  :INPUT; % Reset signal %
   ready  :OUTPUT; % ready for new sequence %
   active  :OUTPUT; % entering sequence %
   unlock  :OUTPUT; % correct input sequence %
)

VARIABLE
   sm  :MACHINE WITH STATES (s0, s1, s2, s3, s4, s5) ;

BEGIN
   sm.(clk, reset) = (clk, rst);

CASE sm IS
   WHEN s0 =>
    IF i[]==D1 AND strobe THEN
       sm = s1;
    ELSE
       sm = s0;
    END IF;
   WHEN s1 =>
    IF i[]==D2 AND strobe THEN
       sm = s2;
    ELSE
       sm = s0;
    END IF;
   WHEN s2 =>
    IF i[]==D3 AND strobe THEN
       sm = s3;
    ELSE
       sm = s0;
    END IF;
   WHEN S3 =>
    IF i[]==D4 AND strobe THEN
       sm = s4;
    ELSE
       sm = s0;
   END IF;
   WHEN s4 =>
```

```
     IF i []==D5 AND strobe THEN
        sm = s5;
     ELSE
        sm = s0;
     END IF;
    WHEN s5 =>
     sm = s5;
END CASE;

IF sm==s0 THEN
    ready = Vcc;
    active = GND;
    unlock = GND;
ELSIF sm == s5 THEN
    ready = GND;
    active = GND;
    unlock = Vcc;
ELSE
    ready = GND;
    active = Vcc;
    unlock = GND;
END IF;

END;
```

The above design of ISR is unsafe as it is fairly easy to break the password. The user easily knows which digit is incorrect and it drastically reduces the number of required combinations to detect the password. It can be enhanced by adding states that make it safer and can still be user friendly. Two things can be readily made to reduce the probability of detecting the password:

- the number of possible combinations can be increased by increasing the number of states and
- five digits are always accepted before resetting the state machine without informing the user which digit in the sequence was incorrect.

The state transition diagram in Figure 6.7 illustrates one possible solution that takes above approach. The states F1, F2, F3, and F4 are introduced to implement described behavior. By "x" we denote any input digit.

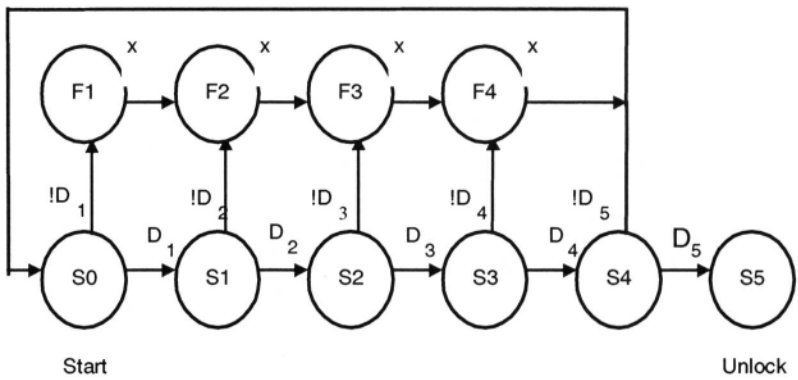



Figure 6.7 Modified ISR state transition diagram

The modified AHDL design of the ISR is given by Example 6.4.

Example 5.4 Modified Input Sequence Recognizer

```
TITLE "Modified Input Sequence Recognizer";

CONSTANT D1 = H"9" ; % First digit in sequence %
CONSTANT D2 = H"8"; % Second digit in sequence %
CONSTANT D3 = H"7"; % Third digit in sequence %
CONSTANT D4 = H"6"; % Fourth digit in sequence %
CONSTANT D5 = H"5"; % Fifth digit in sequence %


SUBDESIGN modified_isr
 (
    clk  :INPUT; % Indicate that the input code is valid %
    i[3..0]   :INPUT; % 4-bit binary code of input digit %
    strobe   :INPUT;
    rst  :INPUT; % Reset signal %
    start   :OUTPUT; % New sequence to be entered %
    more  :OUTPUT; % More digits to be entered %
    unlock  :OUTPUT; % Indicate correct input sequence %
)

VARIABLE
    Sm  :MACHINE WITH STATES
              (s0, s1, s2, s3, s4, s5, f1, f2, f3, f4);
```

```
BEGIN
    sm.(clk, reset) = (clk, rst);

CASE sm IS

    WHEN s0 =>
     start= Vcc; more = GND; unlock = GND;
     IF i[]==D1 AND strobe THEN
        sm = s1;
     ELSE
        sm = f1;
     END IF;
    WHEN s1 =>
     start= GND; more = Vcc; unlock = GND;
     IF i[]==D2 AND strobe THEN
        sm=s2;
     ELSE
        sm = f2;
     END IF;
    WHEN s2 =>
     start= GND; more = Vcc; unlock = GND;
     IF i[]==D3 AND strobe THEN
        sm = s3;
     ELSE
        sm = f3;
     END IF;
    WHEN S3 =>
     start= GND; more = Vcc; unlock = GND;
     IF i[]==D4 AND strobe THEN
        sm = s4;
     ELSE
        sm = f4;
     END IF;
    WHEN s4 =>
     start= GND; more = Vcc; unlock = GND;
     IF i[]==D5 AND strobe THEN
        sm = s5;
     ELSE
        sm = f5;
     END IF;
    WHEN S5 =>
     start= GND; more = GND; unlock = Vcc;
     IF strobe THEN
        sm = s5;
     END IF;
    WHEN f1 =>
     start= GND; more = Vcc; unlock = GND;
     IF strobe THEN
        sm = f2;
```

```
    END IF;
   WHEN f2 =>
    start= GND; more = Vcc; unlock = GND;
    IF strobe THEN
       sm = f3;
    END IF;
   WHEN f3 =>
    start= GND; more = Vcc; unlock = GND;
    IF strobe THEN
       sm = f4;
    END IF;
   WHEN f4 =>
    start= GND; more = Vcc; unlock = GND;
    IF strobe THEN
       sm = s0;
    END IF;
END CASE;
END;
```

The above design slightly differs to one from Example 6.3 in the way how outputs are generated. Obviously, in both cases we use Moore type state machine for the ISR implementation.

### *6.1.3 Piezo Buzzer Driver*

When a key sequence is correctly pressed, the buzzer is held high for approximately 100ms, which is 5000 clock cycles of a 50kHz clock input. The `beep` signal is used to drive the buzzer and indicate the user that the key sequence has been entered correctly. The AHDL description of the piezo buzzer driver is shown in Example 6.5.

Example 6.5 Piezo Buzzer Driver.

```
TITLE "Piezo Buzzer Driver";

SUBDESIGN beeper
 (
   Clk    :INPUT;
   strobe  :INPUT; % Valid key press %
   beep    :OUTPUT; % Piezo buzzer output %
 )

VARIABLE
   buzzer  :SRFF; % Buzzer SR flip-flop %
   count[12..0]     :DFF; % 13 bits for internal counter %
```

```
BEGIN
    count[].clk = clk;
    buzzer.clk = clk;
    buzzer.s = strobe; % set FF when key pressed %
    count[].clrn = buzzer.q; % clear counter when buzzer stops%

IF buzzer.q AND (count[].q < 5000) THEN
    % increment counter %
    count[].d = count[].q + 1;
END IF;

IF (count[].q == 5000) THEN
    % reset buzzer output %
    buzzer.r = Vcc;
END IF;

    beep = buzzer.q;

END;
```

### *6.1.4 Integrated Electronic Lock*

Once all components are designed, we can integrate them into the overall design of the electronic lock. The integrated lock is flexible in respect to different required input sequences (passwords) and different keypad topologies. The final design is represented in Example 6.6.

Example 6.6 Electronic Lock.

```
TITLE "Electronic Lock";

% Prototypes of components %
FUNCTION keyencode(clk, col[3..0])
    RETURNS (row[3..0], key_code[3..0], strobe);

FUNCTION modified_isr(clk, i[3..0], strobe, rst)
    RETURNS (start, more, unlock);

FUNCTION beeper(clk, strobe)
    RETURNS (beep);

SUBDESIGN lock
 (
    clk    :INPUT;
    reset   :INPUT;
    Col[3..0]  :INPUT;
    row[3..0]  :OUTPUT;
```

```
    Start   :OUTPUT;
    more    :OUTPUT;
    unlock     :OUTPUT;
    buzzer     :OUTPUT;
)

VARIABLE
    key_code[3..0]   :NODE;
    strobe   :NODE;

BEGIN
    buzzer = beeper(clk, strobe);
     (row[], key_code[], strobe) = keyencode(clk, col [] ) ;
     (start, more, unlock)= modified_isr(clk, key_code[],
               strobe, reset);

END;
```

## 6.2 Temperature Control System

The temperature control system of this example is capable of keeping the temperature inside a small chamber within a required range between 20° and 99.9°C. The required temperature range can be set with the user interface, in our case the hexadecimal keypad described in section 6.1. The temperature of the chamber is measured using a temperature sensor. When the temperature is below the lower limit of the desired range, the chamber is heated using an AC lamp. If it is above the upper limit of the range, the chamber is cooled using a DC fan. When the temperature is within the range, no control action is taken. The temperature is continuously displayed on a 3-digit hexadecimal display to one decimal place (for instance, 67.4°C). Additional LEDs are used to indicate the state of the control system. The overall approach to the design uses decomposition of the required functionality into several subunits and then their integration into the control system. A simplified diagram of the temperature control system is shown in Figure 6.8 It is divided into the following subunits:

- Temperature sensing circuitry provides the current temperature in digital form.
- Keypad circuitry is used for setting high and low temperature limits and for operator-controller communication.
- Display driving circuitry is used to drive 3-digit 7-segment display.
- DC fan control circuitry is used to switch on and off the DC fan.

- AC lamp control circuitry is used to switch on and off AC lamp.
- Control unit circuitry implements the control algorithm and provides synchronization of operations carried out in the controller. Since the purpose of the design is not to show an advanced control algorithm, a simple on/off control will be implemented.

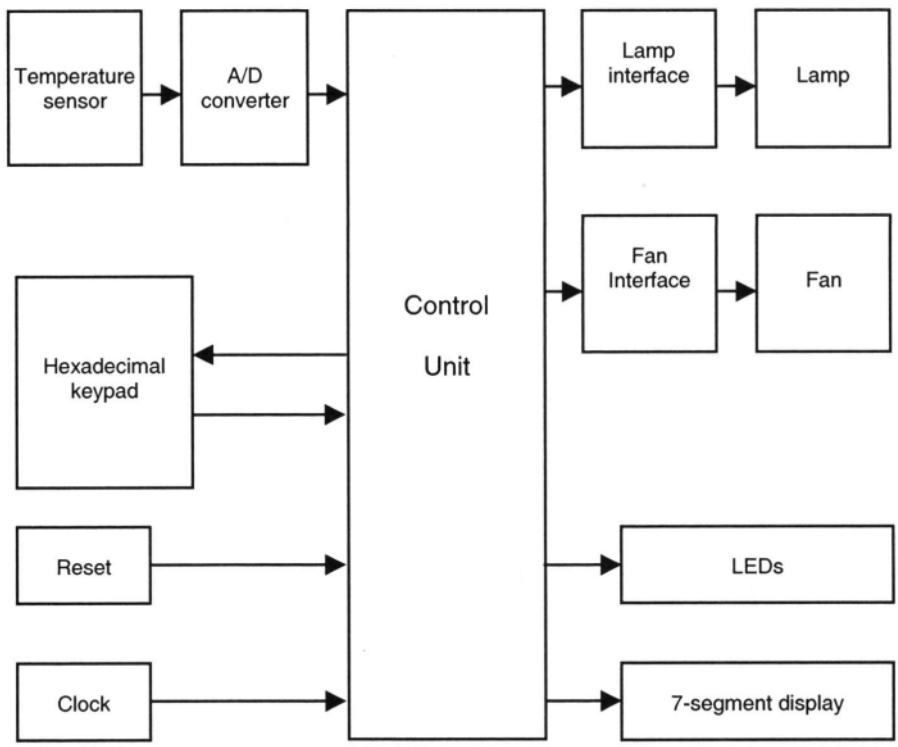



Figure 6.8 Block diagram of the temperature control system

The goal is to implement all interface circuits between the analog and digital parts of circuit, including the control unit, in an FPLD. Our design will be described by graphic and text entry tools provided in Max+Plus II design environment.

### *6.2.1 Temperature Sensing and Measurement Circuitry*

A sensor that measures the temperature inside the chamber is a transducer which converts a physical quantity, the temperature, to another more manageable quantity, a voltage. In our case we decided to use a common LM35 temperature sensor since it is a precision integrated circuit whose output voltage is linearly proportional to the temperature in Celsius degrees. Its important features are a large temperature range and satisfactory accuracy. Linear +10mV/°C scale factor in our case provides the output voltage between 0.2V and 0.99V for the temperature range from 20°C and 99.9°C. The analog voltage at the output of the sensor need to be converted to a digital form by an analog-to-digital (A/D) converter to be processed by the control unit.

A/D converters are available from many manufacturers with a wide range of operating characteristics. For our application, we have chosen the ADC0804 CMOS A/D converter which uses the successive approximation conversion method. Its pinout is illustrated in Figure 6.9.

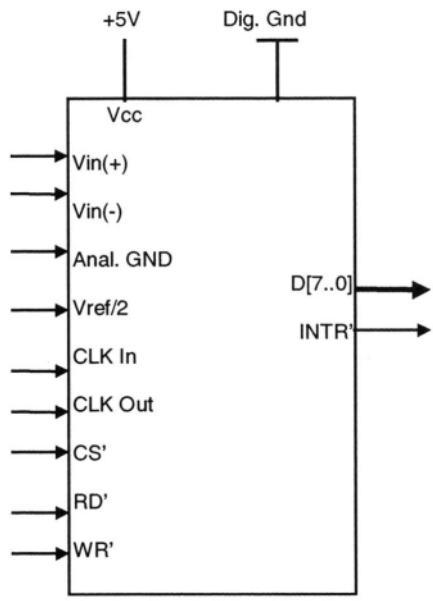



Figure 6.9 ADC0804

The ADC0804 major features are:

- Two analog inputs allowing differential inputs. In our case only one, Vin (+), is used. The converter uses Vcc=+5V as its reference voltage.
- Analog input voltage is converted to an 8-bit digital output which is tri-state buffered. The resolution of the converter is 5V/255=19.6mV.
- In our case Vref/2 is used as an input to reduce the internal reference voltage and consequently the analog input range that converter can handle. In this application, the input range is from 0 to 1V, and Vref/2 is set to 0.5V in order to achieve better resolution which is 1V/255=3.92mV
- The externally connected resistor and capacitor determine the frequency of an internal clock which is used for conversion (in our case the frequency of cca 600KHz provides the conversion time of 100us).
- The chip Select, CS', signal must be in its active-low state for RD' or WR' inputs to have any effect. With CS' high, the digital outputs are in the high-impedance state, and no conversion can take place. In our case this signal is permanently active.
- RD' (Read, Output Enable) is used to enable the digital output buffers to provide the result of the last A/D conversion. In our case this signal is permanently active.
- WR' (Write, Start of Conversion) is activated to start a new conversion.
- INTR' (End of Conversion) goes high at the start of conversion and returns low to signal the end of conversion.

It should be noticed that the output of the A/D converter is an 8-bit unsigned binary number which can be used by the control unit for further processing. However, this representation of the current temperature is not suitable for display purposes, in which case we prefer a BCD-coded number, or for comparison with the low and high temperature limits, which are entered using the keypad and are also in the BCD-coded form. This is the reason to convert the temperature into a BCD-coded format and do all comparisons in that format. This may not be the most effective way of manipulating temperatures. Therefore other possibilities can be explored.

The control unit is responsible for starting and coordinating the activities of A/D conversion, sensing the end of the conversion signal, and generating the control signals to read the results of the conversion.

### 6.2.2 Keypad Control Circuitry

The high and low temperature limits are set using a hexadecimal keypad as shown in Section 5.1. Besides digits 0 trough 9 used to enter values of temperature, this keypad provides the keys, A through F, which can be used as function keys. In our case we assign the functions to functional keys as shown in Table 6.1.

Table 6.1 Key Functions

| Key Code | Function |
|---|---|
| A | SL - Set low temperature limit |
| B | SH - Set high temperature limit |
| C | DL -Display low temperature limit |
| D | DH - Display high temperature limit |
| E | Enter; Return to running mode |

Each time, a function key to set the temperature is pressed first and then the value of the temperature limit. This pressing brings the system into the setting mode of operation. The entry is terminated by pressing E and the system is returned to the running mode of operation. When an incorrect value of the temperature limit is entered, the reset button is to be pressed to bring the system into its initial state and allow repeated entry of the temperature limits. In the case of pressing function keys for displaying temperature limits, the system remains in the running mode, but a low or high limit is displayed. The low and high temperature limits have to be permanently stored in internal control unit registers and available for comparison to the current measured temperature. In order to change the low and high limit, new values have to be passed to the correct registers. The overall structure of the keypad circuitry is illustrated in Figure 6.10. In addition to the already shown keypad encoder, this circuitry contains a functional key decoder, which recognizes pressing of any functional key. Then, it activates the corresponding signal used by the control unit, and two 12-bit registers used to store two 3-digit BCD-coded values of the high and low temperature limit.

The values are entered into these registers digit-by-digit in a sort of FIFO arrangement, and are available on register output in the parallel form, as illustrated in Figure 6.11. As soon as the A or B key is pressed, the corresponding register is cleared, zeros displayed on the 7-segment displays, and the process of entering a new value can start. The register select signal is used to select to which of two registers, HIGH or LOW, will be forwarded.

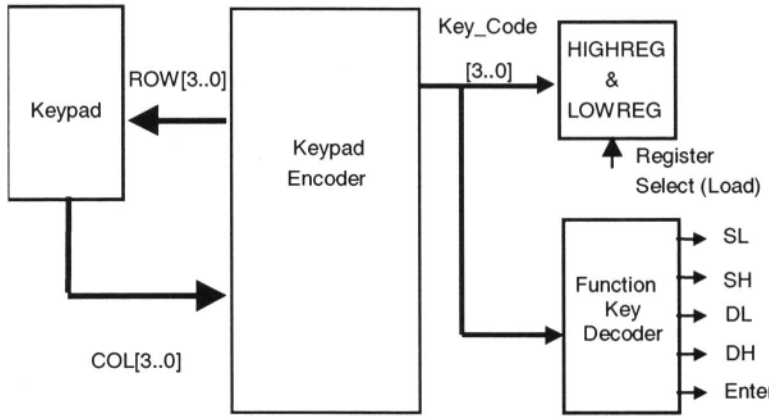



Figure 6.10 Keypad circuitry

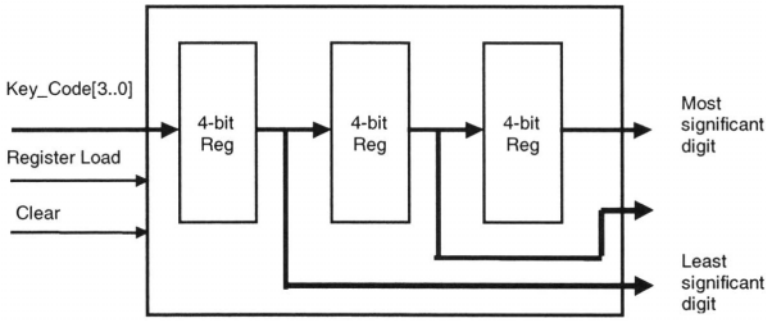



Figure 6.11 Low and high temperature limit registers

### *6.2.3 Display Circuitry*

Display circuitry provides the display for any one of the possible temperatures which are in the range from 00.0°C to 99.9°C. We decided to use 7-segment displays to display individual digits, as shown in figure 6.12.

Since the 7-segment display accepts on its inputs a 7-segment code, it is required to convert the BCD coded digits to 7-segment coded digits. The simplest method is to use a decoder for each digit and operate each independently. The other possibility

is to use time division multiplexing. The displays are selected at the same time as the digits are to be passed through the multiplexer, as illustrated in Figure 5.11. The input to the display circuitry is 12 bits, representing the 3-digit BCD-coded temperature. A modulo-2 counter selects a digit to display and at the same time a 2-to-3 decoder selects a 7-segment display.

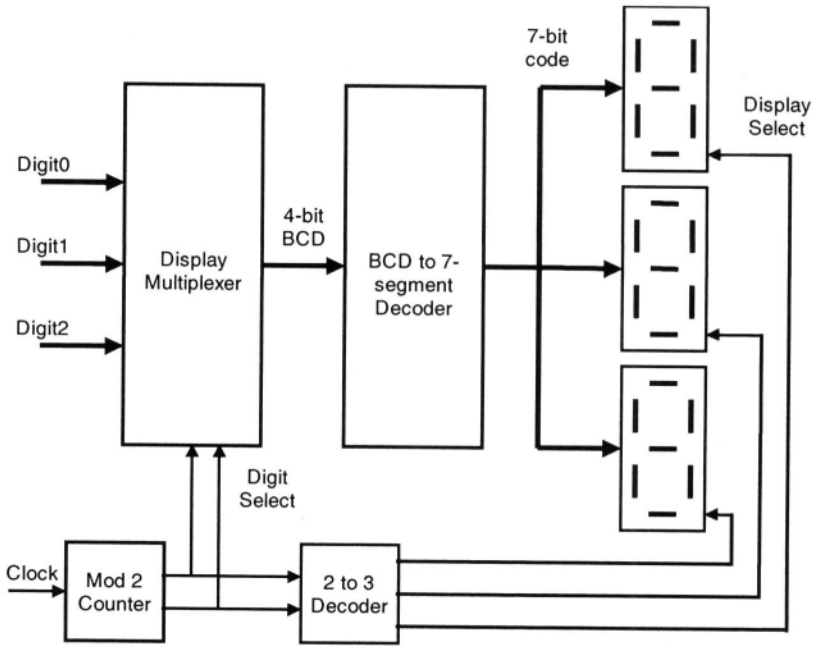



Figure 6.12 Display circuitry

Selection of the currently displayed temperature is done by a 3-to-l multiplexer illustrated in Figure 6.13. Inputs to this multiplexer are the current temperature and the low and high temperature limits. The control unit selects, upon request, the temperature which will appear on the 7-segment displays.

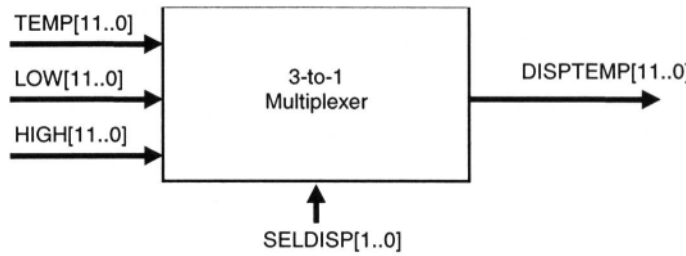



Figure 6.13 Multiplexer for selection of the temperature to display

### *6.2.4 Fan and Lamp Control Circuitry*

A control circuit using a highside driver to control the switching of the DC motor of a fan is used. An example of such a driver is National LM1051. Its worst case switching times for both turn on and off are 2us. A brushless DC fan with a 12V nominal operating voltage and low power consumption was chosen. The DC fan control circuitry is controlled by only one signal from the control unit providing on and off switching of the fan, as it is illustrated in Figure 6.14.

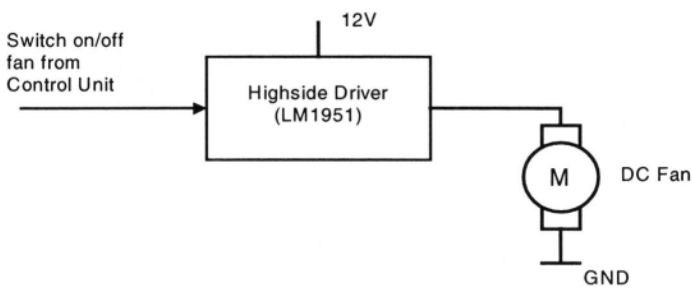



Figure 6.14 DC fan control circuitry

We choose a 150W infra heat AC lamp with instantaneous response and no warm up or cool down delay for heating up the chamber. To control the lamp with a digital signal, we used a triac and a zero crossing optically-coupled triac driver. The triac is a 3-terminal AC semiconductor switch which is triggered into conduction when a low energy signal is applied to its gate terminal. An effective method of controlling the average power to a load through the triac is by phase control. That is,

to apply the AC supply to the load (lamp) for a controlled fraction of each cycle. In order to reduce noise and electromagnetic interference generated by the triac, we used a zero crossing switch. This switch ensures that AC power is applied to the load either in full or half cycles. The triac is gated at the instant the sine wave voltage is crossing zero. An example of a zero crossing circuit is the optoisolator MOC3031, which is used in interfacing with AC powered equipment. The entire AC lamp control circuitry is illustrated in Figure 6.15. It is controlled by signal generated from the control unit to switch the lamp on or off.

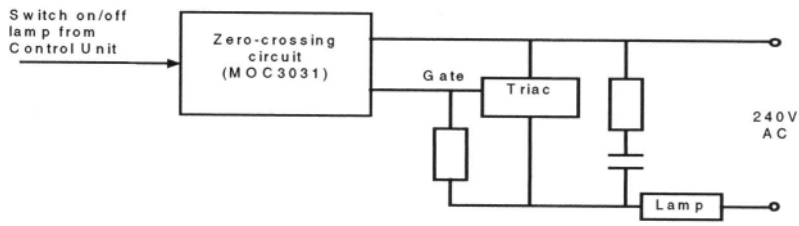



Figure 6.15 AC lamp control circuitry

### *6.2.5 Control Unit*

The control unit is the central point of the design. It provides proper operation of the temperature control system in all its modes including temperature sensing, switching between modes, communication with the operator, control of data flow, and data processing in the data path of the circuit. The main inputs to the control unit are current temperature (including synchronization signals for temperature measurement), high and low temperature limit, and the inputs from the keypad. The main outputs from the control unit are

- signals to select different data paths (to compare current temperature with the low and high temperature limit),
- signals to control on/off switching of the fan and lamp,
- signals to control temperature measurement, and
- signals to control the displays (7-segment and LEDs).

The operation of the control unit is illustrated by the flow diagram in Figure 6.16.

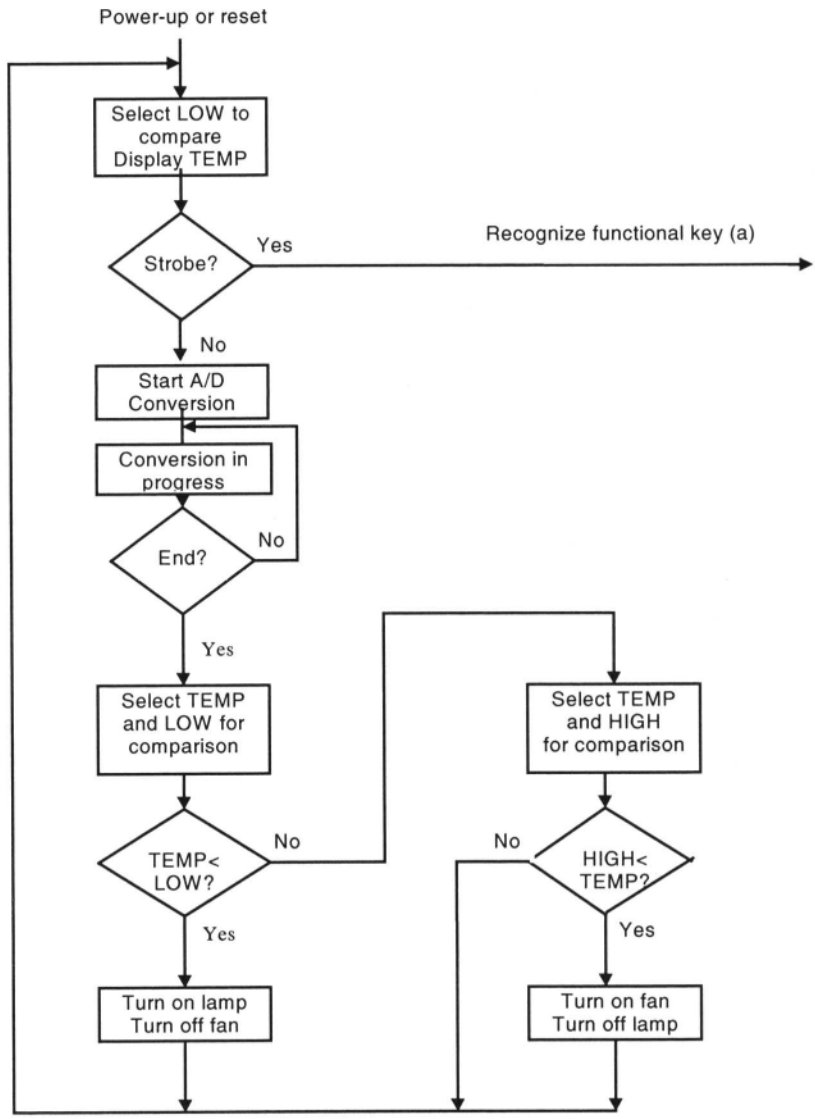



Figure 6.16 Control unit flow diagram

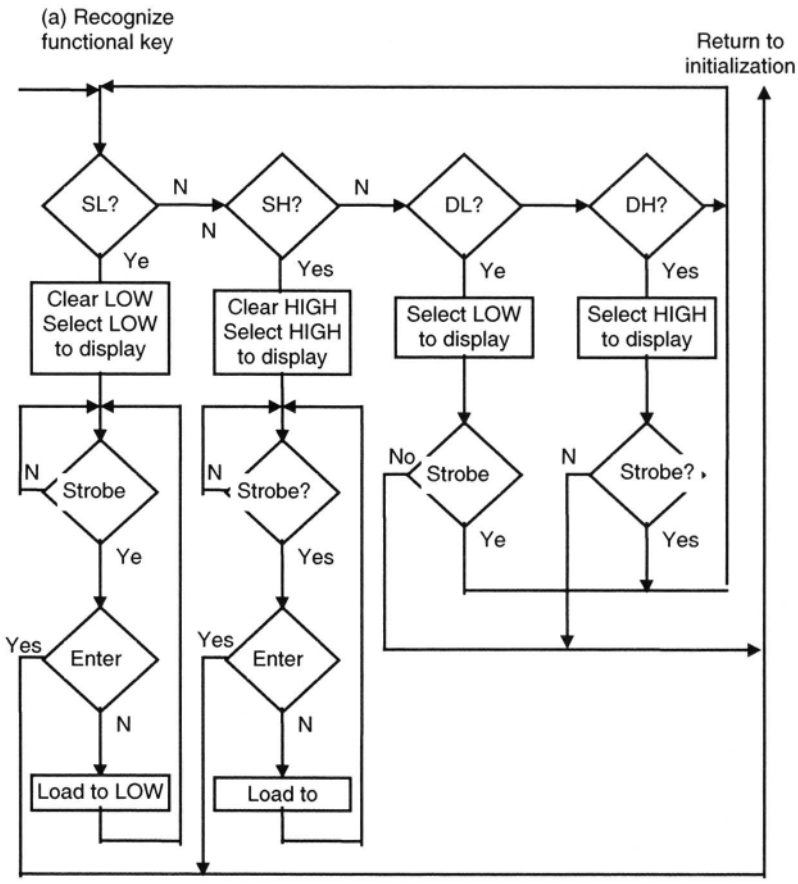



Figure 6.16 Control Unit Flow Diagram (continued)

After power-up or reset, the control unit passes the initialization phase (where all registers are cleared), low temperature limit, LOW, is selected for comparison first, and current temperature, TEMP, is selected to be displayed. After that, the control unit checks the strobe signal from the keypad. If there is no strobe signal, the control unit enters the running mode, which includes

- a new A/D conversion cycle to get the value of the TEMP,
- comparison of the TEMP with the LOW in order to turn on lamp, or
- comparison of the TEMP with the high temperature limit, HIGH, in order to turn on the fan.

This process is repeated until a key press is detected. When a key press is detected, the control unit recognizes the code of the key and appropriate operations are carried out. Upon completion of these operations, the system is returned to the running mode. The following state machine states implement the control unit behavior:

S0 - Initialization, upon power-up or reset

S1 - Start A/D conversion

S2 - A/D conversion in progress

S3 - Comparison of current temperature and low temperature limit

S4 - Comparison of current temperature and high temperature limit

S5 - Setting low temperature limit

S6 - Setting high temperature limit

S7 - Display low temperature limit

S8 - Display high temperature limit

Table 6.2 Control Unit Inputs and Outputs.

| Input | Function |
|---|---|
| END | End of A/D conversion - active LOW signal |
| STROBE | HIGH pulse from Keypad encoder indicating valid key press |
| SL, SH, DL, DH, Enter | Indication of active functional keys received from Keypad encoder |
| AGEB | Indication that temperature A is greater or equal to temperature B |
| ALTB | Indication that temperature A is lower than temperature B |
| Output | Function |
| START | LOW pulse sent to the WR' pin of A/D converter to enable start of the next conversion |
| SET_LOW | Signal indicating that the control system is in setting the LOW temperature limit - used to switch on LOW LED |
| SET_HIGH | Signal indicating that the control system is in setting the HIGH temperature limit - used to switch on HIGH LED |
| CLR_LOW | Signal for clearing the value of the LOW register |
| CLR_HIGH | Signal for clearing the value of the HIGH register |
| LD_LOW | Signal used to load the LOW register with the next BCD- |

| | |
|---|---|
| | coded digit received from keypad at the presence of the STROBE signal |
| LD_HIGH | Signal used to load the HIGH register with the next BCD-coded digit from the keypad at the presence of the STROBE signal |
| SELHILO | Select signal of the multiplexer used to select either HIGH or LOW temperature limit to be forwarded to the comparator and compared to the current temperature TEMP |
| SELDISP[1..0] | Select signal of the multiplexer used to select which signal will be displayed on the 7-segment displays |
| FAN_ON | Signal used to turn on the DC fan and at the same time to turn off the AC lamp |
| LAMP_ON | Signal used to turn on the AC lamp and at the same time to turn off the DC fan |

Finally, we identify inputs and outputs of the control unit. If we assume all data transfers and processing are done within the data path, then the inputs and outputs of the control unit are identified as shown in Table 6.2.

### *6.2.6 Temperature Control System Design*

Our approach to the overall design of the temperature control system follows a traditional line of partitioning into separate design of the data path and the control unit, and their easy integration into the target system. This approach is illustrated in Figure 6.17.

The data path provides all facilities to exchange data with external devices, while the control unit uses internal input control signals generated by the data path, external input control signals to generate internal control outputs that control data path operation, and external control outputs to control external devices.

The data path of the temperature control system is made up of all circuits that provide: interconnections, the data path, and the data transformation:

1. Interconnections between main input subunits (hexadecimal keypad and A/D converter) and main output subunits (7-segment displays, LED displays, and control signals that switch on and off DC fan and AC lamp)
2. Data paths to and from internal registers that store low and high temperature limits.

3. Data transformations and processing (binary to BCD-coded format of the current temperature, comparison of current temperature to low and high temperature limits)

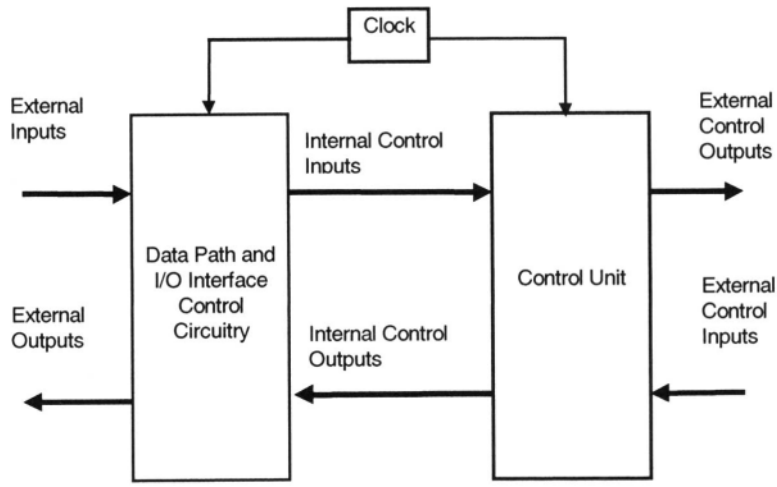



Figure 6.17 Global approach to temperature control system design

The temperature control system data path, together with the circuits that control input and output devices is shown in Figure 6.18. All control signals, except those for clearing the LOW and HIGH registers, generated by the control unit are also shown.

The designs of almost all subunits of the data path have been already shown in various parts of the book. Remaining details and integration to the overall data path are left to the reader as an exercise.

Finally, we come to the design of the control unit explained in Section 6.2.5. This design is easily transformed into the corresponding AHDL description of the state machine. The AHDL design file of the control unit with appropriate comments is shown as Example 6.7.

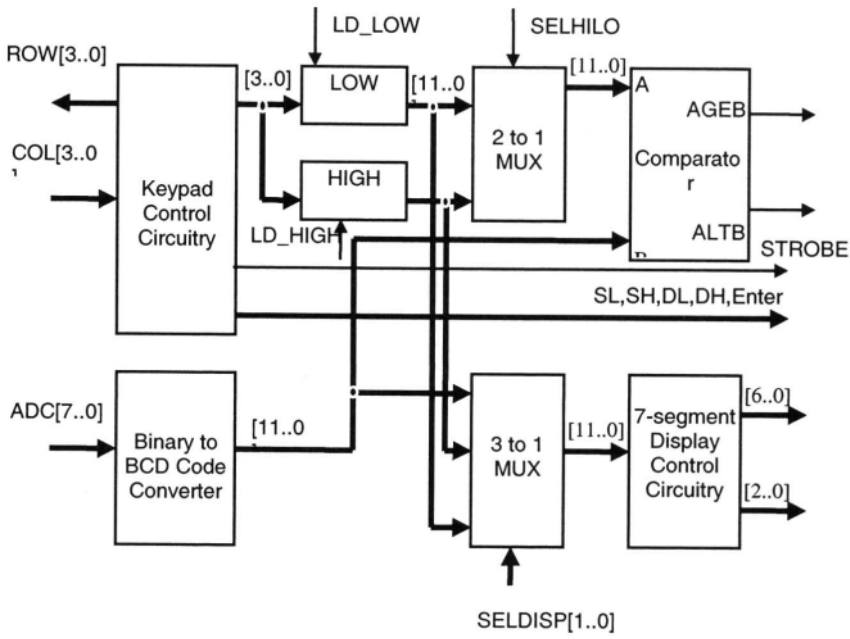



Figure 6.18 Temperature Control System Data Path with interfacing circuitry

Example 6.7 Temperature Control System Unit.

```
TITLE "Temperature Control System Control Unit";

SUBDESIGN ctrlunit
 (
    clk   :INPUT;
    reset   :INPUT;
    sl,sh,dl,dh,enter  :INPUT; % functional key activation
              inputs %
    endc   :INPUT; % end of a/d conversion %
    strobe   :INPUT; % keypress valid signal %
    ageb   :INPUT; % compared temp >= current temp %
    altb   :INPUT; % compared temp < current temp %
    start   :OUTPUT; % start a/d conversion %
    set_low,set_high  :OUTPUT; % activate external leds %
    clr_low,clr_high  :OUTPUT; % clear low, high register %
    ld_low, ld_high  :OUTPUT; % load data into register file%
    selhilo  :OUTPUT; % select reg to be compared %
    seldisp [1..0]     :OUTPUT; %select temperature to display %
```

```
    fan_on,lamp_on  :OUTPUT; % turn on/off fan/lamp %
    oe_adc   :OUTPUT; % read data from a/dc %
)

VARIABLE
    sm: MACHINE WITH STATES
     (s0,s1,s2,s3,s4,s5,s6,s7,s8);

BEGIN
    sm.(clk, reset)=(clk, reset);

CASE sm IS

    WHEN s0 =>
     set_low = GND;
     set_high = GND;
     selhilo=GND; % select low to compare %
     seldisp[]=B"00"; % select temp to display %
     IF !strobe THEN
        sm = s1;
     ELSIF strobe & sl THEN
        clr_low = Vcc;
        seldisp[] = B"01"; % low to display %
        set_low = Vcc; % turn on set_low led %
        sm = s5;
     ELSIF strobe & sh THEN
        clr_high = Vcc;
        seldisp[] = B"10"; % high to display %
        set_high = Vcc; % turn on set_high led %
        sm = s6;
     ELSIF strobe & dl THEN
        seldisp[] = B"01"; % low to display %
        sm = s7;
     ELSIF strobe & dh THEN
        seldisp[] = B"10"; % high to display %
        sm = s8;
     ELSE
        sm = s0; % functional key not pressed %
     END IF;

    WHEN s1 =>
     start = Vcc; % start a/dc %
     sm = s2;

    WHEN s2 =>
     IF endc THEN
        oe_adc = Vcc; % read a/dc %
        sm = s3;
     ELSE
```

```
    sm = s2;
 END IF;

WHEN s3 =>
 IF ageb THEN % low >= temp %
    lamp_on = Vcc;
    sm = s0;
 ELSE
    sm = s4;
 END IF;

WHEN s4 =>
 selhilo = Vcc; % select high to compare %
 IF altb THEN % high < temp %
    fan_on = Vcc;
    sm = s0;
 ELSE
    sm = s0;
 END IF;

WHEN s5 =>
 IF !strobe THEN
    sm = s5; % wait for key press %
 ELSIF strobe & enter THEN
    sm = s0;%enter pressed-new value entered %
 ELSIF strobe & !enter THEN
    ld_low = Vcc; % new digit of low %
    sm = s5;
 END IF;

WHEN s6 =>
 IF !strobe THEN
    sm = s6; % wait for key press %
 ELSIF strobe & enter THEN
    sm = s0;%enter pressed-new value entered %
 ELSIF strobe & !enter THEN
    ld_high = Vcc; % new digit of high %
    sm = s6;
 END IF;

WHEN s7 =>
 seldisp[] = B"01"; % low to display %
 IF strobe & dl THEN
    sm = s7; % dl kept pressed %
 ELSE
    sm = s0; % dl released %
 END IF;

WHEN s8 =>
```

```
      seldisp[] = B"10"; % high to display %
      IF strobe & dh THEN
         sm = s8; % dh kept pressed %
      ELSE
         sm = s0; % dh released %
      END IF;

END CASE;

END;
```

We have done compilation experiments with different target devices to investigate resource utilization. It easily fits into both devices available on the Altera UP-1 board.

## 6.3 Problems and Questions

6.1 An electronic lock receives password consisting of seven hexadecimal characters so that the whole password is stored in the lock before being matched with the correct hard-coded value. Modify the electronic lock from this chapter to provide this feature.

6.2 An electronic lock has the ability to change the password when required. Make the necessary modification that will enable change of the password. In the case of power failure password can be lost, but still the lock provides its initial hard-coded password as the valid one. Make necessary modifications of the design to provide this feature.

6.3 The temperature controller from this chapter implements a more realistic control algorithm maintaining the temperature within the required limits in the following way:

- If current temperature is rising and reaches the upper limit, the heater is deactivated and fan activated. When the temperature to for one quarter of interval between upper and lower limits, fan is also deactivated and temperature starts decreasing without any action.

- If the temperature is falling and reaches the lower limit, the heater is activated until the upper temperature limit is achieved.

Implement this control algorithm by modifying one shown in the example.

6.4 Temperature controller implemented in hardware can control temperature in a number of incubators. Modify controller design in order to enable controlling

temperature in eight incubators at the same time. Assume that you are using an 8-channel A/D converter that has three address lines to select the channel for which conversion is performed.

6.5 Modify the display circuitry and man-machine interface of the temperature controller to enable display of the current temperature and the channel number by adding another 7-segment LED for this purpose. Enable an operator to require display of temperature in individual incubators on request. In normal mode current temperature and channel number are displayed in turn for around five seconds for each channel.

6.6 Change the man-machine interface in the temperature controller from Problem 6.4 to enable communication using only three buttons (keys) that are used to transfer the controller into:

- setting mode in which keys are used to initialize upper and lower temperature limits,
- normal periodic mode in which the controller maintains temperature in the incubators and displays current values of temperatures periodically for each channel,
- on-demand mode in which it enables to select the channel for which temperature will be displayed.

6.7 Assume that an averaging filter filters the samples from the A/D converter that represent binary values of the temperature. Eight successive temperature measurements are taken and their average value is taken as a represent of the current temperature. Add the circuitry that provides this way of sampling and calculates the current temperature value.

6.8 The current temperature, represented by the binary value obtained as the result of processing in Problem 6.7, is taken as an input to the look-up table that stores physical values of temperatures. The temperatures are stored in using BCD coded digits as shown in example controller. Eight most significant bits are used to represent the integer part, and further four bits used to represent the decimal part of the temperature. Look-up table contains altogether 256 entries. Redesign the temperature controller to use look-up table in both setting and running mode and assume that the human operator is always using physical representation of temperatures. You are allowed to use embedded array blocks and FLEX 10K family devices to solve the problem.

# 7 SIMP - A SIMPLE CUSTOMIZABLE MICROPROCESSOR

In this chapter we will discuss and present the design of a simple 16-bit customizable microprocessor called SimP. The SimP can be considered the core for various user specific computing machines. It consists of a set of basic microprocessor features that can be used without any changes for some simple applications, or can be extended by the user in many application specific directions. Extensions can be achieved by adding new instructions or other features to the SimP's core, or by attaching functional blocks to the core without actually changing the core. SimP should be considered as an open project. Its further extensions are shown in Chapter 9. The design of SimP is completely specified in AHDL, and the methodology of adding new features is shown at the appropriate places. As a slightly more complex design, SimP represents a good exercise in a typical data path/control unit type digital system.

## 7.1 Basic Features

The basic features of the SimP core are:

- 16-bit data bus and 12-bit address bus that enable direct access to up to 4096 16-bit memory locations
- two programmer visible 16-bit working registers, called A and B registers, which are used to store operands and results of data transformations
- memory-mapped input/output for communication with the input and output devices
- basically a load/store microprocessor architecture with a simple instruction cycle consisting of four machine cycles per each instruction; all data transformations are performed in working registers
- support of direct and the most basic stack addressing mode, as well as implicit addressing mode

- definable custom instructions and functional blocks which execute custom instructions can be added
- implemented in a low capacity FPLD from the FLEX 8000 or 10K family, with an active serial configuration scheme
- physical pin assignments can be changed to suit the PCB layout.

### *7.1.1 Instruction Formats and Instruction Set*

The SimP instructions have very simple formats. All instructions are 16-bits long and require one memory word. In the case of direct addressing mode, 12 lower instruction bits represent an address of the memory location. All other instructions for basic data processing, program flow control, and control of processor flags use implied addressing mode. The core instruction set is illustrated in Table 7.1.

Table 7.1 SimP instruction set

| Instruction mnemonic | Function |
|---|---|
| LDA | A ← M[address] |
| LDB | B ← M[address] |
| STA | M[address] ← A |
| STB | M[address] ← B |
| JMP | PC ← address |
| JSR | stack ← PC, PC ← address, SP ← SP-1 |
| ADD | A ← A + B |
| AND | A ← A and B |
| CLA | A ← 0 |
| PSHA | stack ← A, SP ← SP-1 |
| PULA | SP ← SP+1, A ← stack |
| CLB | B ← 0 |
| CMB | B ← B' |
| INCB | B ← B + 1. |
| DECB | B ← B - 1 |
| CLflag | flag ← 0 (flag can be Carry,Zero) |
| ION | IEN ← 1, Enable interrupts |
| IOF | IEN ← 0, Disable interrupts |
| SZ | If Z=1, PC ← PC + 1; skip if zero set |
| SC | If C=1, PC ← PC + 1; skip if carry set |
| RET | SP ← SP+1, PC ← stack |

All memory reference instructions use either direct or stack addressing mode and have the format as shown below:




The four most significant bits are used as the operation code (opcode) field. As such operation code field can specify up to 16 different instructions. Twelve least significant bits are used as an address for instructions with direct addressing mode or they have no meaning for instructions using stack (implicit) addressing mode. Although the stack pointer (SP) is present, it is not directly visible to programmer. It is initialized at the power-up of the microprocessor to the value FF0 (hex) and subsequently changes its value as instructions using stack or external interrupts occur.

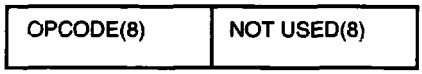
Memory reference instructions with the direct and stack addressing modes are assigned the opcodes as shown in Table 7.2.

Table 7.2 Opcode Instruction Mnemonics

| Opcode i[15..12] | Instruction Mnemonic |
|---|---|
| 0000 | LDA |
| 0001 | LDB |
| 0010 | STA |
| 0011 | STB |
| 0100 | JMP |
| 1000 | JSR |
| 1010 | PSHA |
| 1100 | PULA |
| 1110 | RET |

Instructions in direct addressing mode, (that do not use the stack) have the most significant bit i[15] equal to 0. Those that use the stack have bit i[15] equal to 1. Although it is not shown here, the instructions which belong to the register reference instructions and are not using the stack have the bit i[15] equal to 0 and four most significant bits equal to 7 (hex), and instructions that operate on user-specific functional blocks have the bit i[15] equal to 1 and four most significant bits

equal to F (hex). The remaining SimP core instructions have the following instruction formats




and are assigned the opcodes as shown in Table 7.3.

Table 7.3 Instruction Mnemonics

| Opcode i[15..8] | Instruction Mnemonic |
|---|---|
| 0111 0001 | ADD |
| 0111 0010 | AND |
| 0111 0011 | CLA |
| 0111 0100 | CLB |
| 0111 0101 | COMB |
| 0111 0110 | INCB |
| 0111 0111 | DECB |
| 0111 1000 | CLC |
| 0111 1001 | CLZ |
| 0111 1010 | ION |
| 0111 1011 | IOF |
| 0111 1100 | SC |
| 0111 1101 | SZ |

Register reference instructions operate on the contents of working registers (A and B), as well as on individual flag registers used to indicate different status information within the processor or to enable and disable interrupts. Examples of those instructions are ADD and DEC instructions.

Finally, program flow control instructions are used to change program flow depending on the results of current computation are simple "skip if zero or carry" set (SZ and SC). These instructions in combination with unconditional JMP instruction can achieve conditional branching to any memory address.

Besides the shown instructions, the SimP provides instructions that invoke different application specific functional blocks. These instructions are designated

with instruction bits i[15] and i[12] set to 1. The individual instructions are coded by the least significant bits i[7..0].

### *7.1.2 Register Set*

SimP contains a number of registers that are used in performing microoperations. Some are accessible to the user, and the others are used to support internal operations. All user visible registers are shown in Figure 7.1.

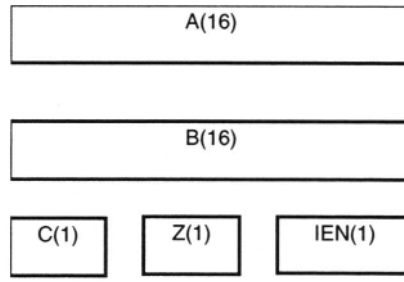



Figure 7.1 User Visible Registers

Registers that are not visible to the user include 12-bit program counter, PC, and a 12-bit stack pointer, SP. The program counter is used during instruction execution to point to the next instruction address. The stack pointer is used to implement the subroutine and interrupt call, and return mechanism (to save and restore return addresses). It also supports the execution of "push to the stack from A" (PSHA) and "pull from the stack to A" (PULA) instructions. At system power-up, the program counter is loaded with the value H"000" and the stack pointer with the value H"FFO". The stack grows towards the lower addresses. Two further registers, the program counter temporary register (TEMP), and the auxiliary stack pointer register (ST), are neither directly nor indirectly accessible by the user. They are used in internal operations to save values (copies) of the program counter and stack pointer and provide a very simple, uniform instruction execution cycle. As we shall see, all instructions execute in exactly four machine (clock) cycles, thus providing eventual performance improvement using pipelining in an advanced version of the microprocessor.

## 7.2 Processor Data Path

Overall SimP structure is presented in the simplified data path of Figure 7.2. The processor contains two internal buses: a 16-bit data bus and a 12-bit address bus.

The data bus is connected to the external pins and enables easy connection with external memory of up to 4,096 16-bit words or to the registers of input and output devices in a memory mapped scheme. The external data bus appears as bi-directional or as separated input and output data lines, while internally it provides separated input and output data bus lines. The address bus is available only for internal register transfers and enables two simultaneous register transfers to take place. Externally, it appears as uni-directional 12-bit address bus.

Additional registers not visible to the user appear in the in the internal structure of the data path. They are the 16-bit instruction register (IR) and the 12-bit address register (AR). The instruction register is connected to the instruction decoder and provides input signals to the control unit. The details of the use of all registers will be explained in upcoming sections.

The Arithmetic-Logic Unit (ALU) performs simple arithmetic and logic operations on 16-bit operands as specified in the instruction set. In its first version, the ALU performs only two operations, unsigned "addition" and "logical and." It can easily be extended to perform additional operations. Some data transformations, such as incrementation and initialization of working registers, are carried out by the surrounding logic of working registers A and B.

Access to the external memory and input output devices is provided through multiplexers that are used to form buses. An external address is generated on the address lines, A[11..0], as the result of selected lines of the memory multiplexer, (MEMMUX). Usually, the effective address is contained in the address register AR, but in some cases it will be taken from another source, stack pointer (SP) or auxiliary stack pointer (ST).

Two other multiplexers (used to form the address and data bus) are not shown in Figure 7.2 but later when we discuss data path implementation. However, it is obvious that several registers or memory can be the source of data on both buses. Two multiplexers, the address bus multiplexer (ABUSMUX) and data bus multiplexer (DBUSMUX) are used to enable access to address and data bus, respectively. The only register that can be the source of data for both these buses is the program counter (PC). If the content of the program counter is transferred by way of data bus, only the 12 least significant data lines are used for the actual physical transfer.

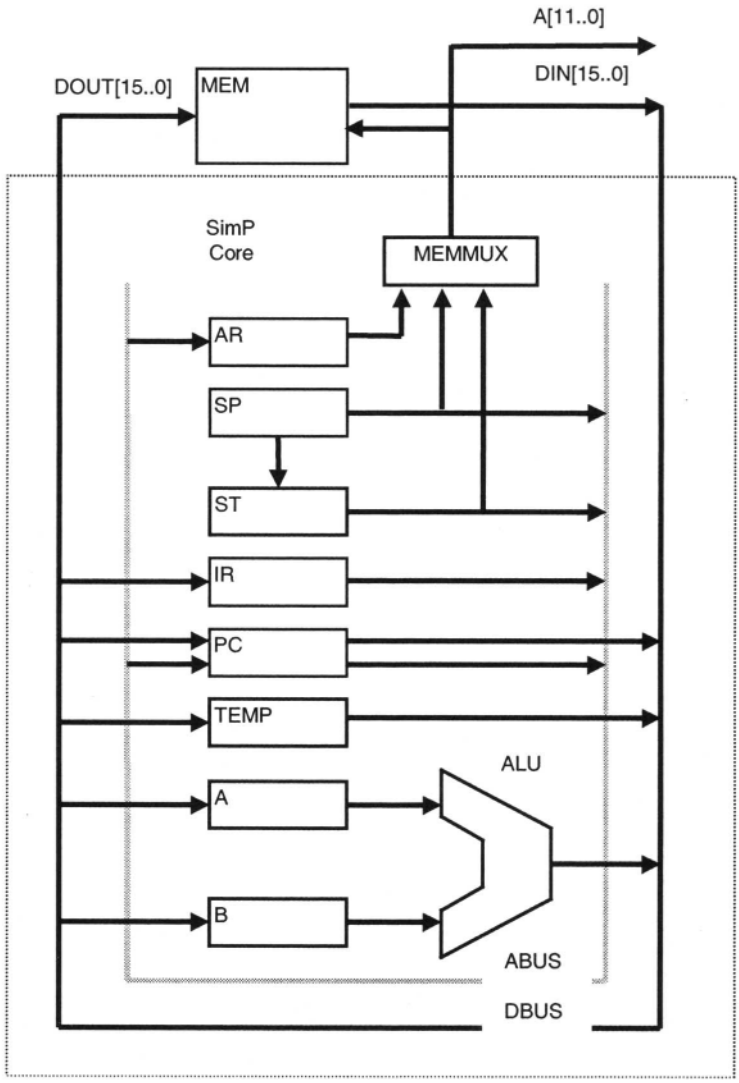



Figure 7.2 SimP Data Path

External memory can be both the source and destination in data transfers. This is determined by the memory control lines that specify either the memory read (MR) or memory write (MW) operation. Memory location (that takes place in data transfer) is specified by the value of the output of MEMMUX, which in turn specifies the effective address.

All register transfers are initiated and controlled by the control unit. It carries out the selection of the data source for each of internal bus, destination of data transfer, as well as operations local to individual resources. For example, the control unit activates the memory read or write control line, initializes an individual register, performs such operations as the incrementation or decrementation of the contents of a register, and selects the operation of the arithmetic-logic unit. All register transfers are synchronized by the system clock and take place at the next clock cycle.

## 7.3 Instruction Execution

The SimP's core instructions are executed as sequences of microoperations presented by register transfers. The basic instruction cycle contains all operations from the start to the end of an instruction. It is divided into three major steps that take place in four machine clock cycles denoted by T0, T1, T2, and T3.

1. Instruction fetch is when a new instruction is fetched from an external memory location pointed to by the program counter. It is performed in two machine cycles. The first cycle, T0, is used to transfer the address of the next instruction from the program counter to the address register. The second cycle T1 is used to actually read the instruction from the memory location into instruction register, IR. At the same time program counter is incremented by one to the value that usually represents the next instruction address.
2. Instruction decode is the recognition of the operation that has to be carried out and the preparation of effective memory address. This is done in the third machine cycle T2 of the instruction cycle.
3. Instruction execution is when the actual operation specified by the operation code is carried out. This is done in the fourth machine cycle T3 of instruction cycle.

Besides these three fundamental operations in each machine cycle, various auxiliary operations are also performed that enable each instruction to be executed in exactly four machine cycles. They also provide the consistency of contents of all processor registers at the beginning of each new instruction cycle.

Instructions are executed in the same sequence they are stored in memory, except for program flow change instructions. Besides this, the SimP provides a very basic single level interrupt facility that enables the change of the program flow based on the occurrence of external events represented by hardware interrupts. A hardware interrupt can occur at any moment since an external device controls it. However, the SimP checks for the hardware interrupt at the end of each instruction execution and,

in the case that the interrupt has been required, it sets an internal flip-flop called interrupt flip-flop (IFF). At the beginning of each instruction execution, SimP checks if IFF is set. If not set, the normal instruction execution takes place.

If the IFF is set, SimP enters an interrupt cycle in which the current contents of the program counter is saved on the stack and the execution is continued with the instruction specified by the contents of memory location called the interrupt vector (INTVEC).

The interrupt vector represents the address of the memory location which contains the first instruction of the Interrupt Service Routine (ISR), which then executes as any other program sequence. At the end of the ISR, the interrupted sequence, represented by the memory address saved on the stack at the moment of the interrupt acknowledgment, is returned to using the "ret" instruction.

The overall instruction execution and control flow of the control unit, including normal execution and interrupt cycle, is represented by the state flowchart of Figure 7.3. This flowchart is used as the basis for the state machine that defines the control unit operation.

Some other 1-bit registers (flip-flops) appear in the flowchart of Figure 7.3. First is the interrupt request flip-flop (IRQ). It is used to record the active transition on the interrupt request input line of the microprocessor. When the external device generates an interrupt request, the IRQ flip-flop will be set and, under the condition that interrupts are enabled, will cause the IFF flip-flop to be set. Consequently, the interrupt cycle will be initiated instead of normal instruction execution cycle. Control of the interrupt enable (IEN) flip-flop is carried out by programmer using instructions to enable or disable interrupts. Initially, all interrupts are enabled automatically. After recognition of an interrupt, further interrupts are disabled automatically. All other interrupt control is the responsibility of the programmer. During the interrupt cycle, the IRQ flip-flop will be cleared enabling new interrupt requests to be recorded. Also, interrupt acknowledgment information will be transferred to the interrupting device in the form of the pulse that lasts two clock cycles (IACK flip-flop is set in the machine cycle T1 and cleared in the cycle T3 of the interrupt cycle).

Now, we will describe the normal instruction execution cycle illustrated in the flowchart of Figure 7.4. In the first machine cycle, T0, the contents of the program counter is transferred to the address register. This register prepares the address of the memory location where the next program instruction is stored. The next machine cycle, T1, is first used to fetch and transfer an instruction to the instruction register to enable further decoding. In the same cycle, two other microoperations are performed.

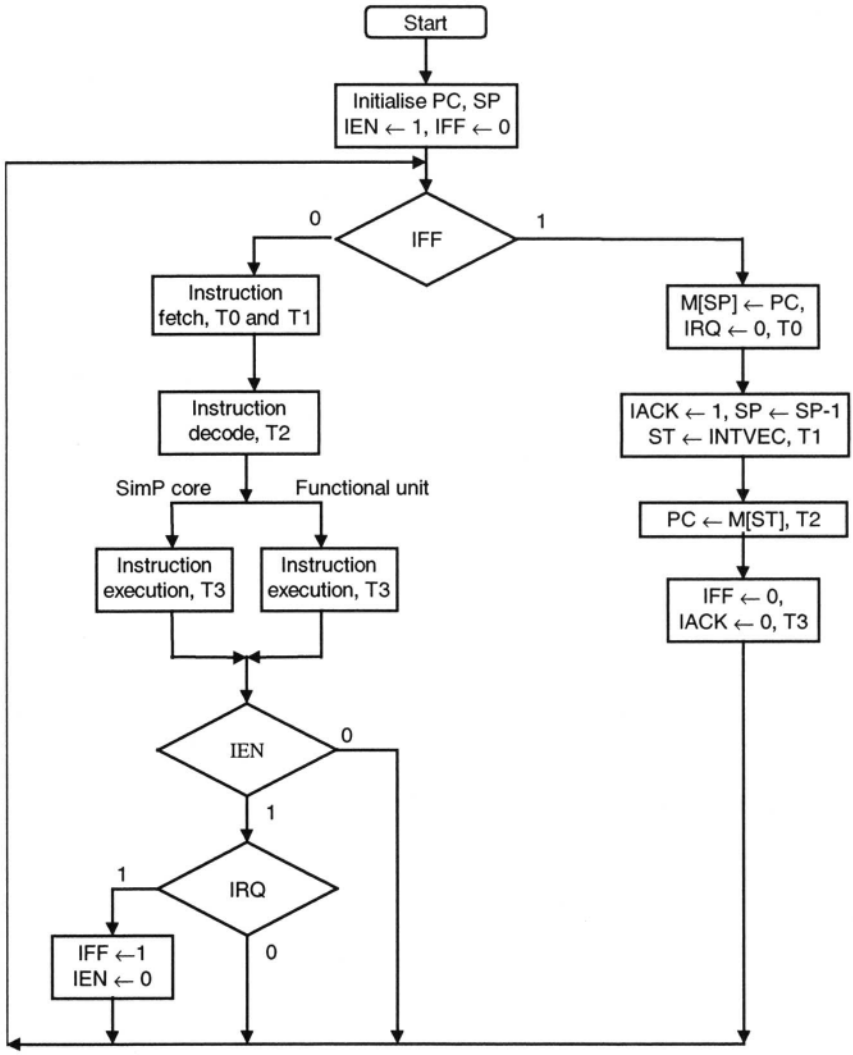
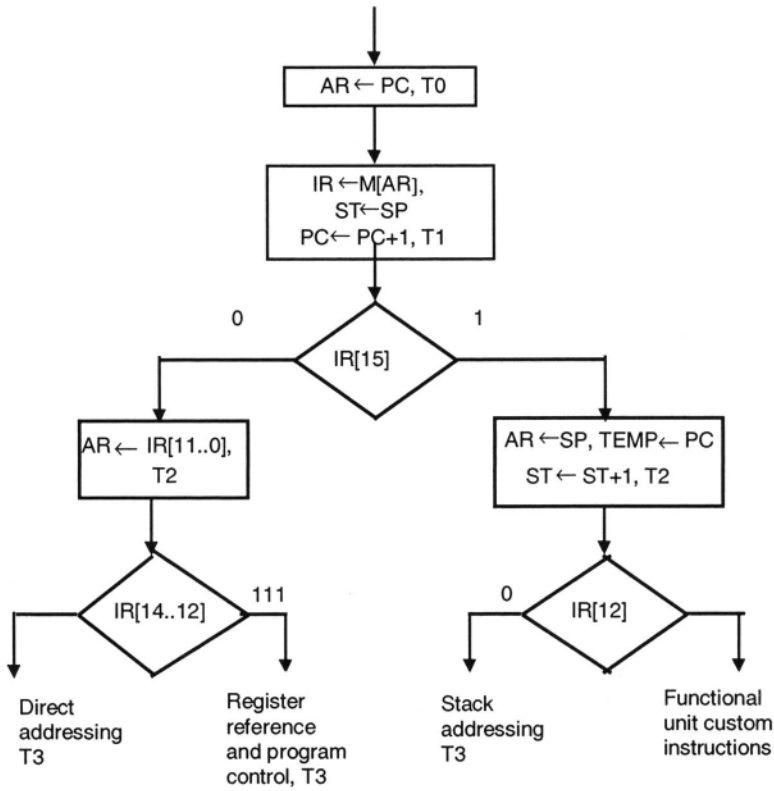



Figure 7.3 SimP's control flow diagram

The program counter is incremented to point to the next instruction which would be executed if there is no change in program flow. Also, the stack pointer (SP) is copied into the ST. This is preparation for the possibility that the instruction uses stack addressing mode in the next machine cycle.




7.4 Instruction execution flowchart

Register transfers that take place in the next machine cycle, T3, depend on the value of the most significant bit of the instruction fetched which is now bit IR[15]. If this value is equal to 0, direct or register addressing mode is used. If direct addressing mode is used, the lower 12 instruction bits, IR[11..0], represent the effective address which is used during the instruction execution step. Therefore, they are transferred to the address register preparing the effective address for the last machine cycle if needed. If IR[15] is equal to 1, two possibilities exist.

First, if the IR[12] bit is also 1, it is an instruction that executes a custom, application-specific instruction in a functional unit. Actions undertaken by the control unit for this case will be explained later. Otherwise, the instruction belongs to one using the stack addressing mode. To execute these instructions efficiently, preparation for all possible directions in which instruction execution can continue

are done. First, the stack pointer is copied into the address register preparing for instructions that will push data onto the stack ("push" and "jump to subroutine" instruction) during the execution step. Second, the program counter is copied into the TEMP register to prepare for instructions that must save the contents of the program counter onto the stack and change the value of the program counter ("jump to subroutine" instruction). Finally, the ST register is incremented to prepare for instructions that pull data from the stack ("pull" and "ret" instructions). These steps also enable the proper updating (incrementing or decrementing) of the SP register in the T3 machine cycle, while the stack is accessed using the AR or ST register as the source of the effective address.

The instruction execution step performed in the T3 machine cycle for all instructions from the SimP's core is presented in the Table 7.4.

Table 7.4 Executed step in the T3 cycle.

| Instruction | Microoperations (Register transfers) |
|---|---|
| LDA | A ← M[AR] |
| LDB | B ← M[AR] |
| STA | M[AR] ← A |
| STB | M[AR] ← B |
| JMP | PC ← IR[11..0] |
| JSR | M[AR] ← TEMP, PC ← IR[11..0], SP ← SP-1 |
| ADD | A ← A+B |
| AND | A ← A and B |
| CLA | A ← 0 |
| PSHA | M[AR] ← A, SP ← SP-1 |
| PULA | A ← M[ST], SP ← SP+1 |
| CLB | B ←0 |
| CMB | B ← B' |
| INCB | B ← B+1 |
| DECB | B ← B-1 |
| CLflag | flag ← 0, flag = C,Z |
| ION | IEN ← 1 |
| IOF | IEN ← 0 |
| SZ | If Z=0 (PC ← PC+1); else nothing |
| SC | If C=0 (PC ← PC+1); else nothing |
| RET | PC ← M[ST], SP ← SP+1 |

## 7.4 SimP Implementation

Our approach to the SimP design follows a traditional path of digital system design partitioned into the data path and control unit parts as illustrated in Figure 7.5. The data path consist of all registers, interconnect structures (including various multiplexers), and data processing resources. The data path enables register transfers under the control of multiplexer selection signals and control signals of the registers, local operations on the contents of the registers, data transformations in the arithmetic-logic unit, and data exchange with the outside world (memory and input/output devices). From an external point of view it provides a 12-bit address bus and a 16-bit data bus. The control unit provides proper timing, sequencing and synchronization of microoperations, and activation of control signals at various points in the data path (as required by the microoperations). It also provides control signals which are used to control external devices such as memory operations and the interrupt acknowledgment signal. The operation of the control unit is based on information provided by the program (instructions fetched from memory), results of previous operations, as well as the signals received from the outside world. In our case the only signal received from the outside world is the interrupt request received from the interrupting device.

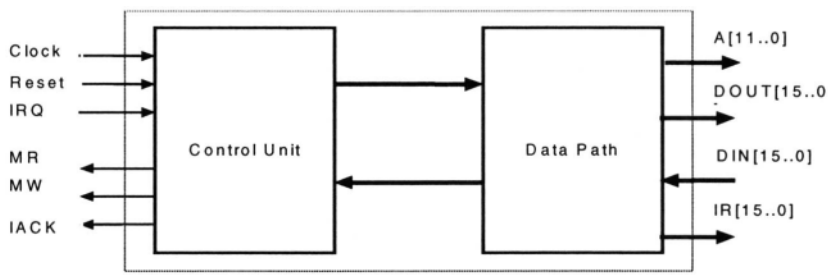



Figure 7.5 Basic partition of SimP design

### *7.4.1 Data Path Implementation*

In order to design all resources of the data path (Figure 7.1), we must first identify data inputs and data outputs of each resource, as well as operations that can be carried out and the control signals that initiate operations.

**Program Counter**

As an example, take the program counter (PC). Its data inputs, data outputs, and control signals are illustrated in Figure 7.6. By analyzing microoperations as well as resource usage, we see that the PC must provide 12-bit inputs from both internal address and data buses. These inputs are called PCA[11..0] and PCD[11..0] respectively. Consequently, the appropriate control signals, which determine the input lines, called LDA and LDD, are provided as well. The PC must provide control signals that enable its initialization at system start-up (power-up), clear (CLR) and incrementation (INC).

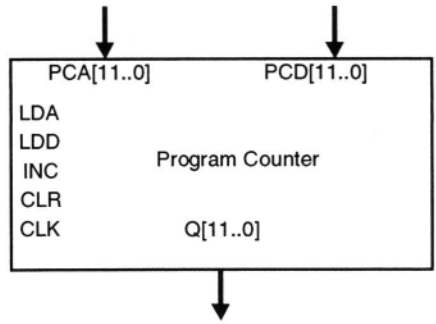



Figure 7.6 Program Counter

The AHDL design that describes the PC operation is given in Example 7.1.

Example 7.1 PC operation

```
TITLE "Program Counter pc";

SUBDESIGN pc
 (
    clk, clr,lda,ldd, inc  :INPUT;
    pcd[11..0]   :INPUT;
    pca [11..0]   :INPUT;
    q[11..0]   :OUTPUT;
)

VARIABLE
    ff[11..0]   :DFF;

BEGIN
    ff[].clk=clk;
    q[]=ff[].q;
    ff[].clrn=!clr;
```

```
IF ldd THEN
    ff[].d=pcd[];
ELSIF lda THEN
    ff[].d=pca[];
ELSIF inc THEN
    ff[].d=ff[].q+1;
ELSE
    ff[].d=ff[].q;
END IF;

END;
```

## Stack Pointer

Another example of a register is the stack pointer (SP). It can be initialized to a specific value (FF0 [hex]) at the system start-up. As specified by its microoperations, the SP can be only initialized, incremented, and decremented. Its data inputs, outputs, and control lines are illustrated in Figure 7.7. The AHDL design describing the SP operation is given in Example 7.2.

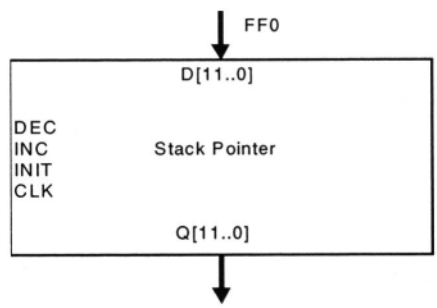



Figure 7.7 Stack Pointer

Example 7.2 SP Operation.

```
TITLE "Stack Pointer sp";

SUBDESIGN sp
 (
    clk, inc, dec, init  :INPUT;
    q[11..0]   :OUTPUT;
)
VARIABLE
    ff[11..0]   :DFF;
    d[11..0]   :NODE;
```

```
BEGIN
    ff[].clk=clk;
    q[]=ff[].q;
    d[]=H"FF0"; %initial value of stack pointer %

IF init THEN
    ff[].d=d[];
ELSIF dec THEN
    ff[].d=ff[].q-1;
ELSIF inc THEN
    ff[].d=ff[].q+1;
ELSE
    ff[].d=ff[].q;
END IF;

END;
```

### Working Registers

Working registers are used to store operands, results of operations, and carry out operations. The B register is slightly more complex and enables the microoperations of incrementing, decrementing, complementing of its contents, clearing, and loading contents from the input data lines. It is illustrated in Figure 7.8.

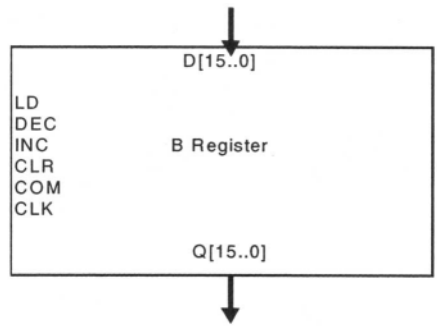



Figure 7.8 B Register

The operation of the B register is described in Example 7.3.

Example 7.3 Working register B

```
TITLE "Working Register b";

SUBDESIGN b
(
   clk, clr,ld,inc,dec,com  :INPUT;
   d[15..0]   :INPUT;
   q[15..0]   :OUTPUT;
)

VARIABLE
   ff[15..0]   :DFF;

BEGIN
   ff[].clk=clk;
   q[]=ff[].q;
   ff[].clrn=!clr;

IF ld THEN
   ff[].d = d[];
ELSIF inc THEN
   ff[].d=ff[].q+1;
ELSIF dec THEN
   ff[].d=ff[].q-1;
ELSIF com THEN
   ff[].d=!ff[].q;
ELSE
   ff[].d=ff[].q;
END IF;

END;
```

Other registers, including 1-bit registers that indicate the result of the most recent arithmetic-logic unit operation, are described by similar AHDL descriptions.

### Arithmetic-Logic Unit

The Arithmetic Logic Unit (ALU) is designed in a hierarchical manner by first designing 1-bit ALU as a basic cell. The basic cell is then iterated 16 times in a structural model to produce the 16-bit ALU. The 1-bit ALU is described in Example 7.4.

Example 7.4 1-bit ALU

```
TITLE "1-bit alu alu1";
```

```
SUBDESIGN alu1
 (
    a,b,cin,als[1..0]   :INPUT;
    q,cout   :OUTPUT;
)

BEGIN

CASE als[1..0]  IS
    WHEN B"00" =>
        q = a $ (b $ cin);
        cout=carry((a & b)#(a & cin)#(b & cin));
    WHEN B"01" =>
        q = a & b;
    WHEN B"10" =>
        q = a;
    WHEN B"11" =>
        q = b;
END CASE;

END;
```

As its inputs the 1-bit ALU has two operand data inputs `a` and `b` and an input carry bit (`cin`) from the previous stage of multi-bit ALU, as well as two lines to select the operation, `als [1..0]` . The output results are present on the data output line (`q`) output carry (`cout`) and are used as input in the next stage of the multi-bit ALU. Operations performed by the 1-bit ALU are 1-bit addition, 1-bit "logical and", and transfer of input argument `a` or `b`. Transfer operations are needed because neither of the working registers has direct access to the data bus, but is accomplished through the ALU. The 16-bit ALU is designed using "pure" structural AHDL description as shown in Example 7.5.

Example 7.5 16-Bit ALU.

```
TITLE "16-bit alu alu16";
INCLUDE "alul";

SUBDESIGN alu16
 (
    alusel[1..0], a[15..0], b[15..0], cin  :INPUT;
    q[15..0], cout, zout   :OUTPUT;
)

VARIABLE
    1alu[15..0]   :ALU1;
```

```
BEGIN
    1alu[].a=a[];
    1alu[].b=b[];
    1alu[].als[]=alusel[];
    1alu[0].cin=cin;
    1alu[1].cin=soft(1alu[0].cout);
    1alu[2].cin=soft(1alu[1].cout);
    1alu[3].cin=soft(1alu[2].cout);
    1alu[4].cin=soft(1alu[3].cout);
    1alu[5].cin=soft(1alu[4].cout);
    1alu[6].cin=soft(1alu[5].cout);
    1alu[7].cin=soft(1alu[6].cout);
    1alu[8].cin=soft(1alu[7].cout);
    1alu[9].cin=soft(1alu[8].cout);
    1alu[10].cin=soft(1alu[9].cout);
    1alu[11].cin=soft(1alu[10].cout);
    1alu[12].cin=soft(1alu[11].cout);
    1alu[13].cin=soft(1alu[12].cout);
    1alu[14].cin=soft(1alu[13].cout);
    1alu[15].cin=soft(1alu[14].cout);
    cout=1alu[15].cout;
    q[]=1alu[].q;

IF (q[15..8]==H"00") AND (q[7..0]==H"00") THEN
    zout=B "1";
ELSE
    zout=B"0";
END IF;

END;
```

We see from this design that the 1-bit ALU design file is included and instantiated as a component in a new design 16 times. Also, additional output signals are introduced to indicate output carry from the overall circuit and the value of the operation equals zero.

### Data and Address Multiplexers

Typical circuits used in the data path are data and address multiplexers. As an example, consider the data bus multiplexer (DBUSMUX) used to provide the data from the source which to appear on the data bus lines. There are four possible sources of data on the input lines of this multiplexer as illustrated in Figure 7.9. They are the PC register, the TEMP register, the ALU, and the main memory.

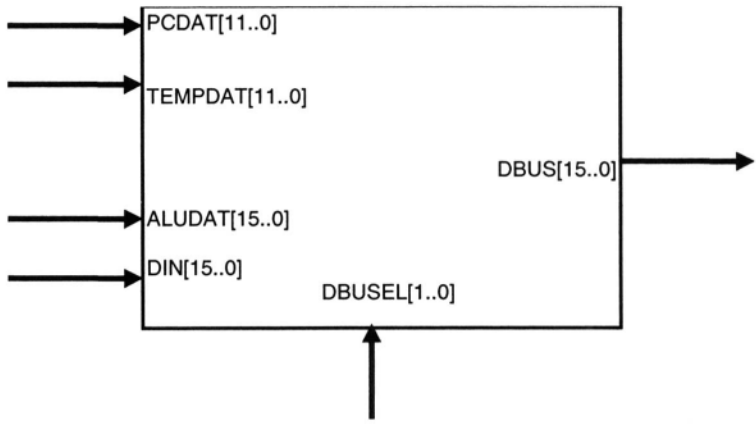



Figure 7.9 Data Bus Multiplexer

Two input lines DBUSEL[ 1..0] are used to select the source that is forwarded to the output of the multiplexer. Output lines represent the internal data bus lines. It should be noted that the PC and TEMP output lines are connected to the lower 12 bits of the data bus. If the contents of these registers is transferred to the data bus, the upper 4 bits will be grounded. This is shown in Example 7.6, which shows AHDL description of the data bus multiplexer. Other multiplexers used in the data path are designed in a similar way.

Example 7.6 Data Bus Multiplexer.

```
TITLE "Data Bus Multiplexer dbusmux";

SUBDESIGN dbusmux
 (
    dbusel[1..0]   :INPUT;
    pcdat[11..0]   :INPUT;
    tempdat[11..0]   :INPUT;
    aludat[15..0]   :INPUT;
    din[15..0]   :INPUT;
    out[15..0]   :OUTPUT;
)

VARIABLE
    pp[15..12]   :NODE;
```

```
BEGIN
    pp[15..12]=GND;

CASE dbusel[]  IS

    WHEN B"00" =>
        out[11. .0] = pcdat[];
        out[15..0]  = pp[15..12];

    WHEN B"01" =>
        out[11..0]  = tempdat[];
        out[15..12]  = pp[15..12];

    WHEN B"10" =>
        out [] = aludat [] ;

    WHEN B"11" =>
        out[]  = din[] ;
END CASE;

END;
```

## Data Path

The overall data path is integrated as the schematic (graphic) file just to visually describe connections of individual components designed using exclusively textual descriptions. It is described in a slightly simplified form in Figure 7.10. The dashed lines represent the control signals that are generated by the control unit to enable required register transfers or initiate local microoperations in registers. They also select source information which will be allowed to the bus.

The data path provides

- data input through external DIN[15..0] lines,
- data output through external DOUT[15..0] lines,
- addresses of memory locations or input/output registers through ADDRESS[11..0] lines,
- indications on the values of computation done in the arithmetic-logic unit through COUT (carry) and ZOUT (zero) lines, and
- current instruction operation code (placed in the instruction register IR) to the control unit to be decoded and executed.

All registers of the data path are connected to the system clock and change values with the clock. Clock inputs into the registers are not shown in Figure 7.10.

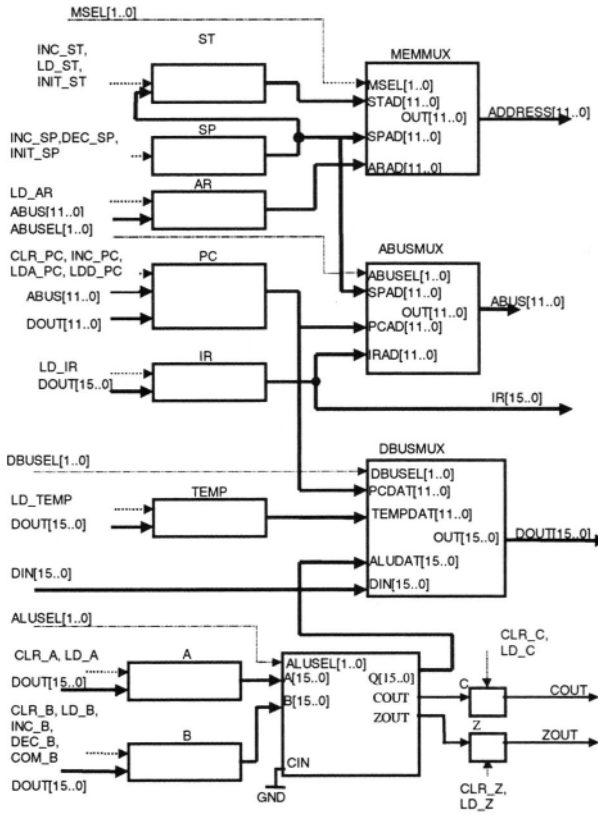



Figure 7.10 SimP's Data Path

### *7.4.2 Control Unit Implementation*

The control unit is the core of the SimP microprocessor. It provides proper timing and sequencing of all microoperations and perform the microoperations as required by the instructions of the user program stored in the external memory, which is external only from the point of view of microprocessor. A low capacity memory can be implemented within the same FPLD used to implement SimP core. The control unit provides proper start-up of the microprocessor upon power-up or manual reset. The control unit is also responsible for interrupt sequences as shown in preceding sections.

The global structure of the control unit is presented in Figure 7.11. It receives information from the data path both concerning the instructions that have to be executed and the results of ALU operations. It also accepts reset and interrupt request signals. Using these inputs it carries out the steps described in the control flowcharts of Figures 7.3 and 7.4.

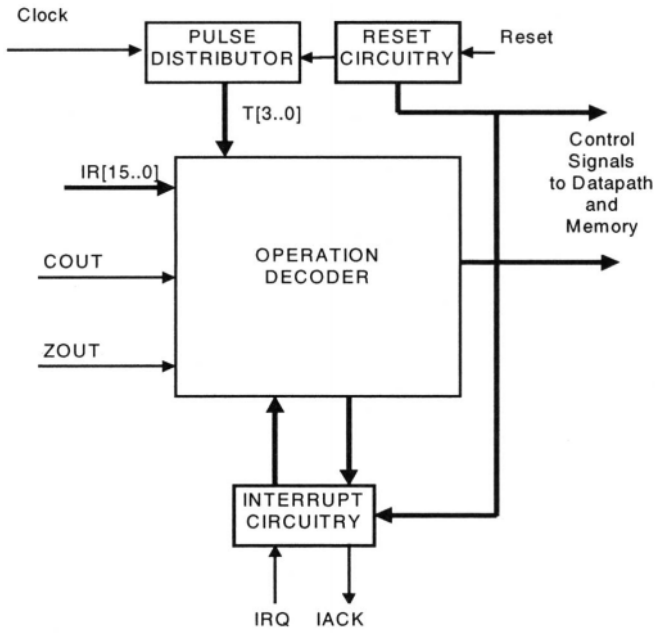



Figure 7.11 SimP's control unit

Obviously, in order to carry out proper steps in the appropriate machine (clock) cycles, a pulse distributor is needed to produce four periodic non-overlapping waveforms, which are used to synchronize individual microoperations. They are used in conjunction with the information decoded by the operation decoder to determine actions, register transfers, and microoperations undertaken by the data path. The only exceptions occur in two cases presented by external hardware signals:

- When the system is powered-up or reset manually, the operation of the pulse distributor is seized for four machine cycles. This time is needed to initialize data path resources (program counter and stack pointer), as well as interrupt control circuitry.

- When the interrupt request signal is activated and if the interrupt structure is enabled, the control unit provides interruption of the current program upon the completion of the current instruction and the jump to predefined starting address of an interrupt service routine. The control unit passes through the interrupt cycle steps presented with the right hand branch of the flowchart in Figure 7.3.

**Pulse Distributor**

The Pulse Distributor takes the system clock and provides four non-overlapping sequences called `T[3..0]`. The Pulse Distributor also has two input control lines as shown in Figure 7.12. The first, called "clear pulse distributor" (`CLR`), is used to bring the pulse distributor to its initial state T[3..0]=0001. It denotes that the `T0` machine cycle is present. The second, called "enable pulse distributor" (ENA) is used to enable operation of the pulse distributor.

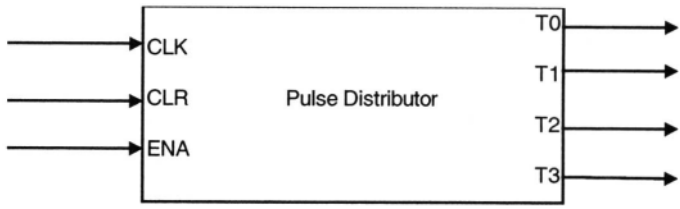



Figure 7.12 Pulse Distributor

The AHDL design file of the pulse distributor is given in Example 7.7.

Example 7.7 Pulse Distributor.

```
TITLE "Pulse Distributor pulsdist";

SUBDESIGN pulsdist
 (
    clk,clr,ena   :INPUT;
    t[3..0]   :OUTPUT;
)

VARIABLE
    ff[1..0]   :DFF;

BEGIN
    ff [].Clk=clk;
    ff [].clrn=!clr;
```

```
IF ena THEN
    ff[].d=ff[].q+1;
ELSE
    ff[].d=ff[].q;
END IF;

TABLE
    ff[1..0]    =>    t3,t2,t1,t0;

    B"00"       =>    0,0,0,1;
    B"01"       =>    0,0,1,0;
    B"10"       =>    0,1,0,0;
    B"11"       =>    1,0,0,0;
END TABLE;

END;
```

**Operation Decoder**

The Operation Decoder represents the combinational circuit that recognizes input signals to the control unit, as well as the current state of the control unit in order to provide the proper control signals.

Input and output ports of the operation decoder are illustrated in Figure 7.13. Input ports are shown on the left-hand side, and the control signals on the right-hand side of the block representing the operation decoder. The AHDL design of the operation decoder is presented in Example 7.8.

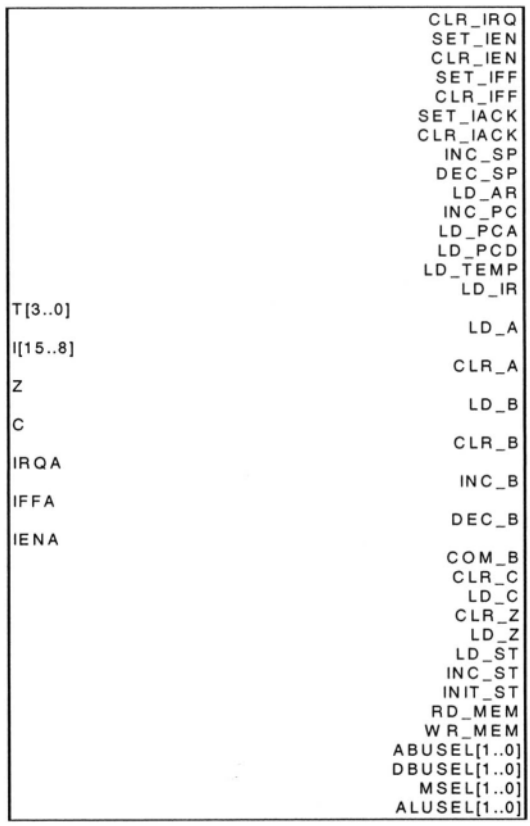



Figure 7.13 Operation Decoder

Example 7.8 The Operation Decoder.

```
TITLE "Operation Decoder opdecode";
SUBDESIGN opdecode
 (
t[3..0]        :INPUT;
i[15..8]       :INPUT;
z,c       :INPUT;
irqa         :INPUT;
iffa         :INPUT;
iena         :INPUT;
set_ien, clr_ien            :OUTPUT;
set_iff, clr_iff            :OUTPUT;
set_iack, clr_iack          :OUTPUT;
clr_irq       :OUTPUT;
inc_sp, dec_sp              :OUTPUT;
```

```
ld_ar          :OUTPUT;
inc_pc         :OUTPUT;
ld_pca, ld_pcd              :OUTPUT;
ld_temp        :OUTPUT;
ld_ir          :OUTPUT;
ld_a, clr_a                 :OUTPUT;
ld_b, clr_b, inc_b, dec_b, com_b :OUTPUT;
clr_c, clr_z, ld_c, ld_z    :OUTPUT;
ld_st, inc_st, init_st      :OUTPUT;
rd_mem         :OUTPUT;
wr_mem         :OUTPUT;
abusel[1..0]                :OUTPUT;% 1-sp,  2-pc,  3-ir %
dbusel[1..0]                :OUTPUT;%0-pc,1-temp,2-alu,3-mem %
msel[1..0]            :OUTPUT;% 0-sp,  1-ar 2-st %
alusel[1..0]                :OUTPUT;% 0-add,1-and,2-a,3-b%
)

BEGIN

IF t [0]  & !iffa THEN

       % normal instruction execution T0 %

    abusel[]= H"2";
    ld_ar=Vcc;
ELSIF t[0]  & iffa THEN

       % interrupt cycle T0 %

    dbusel[]=B"00";
    msel[]=H"0";
    wr_mem=Vcc;
    clr_irq=Vcc;
END IF;

IF t[1]  & !iffa THEN

       % normal instruction execution T1 %

    rd_mem=Vcc;
    msel[]=B"01";
    dbusel[]=B"11";
    ld_ir=Vcc;
    inc_pc=Vcc;
    ld_st=Vcc;
ELSIF t[1]  & iffa THEN

       % interrupt cycle T1 %
    set_iack=Vcc;
```

```
    dec_sp=Vcc;
    init_st=Vcc;
END IF;

IF t[2]  & !iffa THEN

       % decoding of instruction in T2%

    IF i [15]==B"1" THEN

    %stack or funct. block addressing mode%

     abusel[]=B"01"; % ar<-sp %
     ld_ar=Vcc;
     dbusel[]=B"00"; % temp<-pc %
     ld_temp=Vcc;
     inc_st=Vcc;
    ELSIF i[15]==B"0" THEN

    %direct addressing mode%

     abusel[]=B"11"; % ar from ir12 %
     ld_ar=Vcc;
     dbusel[]=B"00"; % temp from pc %
     ld_temp=Vcc;
    END IF;

ELSIF t[2]  & iffa THEN

       % interrupt cycle T2 %
     msel[]=H"2";
     dbusel []=B"11";
     rd_mem=Vcc;
     ld_pcd=Vcc;
END IF;

IF t[3]  & !iffa THEN

% instruction execution T3 %

    CASE i[15..12]  IS
    WHEN B"0000" =>
    %lda%
     ld_a=Vcc;
     rd_mem=Vcc;
     msel []=B"01";
     dbusel[]=B"11";

    WHEN B"0001" =>
```

```
%ldb%
 ld_b=Vcc;
 rd_mem=Vcc;
 msel[]=B"01";
 dbusel[]=B"11";

WHEN B"0010" =>
%sta%
 alusel[]=B"10";
 dbusel[]=B"10";  %from ALU%
 msel[]=B"01";
 wr_mem=Vcc;

when B"0011" =>
%stb%
 alusel[]=B"11";
 dbusel[]=B"10";
 msel[]=B"01";
 wr_mem=Vcc;

 WHEN B"0100" =>
%jmp%
 abusel[]=B"11";  %from ir%
 ld_pca=Vcc;

WHEN B"1000" =>
%jsr%
 dbusel[]=B"01";
 msel[]=H"1";
 wr_mem=Vcc;     % M[ar]<-temp %
 abusel[]=B"11";
 ld_pca=Vcc; % pc<-ir12 %
 dec_sp=Vcc;

WHEN B"1010" =>

 msel[]=H"1";
 wr_mem=Vcc;  % psha %
 dbusel[]=B"10";
 alusel[]=B"10"; %  M[ar]<-a %

WHEN B"1100" =>

 ld_a=Vcc;   % pula %
 msel[]=H"2";
 dbusel[]=B"11";
 rd_mem=Vcc;     % a<-M[st]  %
 inc_sp=Vcc;
WHEN B"1110" =>
```

```
 msel[]=H"2";
 dbusel[]=B"11"; %ret%
 rd_mem=Vcc;
 ld_pcd=Vcc; % pc<-M[st] %
 inc_sp=Vcc;
END CASE;


CASE i [15..8] IS

WHEN H"71" =>

 ld_a=Vcc;%add%
 dbusel[]=B"10"; %ALU select op%
 alusel[]=B"00";
 ld_c=Vcc;
 ld_z=Vcc;

WHEN H"72" =>
 ld_a=Vcc;%aand%
 dbusel[]=B"10"; %ALU select op%
 alusel[]=B"01";

WHEN H"73" =>
 clr_a=Vcc;      %cla%

WHEN H"74" =>
 clr_b=Vcc;      %clb%

WHEN H"75" =>
 com_b=Vcc;      %cmb%

WHEN H"76" =>
 inc_b=Vcc;      %incb%

WHEN H"77" =>
 dec_b=Vcc;      %decb%

WHEN H"78" =>
 clr_c=Vcc;      %clc%

WHEN H"79" =>
 clr_z=Vcc;      %clz%

WHEN H"7A" =>
 set_ien=Vcc;

WHEN H"7B" =>
 clr_ien=Vcc;
```

```
    WHEN H"7C" =>
    IF c==B"1" THEN
     inc_pc=Vcc;     % sc %
    ELSIF c==B"0" THEN
     abusel[]=B"11";
     ld_ar=Vcc;
    END IF;

    WHEN H"7D" =>

    IF z==B"1" THEN
     inc_pc=Vcc;
    ELSIF z==B"0" THEN
     abusel[]=B"11";
     ld_ar=Vcc;
    END IF;

    END CASE;
    IF iena & irqa THEN
     set_iff=Vcc;
     clr_ien=Vcc;
    END IF;


ELSIF t[3]  & iffa THEN

       % interrupt cycle T3 %

    clr_iack=Vcc;
    clr_iff=Vcc;
END IF;

END;
```

**Reset Circuitry**

Reset circuitry initializes the SimP at power-up or manual external reset. The only input is a reset signal, but several outputs are activated for proper initialization. The initialization consists of providing the initial values for the program counter and stack pointer, enabling the interrupt enable flip-flop (IEN), and clearing internal IFF flip-flops. Upon initialization, external interrupts are enabled and the control unit automatically enters the instruction execution cycle. This happens as soon as the pulse distributor is enabled and initialized. Reset circuitry is represented with its input and output ports in Figure 7.14.

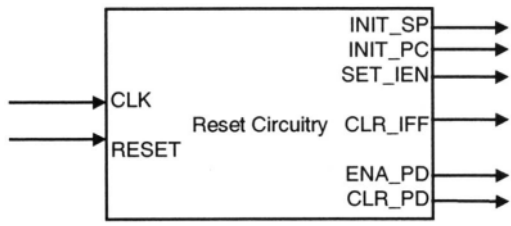



Figure 7.14 Reset Circuitry

Initialization lasts exactly four system clock cycles. In the case of an active RESET signal, an internal SR flip-flop is set. The output of the SR flip-flop represents the enable signal of an internal counter providing the internal counter will count until it reaches value 11. While this counter is counting, the initialization process is repeated in each machine cycle. When it stops, initialization is also stopped and the pulse distributor is enabled so it continues with its normal instruction execution cycle. The AHDL design file representing reset circuitry is given in Example 7.9.

Example 7.9 Initialization Circuitry.

```
TITLE "Initialization circuitry reset1";

SUBDESIGN reset1
 (
    clk, reset  :INPUT;
    init_sp,clr_pc,set_ien,clr_iff,ena_pd,clr_pd  :OUTPUT;
)

VARIABLE
    cnt[1..0]   :DFFE;
    cnt_out[1..0]      :NODE;
    rs   :SRFF;
    ena_cnt   :NODE;

BEGIN
    cnt_out[] =cnt[] .q;
    rs.clk=clk;
IF reset THEN
    rs.s=Vcc;
    rs.r=GND;
    ena_pd=GND;
    clr_pd=Vcc;
ELSIF !reset OR cnt_out[]==B"11" THEN
    rs.r=Vcc;
```

```
    rs.s=GND;
    ena_pd=Vcc;
    clr_pd=GND;
END IF;

    cnt [].clk=clk;
    ena_cnt=rs.q;

IF ena_cnt THEN
    cnt [] .d = cnt [] .q + 1;
ELSE
    cnt [] .d = cnt [] .q;
END IF;

IF ena_cnt THEN
    init_sp=Vcc;
    clr_pc=Vcc;
    set_ien=Vcc;
    clr_iff=Vcc;
ELSIF !ena_cnt THEN
    init_sp=GND;
    clr_pc=GND;
    set_ien=GND;
    clr_iff=GND;
END IF;

END;
```

### Interrupt Circuitry

The interrupt circuitry has only one external input, the interrupt request (IRQ), and one external output, the interrupt acknowledgment (IACK). Upon interrupt request assertion (IRQA), an IRQ flip-flop is set producing the IRQA signal which is used by the operation decode circuitry. If the interrupts are enabled, and IRQA set, the operation decoder will set the interrupt flip-flop (IFF) to force the control unit to enter the interrupt cycle. In the interrupt cycle, the IACK flip-flop, whose output is available to circuitry outside the SimP, is set for two machine cycles. The interrupt enable flip-flop (IEN) can be set by the operation decoder or reset circuitry and the interrupt flip-flop can be cleared by the operation decoder or reset circuitry. The AHDL file describing the operation of the interrupt circuitry is given in Example 7.10

Example 7.10 Interrupt Circuitry.

```
TITLE "Interrupt circuitry interrupt";
```

```
SUBDESIGN interrupt
(
   set_ien,set_ien1,clr_ien,set_iff,clr_iff,clr_iff1 :INPUT;
   set_iack, clr_iack, irq, clk  :INPUT;
   iffa, irqa, iena, iack  :OUTPUT;
)

VARIABLE
   irqff, ienff, iackff, iff  :DFF;
   clr_irq  :NODE;

BEGIN
   clr_irq=iackff.q;
   irqff.clk=clk;
   iackff.clk=clk;
   ienff.clk=clk;
   iff.clk=clk;

IF set_ien # set_ien1 THEN
   ienff.d=Vcc;
ELSIF clr_ien THEN
   ienff.d=GND;
ELSE
   ienff.d=ienff.q;
END IF;

IF set_iff  THEN
   iff.d=Vcc;
ELSIF clr_iff # clr_iff1 THEN
   iff.d=GND;
ELSE
   iff.d=iff.q;
END IF;

IF set_iack THEN
   iackff.d=Vcc;
ELSIF clr_iack THEN
   iackff.d=GND;
ELSE
   iackff.d=iackff.q;
END IF;

IF irq THEN
   irqff.d=Vcc;
ELSIF clr_irq THEN
   irqff.d=GND;
ELSE
   irqff.d=irqff.q;
END IF;
```

```
    iack=iackff.q;
    irqa=irqff.q;
    iena=ienff.q;
    iffa=iff.q;

END;
```

### Control Unit

The overall control unit circuit is represented by the schematic diagram in Figure 7.15. Its AHDL description will be just one using either instantiation or in-line references of the circuits shown in preceding examples.

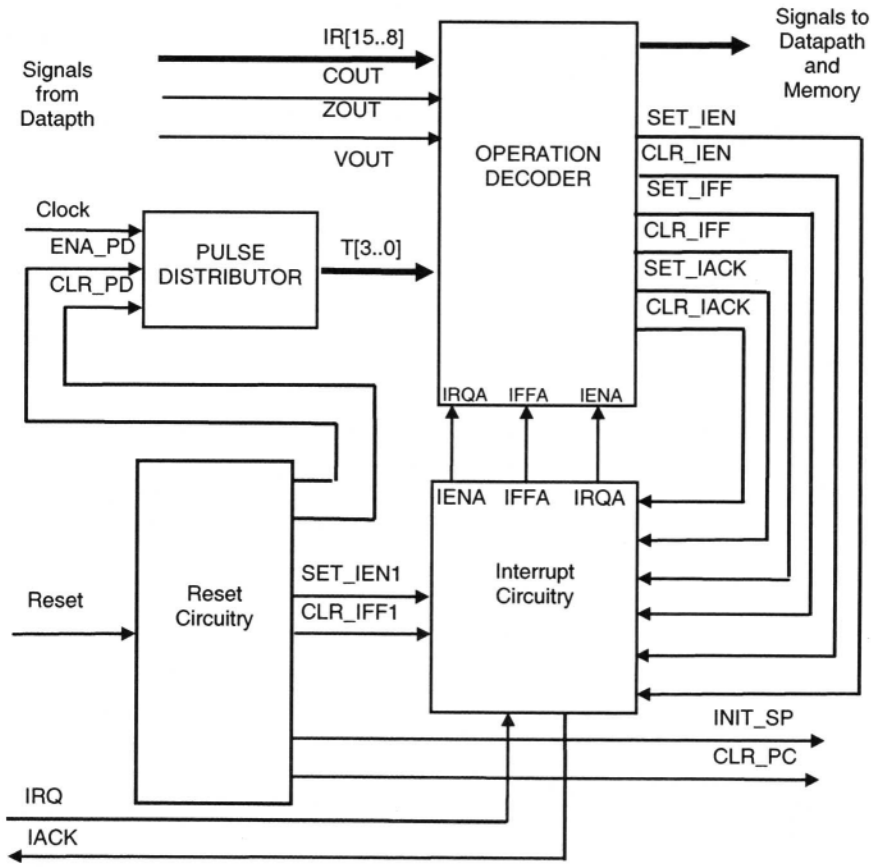



Figure 7.15 Control unit

### *7.4.3 Synthesis Results*

The SimP design, as presented in this chapter is accomplished in a hierarchical manner. No attempt at design optimization has been made. After designing components of the bottom level, the next level of hierarchy is designed. The final integration of the data path and control unit was done using schematic entry because it is represented by the pure structural design. Data summarizing the final compilation process are shown in Table 7.5.

Table 7.5 Compilation times

| Compilation Times | Time (sec) |
|---|---|
| Compiler Nettles Extractor | 00:00:07 |
| Database Builder | 00:00:43 |
| Logic Synthesizer | 00:00:38 |
| Partitioner | 00:00:25 |
| Fitter | 00:00:56 |
| Timing SNF Extractor | 00:00:41 |
| Assembler | 00:00:04 |
| **Total Time** | 00:03:34 |

The Max+Plus II compiler was allowed to perform all resource assignments automatically. The device used to fit the design was selected automatically from the FLEX 8000 series. The compiler found that the device suitable for SimP implementation is the EPF8820 with the basic features given in Table 7.6.

Table 7.6 SimP synthesis features

| | |
|---|---|
| Input pins | 19 |
| Output pins | 33 |
| Logic Cells required | 541 |
| Utilization of logic cells | 541/672 = 80% |
| Total flip-flops required | 117 |
| Total dedicated pins used | 4 |

It should be noticed that the data bus in the compiled design is separated into input and output data lines.

Extensive timing simulation was performed on each hierarchical level of design. It showed compliance with the system requirements. The maximum clock frequency achieved at the simulation level reached 16.6 MHz or a basic clock period (machine cycle) of 60 ns.

The SimP's design gives various opportunities for performance improvements that are left to the readers as exercises in using the design tools and FPLD technology. Some of them are mentioned in the following section. Also, Chapter 10 contains a customized and enhanced version of SimP called SimP-2, which is fully described using VHDL.

## 7.5 Questions and Problems

7.1 Complete SimP's design by:

- representing it using hierarchical description of constituent part to the level that can be described by AHDL descriptions
- designing all remaining parts (registers, multiplexers, flags and flip-flops)
- integrating all parts of data path first and then control unit with the data path
- synthesize SimP using FLEX10K20 FPLD as the target device

7.2 Design a small memory with a capacity of 256x8 and add it to the SimP. Memory should occupy lower 256 addresses. Change accordingly interrupt vector and place it in the highest memory location.

7.3 Prepare a simulation test that tests all features of the above design as complete as possible. Simulate first reset mechanism, and then simulate execution of a small program that uses all instructions. Special attention should be paid to instructions that control program flow (JMP, Sflag, RET). Simulate interrupt requests by preparing a small interrupt service routine. All programs should be stored into memory as specified using memory initialization file (.mif).

7.4 Make a more general description of the assignment of SimP operation codes to the symbols (mnemonics) by isolating them in the beginning of TDF description using CONSTANT statements. Modify the control unit description accordingly. In this way you can easily change operation codes without interfering into the description of control unit.

7.5 Describe the ALU using instantiation of the 1-bit ALU and FOR-GENERATE statement.

7.6 Extend the ALU with the following additional operations:

- subtraction (assuming that the numbers are presented in signed-magnitude format) represented by register transfer A ← A – B and
- logical OR represented by register transfer A ← A or B (bit-wise OR)
- logical XOR represented by register transfer A ← A xor B (bit-wise XOR)

7.7 Extend the SimP instruction set with instructions for arithmetic and logical shift for 1 bit left and right of the content of register A

7.8 Extend SimP's arithmetic unit assuming that the numbers are interpreted as being in two's complement format. Add indication for overflow (V flag). Extend data path accordingly. Add the SV (skip if overflow set) instruction.

7.9 How would you implement loading constants into working registers A and B using existing resources including memory module from 7.2? Analyze how an immediate addressing mode can be added. What type of extensions to the SimP data path have to be made to enable loading of the A and B registers with 16-bit constants? Can it be done with single instructions?

7.10 In order to increase address space to 64K locations, additional address generation mechanism is required. Consider two cases:

- by introducing page register that concatenated with the direct address expands address space. Content of the page register represents the page address and direct address within instruction represents the address of location within the current page.
- by introducing a register indirect addressing mode. For that purpose consider either transformation of one of the working registers to allow to be treated as an address register, or introducing a separate address register.

Perform modifications of the SimP core accordingly.

7.11 Introduce a few instructions that use the addressing modes from Problem 7.10, and accordingly change SimP control unit.

7.12 Add a 16-bit parallel input/output port to SimP that enables communication with input/output devices. The port should be programmable so that both most significant and least significant bytes can be programmed as either inputs or outputs using a data direction register. The port should be placed at any location of the 8 addresses at the top of the base 4K address space except the topmost 8 topmost addresses and accessible by load/store instructions. For the implementation of the port you may consider two cases:

- use external pins on a FLEX 10K device that allow bidirectional input/output
- use only internal resources as the port will be used to connect SimP core with other internally implemented logic

SimP's is described by the following function prototype:

```
FUNCTION SimP(clk, reset, irq, DATAIN[15..0])
    RETURNS (DATAOUT[15..0] , ADDRESS[11..0], m_read,
        m_write, iack);
```

7.13 Modify SimP instruction execution cycle so that after external reset SimP starts with the execution of an instruction stored at location specified by the content of

- the topmost location in the base 4K address space (location H"FFF") or
- the topmost location of physically present memory (if it is implemented in a FLEX 10K device EAB).

7.14 Add the circuit and modify SimP instruction set as needed to enable generation of the PWM (pulse-width modulated) signal. The circuit uses a 16-bit register to implement frequency divider and the duty cycle can in the range allowed by the size of the register. The PWM generator is controlled by contents of two registers: frequency division and duty cycle register.

7.15 Extend SimP interrupt handling circuitry to a 4-level interrupt structure with four different priorities and four different interrupt vectors stored in locations H"FFB", H"FFC", H"FFD" and H"FFE"). Modify the control unit as required to support this feature.

# 8 RAPID PROTOTYPING USING FPLDS - VUMAN CASE STUDY

Rapid prototyping systems composed of programmable components show great potential for full implementation of microelectronics designs. Prototyping systems based on field programmable devices present many technical challenges affecting system utilization and performance.

This chapter addresses two key issues to assess and exploit today's rapid-prototyping methodologies. The first issue is the development of architectural organizations to integrate field-programmable logic with an embedded microprocessor (Intel 386 EX) as well as system integration issues. The second is the design of prototyping systems as Custom Computing Engines.

Prototyping systems can potentially be extended to general custom computing machines in which the architecture of the computer evolves over time, changing to fit the needs of each application it executes. In particular, we will focus on implementing Private Eye display control, PSRAM control, and some secondary logic (PCMCIA control) using FPLD.

## 8.1 System Overview

The VuMan family of wearable computers, developed at the Carnegie Mellon University Engineering Design Research Center, will be used as the platform. One of the products in the line, VuMan 3, mixes off-the-shelf hardware components with software developed in-house to form an embedded system used by the US Marines for military vehicle maintenance. VuMan 3 specializes in maintenance applications for environments requiring nigged, portable electronic tools.

The components necessary to accommodate such needs include processing core, memory, BIOS/Bootcode ROM, display adapter, direct-memory-access (DMA) controller, serial ports, real-time clock, power control, input system, and peripheral controller. A functional diagram depicting the VuMan architecture is shown in Figure 8.1.

An Intel i386EX embedded microprocessor acts as the system core. This 3.3 volt version of the 386 carries several critical system components on-chip, such as DMA controller, interrupt controller, timers, serial ports, chip-select unit, and refresh unit. This makes it an ideal solution for the VuMan 3 embedded system since exploiting

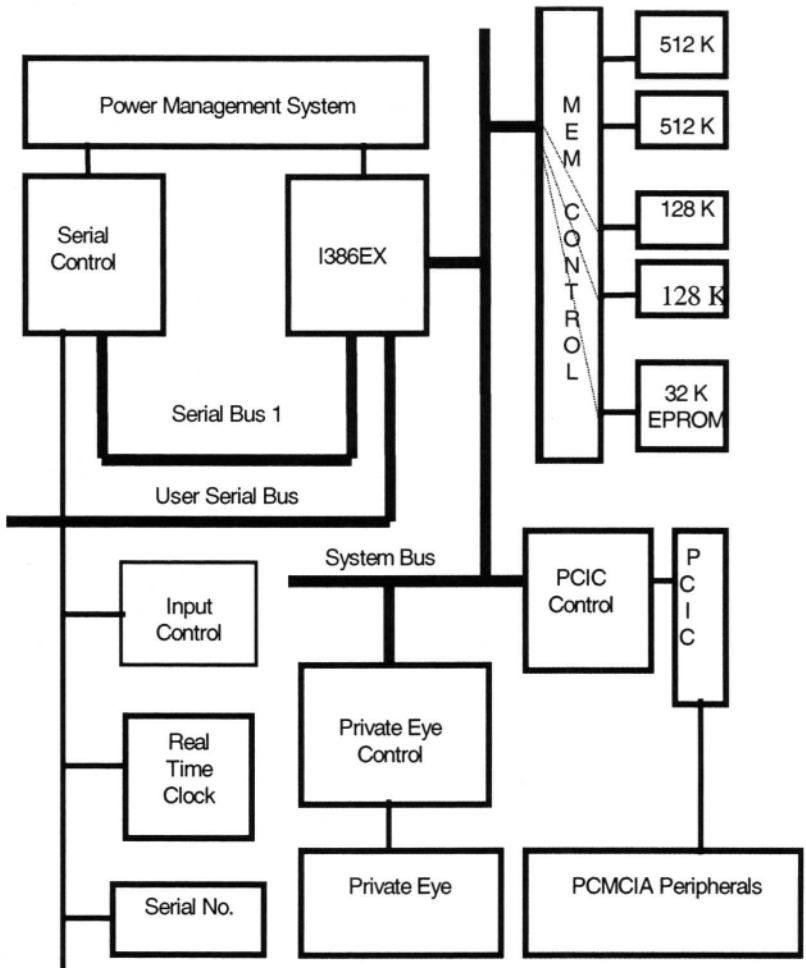



Figure 8.1 VuMan Functional Diagram

the on-chip services helps reduce chip count and board area. Additionally, the processor provides several signals allowing seamless connections to memories and

I/O devices. This feature further reduces the chip-count by eliminating the need for CPU-to-memory interface logic.

The memory subsystem attached to this processor consists of two components. Two Hitachi 3.3 volt 512K P-SRAMs (pseudo-static RAMs) provide a 16-bit path to 1MB of main memory (2 chips, 8 bits each = 16 bits). One chip stores all data on even addresses and the other maintains all odd address bytes. Likewise, two Hitachi 3.3 volt 128K SRAMs offer a 16-bit interface to 256K of RAM. Moreover, these memories obtain power from a battery, not the system power source, so they can be used to store vital data. They require no special interface and can attach gluelessly to the CPU core. Also, these memories use static cells eliminating the need for refresh logic.

The 512K chips, on the other hand, require periodic refreshing since they are not purely static RAMs. The 386EX's refresh control until assists in performing this function. Also, these memories require a pre-charge between accesses, which eliminates the possibility of using the direct CPU-to-memory interface offered by the i386EX. Therefore, this subsystem requires a control system to act as an interface between the system bus and the memories.

An additional non-volatile memory element, the 3.3 volt 32K EPROM, contains the code necessary to boot the system and initialize the hardware. The processor begins executing from the EPROM on power-up and the code stored therein must configure the system as desired. The bootcode sets up the system memory map as shown in Figure 8.2, performs testing of critical components, and then transfers control to the user application. The i386EX bus interface accommodates the EPROM seamlessly; hence, the ROM attaches directly to the system bus.

A Reflection Technology Private Eye provides the system with a 720x280-pixel display. This device allows the system to serially deliver pixel data to be drawn. The Private Eye, however, uses 5V signals so care must be taken when interfacing this device to the 3.3V system bus. The serial protocol used by the Private Eye requires the development of an interface to manage the communications between the processor and the display adapter.

The input system consists of a dial and buttons. When the user presses any of the buttons or rotates the dial, a code is sent to the processor by way of the i386EX's synchronous serial port, and the processor reacts accordingly. Likewise, when the CPU wishes to read the real-time clock reading or the silicon serial number, these values flow to the 386EX through the serial port. A PIC microcontroller manages the serial protocol between the processing core and these devices. This PIC also manages the power supplies and informs the processor when the power reaches dangerously low levels.

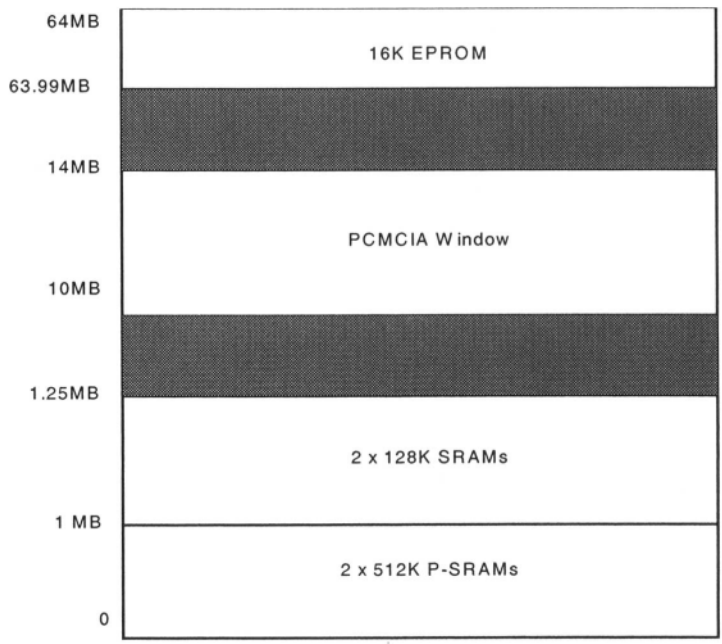



Figure 8.2 VuMan 3 Memory Map

Lastly, the system uses the Intel 8256 PCIC (PCMCIA controller) chip to manage the two PCMCIA slots in the system. These slots allow for connections of memory cards or peripheral cards. The PCIC must be programmed to map the PCMCIA cards into a certain memory region. Then, when the PCIC detects accesses to this region, it forwards the requests to the appropriate slot. Since the PCIC uses 5-volt signals, an interface must exist to manage the communications between the 5-volt PCIC and the 3.3-volt system bus. Also, as the PCIC expects ISA-compatible signals, the interface must convert the 386EX bus signals into semantically identical ISA signals.

## 8.2 Memory Interface Logic

Though the 386EX was designed to allow effortless connection of memory devices to the system bus, this feature cannot be exploited in the VuMan3 design due to the P-SRAMs used. These RAMs require a very strict protocol when performing reads or writes (see Hitachi HM65V8512 datasheet for details): (a) the chips must be turned off (chip-enable deasserted) for 80 ns between consecutive accesses, (b)

during reads, the chip-enable must be on for 15 ns before output-enable can be asserted to deliver the data, and (c) during writes, the write-enable signal must be pulsed for 35 ns while the chip-enable is on. Also, the chip-enable must remain active for 150 ns during each read/write cycle. Additionally, since these chips lack purely static cells, they require periodic refresh. A refresh cycle consists of pulsing the output-enable (OE) signal of the memory while keeping the chip-enable (CE) off. The following are timing requirements for the refresh cycle: a) CE must be off for 80 ns before OE can be asserted, b) OE must be off for at least 40 ns before it gets pulsed, and c) OE must be active for 80 ns during the pulse. To accommodate these requirements, the memory controller of Figure 8.3 was designed.

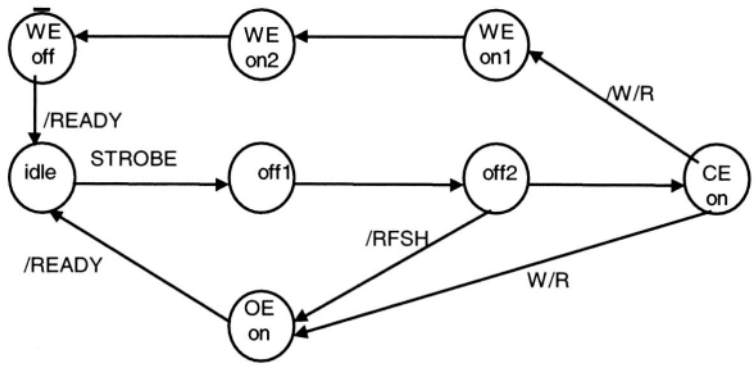



Figure 8.3 P-SRAM Memory Controller

The P-SRAM controller has seven input pins and five output pins defined in Table 8.1 ("/" denotes an active low signal).

The state machine from Figure 8.3 maintains synchronization with the system bus through the 32MHz processor clock (CLK2). Hence, a transition from one state to another state occurs every 1/32MHz = 31.25ns.

Initially, the state machine is idle and the memory chip-select signal remains off. When the processor issues a bus cycle, it first sets the appropriate chip-select and address lines and asserts the /ADS signal . The 512K P-SRAMs tie to the chip-select signal /CS1. Hence, if the i386EX turns on /CS1, it intends to communicate with the 512K RAMs. When the processor needs the low byte of the bus, it asserts the /BLE line and when it needs the high byte, it turns on /BHE. Similarly, for a 16-bit access both /BLE and /BHE are asserted. Hence, when /ADS asserts, if either /BLE or /BHE are asserted and /CS1 is on, the current bus cycle involves the P-SRAM memories; hence, the state machine activates at this point.

Table 8.1 P-SRAM Controller Inputs and Outputs.

| INPUTS | Meaning |
|---|---|
| */ADS* (ADdress Stable) | asserted by processor when a new bus cycle begins |
| */RFSH* | asserted by processor when the cycle is a PSRAM refresh cycle |
| */CS1* (Chip-Select 1) | asserted by processor when it wishes to communicate with the PSRAMs |
| */BLE* (Byte Low Enable) | asserted by processor if it intends to read/write the low-byte (data bits 0-7) of the addressed word |
| */BHE* (Byte High Enable) | asserted by processor if it intends to read/write the high-byte (data bits 8-15) of the addressed word |
| *W/R* | high during write cycle, low during read cycle |
| */READY* | asserted by 386EX's chip select unit when the bus cycle terminates (after X wait states have expired, where X is programmed by the bootcode) |
| *STROBE* | this signal is formed from the input signals as */ADS*[/RFSH+/CS1*(/BLE+?BHE)]* |
| **OUTPUTS** | **Meaning** |
| */PLB* | *(PSRAM_CE_LOWBYTE)*turns on in state CEon if /BLE is asserted and remains on until state idle |
| */PHB* | *(PSRAM_CE_HIGHBYTE)* turns on in state CEon if /BHE is asserted and remains on until state idle |
| */PWE* | *PSRAM_WE* is on during states WEon1 and WEon2 |
| */POE* | *PSRAM_OE* is on during state Oeon |
| */PCE* | *PSRAM_CE* turns on when off2 state finishes and remains on until state idle. |

When this bus cycle begins, the memory controller transitions to the OFF1 state. Since the chip-select is off during the idle state, the memory is guaranteed to have been off for at least 31.25 ns by the time the state machine enters the OFF1 state. At this stage, the chip-select is kept off and a transition is made to the OFF2 state with

the coming of the next clock. This increased the guaranteed memory deselected time to 31.25 ns + 31.25 ns = 62.50 ns. In the OFF2 state, the state machine transitions based on the current bus cycle type: read, write, or refresh.

Recall that a refresh cycle requires that the OE be pulsed while the CE is off. Therefore, if the current cycle is a refresh cycle, the state machine transitions to the OEon state, in which the OE signal turns on. Therefore, by the time the machine enters the OEon state, CE and OE have been off for 93.75ns, which satisfies the refresh timings. A transition back to the idle state happens as soon as the READY signal is asserted, signifying the end of the bus cycle. To meet the timing requirement, the OE pulse must last 80 ns, so the state machine needs to remain in the OEon state for at least 80 ns. Hence, the machine can return to the idle state after 173.75 ns have elapsed from the beginning of the bus cycle. Normally, bus accesses require 2 bus clock periods. During the first bus period (known as T1 [refer to the 386EX manual, ch.7: Bus Interface Unit]), the address and status signals are set by the processor. During the second period (known as T2) the device responds. If a peripheral needs more bus periods, it requests a wait states, each of which lasts for one bus period. The 386EX bus uses a 16 MHz clock, yielding a 1/16 MHz (62.50 ns) bus period. Hence, a normal cycle requires 2 bus periods (125 ns). Since the refresh requires 173.75 ns, it needs an additional 48.75 ns, or 1 wait state.

Read/write cycles proceed similarly. If the current bus cycle is not a refresh cycle, the machine transitions to state CEon. By the time the machine arrives at this state, the memory has been de-selected for 31.25 + 31.25 + 31.25 = 93.75 ns, which meets the pre-charging requirement of 80 ns. When the state machine enters this state, it turns on the chip-select unit for the appropriate memory, as determined by the /BLE and /BHE signals: /PLB is asserted if /BLE is on, /PHB is asserted If /BHE is on, and both are asserted if both /BLE and /BHE are on. Next, the state machine determines whether the cycle is a read or a write and it transitions accordingly.

During a read cycle, the machine enters the OEon state. In this state, the /POE is asserted and remains on until the cycle terminates, indicated by the processor asserting /READY. Hence, the CE is on for 31.25 ns before the OE turns on, satisfying the timing requirements. Additionally, CE must be on for 150ns to meet the access time requirement, so the state machine cannot return to the idle state until (when CE goes on) 243.75 ns have elapsed from the beginning of the cycle. Hence, for read/write, the memory needs 118.75 ns, or 2 wait states. The read bus cycle is depicted in the timing diagram of Figure 8.4.

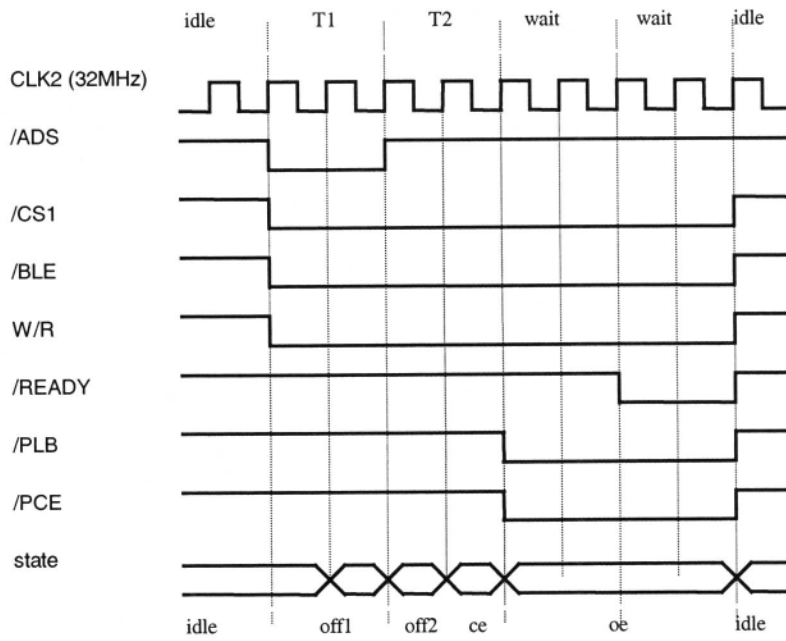



Figure 8.4 Read Cycle

Similarly, during a write cycle, the state machine proceeds from the CEon state to the WEon1 state. In this state, the /PWE signal is asserted, which starts the write to the memory. The machine transitions to the WEon2 on the next CLK2 edge and keeps /PWE active. Then, a transition to the WEoff state is made, and the /PWE is turned off. Hence, the /PWE is on for 2 (62.5 ns), meeting the timing requirement. The state machine remains in the Weoff state until the bus cycle is over, which sends the machine back to the idle state, where it turns off the memory chip-enables and awaits the next access to the memories. The write bus cycle is shown in Figure 8.5. Finally, the refresh bus cycle is shown in Figure 8.6.

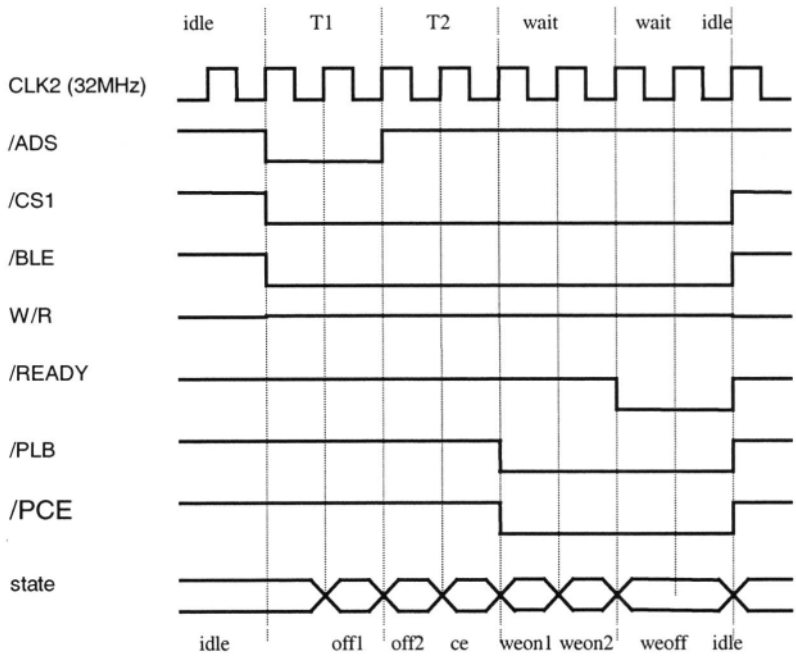



Figure 8.5 Write Cycle

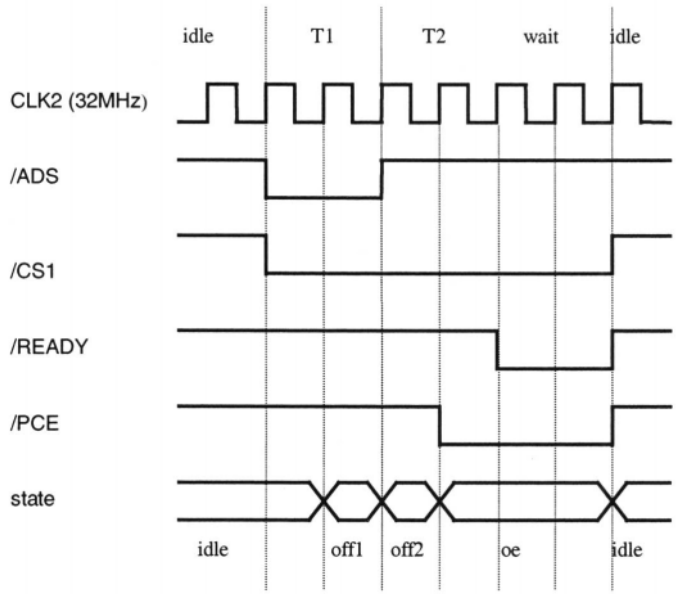



Figure 8.6 Refresh Cycle

This procedure allows for correct timing when accessing the P-SRAMs. The state machine above interfaces the 386EX bus to the P-SRAM memories. It is described using AHDL in Example 8.1.

Example 8.1 PSRAM Controller.

```
SUBDESIGN PSRAM_Controller
 (
    clk, reset                              :INPUT;
    /ADS,/RFSH,/CS1,/BLE,/BHE,/W/R          :INPUT;
    /PCS,/PLB,/PHB                          :OUTPUT;
    /PWE,/POE                               :OUTPUT;
```

```
VARIABLE
    Strobe,BE         :NODE;
    ss  :MACHINE WITH STATES(
            idd, off1, off2, ce, oe, we1, we2, weoff);

BEGIN
    ss.clk=clk;
    ss.reset=reset;

    /PLB=!(((!/BLE)&BE)&/RFSH);
    /PHB=!(((!/BHE)&BE)&/RFSH);
    Strobe    =(!/ADS)&((!/RFSH)#((!/CS1)&((!/BLE)#(!/BHE))));


TABLE

    ss, strobe, /RFSH,/W/R,/READY => ss,BE,/PWE,/POE,/PCS;

    idd,0,x,x,x                    => idd,0,1,1,1;
    idd,1,x,x,x                    => off1,0,1,1,1;
    off1,x,x,x,x                   =>off2,0,1,1,1;
    off2,x,0,x,x                   => oe,0,1,1,1;
    off2,x,1,x,x                   => ce,0,1,1,1;
    ce,x,x,1,x                     => we1,1,1,1,0;
    we1,x,x,1,x                    => we2,1,0,1,0;
    we2,x,x,1,x                    => weoff,1,0,1,0;
    weoff,x,x,x,1                  => weoff,1,1,1,0;
    weoff,x,x,x,0                  => idd,1,1,1,0;
    ce,x,x,0,x                     => oe,1,1,1,0;
    oe,x,x,x,0                     => idd,1,1,0,0;
    oe,x,x,x,1                     => oe,1,1,0,0;

END TABLE;

END;
```

## 8.3 Private Eye Controller

The Private Eye (PE) Controller contains an 8-bit shift register used to receive data in parallel from the microprocessor that is to be displayed and subsequently deliver that information in the serial form to the PE display adapter. The structure of PE Controller is shown in Figure 8.7. Besides the shift register, the controller contains a frequency divider and a counter. The frequency divider divides the system frequency of 32MHz by four to provide the frequency required by the PE display. The counter counts a number of bits transmitted from the Controller and stops shifting process when all bits are delivered.

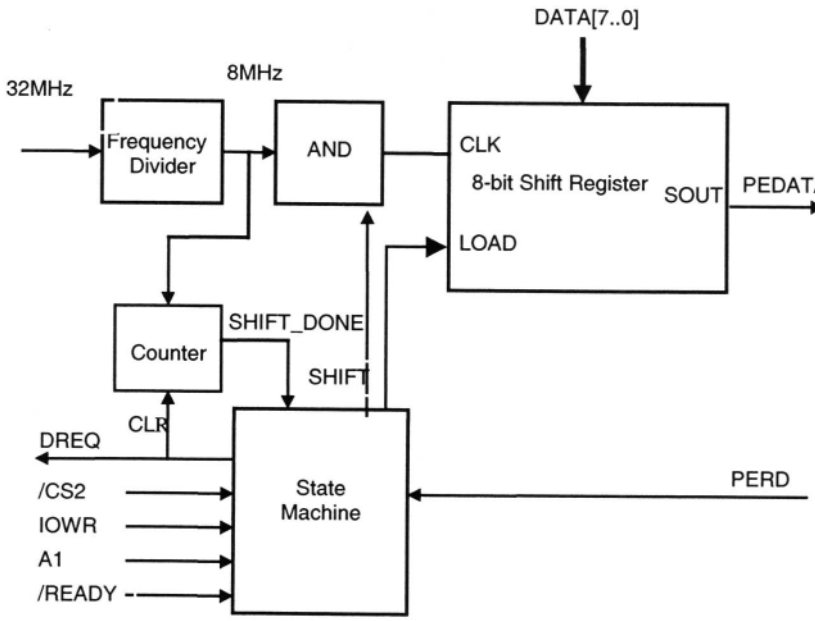



Figure 8.7 PE Controller Structure

Another state machine, described below, interfaces the system bus to the PE display adapter. It provides the PE Controller to be in one of four possible states:

- Idle (starting state from which it can be transferred into receiving state)
- Recv (receiving state in which it receives the next byte from the microprocessor to display it)
- Load (into which it comes upon receiving byte and loading it into shift register, and)
- Shift (the state in which it delivers byte to be displayed to the PE Controller bit by bit)

Its state transition diagram is depicted in Figure 8.8.

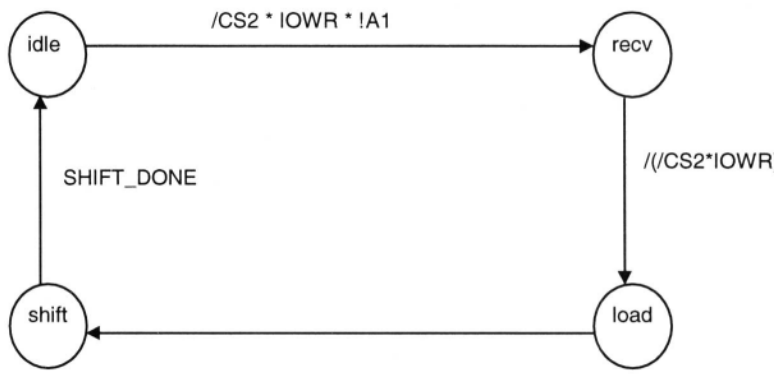



Figure 8.8 PE Controller State Transition Diagram

More detailed description of operation is given below. Signals used by the PE Controller are shown in Table 8.2.

The PE controller provides a mechanism by which the CPU can send pixel data to the display adapter. The PE (PE) display has 4 relevant signals: PEBOS, PERDY, PEDATA, and PECLK. The PEBOS (Beginning of Scan) tells the PE that new screen data is about to be delivered. The PERDY signal tells the controller when the PE is ready to receive data; when the PE refreshes the screen from its internal bitmap memory it deasserts PERDY and cannot accept data. The PEDATA and PECLK provide a data path from a host to the PE. The host places pixel data (1 bit = 1 pixel) on the PEDATA line and delivers it to the PE by clocking the PECLK. Since the screen consists of 720x280 pixels, the host must clock in 201,600 bits per screen. Also, the PE can accept data at a maximum rate of 8MHz.

The PE control resides in the processor's I/O space at addresses 0 and 2. I/O port 0 is used to deliver data to the PE and port 2 is used to set/examine the PEBOS and examine the state of PERDY and of the control state machine. The programmer has direct control over the PEBOS signal by way of port 2: by writing a value with bit 1 set to port 2, the programmer turns on the PEBOS. Likewise, by writing a value with bit 1 clear to I/O port 2, the programmer can turn off the BOS. Using these two features allows the host to issue a BOS pulse, which is necessary to tell the PE that a new screen is coming. After setting BOS, the host can write screen data to I/O port 0 to deliver the screen data. The host writes a byte (8 pixels) at a time, and the pixels (bits) contained in that byte will be shifted serially to the PE by the PE Controller.

Table 8.2 PE Controller Signals.

| Signal | Meaning |
|---|---|
| /CS2 (Chip-Select2) | asserted by processor when it wishes to communicate with the PE control PLD |
| A1 | bit 1 of the 26-bit address placed on the bus by the i386EX |
| IOWR | signifies that the current cycle is an I/O write cycle |
| SHIFT | asserted by the PE Controller when it enters the SHIFT state, indicating the PE Serial Data Shifter should begin delivering the data to the display adapter |
| SHIFT_DONE | asserted by PE Serial Data Shifter state machine indicating the shift is done |
| PERDY | asserted by PE when it is ready to accept data |

Two mechanisms exist for writing data to this data port: direct CPU I/O writes and DMA. For direct CPU I/O writes, the CPU will read a byte from the screen image in memory and write that byte to I/O port 0. Likewise, for DMA, the DMA controller will read a byte from memory and perform an I/O write cycle to port 0. The DMA, however, has a request signal, DREQ, that must be asserted before the transfer begins. The DMA is programmed with a requester, a target, and a byte count. Here, the requester is I/O port 0 (PE data port), the target is the screen image in memory, and the byte count is 201,600 / 8 = 25,200. Once the DMA channel is enabled, it will wait for the DREQ signal to be asserted. When DREQ is active, the DMA will read a byte from memory and write it to I/O port 0, then wait for DREQ again. When DREQ goes active again, the DMA will send the second byte, and so on. Hence, the PE controller must also handle the assertion of DREQ.

The PE controller manages this and behaves as follows. In the idle state, the controller is waiting to receive data, and DREQ is asserted in this state to tell the DMA controller that data should be delivered. When an I/O write cycle to I/O port 0 (initiated either by the CPU or by the DMA controller) occurs, the machine transitions to the RECV state. The processor asserts the PE Controller chip-select (/CS2) when there is I/O access to ports 0 or 2, so the Controller must examine bit 1 of the bus address to determine whether the access is to port 0 or port 2; the state machine only activates during writes to I/O port 0.

The controller remains in the RECV state until the I/O write is complete. The end of the I/O write cycle (denoted by the processor asserting /READY) latches the data byte on the bus into an 8-bit register in the PE Controller and sends the state

machine into the SHIFT state. Also, since the internal buffer is now full, the controller turns off the DREQ signal until the buffers free, telling the DMA to stop sending bytes. At the same time the counter is cleared and starts the counting. The least significant bit (LSB) of this register is attached to the PEDATA line. When the controller enters this state, it causes the PE data shifting to activate. The controller remains in this state until the shift is done, denoted by the internal counter.

Once activated, the Shifter begins to serially deliver the data byte to the PE. Before sending the first bit, it waits for the PE to be ready for data (indicated by an active PERDY). When the PE is ready, the PECLK is asserted, which will deliver one bit to the PE (since bit 0 of the data byte is tied to PEDATA, this bit is the one that is sent to the PE). This process repeats until all 8 bits have been delivered to the PE. Once this is done, the counter generates SHIFT_DONE signal and the Shifter returns to the idle state, and awaits another byte.

Hence, the PE Controller and Data Shifter act in tandem to deliver bytes from the CPU (or DMA) to the PE display adapter. Additionally, the transfer to the PE occurs in the background; once the CPU writes the byte to the PE Controller, it can continue processing while the byte is serially sent to the PE. An AHDL description of the PE Controller design is given in Example 8.2.

Example 8.2 PE Controller.

```
INCLUDE  "8shift"; %library designs not shown here%
INCLUDE "freqdiv";
INCLUDE "4count";

SUBDESIGN PEC
 (
    DATA [7..0], /CS2, A1, /READY, IOWR, PERDY : INPUT;
    clk, reset :INPUT;
    DREQ, PEDATA, PECLK :OUTPUT;
)

VARIABLE
    load_data, strobe, shift, shift_done       :NODE;
    pk              :NODE; %8 MHz clock%
    shifter                : 8shift;
    counter                : 4count;
    fdiv                   : freqdiv;
    ss :MACHINE WITH STATES (idle,recv, load, shift);

BEGIN
    ss.clk=clk;
    ss.reset=reset;
```

```
    fdiv.clk=clk;
    pk=fdiv.q; %output frequency from the divider%
    PECLK=pk & shift & PERDY;
    strobe=(!/CS2)&IOWR&(!A1) ;
    counter.clrn=!DREQ;
    counter.clk=PECLK;
    shifter.d[7..0]=DATA[7..0];
    shifter.clk=!PECLK; %ensure bit0 is delivered before%
                        %shifting%
    shifter.ld=!(load_data&(!/READY));
    PEDATA=shifter.q;
    shift_done=counter.qd;

TABLE
    ss, strobe, /READY, shift_done=>ss, DREQ, load_data,
                                                    shift;
    idle, 0,x,x     =>idle, 1,0,0;
    idle, 1,x,x =>recv, 1,0,0;
    recv, x,1,x =>recv, 0,0,0;
    recv, x,0,x =>load, 0,0,0;
    load, x,x,x =>shift,0,1,0;
    shift, x,x,0 =>shift, 0,0,1;
    shift, x,x,1 =>idle, 0,0,1;

END TABLE;

END;
```

These two subsystems, the P-SRAM Controller and the PE Controller, comprise the heart of the electronics, aside from the processing core. Implementing the memory and PE controllers quickly and easily allows for reduction in the complexity of the system. A simple MAX 7000 device was chosen to implement the function of these interfaces. The simplest low-power 7032S chip accommodates both interfaces with less than 80% of its utilization. In addition, the FPLDs support 5 volt as well as 3.3 volt signals, which accommodates the 5-volt PE nicely.

The complexity of the above interfaces requires much effort to implement using standard parts or custom logic. Using FPLD, however, allows the developer to deal with a high-level description of the subsystem's behavior, rather than with cumbersome low-level details. For example, the state machine can be intuitively represented as a collection of states and transitions. Hence, mapping the memory controller and the PE interface, the two most complex blocks of logic in the VuMan 3 system, to the FPLD helped eliminate much complexity. Therefore, using the FPLD allows rapid prototyping. Without the reduction in complexity and implementation detail, these subsystems would require months to implement. With FPLD in the developer's arsenal, such logic blocks can be designed and implemented in a week instead.

## 8.4 Secondary Logic

An FPLD also provides additional logic signals aside from the state machines. The PCIC requires ISA-bus signals to operate properly and the FPLD is used to perform the conversion from i386EX bus to ISA bus. Namely, the FPLD provided the ISA signals of IORD (I/O read cycle), IOWR (I/O write cycle), MRD (memory read cycle), and MWR (memory write cycle). Also, the FPLD generates the system clocks, EPROM chip-select signals, and buffer control signals used in interfacing the 5 Volt PCMCIA slots to the 3.3 Volt i386EX system bus. These designs are not presented in this Chapter.

The FPLDs, coupled with the 386EX processor core, comprise the essential logic blocks. These allow the system to interface to the memory and the display adapter. The serial controller establishes communications between the CPU and the input, power, real-time clock, and serial-number subsystems. With these components interconnected, the system is ready to function.

## 8.5 Questions and Problems

8.1 Assume that the SimP microprocessor has to be connected to SRAM memory of the 32Kb size. What modifications in the SimP's instruction set, data path, and control unit should be performed in order to enable access to this memory? Consider at least two cases: to add page register that will point to a current page or to add register indirect addressing mode with additional address register that will enable longer effective address.

8.2 Assume that SimP processor from Chapter 7 has to be interfaced with Private Eye Display from this Chapter. Design the interface and describe it AHDL.

8.3 Assume that a wearable computer has to be based on a custom processor that provides interfaces to larger memory, Private Eye Display, provides a universal serial receiver/transmitter (UART) for asynchronous serial transfers and one programmable 8-bit parallel port with individually programmable input/output bits. If the custom processor is based on SimP with some modifications of the instruction set, specify those modifications and design the whole microcomputer using AHDL. Implement the design in Altera FLEX 10K device. What is the minimum capacity device that accommodates the design?

# 9 INTRODUCTION TO VHDL

VHDL (VHSIC Hardware Description Language) is a language used to express complex digital systems concepts for documentation, simulation, verification and synthesis. The wide variety of design tools makes translation of designs described in VHDL into actual working systems in various target hardware technologies very fast and more reliable than in the past when using other tools for specification and design of digital systems. VHDL was first standardized in 1987 in IEEE 1076-1987 standard, and an updated and enhanced version of the language was released in 1993, known as IEEE 1076-1993. In this book VHDL is introduced less formally than in the corresponding standard or the other books. However, we expect that the reader will adopt VHDL easily having knowledge of the other hardware description language, AHDL, already presented in preceding chapters. Another book from the same author (see Selected readings at the end) deals with VHDL at a much more detailed level.

VHDL has had an enormous impact on digital systems design methodology promoting a hierarchical top-down design process similar to the design of programs using high-level programming languages such as Pascal or C++. It has contributed to the establishment of new design methodology, taking the designers away from low level details, such as transistors and logic gates, to a much higher level of abstraction of system description. Similarly, high-level programming languages take the designers away from the details of CPU registers, individual bits and assembly level programming. Unlike programming languages, VHDL provides mechanisms to describe concurrent events being of crucial importance for the description of behavior of hardware. This feature of the language is familiar to designers of digital systems who were using proprietary hardware description languages, such as PALASM, ABEL, or AHDL, used primarily to design for various types of PLDs. Another important feature of VHDL is that it allows design entry at different levels of abstraction making it useful not only to model at the high, behavioral level, but also at the level of simple netlists when needed. It allows part of the design to be described at a very high abstraction level, and part at the level of familiar component level, making it perfectly suitable for simulation. Once the design concepts have been checked, the part described at the high level can be redesigned using features which lead to the synthesizable description. The designers

can start using language at the very simple level, and introduce more advanced features of the language as they need them.

Having these features, the language provides all preconditions to change the design methodologies, resulting in such advantages as:

- shorter design time and reduced time to market
- reusability of already designed units
- fast exploration of design alternatives
- independence of the target implementation technology
- automated synthesis
- easy transportability to other similar design tools
- parallelization of the design process using a team work approach

By providing independence of the target implementation technology, VHDL enables the same design specification to be used regardless of the target technology, making it possible to implement the design, for example in either ASIC or FPLD. The power of VHDL goes even beyond this, enabling us to describe designs on such levels of abstraction as PCBs or MCMs which contain as their parts standard SSI, MSI, or LSI ICs, FPLDs and full-custom ICs. The designers are taken away from low-level details, and can spend more time on aspects of different architectures, design alternatives, and system and test issues. This becomes more important with the growth in complexity of FPLDs and ASICs exceeding the equivalent of 1,000,000 low-level gates. Top-down design methodology and hiding of the details at the higher levels of design hierarchy make readable and understandable designs possible.

## 9.1 What is VHDL for?

VHDL is a hardware description language, which is now an industry standard language used to document electronic systems design from the abstract to the concrete level. As such it aims to model the intended operation of the hardware of a digital system. VHDL is also used as a standardized input and output from various CAE tools that include simulation tools, synthesis tools, and layout tools. VHDL is firstly used for design entry to capture the intended design. For this purpose we use a text editor. The VHDL source code can be the input to simulation in order to verify the functionality of the system, or it can be passed to synthesis tools, which provide implementation of the design for a specific target technology. All examples of digital systems in this book are described in IEEE 1076-1987 VHDL, as its

newer revision IEEE 1076-1993 does not bring new features important for those who are either beginners or will use the language in a standard way. The examples are compiled and simulated with the Accolade PeakVHDL or Altera Max+Plus II compilers and simulation tools. Accolade's compiler and simulator are used for the conceptual stage of the design, and Altera's tools to provide a synthesis for FPLDs as a target technology. VHDL is a very difficult language to learn, and the best way of approaching it, is to use a subset initially, and to use new models and features as required.

VHDL consists of several parts organized as follows:

- The actual VHDL language as specified by IEEE standard
- Some additional data type declarations in the standard package called IEEE standard 1164
- A WORK library reserved for user's designs
- Vendor packages with vendor libraries
- User packages and libraries

A VHDL description lists a design's components and interconnections, and documents the system behavior. A VHDL description can be written at various levels of abstraction:

- Algorithmic or behavioral
- Register transfer
- Gate level functional with unit delay
- Gate level with detailed timing

Using top-down design methodology, a designer represents a system at a higher level of abstraction first, and in more details later. Some design decisions can be left for the latter phases of the design process. VHDL provides ways of abstracting design, or "hiding" implementation details. A designer can design with top down successive refinements specifying more details of how the design is done.

A design description or model, written in VHDL, can be run through a VHDL simulator to demonstrate the behavior of the modeled system. Simulating a design model requires simulated stimulus, a way of observing the model during simulation,

and capturing the results of simulation for later evaluation. VHDL supports a variety of data types useful to the hardware modeler for both simulation and synthesis. These data types will be introduced throughout the following chapters, starting with the simple ones and then presenting advanced types, which make VHDL unique among hardware description languages.

Some parts of VHDL can be used with logic synthesis tools for producing physical design. Many VLSI gate-array or FPLD vendors can convert a VHDL design description into a gate level netlist from which a customized integrated circuit or FPLD implemented piece component can be built. Therefore, VHDL can be applied for the following:

- Documenting a design in a standard way. This guarantees support by newer generations of design tools, and easy transportability of the design to other simulation and synthesis environments.
- Simulating the behavior, which helps verification of the design often using a behavioral instead of a detailed component model. It has many features that enable description of the behavior of an electronic system from the level of a simple gate to the level of the complete microcontrollers or custom chips. The resulting simulation models can be used as building blocks for larger systems which use either VHDL or other design entry methods. Furthermore, VHDL enables specification of test benches, which describe circuit stimuli and expected outputs that verify behavior of a system over time. They are an integral part of any VHDL project and are developed in parallel with the model of the system under design.
- Directly synthesizing logic. Many of the VHDL features, when used in system description, provide not only simulatable but also synthesizable models. After compilation process, the system model is transformed into netlists of low level components that are placed and routed to the chosen target implementation technology. In the case of designs and models presented in this book, the target technology are Altera's FPLDs, although the design can be easily targeted to FPLDs of the other vendors or ASICs.

Designing in VHDL is like programming in many ways. Compiling and running a VHDL design is similar to compiling and running other programming languages. As the result of compiling, an object module is produced and placed in a special VHDL library. A simulation run is done subsequently by selecting the object units from the library and loading them onto the simulator. The main difference is that VHDL design always runs in simulated time, and events occur in successive time steps.

However, there are several differences between VHDL and conventional programming languages. The major differences are the notions of delay and simulation environment, and also the concurrency and component netlisting, which are not found in programming languages. VHDL supports concurrency using the concept of concurrent statements running in simulated time. Simulated time is feature found only in simulation languages. Also, there are sequential statements in VHDL to describe algorithmic behavior.

Design hierarchy in VHDL is accomplished by separately compiling components that are instantiated in a higher-level component. The linking process is done either by compiler or by simulator using the VHDL library mechanism.

Some software systems have version-control systems to generate different versions of loadable program. VHDL has a configuration capability for generating design variations. If not supported by specific simulators and synthesis tools, it is usually by default taken that the latest compiled design is one which is used in further designs.

## 9.2 VHDL Designs

Digital systems are modeled and designed in VHDL using a top-down approach to partition the design into smaller abstract blocks known as components. Each component represents an instant of a design entity, which is usually modeled in a separate file. A total system is then described as a design hierarchy of components making a single higher level component. This approach to the design will be emphasized and used throughout all examples presented in the book.

A VHDL design consists of several separate design units, each of which is compiled and saved in a library. The four source design units that can be compiled are:

1. **Entity**, that describes the design's interface signals and represents the most basic building block in a design. If the design is hierarchical, then the top-level description (entity) will have lower-level descriptions (entities) contained in it.
2. **Architecture**, that describes design's behavior. A single entity can have multiple architectures. Architectures might be of behavioral or structural type, for example.
3. **Configuration**, that selects a variation of design from a design library. It is used to bind a component instance to an entity-architecture pair. A configuration can be considered as a parts list for a design. It describes which behavior to use for each entity.

4. **Package**, that stores together, for convenience, certain frequently used specifications such as data types and subprograms used in a design. Package can be considered as a toolbox used to build designs. Items defined within package can be made visible to any other design unit. They can also be compiled into libraries and used in other designs by a **use** statement.

Typically, a designer's architecture uses previously compiled components from an ASIC or FPLD vendor library. Once compiled, a design becomes a component in a library that may be used in other designs. Additional compiled vendors' packages are also stored in a library.

By separating the entity (I/O interface of a design) from its actual architecture implementation, a designer can change one part of a design without recompiling other parts. In this way a feature of reusability is implemented. For example, a CPU containing a precompiled ALU saves recompiling time. Configurations provide an extra degree of flexibility by saving variations of a design (for example, two versions of CPU, each with a different ALU). A configuration is a named and compiled unit stored in the library.

The designer defines the basic building blocks of VHDL in the following sections:

- Library
- Package
- Entity
- Architecture
- Configuration

In order to introduce intuitively the meanings of these sections, an example of a design unit contained in file my_design.vhd is given in example 9.1 below.

Example 9.1 First VHDL design

```
package my_units is        --package--
   constant unit_delay:  time :=10 ns ;
end my_units;

entity compare is          --entity--
   port  (a, b : in bit ;
          c : out bit);
```

```
end  compare;

library my_library;
use my_library.my_units.all;

architecture first of compare is  --architecture--

begin
    c <=not  (a xor b)  after unit_delay;
end first;
```

There are three design units in a design my_design.vhd. After compilation, there are four compiled units in library my_library:

- Package `my_units` - provides a shareable constant
- Entity `compare` - names the design and signal ports
- Architecture `first` of `compare` - provides details of the design
- A configuration of `compare` - designates `first` as the latest compiled architecture.

Each design unit can be in a separate file and could be compiled separately, but the order of compilations must be as it is shown in the example above. The package my_units can also be used in other designs. The design entity compare can now be accessed for simulation, or used as a component in another design. To use compare, two input values of type bit are required at pins a and b; 10 ns latter a '1' or '0' appears at output pin c.

Keywords of the language are given and will be shown in bold letters. For instance, in the preceding example, the keywords are **architecture**, **package**, **entity**, **begin**, **end**, **is**, etc. Names of user-created objects, such as *compare*, will be shown in lowercase letters. However, it should be pointed out, VHDL is not case sensitive, and this convention is used just for readability purpose. A typical relationship between design units in a VHDL description is illustrated in Figure 9.1.

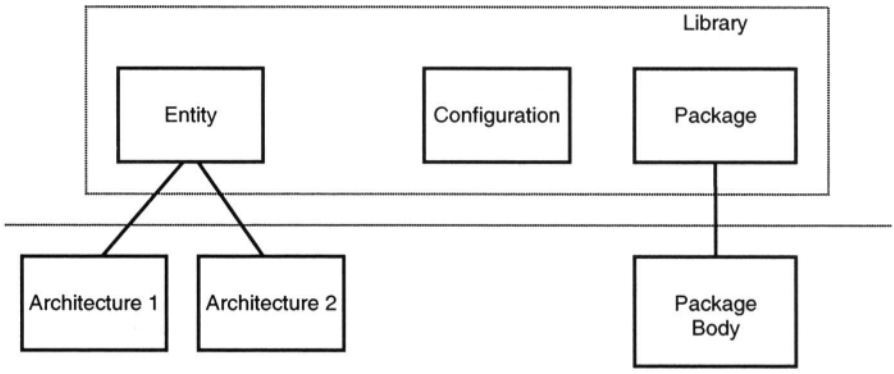



Figure 9.1 VHDL Design Units and Relationships

Basic VHDL design units are described in more details in the following sections.

## 9.3 Library

The results of a VHDL compilation are stored in a library for subsequent simulation, or for use in further or other designs. A library can contain:

- A package - shared declarations
- An entity - shared designs
- An architecture - shared design implementations
- A configuration - shared design versions

The two built-in libraries are WORK and STANDARD, but the user can create other libraries. VHDL source design units are compiled into WORK library unless a user directs it to another library.

To access an existing library unit in a library as a part of new VHDL design, the library name must be declared first. The syntax is:

```
library logical_name;
```

Now, component designs compiled into the specified library can be used. Packages in the library can be accessed via a subsequent **use** statement. If WORK library is used, it does not need to be declared.

Compiled units within a library can be accessed with up to three levels of names:

```
  library_name.package_name.item_name
or

  library_name.item_name
or

  item_name              if the WORK library is assumed
```

Units in a library must have unique names; all design entity names and package names are unique within a library. Architecture names need to be unique to a particular design entity.

In order to locate a VHDL library in a file system, it is sometimes necessary to issue the commands outside of the VHDL language. This is compiler and system dependent, and a user has to refer to appropriate vendor's literature.

## 9.4 Package

The next level of hierarchy within a library is a package. A package collects a group of related declarations together. Typically, a package is used for:

- Function and procedure declarations
- Type and subtype declarations
- Constant declarations
- File declarations
- Global signal declarations
- Alias declarations
- Attribute specifications
- Component declarations
- Use clauses

Package is created to store common subprograms, data types, constants and compiled design interfaces that will be used in more than one design. This strategy promotes the reusability.

A package consists of two separate design units: the package header, which identifies all of the names and items, and the optional package body, which gives more details of the named item.

All vendors provide a package named STANDARD in a predefined library named STD. This package defines useful data types, such as bit, boolean, and bit_vector. There is also a text I/O package called TEXTIO in STD.

A use clause allows access to a package in a library. No **use** clause is required for the package STANDARD. The default is:

```
library STD;
use STD.STANDARD.all;
```

Additionally, component or CAD tool vendors provide packages of utility routines and design pieces to assists design work. For example, VHDL descriptions of frequently used CMOS gate components are compiled into a separate library, and their declarations are kept in a package.

## 9.5 Entity

The design entity defines a new component name, its input/output connections, and related declarations. The entity represents the I/O interface or external specification to a component design. VHDL separates the interface to a design from the details of architectural implementation. The entity describes the type and direction of signal connections. On the other side, an architecture describes the behavior of a component. After an entity is compiled into a library, it can be simulated or used as a component in another design. An entity must have an unique name within a library. If a component has signal ports, they are declared in an entity declaration. The syntax used to declare an entity is:

```
entity entity_name is

    [generics]
    [ports] [declarations {constants, types, signals}]
    [begin statements]   --Typically not used

end [entity] entity_name;
```

An entity specifies the external connections of a component. In Figure 9.2 an AND gate (*and gate*) with two signal lines coming in, and one going out, is presented.

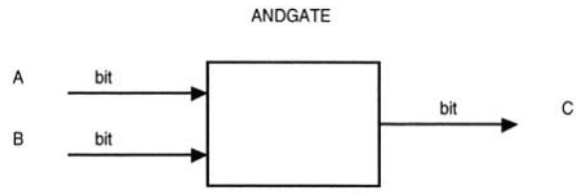



Figure 9.2 Example of AND gate

The diagram emphasizes the interface to the design. All signals are of the bit type, which mandates the usage; the andgate design only works on bit type data. VHDL declaration of this entity is:

```
entity andgate is

   port   (a, b: in bit;
           c: out bit);
end andgate;
```

In this example *andgate* is defined as a new component. The reserved word **is** is followed by the port declarations, with their names, directions (or mode in VHDL) and types.

Any declaration used in an entity port must be previously declared. When an entity is compiled into a library, it becomes a component design that can be used in another design. A component can be used without the knowledge of its internal design details.

All designs are created from entities. An entity in VHDL corresponds directly to a symbol in the traditional schematic entry methodology. The input ports in the preceding example directly correspond to the two input pins, and the output port corresponds to the output pin.

Optionally, the designer may also include a special type of parameter list, called a generic list, which allows additional information to pass into an entity. This information can be especially useful for simulation of the design model, but also for parameterization of the design.

## 9.6 Architecture

An architecture design unit specifies the behavior, interconnections, and components of a previously compiled design entity. The architecture defines the function of the design entity. It specifies the relationships between the inputs and outputs that must be expressed in terms of behavior, dataflow, or structure. The entity design unit must be compiled before the compilation of its architecture. If an entity is recompiled, all its architectures must be recompiled, too.

VHDL allows the designer to model a design at several levels of abstraction or with various implementations. An entity may be implemented with more than one architecture. Figure 9.3 illustrates two different architectures of entity *alu.*

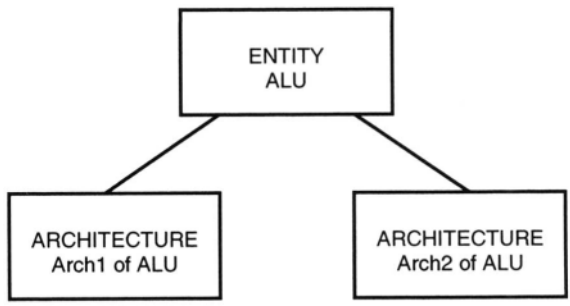



Figure 9.3 Entity with two different architectures

All architectures have identical interfaces, but each needs an unique architecture name. A designer selects a particular architecture of a design entity during configuration (for example *arch1*).

VHDL architectures are generally categorized in styles as:

- Behavioral - defines sequentially described process
- Dataflow - implies a structure and behavior
- Structural - defines interconnections of components

Different styles of VHDL designs actually represent different levels of abstraction of using the language. Generally speaking we can associate levels of abstraction with the architecture styles as in Table 9.1, although the boundaries between

different styles are not strict, and often in the same model we can use a mix of these styles.

A design can use any or all of these design styles. Generally, designs are created hierarchically using previously compiled design entities. They can only be combined using structural style which looks like a list of components wired together (i.e., netlist).

Table 9.1 Architecture styles and levels of abstraction

| Architecture style | Level of abstraction |
|---|---|
| Structural | Physical information<br>Hierarchy of gates<br>Boolean Equations |
| Dataflow | Arithmetic operations<br>Concurrent signal assignment<br>Register transfers |
| Behavioral | State machines<br>Sequential descriptions<br>Test benches<br>High level specifications |

The architecture is defined in VHDL with the following syntax:

```
architecture architecture_name of entity_name is
    [architecture_declarative_part]
begin
    [architecture_statement_part]
end [architecture] [architecture_name];
```

The architecture_declarative_part declares items used only in this architecture such as types, subprograms, constants, local signals and components are declared, the architecture_statement_part is the actual design description, all statements between the **begin** and **end** statement are called concurrent statements, because all of the statements execute concurrently. This concept is analogous to the concept of concurrent statements in AHDL.

The architecture can be considered as a counterpart to the schematic for the component in traditional designs.

### 9.6.1 Behavioral Style Architecture

An example of an architecture called *arch1* of entity *andgate* is shown in Example 9.2 below.

Example 9.2 Behavioral architecture of an AND gate

```
architecture archl of andgate is

begin
   process (a, b) ;
   begin
      if a ='1' and b ='1' then
        c <='1' after 1 ns;
      else
        c <='0' after 1 ns;
      endif;
    end process;
end archl;
```

It contains a process that uses signal assignment statements. If both input signals a and b have the value '1', c gets a '1'; otherwise c gets a '0'. This architecture describes a behavior in a "program-like" or algorithmic manner.

VHDL processes may run concurrently. The list of signals for which the process is waiting (sensitive to) is shown in parentheses after the word process. Processes wait for changes in an incoming signal. Process is activated whenever input signals change. The output delay of signal c depends upon the **after** clause in the assignment.

Parallel operations can be represented with multiple processes. An example of processes running in parallel is shown in Figure 9.4. The processes communicate with each other; they transfer data with signals. A process gets its data from outside from a signal. Inside, the process operates with variables. The variables are local storage and cannot be used to transfer information outside the process. Sequential statements, contained in process, execute in order of appearance as in conventional programming languages.

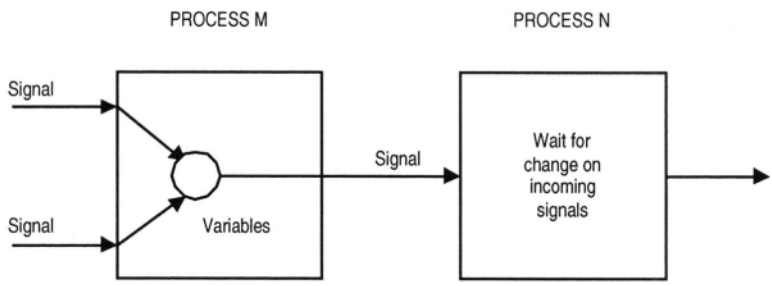



Figure 9.4 Process model

Process N in Figure 1.4 receives signals from process M. The running of one process can depend upon results of operation of another process.

In top-down design style, behavioral description is usually the first step; the designer focuses on the "abstract" behavior design. Later, the designer can choose the precise signal-bus and coding.

### *9.6.2 Dataflow Style Architecture*

Dataflow architecture models the information or dataflow behavior of combinational logic functions such as adders, comparators, multiplexers, decoders, and other primitive logic circuits. Example 9.3 defines the entity and architecture, in a dataflow style, of xor2, an exclusive-OR gate. xor2 has input ports a and b of type bit, and an output port c of type bit. There is also a delay parameter m, which defaults to 1.0ns. The architecture dataflow gives output c exclusive-OR of a and b after m (1ns).

Example 9.3 Dataflow type architecture of an xor gate

```
entity xor2 is

   generic (m: time :=1.0 ns);
   port (a, b: in bit;
      c: out bit);
end xor2;
```

```
architecture dataflow of xor2 is
begin
    c <= a xor b after m;
end dataflow;
```

Once this simple gate is compiled into a library, it can be used as a component in another design by referring to the entity name xor2, and providing three port parameters and, optionally, a delay parameter.

### *9.6.3 Structural Style Architecture*

Top-level VHDL designs use structural style to instance and connect previously compiled designs. Example 9.4 uses two gates xor2 (exclusive-OR) and inv (inverter) to realize a simple comparator. The schematic in Figure 9.5 represents a comparator. Inputs in the circuit, labeled a and b, are inputs into first *xor2* gate. The signal wire I from *xor2* connects to the next component *inv*, which provides an output c.

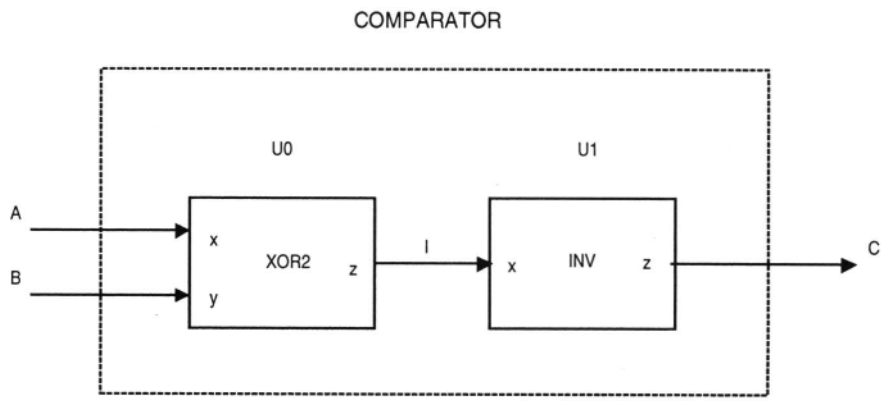



Figure 9.5 Schematic representation of comparator

Example 9.4 Structural architecture of a comparator

```
entity comparator is
   port (a, b: in bit; c: out bit);
end comparator;

architecture structural of comparator is
   signal i: bit;
```

```
component xor2
   port (x, y: in bit; z: out bit);
end component;

component inv
   port (x: in bit; z: out bit);
end component;

begin
   u0:   xor2 port map (a, b, i);
   u1:   inv port map (i, c) ;
end structural;
```

The architecture has an arbitrary name structural. Local signal i is declared in the declaration part of architecture. The component declarations are required unless these declarations are placed in a package. Two components are given instance names u0 and u1. The port map indicates the signal connections to be used. The design entities xor2 and inv are found in library WORK, since no library is declared.

Which architecture will be used depends on the accuracy wanted, and whether structural information is required. If the model is going to be used for PCB layout purposes, then probably the structural architecture is most appropriate. For simulation purposes, however, behavioral models are probably more efficient in terms of memory space required and speed of execution.

## 9.7 Configuration

The configuration assists the designer in experimenting with different variations of a design by selecting particular architectures. Two different architectures of the entity andgate, called arch1 and arch2, have been illustrated in Figure 9.3. A configuration selects a particular architecture, for example arch1, from a library. The syntax is:

```
configuration identifier of entity_name is

   [specification]

end configuration identifier;
```

Different architectures may use different algorithms or levels of abstraction. If the design uses a particular architecture, a **configuration** statement is used. A configuration is a named and compiled unit stored in library. The VHDL source description of a configuration identifies by name other units from a library. For example:

```
configuration alul_fast of alu is

for alul;
for u0:comparator use entity work.comparator(dataflow);
```

In this example configuration alul_fast is created for the entity alu and architecture alul. The **use** clause identifies a library, entity, and architecture of a component (comparator). The final result is the configuration called alul_fast. It is a variation of design alu. Configuration statements permit selection of a particular architecture. When no explicit configuration exists, the latest compiled architecture is used (it is called null configuration).

The power of the configuration is that recompilation of the whole design is not needed when using another architecture; instead, only recompilation of the new configuration is needed.

Configuration declarations are optional regardless of the complexity of a design. If configuration declaration is not used, the VHDL standard specifies a set of rules that provide the design with a default configuration. For example, if an entity has more than one architecture, the last architecture compiled will be bound to the entity.

## 9.8 Questions and Problems

9.1 Describe in your own words what are:

- Printed Circuit Board (PCB)
- Multi-Chip Module (MCM)
- Field-Programmable Logic Device (FPLD)
- Application-Specific Integrated Circuit (ASIC)

Try to explain how you would model each of them using features and different types of models supported by VHDL.

9.2 How would you define and describe modeling, design, prototyping, simulation, and synthesis?

9.3 Knowing typical architectures of logic elements in FPLDs, try to model conceptually one of standard logic elements using VHDL:

- Logic element from MAX 7000 FPLDs
- Logic element from FLEX 10K FPLDs

9.4 What types of modeling are supported in VHDL? Use as an example a simple digital circuit a shift register for which you can describe all types of modeling.

9.5 What are the major similarities and differences between VHDL and high-level programming languages?

9.6 What are the VHDL design units? On an example of a digital system illustrate in parallel VHDL design units and equivalent descriptions using conventional tools for describing digital systems.

9.7 Compare design units of VHDL with equivalent constructs of AHDL. What do you see as major similarities at the first glance?

9.8 How are parallel operations described in VHDL?

9.9 What do you consider under “partial recompilation” in VHDL?

# 10 OBJECTS, DATA TYPES AND PROCESSES

VHDL includes a number of language elements, called objects that can be used to represent and store data in the system being modeled. The three basic types of objects used in description of a design are signals, variables and constants. Each object has its name and a specific data type, and a unique set of possible data values. VHDL provides a variety of data types and operators in the package STANDARD that support the methodology of top-down design, using abstractions of hardware in early versions of design. Recent changes in the language itself extended standards further. These changes helped synthesis tool users and vendors by making standard, portable data types and operations for numeric data, and by clarifying meaning for values in IEEE 1164 data types. In this chapter we will concentrate on the basic language elements first and then on more advanced features. Besides standard types and operations, it supports user defined data types that can be included in own user packages. The advanced data types include enumerated types that allow for identifying specified values for a type and for subtypes, which are variations of existing types. There are composite types that include arrays and records. In this chapter we cover only those types that are of interest for synthesis purposes.

VHDL is a strongly typed language, which assists designers to catch errors early in the development cycle. The compiler's analyzer is very exact and displays the errors for not using the correct data representation.

The primary concurrent statement in VHDL used for behavioral modeling is a PROCESS statement. A number of processes may run at the same simulated time. Within a process, sequential statements specify the step-by-step behavior of the process, or, essentially, the behavior of architecture. Sequential statements define algorithms for the execution within a process or a subprogram. They belong to the conventional notions of sequential flow, control, conditionals, and iterations in the high level programming languages such as Pascal, C, or Ada. They execute in the order in which they appear in the process. In an architecture for an entity, all statements are concurrent. The process statement is itself a concurrent statement. It can exist in an architecture and define regions in the architecture where all

statements are sequential. This chapter covers also the basic features of processes and their use.

## 10.1 Literals

A literal is an explicit data value which can be assigned to an object or used within expressions. Although literals represent specific values, they do not always have an explicit type. A scalar is a literal made up of characters or digits or it can be named. There are predefined scalar types, but the user can define other types. The predefined scalar types are:

- character
- bit
- real
- integer
- physical_unit

### *10.1.1 Character and String Literals*

A character literal defines 1-character ASCII value by using a single character enclosed in single quotes **'x'.** Although VHDL is not case sensitive, it does consider case for character literals. Character literal can be any alphabetic letter a-z, digit 0-9, blank, or special character. Examples of character literals are

**'X' 'x' '1' '$'**

The data type of the object assigned these values dictates whether a given character literal is valid. The same value, for example **'A',** is a valid literal when assigned to a **character** type object, but is not valid when assigned to a **bit** data type.

Literal character strings are collections of one or more ASCII characters and are enclosed in double quote characters. For example:

"the value must be in the range"

They can be assigned to arrays of single-character data types or objects of the built-in type **string.** They are useful when designing test benches around the synthesizable model.

### *10.1.2 Bit, Bit String and Boolean Literals*

The value of the signal in a digital system is often represented by a bit. A bit literal represents a value by using the character literals '0', or '1'. Bit literals differ from integers or real numbers. Bit data is also distinct from Boolean data, although conversion functions may be implemented. Bit string literal is an array of bits enclosed in double quotes. They are used to represent binary, octal and hexadecimal numeric data values. When representing a binary number, a bit string literal must be preceded by the special character 'B', and may contain only the characters '0' and '1'. When representing an octal number, the bit string literal must include only characters '0' through '7', and it must be preceded by the special character 'O'. When representing a hexadecimal value, the bit string literal may include only characters '0' through '9' and 'A' through 'F', and must be preceded by the special character 'X'. The underscore character '_' may be used within bit string literals to improve readability, but has no effect on the value of the bit string literal. Examples of bit string literals are:

```
B"011001"
B"1011_0011"
O"1736"
X"EB0F"
```

Bit literals are used to describe the value of a bus in a digital system. Most simulators include additional bit types representing unknowns, high impedance states, or other electrically related values. In VHDL standard 1076-1987, bit string literals are only valid for the built-in type **bit_vector,** but in 1076-1993 standard they can be applied to any string type including **std_logic_vector.**

A Boolean literal represents a true or false value. It has no relationship to a bit. Relational operators like **=, <=, >=,** and **/=** produce a Boolean result. Boolean literals are

```
true  TRUE  True  false  FALSE  False
```

A Boolean signal is often used to represent the state of an electronic signal or a condition on a bus.

### *10.1.3 Numeric Literals*

Two basic types of numeric literals are supported in VHDL, real literals and integer literals.

Real literals define a value with real numbers. They represent numbers from -1.0E+38 to +1.0E+38. A real number must always be written with a decimal point. Examples of real literals are:

+1.25
3.4
-2.5

Integer literals define values of integers in the range -2,147,483,647 to +2,147,483,647 (32 bits of precision, including the sign bit), but the instances can be constrained to any subrange of this one. It is not allowed to use decimal point in representing integers. Examples of integers are:

+5
-223
123

When the bit operations are needed, conversion functions from integers to bits must be applied. During design development, integers can be used as an abstraction of a signal bus or may represent an exact specification.

Numeric literals may include underscore character '_' to improve readability.

### *10.1.4 Physical literals*

Physical literal represents a unit of measurement. VHDL allows the use of a number and unit of measure, such as voltage, capacitance, and time. Number must be separated from unit by at least one space. Examples of physical literals are:

1000 ps (picoseconds)
2 min (minutes)
12 v (volts)

Physical literals are most useful in modeling and representing physical conditions during design testing.

### *10.1.5 Range Constraint*

A range constraint declares the valid values for a particular type of signal or variable assignment. It must be compatible with the type it constrains, and be in a compatible direction with the original declaration of the type. The syntax is

```
range low_val to high_val
```

In the example below a range constraint is used in port declaration:

```
port (b, a: in integer range 0 to 9 :=0)
```

### *10.1.6 Comments*

Comments in VHDL start with two adjacent hyphens (‘- -‘) and extend to the end of the line. They have no part in the meaning of a VHDL description.

## 10.2 Objects in VHDL

VHDL includes a number of language elements, called objects that can be used to represent and store data in the system being modeled. The three basic types of objects used in description of a design are signals, variables and constants. Each object has its name and a specific data type, and a unique set of possible data values.

### *10.2.1 Names and Named Objects*

Symbolic names are used for objects in VHDL. A name (identifier) must begin with an alphabetic letter (a-z), followed by a letter, underscore or digit.

Other named objects are architecture names, process names, and entity names. VHDL has over 100 reserved words that may not be used as identifiers.

Examples of names in VHDL are:

my_unit
x_5 X23
my_unit.unit_delay

Examples of invalid names (reserved words) are:

```
process
    in
    out
    library
    map
```

Names are usually relative to a named entity and can be selected from a package or a library:

```
library_name.item_name
package_name.item_name
```

Named signal in VHDL represents a wire in a physical design. This signal is represented by a stored value during simulation. This allows us to observe changes in a signal value. Named objects are either constant (like fixed value of the signal) or varying in value.

Unlike programming languages, VHDL has two elements that can vary: the variable, which behaves just like a programming language variable, and the signal, which is assigned value at some specific simulated time. The type of variables and signals must be declared in VHDL. There are three object declarations in VHDL:

- constant_declaration
- signal_declaration
- variable_declaration

### *10.2.2 Indexed names*

Variables and signals can be scalars or arrays. Array references can be made to the entire array, to an element, or to a slice of an array. Examples are:

```
a           Array
a (5)           Element of array
a (1 to 5)              Slice of an array
```

Arrays are especially useful in documenting a group of related signals such as a bus.

### 10.2.3 Constants

A constant is name assigned to a fixed value when declared. Constants are useful for creating more readable designs, and make easier to change the design at a later time. If it is necessary to change the value of a constant, it is needed to change the constant declaration in one place. A constant consists of a name, a type, and a value. The syntax is:

```
constant identifier: type_indication [:=expression];
```

Examples of constant declarations are:

```
constant register: bit_vector (0 to 15):=X"ABCD"
constant v: real := 3.6;
constant t1: time := 10 ns;
```

Constants can be declared in a package, in a design entity, an architecture, or a subprogram. Frequently used or shared constants should be declared in a user-defined package. A constant specification can also be used to specify a permanent electrical signal in a digital circuit.

### 10.2.4 Variables

A variable is a name assigned to a changing value within a process. It is used to store intermediate values between sequential VHDL statements. A variable assignment occurs immediately in simulation, as opposed to a signal that is scheduled in simulated time. A variable must be declared before its use. The syntax is:

```
variableidentifier(s):type_indication
    [constraint][:=expression];
```

A variable can be given a range constraint or an initial value. The initial value, by default, is the lowest (leftmost) value of range for that type. Examples of variable declarations are:

```
variable alpha: integer range 1 to 90 :=2;
variable x, y: integer;
```

Variables are scalars or arrays that can only be declared in a process or a subprogram. They represent a local data storage during simulation of a process or subprogram. Variables cannot be used to communicate between processes. The important distinctions between variables and signals are covered in more detail in the later sections.

### *10.2.5 Signals*

Signals connect concurrent design entities together and communicate changes in values within an electronic design. Signal assignments use simulated time to execute in VHDL. A signal must be declared before it is used. The syntax is:

```
signal identifier: type_indication
    [constraint]  [:=expression]
```

Signals can be declared in an entity, an architecture, or in a package. If signal has to be initialized, it is indicated by literal in [:=expression]. The default initial value is the lowest value of that type. Examples of signal declaration are:

```
signal cnt: integer range 1 to 10;
signal gnd: bit :='0';
signal abus: std_logic_vector (7 downto 0):=
          (others=>'1");
```

The last signal declaration and initialization statement assigns all signals of the array abus initial value of '1'. Initialization values are commonly ignored by synthesis tools. However, they can be useful for simulation purposes.

Signal value changes are scheduled in simulated time. For example:

```
signal s: bit;
s <= '1' after 2 ns;
```

Signals cannot be declared in a process. If they are used within a process, unexpected results can be obtained because the value assignment is delayed until WAIT statement is executed. They provide global communication in an architecture or entity. Signals are usually used as abstractions of physical wires, busses, or to document wires in an actual circuit.

## 10.3 Expressions

An expression is a formula that uses operators and defines how to compute or qualify the value. The operators must perform a calculation compatible with its operands. Generally, operands must be of the same type. No automatic type conversion is done. In an expression, an operand can be a name, a numeric, or a character literal, but also a function call, qualified expression, type conversion, etc. The result of an expression has a type that depends upon the types of operands and operators.

A summary of VHDL operators is presented in Table 10.1. These operators create expressions that can calculate values. Logical operators, for example, work on predefined types, either bit or Boolean. They must not be mixed. The resulting expression has the same type as the type of operand. Relational operators compare two operands of the same type and produce a Boolean. The result of an expression formed with a relational operator is of type Boolean.

Concatenation is defined for characters, strings, bits, and bit vectors and for all one-dimensional array operands. The concatenation operator builds arrays by combining the operands. For example:

"ABCDEF" & "abcdef" results in "ABCDEFabcdef"
"11111" & "00000" results in "1111100000"

in some cases operators are specifications for a hardware block to be built using logic synthesis tools. A plus (+) corresponds to an adder, and logical operators are models of gates. Table 10.2 lists precedence of operators. Each row represents operators with the same precedence. An operator's precedence determines whether it is applied before or after adjoining operators.

The default precedence level of the operators can be overridden by using the parentheses. More detailed insight to the use of VHDL operators will be covered in the later sections through a number of example designs.

## 10.4 Basic Data Types

VHDL allows the use of variety of data types, from scalar numeric types, to composite arrays and records, or file types. In the preceding chapters we have introduced the basic data types and objects supported by VHDL, particularly:

- signals, that represent interconnection wires that connect component instantiation ports together
- variables, that are used for local storage of temporary data visible only inside a process, and
- constants, that are used to name specific value

All these objects can be declared using a type specification to specify the characteristics of the object. VHDL contains a wide range of types that can be used to create objects. To define a new type, a type declaration must be used. A type declaration defines the name of the type and the range of the type. Type declarations

are allowed in package declaration sections, entity declaration sections, architecture declaration sections, subprogram declaration sections, and process declaration sections.

The four broad categories of the types available in VHDL are

- Scalar types, that represent a single numeric value. The standard types belonging to this class are integer, real, physical, and enumerated types.
- Composite types, that represent a collection of values. There are two classes of composite types: arrays which contain elements of the same type, and records which contain elements of different types.
- Access types, that provide references to objects similar to the pointers used to reference data in programming languages.
- File types, that reference objects that contain a sequence of values (for example, disk files)

Table 10.1 Operators in VHDL

| Operations | Operators | Description | Operands |
|---|---|---|---|
| logical | and | logical and | same type |
| | or | logical or | same type |
| | nand | complement of and | same type |
| | nor | complement of or | same type |
| | xor | logical exclusive or | same type |
| | not | complement | same type |
| | xnor | logical exclusive nor | |
| relational | = | equal | same type |
| | /= | not equal | same type |
| | < | less than | same type |
| | <= | less than or equal | same type |
| | > | greater than | same type |
| | >= | greater than or equal | same type |

| | | | |
|---|---|---|---|
| arithmetic | +<br>-<br>abs<br><br>+<br>-<br><br><br>*<br>/<br><br>mod<br>rem<br>** | unary plus<br>unary minus<br>absolute value<br><br>addition<br>subtraction<br><br><br>multiplication<br>division<br><br>modulus<br>remainder<br>exponentiation | any numeric<br>any numeric<br>any numeric<br><br>same type<br>same type<br><br><br>same type<br>same type<br><br>same type<br>same type |
| shift<br>(1076-1993 only) | sll<br>srl<br>sla<br>sra<br>rol<br>ror | logical shift left<br>logical shift right<br>arithmetic shift left<br>arithmetic shift right<br>logical rotate left<br>logical rotate right | one-dimensional integer array of bit or boolean |
| concatenation | & | concatenation | same type |

Table 10.2 Operators and their precedence

| Type | | | | | | | | Precedence |
|---|---|---|---|---|---|---|---|---|
| Logical | **and** | **or** | **nand** | **nor** | **xor** | | | Lowest |
| Relational | = | /= | | < | <= | > | >= | |
| Adding | + | - | & | | | | | |
| Unary | + | - | | | | | | |
| Multiplying | * | / | **mod** | **rem** | | | | |
| Misc. | ** | **abs** | **not** | | | | | Highest |

Each type in VHDL has a defined set of values. In most cases the designer is interested only in a subset of the possible values of specific type. VHDL provides a mechanism to specify a constraint in the declaration of an object. For example, declaration

```
signal  data12: integer range 0 to 4095;
```

specifies that signal data12 can take values of unsigned positive integer values 0 through 4095.

Similarly, VHDL provides **subtype** mechanism for creation of an alternate data type that is a constrained version of an existing type. For example, the declaration

```
subtype data16 integer range 0 to 2**16-1;
```

creates a scalar type with a limited range. The subtype data16 carries with it all operations available for the integer base type.

Basic data types have been already introduced in an informal way. They belong to scalar types represented by a single value, and are ordered in some way that relational operators can be applied to them. Table 10.3 lists the built-in scalar types defined in VHDL Standard 1076.

Table 10.3 Built-in scalar types that allow application of relational operators

| Data Type | Example of Values | Comment |
|---|---|---|
| Bit | '0', '1' | Defined as an enumerated type. |
| Character | '0', '1', 'a', 'b', '#', etc. | ISO 8859-1 character set. |
| Boolean | False, True | Defined as an enumerated type. |
| Integer | 1, -56, 128 | Minimum range of -2147483647 to +2147483647. |
| Real | 1.28, -23.5E10 | Minimum range of -1.0E38 to +1.0E38 |
| Severity_error | NOTE, ERROR | Enumerated type used in severity clause report. |
| Time | 10 ms | A physical type. |

### *10.4.1 Bit Type*

The bit type is used in VHDL to represent the most fundamental objects in a digital system. It has only two possible values, '0' and '1', that are usually used to represent logical 0 and 1 values in a digital system. The following example uses bit data type to describe the operation of a 2-to-4 decoder:

```
entity decoder2to4 is

   port (a: in bit;
         b: in bit;
         d0: out bit;
         d1: out bit;
         d2: out bit;
         d3: out bit)
end decoder2to4;

architecture concurrent of decoder2to4 is
begin
   d0 <= 1 when b = '0' and a = '0' else '0';
   d1 <= 1 when b = '0' and a = '1' else '0';
   d2 <= 1 when b = '1' and a = '0' else '0';
   d3 <= 1 when b = '1' and a = '1' else '0';
end concurrent;
```

The bit data type supports logical and relational operations. The IEEE 1164 specification, which is now commonly used, describes an alternative to bit called std_ulogic. Std_ulogic is defined as an enumerated type that has nine possible values, allowing a more accurate description of values and states of signals in a digital system. A more detailed presentation of IEEE 1164 standard logic specification is given in the later sections of this Chapter.

### *10.4.2 Character Type*

This type is similar to the character types in programming languages. Characters can be used to represent actual data values in design descriptions, but more often they are used to represent strings and to display messages during simulation. However, characters in VHDL are defined as enumerated type and have no explicit value, and, therefore, they cannot be simply mapped onto numeric data types. In order to do that, type conversion functions are required. The character type defined in the 1076-1987 package is:

```
type character is (

NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
BS, HT, LF, VT, FF, CR, SO, SI,
DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
CAN, EM, SUB, ESC, FSP, GSP, RSP, USP,

' ', '!', '"', '#', '$', '%', '&', ''',
'(', ')', '*', '+', ',', '-', '.', '/',
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', ':', ';', '<', '=', '>', '?',
```

```
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',

'x', 'y', 'z', '{', '|', '}', '~', DEL);
```

The IEEE 1076-1993 specification extends the character set to the 256-character ISO 8859 standard.

### *10.4.3 Boolean Type*

The Boolean type is defined as an enumerated type with two possible values, True and False. It is a result of a logical test which is using relational operators or can be the result of an explicit assignment.

### *10.4.4 Integer Type*

The predefined integer type includes all integer values in range of -2147483647 to +2147483647, inclusive. New integer constrained subtypes can be declared using **subtype** declaration. The predefined subtype natural restricts integers to the range of 0 to the specified (or default) upper range limit, and predefined subtype positive restricts integers to the range of 1 to the specified upper limit:

```
type integer is range -2147483647 to 2147483647;
subtype natural is integer range 0 to 2147483647;
subtype positive is integer range 1 to 2147483647;
```

IEEE Standard 1076.3 defines an alternative to the integer type defining signed and unsigned data types, which are array types that have properties of both array and numeric data types. They allow to perform shifting and masking operations like on arrays, and arithmetic operations, like on integers. These types are presented in more details in subsequent sections.

In order to illustrate the use of integer data type consider 2-to-4 decoder as given in Example 10.1.

Example 10.1 Use of integer type

```
entity decoder2to4 is
   port (x: in integer range 3 downto 0;
      d0: out bit;
      d1: out bit;
      d2: out bit;
      d3: out bit)
end entity decoder24;

architecture second of decoder2to4 is

begin
    d0 <= 1 when x=0;
    d1 <= 1 when x=1;
    d2 <= 1 when x=2;
    d3 <= 1 when x=3;
end architecture second;
```

In this example, the input port of the decoder is declared as constrained integer. The description of the decoder behavior is simplified, and the checks of the input values are left to the VHDL compiler.

### *10.4.5 Real Types*

Real types have little use due to the fact that synthesizers do not support this type. They are primarily used for simulation purposes allowing to declare objects of this type and assign them real values in the specified range of -1.0E38 to +1.0E38. The real type supports arithmetic operations.

### *10.4.6 Severity_Level Type*

Severity_Level type is a data type used only in the report section of an **assert** statement. It is an enumerated type with four possible values that can be assigned to the objects of this type: note, warning, error and failure. These values may be used to control simulation process to indicate simulator to undertake an action if certain specific conditions appear.

### *10.4.7 Time Type*

Time data type is built-in VHDL data type used to measure time. Time has units of measure which are all expressed as multiples of a base unit, femtosecond (fs). The definition for type time might be as follows:

```
type time is range of -2147483647 to +2147483647
 units
    fs;
    PS = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min =60 sec;
    hr = 60 min;
 end units;
```

## 10.5 Extended Types

As we have already seen, the VHDL language does not include many built-in types for signals and variables, but allows users to add new data types. The Package STANDARD, included in every implementation, extends the language to allow data types useful for description of hardware. These types include boolean, bit, bit_vector, character, string, and text. For example, declaration of bit type is:

```
type bit is ('0', '1');
```

It enumerates two possible values of type bit. However, in most environments, few more logical strengths, such as unknown, high impedance, weak 1, weak 0, are needed. Some vendors have up to 47 signal strengths.

To extend the available data types, VHDL provides a type-declaration capability and a package facility to save and deliver these new data types. VHDL also provides overloaded operators so that the use of new data types is natural and easy.

### *10.5.1 Enumerated Types*

As shown in preceding sections, enumerated types are used to describe many of the standard VHDL data types. They can be used to describe unique data types and make easier the description of a design. The enumerated type declaration lists a set of names or values defining a new type:

```
type identifier is   (enumeration_literal
          {,enumeration_literal});
```

where enumeration_literal can be identifier or character_literal. This allows us to declare a new type using character literals or identifiers. For example, using identifiers we can declare:

```
type colors is (black, white, red};
```

The example identifies three different values in a particular order that define type colors. In subsequent declarations of a variable or signal designated type colors, assigned values could only be black, white, and red.

In some applications it is convenient to represent codes symbolically by defining own data type. In the following example data type fourval is defined using literals:

```
type fourval is ('0', '1', 'z', 'x');
```

If we declare type fourval in design, then we can declare ports, signals, and variables of this type. Example 10.2 represents a simplified CPU, using enumerated type instruction_code to be the operation codes 1da, 1db, sta, stb, aba, and sba. The CPU has two working registers a and b which are used to store operands and results of operations.

Example 10.2 Use of enumerated type

```
architecture behavior of simple_cpu is
   type instruction_code is (aba, sba, sta, stb, lda,
                                          ldb);
begin
process

   variable a, b, data: integer;
   variable instruction: instruction_code;
   begin

    case instruction is
      when lda => a:= data;
      when ldb => b:= data;
      when sta => data:=a;
      when stb => data:=b;
      when aba => a:= a + b;
      when sba => a:= a - b;
    end case;
    wait on data;
   end process;
end behavior;
```

The only values that variable instruction can take on are enumerated values of the type instruction_code. Some extensions to VHDL allow to assign the numeric encodings, for example in the later stages of a top-down design.

Enumerated types provide through abstraction and information hiding a more abstract design style often referred to as object oriented. For example, they allow to observe symbolic type names during simulation process, or defer the actual encoding of the symbolic values until the time when the design is implemented in hardware.

### *10.5.2 Qualified Expressions*

If there is ambiguity in using the specific values in terms of its type, it is necessary to do typecasting to be explicit on the type of the value. The type is cast in VHDL by using a qualified expression. For example:

```
type name is (alpha, beta);
```

When a type has a shared value with the other types, the type can be clarified by using qualified expression with the following syntax:

```
type' (literal or expression)
```

for example

```
name' (alpha)
```

It is sometimes necessary to map one data type to another. A variable or signal can be converted by using conversion function. Assume that we have two types: incoming value of type threeval, and we want to convert it to outgoing value named value3. Example 10.3 shows a conversion function that maps one to another type.

Example 10.3 Conversion function

```
type threeval is ('l', 'h', 'z');
type value3 is ('0', 'l', 'z')
function convert (a: threeval) return value3 is
begin
    case a is
       when 'l' => return '0';
       when 'h' => return 'l';
       when 'z' => return 'z';
    end case
end convert;
```

An example of the call to a conversion function is given below:

```
process (inp);
 variable inp: threeval;
 variable outp: value3;
begin
 outp := convert(inp);
end process;
```

### *10.5.3 Physical Types*

Physical types are used to represent physical quantities such as time, distance, current, etc. A physical type provides for a base unit, and successive units are then derived from the base unit. The smallest unit representable is one base unit; the largest is determined by the range specified in the physical type declaration.

An example of user-defined physical type follows:

```
type voltage is range 0 to 20000000
units
   uv;        -- micro volts
   mv = 1000 uv;     -- milivolts
   v = 1000 mv;      -- volts
end units;
```

The type definition begins with a statement that declares the name of the type (voltage) and the range of the type in base units (0 to 20,000,000). The first unit declared is the base unit. After the base unit is defined, the other units can be defined in terms of the base unit or the other units already defined. The unit identifiers all must be unique within a single type.

## 10.6 Composite Types - Arrays

VHDL provides array as a composite type, containing many elements of the same type. These elements can be scalar or composite. They can be accessed by using an index. The only predefined array types in the Package STANDARD are bit_vector and string. New types have to be declared for real and integer arrays.

Access depends upon declaration. For example:

```
variable c: bit_vector (0 to 3);
variable d: bit_vector (3 downto 0);
```

In this example the indices for variable c are 0 for leftmost bit c(0) and 3 for the rightmost bit c(3); for variable d, 3 is the index for leftmost bit d(3), and 0 is the index for rightmost bit d(0). VHDL has no particular standard for the ordering of bits or the numbering scheme. One can number from 1 to 4, or 4 to 7, etc.

Examples of valid bit_vector assignments are given below:

```
c := "1100";
d := ('1', '0', '1', '0') ;
d : = a & b & f & g;
```

In the last case a, b, f, and g must be 4 1-bit single variables concatenated by ampersand (&).

VHDL allows an access to the slice of an array that defines its subset. For example:

```
variable c: bit_vector (3 downto 0);
variable d: bit_vector (7 downto 0);

d(7 downto 4)  := c;
```

Four bits of c are assigned to upper four bits of d. Any subrange or slice must declare subscripts in the same direction as initially declared.

### *10.6.1 Aggregates*

An array reference can contain a list of elements with both positional and named notation, forming a typed aggregate. The syntax is:

```
type_name' ([choice=>] expression
          {, [others =>] expression})
```

where type_name can be any constrained array type. The optional choice specifies an element index, a sequence of indices, or [**others=>**]. Each expression provides values for the chosen elements and must evaluate to a value of the element's type. An element's index can be specified by using positional or named notation. Using positional notation, each element is given the value of its expression:

```
variable x: bit_vector (1 to 4) ;
variable a, b: bit;

x := bit_vector' ('1', a and b, '1', '0');
x := (1 => '1', 3 => '1', 4 => 'O', 2 => a and b);
```

An aggregate can use both positional and named notation, but positional expressions must come before any named [**choice** =>] expressions. If some values are not specified they are given a value by including [**others** =>] expression as the last element of the list. An example is given below:

```
variable b: bit;
variable c: bit_vector (8 downto 1)

c := bit_vector' ('1', '0', b, others => '0');
```

Eight bits on the right side come from various sources. The symbol => is read as "gets".

### *10.6.2 Array Type Declaration*

The syntax used to declare a new type that is an array is:

```
type array_name is array
          [index_constraint] of element_type
```

where index_constraint is:

```
 [range_spec]
index_type range [range_spec]
index_type range <>
```

Examples of array type declarations are:

```
type byte is array (0 to 7) of bit;
type ram is array (0 to 7, 0 to 255) of bit;
```

After a new type is declared, it can be used for signal or variable declaration:

```
variable word: byte;
```

An enumerated type or subtype also can be used to designate the range of subscript values:

```
type instruction is (aba, sba, lda, ldb, sta, stb);
subtype arithmetic is instruction range aba to sba;
subtype digit is integer range 1 to 9;
type ten_bit is array (digit) of bit;
type inst_flag is array (instruction) of digit;
```

Hardware systems frequently contain arrays of registers or memories. Two-dimensional arrays can be useful for simulating RAMs and ROMs. VHDL allows multiple-dimensional arrays. A new array type must be declared before we declare own variable or signal array as illustrated in Example 10.4.

Example 10.4 Use of arrays

```
type memory is array (0 to 7, 0 to 3) of bit;
constant rom: memory :=( ('0', '0', '1', '0'),
        ('1', '1', '0','1'),
        ('0', '0', '1','0'),
        ('1', '1', '1','1'),
        ('0', '0','1', '1'),
        ('0', '1', '1', '0'),
        ('1', '0', '1', '0'),
        ('1', '0', '1', '1'));
cont := rom(2, 1);
```

Multiple-dimensional arrays are not generally supported in synthesis tools, but can be useful for simulation purposes for describing test stimuli, memory elements, or other data that require tabular form. VHDL also allows declaration of array of arrays. Always array must be declared before a variable or signal of that type are declared.

Sometimes it is convenient to declare a new type (subtype) of an existing array type. For example:

```
subtype byte is bit_vector (7 downto 0) ;
```

Variable and signal declarations can now use the subtype:

**variable** alpha: byte;

## 10.7 Records and Aliases

Records group objects of different types into a single object. These elements can be of scalar or composite types and are accessed by name. They are referred to as fields. Each field of record can be referenced by name. The period "." is used to separate record names and record element names when referencing record elements. Example 10.5 of record declaration and use of record is shown below.

Example 10.5 Use of record

```
type two_digit is
   record sign: bit;
      msd:  integer range 0 to 9;
      lsd:  integer range 0 to 9;
   end record;

process;
   variable cntr1, cntr2: two_digit;
begin
   cntr1.sign := '1';
   cntr1.msd := 1;
   cntr1.lsd := cntr1.msd;
   cntr2 := two_digit' ('0', 3, 6);
end process;
```

Records are not generally synthesizable, but they can be useful when describing test stimuli for simulation purposes.

An alias is an alternate name assigned to part of an object, which allows simple access. For example, a 9-bit bus count has three elements: a sign, the msd, and lsd. Each named element can be operated using an alias declaration:

```
   signal count: bit_vector (1 to 9);
   alias sign: bit is count (1);
   alias msd: bit_vector (1 to 4) is count (2 to 5);
   alias lsd: bit_vector (1 to 4) is count (6 to 9);
```

The examples of accesses are:

```
   sign := '0';
   msd := "1011"
   count := "0_1110_0011"
```

## 10.8 Symbolic Attributes

VHDL has symbolic attributes that allow a designer to write more generalized code. Some of these attributes are predefined in VHDL, others are provided by CAD vendors. The designer can also define his own attributes. Attributes are related to arrays, types, ranges, position, and signal characteristics.

The following attributes work with arrays and types:

- aname'left returns left bound of index range
- aname'right returns right bound of index range

- aname'high returns upper bound of index range
- aname'low returns lower bound of index range
- aname'length returns the length (number of elements) of an array
- aname'ascending (VHDL '93) returns a Boolean true value of the type or subtype if it is declared with an ascending range

where character " ' " designates a separator between the name and the attribute. If the numbers that designate the lower and upper bounds of an array or type change, no change in the code that uses attributes. Only declaration portion should be changed. In the multirange arrays attributes are specified with the number of the range in the parentheses. For example, for array:

```
variable: memory  (0 to 10) of bit;

memory'right
```

will give value 10 because the second range has the index equal to 10.

Similarly array length attribute returns the length of an array:

```
a := memory'length ;
```

and a has a value of 11. The length of an array can be specified symbolically rather than with a numeric value.

Example 10.6 illustrates the use of function array attributes implementing an integer-based RAM device with 1024 integer locations and two control lines.

Example 10.6 Using function attributes in modeling RAM device

```
package package_ram is
    type t_ram_data is array (0 to 1023) of integer;
    constant x_val: integer := -1;
    constant z_val: integer := -2;
end package_ram;

use work.package_ram.all;
use work.std_logic_1164.all;

entity ram_1024 is
   port (data_in, addr: in integer;
```

```
      data_out: out integer;
      cs, r_w: in std_logic);
end ram_1024;

architecture ram of ram_1024 is

begin
process (cs, addr, r_w)
    variable ram_data: t_ram_data;
    variable ram_init: boolean := false;
    begin
       if not(ram_init) then
        for i in ram_data'low to ram_data'high loop
          ram_data(i) := 0;
        end loop;
         ram_init := true;
       end if;
       if (cs = 'x') or (r_w = 'x')then
         data_out <= x_val;
       elsif (cs = '0') then
         data_out <= z_val;
       elsif (r_w = '1') then
          if (addr=x_val) or (addr=z_val)then
           data_out <= x_val;
          else
           data_out <= ram_data(addr) ;
          end if;

       else
          if (addr=x_val) or (addr=z_val)then
           assert false
          report "writing to unknown address"
           severity error;
           data_out <= x_val;
          else
           ram_data(addr) := data_in;
           data_out <= ram_data(addr);
          end if;
       end if;
    end process;
end ram;
```

This model contains an IF statement that initializes the contents of the RAM to a known value. A Boolean variable ram_init keeps track of whether the RAM has been initialized or not. The first time the process is executed, variable ram_init will be false, and if statement will be executed, and the locations of the RAM initialized to the value 0. Setting the variable ram_init to true will prevent the initialization

loop from executing again. The rest of the model implements the read and write functions based on the values of addr, data_in, r_w, and cs.

The range attribute returns the range of an object. The name'range and name'reverse_range are used to return the range of particular type in normal or reverse order. The best use of these attributes is when we actually do not know the length of an array, and varying sizes are provided.

Another use of symbolic attributes is for enumerated types. Enumerated type has the notion of successor and predecessor, left and right of the position number of the value:

- typename'succ (v) returns next value in type after v
- typename'pred (v) returns previous value in type before v
- typename'leftof (v) returns value immediately to left of v
- typename'rightof (v) returns value immediatelly to right of v
- typename'pos (v) returns type position number of v
- typename'val (p) returns type value from position value p
- typename'base returns base type of type or subtype.

Example below explains the usage of symbolic attributes in enumerated types:

```
type color is (red, black, blue, green, yellow);
subtype color_ours is color range black to green;
variable a: color;

a:= color'low
a:= color'succ (red);
a:= color_ours'base'right;
a:= color_ours'base'succ (blue);
```

Assignment statements assign to variable a following values:

- red
- black
- yellow
- green

respectively.

Signal attributes work on signals, they provide information about simulation time events:

- signalname'event returns true if an event occured this time step
- signalname'active returns true if a transaction occured this time step
- signalname'last_event returns the elapsed time since the previous event transaction
- signalname last_value returns previous value of signal before the last event transition
- signalname'last_active returns time elapsed since the previous transaction occurred

Signal attributes allow designer to do some complicated tests as shown in Example 10.7.

Example 10.7 Using signal attributes

```
entity dff is
   port (d, clk: in std_logic;
       q: out std_logic);
end dff;

architecture dff_1 of dff is

begin
process (clk)
   begin
       if (clk = '1') and (clk'event)
          and (clk'last_value = '0') then q <= d;
       end if;
    end process;
end dff_1;
```

The process tests if clk is '1' and clk'event, which means the clock is changed to '1'. If the last previous value of clock is zero, then we have a true rising edge.

It should be noted that all event oriented attributes, except 'event, are not generally supported in synthesis tools.

Another group of signal attributes create special signals that have values and types based on other signals. These special signals can be used anywhere in the design description where a normally declared signal could be used. Special kind signal attributes are:

- aname'delayed(time), that creates a delayed signal that is identical in the waveform to the signal the attribute is applied to.
- aname'stable(time), that creates a signal of type Boolean that becomes true when the signal is stable (has no events) for a given period of time
- aname'quiet(time), that creates a signal of type Boolean that becomes true when the signal has no transactions (scheduled events) for a given period of time.
- aname'transaction, that creates a signal of type bit that toggles its value whenever a transaction or actual event occurs on the signal the attribute is applied to.

There are two additional attributes that return value and can be used to determine information about blocks or attributes in a design. The 'structure attribute returns true if there are references to the lower-level components, and false if there are no references to lower-level components. The 'behavior attribute returns false if there are references to lower-level components; otherwise it returns true. The prefix to both these attributes must be an architecture name. VHDL 1076-1993 adds three new attributes that can be used to determine the configuration of entities in a design description. For more information about these attributes, refer to the IEEE VHDL Language Reference Manual.

## 10.9 Standard Logic

After 1076 standard, two other IEEE standards, 1164 and 1076.3, were introduced adding important capabilities for both simulation and synthesis.

### *10.9.1 IEEE Standard 1164*

One of the serious limitations of the first release of VHDL was the lack of the ability to provide multiple values (for example high-impedance, unknown, etc.) to be represented for a wire. These metalogic values are important for accurate simulation. To solve this problem, simulation vendors invented their own proprietary data types using enumerated types. Those proprietary data types were having four, seven or even thirteen unique values. IEEE 1164 is a standard logic data type with nine values as shown in Table 10.4.

Table 10.4 IEEE 1164 standard logic data type

| Value | Description |
|---|---|
| 'U' | Uninitialized |
| 'X' | Unknown |
| '0' | Logic 0 (driven) |
| '1' | Logic 1 (driven) |
| 'Z' | High impedance |
| 'W' | Weak 1 |
| 'L' | Logic 0 (read) |
| 'H' | Logic 1 (read) |
| '-' | Don't-care |

Having these nine values, it becomes possible to accurately model the behavior of a digital system during simulation. However, the standard is also valuable for synthesis purposes because it enables modeling of circuits that involve output enables, as well as to specify don't care logic that is used to optimize the combinational logic.

There are many situations in which it becomes useful to use IEEE 1164 standard logic. For example, if we want to observe during simulation behavior of the system when we apply to the inputs other values than '0' and '1', or if we want to check what happens when the input with an unknown or don't care value is applied. The resolved standard logic data types can be used to model the behavior of multiple drivers in a circuit. The resolved types and resolution functions are beyond the scope of this book.

However, the most important reason to use standard logic data types is to provide portability between models written by different designers, or when moving models and designs between different simulation and synthesis environments.

Two statements are added to the beginning of source VHDL files to describe that standard logic types will be used. Those two statements are found in the most of our previous examples:

```
library ieee;
use ieee.std_logic_1164.all;
```

If the source file contains several design units, the use clause has to be placed prior to each design unit. The exception is architecture declaration. If the corresponding entity declaration includes a use statement, then the use statement

need not to be used before architecture declaration. These two statements are used to load the IEEE 1164 standard library and its contents (the std_logic_1164 package).

### *10.9.2 Standard Logic Data Types*

The std_logic_1164 package provides two fundamental data types, std_logic and std_ulogic. These two data types are enumerated types defined with nine symbolic values. The std_ulogic type is defined in the IEEE 1164 standard as:

```
type std_ulogic is ('U', -- Uninitialized
        'X', -- Forcing Unknown
        '0', -- Forcing 0
        '1', -- Forcing 1
        'Z', -- High Impedance
        'W', -- Weak Unknown
        'L', -- Weak 0
        'H', -- Weak 1
        '-', -- Don't care
            );
```

The std_ulogic data type is an unresolved type. It does not allow for two values to be simultaneously driven onto a signal of type std_ulogic. If two or more values can be driven onto a wire, another type, called std_logic, has to be used. The std_logic data type is a resolved type based on std_ulogic and has the following definition:

```
subtype std_logic is resolved std_ulogic:
```

Resolved types are declared with resolution functions, which define behavior when an object is driven with multiple values simultaneously. In the case of multiple drivers, the nine values of std_logic are resolved to values as indicated in Table 10.5.

Table 10.5 Resolution of std_logic data type

| | 'U' | 'X' | '0' | '1' | 'Z' | 'W' | 'L' | 'H' | '-' |
|---|---|---|---|---|---|---|---|---|---|
| 'U' | 'U' | 'U' | 'U' | 'U' | 'U' | 'U' | 'U' | 'U' | 'U' |
| 'X' | 'U' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' |
| '0' | 'U' | 'X' | '0' | 'X' | '0' | '0' | '0' | '0' | 'X' |
| '1' | 'U' | 'X' | 'X' | '1' | '1' | '1' | '1' | '1' | 'X' |
| 'Z' | 'U' | 'X' | '0' | '1' | 'Z' | 'W' | 'L' | 'H' | 'X' |
| 'W' | 'U' | 'X' | '0' | '1' | 'W' | 'W' | 'W' | 'W' | 'X' |
| 'L' | 'U' | 'X' | '0' | '1' | 'L' | 'W' | 'L' | 'W' | 'X' |
| 'H' | 'U' | 'X' | '0' | '1' | 'H' | 'W' | 'W' | 'H' | 'X' |
| '-' | 'U' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' | 'X' |

Both these standard logic types may be used as one-to-one replacement for the built-in type bit. Example 10.8 shows how std_logic type may be used to describe a simple 2-to-4 decoder coupled to an output enable.

Example 10.8 Using std_logic type

```
library ieee;
use ieee.std_logic_1164.all;

entity decoder is
   port (a, b, oe: in std_logic;
      y0, y1, y2, y3: out std_logic);
end entity decoder;

architecture arch1 of decoder is
   signal s0, s1, s2, s3: std_logic;
begin
    s0 <= not(a) and not(b);
    s1 <= a and not(b);
    s2 <= not(a) and b;
    s3 <= a and b;
    y0 <= s0 when oe='0' else 'Z';
    y1 <= s1 when oe='0' else 'Z';
    y2 <= s2 when oe-'0' else 'z';
    y3 <= s3 when oe-'0' else 'Z' ;
end architecture arch1;
```

In addition to the single-bit data types std_logic and std_ulogic, IEEE standard 1164 includes array types corresponding to each of these types. The std_logic_vector and std_ulogic_vector are defined in the std_logic_1164 package as unbounded arrays similar to the built-in type bit_vector with the following definitions:

```
type std_ulogic_vector is array
        (natural range <>) of std_ulogic;
type std_logic_vector is array
        (natural range <>) of std_logic;
```

In actual models or designs, the user will use an explicit width or will use a subtype to create a new data type on std_logic_vector or std_ulogic_vector with the required width. Example 10.9 shows the use of a new subtype (defined in an external package) to create a 16-bit array based on std_logic_vector.

Example 10.9 Using std_logic_vector

```
library ieee;
use ieee.std_logic_1164.all;

package new_type is
    subtype word is std_logic_vector(15 downto 0);
end package new_type;

use ieee.std_logic_1164.all;

entity word_xor is
   port(a_in, b_in: in word; oe: in std_logic;
      c_out: out word)
end entity word_xor;

architecture arch1 of word_xor is
   signal int: word;

begin
   int <= a_in xor b_in;
   c_out <= int when oe='0' else 'ZZZZ_ZZZZ_ZZZZ_ZZZZ';

end architecture arch1;
```

In this example a new subtype word is defined as 16-element array of std_logic_vector. The width of the word_xor circuit is defined in the package new_type, and easily can be modified. There is no need to modify the rest of description of the circuit.

If the designer needs to simplify operations on standard logic data, for example to use 3-, 4-, or 5-valued logic, the std_logic_1164 package contains the following subtypes:

```
subtype X01 is resolved std_ulogic
          range 'X' to '1'; --('X','0','1')
subtype X01Z is resolved std_ulogic
```

```
            range 'X' to 'z'; --('X','0','1','Z')
subtype UX01 is resolved std_ulogic
            range 'U' to '1'; --('U','X','0','1')
subtype UX01Z is resolved std_ulogic
            range 'U' to 'Z'; --('U','X','0','1','Z')
```

### *10.9.3 Standard Logic Operators and Functions*

Standard logic data types are supported by a number of operators defined as:

```
function "and" (l: std_ulogic; r: std_ulogic)
            return UX01;
function "nand" (l: std_ulogic; r: std_ulogic)
            return UX01;
function "or" (l: std_ulogic; r: std_ulogic)
            return UX01;
function "nor" (l: std_ulogic; r: std_ulogic)
            return UX01;
function "xor" (l: std_ulogic; r: std_ulogic)
            return UX01;
   function "xnor" (l: std_ulogic; r: std_ulogic)
            return UX01; -- only -- standard 1076-1993
function "not" (l: std_ulogic) return UX01;
function "and" (l. r: std_logic_vector)
            return std_logic_vector;
function "and" (l. r: std_ulogic_vector)
            return std_ulogic_vector;
function "nand" (l. r: std_logic_vector)
            return std_logic_vector;
function "nand" (l. r: std_ulogic_vector)
            return std_ulogic_vector;
function "or" (l. r: std_logic_vector)
            return std_logic_vector;
function "or" (l. r: std_ulogic_vector)
            return std_ulogic_vector;
function "nor" (l. r: std_logic_vector)
            return std_logic_vector;
function "nor" (l. r: std_ulogic_vector)
            return std_ulogic_vector;
function "xor" (l. r: std_logic_vector)
            return std_logic_vector;
function "xor" (l. r: std_ulogic_vector)
            return std_ulogic_vector;
function "xnor" (l. r: std_logic_vector)
            return std_logic_vector; -- only 1076-1993
function "xnor" (l. r: std_ulogic_vector)
            return std_ulogic_vector; -- only 1076-1993
```

```
function "not" (l. r: std_logic_vector)
          return std_logic_vector;
function "not" (l. r: std_ulogic_vector)
          return  std_ulogic_vector;
```

The strength stripping functions convert the 9-valued types std_ulogic and std_logic to the 3-, 4-, and 5-valued types, converting strength values 'H', 'L', and 'W' to their '0' and '1' equivalents:

```
function To_X01 (s: std_logic_vector)
          return  std_logic_vector;
function To_X01 (s: std_ulogic_vector)
          return std_ulogic_vector;
function To_X01 (s: std_ulogic) return X01;
function To_X01 (b: bit_vector)
          return std_logic_vector;
function To_X01 (b: bit_vector)
          return  std_ulogic_vector;
function To_X01 (b: bit) return X01;
function To_X01Z (s: std_logic_vector)
          return std_logic_vector;
function To_X01Z (s: std_ulogic_vector)
          return  std_ulogic_vector;
function To_X01Z (s: std_ulogic)
          return X01Z;
function To_X01Z (b: bit_vector)
          return std_logic_vector;
function To_X01Z (b: bit_vector)
          return std_ulogic_vector;
function To_X01Z (b: bit) return X01Z;
function To_UX01 (s: std_logic_vector)
          return std_logic_vector;
function To_UX01 (s: std_ulogic_vector)
          return std_ulogic_vector;
function To_UX01 (s: std_ulogic) return UX01;
function To_UX01 (b: bit_vector)
          return std_logic_vector;
function To_UX01 (b: bit_vector)
          return std_ulogic_vector;
function To_UX01 (b: bit) return UX01;
```

The edge detection functions rising_edge() and falling_edge() provide a concise way to describe the behavior of an edge-triggered device such as a flip-flop:

```
function rising_edge(signal s: std_ulogic)
          return boolean;
function falling_edge(signal s: std_ulogic)
          return boolean;
```

The following functions can be used to determine if an object or literal is don't-care, which is in this case defined as any of the five values 'U', 'X', 'Z', 'W' or '-':

```
function is_X (s: std_ulogic_vector) return boolean;
function is_X (s: std_logic_vector) return boolean;
function is_X (s: std_ulogic) return boolean;
```

### *10.9.4 IEEE Standard 1076.3 (The Numeric Standard)*

IEEE Standard 1076.3 provides numeric data types and operations to help synthesis and modeling. It defines the numeric_std package that allows the use of arithmetic operations on standard logic data types. The numeric_std package defines the numeric types signed and unsigned and corresponding arithmetic operations and functions based on std_logic data type. Two numeric types declared in numeric_std package, unsigned and signed, are defined as follows:

```
type unsigned is array (natural range <>) of std_logic;
type signed is array (natural range <>) of std_logic;
```

Unsigned represents unsigned integer data in the form of an array of std_logic elements. Signed represents signed integer data in two's complement form. The leftmost bit is treated in both these types as the most significant bit.

Example 10.10 illustrates how the type unsigned may be used to simplify the description of a 16-bit up-down counter.

#### Example 10.10 Using unsigned type

```
library ieee;
use ieee.std_logic_1164.all;
use numeric_std.all;

entity counter is
   port (clk, load, clr, up, down: in std_logic;
       data: in std_logic_vector(3 downto 0) ;
       count: out std_logic_vector(3 downto 0)
   );
end entity counter;

architecture count4 of counter is
   signal cnt: unsigned(3 downto 0);

begin
```

```
    process(clr, clk)
    begin
        if clr='1' then      -- asynchronous clear
          cnt <= '0000';
        elsif clk'event and clk='1'then
          if load='1' then
              cnt <= unsigned(data); -- type conversion
          elsif up='1' then
              if cnt='1111' then
                    cnt <='0000';
              else
                    cnt <= cnt+1;
              end if;
          elsif down='1' then
              if cnt='0000' then
                    cnt <='1111';
              else
                    cnt <= cnt-1;
              end if;
           else
                    cnt <= cnt;
          end if;
         end if;

         count<= std_logic_vector(cnt); -- type conversion

     end process;
end architecture count4;
```

The type unsigned is used in this example within the architecture to represent the current state of the counter. IEEE 1076.3 standard describes the add operation ('+') and subtract operation ('-') for type unsigned, so the counter can be easily described. Conversion between unsigned and std_logic_vector is straightforward because these two types are based on the same element type std_logic.

### *10.9.5 Numeric Standard Operators and Functions*

#### Arithmetic Operators

```
function "abs" (ARG: signed) return signed;
function "-" (ARG: signed) return signed;
function "+" (L, R: unsigned) return unsigned;
function "+" (L, R: signed) return signed;
function "+" (L: unsigned; R: natural) return unsigned;
function "+" (L: natural; R: unsigned) return unsigned;
function "+" (L: integer; R: signed) return signed;
```

```
function "+" (L: signed; R: integer) return signed;
function "-" (L, R: unsigned) return unsigned;
function "-" (L, R: signed) return signed;
function "-" (L: unsigned; R: natural) return unsigned;
function "-" (L: natural; R: unsigned) return unsigned;
function "-" (L: integer; R: signed) return signed;
function "-" (L: signed; R: integer) return signed;
function "*" (L, R: unsigned) return unsigned;
function "*" (L, R: signed) return signed;
function "*" (L: unsigned; R: natural) return unsigned;
function "*" (L: natural; R: unsigned) return unsigned;
function "*" (L: integer; R: signed) return signed;
function "*" (L: signed; R: integer) return signed;
function "/" (L, R: unsigned) return unsigned;
function "/" (L, R: signed) return signed;
function "/" (L: unsigned; R: natural) return unsigned;
function "/" (L: natural; R: unsigned) return unsigned;
function "/" (L: integer; R: signed) return signed;
function "/" (L: signed; R: integer) return signed;
function "rem" (L, R: unsigned) return unsigned;
function "rem" (L, R: signed) return signed;
function "rem" (L: unsigned; R: natural)
         return unsigned;
function "rem" (L: natural; R: unsigned)
         return unsigned;
function "rem" (L: integer; R: signed) return signed;
function "rem" (L: signed; R: integer) return signed;
function "mod" (L, R: unsigned) return unsigned;
function "mod" (L, R: signed) return signed;
function "mod" (L: unsigned; R: natural)
         return unsigned;
function "mod" (L: natural; R: unsigned)
         return unsigned;
function "mod" (L: integer; R: signed) return signed;
function "mod" (L: signed; R: integer) return signed;
```

## Numeric Logical Operators

```
function "not" (L:unsigned) return unsigned;
function "and" (L, R: unsigned) return unsigned;
function "or" (L, R: unsigned) return unsigned;
function "nand" (L, R: unsigned) return unsigned;
function "nor" (L, R: unsigned) return unsigned;
function "xor" (L, R: unsigned) return unsigned;
function "xnor" (L, R: unsigned)
         return unsigned; -- 1076-1993 only
function "not" (L:signed) return signed;
```

```
function "and" (L, R: signed) return signed;
function "or" (L, R: signed) return signed;
function "nand" (L, R: signed) return signed;
function "nor" (L, R: signed) return signed;
function "xor" (L, R: signed) return signed;
function "xnor" (L, R: signed)
         return signed; -- 1076-1993 only
```

**Relational Operators**

```
function ">" (L, R: unsigned) return boolean;
function ">" (L, R: signed) return boolean;
function ">" (L: natural; R: unsigned) return boolean;
function ">" (L: integer; R: signed) return boolean;
function ">" (L: unsigned; R: natural) return boolean;
function ">" (L: signed; R: integer) return boolean;
function "<" (L, R: unsigned) return boolean;
function "<" (L, R: signed) return boolean;
function "<" (L: natural; R: unsigned) return boolean;
function "<" (L: integer; R: signed) return boolean;
function "<" (L: unsigned; R: natural) return boolean;
function "<" (L: signed; R: integer) return boolean;
function "<=" (L, R: unsigned) return boolean;
function "<=" (L, R: signed) return boolean;
function "<=" (L: natural; R: unsigned) return boolean;
function "<=" (L: integer; R: signed) return boolean;
function "<=" (L: unsigned; R: natural) return boolean;
function "<=" (L: signed; R: integer) return boolean;
function ">=" (L, R: unsigned) return boolean;
function ">=" (L, R: signed) return boolean;
function ">=" (L: natural; R: unsigned) return boolean;
function ">=" (L: integer; R: signed) return boolean;
function ">=" (L: unsigned; R: natural) return boolean;
function ">=" (L: signed; R: integer) return boolean;
function "=" (L, R: unsigned) return boolean;
function "=" (L, R: signed) return boolean;
function "=" (L: natural; R: unsigned) return boolean;
function "=" (L: integer; R: signed) return boolean;
function "=" (L: unsigned; R: natural) return boolean;
function "=" (L: signed; R: integer) return boolean;
function "/=" (L, R: unsigned) return boolean;
function "/=" (L, R: signed) return boolean;
function "/=" (L: natural; R: unsigned) return boolean;
function "/=" (L: integer; R: signed) return boolean;
function "/=" (L: unsigned; R: natural) return boolean;
function "/=" (L: signed; R: integer) return boolean;
```

**Shift and Rotate Functions**

```
function shift_left (ARG: unsigned, COUNT: natural)
          return unsigned;
function shift_right (ARG: unsigned, COUNT: natural)
          return unsigned;
function shift_left (ARG: signed, COUNT: natural)
          return signed;
function shift_right (ARG: signed, COUNT: natural)
          return signed;
function rotate_left (ARG: unsigned, COUNT: natural)
          return unsigned;
function rotate_right (ARG: unsigned, COUNT: natural)
          return unsigned;
function rotate_left (ARG: signed, COUNT: natural)
          return signed;
function rotate_right (ARG: signed, COUNT: natural)
          return signed;
```

The following shift and rotate operators are only supported in IEEE 1076-1993:

```
function "sll" (ARG: unsigned, COUNT: natural)
          return unsigned;
function "srl" (ARG: unsigned, COUNT: natural)
          return unsigned;
function "sll" (ARG: signed, COUNT: natural)
          return signed;
function "srl" (ARG: signed, COUNT: natural)
          return signed;
function "rol" (ARG: unsigned, COUNT: natural)
          return unsigned;
function "ror" (ARG: unsigned, COUNT: natural)
          return unsigned;
function "rol" (ARG: signed, COUNT: natural)
          return signed;
function "ror" (ARG: signed, COUNT: natural)
          return signed;
```

## 10.10 Type Conversions

As VHDL is a strongly typed language, it does not allow us to assign a literal value or object of one type to an object of another type. If transfers of data between objects of different types are needed, VHDL requires to use type conversion

features, for the types that are closely related, or to write conversion functions for types that are not closely related.

Explicit type conversions are allowed between closely related types. Two types are said to be closely related when they are either abstract numeric types (integers or floating point), or if they are array types of the same dimensions and share the same types for all elements in the array. If two subtypes share the same base type, then no explicit type conversion is required.

To convert data from one type to an unrelated type (for example from integer to an array type), conversion functions must be used. Type conversion functions are often found in standard libraries and vendor supplied libraries, but the designer can also write his/her own type conversion functions.

A type conversion function is a function that accepts an argument of a specified type and returns the equivalent value in another type. Two conversion functions needed to convert between integer and std_ulogic_vector types are presented in Example 10.11.

Example 10.11 Using conversion functions

```
-- Convert an integer to std_logic_vector
function int_to_std_ulogic_vector( size: integer; value:
            integer)
   return std_ulogic_vector is
   variable vector: std_ulogic_vector (1 to size);
   variable q: integer;

begin
   q:= value;
   for i in size downto 1 loop
       if((q mod 2)=1) then
         vector(i)='1';
       else
         vector(i)='0';
       end if;
         q:=q/2;
   end loop;
   return vector;
end int_to_std_ulogic_vector;
-- Convert a std_ulogic_vector to an unsigned integer

function std_ulogic_vector_to_uint (q: std_ulogic_vector)
   return integer is
   alias av: std_ulogic_vector (1 to a'length) is a;
```

```
    variable value: integer:= 0;
    variable b: integer:= 1;
begin
    for i in a'length downto 1 loop
        if (av(I) = '1') then
          value:= value+b;
        end if;
          b:= b*2;
    end loop;
    return value;
end std_ulogic_vector_to_uint;
```

Some type conversion functions are provided in IEEE std_logic_1164 package. They help to convert data between 1076 standard data types (bit and bit_vector) and IEEE 1164 standard logic data types:

```
    function To_bit (s: std_ulogic; xmap: bit:= '0')
             return bit;
    function To_bitvector (s: std_logic_vector;
             xmap: bit:= '0') return bit_vector;
    function To_bitvector (s: std_ulogic_vector;
             xmap: bit:= '0') return bit_vector;
    function To_StdUlogic (b: bit) return std_ulogic;
    function To_StdLogicVector (b: bit_vector)
             return std_logic_vector;
    function To_StdLogicVector (s: std_ulogic_vector)
             return std_logic_vector;
    function To_StdULogicVector (b: bit_vector)
             return std_ulogic_vector;
    function To_StdULogicVector (s: std_logic_vector)
             return std_ulogic_vector;
```

Other conversion functions found in IEEE std_logic_1164 package are used to convert between integer data types and signed and unsigned data types:

```
    function to_integer (ARG: unsigned) return natural;
    function to_integer (ARG: signed) return natural;
    function to_unsigned (ARG, SIZE: unsigned)
             return unsigned;
    function to_integer (ARG, SIZE: natural) return signed;
```

The matching functions (std_match) are used to determine if two values of type std_logic are logically equivalent, taking into consideration the semantic values of the 'X' (uninitialized) and '-' (don't-care) literal values:

```
    function std_match (L, R: std_ulogic) return boolean;
    function std_match (L, R: unsigned) return boolean;
```

```
function std_match (L, R: signed) return boolean;
function std_match (L, R: std_logic_vector)
         return boolean;
function std_match (L, R: std_ulogic_vector)
         return boolean;
```

However, they do not convert between standard logic data types and numeric data types such as integers or unsigned and signed types. Conversion between these types is usually provided by vendors of design tools, or the designer must provide their own conversion functions.

Table 10.6 defines the matching of all possible combinations of the std_logic values.

Table 10.6 Matching std_logic values

| | 'U' | 'X' | '0' | '1' | 'Z' | 'W' | 'L' | 'H' | '-' |
|---|---|---|---|---|---|---|---|---|---|
| 'U' | F | F | F | F | F | F | F | F | T |
| 'X' | F | F | F | F | F | F | F | F | T |
| '0' | F | F | T | F | F | F | T | F | T |
| '1' | F | F | F | T | F | F | F | T | T |
| 'Z' | F | F | F | F | F | F | F | F | T |
| 'W' | F | F | F | F | F | F | F | F | T |
| 'L' | F | F | T | F | F | F | T | F | T |
| 'H' | F | F | F | T | F | F | F | T | T |
| '-' | F | T | T | T | T | T | T | T | T |

## 10.11 Process Statement and Processes

The process statement defines the scope of each process. It determines the part of an architecture, where sequential statements are executed (components are not permitted in a process). The process statement provides a behavioral style description of design. The syntax is:

```
 [process_label :]
process  [(sensitivity-list)]
 subprogram_declaration or subprogram_body
 type_declaration
 subtype_declaration
 constant_declaration
 variable_declaration
 file_declaration
 alias_declaration
```

```
    attribute_declaration
    attribute_specification
    use_clause
  begin
    sequential_statements
  end process   [process_label];
```

The process statement can have an explicit sensitivity list. This list defines the signals that will cause the statements inside the process statement to execute whenever one or more elements of the list change value. Changes in these values, sometimes called the events, will cause the process to be invoked. The process has either sensitivity list or a wait statement, as we will see later. Sequential statements within process or subprogram body are logical, arithmetic, procedure calls, case statements, if statements, loops, and variable assignments.

Processes are usually used to describe the behavior of circuits that respond to external events. These circuits may be combinational or sequential, and are connected with other circuits via signals to form more complex systems. In a typical circuit specification, a process will include in its sensitivity list all inputs that have asynchronous behavior (such as clocks, reset signals, functional inputs to a circuit, etc.).

A process statement in an architecture is shown in Example 10.12 The circuit counts the number of bits with the value 1 in 3-bit input signal inp_sig.

Example 10.12 Use of process to describe behavior

```
entity bit_count is
   port ( inp_sig: in bit_vector (2 downto 0);
      q: out integer range 0 to 3);
end entity bit_count;

architecture count of bit_count is

begin
   process (inp_sig)
      variable n: integer;
   begin
      n := 0;
    for i in inp_sig'range loop
      if inp_sig(i) = '1' then
         n := n + 1;
      end if;
     end loop;
        q <= n;
   end process;
```

```
end architecture count;
```

The entity declares 3-bit input ports for the circuit that form an inp_sig array and one 2-bit output port q. The architecture contains only one statement, a concurrent process statement. The process declaration section declares one local variable called n. The process is sensitive to the signal inp_sig. Whenever the value of any bit in input signal changes, the statements inside the process will be executed. The variable n is assigned to the signal q. After all statements have been executed once, the process will wait for another change in a signal or port in its sensitivity list.

## 10.12 Sequential Statements

VHDL contains a number of facilities for modifying the state of objects and controlling the flow of execution of models. These facilities are introduced in the following sections.

### *10.12.1 Variable Assignment Statement*

A variable assignment statement replaces the current value of a variable with a new value specified by an expression. The syntax is:

```
target  :=expression;
```

In the simplest case, the target of an assignment is a variable name, and the value of the expression is given to the named variable. The variable on the left side of the assignment statement must be previously declared. The right side is an expression using variables, signals, and literals. The variable and the value must have the same base type. This statement executes in zero simulation time. Variable assignment happens immediately when the statement is executed. Examples of variable assignment statements are:

```
a := 2.0;
c := a + b;
```

It is important to remember that variables cannot pass values outside of process.

The target of the assignment can be an aggregate. In that case the elements listed must be object names, and the value of the expression must be a composite value of the same type as the aggregate. In this case variable assignment becomes effectively a parallel assignment.

### *10.12.2 If Statement*

If statements represent hardware decoders in both abstract and detailed hardware models. The if statement selects for execution one or more of the enclosed sequences of statements, depending on the value of one or more corresponding conditions. The conditions are expressions resulting in Boolean values. The conditions are evaluated successively until one found that yield the value true. In that case the corresponding sequence of statements is executed. Otherwise, if the else clause is present, its statement is executed. The syntax of if statement is:

```
if condition then
 sequence_of_statements

[elseif condition then
 sequence_of_statements]

[else
 sequence_of_statements]

end if;
```

The if statement can appear in three forms, as **if...then, if...then...else,** and **if...then...elseif.** Examples of these statements are given below:

```
if (x) then
          t-:=a;
   end if;

if (y) then
          t:= b;

 t:=0;
   end if;

if (x) then
            t:=a;
elseif  (y)
          then t:= b;
else
 t:=0;
end if;
```

### *10.12.3 Case Statement*

Case statements are useful to describe decoding of buses and other codes. The case statement selects for execution one of a number of alternative sequences of

statements. The chosen alternative is defined by the value of an expression. The expression must result either in a discrete type, or a one-dimensional array of characters. The syntax of the case statement is:

```
case expression is

    case_statement_alternative

    [case_statement_alternative]

end case;
```

where case_statement_alternative is:

```
when choices =>

    sequence_of_statements
```

All choices must be distinct. Case statement contains multiple when clauses. When clauses allow designer to decode particular values and enable actions following the right arrow **(=>)**. Choices can be in different forms. Examples are given below:

```
case (expression) is

   when 1 => statements;
   when 3 | 4 => ....  --| means "or"
   when 7 to 10 => ......
   when others =>.....

end case;
```

Important rule is that case statement must enumerate all possible values of expression or have an **others** clause. The **others** clause must be the last choice of all the choices. If the expression results in an array, then the choices may be strings or bit strings. Example 10.13 documents behavior of a BCD to seven-segment decoder circuit.

Example 10.13 Describing behavior of a BCD to seven-segment decoder

```
case bcd is
    when "0000" => led <= "1111110";
    when "0001" => led <= "1100000";
```

```
    when "0010" => led <= "1011011"
    when "0011" => led <= "1110011"
    when "0100" => led <= "1100101"
    when "0101" => led <= "0110111"
    when "0110" => led <= "0111111"
    when "0111" => led <= "1100010"
    when "1000" => led <= "1111111"
    when "1001" => led <= "1110111"
    when others => led <= "1111110"
end case;
```

### *10.12.4 Loop Statement*

Loop statements provide a convenient way to describe bit-sliced logic or iterative-circuit behavior. A loop statement contains a sequence of statements that are to be executed repeatedly, zero or more times. The syntax of loop statement is:

```
[loop_label:]
[iteration_scheme]  loop
   sequence_of_statements
end  loop  [loop_label];
```

Iteration scheme is

```
while condition
for loop_parameter_specification
```

Loop_parameter_specification is

```
identifier in  discrete_range
```

There are two different styles of the loop statement: the **for loop** and **while loop.** Examples of the use of these statements are shown below:

```
for k in 1 to 200 loop
   k_new:=k*k;
end loop;

k:=1;

while (k<201) loop
 k_new := k*k;
 k := k+1;
end loop;
```

In the second example, if a while condition evaluates to true it continues to iterate.

The index value in a for loop statement is locally declared by the for statement. This variable does not have to be declared explicitly in the process, function, or procedure. If another variable of the same name exists in the process, function, or procedure, then these two variables are treated as separate variables and are accessed by context. The index value is treated as an object within the statements enclosed into loop statement, and so may not be assigned to. The object does not exist beyond execution of the loop statement.

### *10.12.5 Next Statement*

The next statement is used to skip execution to the next iteration of an enclosing loop statement. The statement can be conditional if it contains condition. The syntax is:

```
next [loop_label] [when condition];
```

Next statement stops execution of the current iteration in the loop statement and skips to successive iterations. Execution of the next statement causes iteration to skip to the next value of the loop index. The loop_label can be used to indicate where the next iteration starts. If the iteration limit has been reached, processing will stop. In the case that execution of the loop has to stop completely, the exit statement is used.

### *10.12.6 Exit Statement*

The exit statement completes the execution of an enclosing loop statement. This completion can be conditional. The syntax is:

```
exit [loop_label] [when condition];
```

Exit stops execution of the iteration of the loop statement. For example:

```
for i in 0 to max loop
   if (p(i) < 0) then exit;
end if;
   p(i) <= (a * i);
end loop;
```

If p(i) <= 0, then exit causes execution to exit the loop entirely. The loop_label is useful to be used in the case of nested loops to indicate the particular loop to be exited. If the exit statement contains loop_label, then it will complete execution of the loop specified by loop_label. The exit statement provides a quick and easy method of exiting a loop statement when all processing is finished or an error or warning condition occurs.

### *10.12.7 Null statement*

The null statement has no effect. It may be used to show that no action is required in specific situation. It is most often used in case statements, where all possible values of the selection expression must be listed as choices, but for some choices no action is required. An example is given below:

```
case op_code is
   when aba => a:=a+b;
   when lda => a:=data;
   ..............
   when nop => null;
end case;
```

### *10.12.8 Assert Statement*

During simulation, it is convenient to output a string message as a warning or error message. The assert statement allows for testing a condition and issuing a message. It checks to determine if a specified condition is true, and displays a message if condition is false. The syntax is:

```
assert  condition
    [report expression]
    [severity expression];
```

Assert writes out text messages during simulation. The assert statement is useful for timing checks, out-of-range conditions, etc. If the severity clause is present, the expression must be of the type severity_level. There are four levels of severity: failure, error, warning, note. If it is omitted the default is error. If the report clause is present, the result of the expression must be a string. This is a message that will be reported if the condition is false. If it is omitted, the default message is "Assertion violation". A simulator may terminate execution if an assertion violation occurs and the severity value is greater than some implementation dependent threshold.

Example of the use of the assert statement is given below:

```
process (clk,din)
    variable x: integer;
    ..............
begin
    ..............
    assert (x > 3)
    report "setup violation"
    severity warning;
end process;
```

The message "setup violation" will be printed if condition is false.

## 10.13 Wait Statement

WAIT statement belongs to sequential statements. It is used in processes for modeling signal-dependent activation. It models a logic block that is activated by one or more signals. It also causes the simulator to suspend execution of a process statement or a procedure, until some conditions are satisfied. The syntax is:

```
wait
    [on signal_name { ,signal_name}]
    [until conditional_expression]
    [for time_expression]
```

A wait statement can appear more than once within a process. Essentially, it can be used in one of three forms: **wait...on, wait...until,** and **wait—for.** In the case of **wait...on** statement, the specified signal(s) must have a change of value that causes the process to resume execution.

Example 10.14 represents a process used to generate a basic sequential circuit, in this case a D flip-flop.

Example 10.14 Description of behavior of a D flip-flop

```
process
begin
    wait until clock = '1' and clock'event;
        q <= d;
end process;
```

The value of d is clocked to q when the clock input has the rising edge. The attribute 'event attached to input clock will be true whenever the clock input has had an event during the current delta time point.

A D flip-flop with asynchronous Clear signal is given in Example 10.15.

Example 10.15 D flip-flop with asynchronous Clear

```
process (clock, clear)
begin
    if clear = '1' then
        q <= d;
    elsif clock'event and clock = '1' then
        q <= d;
    end if;
end process;
```

Instead of listing input signals in the process sensitivity list WAIT statement can be used as in Example 10.16.

Example 10.16 D flip-flop with asynchronous Clear using WAIT statement

```
process
begin
    if clear = '1' then
        q <= d;
    elsif clock'event and clock = '1' then
        q <= d;
    end if;

    wait on (clear, clock);
end process;
```

The wait statement can be used with different operations together. A single statement can include an **on** signal, **until** expression, and **for** time_expression clauses. However, one must ensure that the statement contains expressions in which at least one signal appears. This is necessary to ensure that wait statement does not wait forever. Only signals have events on them, and only they can cause a wait statement or concurrent signal assignment to reevaluate. Some further properties of signals, concurrent assignment statements, and the use of wait statement will be discussed in the subsequent chapters.

The process that does not include a sensitivity list executes from the beginning of the process body to the first occurrence of a wait statement, then suspends until the condition specified in the wait statement is satisfied. If the process includes only single wait statement, the process resumes when the condition is satisfied and continues to the end of the process body, then begins executing again from the beginning until encounters the wait statement. If there are multiple wait statements in the process, the process executes only until the next wait statement is encountered. In this way very complex behaviors can be described, including multiple-clock circuits and systems.

## 10.14 Subprograms

Subprograms are used to document frequently used functions in behavioral design descriptions. There are two different types of subprograms:

- procedures, that returns multiple values, and
- functions, that returns single value.

A subprogram contains sequential statements, just like a process. Subprograms can declare local variables that exist only during execution of subprogram. They are declared using the syntax:

```
procedure designator  [formal_parameter_list]
```

or

```
function designator  [formal_parameter_list]
   return type_mark
```

A subprogram declaration in this form names the subprogram and specifies parameters required. The body of statements defining the behavior of the subprogram is deferred. For functions, the declaration also specifies the type of the result returned when function is called. This type of subprograms is typically used in package specifications, where the subprogram body is given in the package body.

The formal_parameter_list contains declaration of interface elements which includes constants, variables and signals. If constants are used they must be in **in** mode.

When the body of a subprogram is specified, the syntax used is as follows:

```
procedure designator [formal_parameter_list] is

   subprogram_declarative_part

begin
   subprogram_statement_part
end [designator];
```

or

```
function designator  [formal_parameter_list]  return
          type_mark is

   subprogram_declarative_part
```

```
begin

    subprogram_statement_part

end [designator];
```

The subprogram_declarative_part can contain any number of following:

- subprogram declaration
- subprogram body
- type declaration
- subtype declaration
- constant declaration
- variable declaration
- alias declaration

The declarative items listed after the subprogram specification declare things which are to be used locally within the subprogram body. The names of these items are not visible outside of the subprogram, but are visible within locally declared subprograms. They also shadow all things with the same names declared outside of the subprogram.

The subprogram_statement_part contains sequential statements. When the subprogram is called, the statements in the subprogram body are executed until the end statement is encountered or a return statement is executed. The syntax of return statement is:

```
return [expression];
```

The return statement in the procedure body must not contain an expression. However, in the case of function, there must be at least one return statement with expression, and a function must complete by executing a return statement. The value of the expression is the value returned to the function call.

### *10.14.1 Functions*

User-defined function must be declared in advance, before it is used. The function accepts the values of input parameters, but returns only one value. It actually executes and evaluates like an expression.

Consider an example of function declaration:

```
function byte_to_int(alpha: byte) return integer;
```

which converts a variable of the type byte into integer. For functions, the parameter mode must be **in,** and this is assumed. The only parameter alpha is of type byte. If its class is not specified, it is assumed to be **constant.** The value returned by the body of this function must be an integer. The body of this function and the call to the function are given in Example 10.17.

Example 10.17 Defining and using function

```
function byte_to_int(alpha: byte) return integer is
   variable result: integer:=0;
begin
    for n in 0 to 7 loop
       result:= result*2 + bit'pos(alpha(n));
    end loop;
    return result;
end byte_to_int;

process
    variable data: byte;
begin
     .....
     byte_to_int(data);
     ......
end process;
```

Similarly, functions can be declared in an entity or in a package. Vendors usually provide utility functions in a VHDL package. These are source code design units that are compiled and used from VHDL library.

### *10.14.2 Procedures*

A procedure is also a type of subprogram. With the procedure, more than one value can be returned using parameters. The parameters are of type **in, out,** and **inout.** If not specified, the default value is **in.** If mode in is used, it brings a value in, **out** sends a value back through an argument list, and **inout** brings a value in and sends it back. Parameters can be variables or signals. Signals must be declared. Procedures can contain wait statements, and signal parameters can pass signals to be waited on. Local variables can be declared in a procedure. A procedure call is a statement. The procedure must be declared in a package, in a process header or in architecture declaration prior to its call. Parameters can be assigned a default value that is used when no actual parameter is specified in a procedure call.

Procedure shown in Example 10.18 converts a vector of bits into an integer. The procedure body assigns the value to q and converts bits in z to an integer q. The procedure also returns a flag to indicate whether the result was 0.

Example 10.18 Defining procedure

```
procedure vector_to_int (z: in bit_vector;
    zero_flag: out boolean; q: inout integer) is

begin
    q := 0;
    zero_flag := true;
    for i in 1 to 8 loop
       q := q*2;
       if (z(i) = '1') then
          q := q + 1;
          zero_flag := false;
       end if;
     end loop;
end vector_to_unt;
```

In addition to giving back q, an integer, it also returns a zero_flag that tells if the result was a 0 or not; the result is true or false.

## 10.15 Questions and Problems

10.1 Explain what is a strongly typed language.

10.2 What are the literal types supported in VHDL? Explain the difference between the following literals:

1, '1', 1.0

10.3 What are the basic data types supported in VHDL?

10.4 Why is VHDL object-oriented language? What are the objects supported in VHDL?

10.5 What is the difference between variable and signal?

10.6 Two parts of a digital system communicate using TRANSFER signal that enables transfer of 20,000 different values. Show at least two data types that enable description of this signal.

10.7 What are the similarities and differences between bit and Boolean data type?

10.8 What is the physical data type useful for? Explain it on the example of time physical type.

10.9 Is the real type synthesizable? Explain it.

10.10 Is the real type synthesizable? Explain it.

10.11 What the enumerated types are useful for? Give a few examples of using the enumerated types.

10.12 Is the enumerated type synthesizable?

10.13 Given a bus in a computer system that contains 16-bit address lines, 8-bit data lines, and two control lines to read from and write to memory. Declare the single composite type that describes the bus and its constituent components. Use two approaches: a) declare first the bus and then use aliases to describe its components, and b) declare its components and then integrate them into the bus.

10.14 What is the difference between the following tests:

```
•if (clk = '1') then

•if (clk = '1') and (clk'event)

•if (clk = '0') and (clk'event)
```

10.15 Describe the difference between bit and std_logic type.

10.16 What is the IEEE library package std_logic_1164? What the overloaded language operators defined in this package?

10.17 What is the IEEE Standard 1076.3 (The Numeric Standard)? Why is it introduced?

10.18 What are the type conversions? What are closely related types? Explain why some type conversions are synthesizable, while the other are not.

10.19What is the difference between concurrent and sequential statements in VHDL?

10.20Describe the role of processes in VHDL. What is the sensitivity list?

10.21 You have to design a modulo-n counter. It has to be described using two processes: the first one is a free-running counter, and the second one is checking the state of the free-running counter to reset it when the final state has been reached. Describe the role of variables and signals in the description of the counter.

10.22Describe the role of wait statement within the processes. Compare the use of wait statements and sensitivity lists.

10.23Describe the use of functions and procedures in VHDL.

# 11 VHDL AND LOGIC SYNTHESIS

This Chapter presents a quick tour through VHDL synthesis of all types of logic circuits and specifics of VHDL in the Max+Plus II design environment. First, we present some specifics of VHDL in Altera's environment. After that, we consider purely combinational logic including some of the standard combinational logic functions, then standard sequential logic circuits, and, finally, general sequential logic represented by finite state machines. Further examples of VHDL descriptions, which result in valid synthesized circuits for Altera FPLDs, are also presented.

## 11.1 Specifics of Altera's VHDL

VHDL is fully integrated into the Max+Plus II design environment. VHDL designs can be entered with any text editor, and subsequently compiled with the compiler to create output files for simulation, timing analysis, and device programming. Max+Plus II supports a subset of IEEE 1076-1987/1993 VHDL as described in the corresponding Altera documentation. VHDL design files, with the extension .vhd, can be combined with the other design files into a hierarchical design, called project. Each file in a project hierarchy, i.e., each macrofunction, is connected through its input and output ports to one or more design files at the next higher hierarchy level.

Max+Plus II environment allows a designer to create a symbol that represents a VHDL design file and incorporate it into graphic design file. The symbol contains graphical representation of input and output ports, as well as some parameters, which can be used to customize design to the application requirement. Custom functions, as well as Altera-provided macrofunctions, can be incorporated into any VHDL design file. Max+Plus II Compiler automatically processes VHDL design files and optimizes them for Altera FPLD devices. The Compiler can be directed to create a VHDL Output File (.vho file) that can be imported into an industry standard environment for simulation. On the other side, after a VHDL project has compiled successfully, optional simulation and timing analysis with Max+Plus II can be done, and then Altera device programmed.

Altera provides ALTERA library that includes the maxplus2 package, which contains all Max+Plus II primitives and macrofunctions supported by VHDL. Besides that, Altera provides several other packages located in subdirectories of the \maxplus2\max2vhdl directory, as it is shown in Table 11.1.

In addition, Altera provides the STD library with the STANDARD and TEXTIO packages that are defined in the IEEE standard VHDL Language Reference Manual. This library is located in the \maxplus2\max2vhdl\std directory.

Table 11.1 Max+Plus II Packages

| Package | Library | Contents |
|---|---|---|
| maxplus2 | altera | Max+Plus II primitives and macrofunctions supported by VHDL |
| std_logic_1164 | ieee | Standard for describing interconnection data types for VHDL modeling, and the std_logic and std_logic_vector types |
| std_logic_arith | ieee | Signed and unsigned types, arithmetic and comparison functions for use with signed and unsigned types, and the conversion functions conv_integer, conv_signed, and conv_unsigned |
| std_logic_signed | ieee | Functions that allow Max+Plus II to use std_logic_vector types as if they are signed types |
| std_logic_unsigned | ieee | Functions that allow Max+Plus II to use std_logic_vector types as if they are unsigned types |

## 11.2 Combinational Logic Implementation

Combinational logic circuits are commonly used in both the data path and control path of more complex systems. They can be modeled in different ways using signal assignment statements which include expressions with logic, arithmetic and relational operators, and also can be modeled using if, case and for statements. If conditional signal assignment statements are used, then the selected signal assignment statement corresponds to the if statement, and the conditional signal assignment statement corresponds to the case statement. However, all signal assignment statements can also be considered as processes that include single

statement. This further means that they are always active (during the simulation process) waiting for events on the signals in the expressions on the right side of the assignment statement. Another way of modeling combinational logic is to use processes and sequential statements within processes. Both of these statements should be placed in the architecture body of a VHDL design file, as shown in the following template:

```
architecture arch_name of and_gate is
begin
    [concurrent_signal_assignments]
    [process_statements]
    [other concurrent statements]
end arch_name;
```

### *11.2.1 Logic and Arithmetic Expressions*

Both logic and arithmetic expressions may be modeled using logic, relational and arithmetic operators. The expressions take the form of continuous dataflow-type assignments. Some VHDL operators are more expensive to compile because they require more gates to implement (like programming languages where some operators take more cycles to execute). The designers need to be aware of these factors. If an operand is a constant, less logic will be generated. If both operands are constants, the logic can be collapsed during compilation, and the cost of the operator is zero gates. Using constants wherever possible means that the design description will not contain extra functionality. The design will be compiled faster and produce a smaller implementation.

#### Logical Operators

Standard VHDL logical operators are defined for the types bit, std_logic, boolean and arrays of bit, std_logic or boolean (for example, bit_vector or std_logic_vector). The synthesis of logic is fairly direct from the language construct, to its implementation in gates, as shown in examples 11.1 and 11.2.

Example 11.1 Synthesizing logic from the language construct

```
library ieee;
use ieee.std_logic_1164.all;

entity logic_operators_1 is
   port (a, b, c, d: in std_logic; y: out std_logic);
end logic_operators_1;
```

```
architecture arch1 of logic_operators_1 is
   signal e: bit;

begin
   y <= (a and b) or e; --concurrent signal assignments
           e <= c  or d;
end arch1;
```

Schematic diagram corresponding to this example is shown in Figure 11.1.

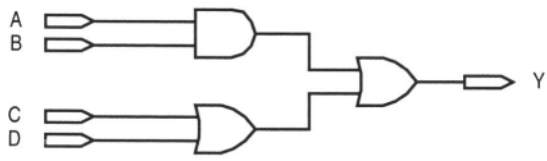



Figure 11.1 Schematic diagram corresponding to Example 11.1

Example 11.2 Synthesizing logic from the language construct

```
library ieee;
use ieee.std_logic_1164.all;

entity logic_operators_2 is
   port (a, b: in std_logic_vector (0 to 3) ;
                  y: out std_logic_vector (0 to 3));
end entity logical_ops_2

architecture arch2 of logic_operators_2 is

begin
   y <= a and b;
end architecture arch1;
```

Schematic diagram corresponding to this example is shown in Figure 11.2.

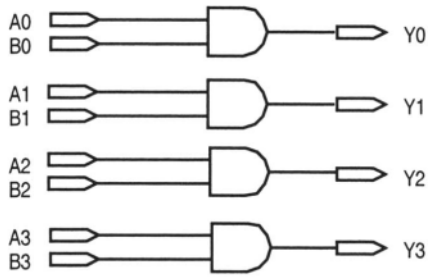



Figure 11.2 Schematic diagram corresponding to 4-bit "and"

**Relational Operators**

The simple comparisons operators ( **=** and **/=** ) are defined for all types. The ordering operators ( **>=, <=, >, <** ) are defined for numeric types, enumerated types, and some arrays. The resulting type for all these operators is Boolean. The simple comparisons, equal and not equal, are cheaper to implement (in terms of gates) than the ordering operators. To illustrate, Example 11.3 below uses an equal operator to compare two 4-bit input vectors. Corresponding schematic diagram is presented in Figure 11.3.

Example 11.3 Synthesizing logic from relational operators

```
library ieee;
use ieee.std_logic_1164.all;

entity relational_operators_1 is
   port (a, b: in std_logic_vector (0 to 3); y: out boolean);
end relational_operators_1;

architecture arch1 of relational_operators_1 is

begin
   y <= a = b;
end arch1;
```

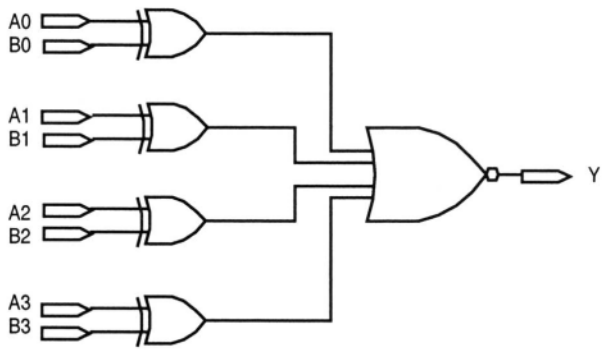



Figure 11.3 Schematic diagram corresponding to 4-bit equality comparator

Example 11.4 uses a greater-than-or-equal-to operator ('>=').

Example 11.4 Synthesizing logic from relational operators

```
library ieee;
use ieee.std_logic_1164.all;

entity relational_operators_2 is
   port (a, b: in integer range 0 to 15; y: out boolean);
end relational_operators_2;

architecture arch2 of relational_ops_2 is

begin
   y <= a >= b
end arch2;
```

As it can be seen from the schematic corresponding to this example, presented in Figure 11.4, it uses more than twice as many gates as the previous example.

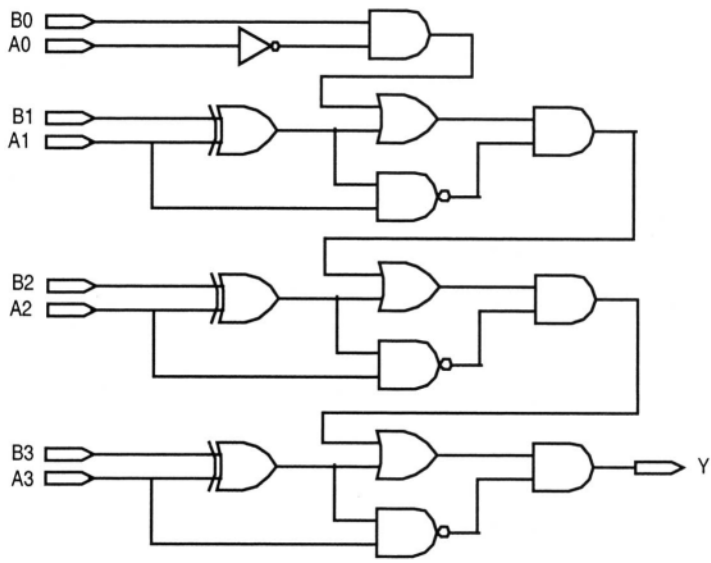



Figure 11.4 Schematic diagram corresponding to "a =>b" comparator

### Arithmetic Operators

While the adding operators (+, -) are fairly expensive in terms of number of gates to implement, the multiplying operators (*, /, **mod**, **rem**) are very expensive. Implementation of these operators is highly dependent on the target technology. Example 11.5 illustrates the use of arithmetic operators and parentheses to control synthesized logic structure.

Example 11.5 Using arithmetic operators

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity arithmetic_operators is
   port (a, b, c, d: in unsigned(7 downto 0);
         y1, y2: out unsigned(9 downto 0);
end arithmetic_operators;

architecture arch1 of arithmetic_operators is
```

```
begin
    y1 <= a + b + c + d;
    y2 <= (a + b) + (c+d);
end arithmetic_operators;
```

Another possibility is to enclose signal assignment statements into a process with all input signals in the sensitivity list of a process. From the synthesis point of view, there will be no difference. However, simulation can be simpler if a process is used to describe the same circuit.

### *11.2.2 Conditional Logic*

VHDL provides two concurrent statements for creating conditional logic:

- conditional signal assignment, and
- selected signal assignment

and two sequential statements for creating conditional logic:

- if statement, and
- case statement

Examples 11.6 and 11.7 illustrate the use of these statements for creating conditional logic.

Example 11.6 Using conditional signal assignment

```
library ieee;
use ieee.std_logic_1164.all;

entity condit_stmts_1 is
    port (sel, b, c: boolean; y: out boolean);
end condit_stmts_1;

architecture concurrent of condit_stmts_1 is

begin
    y <= b when sel else c;
end concurrent;
```

The same function can be implemented using sequential statements and occur inside a process statement. The condition in an if statement must evaluate to true or false (that is, it must be a Boolean type).

Example 11.7 Using process to synthesize logic

```
architecture sequential of condit_stmts_1 is
begin
   process (s, b, c)
      variable n: boolean;
   begin
      if sel then
        n := b;
      else
        n := c;
      end if;
      y<=n;
   end process;
end architecture sequential;
```

The schematic diagram of the circuit generated from the above examples is shown in Figure 11.5.

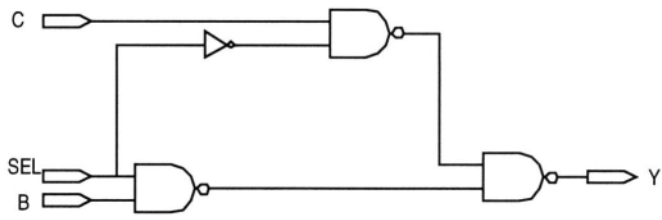



Figure 11.5 Schematic diagram corresponding to Example 11.7

Example 11.8 shows the use of the selected signal assignment for creating conditional logic that implements a multiplexer. All possible cases must be used for selected signal assignments. The designer can be certain of this by using an **others** case.

Example 11.8 Synthesizing multiplexer using selected signal assignment

```
library ieee;
use ieee.std_logic_1164.all;

entity condit_stmts_2 is
   port (sel: in std_logic_vector (0 to 1);
      a,b,c,d : in std_logic; y: out bit);
end condit_stmts_2;

architecture concurrent of condit_stmts_2 is
begin
   with sel select
      y <= a when '00',
      y <= b when '01',
      y <= c when '10',
      y <= d when others;
end concurrent;
```

The same function can be implemented using sequential statements and occur inside a process statement. Example 11.9 illustrates the use of case statement.

Example 11.9 Synthesizing multiplexer using process statement

```
architecture sequential of condit_stmts_2 is
begin
   process (sel,a,b,c,d)
   begin
      case sel is
        when '00' => y <= a;
        when '01' => y <= b;
        when '10' => y <= c;
        when others => y <= d;
      end case;
   end process;
end sequential;
```

Schematic diagram illustrating generated logic for examples 11.8 and 11.9 is shown in Figure 11.6.

Using a case statement (or selected signal assignment) will generally compile faster and produce logic with less propagation delay than using nested if statements (or a large selected signal assignment). VHDL requires that all the possible conditions be represented in the condition of a case statement. To ensure this, use the **others** clause at the end of a case statement to cover any unspecified conditions.

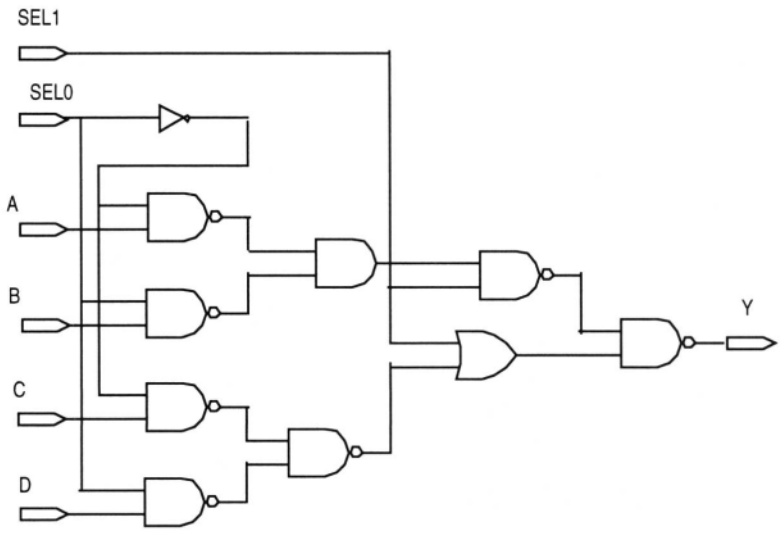



Figure 11.6 Schematic diagram corresponding to Examples 11.8 and 11.9

### *11.2.3 Three-State Logic*

When data from multiple possible sources need to be directed to one or more destinations we usually use either multiplexers or three-state buffers. This section shows the different ways in which three-state buffers may be modeled for inference by synthesis tools. VHDL provides two methods to describe three-state buffers: either by using the 'Z' high-impedance value in standard logic defined in IEEE std_logic_1164, or using an assignment of null to turn off a driver. The first method applies to the type std_logic only, the second method applies to any type. Three-state buffers are modeled then using conditional statements:

- if statements,
- case statements,
- conditional signal assignments

A three-state buffer is inferred by assigning a high-impedance value 'Z' to a data object in the particular branch of the conditional statement. In the case of modeling multiple buffers that are connected to the same output, each of these buffers must be described in separate concurrent statement. Example 11.10 shows a four-bit three-state buffer.

Example 11.10 Synthesizing three-state buffer

```
library ieee;
use ieee.std_logic_1164.all;

entity tbuf4 is
   port (enable : std_logic;
      a : std_logic_vector(0 to 3);
      y : out std_logic_vector(0 to 3));
end tbuf4;

architecture arch1 of tbuf2 is
begin
   process (enable, a)
   begin
      if enable= '1' then
        y <= a;
      else
        m <= 'Z' ;
      end if;
   end process;
end tbuf4;
```

The same function can be achieved by using the equivalent concurrent statement:

```
architecture arch2 of tbuf4 is
begin
    y <= a when enable='1' else 'Z';
end;
```

Schematic diagram of the circuit corresponding to Example 11.10 is shown in Figure 11.7.

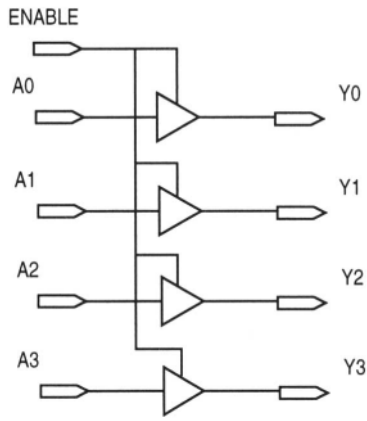



Figure 11.7 Schematic diagram corresponding to Example 11.10

An internal tristate bus may be described as in Example 11.11.

Example 11.11 An internal three-state bus

```
library ieee;
use ieee.std_logic_1164.all;

entity tbus is
   port (enable1, enable2, enable3 : std_logic;
      a, b, c : std_logic_vector(0 to 3);
      y : out std_logic_vector(0 to 3));
end  tbus;

architecture arch of tbus is
begin
   y <= a when enable0 = '1'  else  'Z';
   y <= b when enable1 = '1'   else  'Z';
   y <= c when enable2 = '1'  else  'Z';
end arch;
```

Schematic diagram corresponding to this example is shown in Figure 11.8.

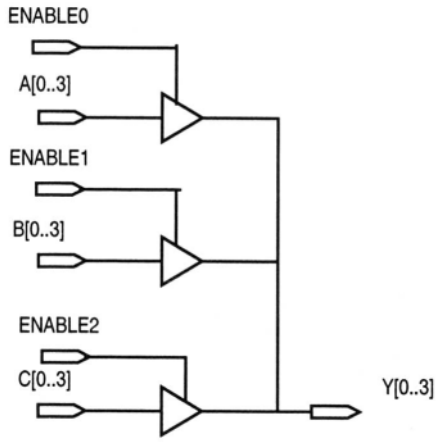



Figure 11.8 Schematic diagram corresponding to example

Three-state buffers can be modeled using case statements. Example 11.12 shows the use of case statement.

Example 11.12 Three-state buffer using process statement

```
library ieee;
useieee.std_logic_1164.all;

entity tbuf is
   port (a : in std_logic_vector(0 to 2);
      enable: in integer range 0 to 3;
      y : out std_logic;
end tbuf;

architecture arch1 of tbuf is
begin
   process (enable, a)
      case enable is
        when 0      y <= a(0) ;
        when 1  y <= a(1);
        when 2  y <= a(2);
        when others  y <= 'Z' ;
   end case;
end process;
end arch1;
```

The problem with case statement is that others clause cannot be used to assign both three-state and don't-care output value to reduce logic. In that case the solution is to use case statement for minimization of logic by employing don't-care conditions, and to use a separate conditional signal assignment to assign the high-impedance value to infer three-state buffer.

Another way to model three-state buffers is to use the assignment of **null** to a signal of kind **bus** to turn off its driver. When embedded in an if statement, a **null** assignment is synthesized to a three-state buffer as shown in Example 11.13.

Example 11.13 Synthesis of three state-buffers using null assignment

```
library ieee;
use ieee.std_logic_1164.all;

package pack_bus is
    subtype bus8 is integer range 0 to 255;
end pack_bus;

use work.pack_bus.all;

entity tbuf8 is
   port (enable: in boolean; a: in bus8; y: out bus8 bus);
end tbuf8;

architecture arch1 of tbuf is
begin
   process (enable, a)
   begin
       if enable then
          y <= a;
       else
          y <= null;
   end if;
   end process;
end arch1;
```

### *11.2.4 Combinational Logic Replication*

VHDL provides a number of constructs for creating replicated logic. In Chapter 6 we have considered component instantiation and the use of a generate statement as a concurrent loop statement in structural type of VHDL models. However, a number of other constructs are also used to provide replication of logic, namely:

- loop statement,
- function, and
- procedure.

Functions and procedures are referred to as subprograms. These constructs are synthesized to produce logic that is replicated once for each subprogram call, and once for each iteration of a loop. If possible, loop and generate statement ranges should be expressed as constants. Otherwise, the logic inside the loop may be replicated for all the possible values of loop ranges. This can be very expensive in terms of gates.

Example 11.14 shows how loop statement can be used to replicate logic.

Example 11.14 Using loop to replicate logic

```
library ieee;
use ieee.std_logic_1164.all;

entity loop_stmt is
   port (a: in std_logic_vector (0 to 3) ;
      y: out std_logic_vector (0 to 3)) ;
end loop_stmt;

architecture arch1 of loop_stmt is
begin
   process (a)
      variable temp: std_logic;
   begin
      temp := '1';
      for i in 0 to 3 loop
        temp := a(3-i) and temp;
        y(i) <= temp;
      end loop;
   end process;
end arch1;
```

Schematic diagram illustrating synthesized circuit from this example is shown in Figure 11.9.

A loop statement replicates logic, therefore, it must be possible to evaluate the number of iterations of a loop at compile time. Loop statements may be terminated with an exit statement, and specific iterations of the loop statement terminated with a next statement as it was shown in preceding chapter. While exit and next can be useful in simulation, in synthesis they may synthesize logic that gates the following

loop logic. This may result in a carry-chain-like structure with a long propagation delay in the resulting hardware.

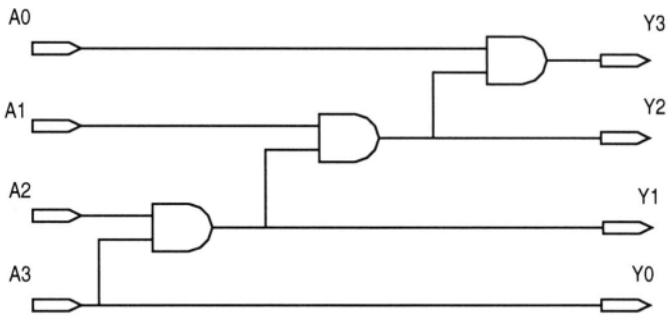



Figure 11.9 Schematic diagram corresponding to Example 11.14

A function is always terminated by a return statement, which returns a value. A return statement may also be used in a procedure, but it never returns a value. Example 11.15 is using function to generate replicated logic.

Example 11.15 Using functions to replicate logic

```
library ieee;
use ieee.std_logic_1164.all;

entity replicate is
   port (a: in std_logic_vector (0 to 3);
      y: out std_logic_vector (0 to 3));
end replicate;

architecture arch1 of replicate is

function replica (b, c, d, e: std_logic) return std_logic is
begin
   return not ( (b xor c) ) and (d xor e);
end;

begin
   process (a)
   begin
      y(0) <= replica(a(0), a(1), a(2), a(3));
```

```
        y(1) <= replica(a(3), a(0), a(1), a(2));
        y(2) <= replica(a(2), a(3), a(0), a(1));
        y(2) <= replica(a(1), a(2), a(3), a(0));
    end process;
end architecture arch1;
```

Schematic diagram which illustrates this example is shown in Figure 11.10.

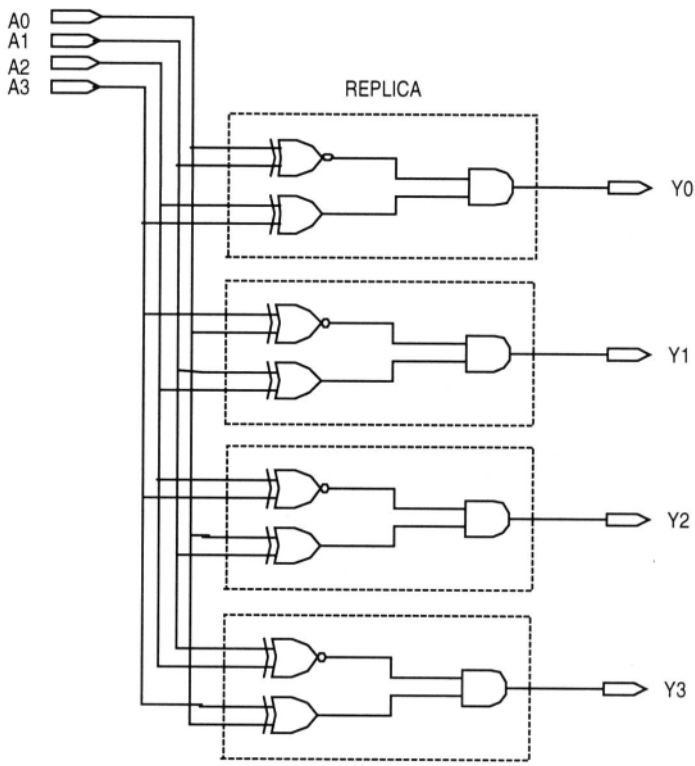



Figure 11.10 Schematic diagram corresponding to Example 11.15

### *11.2.5 Examples of Standard Combinational Blocks*

In this subsection we will present a number of examples of standard combinational blocks and various ways to describe them in VHDL. These blocks usually represent units that are used to form data paths in more complex digital designs. All of these

designs are easily modifiable to suit the needs of specific application. Different approaches to modeling are used to demonstrate both versatility and power of VHDL.

Multiplexer is modeled with two concurrent statements: one is producing an intermediate internal signal sel (of integer type) which selects the input that is forwarded to the output of multiplexer in another concurrent statement. Example 11.16 demonstrates the use of various data types, and conversion between those types.

Example 11.16 8-bit 4 to 1 multiplexer - behavioral model

```
library ieee;
use ieee.std_logic_1164.all;

entity mux41beh is
   port(a, b, c ,d: in std_logic_vector(7 downto 0);
       s0, s1: in std_logic; -- select input lines
       y: out std_logic_vector(7 downto 0));
end mux41beh;

architecture beh of mux41beh is
   signal sel: integer;
begin
    sel <= 0 when <s1='0' and s0 ='0') else
    1 when (s1='0' and s0 ='1') else
    2 when (s1='1' and s0 ='0') else
    3 when (s1='1' and s0 ='1') else
    4;

   with sel select
       y <= a when 0,
       b when 1,
       c when 2,
       d when 3,
       "XXXXXXXX" when others;

end beh;
```

In order to model the same multiplexer structurally, we first design elementary logic gates (minv inverter, *mand3* 3-input and gate and mor4 4-input or gate) and include them in the package my_gates. Components from this package are used in the structural model. Because of the multi-bit inputs and outputs to the multiplexer, components are instantiated using for-generate statement. The whole VHDL model

including the order in which design units are compiled (first individual components, then package, and at the end multiplexer unit) is shown in Example 11.17 below.

Example 11.17 8-bit 4 to 1 multiplexer - structural model

```
library ieee;  -- must be declared before each design unit
use ieee.std_logic_1164.all;

entity minv is       -- inverter
   port (a: in std_logic; y: out std_logic);
end minv;

architecture dflow of minv is
begin
   y <= not a;
end dflow;

library ieee;
use ieee. std_logic_1164. all;

entity mand3 is     -- 3-input and gate
   port (a, b, c: in std_logic; y: out std_logic);
end mand3;

architecture dflow of mand3 is
begin
   y <= a and b and c;
end dflow;

library ieee;
use ieee.std_logic_1164.all;
entity mor4 is      -- 4-input or
   port (a, b, c, d: in std_logic;
                     y: out std_logic);
end mor4;

architecture dflow of mor4 is
begin
   y <= a or b or c or d;
end dflow;

library ieee; -- separately compiled package
use ieee.std_logic_1164.all;
use work.all;  -- all previously declared components are in
                     --work library

package my_gates is
```

```
component minv
   port(a: in std_logic; y: out std_logic);
end component;

component mand3
   port(a, b, c: in std_logic; y: out std_logic);
end component;

component mor4
   port(a, b, c,d: in std_logic; y: out std_logic);
end component;

end my_gates;

library ieee;
use ieee.std_logic_1164.all;
use work.my_gates.all;  -- package used in structural model

entity mux41str is
   port(a, b, c, d: in std_logic_vector(7 downto 0);
       s1, s0:      in std_logic;
       y: out std_logic_vector(7 downto 0)) ;
end mux41str;

architecture struct of mux41str is
   signal s1n, s0n: std_logic;  -- internal signals
                          --to interconnect components
   signal ma,mb,mc,md: std_logic_vector(7 downto 0) ;

begin
   u_inv0: minv port map(s0, s0n) ;
   u_invl: minv port map(s1, s1n);

   f1:  for i in 0 to 7 generate
       u_ax: mand3 port map (s1n, s0n, a(i), ma(i));
       u_bx: mand3 port map (s1n, s0, b(i), mb(i));
       u_cx: mand3 port map (s1, s0n, c(i), mc(i));
       u_dx: mand3 port map (s1, s0, d(i), md(i));
       u_ex: mor4 port map (ma(i), mb(i), mc(i),md(i), y(i));
   end generate f1;
end struct;
```

Example 11.18 shows two different behavioral architectures of 8-to-3 encoder. The first architecture uses if statement while the second architecture uses a case statement within a process. The use of the if statements introduces delays because the circuit inferred will evaluate expressions in the order in which they appear in the model (the expression at the end of the process is evaluated last). Therefore, the use of the case statement is recommended. It also provides a better readability.

Example 11.18 8-to-3 Encoder

```
library ieee;
use ieee.std_logic_1164.all;

entity encoder83 is
   port(a: in std_logic vector (7 downto 0);
      y: out std_logic_vector(2 downto 0)) ;
end encoder83;

architecture arch1 of encoder83 is
begin
   process(a)
   begin
      if (a="00000001") then
        y <= "000";
      elsif (a="000000010") then
        y <= "001";
      elsif (a="00000100") then
        y <= "010";
      elsif (a="00001000") then
        y <= "011";
      elsif (a="00010000") then
        y <= "100";
      elsif (a="00100000") then
        y <= "101";
      elsif (a="01000000") then
        y <= "110";
      elsif (a="10000000") then
        y <= "111";
      else
        y <= "XXX";
      end if;
   end process;
end arch1;

-- alternative case

architecture arch2 of encoder83 is
begin
   process(a)
   begin
      case a is
        when "00000001" => y <= "000";
        when "00000010" => y <= "001";
        when "00000100" => y <= "010";
        when "00001000" => y <= "011";
        when "00010000" => y <= "100";
        when "00100000" => y <= "101";
```

```
        when "01000000" => y <= "110";
        when "10000000" => y <= "111";
      end case;
   end process;
end arch2;
```

Example 11.19 of a 3-to-5 decoder is straightforward. However, it is important to notice that the behavior for unused combinations is specified to avoid generation of unwanted logic or latches.

Example 11.19 3-to-5 binary decoder with enable input

```
library ieee;
use ieee.std_logic_1164.all;

entity decoder35 is
   port(a: in integer; en: in std_logic;
      y: out std_logic_vector(4 downto 0)) ;
end encoder83;

architecture arch1 of encoder83 is
begin
   y  <= 1 when (en='1' and  a=0) else
         2 when (en='1' and  a=1) else
         4 when (en='1' and  a=2) else
         8 when (en='1' and  a=3) else
        16 when (en='1'  and  a=1) else
        0;
end arch1;
```

Example 11.20 is introduced just to illustrate an approach to the description of a simple arithmetic and logic unit (ALU) as a more complex, but still common, combinational circuit. However, most of the issues in the design of the real ALUs are related to efficient implementation of basic operations (arithmetic operations such as addition, subtraction, multiplication, and division, shift operations, etc.). The ALU in this example performs operations on one or two operands that are received on two 8-bit busses (a and b) and produces output on 8-bit bus (f). Operation performed by the ALU is specified by operation select (opsel) input lines. Input and output carry are not taken into account.

Example 11.20 A simple arithmetic and logic unit

```
library ieee;
use ieee.std_package_1164.all;

type ops is (add, nop, inca, deca, loada, loadb, op_nega,
op_negb, op_and, op_or, shl, shr) ;

entity alu is
   port (a, b: in std_logic_vector(7 downto 0);
      opsel: in ops;
      clk: in std_logic;
      f: out std_logic_vector(7 downto 0));
end alu;

architecture beh of alu is
begin
   process

   procedure "+" (a, b: std_logic_vector)
      return std_logic_vector is
      variable sum: std_logic_vector (0 to a'high);
      variable c: std_logic:= '0';
   begin
      for i in 0 to a'high loop
         sum(i) := a(i) xor b(i) xor c;
         c := (a(i) and c) or (b (i) and c) or(a(i) and b(i));
   end loop;
   return sum;
   end;

   function shiftl(a: std_logic_vector)
      return std_logic_vector is
      variable shifted: std_logic_vector (0 to a'high);
   begin
      for i in 0 to a'high -1 loop
         shifted(i +1) := a(i);
      end loop;
   return shifted;
   end;

   function shiftr(a: std_logic_vector)
      return std_logic_vector is
      constant highbit: integer := a'high;
      variable shifted: bit_vector (0 to highbit);
   begin
      shifted(0 to highbit - 1) := a(1 to highbit);
   return shifted;
   end;
```

```
    begin
        wait until clk;
        case opsel is
           when add  => f <= a + b;
           when inca => f <= a + 1;
           when deca => f <= a -1;
           when nop => null; -- A null statement,
           when op_nega => f <= not a;
           when op_negb => f <= not b;
           when op_and => f <= a and b;
           when op_or => f <= a or b;
           when shl => f <= shiftl(a);
           when shr => f <= shiftr(a);
           when loada => f <= a;
           when loadb => f <= b;
        end case;
    end process;
end beh;
```

## 11.3 Sequential Logic Synthesis

In VHDL we describe the behavior of a sequential logic element, such as a latch or flip-flop, as well as the behavior of more complex sequential machines. This section shows how to model simple sequential elements, such as latches and flip-flops, or more complex standard sequential blocks, such as registers and counters. The behavior of a sequential logic element can be described using a process statement (or the equivalent procedure call, or concurrent signal assignment statement) because the sequential nature of VHDL processes makes them ideal for the description of circuits that have memory and must save their state over time. At this point it is very important to notice that processes are equally suitable to describe combinational circuits, as it was shown in preceding section. If our goal is to create sequential logic (using either latches or flip-flops) the design is to be described using one or more of the following rules:

1. Write the process that does not include all entity inputs in the sensitivity list (otherwise, the combinational circuit will be inferred).

2. Use incompletely specified if-then-elsif logic to imply that one or more signals must hold their values under certain conditions.

3. Use one or more variables in such a way that they must hold a value between iterations of the process.

### *11.3.1 Describing Behavior of Basic Sequential Elements*

The two most basic types of synchronous element, which are found in majority of FPLD libraries, which synthesis tools map to, are:

- the D-type flow-through latch, and
- the D-type flip-flop

Some of the vendor libraries contain other types of flip-flops, but very often they are derived from the basic D-type flip-flop. In this section we consider the ways of creating basic sequential elements using VHDL descriptions.

A D-type flow-through latch, or simply latch, is a level sensitive memory element that is transparent to signal passing from the D input to Q output when enabled (ENA = 1), and holds the value of D on Q at the time when it becomes disabled (ENA = 0). The model of the latch is presented in Figure 11.11.

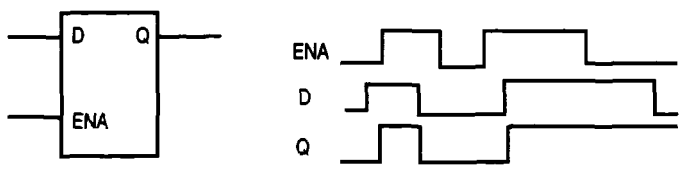



Figure 11.11 Model of the level sensitive latch

The D-type flip-flop is an edge-triggered memory element that transfers a signal's value on its input D to its Q output when an active edge transition occurs on its clock input. The output value is held until the next active clock edge. The active clock edge may be transition of the clock value from '0' to '1' (positive transition) or from '1' to '0' (negative transition). The Qbar signal is always the complement of the Q output signal. The model of the D-type flip-flop with positive active clock edge is presented in Figure 11.12.

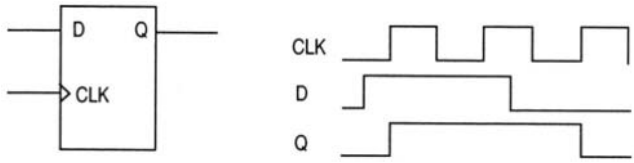



Figure 11.12 Model of the edge-triggered D-type flip-flop

There are three major methods to describe behavior of basic memory elements:

- using conditional specification,
- using a wait statement, or
- using guarded blocks.

Conditional specification is the most common method. This relies on the behavior of an if statement, and assigning a value in only one condition. The following example describes the behavior of a latch:

```
process (elk)
begin
    if clk = '1' then
          y <= a;
          else  --do nothing
    end if;
end process;
```

If the clock is high the output (y) gets a new value, if not the output retains its previous value. Note that if we had assigned in both conditions, the behavior would be that of a multiplexer.

The key to specification of a latch or flip-flop is incomplete assignment using the if statement. However, that incomplete assignment is within the context of the whole process statement. A rising edge flip-flop is created by making the latch edge sensitive:

```
if clk and clk'event then
      y <= a;
end if;
```

The second method uses a wait statement to create a flip-flop. The evaluation is suspended by a wait-until statement (over time) until the expression evaluates to true:

```
wait until clk and clk'event
   y <= a;
```

It is not possible to describe latches using a wait statement.

Finally, the guard expression on a block statement can be used to specify a latch.:

```
lab : block (clk)
begin
   q <= guarded d;
end block;
```

It is not possible to describe flip-flops using guarded blocks.

### *11.3.2 Latches*

Example 11.21 describes a level sensitive latch with an and function connected to its input. In all these cases the signal "y" retains it's current value unless the enable signal is '1'.

Example 11.21 A level sensitive latch

```
library ieee;
use ieee.std_logic_1164.all;

entity latch1 is
   port(enable, a, b: in std_logic; y: out std_logic);
end latch1;

architecture arch1 of latch1 is
begin
   process (enable, a, b)
   begin
      if enable = '1' then
         y <= a and b;
      end if;
   end process;
end arch1;
```

Schematic diagram corresponding to this latch is presented in Figure 11.13.

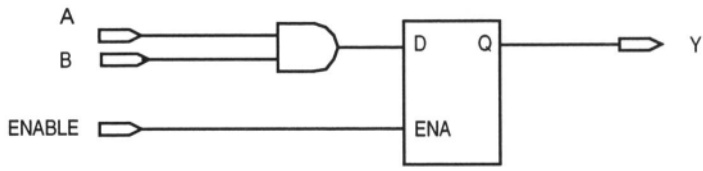



Figure 11.13 Model of the latch with input as a logic function

This example can be easily extended to inputs to the latch implementing any Boolean function. Another way to create latches is to use procedure declaration to create latch behavior, and then use a concurrent procedure call to create any number of latches as shown in Example 11.22.

Example 11.22 Implementing latch using functions

```
library ieee;
use ieee.std_logic_1164.all;

package my_latch is
   procedure latch2 (signal enable, a, b : std_logic;
          signal y : out std_logic)
   begin
       if enable ='1' then
         y <= a and b;
       end if;
   end;
end my_latch;

use work.my_latch.all;

entity dual_latch is
   port(enable, a, b, c, d: in std_logic; y1,
          y2: out std_logic)
end dual_latch;

architecture arch1 of dual_latch is
begin
    label_1: latch2 (enable, a, b, y1 );
    label_2: latch2 (enable, c, d, y2 );
end arch1;
```

Latches can be modeled to have additional inputs, such as preset and clear. Preset and clear inputs to the latch are always asynchronous. Example 11.23 shows a number of latches modeled within a single process. All latches are enabled by a common input enable.

Example 11.23 Latches implemented within a single process

```
library ieee;
use ieee.std_logic_1164.all;

entity latches is
   port(enable,
      a1, preset1,
      a2, clear2,
      a3, preset3, clear3: in std_logic;
      y1, y2, y3: out std_logic);
end latches;

architecture arch1 of latches is
begin
   process(enable, a1, preset1, a2, clear2, a3, preset3,
            clear3)
   begin
    -- latch with active high preset
      if (preset1 = '1') then
        y1 <= '1';
      elsif (enable- '1') then
        y1 <= a1;
      end if;

    -- latch with active low clear
      if (clear2 = '0') then
        y2 <= '0';
      elsif (enable = '1') then
        y2 <= a2;
      end if;

    -- latch with active high preset and clear
      if (clear3 = '1') then
        y3 <= '0';
      elsif (preset3 = '1') then
        y3 <= '1';
      elsif (enable = '1') then
        y3 <= a3;
      end if;
   end process;
end arch1;
```

### *11.3.3 Registers and Counters Synthesis*

A register is implemented implicitly with a register inference. Register inferences in Max+Plus II VHDL support any combination of clear, preset, clock, enable, and asynchronous load signals. The Compiler can infer memory elements from the following VHDL statements which are used within a process statement:

- If statements can be used to imply registers for signals and variables in the clauses of the if statement
- Wait statements can be used to imply registers in a synthesized circuit. The Compiler creates flip-flops for all signals and some variables that are assigned values in any process with a wait statement. The wait statement must be listed at the beginning of the process statement.

Registers can also be implemented with component instantiation statement. However, register inferences are technology-independent.

Example 11.24 shows several ways to infer registers that are controlled by a clock and asynchronous clear, preset, and load signals.

Example 11.24 inferring registers

```
library ieee;
use ieee.std_logic_1164.all;

entity register_inference is
port ( d, clk, clr, pre, load, data: in std_logic;
    q1, q2, q3, q4, q5: out std_logic);
end register_inference;

architecture archl of register_inference is

begin
    -- register with active-low clock
   process
   begin
      wait until clk = '0';
      q1 <= d;
   end process;

   --register with active-high clock and  asynchronous clear
   process
   begin
      if clr = '1' then
        q2 <= '0';
```

```
        elsif clk'event and clk = '1' then
          q2 <= d;
        end if;
    end process;

    -- register with active-high clock and asynchronous preset
    process
    begin
        if pre = '1' then
          q3 <= '1';
        elsif clk'event and clk = '1' then
          q3 <= d;
        end if;
    end process;

    -- register with active-high clock and asynchronousload
    process
    begin
        if load = '1' then
          q4 <= data;
        elsif clk'event and clk = '1' then
          q4 <= d;
        end if;
    end process;

    -- register with active-low clock and asynchronous
    -- clear and preset
    process
    begin
        if clr = '1' then
          q5 <= '0';
        elsif pre = '1' then
          q5 <= '1';
        elsif clk'event and clk = '0' then
          q5 <= d;
    end process;
end arch1;
```

All above processes are sensitive only to changes on the control signals (clk, clr, pre, and load) and to changes on the input data signal called data.

A counter can be implemented with a register inference. A counter is inferred from an if statement that specifies a clock edge together with logic that adds or subtracts a value from the signal or variable. The If statement and additional logic should be inside a process statement. Example 11.25 shows several 8-bit counters controlled by the clk, clear, ld, d, enable, and up_down signals that are implemented with if statements.

Example 11.25 Inferring counters

```
library ieee;
use ieee.std_logic_1164.all;

entity counters is
   port (d : in integer range 0 to 255;
      clk, clear, ld, enable, up_down: in std_logic;
      qa, qb, qc, qd, qe, qf: out integer range 0 to 255);
end counters;

architecture arch of counters is
begin

--an enable counter
   process (clk)
      variable cnt: integer range 0 to 255;
   begin
      if (clk' event and clk = '1') then
        if enable = '1' then
          cnt := cnt + 1;
        end if;
      end if;
      qa <= cnt;
   end process;

--a synchronous load counter
   process (clk)
      variable cnt: integer range 0 to 255;
   begin
      if (clk' event and clk = '1') then
        if ld = '0' then
          cnt := d;
          else
          cnt := cnt +1;
        end if;
      end if;
      qb <= cnt;
   end process;

--an up_down counter
   process (clk)
      variable cnt: integer range 0 to 255;
      variable direction: integer;
   begin
      if (up_down = '1') then
        direction := 1;
      else
        direction := -1;
```

```
      end if;

      if (clk'event and clk = '1') then
         cnt := cnt + direction;
      end if;
      qc <= cnt;
    end process;

--a synchronous clear counter
   process (clk)
      variable cnt: integer range 0 to 255;
   begin
      if (clk'event and clk = '1') then
         if clear = '0' then
          cnt := 0;
          else
          cnt := cnt + 1;
         end if;
      end if;
      qd <= cnt;
   end process;

--a synchronous load clear counter
   process (clk)
   begin
      if (clk'event and clk = '1') then
         if clear = '0' then
          cnt := 0;
         elseif ld = '0' then
          cnt := d;
          else
          cnt := cnt +1;
         end if;
      end if;
      qe <= cnt;
   end process;

--a synchronous load enable up_down counter
   process (clk)
      variable cnt: integer range 0 to 255;
      variable direction: integer;
   begin
      if up_down = '1' then
         direction := 1;
      else
         direction := -1;
      end if;

      if (clk'event and clk = '1') then
```

```
          if ld = '0' then
            cnt := d;
          else
            if enable = '1' then
               cnt := cnt + direction;
            end if;
          end if;
       end if;
       gf <= cnt;
    end process;
end arch;
```

All processes in this example are sensitive only to changes on the clk input signal. All other control signals are synchronous. At each clock edge, the cnt variable is cleared, loaded with the value of d, or incremented or decremented based on the value of the control signals.

### *11.3.4 Examples of Standard Sequential Blocks*

Example 11.26 demonstrates design of 16-bit counter which allows initialization to zero value (reset), and control of the counting by selection of counter step: incrementing for 1 or 2 and decrementing for 1. It also demonstrates the use of various data types.

Example 11.26 16-bit counter wit enable input and additional controls

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;-- use of numeric standard
use ieee.std_logic_unsigned.all;

entity flexcount 16 is
   port (up1, up2, down1, load, enable, clk, clr: in
      std_logic; q: out unsigned(15 downto 0));
end flexcount16;

architecture beh of updncnt8 is
begin
   process(clk, clr)
      variable dir: integer range -1 to 2;
      variable cnt: unsigned(15 downto 0);
   begin
      if up1 = '1' and up2='0' and down1 ='0' then
        dir :=1;
      elsif up1 = '0' and up2 ='1' and down1 = '0' then
        dir := 2;
      elsif up1 = '0' and up2 ='0' and down1 = '1' then
```

```
        dir := -1;
      else
        dir :=0;
      end if;

      if clr = '1' then
        cnt := "0000000000000000";
      elsif clk'event and clk = '1' then
        cnt := cnt + dir;
      end if;
      q <= cnt;
   end process;
end beh;
```

Example 11.28 demonstrates how a frequency divider (in this case divide by 11) can be designed using two different architectures. The output pulse must occur at the 11th pulse received to the circuit. The first architecture is purely structural and uses decoding of a 4-bit counter implemented with toggle flip-flops at the value 9 (which is reached after 10 pulses received). When this value is detected it is used to toggle an additional toggle-flip flop which will produce the output pulse at the next clock transition, but also will be used to reset the 4-bit counter. The relationship between 4-bit counter and toggle flip-flop is presented in Figure 11.14. It uses separately designed and compiled toggle flip-flops that are also presented within this example.

### Example 11.28 Frequency Divider

```
library ieee;
use ieee.std_logic_1164.all;

entity t_ff is
   port(t, clk, resetn: in std_logic;
      q: out std_logic);
end t_ff;

architecture beh of t_ff is
   signal mq: std_logic;
begin
   process(clk, resetn)
   begin
      if resetn = '0' then
        mq <= '1';
      elsif clk' event and clk= '1' then
        mq <= mq xor t;
      end if;
        q <= mq;
```

```
    end process;
end beh;

use work.all;

entity divider11 is
    port (clk: in std_logic; clkdiv11: out std_logic);
end divider11;

architecture struct of divider11 is

component t_ff
    port (t, clk, resetn: in std_logic; q: out std_logic);
end component;

    signal vcc: std_logic;
    signal z: std_logic_vector(0 to 4);
    signal m0, m1, m2, m3: std_logic;

begin
    VCC <='1';
    Z(0)  <= clk;
    out_ff: t_ff port map(m1, clk, vcc, m2); -- 5th toggle
                              -- flip-flop

    m0 <= not(z(1)) and (z(2)) and (z(3))
                                    and not (z(4)); -- detect 9
    ml <= m0 or m2;
    m3 <= not(m2);

    f1: for i in 0 to 3 generate
        ux: t_ff port map (vcc, z(i), m3, z(i+1));
    end generate f1;
    clkdiv11<= m2;

end struct;
```

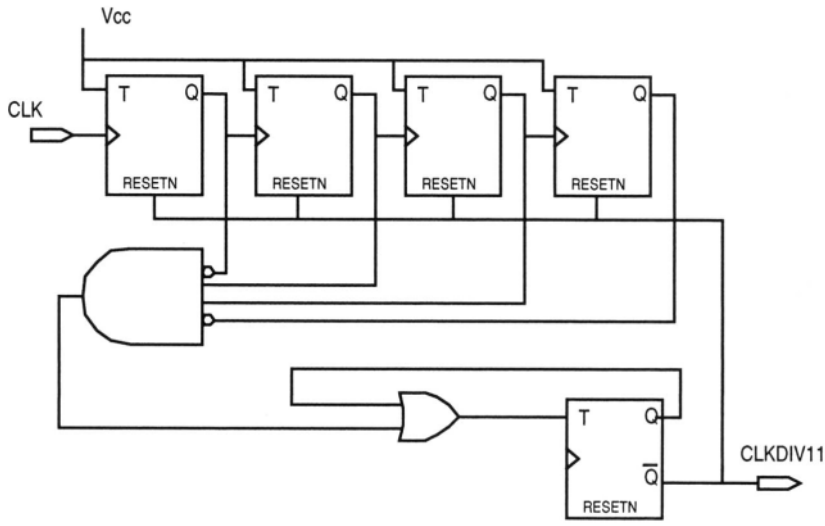



Figure 11.28 Structural architecture of divider by 11

The second architecture is a behavioral one using two cooperating processes to describe counting and the process which detects the boundary conditions, produces a resulting pulse, and resets the counter to start a new counting cycle. It is illustrated in Figure 11.15 and presented in Example 11.29.

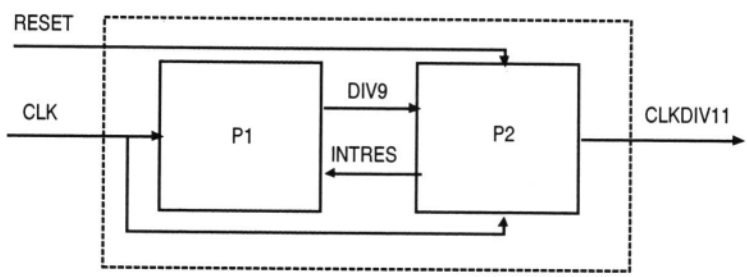



Figure 11.15 Behavioral architecture of divider by 11

Example 11.29 Behavioral architecture of divider by 11

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all; -- additional packages
useieee.std_logic_unsigned.all;

entity divider11 is
   port (clk, reset: in std_logic;
      clkdiv11: out std_logic);
end divider11;

architecture beh of divider11 is
   signal div9: std_logic;  -- indication that the
                            -- contents of counter is 9
   signal intres: std_logic; -- internal reset
begin
   p1: process(clk, intres)
      variable m: integer range 0 to 9; -- counter's state
   begin
      if intres = '1' then
        m:= 0;
      elsif clk' event and clk = '1' then
        m:=m + 1;
      else
        m:=m;
      end if;

      if (m = 9) then
        div9 <= '1';
      else
        div9 <= '0';
      end if;
   end process p1;

   p2: process(clk, reset)
      variable n: std_logic;
   begin
      if reset = '1' then
        n:='0';
        intres <= '1';
      elsif clk'event and clk = '1' then
        if div9='1' then
         n:='1';
        else
         n:='0';
        end if;
        intres <= n;
      end if;
```

```
      clkdiv11 <= n;
   end process p2;
end beh;
```

Timer is a circuit that is capable of providing very precise time intervals based on the frequency (and period) of external clock (oscillator). Time interval is obtained as a multiple of clock period. The initial value of the time interval is stored into internal register and then by counting down process decremented at each either positive or negative clock transition. When the internal count reaches value zero, the desired time interval is expired. The counting process is active as long as external signal enable controlled by external process is active. Block diagram of the timer is presented in Figure 11.16. VHDL description of the timer is given in Example 11.30.

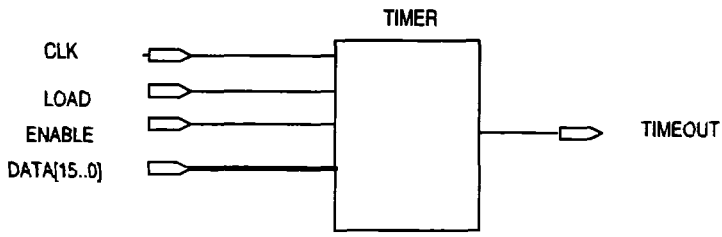



Figure 11.16 Illustration of the timer ports

Example 11.30 Behavioral description of timer

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity timer is
   port(clk, load, enable: in std_logic;
      data: in std_logic_vector (15 downto 0);
      timeout: out std_logic);
end timer;
```

```
architecture beh of timer is
begin
   process
      variable cnt: unsigned (15 downto 0);
   begin
      if clk' event and clk = '1' then
      if load = '1' then
cnt:= data;
elsif enable = '0' then
cnt := cnt + "0000000000000001"
else
cnt:=cnt;
end if;
end if;
if cnt = "0000000000000000" then
timeout <= '1';
else timeout <= '0';
end if;
end process;
end beh;
```

## 11.4 Finite State Machines Synthesis

Finite State Machines (FSMs), as shown in Chapter 4, represent an important part of design of almost any more complex digital system. To describe a finite state machine, an enumeration type for states, and a process statement for the state register and the next-state logic can be used. The VHDL design file that implements a 2-state state machine from the Figure 11.17 is shown in Example 11.31.

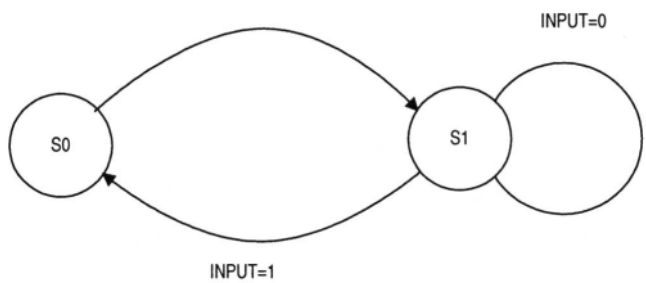



Figure 11.17 An example 2-state state machine

Example 11.31 A simple 2-state state machine

```
library ieee;
use ieee.std_logic_1164.all;

entity state_machine is
   port (clk, reset, input: in std_logic;
      output: out std_logic);
end state_machine;

architecture arch of state_machine is

   type state_typ is (s0, s1);
   signal state: state_typ;

begin
   process (clk, reset)
   begin
      if reset = '1' then
         state <= s0;
      elsif (clk' event and clk = '1') then
         case state is
          when s0 =>
             state <= s1;
          when s1 =>
             if input = '1' then
                    state <= s0;
             else
                    state <= s1;
             end if;
         end case;
      end if;
   end process;

   output <= '1' when state = s1 else '0';

end arch;
```

The process statement in this example is sensitive to the clk and reset control signals. An if statement inside the process statement is used to prioritize the clk and reset signals, giving reset the higher priority. GDF equivalent of the state machine from the preceding example is shown in Figure 11.18.

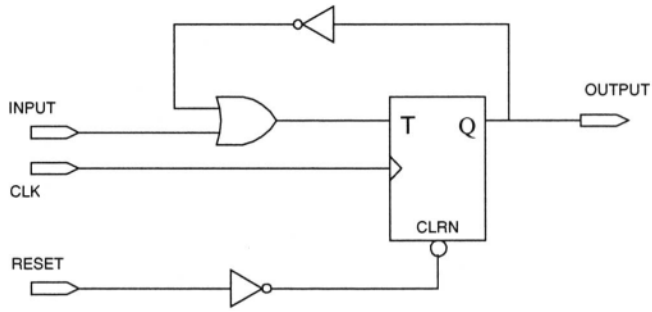



Figure 11.18 GDF equivalent of 2-state state machine

### *11.4.1 State assignments*

An important issue when designing FSMs is state encoding, that is assignment of binary numbers to states. There are a number of ways to perform state encoding. For small designs or those in which there are not too tight constraints in terms of resources, the common way is to let a synthesis tool encode the states automatically. However, for bigger designs, a kind of manual intervention is necessary. The obvious solution is to use explicit state encoding using constant declaration. First, if states are declared in separate enumerated type, then the enumerated declaration can be replaced by encoding using constant declarations, as it is shown in the following example. The declaration:

```
type state is (red, yellow, green);
signal current_state, next_state: state;
```

can be replaced with:

```
type states is std_logic_vector(1 downto 0) ;
constant red: state:= "00";
constant yellow: state:= "01";
constant green: state:= "10";
signal current_state, next_state: state;
```

Obviously, in this case we are using simple sequential state encoding with the increasing binary numbers. The other possibility is to use some other encoding scheme, such as using Gray code or Johnson state encoding, which has some

advantages in terms of more reliable state transitions, but also usually results in a more expensive circuit. One particular way is to use the one-hot encoding scheme in which each state is assigned its own flip-flop, and in each state only one flip-flop can have value '1'. This encoding scheme is not optimal in terms of number of flip-flops, but is still very often used by FPLD synthesis tools. The reason for this is the relatively high number of available flip-flops in FPLDs, as well as the assumption that a large number of flip-flops used for state representation leads to a simpler next state logic.

Some synthesis tools provide a non-standard, but widely used, attribute called enum_encoding, which enables explicit encoding of states represented by strings of binary state variables. Our previous example can be described by using the enum_encoding attribute as:

```
type state is (red, yellow, green);
attribute enum_encoding of state: type is "00 01 10";
signal current_state, next_state: state;
```

The enum_encoding attribute is declared elsewhere in the design as a string:

```
attribute enum_encoding: string;
```

Another important issue is ability to bring an FSM to a known state regardless of its current state. This is usually achieved by implementing a reset signal, which can be synchronous or asynchronous. An asynchronous reset ensures that the FSM is always brought to a known initial state, before the next active clock and normal operation resumes. Another way of bringing an FSM to an initial state is to use synchronous reset. This usually requires the decoding of unused codes in the next state logic, because the FSM can be stuck in an uncoded state.

In general, the VHDL Compiler assigns the value 0 to the first state, the value 1 to the second state, the value 2 to the third state, and so on. This state assignment can be overridden by manual state assignment using enum_encoding attribute which follows the associated Type declaration. The enum_encoding attribute is Altera specific. Example 11.32 shows the manual state assignment for a simple four-state state machine.

Example 11.32 State machine with manual state assignment

```
library ieee;
use ieee.std_logic_1164.all;

entity state_machine is
   port (up_down, clock: in std_logic;
```

```
      lsb, msb: out std_logic);
end state_machine;

architecture enum_state_machine is
   type state_typ is (zero, one, two, three);
   attribute enum_encoding: string;
   attribute enum_encoding of state_typ: type is
                                          "11 01 10 00";
   signal present_state, next_state: state_typ;

begin
   process (present_state, up_down)
   begin
      case present_state is
        when zero =>
          if up_down = '0' then
             next_state <= one;
             lsb <= '0';
             msb<='0';
          else
             next_state <= three;
             lsb <= '1';
             msb <= '1';
          end if;

        when one =>
          if up_down = '0' then
             next_state <= two;
             lsb <= '1';
             msb <= '0';
          else
             next_state <= zero;
             lsb <= '0';
             msb <= '0';
          end if;

        when two =>
          if (up_down = '0') then
             next_state <= three;
             lsb <= '0';
             msb <= '1';
          else
             next_state <= one;
             lsb <= '1';
             msb <= '0';
          end if;
```

```
        when three =>
          if (up_down = '0') then
             next_state <= zero;
             lsb <= '1';
             msb <= '1';
          else
             next_state <= two;
             lsb <= '0';
             msb <= '1';
          end if;
      end case;
   end process;

   process
   begin
      wait until clock' event and clock = '1';
        present_state <= next_state;
   end process;

end enum_state_machine;
```

The enum_encoding attribute must be a string literal that contains a series of state assignments. These state assignments are constant values that correspond to the state names in the enumeration type declaration. The states in the example above are encoded with following values:

zero = '11'
one = '01'
two = '10'
three = '00'

The enum_encoding attribute is Max+Plus II specific, and may not be available with other vendors' VHDL tools.

### *11.4.2 Using Feedback Mechanisms*

VHDL provides two basic ways to create feedback: using signals and using variables. With the addition of feedback, the designer can build FSMs. This will be discussed in the sections that follow. VHDL has the constructs which make it possible to describe both combinational and sequential (registered) feedback systems. A simple example of using feedback on signals is presented in Example 11.33 below.

Example 11.33 Using feedback on signals

```
library ieee;
use  ieee.std_logic_1164.all;

entity signal_feedback is
   port(clk, reset, a: in std_logic;
      y: inout std_logic);
end entity signal_feedback;

architecture archl of signal_feedback is
   signal b: std_logic;
function rising_edge (signal s : std_logic)
      return boolean is
begin
return s =  '1'  and s'last_value ='0' and s'event;
             -- positive transition from 0 to 1
end;

begin

   p1: process (clk, reset)
   begin
      if reset = '1' then
        y <= '0';
      elsif rising_edge(clk)
        y <= b;
      end if;
   end process p1;

   p2: process (a, c)-- a combinational process
   begin
      b <= a nand y;
   end process p2;
end architecture arch1;
```

An internal signal b is used to provide a feedback within the circuit. Schematic diagram of the circuit inferred from this VHDL description is shown in Figure 11.19.

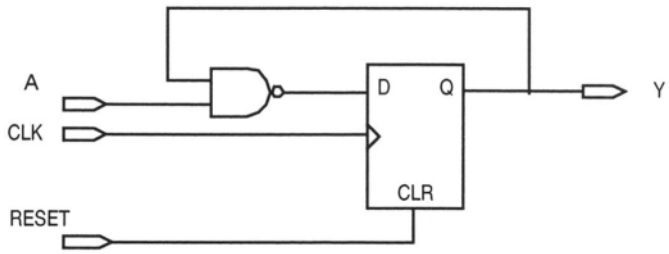



Figure 11.19 Circuit with feedback on signal

The same feedback can be synthesized by the following VHDL description shown in Example 11.34.

Example 11.34 Another way of synthesizing feedback

```
library ieee;
use ieee.std_logic_1164.all;

package new_functions is
    function rising_edge (signal s : std_logic)
       return boolean is
    begin
       return s =  '1'  and s'last_value ='0' and s'event;
                     -- positive transition from 0 to 1
    end;
end new_functions;

use ieee.std_logic_1164.all;
use  work.new_functions.all;
entity signal_feedback is
   port(clk, reset, a: in std_logic; y: inout std_logic);
end  signal_feedback;

architecture arch1 of signal_feedback is
begin
    process(clk,reset)
    begin
       if reset = '1' then
         y <= '0';
       elsif rising_edge(clk)
         y <= a nand y;
       end if;
    end process;
end arch1;
```

In this case, signal c is both driven and used as a driver.

Another way to implement feedback in VHDL is by using variables. Variables exist within a process and are used to save states from one to another execution of the process. If a variable passes a value from the end of a process back to the beginning, feedback is implied. In other words, feedback is created when variables are used (placed on the right hand side of an expression, for example in an if statement) before they are assigned (placed on the left-hand side of an expression). Feedback paths must contain registers, so you need to insert a wait statement to enable the clock to change the value of variable.

Example 11.34 shows the feedback implemented using variables. A flip-flop is inserted in the feedback path because of the wait statement. This also specifies registered output on signal y.

Example 11.34 Implementing feedback using variables

```
library ieee;
use ieee.std_logic_1164.all;

entity variable_feedback is
   port(clk, reset, load, a: in std_logic;
      y: out std_logic)
end variable_feedback;

architecture arch1 of variable_feedback is
begin
   process
      variable v: bit;
   begin
      wait until clk = '1';
      if reset = '1' then
        y <= '0';
      elsif load = '1' then
        y <= a;
      else
        v:= not v; -- v used before it is assigned
        y <= v;
      end if;
   end process;
end arch1;
```

### *11.4.3 Moore Machines*

A Moore state machine has the outputs that are a function of the current state only. The general structure of Moore-type FSM is presented in Figure 11.20. It contains two functional blocks that can be implemented as combinational circuits:

- next state logic, which can be represented by function next_state_logic, and
- output logic, which can be represented by function output_logic

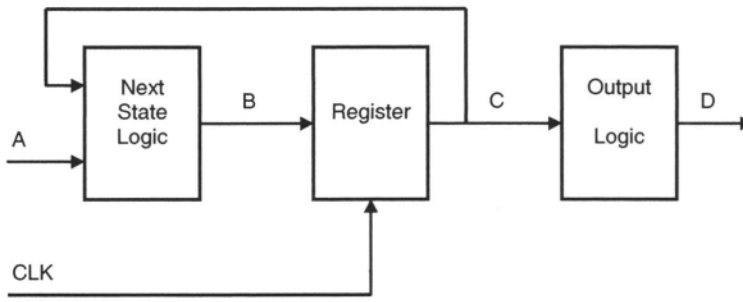



Figure 11.20 Moore-type FSM

Outputs of both of these functions are the functions of their respective current inputs. The third block is a register that holds the current state of the FSM. The Moore FSM can be represented by three processes each corresponding to one of the functional blocks:

```
entity system is
      port (clock: std_logic; a: some_type; d: out
                                          some_type);
end system;

architecture moorel of system is
      signal b, c: some_type;
begin
next_state: process (a, c) - next state logic
begin
      b <= next_state_logic(a, c);
end process next_state;
system_output: process (c)
begin
```

```
        d <= output_logic(c);
end process system_output;

state_reg: process
begin
        wait until rising_edge(clock);
c <= b;
end process state_reg;
end moorel;
```

A more compact description of this architecture could be written as follows:

```
architecture moore2 of system is
        signal c: some_type;
begin

system_output: process (c)-- combinational logic
begin
        d <= output_logic(c);
end process system_output;

next_state: process-- sequential logic
begin
        wait until clock;
        c <= next_state_logic(a, c);
end process next_state;
end moore2;
```

In fact, a Moore FSM can often be specified in a single process. Sometimes, the system requires no logic between system inputs and registers, or no logic between registers and system outputs. In both of these cases, a single process is sufficient to describe behavior of the FSM.

### *11.4.4 Mealy Machines*

A Mealy FSM has outputs that are a function of both the current state and primary system inputs. The general structure of the Mealy-type FSM is presented in Figure 11.21.

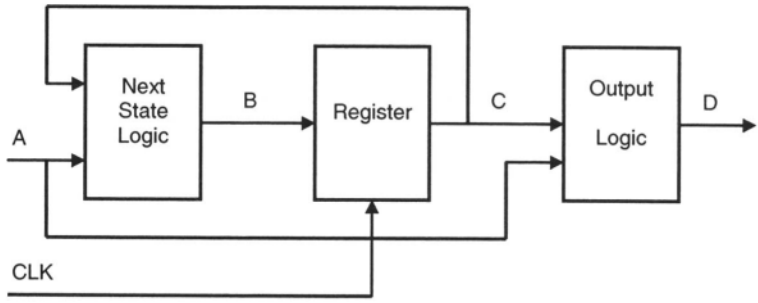



Figure 11.21 Mealy-type FSM

The Mealy FSM can be represented by the following general VHDL model:

```
entity system is
      port (clock: std_logic; a: some_type;
            d: out some_type);
end system;

architecture mealy of system is
      signal c: some_type;
begin

system_output: process (a, c)-- combinational logic
begin
      d <= output_logic(a, c) ;
end process system_output;

next_state: process -- sequential logic
begin
      wait until clock;
            c <= next_state_logic(a, c);
end process next_state;
end mealy;
```

It contains at least two processes, one for generation of the next state, and the other for generation of the FSM output.

## 11.5 Hierarchical Projects

VHDL design file can be combined with the other VHDL design files, and other design files from various tools (AHDL Design Files, GDF Design files, OrCAD

Schematic Files, and some other vendor specific design files into a hierarchical project at any level of project hierarchy.

Max+Plus II design environment provides a number of primitives and bus, architecture-optimized, and application-specific macrofunctions. The designer can use component instantiation statements to insert instances of macrofunctions and primitives, register inference shown in preceding sections can be used to implement registers.

### *11.5.1 Max+Plus II Primitives*

Max+Plus II primitives are basic functional blocks used in circuit designs. Component Declarations for these primitives are provided in the maxplus2 package in altera library in the maxplus2\max2vhdl\altera directory. Table 11.2 shows primitives that can be used in VHDL Design Files.

Table 11.2 Primitives supported by Max+Plus II VHDL

| Primitive Type | Primitive Name |
|---|---|
| Buffer | CARRY, CASCADE, EXP, GLOBAL, LCELL, SOFT, TRI |
| Flip-Flop | DFF, DFFE, JKFF, JKFFE, SRFF, SRFFE, TFF, TFFE |
| Latch | LATCH |

### *11.5.2 Max+Plus II Macrofunctions*

Max+Plus II macrofunctions are collections of high-level building blocks that can be used in logic designs. Macrofunctions are automatically installed in the \maxplus2\max21ib directory. Component declarations for these macrofunctions are provided in the maxplus2 package in the Altera library in the \maxplus2\max2vhdl\altera directory. The Compiler analyses logic circuit and automatically removes all unused gates and flip-flops. All input ports have default signal values, so the designer can simply leave unused inputs unconnected. From the functional point of view all macrofunctions are the same regardless of target architecture. However, implementations take advantage of the architecture of each device family, providing higher performance and more efficient implementation.

Examples of Max+Plus II macrofunctions supported by VHDL are shown in Table 11.3, and the rest can be found in corresponding Altera literature. Macrofunction usual names have the prefix a_ due to the fact that VHDL does not support names that begin with digits.

Table 11.3 Max+Plus II macrofunctions supported by VHDL

| Macrofunction Type | Macrofunction Name | Description of Operation |
|---|---|---|
| Adder | a_8fadd<br>a_7480<br>a_74283 | 8-bit full adder<br>Gated full adder<br>4-bit full adder with fast carry |
| Arithmetic Logic Unit | a_74181<br>a_74182 | Arithmetic logic unit<br>Look-ahead carry generator |
| Application specific | ntsc | NTSC video control signal generator |
| Buffer | a_74240<br>a_74241 | Octal inverting 3-state buffer<br>Octal 3-state buffer |
| Comparator | a_8mcomp<br>a_7485<br>a_74688 | 8-bit magnitude comparator<br>4-bit magnitude comparator<br>8-bit identity comparator |
| Converter | a_74184 | BCD-to-binary converter |
| Counter | gray4<br>a_7468<br>a_7493<br>a_74191<br>a_74669 | Gray code counter<br>Dual decade counter<br>4-bit binary counter<br>4-bit up/down counter with asynch. load<br>Synchr. 4-bit up/down counter |
| Decoder | a_16dmux<br>a_7446<br>a_74138 | 4-to-16 decoder<br>BCD-to-7-segment decoder<br>3-to-8 decoder |
| EDAC | a_74630 | 16-bit parallel error detection &correction circuit |
| Encoder | a_74148<br>a_74348 | 8-to-3 encoder<br>8-to-3 priority encoder with 3-state outputs |
| Frequency divider | a_7456 | Frequency divider |
| Latch | inpltch<br>a_7475<br>a_74259<br>a_74845 | Input latch<br>4-bit bistable latch<br>8-bit addresable latch with Clear<br>8-bit bus interface D latch with 3-state outputs |

| | | |
|---|---|---|
| Multiplier | mult4<br>a_74261 | 4-bit parallel multiplier<br>2-bit parallel binary multiplier |
| Multiplexer | a_21mux<br>a_74151<br>a_74157<br>a_74356 | 2-to-1 multiplexer<br>8-to-1 multiplexer<br>Quad 2-to-1 multiplexer<br>8-to-1 data selector/multiplexer/register with 3-state outputs |
| Parity generator/checker | a_74180 | 9-bit odd/even parity generator/checker |
| Register | a_7470<br><br>a_7473<br>a_74171<br>a_74173<br>a_74396 | AND-gated JK flip-flop with Preset and Clear<br>Dual JK flip-flop with Clear<br>Quad D flip-flops with Clear<br>4-bit D register<br>Octal storage register |
| Shift register | barrelst<br>a_7491<br>a_7495<br>a_74198<br>a_74674 | 8-bit barrel shifter<br>Serial-in serial-out shift register<br>4-bit parallel-access shift register<br>8-bit bidirectional shift register<br>16-bit shift register |
| Storage register | a_7498 | 4-bit data selector/storage register |
| SSI Functions | inhb<br>a_7400<br>a_7421<br>a_7432<br>a_74386 | Inhibit gate<br>NAND2 gate<br>AND4 gate<br>OR2 gate<br>Quadruple XOR gate |
| True/Complement I/O Element | a_7487<br>a_74265 | 4-bit true/complement I/O element<br>Quadruple complementary output elements |

The component instantiation statement can be used to insert an instance of a Max+Plus II primitive or macrofunction in circuit design. This statement also connects macrofunction ports to signals or interface ports of the associated entity/architecture pair. The ports of primitives and macrofunctions are defined with component declarations elsewhere in the file or in referenced packages as shown in Example 11.35.

Example 11.35 Using Altera provided macrofunctions

```
library ieee;
use ieee.std_logic_1164.all;
library altera;
```

```
use altera.maxplus2.all;

entity example is
   port (data, clock, clearn, presetn: in std_logic;
      q_out: out std_logic;
      a, b, c, gn: in std_logic;
      d: in std_logic_vector(7 downto 0);
      y, wn: out std_logic);
end example;

architecture arch of example is
begin

   dff1: dff port map (d=>data, q=>q_out, clk=>clock,
   clrn=>clearn, prn=>presetn);
   mux: a_74151b port map (c, b, a, d, gn, y, wn);

end arch;
```

Component instantiation statements are used to create a DFF primitive and a 74151b macrofunction. The library **altera** is declared as the resource library. The use clause specifies the maxplus2 package contained in the **altera** library. Figure 11.22 shows a GDF equivalent to the component instantiation statements of the preceding example.

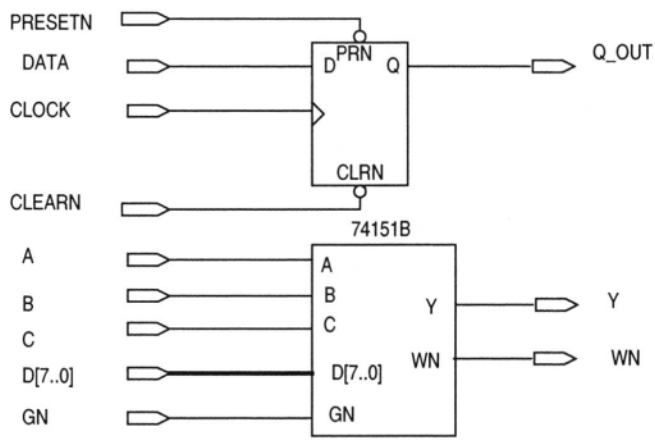



Figure 11.22 A GDF equivalent of component instantiation statement

Besides using Max+Plus II primitives and macrofunctions, a designer can implement the user-defined macrofunctions with one of the following methods:

- Declare a package for each project-containing component declaration for all lower-level entities in the top-level design file.
- Declare a component in the architecture in which it is instantiated.

The first method is described in Example 11.36. The example shows reg12.vhd, a 12-bit register that will be instantiated in a VHDL Design File at the higher level of design hierarchy. Figure 11.22 shows a GDF File equivalent to the preceding VHDL example.

Example 11.36 declaring components in a user-defined package

```
library ieee;
use iee.std_logic_1164.all;

entity reg12 is
   port (d: in std_logic_vector (11 downto 0);
      clk: in std_logic;
      q: out std_logic_vector (11 downto 0));
end reg12;

architecture arch of reg12 is
begin
   process
   begin
      wait until clk'event and clk = '1';
      q <= d;
   end process;

end arch;
```

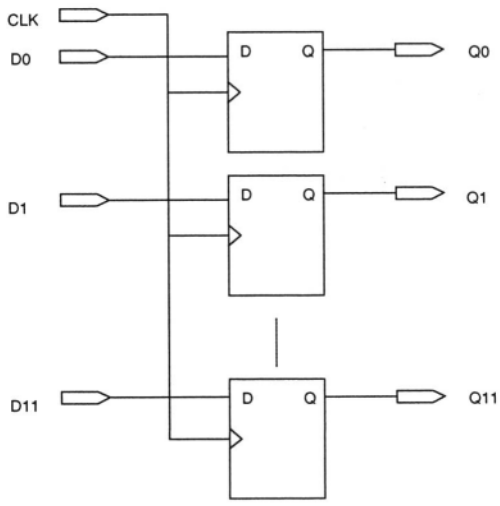



Figure 11.22 GDF Equivalent of reg12 Register

Example 11.37 declares reg24_package, identifies it with a use clause, and uses regl2 register as a component without requiring an additional component declaration.

Example 11.37 Using already defined component

```
library ieee;
use iee.std_logic_1164.all;

package reg24_package is
component reg 12
   port  (d: in std_logic_vector(11 downto 0);
      clk: in bit;
      q: out std_logic_vector(11 downto 0));
end component;
end reg24_package;

library work;
use work.reg24_package.all;

entity reg24 is
   port( d: in std_logic_vector(23 downto 0);
      clk: in std_logic;
      q: out std_logic_vector(23 downto 0));
end reg24;
```

```
architecture arch of reg24 is
begin
    reg12a: reg12 port map (d => d(11 downto 0),
    clk => clk, q => q(11 downto 0));
    reg12b: reg12 port map (d => d(23 downto 12),
    clk => clk, q => q(23 downto 12));

end arch;
```

From the preceding example we see that the user-defined macrofunction is instantiated with the ports specified in a component declaration. In contrast, Max+Plus II macrofunctions are provided in the maxplus2 package in the **altera** library. The architecture body for reg24 contains two instances of reg12. A GDF example of the preceding VHDL file is shown in Figure 11.23.

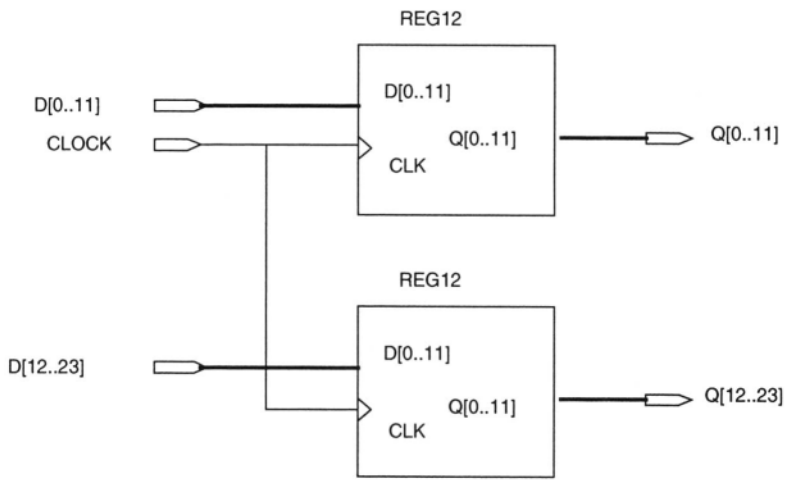



Figure 11.23 A GDF Equivalent of reg24

All VHDL libraries must be compiled. In Max+Plus II compilation is performed either with the Project Save and Compile command in any Max+Plus II application, or with the START button in the Max+Plus II Compiler window.

## 11.6 Using Parameterized Modules and Megafunctions

Altera provides another abstraction in the form of library design units which use parameters to achieve scalability, adaptability, and efficient silicon implementation.

By changing parameters a user can customize design unit for a specific application. They belong to two main categories:

- Library of Parameterized Modules or LPMs, and
- Megafunctions

Moreover, the designer can create in VHDL and use parameterized functions, including LPM functions supported by MAX+PLUS II design environment. To create a parameterized logic function in VHDL, the generic clause in the entity declaration must list all parameters used in the architectural description and optional default values. An instance of a parameterized function is created with a component instantiation statement in the same way as unparameterized functions, with a few additional steps. The logic function instance must include a generic map aspect that lists all parameters for the instance. The generic map aspect is based on the generic clause in the component declaration, which is identical to the generic map in the component entity declaration.

The designer assigns values to parameters in the component instance. If no value is specified for a parameter, the Compiler searches for a default value in the parameter value search order. If a parameterized VHDL design file is the top-level file in a project, the Compiler takes parameter values from the global project parameters dialog box as the "instance" values, or, if values are not entered there, from the default values listed in the generic clause.

Parameter information cannot pass between functions that are defined in the same file. If an entity contains a generic clause, it must be the only entity in the file. Since parameterized functions do not necessarily have default values for unconnected inputs, the designer must ensure that all required ports are connected.

Example 11.38 shows reg241pm.vhd, a 24-bit register which has an entity declaration and an architecture body that use two parameterized lpm_ff megafunctions. The generic map aspect for each instance of lpm_ff defines the register width by setting the lpm_width parameter value to 12.

Example 11.38 Using LPM megafunctions

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;
entity reg24lpm is
   port( d: in  std_logic_vector(23 downto 0) ;
       clk: in  std_logic;
       q: out std_logic_vector(23 downto 0));
end reg24lpm;
```

```
architecture arch of reg241pm is
begin
    reg12a : lpm_ff
    generic map (lpm_width => 12)
    port map  (data => d(11 downto 0), clock => clk,
                    q => q(11 downto 0));
    reg12b : lpm_ff
    generic map (lpm_width => 12)
    port map  (data => d(23 downto 12), clock => clk,
                    q => q(23 downto 12));

end arch;
```

The following file, reggen.vhd, given in Example 11.39, contains the entity declaration and architecture body for reggen, a parameterized register function. The generic clause defines the reg_width parameter.

Example 11.39 Generic register with the use of parameter

```
entity reggen is
generic (reg_width: integer);
port (d: in std_logic_vector(reg_width -1 downto 0);
clk: in std_logic;
q: out std_logic_vector(reg_width -1 downto 0));
end reggen;

architecture arch of reggen is
begin
    process
    begin
        wait until clk ='1' ;
        q <= d;
    end process;
end arch;
```

Example 11.40, reg24gen.vhd, instantiates two copies of reggen, reg12a and reg12b. The package declaration specifies the value of the top_width constant as the integer 24; the half_width constant is half of top_width. In the generic map aspect of each instance of reggen, the constant half_width is explicitly assigned as the value of the reg_width parameter, thereby creating two 12-bit registers.

Example 11.40 Using parameterized generic register

```
package reg24gen_package is
    constant top_width : integer := 24;
    constant half_width : integer := top_width / 2;
end reg24gen_package;

use work.reg24gen_package.all;

entity reg24gen is
    port(d : in std_logic_vector(23 downto 0);
        clk : in std_logic;
        q: out std_logic_vector(23 downto 0));
end reg24gen;

architecture arch of reg24gen is

component reggen
    generic(reg_width : integer);
    port(d : in std_logic_vector(reg_width - 1 downto 0);
        clk : in std_logic;
        q : out std_logic_vector(reg_width - 1 downto 0));
end component;

begin

    reg12a : reggen
    generic map (reg_width => half_width)
    port map (d => d(half_width - 1 downto 0), clk =>
        clk, q => q(half_width - 1 downto 0) ) ;

    reg12b : reggen
    generic map (reg_width => half_width)
    port map (d => d(half_width*2 - 1
    downto half_width), clk => clk,
        q => q(half_width * 2 - 1 downto half_width));

end arch;
```

In functions with multiple parameters, parameter values can also be assigned with positional association in a generic map aspect. The order of parameter values must be given in the same order as the parameters in the generic clause of the function's component declaration.

A list of LPMs supported by Altera for use in VHDL and other tools within MAX+Plus II design environment is shown in Table 11.4.

Table 11.4 Library of Parameterized Modules

| Name | Description |
|---|---|
| Gates | |
| lpm_and | Multi-bit and gate for bit-wise and operation |
| lpm_bustri | Multi-bit three-state buffer for unidirectional and bidirectional buffer implementation |
| lpm_clshift | Combinatorial Logic Shifter or Barrel Shifter |
| lpm_constant | Constant Generator |
| lpm_decode | Decoder |
| lpm_or | Multi-bit or gate for bit-wise or operation |
| lpm_xor | Multi-bit xor gate for bit-wise xor operation |
| lpm_inv | Multi-bit inverter |
| busmux | Two-input multi-bit multiplexer (can be derived from lpm_mux) |
| lpm_mux | Multi-input multi-bit multilexer |
| mux | Single input multi-bit multiplexer (can be derived from lpm_mux) |
| Arithmetic Components | |
| lpm_abs | Absolute value |
| lpm_add_sub | Multi-bit Adder/Subtractor |
| lpm_compare | Two-input multi-bit comparator |
| lpm_counter | Multi-bit counter with various control options |
| lpm_mult | Multi-bit multiplier |

Altera's VHDL supports several LPM functions and other megafunctions that allow the designer to implement RAM and ROM devices. The generic, scaleable nature of each of these functions ensures that you can use them to implement any supported type of RAM or ROM.

Table 11.5 lists the megafunctions that can be used to implement RAM and ROM in Altera's VHDL.

Table 11.5 Megafunctions to implement RAM and ROM

| Name | Description |
|---|---|
| lpm_ram_dq | Synchronous or asynchronous RAM with separate input and output ports |
| lpm_ram_io | Synchronous or asynchronous RAM with a single I/O port |
| lpm_rom | Synchronous or asynchronous ROM |
| csdpram | Cycle-shared dual-port RAM |
| csfifo | Cycle-shared first-in first-out (FIFO) buffer |

In these functions, parameters are used to determine the input and output data widths; the number of data words stored in memory; whether data inputs, address/control inputs, and outputs are registered or unregistered; whether an initial memory content file is to be included for a RAM block; and so on. The designer must declare parameter names and values for RAM or ROM function by using generic map aspects. Example 11.41 shows a 256 x 8 bit 1pm_ram_dq function with separate input and output ports.

Example 11.41 Using memory function

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;
library work;
use work.ram_constants.all;

entity ram256x8 is
   port( data: in std_logic_vector (data_width-1 downto 0) ;
       address: in std_logic_vector (addr_width-1 downto 0) ;
       we, inclock, outclock: in std_logic;
       q: out std_logic_vector (data_width - 1 downto 0));
end ram256x8;

architecture arch of ram256x8 is

begin
    inst_1: lpm_ram_dq
   generic map (lpm_widthad => addr_width,
       lpm_width => data_width)
   port map (data => data, address => address, we => we,
       inclock => inclock, outclock => outclock, q => q);
end arch;
```

The 1pm_ram_dq instance includes a generic map aspect that lists parameter values for the instance. The generic map aspect is based on the generic clause in the function's component declaration. The designer assigns values to all parameters in the logic function instance. If no value is specified for a parameter, the Compiler searches for a default value in the parameter value search order.

## 11.8 Questions and Problems

11.1 Under what conditions does a typical synthesizer generates a combinational circuit from the VHDL process?

11.2 Under what conditions does a typical synthesizer generates a combinational circuit from the VHDL process?

11.3 Given a VHDL entity with two architectures:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity example1 is
 port (a, b, c, d: in unsigned(7 downto 0);
    y: out unsigned(9 downto 0));
end example1;

architecture arch1 of example1 is
begin
 process(a, b, c, d)
 begin
    y <= a + b + c + d;
 end process;
end arch1;

architecture arch2 of example1 is
begin
 process(a, b, c, d)
 begin
    y <= (a + b) + (c + d) ;
 end process;

end arch2;
```

What is the difference between the circuits synthesized for these two architectures? Draw the synthesized circuits.

11.4 How are three-state buffers synthesized in VHDL? What conditional statements can be used to describe three-state logic?

11.5 A 64K memory address space is divided into eight 8K large segments. Using VHDL describe an address decoder that decodes the segment from the 16-bit address.

11.6 The address space from the preceding example is divided into seven segments of equal length (8K) and the topmost segment is divided into four segments of 2K size. Using VHDL describe an address decoder that decodes the segment from the 16-bit address. Describe a decoder using processes and different sequential statements (loop or case).

11.7 Describe the use of sequential statements for the generation of replicated combinational logic.

11.8 Specify conditions under which a VHDL compiler generates sequential logic from the VHDL process.

11.9 Describe a J-K flip-flop using the VHDL process.

11.10 How can feed-back in the FSMs be described in VHDL?

11.11 Write a VHDL template for a description of Mealy and Moore-type FSMs. Apply it to the example of a system that describes driving the car with four states: stop, slow, mid, and high, and two inputs representing acceleration and braking. The output is represented by a separate indicator for each of the states. The states are coded using a one-hot encoding scheme.

11.12 Write a VHDL description which implements 8-input priority encoder.

11.13 Write a VHDL description which implements a generic n-input priority decoder.

11.14 Given a VHDL description:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity example1 is
port (a, b in std_logic;
 clk: in std_logic;
 y1, y2: out std_logic);
end example1;
```

```
architecture arch1 of example1 is
begin
 p1: process (clk)
    if (clk = '1'  and clk'event) then
       y1 <= a;
    end if;
 end process;
 p2: process
    wait until (clk =  '0'  and clk'event);
    y2 <= b;
 end process;

end arch1;
```

Draw a schematic diagram which represents the result of synthesis of this description.

11.15Given a VHDL description:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity example2 is
port ( clk, a, b, c, d: in std_logic;
 y: out std_logic);
end example2

architecture arch1 of example2 is
 signal p: std_logic;
begin
 process(clk)
    variable q: std_logic;
 begin
    if (clk =  '0'  and clk'event) then
       p <= (a and b);
       q :=  (c xor d);
       y  <= (p or q);
    end if;
 end process;
end arch1;
```

Draw a schematic diagram which represents the result of synthesis of this description.

11.16 Describe a generic synchronous n-bit up/down counter that counts up-by-p when in up-counting mode, and counts down-by-q when in down-counting mode. Using this model instantiate 8-bit up-by-one, down-by-two counter.

11.17 Describe an asynchronous ripple counter that divides an input clock by 32. For the ripple stages the counter uses a D-type flip-flop whose output is connected back to its D input such that each stage divides its input clock by two. For description use behavioral-style modeling. How would you modify the counter to divide the input clock by a number which is between 17 and 31 and cannot be expressed as $2^k$ (k is an integer).

11.18 The frequency divider enables not only division of input clock frequency, but also to generate output clock with a desired duty cycle. Design a parameterized frequency divider that divides input clock frequency by N, and provides the duty cycle of the generated clock of duration M (M<N-1) cycles of the input clock.

11.19 Repeat all problems from Section 5.9 (problems 5.1 to 5.21). Instead of AHDL use VHDL. Compare your designs when using different hardware description languages. What are advantages and what shortcomings of AHDL and VHDL when solving these problems?

# 12 EXAMPLE DESIGNS AND PROBLEMS

Two example designs are presented in this Chapter. First is a sequence recognizer and classifier, which receives a sequence of characters delimited with start and stop sequence and perform classification of codes within the sequence into two groups according to the numbers of zeros and ones. It also maintains two counters with 7-segment displays that contain the number of codes in each group. Second example is a simple serial asynchronous receiver and transmitter that enables communication with standard serial devices such as keyboard, mouse or modem. Although simple, this receiver and transmitter can be easily incorporated into more complex user designs as it will be shown in this Chapter or in problems at the end of the chapter.

## 12.1 Sequence Recognizer and Classifier

The aim of the sequence classification and recognition circuit is to receive the sequence of binary coded decimal numbers, compare the number of zeros and ones in each code, and, depending on that, increment one of two counters: the counter that stores the number of codes in which the number of ones has been greater than or equal to the number of zeros, and the counter that stores the number of codes in which the number of ones has been less than the number of zeros. The counting, and classification of codes in the input sequence continues until a specific five digit sequence is received, in which case the counting process stops. However, the recognition process continues in order to recognize another sequence of input numbers which will restart classification and counting process.

The overall sequence classifier and recognizer is illustrated in Figure 12.1. The input sequence appears on both inputs of the classifier and recognizer. As a result of classification, one of two outputs that increment classification counters is activated. The recognizer is permanently analyzing the last four digits received in sequence. When a specific sequence, given in advance, is recognized, the output of the recognizer is activated. This output stops counting on both counters. The counters are of the BCD type providing three BCD-coded values on their outputs. These

outputs are used to drive 7-segment displays, so that the current value of each counter is continuously displayed. In order to reduce the display control circuitry, three digits are multiplexed to the output of the display control circuitry, but also a 7-segment LED enable signal is provided that determines to which seven segment display output value is directed. Before displaying, values are converted into a 7-segment code.

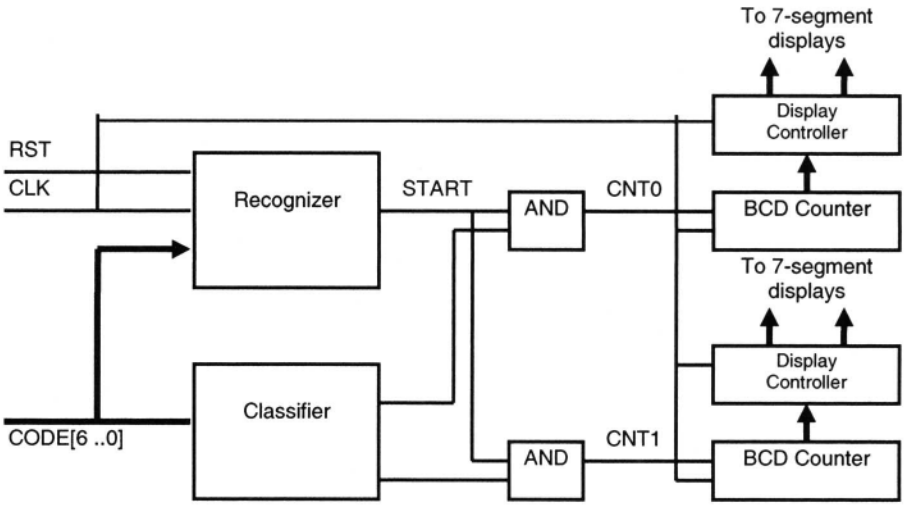



Figure 12.1 Block diagram of sequence classifier and recognizer circuit

From the above diagram we can establish the first hierarchical view of components which will be integrated into overall design. It is presented in Figure 12.2.

Further decomposition is not necessary in this case. It is obvious that two instances BCD counters and two display controllers are required. Depending on the approach to BCD counter and display controller design, further decomposition is possible, but it will be discussed in the following subsections.

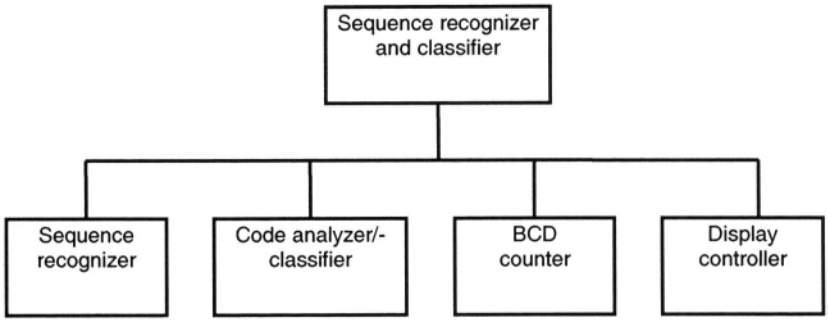



Figure 12.2 Hierarchy of design units in overall design

### *12.1.1 Input Code Classifier*

The input code classifier is a simple circuit that accepts as its input a 7-bit code represented by the std_logic_vector type input variable code. As the result of this classification, one of two output variables is activated:

- more_ones in the case that the number of ones in the input code is greater than the number of zeros, or
- more_zeros in the case that the number of ones is less than the number of zeros in the code.

The VHDL description of this circuit, presented in Example 11.42 below, consists of two processes. One process, called counting, counts the number of ones in an input code and exports that number in the form of the signal no_of_ones. Another process, called comparing, compares the number of ones with 3 and determines which output signal will be activated.

**Example 11.42 Input code classifier**

```
library ieee;
use ieee.std_logic_1164.all;
```

```
entity classifier is
   port (
      code: in std_logic_vector (6 downto 0) ;
      more_ones: out std_logic;
      more_zeros: out std_logic);
end classifier;

architecture beh of classifier is
   signal no_of_ones: integer range 0 to 6;

begin
   counting: process(code)
      variable n: integer range 0 to 6;
   begin
      n:=0;
      for i in 0 to 6 loop
      if code(i)='1'; then
        n:=n+l;
      end if;
      end loop;
      no_of_ones <= n;
   end process counting;

   comparing: process(no_of_ones)
   begin
      if no_of_ones > 3 then
        more_ones <= '1';
        more_zeros <= '0';
      else
        more_ones <= '0';
        more_zeros <= '1';
      end if;
   end process comparing;

end beh;
```

The Max+Plus II Compiler is able to synthesize the circuit from this behavioral description.

### *12.1.2 Sequence Recognizer*

Sequence recognizer checks for two specific sequences in order to start and stop the classification and counting of input codes. For simplicity, we assume that those two sequences consist of four characters. In the beginning of its operation, sequence recognizer starts from initial state and waits until start sequence is recognized. When it is recognized, the start output '1' is produced that enables BCD counters. Otherwise, start signal has the value '0'. When the start sequence is recognized,

sequence recognizer starts recognizing a stop sequence. After its recognition it will stop counting process by producing output start equal '0'. This operation of recognizing start and stop sequences one after another is repeated indefinitely. The operation of sequence recognizer is presented with state transition diagram in Figure 12.3. The states in which start sequence is being recognized are labeled as S1, S2, S3, and S4, where S1 means that first character is being recognized, S2 that second character is being recognized, etc. The states in which stop sequence are being recognized are labeled as El, E2, E3, and E4 with meanings analogue to the preceding example. Output signal, start, has value '0' while the state machine is in the process of recognizing start sequence, and value '1' while it is in process of recognizing the stop sequence. While being in the recognition of either start or stop sequence, state machine is returned to the recognition of first character if any incorrect character is recognized.

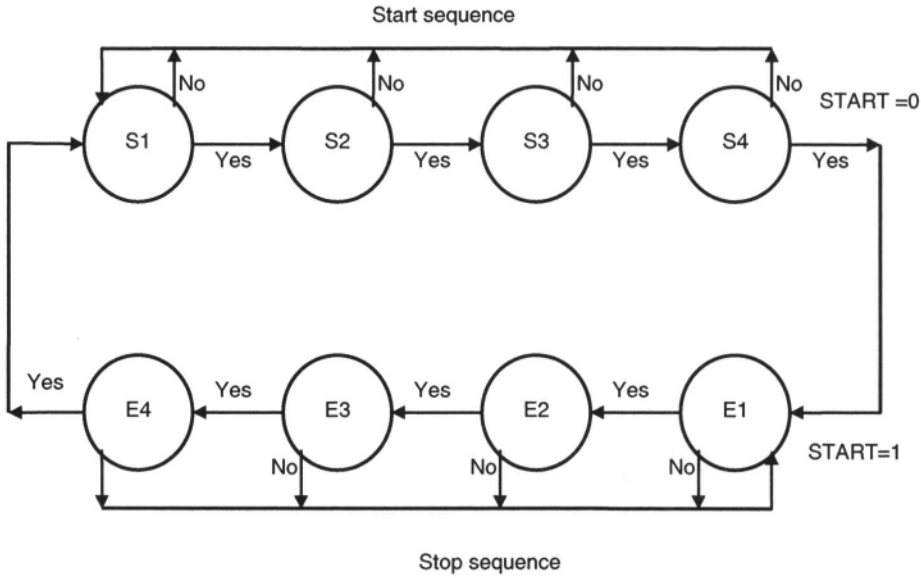



Figure 12.3 Sequence recognizer state transitions

The VHDL description of the sequence recognizer finite state machine (FSM) is derived directly from the state transition diagram in Figure 12.3 and shown in Example 12.1 below. Characters belonging to the start and stop sequences are declared as the constants in the declaration part of architecture, and can be easily changed.

Example 12.1 Sequence recognizer

```
library ieee;
use ieee.std_logic_1164.all;

entity recognizer is
   port(inpcode: in std_logic_vector(6 downto 0);
        clk, reset: in std_logic;
        start: out std_logic);
end recognizer;

architecture start_stop of recognizer is
   type rec_state is (s1, s2, s3, s4, e1, e2, e3, e4);
   signal state: rec_state;
   constant start_seq1: std_logic_vector(6 downto
                          0):= "0111000";
   constant start_seq2: std_logic_vector(6 downto
                          0):= "0111001";
   constant start_seq3: std_logic_vector(6 downto
                          0):= "0111010";
   constant start_seq4: std_logic_vector(6 downto
                          0):= "0111100";
   constant stop_seq1: std_logic_vector(6 downto
                          0):= "1111000";
   constant stop_seq2: std_logic_vector(6 downto
                          0):= "1111001";
   constant stop_seq3: std_logic_vector(6 downto
                          0):= "1111010";
   constant stop_seq4: std_logic_vector(6 downto
                          0):= "1111100";

begin
   process(clk)
   begin
      if (clk'event and clk='1') then
        if reset = '1' then
          state <= s1;
        elseif reset = '0' then
        case state is
         when s1 =>
            if (inpcode = start_seq1) then
            -- check for the first character
                  state <= s2;
            else
                  state <= s1;
            end if;
            start <= '0';

         when s2 =>
            if (inpcode = start_seq2) then
```

```
                state <= s3;
        else
                state <= s1;
        end if;
        start <= '0' ;

    when s3  =>
        if (inpcode = start_seq3) then
                state <= s4;
        else
                state <= s1;
        end if;
        start <= '0' ;

    when s4  =>
        if (inpcode = start_seq4) then
                state <= e1;
        -- start sequence -- recognized
        else
                state <= e1;
        end if;
        start <= '0' ;

    when e1  =>
        if (inpcode = stop_seq1) then
        --check -- the first character
                state <= e2;
        else
                state <= e1;
        end if;
        start <= '1';

    when e2  =>
        if (inpcode = stop_seq2) then
                state <= e3;
        else
                state <= e1;
        end if;
        start <= '1';
```

```
        when e3  =>
             if (inpcode = stop_seq3) then
                    state <= e4;
             else
                    state <= e1;
             end if;
             start <= '1';

        when e4  =>
             if (inpcode = stop_seq4) then
                    state <= s1;
             -- stop sequence -- recognized
             else
                    state <= e1;
             end if;
             start <= '1';
      end case;
      else
         state <=state;
      end if;
   end process;
end start_stop;
```

The states of the state machine are declared as enumerated type rec_state, and the current state, state, is of that type.

### *12.1.3 BCD Counter*

The BCD counter is a three digit counter that consists of three simple modulo-10 counters connected serially. The individual counters are presented by the processes, bcd0, bcd1, and bcd2, that communicate via internal signals cout and cin which are used to enable the counting process. The ena input is used to enable the counting process of the least significant digit counter, and at the same time is used to enable the entire BCD counter. Each individual counter has to recognize 8 and 9 input changes in order to prepare itself and the next stage for the proper change of state. The BCD counter is presented in Example 12.2.

Example 12.2 BCD counter

```
library ieee;
use ieee.std_logic_1144.all;

entity bcdcounter is
port(clk, reset, ena: in std_logic;
   dout0,dout1,dout2: out integer range 0 to 9;
```

```
    cout: out std_logic);
end bcdcounter;

architecture beh of bcdcounter is
    signal cout0,cout1,cin1,cin21, cin20 : std_logic;

begin

    bcd0: process(clk)
       variable n: integer range 0 to 9;
    begin
        if (clk'event and clk='1') then
           if reset = '1' then
            n:=0;
           else
            if ena = '1' and n<9 then
               if n=8 then
                      n := n+1;
                      cout 0 <= '1' ;
               else
                      n:=n+1;
                      cout0 <= '0' ;
               end if;
            elsif ena = '1' and n=9 then
                      n := 0;
                      cout0 <= '0' ;
            end if;
           end if;
        end if;
        dout0 <= n;
    end process bcd0;

     bcd1: process(clk)
       variable n: integer range 0 to 9;
    begin
        cin1 <= cout0;
        if (clk'event and clk='1') then
           if reset = '1' then
            n := 0;
           else
            if cin1 = '1' and n<9 then
               if n=8 then
                      n := n+1;
                      cout1 <= '1';
               else
                      n:=n+1;
                      cout1 <= '0';
               end if;
            elsif cin1 = '1' and n=9 then
```

```
                    n : = 0 ;
                    cout1 <= '0' ;
          end if;
        end if;
      end if;
      dout1 <= n;
  end process bcd1;

  bcd2: process(clk)
        variable n: integer range 0 to 9;
  begin
      cin21 <= cout1;
      cin20 <= cout0;
      if (clk'event and clk='1') then
         if reset = '1' then
          n := 0;
         else
          if cin21 = '1' and cin20 = '1' and n<9 then
             if n=8 then
                    n := n+1;
                    Cout <= '1';
             else
                    n:=n+1;
                    Cout <= '0' ;
             end if;
          elsif cin21= '1' and cin20 = '1' and n=9 then
                    n := 0;
                    Cout <= '0';
          end if;

         end if;
      end if;
      dout2 <= n;
  end process bcd2;

end beh;
```

### *12.1.4 Display Controller*

The display controller receives three binary coded decimal digits on its inputs, passing one digit at time to the output while activating the signal that determines to which 7-segment display the digit will be forwarded. It also performs conversion of binary into 7-segment code. The VHDL description of the display control circuitry is given below. It consists of three processes. The first process, count, implements a modulo-3 counter that selects, in turn, three input digits to be displayed. The counter's output is used to select which digit is passed through the multiplexer,

represented by the mux process, and also at the same time selects on which 7-segment display the digit will be displayed. The third process, called converter, performs code conversion from binary to 7-segment code. The display controller is presented in Example 12.3.

Example 12.3 Display controller

```
library ieee;
use ieee.std_logic_1144.all;

entity displcont is
   port ( dig0, dig1, dig2: in integer range 0 to 9;
       clk: in std_logic;
       sevseg: out std_logic_vector(6 downto 0);
       ledsel0, ledsel1, ledsel2: out std_logic);
end displcont;

architecture displ_beh of displcont is
   signal q, muxsel: integer range 0 to 2;
   signal bcd: integer range 0 to 9;

begin
   count: process(clk)
      variable n: integer range 0 to 2;
   begin
      if (clk'event and clk='1') then
         if n < 2 then
          n:=n+1;
         else
          n:=0;
         end if;
         if n=0 then
          ledsel0 <= '1';
          ledsel1 <= '0';
          ledsel2 <=  '0';
          q <= n;
         elsif n=1 then
          ledsel1 <='1';
          ledsel0 <= '0';
          ledsel2 <= '0';
          q<=n;
         elsif n=2 then
          ledsel2 <= '1';
          ledsel0 <= '0' ;
          ledsel1 <= '0' ;
          q<=n;
         else
          ledsel0 <= '0' ;
```

```
            ledsel1 <= '0' ;
            ledsel2 <= '0' ;
          end if;
       end if;
    end process count;

    mux: process(dig0, dig1, dig2, muxsel)
    begin
       muxsel <=q;

       case muxsel is
          when 0 =>
            bcd <= dig0;
          when 1 =>
            bcd <= dig1;
          when 2 =>
            bcd <= dig2;
          when others =>
            bcd <= 0;
       end case;
    end process mux;

    converter: process(bcd)
    begin
       case bcd is
          when 0 => sevseg <= "1111110" ;
          when 1 => sevseg <= "1100000" ;
          when 2 => sevseg <= "1011011" ;
          when 3 => sevseg <= "1110011" ;
          when 4 => sevseg <= "1100101" ;
          when 5 => sevseg <= "0110111" ;
          when 6 => sevseg <= "0111111" ;
          when 7 => sevseg <= "1100010" ;
          when 8 => sevseg <= "1111111" ;
          when 9 => sevseg <= "1110111" ;
          when others => sevseg <= "1111110";
       end case;
    end process converter;

end displ_beh;
```

### *12.1.5 Circuit Integration*

The sequence classifier and recognizer is integrated using already specified components and structural modeling. All components are declared in the architecture declaration part, and then instantiated a required number of times. The

interconnections of components are achieved using internal signals declared in the architecture declaration part of the design. As the result of its operation the overall circuit provides two sets of 7-segment codes directed to 7-segment displays, together with the enable signals which select a 7-segment display to which the resulting code is directed. The integrated circuit is shown in Example 12.4.

Example 12.4 Integrated sequence recognizer and classifier

```
library ieee;
use ieee.std_logic_1164.all;

entity recognizer_classifier is
   port (code: in std_logic_vector(3 downto 0);
       clk, rst: in std_logic;
       sevsega, sevsegb: out std_logic_vector(6 downto 0);
       leda0, leda1, leda: out std_logic;
       ledb0, ledb1, ledb2: out std_logic;
       overfl0, overfl1: out std_logic);
end recognizer_classifier;

architecture structural of recognizer_classifier is
   signal cnt0, cnt1, start: std_logic;
   signal clas0, clas1: std_logic;
   signal succ: std_logic;
   signal d0out0, d0out1, d0out2 : integer range 0 to 9;
   signal d1out0, d1out1, d1out2 : integer range 0 to 9;

component recognizer
   port (inpcode: in std_logic_vector(6 downto 0);
       clk, reset: in std_logic;
       start: out std_logic);
end component;

component bcdcounter
   port ( clk, reset, ena: in std_logic;
       dout0,dout1,dout2: out integer range 0 to 9;
       cout: out std_logic);
end component;

component displcont
   port (dig0, dig1, dig2: in integer range 0 to 9;
       clk: in std_logic;
       sevseg: out std_logic_vector(6 downto 0);
       ledsel0, ledsel1, ledsel2: out std_logic);
end component;

component classifier
   port (code: in std_logic_vector(6 downto 0);
```

```
        more_ones: out std_logic;
        more_zeros: out std_logic);
end component;

begin

    cnt0 <= clas0 and start;
    cnt1 <= clas1 and start;

    recogn: recognizer port map (code, clk, rst, start);
    classif: classifier port map (code, clas1, clas0);
    bcdcnt0: bcdcounter port map (clk, rst, cnt0,
                    d0out0, d0out1, d0out2, overfl0);
    bcdcnt1: bcdcounter port map (clk, rst,
                    cnt1, d1out0, d1out1, d1out2, overfl1);
    disp0: displcont port map (d0out0, d0out1,
                    d0out2, clk, sevsega, leda0,
                    leda1, leda2);
    disp1: displcont port map (d1out0, d1out1,
                    d1out2, clk, sevsegb, ledb0,
                    ledb1, ledb2);
end structural;
```

A modified design of the sequence recognizer and classifier uses only one binary to 7-segment converter as it is illustrated in Figure 12.4. The digits from BCD counters are brought to a common multiplexer from which only one is selected to display, and the corresponding LED selection signal is activated. This design requires less FPLD resources. The VHDL description of the modified display controller is given in Example 12.5 below.

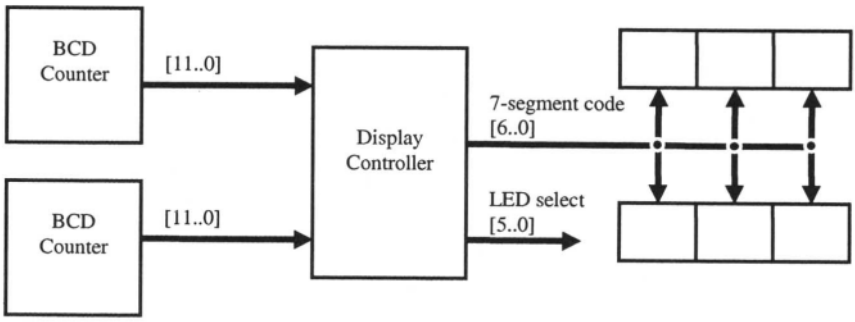



Figure 12.4 The modified display controller

Example 12.5 Modified display controller

```
library ieee;
use ieee.std_logic_1144.all;

entity displcon1 is
port (adig0, adig1, adig2: in integer range 0 to 9;
     bdig0, bdig1, bdig2: in integer range 0 to 9;
     clk: in std_logic;
     sevseg: out std_logic_vector(6 downto 0);
     aledsel0, aledsel1, aledsel2: out std_logic;
     bledsel0, bledsel1, bledsel2: out std_logic);
end displcon1;

architecture displ_beh of displcon1 is
    signal q, muxsel: integer range 0 to 5;
    signal bcd: integer range 0 to 9;

begin
    count: process(clk)
       variable n: integer range 0 to 5;
    begin
       if (clk'event and clk='1') then
          if n < 5 then
           n:=n+l;
          else
           n:=0;
          end if;

          if n=0 then
           aledsel0 <= '1';
           aledsel1 <= '0';
           aledsel2 <= '0';
           q <= n;
          elsif n=1 then
           aledsel0 <= '0';
           aledsel1 <= '1';
           aledsel2 <= '0';
           q<=n;
          elsif n=2 then
           aledsel0 <= '0';
           aledsel1 <= '0';
           aledsel2 <= '0';
           q<=n;
          elsif n=3 then
           bledsel0 <= '1';
           bledsel1 <= '0';
           bledsel2 <= '0';
           q<=n;
          elsif n=4 then
```

```
            bledsel0 <= '0' ;
            bledsel1<='1' ;
            bledsel2 <= '0';
            q <=n;
          else
            bledsel0 <= '0';
            bledsel1 <= '0' ;
            bledsel2 <= '1';
          end if;
        end if;
    end process count;

    mux: process(adig0, adig1, adig2, bdig0, bdig1, bdig2,
              muxsel)
    begin
        muxsel <=q;
        case muxsel is
          when 0 =>
          bcd <= adig0;
          when 1 =>
          bcd <= adig1;
          when 2 =>
          bcd <= adig2;
          when 3 =>
          when 4 =>
          bcd <= bdig1;
          when 5 =>
          bcd <= bdig2;
        end case;
    end process mux;

    converter: process(bcd)
    begin
        case bcd is
          when 0 => sevseg <= "1111110"
          when 1 => sevseg <= "1100000"
          when 2 => sevseg <= "1011011"
          when 3 => sevseg <= "1110011"
          when 4 => sevseg <= "1100101"
          when 5 => sevseg <= "0110111"
          when 6 => sevseg <= "0111111"
          when 7 => sevseg <= "1100010"
          when 8 => sevseg <= "1111111"
          when 9 => sevseg <= "1110111"
          when others => sevseg <= "1111110";
        end case;
    end process converter;

end displ_beh;
```

## 12.2 SART - A Simple Asynchronous Receiver-Transmitter

Serial data transfers are used in most computers and microcontroller to provide communication with input/output devices such as keyboard, mouse and modem or to provide a low-speed link between computers. In this section we present a simple asynchronous receiver-transmitter, called SART, as a generic circuit that can transmit serial data on over its transmit line (TxD) and receive serial data over its receive line (RxD). The circuit can be easily customized to support variations of data format. The purpose of the circuit is not to be programmable, but easily customizable to satisfy specific requirements of external circuitry that uses the SART. The design that is presented in this section has as its basic parameter value of the divisor used to divide system clock frequency SysClk by any feasible integer to achieve the required serial transmission speed.

Data is transmitted asynchronously, most often byte at a time, although its bit-length is the design parameter and can support different data formats, which is illustrated in Figure 12.5. Data line remains high when there is no transmission. To mark the start of transmission, data line goes low for one bit time, which is referred to as the start bit. Then, data bits are transmitted with the least significant bit first. Very often, data bits contain a single character ASCII code, which is represented by 7 bits, and the eight bit is used for parity check. Finally, data line must go high for at least one bit time, which is referred to as stop bits. Depending on the data format the number of stop bits can be one, one and a half or two, which refers to the duration of stop bits.

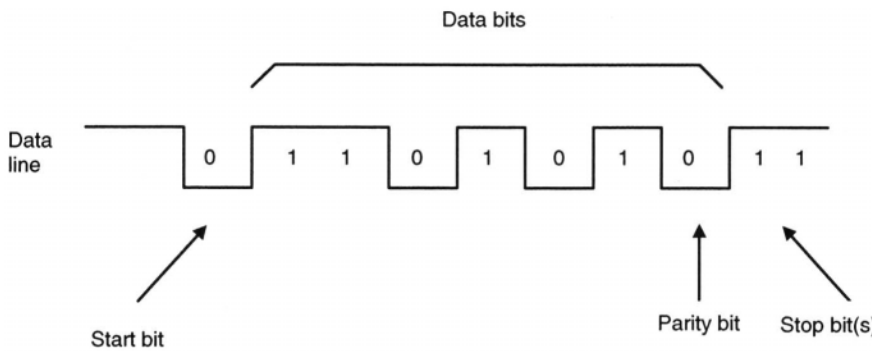



Figure 12.5 Serial data format

The number of bits transmitted per second is often referred to as baud rate or transmission speed. A baud rate generator determines baud rate. It divides the system clock to provide the bit clock. Typical standard baud rates are 300, 600,

1200, 2400, 4800, 9600 and 19200 bps (bits per second), but can be any as an application requires.

### *12.2.1 SART Global Organization*

The SART input and output ports are illustrated in Figure 12.6. In this case we will assume that the SART takes eight bits of parallel data on data_in lines and converts it to a serial bit stream that appears on the TxD line in the format described in Figure 12.5. When receiving data on serial line, the SART first detects the start bit, receives the data bits, and detects the stop bit(s). Data between the start and stop bit(s) is stored and provided in parallel form on data_out lines. As there is no synchronization between transmitter and receiver, the SART must synchronize with the incoming bit stream using the local clock. A number of control lines is provided to enable easy interfacing with external world, for instance with microprocessors. Those lines are:

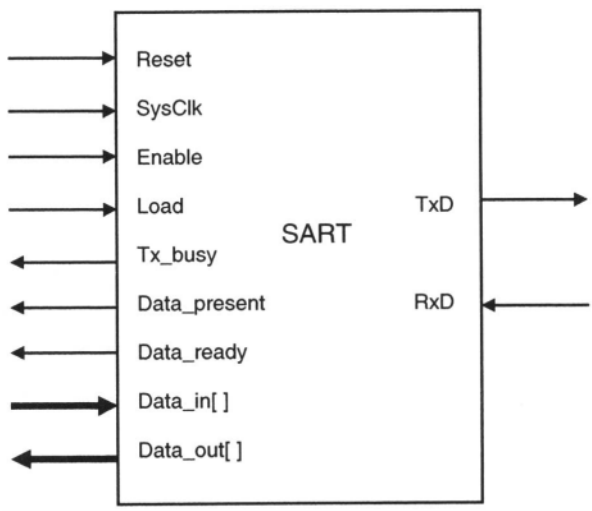



Figure 12.6 SART input and output ports

- reset, input that brings the SART into known initial state
- load, input that synchronizes parallel data transfer from external circuitry to the SART transmit data register
- enable, input that enables the SART
- tx_busy, output that indicates the transmission process on TxD line is being in progress and data is being transmitted from data transmit register

- data_present, output that indicates the data is present on RxD line
- data_ready, output that indicates that valid data has been received and can be read in parallel form from the receive data register

A more detailed block diagram representing SART is shown in Figure 12.7. The SART consists of three major parts: receiver, transmitter and baud generator. The role of each part is described in more details below.

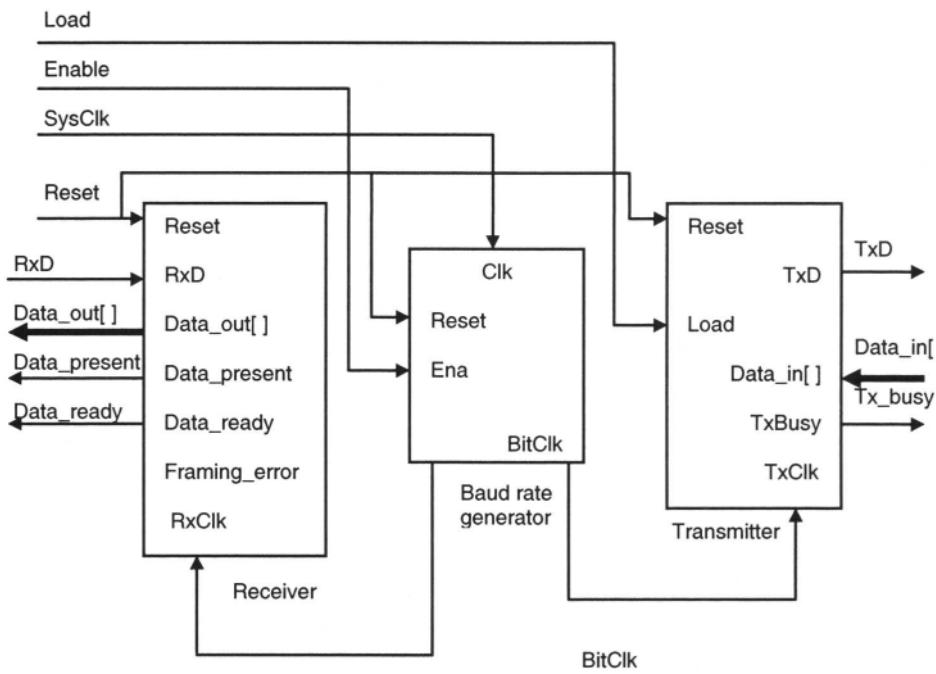



Figure 12.7 SART functional decomposition

### *12.2.2 Baud Rate Generator*

Baud rate generator is a frequency divider that provides two internal clocks. One is used as transmitting clock, TxClk, with output clock frequency as close as possible to the required baud rate. The other one is receiving clock, RxClock, and it is usually 8 or 16 times faster than transmitting clock. In our case we will use 16 times faster receiving clock. The RxClk enables sampling of the received signal 16 times per bit as we will further explain in Section 12.2.4. VHDL code for the baud rate

generator is given in Example 12.6. It is obvious that this divider provides only one parameter, generic divisor, for customization of the baud rate depending on the frequency of the input system clock.

Baud rate generator is described by two processes. The first process performs the counting task. Two counters are implemented. One is initialized at the value (DIVISOR-1), which provides receiving clock frequency. When it counts down to 0 it activates an internal signal (rx_cnt), which stays high for one system clock cycle. This signal is used, after deglitching, as the RxClk signal. The second counter divides frequency generated by the first counter by 16 and provides frequency corresponding to TxClk. In order to prevent glitches, outputs from both counters are used to activate another process whose only function is to keep these values delayed for one system clock cycle and in that way performs deglitching.

It would be straightforward to convert this baud rate generator into a programmable one with a control word that selects the baud rate from a selection of fixed baud rates that are a function of the basic one. That task is left to the reader (see problems at the end of this Chapter).

Example 12.6 Baud rate generator code

```
library ieee;
use ieee.std_logic_1164.all;

entity baud_rate_generator is
   generic(
    DIVISOR: integer: =12
    );
   port(
    clk: in std_logic;
    ena: in std_logic: = '1' ;
    reset: in std_logic: = '0' ;
    rx_clk: out std_logic;
    tx_clk: out std_logic
    );
end  baud_rate_generator;

architecture arch of baud_rate_generator is

   signal rx_cnt, tx_cnt: std_logic;

begin

counter: process(clk)
   variable cnt: integer range 0 to divisor-1;
   variable cnt16: integer range 0 to 15;
begin
```

```
    if clk'event and clk='1' then
     if (ena='1' and rx_cnt='1') or reset ='1' then
        cnt:=DIVISOR-1;
        cnt16 : =cnt16-1
        elsif ena ='1' and rx_cnt='0' then
        cnt:=cnt-1;
     elsif ena ='0' then
        cnt:=cnt;
        cnt16:=cnt16 ;
     end if;
    end if;

    if cnt=0 then
     rx_cnt <='1';
    else
     rx_cnt<='0';
    end if;

    if cnt16=0 and rx_cnt='1' then
     tx_cnt <='1';
    else
     tx_cnt <='0';
    end if;

end process counter;

deglitch: process(clk)
    variable degrx: std_logic;
    variable degtx: std_logic;
begin
    if  clk'event and clk='1' then
     if (rx_cnt and ena) ='1' then
        degrx:'1';
     else
        degrx:'0';
     end if;
     if (tx_cnt and ena) ='1' then
        degtx:='1';
     else
        degtx:='0' ;
     end if;

    end if;

    tx_clk < = degtx;
    rx_clk <= degrx;

end process deglitch;
end arch;
```

### *12.2.3 SART Transmitter*

The SART transmitter receives data in parallel form. The load signal controls when this data is stored in the transmit shift register (TSR). There, it is framed with start and the appropriate number of stop bits and sent to the transmit data line, TxD, in serial form bit by bit under the control of the transmit clock that is generated by the baud generator. The TSR is described by a process that performs different functions depending on the current state of the SART transmitter. The SART transmitter is represented and controlled by a finite state machine (FSM) that has two main functions: to maintain a counter of bits left for transmission (dbcount) and to provide synchronization within transmission process. Operation of the transmitter FSM is illustrated by the flowchart in Figure 12.8. The transmitter FSM can be in three states:

**Idle**. The transmitter is in idle state whenever there is no more data bits to transfer or after activation of reset signal. When in idle state dbcount bit counter is always initialized to the number of bits to be transmitted minus one (includes start and stop bits). The FSM is taken to await state when new data is loaded into the transmit shift register.

**Await** (Await name is used as wait is a VHDL keyword).This is the state in which the SART transmitter waits for synchronization with transmit clock (TxClk) to send next data bit on TxD line. Bit transmission is not performed in await but in the send state.

**Send**. The SART transmitter sends next data bit on TxD line in this state, updates by decrementing dbcount, and returns to await state if there is more data bits to transfer. If all data bits have been transferred, the FSM returns to idle state.

Actual register transfer operations are not shown in the flowchart for simplicity reasons. The transmitter FSM also generates an output signal (TxBusy) which indicates that the transmitter is busy with sending data and is not ready to accept new data on data in lines. This can be used by external circuitry to determine the moment when new data can be loaded into the transmitter shift register. The SART transmitter VHDL code is given in Example 12.7.

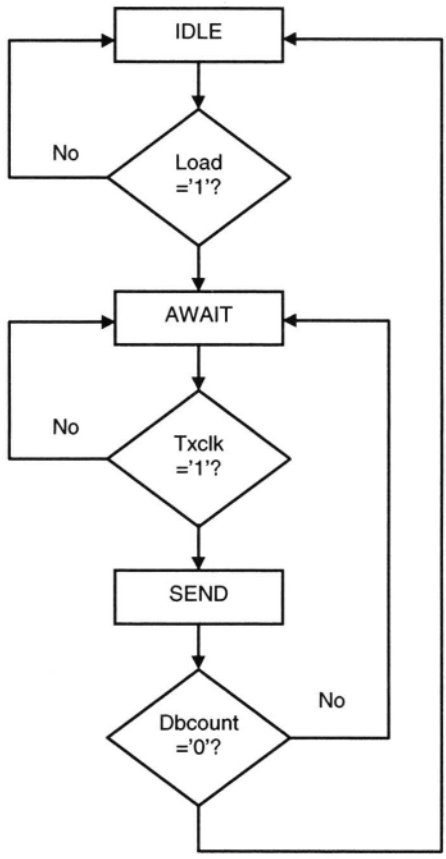



Figure 12.8 SART FSM operation

Example 12.7 SART transmitter code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

entity transmitter is
   generic(
    DATA_WIDTH : integer:=8;
    STOP_BITS: integer:=1
    );
```

```
   port(
    clk, reset: in std_logic;
    txclk: in std_logic;
    load: in std_logic;
    data_in : in std_logic_vector(
                    DATA_WIDTH-1 downto 0);
    txd: out std_logic;
    txbusy: out std_logic
    );
end transmitter;

architecture arch of transmitter is

-- data first loaded into transmit shift register
-- load signals that input data is valid
-- data is being sent in send state
-- state machine controls transitions between idle, await,
   and send states
-- txbusy is active during transmission of data

   type state_type is (idle, await, send);
   signal state     : state_type;

begin
transmit_fsm: process(clk)

-- dbcount holds the number of bits to be transmitted
-- including stop bits
-- when it becomes 0, all data bits have been sent

variable dbcount: integer range 0 to DATA_WIDTH+STOP_BITS;

begin
   if reset='1' then state<=idle;
   elseif clk'event and clk='1' then
   case state is
    when idle =>
      if load='1'   then
        state<= await;
      else
        state<= idle;
      end if;
      dbcount:= DATA_WIDTH+STOP_BITS;

    when await =>
      if txclk='1' then
        state<=send;
      else
        state<= await;
```

```
        end if;
        dbcount:=dbcount;
     when send =>
        if dbcount /= 0 then
           state<=await;
        else
           state<=idle;
        end if;
        dbcount:=dbcount-1;
    end case;

    if state /= idle then
     txbusy <= '1' ;
    else
     txbusy <='0';
    end if;

    end if;

end process;

TSR: process(clk)

variable shift:  std_logic_vector(DATA_WIDTH+1 downto 0);

begin
if clk'event and clk='1' then

case state is
when idle=>
    shift(0):='1'; -- stop bit
    shift(1):='0'; -- start bit
    shift(DATA_WIDTH +1 downto 2):=data_in(
              DATA_WIDTH -1 downto 0);

when await =>
    shift:=shift;

when send =>
    shift(DATA_WIDTH downto 0):=shift(DATA_WIDTH+1
              downto 1);
    shift(DATA_WIDTH+1):='1';
end case;
end if;

-- transfer contents of the topmost bit to txd output line
    txd <=shift(0);
end process;
end arch;
```

### *12.2.4 SART Receiver*

The SART receiver receives data in serial form on RxD line and stores it in receiver shift register (RSR). When all data bits are received, they are transferred to the receive data register (RDR), which is directly connected to dataout lines. External circuitry can read received data via dataout lines. By using data_ready status line the SART receiver provides an indication of properly received data. The receiver also indicates that it is just receiving data on serial RxD line using data_present status line. As data stream on RxD line is not synchronized with the local bit clock (TxClk), problems can be encountered when attempting to read RxD bit at the rising edge of bit clock if RxD value changes near the clock edge. This especially becomes true when the bit rate of the incoming signal differs from the local bit clock even for a very small value. To avoid problems we can sample RxD line near the middle of each bit interval using 16 times faster clock (RxClk). When RxD line goes for the first time to zero, we will wait for 8 RxClk cycles and sample it near the middle of the start bit. After that we will sample every 16 RxClk cycles until we reach stop bits.

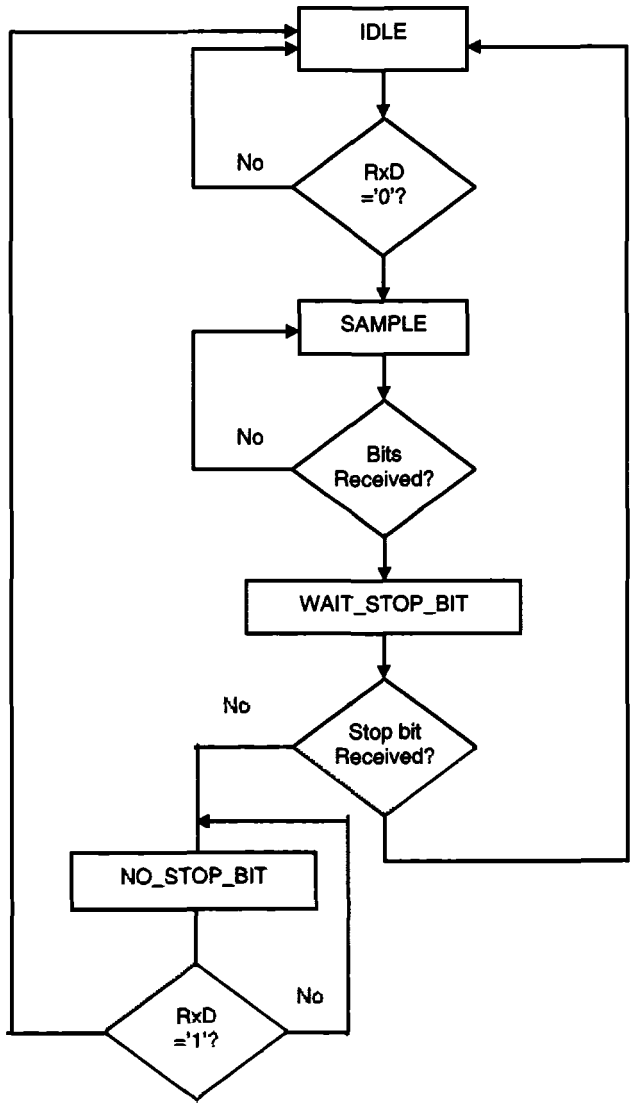
The receiver FSM performs the global control of the SART receiver operation. It is described by flowchart in Figure 12.9. The receiver FSM can be in four main states:

**Idle**. The receiver is in the idle state when there is no transfer of data on the line. The data transfer is detected with the transition from 1 to 0 on RxD line. After detecting an incoming bit two circuits are activated. First, the synchronization circuit that determines exact moment of bit sampling, and second, the bit counter that determines how many more bits have to be received.

**Sample**. In the sample state the receiver always checks are there more bits to be received and samples the bits near the middle of bit duration time. When the first bit is received, the Data_present line is activated to indicate that receiving new data is on the way. At each sample time, new bit is received and transferred into the RSR register. When all data bits are received, the FSM is transferred into wait_stop_bit. The Data_ready signal is activated and data is transferred from the RSR into RDR register.

**Wait_stop_bit**. In this state the FSM checks validity of the stop bit. If the value of the bit, which comes at the bit time of the stop bit is '1', the data is properly received and the receiver goes into idle state to wait for next data. If the value received is '0', a framing error is detected and it is indicated on Framing_error line. The FSM in this case goes into the no_stop_bit.

**No_stop_bit**. In this state FSM stays until the RxD line becomes quiet represented by '1' value. When it occurs the FSM is again transferred to the idle state and waits for a new data on RxD line.




Example 12.8 SART receiver VHDL code

The SART receiver VHDL code is presented in Example 12.8. It is implemented with three processes: synchronization process, global receiver FSM, and shift register process that implements both RSR and RDR registers.

Example 12.8 SART receiver VHDL code

```
library ieee;
use ieee.std_logic_1164.all;

entity receiver is
   generic(
    DATA_WIDTH : integer:=8;
             FIRST_COUNT: integer:=8;
    STOP_BITS: integer:=1
    );
   port(
    clk,reset: in std_logic;
    rxclk: in std_logic;
    rxd: in std_logic;
    data_out: out std_logic_vector(data_width-1
             downto 0);
    data_present: out std_logic;
    data_ready: inout std_logic
    );
end receiver;

architecture arch of receiver is

-- state machine controls reception of bits
-- it is in an idle state as long as first '1' is being
-- received
-- sampling is done 8 receive clock cycles after the
-- begining of any bit
-- received data format : start bit='0', 8 data bits, stop
-- bit(s) ='1'

type state_type is (idle, sample,  wait_stop_bit,
      no_stop_bit);

signal state: state_type;
signal sample_take: std_logic; -- determines sampling moment
signal bitsreceived: std_logic; -- number of bits to receive

begin

sync: process(clk)-- to provide proper sampling moment and
                  -- count received bits
```

```
   variable sync_cnt: integer range 0 to 15;
   variable bcnt: integer range 0 to data_width;

begin

if clk'event and clk='1' then
    if state=idle then
     sync_cnt:=first_count;
     bcnt:=data_width+stop_bits;
    else
     if rxclk='0' then
        sync_cnt:=sync_cnt;
        bcnt:=bcnt;
     else
        sync_cnt:=sync_cnt-1;
        if sample_take='1' then
          bcnt:=bcnt-1;
        end if;
     end if;
    end if;

end if;

if sync_cnt = 0 and rxclk='1' then
    sample_take <='1'; -- take sample
else
    sample_take <='0';
end if;

if bcnt=0 then
   bitsreceived<='0'; -- all data bits received
else
   bitsreceived<='0';
end if;

end process sync;


RSR: process(clk)

variable shift_data: std_logic_vector(DATA_WIDTH-1 downto 0);
variable RDR: std_logic_vector(DATA_WIDTH-1 downto 0);

begin

if clk'event and clk='1' then

if state = idle then  -- shift register initialized to all 0s
    shift_init:
```

```
    for i in 0 to DATA_WIDTH -1 loop
     shift_data(i):='0';
    end loop shift_init;

elsif state=sample then
    if sample_take='1' then   -- store new received bit
     shift_data(DATA_WIDTH -2 downto
                    0):=shift_data(DATA_WIDTH -1 downto 1);
     shift_data(DATA_WIDTH-1):=rxd;
    else
     shift_data:=shift_data;
    end if;

else
    shift_data:=shift_data;

end if;
end if;

if data_ready ='1' and state=wait_stop_bit then
    RDR:=shift_data; -- all data bits received and
                     -- transferred to receive data register

else
    RDR:=RDR;
end if;

    data_out <= RDR;

end process RSR;

Receiver_FSM : process(reset,clk)

variable dataready: std_logic;
variable datapresent: std_logic;

begin

if reset='1' then state <=idle;

elseif clk'event and clk='1' then

case state is

   when idle =>
         if rxd='0' then
          state <= sample;
         else
          state <= idle;
```

```
          end if;

     dataready:='0';

    when sample =>

          if bitsreceived = '0' then
        state <= sample;
          else
            state <= wait_stop_bit;
        dataready: = '1';
        end if;

     datapresent: ='1';
     framingerror: ='0';

    when wait_stop_bit =>

          if sample_take='1' then
        if rxd='1' then
              state <= idle;
          datapresent: ='0';
          framingerror:='0';
        else
          state <= no_stop_bit;
          framingerror:='1';
        end if;
          else
            state <= wait_stop_bit;
          end if;

    when no_stop_bit =>
        if rxd='1' then
          state <= idle;
        else
          state <=no_stop_bit;
        end if;

end case;

       framing_error<= framingerror;
       data_present<= datapresent;

end if;

       data_ready <= dataready;
end process Receiver_FSM;

end arch;
```

Integration of the above shown SART parts into the final design is left to the reader as an exercise (see Problems below).

## 12.3 Questions and Problems

12.1 The input code classifier and recognizer from Section 12.1 is to be redesigned to accept 7-bit codes and classifies them into eight different groups depending on the number of ones in each received code (number of ones can be 0,1,..., 7). A 16-bit binary counter is maintained for each of the groups. A single binary to BCD coder is used to decode the values accumulated in counters and its display on the seven segment displays in BCD code. Describe the circuit using VHDL.

12.2The circuit from 12.1 receives data and performs classification using the criteria described in Table 12.1. Value A is calculated as 4xB, where B is the decimal value corresponding to the received input code.

**Table 12.1 Classification criteria**

| Group | Calculated value A |
|---|---|
| 1 | $0 \leq A < 50$ |
| 2 | $51 \leq A < 250$ |
| 3 | $251 \leq A < 450$ |

12.3Integrate the transmitter, receiver and baud rate generator into the full SART using structural VHDL. Use generics to parameterize design with at least few parameters.

12.4SART receiver transmitter from Section 12.2 is extended to become programmable in terms of the data rate and enable data rate selection in system run-time. Extend the design to enable the data rates that are 2, 4 and 8 times faster than original one. Introduce a data rate selection register that will enable selection of the speed from an external circuit.

12.5The SART is extended by an additional data transmit register TDR that is accessible from external circuitry. The data is first written to the TDR and then transferred to the TSR to be transmitted on serial TxD line. In order to enable better communication with external circuitry, a TDR_empty line indicates that data is transferred from TDR to TSR.

12.6 Add an optional parity generation/checking function to the SART. The data format is seven data bits plus one parity bit. Selection of parity (odd, even or none) is programmable. The receiver should check the parity of the received data and indicate parity error.

12.7 Add two externally addressable registers to the SART: control register (CR) that enables selection of various options in-run time and status register (SR) that enables reading of the current status of the SART. Make decision which signals are available to the external circuitry and which ones must be accessed through CR and SR registers.

12.8 Redesign SimP processor presented in Chapter 7 using VHDL. Make a more generic SimP's description by separating instruction codes and other parameterizable features into the package(s). Then consider other problems presented in Section 7.5 (problems 7.1 to 7.9) and solve them using VHDL instead of AHDL. What are the advantages of using VHDL when designing a more complex system, such as a full microprocessor/microcomputer?

12.9 Extend SimP with the SART that will enable asynchronous serial data transfers. Also add a programmable 8-bit parallel port. Both parallel port and SART should be placed in SimP address space at locations at the top of the original 4K address space.

Write programs that will demonstrate the use of both parallel and serial port. Place the programs onto RAM implemented in EABs and perform both simulation and real runs using Altera UP1 board.

# 13 INTRODUCTION TO VERILOG HDL

This chapter presents a rather informal introduction to Verilog hardware description language. In conjunction with Chapter 14 it gives the basic flavor of the language and enables easy comparison with two other languages already introduced, AHDL and VHDL. While Verilog represents a kind of a middle path between AHDL and VHDL, it also adopts much strength found in both those languages. The language is suitable for both conceptual and high-level design and simulation, including design of own testbenches, on one side, and synthesis of digital circuits, on the other side. In this chapter we focus on the basic mechanisms and syntax features of the language useful for modeling for both simulation and synthesis, and in the next chapter we demonstrate the use of the language in synthesis of most common parts of digital systems. The main reason for presenting yet another hardware description language is its popularity and fact that it is adopted as an IEEE standard. The main reason for leaving Altera's proprietary AHDL in this book is its explicit power in description of synthesizable circuits, very often much more efficient than when using standard languages.

## 13.1 What is Verilog HDL?

The Verilog hardware description language, or Verilog HDL, or simply Verilog, was adopted as IEEE standard 1364 in 1995. It was created by Gateway Design Automation in 1983 and since then has gained a strong foothold among hardware designers for both simulation and synthesis of complex digital systems. It quickly became popular because of its similarities with programming language C. Initially it was aimed at simulation of digital designs described using behavioral modeling, but very soon it was adopted as an input into synthesis products and became a de-facto standard HDL.

Verilog can describe and simulate real circuits using built-in primitives, user-defined primitives, delays and timing requirements and has the ability to apply user-specified stimulus to design enabling testing and verification before synthesis. In this chapter we will make a short tour through the language and introduce its basic

features suitable for both simulation and synthesis, and then show its power on a number of examples of circuits designed for FPLD technology. As Verilog has many similarities with both AHDL and VHDL, its presentation will be done in a less formal way.

Verilog can model hardware structure equally effectively as VHDL. The choice of which language to use is very often based on personal preferences and other issues such as availability of tools and commercial terms. Most of today's design tool vendors support equally both languages. In many aspects Verilog is simpler than VHDL and does not have abstracts concepts, such as user-defined data types. Verilog data types are defined by the Verilog language and are suitable for modeling hardware structure. Two basic objects used in Verilog are *net* corresponding to electrical wire, and *reg* corresponding to memory element. Verilog has no concept of packages and therefore design reusability becomes more difficult. As the language was originally developed for gate level modeling, it has good constructs for modeling at that level. However, it also supports design description on even lower level of the layout of wires, resistors and transistors, but also on higher levels of abstraction such as the registers and the register transfer level (RTL). In this introduction we will concentrate on the RTL level as it is usually used when designing FPLDs.

As the other HDLs Verilog is a concurrent language. The basic concurrent statements are continuous assignment and the *always* statement. A continuous assignment statement uses the reserved word *assign* to assign data objects of the net data types.

Sequential statements are used within an *always* statement. The assigned objects are of type reg or integer.

## 13.2 Basic Data Types and Objects

A design entity in Verilog has only one design unit - the module declaration. It describes both a design's interface to the external world and its functional composition. However, the module can incorporate, by instantiating or simply by including other Verilog system file, other modules as something already described on the lower hierarchical design level. The module does not contain any declarative region and does not need to be declared. The same is case for subprograms, called a task or function.

Verilog supports a single base data type that is supported in synthesis which has the following four values:

0 – represents a logic zero or false condition
1 – represents a logic one or true condition
X – represents an unknown logic value
Z – represents high-impedance state

Data objects of this type can have a single element or an array of elements.

Verilog has more kinds of data objects than VHDL, and they relate more closely to the hardware structure being modeled:

**Signal nets**
wire
tri
**Wired nets**
wand
triand (*)
wor
trior (*)
trireg (*)
tri0 (*)
tri1(*)
**Supply nets**
supply0
supply1
**Register**
**Parameter**
**Integer**
**Time** (*)
**Memory (array)**

Asterix denotes those data objects that are not supported by synthesis tools.

If a net or register data objects are declared without a range, they are by default considered one bit wide (scalars). If a range is declared, it has multiple bits and is known as a vector.

### *13.2.1 Nets*

The synthesizable net data objects represent and model physical connection of signals. They are used for the following modeling situations:

- wire - models a wire that physically connects two signals together
- wor - models a wired-OR of several signal drivers driving the same net
- wand - models a wired-AND of several signal drivers driving the same net
- supply0 - models power supply in a circuit
- supply1 - models power supply in a circuit

A continuous assignment statement assigns values to any of the net data types.

Nets represent the continuous updating of outputs with respect to their changing inputs. For example in the Figure 1, output c is connected to input a by a not gate. If c is declared and initialised as shown, it will continuously be driven by the changing value of a. Default value for net objects is Z (high impedance).

### *13.2.2 Registers*

The register (reg) data object holds its value from one procedural assignment statement to the next and holds its value from one to the next simulation cycle. It does not imply that a physical register will be synthesized, although it is usually used for that purpose. The fundamental difference between nets and registers is that registers have to be assigned values explicitly. Once a value is assigned to a register data type object, it is held until the next procedural assignment to that object. This property can, for example, be used to model a D-type flip-flop with enable input as shown in Example 13.1, which represents our first example of a full valid Verilog description.

Example 13.1 Verilog description of a flip-flop

```
module Newff (q, data, enable, reset, clock);
    output q;
    input data, enable, reset, clock;
    reg q;

    always @(posedge clock)        // whenever the clock makes a
                          //transition to 1
     if (reset == 0)
       q = 1'b0;
     else if(enable==1)
       q = data;
     // implicitly : else q = q:

endmodule
```

Register q holds the same value until it us changed by an explicit assignment.

### *13.2.3 Parameters*

A parameter data object defines a constant. The position of parameter declaration defines whether the parameter is global to a module or local to a particular always statement. Only integer parameter constants are used in synthesizable designs. Examples of parameters are shown below.

```
parameter alpha = 8'hF4, width = 8
parameter one = 1, two =2, three = 3
```

### *13.2.4 Literals*

A literal is an explicit data value which can be assigned to an object or used within expressions. Verilog supports a number of literals that can be assigned to objects:

- Integer
- Z and X value
- Real
- String

**Integers**

Integers can be in binary ( b or B ), decimal ( d or D ), hexadecimal ( h or H ) or octal (o or O). Numbers are specified by

```
<size>'<base><number>  - for a full description
<base><number> - this is given a default size which is
                 machine dependant but at least 32 bits
<number> - this is given a default base of decimal
```

The size specifies the exact number of bits used by the number. For example, a 4 bit binary will have 4 as the size specification and a 4 digit hexadecimal will have 16 as the size specification since each hexadecimal digit requires 4 bits.

```
8'b10110010   // 8 bit number in binary representation
8'hF3   // 8 bit number in hexadecimal representation
```

**X and Z values**

X (or x) represents an unknown, and Z (or z) a high impedance value. An x declares 4 unknown bits in hexadecimal, 3 in octal and 1 in binary.

Z declares high impedance values similarly. Alternatively z, when used in numbers, can be written as ? This is advised in case expressions to enhance readability.

```
4'b11x0 // 4 bit binary with 2nd least sig. fig. unknown
4'b101z // 4 bit binary with least significant bit of high
    //impedance
16'dz  // 16 bit decimal high impedance number
24'd? // 24 bit decimal high impedance 'don't-care' number
8'hx5 // 8 bit number in hexadecimal representation with the
          // four most significant bits unknown
```

**Negative numbers**

A number can be declared to be negative by putting a minus sign in front of the size. The minus sign must appear at the start of a number (in all three formats given above). Examples of legal negative numbers are given below.

```
-8'd5    // 2's compliment of 5, held in 8 bits
-16'hF345  // 2's complement of hexadecimal number F345
         // held in 16 bits
```

**Underscore**

Underscores can be put anywhere in a number, except the beginning, to improve readability.

```
16'b0001_1011_1100_1111  //use of underscore
```

**Real**

Real numbers can be in either decimal or scientific format, if expressed in decimal format they must have at least one digit on either side of the decimal point.

```
1.8
3_2387.3398_3047
3.8e10  // e or E for exponent
2.1e-9
```

**Strings**

Strings are delimited by double quotes "...", and cannot be on multiple lines.

```
"hello world";          // legal string
"too high value";
```

## 13.3 Complex Data Types

Verilog basic data types can be combined into more complex structures often useful for description of digital systems. They include vectors, arrays, memories and tri—state data type.

### *13.3.1 Vectors*

Both the register and net data objects can be any number of bits wide if declared as vectors. Vectors can be accessed either in whole or in part, the left hand number is always the most significant number in the vector. Examples of vector declarations are shown below.

```
reg [7:0] accumulator; // accumulator is an 8-bit
                              // register
wire [31:0] data;   // data is a 32-bit wire
data[7:0] = accumulator;  // partial assignment
accumulator = 8'b0101_1100;   // full assignment
```

It is important to be consistent in the ordering of the vector width declaration. Normally the most significant figure is written first.

```
reg [3:0] a;  // it is important to adopt one
                    // convention for
reg [0:3] b;  // the declaration of vector width.
```

### *13.3.2 Arrays*

Registers, integers and time data types can be declared as arrays, as shown in the example below. Note the size of the array comes after the variable name in the declaration and after the variable name but before the bit reference in an assignment. The syntax and an example for array declaration are given below.

```
declaration:
```

```
<data_type_spec> {size} <variable_name> {array_size}

 reference:
 <variable_name> {array_reference} {bit_reference}

 reg data [7:0]; // 8 1-bit data elements
 integer [8:0] out [31:0]; // 32 8-bit output elements

 data[4]; // referencing the 4th data element
```

### *13.3.3 Memories*

Memories are simply an array of registers. The syntax is the same as above. They are useful to model RAM and ROM memories in digital systems.

```
reg [15:0] meml6_1024 [1023:0]; // memory mem16_1024 is 1K
                                // of 16-bit elements
 mem16_1024[489]; // referencing element 489 of mem16_1024
```

It is always good practice to use informative names like meml6_1024 to help keep track of memories.

### *13.3.4 Tri-state*

A tri-state driver is one that will output either high, low or "nothing". In some architectures, many different modules need to be able to put data onto (to drive) the same bus, at different times. Thus they all connect to the one common bus - but a set of control signals seek to ensure that only one of them is driving a signal at any one time.

In Verilog, this is modeled using different signal "strengths". There is a signal value: z, which is called "high-impedance". This basically means that a node is isolated, that is not driven. It is possible to assign this value to a net.

Normally if two values are simultaneously written to a net, the result is unknown: x; however, if a driven value is also assigned to the same net as a high-impedance value, the driven value will over-ride the z. This is the basis for the following tri-state driver:

```
module tri_state_driver (bus, drive, value);
    inout [7:0] bus;
    inputdrive;
    input [7:0] value;
    assign #2 bus = (drive == 1) ?value:8'bz;
endmodule // tri_state_driver
```

When the drive signal is high, the bus is driven to the data value, otherwise, this driver outputs only a high-impedance and hence can be over-ridden by any other driven value.

It should be noted that the bus is a wire and is designated as an inout variable on the port declarations.

## 13.4 Operators

Depending on the number of operands, Verilog has three types of operators. They take either one, two or three operands. Unary operators appear on the left of their operand, binary in the middle, and ternary separates its three operands by two operators. Examples of operators are given below.

```
clock = ~clock;  // ~ is the unary bit-wise negation
                        // operator, clock is the operand
c = a || b;  // || is the binary logical or, a and b are
                    // the operands
r = s ? t : u;    // ?: is the ternary conditional
                    // operator, which reads r = [if s is true
                    // then t else u]
```

### *13.4.1 Arithmetic operators*

The binary operators are multiply (*), divide (/), add (+), subtract (-) and modulus (%) used as shown in the Example 13.2.

Example 13.2 Arithmetic operators

```
module arithmetic_operators;
   reg [3:0] a, b;
    initial begin
      a = 4'b1100; // 12
      b = 4'b0011; // 3
      $displayb(a * b); // multiplication, evaluates to
```

```
        // 4'b1000 the four least significant bits of 36
      $display(a / b);  // division, evaluates to 4
      $display(a + b);  // addition, evaluates to 15
      $display(a - b);  // subtraction, evaluates to 9
      $display((a + 1'bl) % b); // modulus, evaluates to 1
   end

endmodule // arithTest
```

In this and other examples commands beginning with $ represent the system tasks that are useful when using Verilog for simulation. They enable communication of the testbench with the designer and provide an insight into the design behavior.

The unary operators are plus (+) and minus (-), and have higher precedence than binary operators. Note If any bit of an operand is unknown (x), then the result of any arithmetic operation is also unknown.

### *13.4.2 Logical Operators*

The logical operators are logical-and (&&), logical-or (||) and logical-not (!). All logical operators evaluate to either true ( 1 ), false ( 0 ), or unknown ( x ). An operand is true if it is non zero, and false if it is zero. An unknown or a high impedance value evaluates as false. An operand can be a variable or an expression that evaluates to either true or false as defined above. Example 13.3 illustrates the use of logical operators.

Example 13.3 Logical operators

```
module logical_operators;
   reg [3:0] a, b, c;

   initial begin
    a = 2;
    b = 0;
    c = 4'hx;

    $display(a && b) ;     // logical and, evaluates to 0
    $display(a || b) ;     // logical or, evaluates to 1
    $display(!a);   // logical not, evaluates to 0
    $display(a || c);     / evaluates to 1, unknown || 1 (=1)
    $display(!c); // evaluates to unknown
   end
endmodule // logical_operators
```

### 13.4.3 Relational Operators

The relational operators are less-than (<), more-than (>), less-then-or-equal-to (<=) and more-than-or-equal-to (>=). The true and false values are defined in the same way as above in the logical operator. In this case if any operand is unknown the whole expression evaluates to unknown. Example 13.4 shows the use of relational operators.

Example 13.4 Relational operators

```
module relational_operators;
   reg [3:0] a, b ,c, d;

   initial begin
    a=2;
    b=5;
    c=2;
    d=4'hx;

    $display(a < b) ;      // LHS less than RHS, evaluates to
                           // true, 1
    $display(a > b) ;      // LHS more than RHS, evaluates to
                           //false, 0
    $display(a >= c) ;     // more than or equal to, evaluates
                           // to true, 1
    $display(d <= a) ;     // less than or equal to, evaluates
                           // to unknown
   end
endmodule // relational_operators
```

### 13.4.4 Equality operators

The equality operators are logical-equality (==), logical-inequality (!=), case-equality (===) and case-inequality (!==). These operators compare the operands bit-by-corresponding-bit for equality.

The logical operators will return unknown if "significant" bits are unknown or high-impedance (x or z).

The case operators look for "equality" also with respect to bits that are unknown or high impedance.

If one operand is shorter than the other, it is expanded with 0s unless the most significant bit is unknown. The Example 13.5 shows the use of equality operators.

Example 13.5 Equality operators

```
module equality_operators;
    reg [3:0] a, b ,c, d, e, f;

    initial begin
     a = 4; b = 7;    // default to decimal base
     c = 4'b010;
     d = 4'bx10;
     e = 4'bx101;
     f = 4'bxx01;

     $displayb(c); // outputs 0010
     $displayb(d);  // outputs xx10
     $display(a == b) ; // logical equality, evaluates to 0
     $display(c != f) ; // logical inequality, evaluates to 1
     $display(d === e); // case equality, evaluates to 0
     $display(c !== d) ; // case inequality, evaluates to 1
    end
endmodule // equality_operators
```

### *13.4.5 Bitwise Operators*

The bitwise operators are negation (~), and (&), or (|), xor (^) and xnor (~^, ^~). Bitwise operators perform a bit-by-bit operation on the corresponding bits of both operands. If one operand is shorter it is bit extended to the left with zeros. Example 13.6 shows the use of bit-wise operators.

Example 13.6 Bitwise operators

```
module bitwise_operators;
    reg [3:0] a, b , c;

    initial begin
     a = 4'b1100; b = 4'b0011; c = 4'b0101;
     $displayb(~a);// bitwise negation, evaluates to 4'b0011
     $displayb(a & c); // bitwise and, evaluates to 4'b0100
     $displayb(a | b);  // bitwise or, evaluates to 4'b1111
     $displayb(b ^ c);  // bitwise xor, evaluates to 4'b0110
     $displayb(a ~^ c); //bitwise xnor, evaluates to 4'b0110

    end
endmodule // bitwise_operators
```

### 13.4.6 Reduction Operators

The reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^) xnor (~^) and an alternative xnor (^~). They take one operand and perform a bit-by-next-bit operation, starting with the two leftmost bits, giving a 1-bit result. Example 13.7 shows how to use reduction operators.

Example 13.7 Reduction operators

```
module reduction_operators;
    reg [3:0] a, b , c;

    initial begin
     a = 4'b1111;
     b = 4'b0101;
     c = 4'b0011;

        $displayb(& a); // bitwise and, (same as 1&1&1&1),
                        // evaluates to 1
     $displayb(| b);    // bitwise or, (same as 0|1|0|1),
                    //evaluates to 1
     $displayb(^ b) ;   // bitwise xor, (same as 0^1^0^1),
                    // evaluates to 0
    end

endmodule // reductTest
```

As an example, the reduction operators xor and xnor are useful in generating parity checks.

You should note the differences in logical, bit-wise and reduction operators. The symbols for bit-wise and reduction overlap but the number of operands is different in those cases.

### 13.4.7 Shift Operators

The shift operators are shift-left (<<) and shift-right (>>). The shift operator takes a vector and a number indicating the shift. The empty bits caused by shifting are filled with zeros as shown in Examples 13.8.

Example 13.8 Shift operators

```
module shift_operators;
    reg [7:0] a;

    initial begin
     a = 8'b10101010;

     $displayb(a << 1);    // shift left by 1, evaluates to
                           // 8'b01010100
     $displayb(a >> 2);  // shift right by 2, evaluates to
                           // 8'b01010101
    end

endmodule // bitwise_operators
```

The shift operators are useful in modeling shift registers, long multiplication algorithms, etc.

### *13.4.8 Concatenation operator*

The concatenation operator ({,}) appends sized nets, registers, bit select, part select and constants as shown in Example 13.9.

Example 13.9 Concatenation operator

```
module concatenation_operator;
    reg a;
    reg [1:0] b;
    reg [5:0] C;

    initial begin
     a = 1'b1;
     b = 2'b00;
     c = 6'b101001;

     $displayb({a, b});  // produces a 3-bit number 3'b100
     $displayb({c[5:3], a}); // produces 4-bit number
                           // 4'b1011

    end

endmodule // concatenation_operator
```

### *13.4.9 Replication operator*

Replication can be used along side concatenation to repeat a number as many times as specified as shown in Example 13.10.

Example 13.10 replication operator

```
module  replication_operator;
    reg a;
    reg  [1:0]  b;
    reg  [5:0]  c;

    initial begin
     a = 1'b1;
     b = 2'b00;

     $displayb({4{a}});     // evaluates as 1111
     c = {4{a}};
     $displayb(c);          // evaluates as 001111
    end
endmodule      // replication_operator
```

## 13.5 Design Blocks and Ports

Basic Verilog design unit is a module. Modules connect to the remaining world through their ports similarly as in AHDL and VHDL. In this section we introduce those basic design mechanisms that are used in both simulation and synthesis of digital systems.

### *13.5.1.Modules*

The Verilog language describes a digital system as a set of modules. Each of these modules has an interface to other modules in a form of input and output ports to describe how they are interconnected. Usually we place one module per file but that is not a requirement. The modules may run concurrently, but very often we have one top-level module that specifies a closed system containing both test data and hardware models. The top-level module invokes instances of other modules.

Modules can represent bits of hardware ranging from simple gates to complete systems such as processors, standard interfaces etc. Modules can either be specified behaviorally or structurally (or a combination of the two). A behavioral specification defines the behavior of a digital system (module) using traditional

programming language constructs such as procedural statements. A structural specification expresses the behavior of a digital system (module) as a hierarchical interconnection of sub modules. At the bottom of the hierarchy the components must be primitives or specified behaviorally.

The syntax used to describe a module is as follows:

```
module <module name> (<port list>);
  <declarations>
  <module items>
endmodule
```

The <module name> is an identifier that uniquely names the module. The <port list> is a list of input, inout and output ports that are used to connect to other modules. The <declarations> section specifies data objects as registers, memories and wires as well as procedural constructs such as functions and tasks.

The <module items> may be initial constructs, always constructs, continuous assignments or instances of modules. They describe concurrent entities within the module using either behavioral or structural descriptions. All these types of module items are described in following sections.

**Continuous Assignment**

Continuous assignments drive wire variables and are evaluated and updated whenever an input operand changes value. Below in Example 13.11 is a behavioral specification of a module named newand. The output c is the and of the inputs a and b.

Example 13.11 Behavioral specification of an AND gate

```
module newand (a, b, c);
    input a, b;
    output c;

    assign c = (a & b);

endmodule
```

The ports a, b and c are labels on wires. The continuous assignment, which uses the keyword assign and operator = to distinguish from procedural assignments, continuously watches for changes to variables in its right hand side and whenever

that happens the right hand side is re-evaluated and the result immediately propagated to the left hand side.

### Initial block

An initial block consists of a statement or a group of statements enclosed in begin... end which will be executed only once at simulation time 0. If there is more than one block they execute concurrently and independently. The initial block is normally used for initialization, monitoring, generating waveforms (eg, clock pulses) and processes which are executed once in a simulation. Example 13.12 shows initialization and wave generation using two initial blocks.

Example 13.12 Use of initial blocks for initialization and waveform generation

```
initial
       clock = 1'b0;       // variable initialization

initial
       begin  // multiple statements have to be grouped
       alpha = 0;
       #10 alpha = 1;       // waveform generation
       #20 alpha = 0;
       #5  alpha = 1;
       #7  alpha = 0;
       #10 alpha = 1;
       #20 alpha = 0;
end;
```

### Always Block

An always block is similar to the initial block, but the statements inside an always block will repeat continuously, in a looping fashion, until stopped by $finish or $stop. One way to simulate a clock pulse is shown in the example below. Note, this is not the best way to simulate a clock. In the section on the forever statement, a better method for generating clock is described (Example 13.22).

Example 13.13 Generation of clock

```
module pulse;
   reg clock;
```

```
    initial clock = 1'b0;  // start the clock at 0

    always #10 clock = ~clock; // toggle every 10 time units

    initial #5000 $finish  // end the simulation after 5000
                    // time units

endmodule
```

The always blocks allow us to describe the same behavior in different ways. For example, the and gate can be described using a different, non-blocking procedural assignment within an always block (operator <= used) as shown in Example 13.14.

Example 13.14 Alternative description of and gate using non-blocking assignment

```
module newand (a, b, c);
    input  a, b;
    output  c;
    reg  c;

    always
     begin
        c <=a & b;
     end

endmodule
```

The always statement is used without conditions to denote that the assignment statement will execute whenever the block surrounded by begin...end statements is executed.

The assignment statements are used to model combinational circuits where the outputs change whenever any input changes.

## Module Instantiation

Here in Example 13.15 we describe a structural specification of a module newand3 that represents a 3-input and gate obtained by connecting the output of one 2-input and gate and the third input to the inputs of the second 2-input and gate. The 2-input and gates are those shown in Example 13.13.

Example 13.15 Structural model of a 3-input and gate

```
module newand3 (a, b, c, d);
    input a, b, c;
    output d;
    wire w1;

    // two instances of the module newand
    newand and1(a, b, w1);
    newand and2 (w1, c, d);

endmodule
```

This module has two instances of the newand module called and1 and and2 connected together by an internal wire w1. The example shows the principles of hierarchical designs using Verilog language: already designed modules can be used at the next hierarchical design level simple instantiation and structural interconnections.

The general form to invoke an instance of a module is :

<module name> <parameter list> <instance name> (<port list>);

where <parameter list> are values of parameters passed to the instance.

### *13.5.2 Ports*

Ports provide a means for a module to communicate through input and output with the other modules. Every port in the port list must be declared as input, output or inout, in the module. All ports declared as one of the above are assumed to be a wire by default, to declare it otherwise it is necessary to declare it again. For example in a D-type flip-flop we want the output to hold on to its value until the next clock edge so it has to be a register:

```
module d_ff(q, d, reset, clock);
   output q;            // all ports must be declared
   input d, reset, clock;     // as input or output
   reg q; // the ports can be declared again as required
```

By convention, the outputs of the module are always first in the port list. This convention is also used in the predefined modules in Verilog.

### Inputs

In an inner module inputs must always be of a net type, since values will be driven onto them. In the outer module the input may be a net type or a reg.

### Outputs

In an inner module outputs can be of a net type or a reg. In an outer module the output must be of a net type since values will be driven onto them by the inner module.

### Inouts

Inouts must always be of a net type.

### Port Matching

When calling a module the width of each port must be the same, eg, a 4-bit register cannot be matched to a 2-bit register.

Output ports may remain unconnected, by missing out their name in the port list. This would be useful if some outputs were for debugging purposes or if some outputs of a more general module were not required in a particular context. However input ports cannot be omitted for obvious reasons.

### Connecting Ports

Ports can be connected by either ordered list or by name. The ordered list method is recommended for the beginner, in this method the port list in the module instantiation is in the same order as in the module definition as shown in Example 13.16.

Example 13.16 Using ordered list to connect ports

```
module d_ff( q, d, reset, clock);

endmodule

module e_ff(q, d , enable, reset, clock);
```

```
    output q;
    input d, enable, reset, clock;
    wire w;

    d_ff dff0(q, w, reset, clock);

endmodule
```

The second method is connecting ports by name. When instantiating, the ports in the definition are accompanied by the corresponding port name in the instantiation.

## 13.6 Procedural Statements

Verilog HDL has a rich collection of control statements, similar to those in traditional programming languages, which can used in the procedural sections of code, i. e., within an initial or always block.

### *13.6.1 Selection - if and case Statements*

The if statement has the following syntax:

```
    if(<conditional_expression>) statement{ elsestatement}
```

The if statement causes a conditional branch. If the conditional_expression evaluates to true the first statement or set of statements is executed, else the second statement or set of statements is executed. The keywords begin... end are used to group statements. An example of if statement, which models a part of 4-to-1 multiplexer, is given below.

Example 13.17 Using if statement to describe multiplexer behavior

```
if ( sel0 == 1)
    if(sel1 == 1)
     out = in3;
    else out = in2;

    if (sel1 == 1)
     out = in1;
    else out = in0;
```

In the case statement the first <value> that matches the value of the <expression> is selected and the associated statement is executed. Then, control is transferred to after the endcase. The case statement has the following syntax.

```
case (<expression>)
  <value1>: <statement>
  <value2>: <statement>
  default: <statement>
endcase
```

The following example checks a 2-bit signal for its value to select input that will be forwarded to the output and can be used to model 4-to-1 multiplexer.

Example 13.18 Using case statement to describe multiplexer behavior

```
case ({sel1, sel0}) // concatenation
    2'b00 : out = in0;
    2'b01 : out = in1;
    2'b10 : out = in2;
    2'b11 : out = in3;
endcase
```

Variants of the case statement are casez and casex. Whereas the case statement compares the expression to the condition bit by bit, insuring the 0, 1, x, and z match, the casez treats all the zs in the condition and expression as ?s, ie "don't cares". The casex similarly treats all the xs and zs as ?s. These alternatives must be used carefully as they can easily lead to bugs.

### *13.6.2 Repetition - for, while, repeat and forever Statements*

The four constructs for repetitive operations in Verilog are: for, while, repeat and forever. All of these can only appear inside initial and always blocks.

#### For Statement

The for statement executes a statement or a set of statements specified number of times. This loop can initialize, test and increment the index variable used to count a number of passes through the loop. The for statement has the following syntax:

```
for (reg_initialization; conditional; reg_update) statement
```

It has three parts: the first part is executed once, before the loop is entered; the second part is the test which when true causes the loop to re-iterate; and the third part is executed at the end of each iteration.

Example 13.19 For statement

```
for(i = 0; i < 10; i = i + 1)
   begin
    $display("i= %0d", i);
   end
```

### While Statement

The while statement executes a statement or set of statements while a condition is true. It has the following syntax:

```
   while (conditional) statement
```

The while statement executes while the conditional is true. The conditional can consist of any logical expression. Statements in the loop can be grouped using the keywords begin...end as illustrated in Example 13.20.

Example 13.20 While statement

```
i = 0;
while(i < 10)
   begin
    $display("i= %0d", i);
    i = i + 1;
   end
```

### Repeat Statement

The repeat statement repeats the following block a fixed number of times. It has the following syntax:

```
   repeat (conditional) statement
```

The conditional can be a constant, variable or a signal value, but must contain a number. If the conditional is a variable or a signal value, it is evaluated only at the

entry to the loop and not again during execution. Example 13.21 illustrates the repeat statement.

Example 13.21 Repeat statement

```
repeat  (20)
   begin
    $display("i= %0d", i);
    i = i + 1;
   end
```

### Forever Statement

The forever statement executes continuously until the end of a simulation is requested by a $finish. It can be thought of as a while loop whose condition is never false. The forever statement must be used with a timing control to limit its execution, otherwise its statement would be executed continuously at the expense of the rest of the design. Its has the following syntax:

```
forever statement
```

The use of a forever statement is shown in Example 13.22.

Example 13.22 Forever statement

```
reg clock;

initial begin
   clock = 1'b0;
   forever #10 clock = ~clock;  // the clock flips every 10
                                // time units
end

initial #30000 $finish;
```

### Blocking and Non-blocking Procedural Assignments

The Verilog language has two forms of the procedural assignment statement: blocking and non-blocking. The two are distinguished by the = and <= assignment operators, respectively. The blocking assignment statement (= operator) acts much like in traditional programming languages. The whole statement is carried out

before control passes on to the next statement. The non-blocking (<= operator) evaluates all the right-hand sides for the current time unit and assigns the left-hand sides at the end of the time unit. Verilog description illustrating both types of assignments and the output produced from the Verilog simulator are shown in Example 13.23.

Example 13.23 Blocking and non-blocking procedural assignments

```
module procedural_assignments;
    reg [0:7] A, B;
    initial begin: init1
     A = 3;
     // blocking procedural assignments
     #1 A = A + 1;
     B = A + 1;
     $display("Blocking:    A= %b B= %b", A, B );

     // non-blocking procedural assignments
     A = 3;
     #1 A <= A + 1;
     B <= A + 1;

     #1 $display("Non-blocking: A= %b B= %b", A, B );
    end
endmodule
```

Output produced by the simulator will be as follows:

```
Blocking:        A= 00000100 B= 00000101
Non-blocking:    A= 00000100 B= 00000100
```

The effect is for all the non-blocking assignments to use the old values of the variables at the beginning of the current time unit and to assign the registers new values at the end of the current time unit. This reflects how register transfers occur in some hardware systems.

### Tasks and Functions

Tasks in Verilog are like procedures in other programming languages. Tasks may have zero or more arguments and do not return a value. Functions in Verilog act

like function subprograms in other programming languages with two important exceptions:

- A Verilog function must execute during one simulation time unit. That is, notime controlling statements, such as delay control (#), event control (@) or wait statements, are allowed.

- A task can contain time-controlled statements.Verilog function can not invoke (call, enable) a task; whereas a task may call other tasks and functions.

The task has the following syntax:

```
task <task name>;
   <argument ports>
   <declarations>
   <statements>
endtask
```

An invocation of a task is of the following form:

```
<name of task> (<port list>);
```

where <port list> is a list of expressions which correspond to the <argument ports> of the definition. Port arguments in the definition may be input, inout or output. Since the <argument ports> in the task definition look like declarations, the designer must be careful in adding declarations at the beginning of a task. Example 13.24 illustrates the definition and use of a task.

Example 13.24 Task definition and invocation

```
module task_example;

task add;              // task definition
   input a, b;    // two input argument ports
   output c;      // one output argument port
   reg R;         // register declaration
   begin
    R = 1;
    if (a == b)
      C = 1 & R;
    else
      C = 0;
   end
endtask
```

```
initial begin: init1
    reg p;
    add(1, 0, p) ;    // invocation of task with 3 arguments
    $display("p= %b", p);
end

endmodule
```

Input and inout parameters are passed by value to the task and output and inout parameters are passed back to invocation by value on return. Call by reference is not available.

Allocation of all variables is static. Therefore, a task may call itself but each invocation of the task uses the same storage, i. e., the local variables are not pushed on a stack. Since concurrent threads may invoke the same task, the programmer must be aware of the static nature of storage and avoid unwanted overwriting of shared storage space.

The purpose of a function is to return a value that is to be used in an expression. A function definition must contain at least one input argument. The passing of arguments in functions is the same as with tasks A function has the following syntax:

```
function <range or type> <function name>;
 <argument ports>
 <declarations>
 <statements>
endfunction
```

where <range or type> is the type of the results passed back to the expression where the function was called. Inside the function, one must assign the function name a value. Example 13.25 defines a function which is similar to the task from Example 13.24.

Example 13.25 Function definition and invocation

```
module functions;

function [1:1] add2; // function definition
    input a, b;          // two input argument ports
    reg R;               // register declaration
    begin
     R = 1;
     if (a == b)
       add2 = 1 & R;
```

```
     else
        add2 = 0;
    end
endfunction

initial begin: init1
    reg p;
    p = add2(1, 0);  // invocation of function with 2
                     // arguments
    $display("p= %b", p) ;
    end

endmodule
```

### Timing Control

The Verilog language provides two types of explicit timing control when simulation time procedural statements are to occur. The first type is a delay control in which an expression specifies the time duration between initially encountering the statement and when the statement actually executes. The second type of timing control is the event expression, which allows statement execution. The third subsection describes the wait statement which waits for a specific variable to change.

Verilog is a discrete event time simulator, i. e., events are scheduled for discrete times and placed on an ordered-by-time wait queue. The earliest events are at the front of the wait queue and the later events are behind them. The simulator removes all the events for the current simulation time and processes them. During the processing, more events may be created and placed in the proper place in the queue for later processing. When all the events of the current time have been processed, the simulator advances time and processes the next events at the front of the queue.

If there is no timing control, simulation time does not advance. Simulated time can only progress by one of the following:

1. gate or wire delay, if specified.
2. a delay control, introduced by the # symbol.
3. an event control, introduced by the @ symbol.
4. the wait statement.

The order of execution of events in the same clock time may not be predictable.

**Delay Control** (#)

A delay control expression specifies the time duration between initially encountering the statement and when the statement actually executes. For example:

```
initial begin
    a = 0;          // executed at simulation time 0
    #10 b = 2;      // executed at simulation time 10
    #15 c = a;      //    ... at time 25
    #b c = 4;       //    ... at time 27
    b=5;            //    ... at time 27
end
```

The delay value can be specified by a constant or variable. Note that the time is not in seconds, but it is relative to the current unit of time.

A common example of using delay control is the creation of a clock signal:

```
initial begin
    clock = 1'b0;
    forever #10 clock = ~clock;
end
```

**Events**

The execution of a procedural statement can be triggered with a value change on a wire or register, or the occurrence of a named event. Event control statement has the following syntax:

```
    @ event_identifier or @ (event_expression)
```

where event_expression can be

```
    exppression
    event_id
    posedge exppression
    negedge expression
    event_exppression or event_expression.
```

Event-based timing control allows conditional execution based on the occurrence of a named event. Verilog waits on a predefined signal or a user defined variable to change before it executes a block. Examples of event driven executions are shown below:

```
@reset
begin  // controlled by any value change in
```

```
   a = b & c; // the signal reset
end

@(posedge clock1) a = b & c;   // controlled by positive edge
                              // of clock1

@(negedge clock2) a = b & c;     // controlled by negative
                        // edge of clock2

forever @(negedge clock)     // controlled by negative edge
   begin                    // of clock
    A = B&C;
   end

a = @(posedge clock) b;    // evaluate b immediately and
              // assign to a on a positive clock edge
```

When using posedge and negedge, they must be followed by a 1-bit expression, typically a clock. A negedge is detected on the transition from 1 to 0 (or unknown). A posedge is detected on the transition from 0 to 1 (or unknown).

### Triggers

Verilog also provides features to name an event and then to trigger the occurrence of that event. We must first declare the event:

```
event external_event;
```

To trigger the event, we use the –> symbol:

```
-> external_event;
```

To control a block of code, we use the @ symbol as shown:

```
@(external_event) begin
 < procedural code>
end
```

We assume that the event occurs in one thread of control, i. e., concurrently, and the controlled code is in another thread. Several events may to or-ed inside the parentheses.

If we wish to execute a block when any of a number of variables change we can use the sensitivity list to list the triggers separated by or

```
always @ (a_change or b_changes or c_changes)
indicate_change = 1;
```

A change in any of the variables will cause execution of the second statement. As you can see this is just a simple extension to the idea of event based timing control described in the previous section.

### Wait Statement

The wait statement allows a procedural statement or a block to be delayed until a condition becomes true. The following is an example of using wait statement:

```
wait (a == 1)
begin
   a = b & c;
end
```

The difference between the behavior of a wait statement and an event is that the wait statement is level sensitive whereas @(**posedge** clock); is triggered by a signal transition or is edge sensitive.

### Gate Delays

This type of delay is only associated with the primitive gates defined within Verilog. An example of using a gate delay of 2 time units in a 2-input and gate is shown below:

```
and #(2) and1(c, a, b) ;
```

## 13.7 Simulation Using Verilog

One of the major Verilog applications is simulation. Although our aim in this book is at presenting those features used primarily in synthesis, in this section we introduce briefly those features of the language that are useful in writing testbenches. For certain routine operations Verilog provides so called system tasks that enable communication of the Verilog model with the designer. The system tasks are distinguished from the other Verilog keywords by using $ prefix and have a general form $keyword, where keyword represents system task's name. The most important system tasks related to simulation are those that enable writing to output, monitoring a simulation and ending a simulation.

### 13.7.1 Writing to Standard Output

Writing to standard output is similar to one found in programming language C. Verilog provides two groups of system tasks for writing to standard output:

- $display tasks, which provide writing including automatic placement of a newline at the end of text that will be displayed.
- $write tasks, which have the same function as $display except they do not put newline at the end of text that will be displayed.

The list of all tasks for writing to standard output is shown in Table 13.1. The most useful of these is $display. This can be used for displaying strings, expression or values of variables. Below are some examples of usage.

```
$display("Hello World");
   output: Hello World

$display($time) // current simulation time.
   output: 1280

contents = 8'b0110;
$display(" The content is %b", contents);
   output:  The content is 00000110
```

**Table 13.1 System tasks for writing on standard output**

| Task | Default format |
|---|---|
| $display | decimal |
| $displayb | binary |
| $displayh | hexadecimal |
| $displayo | octal |
| $write | decimal |
| $writeb | binary |
| $writeh | hexadecimal |
| $writeo | Octal |

The formatting syntax is similar to that of printf in the C programming language. Format specifications are shown in Table 13.2, and escape sequences for printing special characters are shown in Table 13.3.

Table 13.2 Format specifications

| Format | Display |
|---|---|
| %d or %D | decimal |
| %b or %B | binary |
| %h or %H | hexadecimal |
| %o or %O | octal |
| %m or %M | hierarchical name |
| %t or %T | time format |
| %e or %E | real in scientific format |
| %f or %F | real in decimal format |
| %g or %G | real in shorter of two real formats |

Table 13.3 Escape sequences

| Sequence | Effect |
|---|---|
| \n | newline |
| \t | tabulate |
| \\ | print \ |
| \" | print " |
| %% | print % |

### *13.7.2 Monitoring and Ending Simulation*

Verilog provides three tasks for monitoring a simulation run: $monitor, $monitoron and $monitoroff. The format of $monitor task is exactly the same as $display. The difference is that an output occurs on any change in the variables, rather than at specified times. Thus the $monitor task establishes a list of variables which are watched, and in practice it is written to be executed at the beginning of the simulation.

Monitoring can be enabled or disabled using $monitoron or $monitoroff respectively. Monitoring is on by default at the beginning of a simulation.

Example 13.26 illustrates how a simulation run can be monitored. Three initial blocks are used in this example. One is used to perform required processing in time including time advance. The other two are used to control monitoring and ending the simulation run.

Example 13.26 Control of a Simple Simulation Run

```
module simple_simulation;
    integer a, b, c;

    initial
     begin
        a = 3;
        b = 4;
        C = 0;
        forever
          begin
            #10 a = a + b;
            #10 b = a - 1;
            #10 c = c + 1;
          end
     end

    initial #100 $finish;

    initial
     begin
        $monitor($time, " c = %d, a = %d, b = %d", c, a, b) ;
     end

endmodule
```

The above model will produce the following output:

```
0  c = 0, a = 3, b = 4
10 c = 0, a = 7, b = 4
20 c = 0, a = 7, b = 6
30 c = 1, a = 7, b = 6
40 c = 1, a = 13, b = 6
50 c = 1, a = 13, b = 12
60 c = 2, a = 13, b = 12
70 c = 2, a = 25, b = 12
80 c = 2, a = 25, b = 24
90 c = 3, a = 25, b = 24
```

Two tasks are provided to end a simulation run: $stop and $finish. $finish exits the simulation and passes control to the operating system. $stop suspends the simulation and puts Verilog into interactive mode.

## 13.8 Questions and Problems

13.1 What are the basic data types supported in Verilog? Compare them with those in VHDL.

13.2 What are the basic object built-in Verilog? Compare them with those in VHDL.

13.3 Is Verilog a strongly typed language? Explain it.

13.4 How changes on signals can be described in Verilog. Give examples of change checks that are used to check the level and transition on a signal.

13.5 What are the basic mechanisms that support concurrency in Verilog? Compare them with those in VHDL. Which language gives more flexibility in describing models and their test benches?

13.6 Use Verilog to describe a clock that with duty cycle equal 0.25.

13.7 Use Verilog to describe four-phase non-overlapping clock.

13.8 Use Verilog to describe 4-to-1 multiplexer using structural model based on individual user-defined components (and, or, not) interconnected at a higher level of hierarchy.

# 14 VERILOG AND LOGIC SYNTHESIS BY EXAMPLES

The aim of this chapter is to demonstrate features and capabilities of Verilog for description of synthesizable digital circuits, and restrict the use of only those constructs that are useful for logic synthesis. As our target technology are FPLDs we primarily use those constructs supported by Altera compiler within Max+Plus II design environment. However, those or very similar constructs are supported by other synthesis tools. The reader can compare Verilog designs with similar designs using AHDL and VHDL and see both strengths and weaknesses of the language compared to those other languages.

## 14.1 Specifics of Altera's Verilog HDL

The Verilog HDL is a high-level, modular language that is completely integrated into the Max+Plus II design environment. Verilog design files (with the .v extension) can be created using existing Max+Plus II text editor or another text. Then they can be compiled and simulated before downloading to Altera devices.

Max+Plus II software includes a built-in Verilog Netlist Reader for directly processing Verilog HDL files. Verilog design files can contain any combination of Max+Plus II-supported constructs. They can also contain Altera-provided logic functions, including primitives, megafunctions and macrofunctions, and user-defined logic functions.

Verilog HDL constructs allow to create entire hierarchical projects with Verilog HDL, or mix Verilog design files with other types of design files in a hierarchical project. Verilog HDL designs are easily incorporated into a design hierarchy. In the text editor, you can automatically create a symbol that represents a Verilog design file and incorporate it into a GDF. Similarly, you can incorporate custom functions, as well as Altera-provided logic functions, into any Verilog design file.

The Max+Plus II compiler allows to check Verilog HDL syntax quickly or perform a full compilation to debug and process the project. The Max+Plus II

message processor can be used to locate errors automatically and highlight them in the text editor window. After the project has compiled successfully, optional simulation and timing analysis with Max+Plus II software can be performed. The Compiler can also create Verilog output files and Standard Delay Format (SDF) output files for use with third-party simulation tools.

The designer can specify resource and device assignments for a Verilog design file to guide logic synthesis and fitting for the project or can choose to have the Compiler automatically fit the project into the best combination of devices from a target device family and assign the resources within them.

The Max+Plus II software supports a subset of the constructs defined by the IEEE Std 1364-1995, i.e., it supports only those constructs that are relevant to logic synthesis. A list of supported constructs can be found in Altera's Max+Plus II documentation.

## 14.2 Combinational Logic Implementation

Combinational logic circuits are commonly used in both the data path and control path of more complex systems. They can be modeled in different ways using continuous assignment statements which include expressions with logic, arithmetic and relational operators, and also can be modeled using if and case statements. Combinatorial logic is modeled in Verilog HDL also using always blocks that describe purely combinatorial behavior, i.e., behavior that does not depend on clock edges, by using procedural (sequential) statements. Both of these statements should be placed within the module body as in the following template:

```
module module_name (ports);
   ...
   [continuous_assignments]
   [always_blocks]

endmodule;
```

### *14.2.1 Logic and Arithmetic Expressions*

Both logic and arithmetic expressions may be modeled using logic, relational and arithmetic operators. The expressions take the form of continuous dataflow-type assignments.

## Logical Operators

Standard Verilog logical operators can be used to synthesize combinational circuits. Examples 14.1 and 14..2 correspond examples 11.1 and 11.2 of VHDL based synthesis using logic operators.

Example 14.1 Synthesizing logic from the language construct

```
module logic_operators (y, a, b, c, d);
    input a, b, c, d;
    output y;
    wire e;

    assign y = (a & b) | e;
    assign e = (c | d) ;

endmodule
```

Schematic diagram corresponding to this example is shown in Figure 11.1.

Example 14.2 Synthesizing logic from the language construct

```
module logic_operators_2 (y, a, b);
    input [3:0] a, b;
    output [3:0] y;

    assign y = a & b;

endmodule
```

Schematic diagram corresponding to this example is shown in Figure 11.2.

## Relational Operators

The simple comparisons operators ( **= =** and **/= =** ) are defined for all types. The resulting type for all these operators is Boolean. The simple comparisons, equal and not equal, are cheaper to implement (in terms of gates) than the ordering operators. To illustrate, Example 14.3 below uses an equality operator to compare two 4-bit input vectors. Corresponding schematic diagram is presented in Figure 11.3.

Example 11.3 Synthesizing logic from relational operators

```
module relational_operators_1 (y, a, b);
    input [3:0] a, b;
    output y;

    assign y = (a == b);

endmodule
```

Example 14.4 uses a greater-than-or-equal-to operator (‘>=’).

Example 14.4 Synthesizing logic from relational operators

```
module relational_operators_2 (y, a, b) ;
    input [3:0] a, b;
    output y;

    assign y = (a >= b);

endmodule
```

As it can be seen from the schematic corresponding to this example, presented in Figure 11.4, it uses more than twice as many gates as the previous example.

### Arithmetic Operators

Implementation of these operators is highly dependent on the target technology. Example 14.5 illustrates the use of arithmetic operators and parentheses to control synthesized logic structure.

Example 14.5 Using arithmetic operators

```
module arithmetic_operators (y1, y2, a, b, c, d);
    input [7:0] a, b, c, d;
    output [9:0] y1, y2;

    assign y1 = a+ b + c + d ;
    assign y2 = (a + b) + (c + d) ;
endmodule
```

Another possibility is to enclose signal assignment statements into an always block with all input signals in the sensitivity list of the always statement. From the synthesis point of view, there will be no difference. However, simulation can be simpler if the always block is used to describe the same circuit. Example 14.5 can be rewritten in that case and represented by the description given in Example 14.6.

### Example 14.6 Using always block to describe arithmetic circuit

```
module arithmetic_operators_1 (y1, y2, a, b, c, d) ;
    input [7:0] a, b, c, d;
    output [9:0] y1, y2;
    reg[9:0] y1, y2;

    always @(a or b or c or d)
    begin
     y 1 = a + b + c + d ;
     y2 = (a + b) + (c + d) ;
    end

endmodule
```

### *14.2.2 Conditional Logic*

Verilog provides two sequential statements for creating conditional logic:

- if statement, and
- case statement

Example 14.7 illustrate the use of the if statement for creating conditional logic.

### Example 14.7 Using conditional signal assignment

```
module condit_stmts_1 (y, a, b, sel);
    input [7:0] a, b;
    input sel;
    output [7:0] y;
    reg [7:0] y;

    always @(a or b or sel)
     begin
        if (sel==1)
```

```
            y = b;
        else
            y = a;
    end
endmodule
```

The schematic diagram of the circuit generated from the above examples is shown in Figure 11.5.

Example 14.8 shows the use of the case statement for creating of conditional logic that implements a multiplexer. All possible cases must be used for selected signal assignments. The designer can be certain of this by using an default case.

Example 14.8 Synthesizing multiplexer using selected signal assignment

```
module condit_stmts_2 (y, a, b, c, d, sel);
    input a, b, c, d;
    input  [1:0]  sel;
    output y;
    reg y;

    always @(sel or a or b or c or d)
        case(sel)
            0: y = a;
            1: y = b;
            2 : y = C;
            3: y = d;
        default: y = a;

        endcase
endmodule
```

Schematic diagram illustrating generated logic for examples 14.8 is shown in Figure 11.6.

### *14.2.3 Three-State Logic*

When data from multiple possible sources need to be directed to one or more destinations we usually use either multiplexers or three-state buffers. Three-state buffers are modeled in Verilog using conditional statements:

- if statements,
- case statements,
- onditional continuous assignments

A three-state buffer is inferred by assigning a high-impedance value 'Z' to a data object in the particular branch of the conditional statement. In the case of modeling multiple buffers that are connected to the same output, each of these buffers must be described in separate concurrent statement. Example 14.9 shows a four-bit three-state buffer.

Example 14.9 Synthesizing three-state buffer

```
module tbuf4 (y, a, enable);
    input [3:0] a;
    input enable;
    output [3:0] y;
    assign y= enable? a: 4'bZ;

endmodule
```

The same function can be achieved by using the equivalent if statement:

```
module tbuf4 (y, a, enable);
    input [3:0] a;
    input enable;
    output [3:0]yy;
    reg [3:0] y;

    always @(enable or a)
    if (enable == 1)
     y = a;
    else
     y= 4'bZ;

endmodule
```

Schematic diagram of the circuit corresponding to Example 14.9 is shown in Figure 11.7.

### *14.2.4 Examples of Standard Combinational Blocks*

In this subsection we will present a number of other standard combinational blocks and various ways to describe them in Verilog. These blocks usually represent units

that are used to form data paths in more complex digital designs. All of these designs are easily modifiable to suit the needs of specific application. Different approaches to modeling are used to demonstrate both versatility and power of Verilog.

Example 14.10 shows two different behavioral architectures of 8-to-3 encoder. The first architecture uses if statement while the second architecture uses a case statement within an always block. The use of the if statements introduces delays because the circuit inferred will evaluate expressions in the order in which they appear in the model (the expression at the end of the process is evaluated last). Therefore, the use of the case statement is recommended. It also provides a better readability.

Example 14.10 8-to-3 Encoder

```
module encoder83 (y, a);
    input [7:0] a;
    output [2:0] y;
    reg [2:0] y;

    always @(a)
    begin
     if (a == 8'b00000001) y = 0;
     else if(a == 8'b00000010) y = 1;
     else if (a == 8'b00000100) y = 2;
     else if (a == 8'b00001000) y = 3;
     else if (a == 8'b00010000) y = 4;
     else if (a == 8'b00100000) y = 5;
     else if (a == 8'b01000000) y = 6;
     else if (a == 8'b10000000) y = 7;
     else y = 3'bX;
    end

endmodule


module encoder83 (y, a) ;
    input [7:0] a;
    output [2:0] y;
    reg [2:0] y;
    always @(a)
    begin
    case(a)

     8'b00000001: y = 0;
     8'b00000010: y = 1;
```

```
     8'b00000100: y = 2;
     8'b00001000: y = 3;
     8'b00010000: y = 4;
     8'b00100000: y = 5;
     8'b01000000: y = 6;
     8'b10000000: y = 7;
     default: y = 3'bX;
    endcase
    end

endmodule
```

The following model of 8-to-3 priority encoder, presented in Example 14.11 uses for statement to describe its behavior, and valid output indicates that there is at least one input bit at logic level 1.

Example 14.11 Priority encodes 8-to-3

```
module priority83 (y, valid, a);
    output [2:0] y;
    output valid;
    input  [7:0]  a;
    reg [2:0] y;
    reg valid;

    integer N;

    always @ (a)
    begin
     valid =0;
     y = 3'bX;
     for (N=0; N<=7; N=N+1)
        if (a[N])
          begin
            y = N;
            valid = 1;
          end
    end
endmodule
```

Example 14.12 shows a 3-to-5 decoder described by a for loop statement.

Example 14.12 3-to-5 binary decoder with enable input

```
module decoder35 (y, a);
    input  [2:0]  a;
```

```
    output [4:0] y;
    reg [4:0] y;

    integer N;

    always @(a)
    begin
     for (N=0; N<=4; N=N+1)
        if (a == N)
          y[N] =1;
        else
          y[N] = 0;
    end

endmodule
```

Example 14.13 shows an address decoder that provides selection signals for segments of memory. Memory address space contains 1K locations represented by 10 address bits. First two segments have 256 locations each, and the third one 512 locations.

### Example 14.13 Address decoder implementation

```
module address_decoder (select0, select1, select2, address);
    output select0, select1, select2;
    input [9:0] address;

    reg select0, select1, select2;

    always @(address)
    begin
    //first segment
     if(address>=0 && address<=255)
        select0=1;
      else
        select0=0;

    //second segment
     if(address>=256 && address<=511)
        select1=1;
      else
        select1=0;

     //third segment
     if(address>= 512)
        select2=1;
      else
```

```
        select2=0;
     end
endmodule
```

Example 14.14 is introduced just to illustrate an approach to the description of a simple arithmetic logic unit (ALU) as a more complex combinational circuit. However, most of the issues in the design of the real ALUs are related to efficient implementation of basic operations (arithmetic operations such as addition, subtraction, multiplication, and division, shift operations, etc.). The ALU in this example performs operations on one or two operands that are received on two 8-bit busses (a and b) and produces output on 8-bit bus (f). Operation performed by the ALU is specified by operation select (opsel) input lines. Input and output carry are not taken into account. Operation codes are specified using the parameter statement that enables easy change of the code values at the beginning of the description.

Example 14.14 A simple arithmetic and logic unit

```
module alu (f, a, b, opsel);
   parameter addab = 4'b0000, inca = 4'b0001, incb = 4'b0010,
             andab = 4'b0011, orab = 4'b0100, nega = 4'b0101,
             shal = 4'b0110, shar = 4'b0111,
             passa = 4'b1000, passb = 4'b1001;

   output [7:0] f;
   input [7:0] a, b;
   input [3:0] opsel;
   reg [7:0] f;

   always @(a or b or opsel)
   begin
    case (opsel)
       addab: f = a + b;
       inca: f = a + 1;
       incb: f = b + 1;
       andab: f = a & b;
       orab: f = a | b;
       nega: f = !a;
       shal: f = a << 1;
       shar: f = a >> 1;
       passa: f = a;
       passb: f = b;
       default: f = 8'bX;
    endcase
   end
endmodule
```

## 14.3 Sequential Logic Synthesis

Verilog allows us to describe the behavior of a sequential logic element, such as a latch or flip-flop, as well as the behavior of more complex sequential machines. This section mostly follows Section 11.3 to show how to model simple sequential elements, such as latches and flip-flops, or more complex standard sequential blocks, such as registers and counters using Verilog. The behavior of a sequential logic element can be described using an always blocks because of their the sequential nature that makes them ideal for the description of circuits that have memory and must save their state over time. If our goal is to create sequential logic (using either latches or flip-flops) the design is to be described using one or more of the following rules:

4. Write the always block that does not include all module inputs in the sensitivity (event) list (otherwise, the combinational circuit will be inferred).

5. Use incompletely specified if-then-elseif logic to imply that one or more signals must hold their values under certain conditions.

6. Use one or more variables in such a way that they must hold a value between iterations of the always block.

### *14.3.1 Describing Behavior of Basic Sequential Elements*

the D-type flow-through latch, and
FPLD libraries, which synthesis tools map to, are:

- the D-type flow-through latch, and
- the D-type flip-flop

Some of the vendor libraries contain other types of flip-flops, but very often they are derived from the basic D-type flip-flop. Behavior of the both these circuits is described in section 11.3.1. In this section we consider the ways of creating basic sequential elements using Verilog descriptions.

There are three major methods to describe behavior of basic memory elements:

- using conditional if and case statements, or
- using a wait statement

The second method of using wait statement, however, is not supported by synthesis tools and will not be used in this presentation. Also, as there is no way to explicitly specify enable signal using case statement, it is better to avoid its use.

### *14.3.2 Latches*

Example 14.15 describes a level sensitive latch with an and function connected to its input. In all these cases the signal "y" retains it's current value unless the enable signal is '1'.

Example 14.15 A level sensitive latch

```
module latch1 (y, a, b, enable);
    output y;
    input a, b;
    input enable;
    reg y;

    always @(enable or a or b)
    begin
     if (enable)
        y = a & b; //blocking signal assignment
    end
endmodule
```

This example can be easily extended to inputs to the latch implementing any Boolean function or to those that have additional inputs such as asynchronous preset and clear. Example 14.16 shows a number of latches modeled within a single process. All latches are enabled by a common input enable.

Example 14.16 Latches implemented within a single process

```
module latches (y1, y2, y3,enable,
          a1, preset1,
          a2, clear2,
          a3, preset3, clear3);

    output y1, y2, y3;
    input a1, preset1;
    input a2, clear2;
    input a3, preset3, clear3;
    input enable;
```

```
    reg y1, y2, y3;

    always @(enable or a1 or a2 or a3 or
       preset1 or clear2 or preset3 or clear3)
    begin
     if (preset1)
       y1 = 1;
     else if (enable)
       y1 = a1;

     if (clear2)
       y2 = 0;
     else if(enable)
       y2 = a2;

     if (preset3)
       y3 = 1;
     else if (clear3)
       y3 = 0;
     else if (enable)
       y3 = a3;

    end
endmodule
```

### *14.3.3 Registers and Counters Synthesis*

A register is implemented implicitly with a register inference. Register inferences in Max+Plus II Verilog support any combination of clear, preset, clock, enable, and asynchronous load signals. The Compiler can infer memory elements from edge-triggered always statement. The Verilog always statement is edge-triggered by including posedge or negedge clause in the sensitivity list. However, a single always block can model only either purely sequential or purely combinational logic. So, to model purely combinational logic, a separate always block must be used. Also, when an asynchronous clear or preset flip-flop is modeled, a second posedge or negedge clause must be used in the sensitivity list of the always statement as it is shown in Example 14.17. Example 14.24 shows several ways to infer registers that are controlled by a clock and asynchronous clear, preset, and load signals.

Example 14.17 Inferring registers

```
module register_inference (q1, q2, q3, q4, q5,
         d, clk, clear, preset, load);
```

```
    output q1, q2, q3, q4, q5;
    input d, clk, clear, preset, load;

    reg q1, q2, q3, q4, q5;


    // register with active-low clock
    always @(negedge clk)
     q1 = d;

    // register with active-high clock and  asynchronous clear
    always @(posedge clk or posedge clear)
     if (clear)
        q2 = 0;
     else
        q2 = d;

    // register with active-high clock and asynchronous preset
    always @(posedge clk or posedge preset)
     if (preset)
        q3 = 1;
     else
        q3 = d;

    // register with active-high clock and synchronous load

    always @(posedge clk)
     if (load)
        q4 = d;
     else
        q4 = q4;

    // register with active-low clock and asynchronous clear
    // and preset

    always @(negedge clk or posedge clear or posedge preset)
     if (clear)
        q5 = 0;
     else if(preset)
        q5 = 1;
     else
        q5 = d;

endmodule
```

A counter can be implemented with a register inference. A counter is inferred from an if statement that specifies a clock edge together with logic that adds or subtracts a value from the variable. The if statement and additional logic should be inside an always statement. Example 14.18 shows several 8-bit counters controlled

by the clk, clear, ld, d, enable, and up_down signals that are implemented with if statements.

Example 14.18 Inferring counters

```
module counters (qa, qb, qc, qd, qe, qf,
          d, clk, enable, clear, load, up_down);

    output [7:0] qa, qb, qc, qd, qe, qf;
    input  [7:0] d;
    input clk, enable, clear, load, up_down;

    reg [7:0] qa, qb, qc, qd, qe, qf;

    integer direction;

    // An enable counter
    always @(posedge clk)
    begin
     if (enable)
        qa = qa + 1;
    end


    //A synchronous load counter
    always @(posedge clk)
    begin
     if (load)
        qb = d;
     else
        qb = qb + 1;
    end

    // A synchronous clear counter
    always @(posedge clk)
    begin
     if (clear)
        qc = 0;
    else
     qc = qc + 1;
    end


    //An up/down counter
    always @(posedge clk)
    begin
     if (up_down)
```

```
      direction = 1;
    else
      direction = -1;
      qd = qd + direction;
  end

  // A synchronous load clear counter
  always @ (posedge clk)
  begin
    if (clear)
      qe = 0;
    else if (load)
      qe = d;
    else
      qe = qe + 1;
  end

  // A synchronous load enable up/down counter
  always @(posedge clk)
  begin
    if (up_down)
      direction = 1;
    else
      direction = -1;

    if (load)
      qf = d;
    else if (enable)
      qf = qf + direction;
  end

endmodule
```

All always statements in this example are sensitive only to changes on the clk input signal. All other control signals are synchronous.

### *14.3.4 Examples of Standard Sequential Blocks*

Example 14.19 demonstrates design of 16-bit counter which allows initialization to zero value (reset), and control of the counting by selection of counter step: incrementing for 1 or 2 and decrementing for 1. It also demonstrates the use of various data types.

Example 14.19 16-bit counter wit enable input and additional controls

```
module flexcount16 (q, up1, up2, down1, clk, enable, clear,
load, d);

    output [15:0] q;
    input up1, up2, down1, clk, enable, clear, load;
    input [15:0] d;

    reg [15:0] q;

    integer direction;

    always @(posedge clk or posedge clear)
    begin
     if ((up1 == 1) & (up2==0) & (down1==0))
        direction = 1;
     else if ((up1 == 0) & (up2==1) & (down1==0))
        direction = 2;
     else if ((up1 == 0) & (up2==0) & (down1==1))
        direction = -1;
     else
        direction = 0;

     if (clear)
        q = 16'b0000_0000_0000_0000;
     else if (load)
        q = d;
     else if (enable)
        q = q + direction;
    end

endmodule
```

Example 14.20 demonstrates how a frequency divider (in this case divide by 11). The output pulse must occur at the 11th pulse received to the circuit.

Example 14.20 Frequency Divider

```
module divider11 (clkdiv11, clk, reset);
    output clkdiv11;
    input clk, reset;
    reg clkdiv11;
    reg [3:0] cnt;
    reg n;
```

```
    always @(posedge clk or posedge reset)
    begin
    if (reset)
     begin
      cnt = 0;
      n = 0;
     end
    else
     cnt = cnt + 1;
    if (cnt ==11)
     n = 1;
    else
     n = 0;
    if (n == 1)
     cnt = 0;

    end

    always @(n)
     clkdiv11 = n;

endmodule
```

Timer is a circuit that is capable of providing very precise time intervals based on the frequency (and period) of external clock (oscillator). Time interval is obtained as a multiple of clock period. The initial value of the time interval is stored into internal register and then by counting down process decremented at each either positive or negative clock transition. When the internal count reaches value zero, the desired time interval is expired. The counting process is active as long as external signal enable controlled by external process is active. Block diagram of the timer is presented in Figure 11.16. Verilog description of the timer is given in Example 14.21.

Example 14.21 Behavioral description of timer

```
module timer (timeout, clk, load, enable, data);
    output timeout;
    input clk, load, enable;
    input [15:0] data;
    reg timeout;
    reg [15:0] cnt;

    always @(posedge clk)
    begin
    if (load & !enable)
```

```
    cnt = data;
   else if (!load & enable)
    cnt = cnt - 1;
  else
    cnt = cnt;

   if (cnt == 0)
    timeout = 1;
   else
    timeout = 0;
   end

endmodule
```

## 14.4 Finite State Machines Synthesis

Finite State Machines (FSMs), as shown in Chapter 4, represent an important part of design of almost any more complex digital system. In this section we only mention some specifics of description FSMs in Verilog. As we have already seen, the code describing an FSM can be structured into three parts corresponding to next state logic, current state and output logic. These parts can be grouped in different ways when described in an HDL. The next state logic is best modeled in Verilog using case statement. The default clause used in case statement avoids the need to explicitly define all possible combinations of state variables as they are usually not a part of the FSM. The way output logic is modeled depends weather we use Moore or Mealy type FSM and will be shown in the following sections. As most FSMs require facility to bring the FSM to a known initial state, an asynchronous or synchronous reset can be used for this purpose. In Verilog only the if statement can be used to describe behavior of this type, and in the case of asynchronous reset it must be included in the sensitivity list of the always statement with posedge or negedge clause. For description of states and state encoding the parameter statement can be used as it allows changes of state assignment at a single place, if required. In this section we will illustrate description of Moore and Mealy FSMs using Verilog on the same examples used for presentation of FSM description in VHDL.

### *14.4.1 Verilog FSM Example*

FSM presented in Example 14.22 is similar to one described in VHDL in Example 11.32 with the difference that an asynchronous reset signal, whenever is activated, brings the FSM to the initial known state. The FSM has a single control input (up_down), two outputs (lasb and msb) and reset input. It can be in four states that are assigned binary values using the parameter statement. Internal variables present_state and next_state are used to describe state transitions. State transitions

are described using the case statement within the always block that is activated whenever a change on input control signal or present state occurs. Another always statement is used to synchronize state transitions with the clock (on posedge event) or to bring the FSM into initial state (when reset occurs).

Example 14.22 FSM with four states

```
module state_machine (lsb, msb, up_down, clk, reset);
    output lsb, msb;
    input up_down, clk, reset;

    parameter [1:0] st_zero = 2'b11, st_one = 2'b01,
            st_two = 2'b10, st_three = 2'b00;

    reg lsb, msb;

    reg [1:0] present_state, next_state;

    always @(up_down or present_state)
    //Combinational part
    begin
    case (present_state)

     st_zero: if (up_down == 0)
          begin
            next_state = st_one;
            lsb = 0;
            msb = 0;
          end
          else
          begin
            next_state = st_three;
            lsb = 1;
            msb = 1;
          end

     st_one: if (up_down == 0)
          begin
            next_state = st_two;
            lsb = 1;
            msb = 0;
          end
          else
          begin
            next_state = st_zero;
            lsb = 0;
            msb = 0;
```

```
          end

     st_two: if (up_down == 0)
          begin
           next_state = st_three;
           lsb = 0;
           msb = 1;
          end
          else
          begin
           next_state = st_one;
           lsb = 1;
           msb = 0;
          end

     st_three: if (up_down == 0)
          begin
           next_state = st_zero;
           lsb = 1;
           msb = 1;
          end
          else
          begin
           next_state = st_two;
           lsb = 0;
           msb = 1;
          end
    endcase
    end

    //Sequential part
    always @(posedge clk or posedge reset)
    begin
    if (reset)
     next_state = st_zero;
     else
     present_state = next_state;
    end
endmodule
```

### 14.4.2 Moore Machines

A Moore state machine has the outputs that are a function of the current state only. The general structure of Moore-type FSM is presented in Figure 14.20 with two functional blocks that can be implemented as combinational circuits:

- next state logic, which can be represented by function next_state_logic, and
- output logic, which can be represented by function output_logic

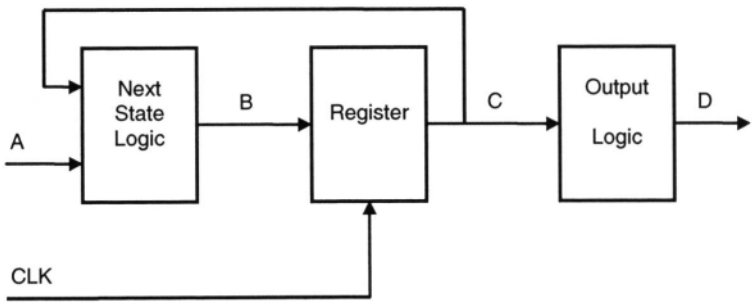



Figure 14.20 Moore-type FSM

Outputs of both of these functions are the functions of their respective current inputs. The third block is a register that holds the current state of the FSM. The Moore FSM can be represented by three always statements each corresponding to one of the functional blocks:

```
module moore1 (d, a, clk)
   output d;
   input a;
   input clk;
   reg d;

   reg b; // output from next state logic
   reg c; // present state

   function [..] next_state_logic;

    input a, c;
    begin

      describe mapping of a, c onto next_state_logic

    end
   endfunction

   function [..] output_logic;
```

```
    input c;
    begin

      describe mapping of c onto output_logic

    end

   // next state generation
   always @(a or c)
   begin
    b = next_state_logic(a, c);
   end

   // system output
   always @(c)
   begin
    d = output_logic(c);
   end

   // state transition
   always @(posedge clk)
   begin
    C = b;
   end
endmodule
```

A more compact description of this architecture could be written as follows:

```
module moore2 (d, a, clk)
   output d;
   input a;
   input clk;
   reg d;

   reg c; // present state

   function [..] next_state_logic;

    input a, c;
    begin

      describe mapping of a, c onto next_state_logic
```

```
    end
  endfunction

  function [..] output_logic;

    input c;
    begin

      describe mapping of c onto output_logic

    end

  // system output
  always @(c)
  begin
    d = output_logic (c);
  end

  // state transition
  always @(posedge clk)
  begin
    c = next_state_logic(a, c);
  end

endmodule // moore2
```

In fact, a Moore FSM can often be specified in a single process. Sometimes, the system requires no logic between system inputs and registers, or no logic between registers and system outputs. In both of these cases, a single process is sufficient to describe behavior of the FSM.

In both these models functions are used to describe generation of the next state and output from the circuit. They may be implemented using any procedural statements that combine inputs and local variables to function to form their output. Brackets [..] are used to denote dimension (range of bits) of the output value. Always blocks are used to separate description of combinational logic blocks and sequential logic blocks that make the FSM.

### *14.4.3 Mealy Machines*

A Mealy FSM has outputs that are a function of both the current state and primary system inputs. The general structure of the Mealy-type FSM is presented in Figure 14.2.

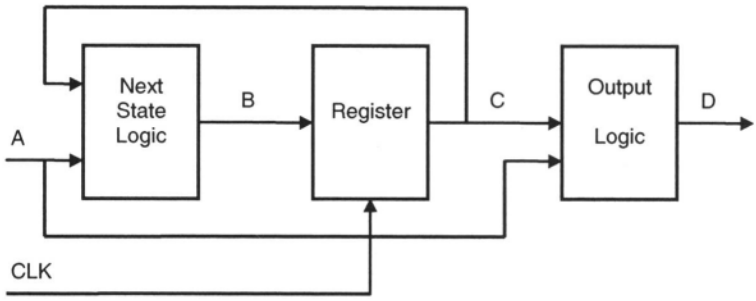



Figure 11.21 Mealy-type FSM

The Mealy FSM can be represented by the following general Verilog model, similar as Moore machines:

```
module mealy (d, a, clk)
   output d;
   input a;
   input clk;
   reg d;
   reg c; // present state

   function [..] next_state_logic;
    input a, c;
    begin

      describe mapping of a, c onto next_state_logic

    end
   endfunction

   function [..] output_logic;
    input a, c;
    begin

      describe mapping of a, c onto output_logic
    end
   endfunction

   always @(posedge clk)
    begin

      c = next_state_logic(a, c);
```

```
    end

   always @(a,  c)
    begin

      d = output_logic(a, c) ;
    end

endmodule // mealy
```

It contains at least two always blocks, one for generation of the next state, and the other for generation of the FSM output.

## 14.5 Hierarchical Projects

Verilog design file can be combined with other Verilog design files, and other design files from various tools into a hierarchical project at any level of project hierarchy. Discussion from Section 11.5 on hierarchical VHDL projects is practically completely applicable to Verilog projects.

Besides Verilog primitives, Max+Plus II design environment provides a number of other primitives and bus, architecture-optimized, and application-specific macrofunctions and library of parameterized module (LPM) functions. The designer can use component instantiation statements to insert instances of primitives, macrofunctions and LPMs, as well as previously defined user components. The range of these functions was presented in tables 11.2 to 11.5, and a more complete list can be found in corresponding Altera documents including help files in Max+Plus II environment. In this section we show on a number of simple examples how those components can be instantiated. The purpose of this presentation is just introducing corresponding Verilog syntax and mechanics of the instantiation

### *14.5.1 User Defined Functions*

Verilog allows to create user defined functions. Any Verilog design can become a user defined function after compilation and generation of AHDL include (.inc) file. Example 14.23 shows reg8.v, an 8-bit register design. After you create an AHDL include file, reg12.v can be instantiated in a Verilog design file that is higher in the project hierarchy.

Example 14.23 8-bit register

```
module reg8 (q, d, ena, clk);

    output [7:0]q;
    input  [7:0]  d;
    input ena,  clk;
    reg  [7:0]  q;

    always @ (posedge clk)
    if  (ena)
     q = d;
endmodule
```

Example 14.24 shows reg24.v, a Verilog design that declares reg24, then instantiates the reg8 function without requiring any module declaration. Three instances of reg8 are named regA, regB and regC. During design processing, the MAX+PLUS II Verilog netlist reader automatically refers to reg8.inc for information on port names and their order.

Example 14.24 24-bit register using instances of 8-bit register

```
module reg24 (out, data, enable, clk);

    input    [23:0] data;
    input   enable, clk;
    output  [23:0] out;
    reg8 regA (.q (out[7:0]), .d (data[7:0]), .ena(enable),
      .clk (clk));
    reg8 regB (.q (out[15:8]), .d (data[15:8]), .ena(enable),
      .clk (clk));
    reg8 regC (.q (out[23:16]), .d (data[23:16]),
      .ena(enable), .clk (clk));

endmodule
```

### *14.5.2 Using Parameterized Modules and Megafunctions*

Altera provides another abstraction in the form of library design units which use parameters to achieve scalability, adaptability, and efficient silicon implementation. By changing parameters a user can customize design unit for a specific application. For example, in memory functions, parameters are used to determine the input and output data widths; the number of data words stored in memory; whether data

inputs, address/control inputs, and outputs are registered or unregistered; whether an initial memory content file is to be included for a RAM block; and so on. The designer must declare parameter names and values for RAM or ROM function by using generic map aspects. Example 14.25 shows a 512 x 8 bit lpm_ram_dq function with separate input and output ports.

Example 14.25 Using memory function

```
module ram256x8 (dataout, datain, address, we, inclock,
    outclock);

output[7:0] dataout;
input[7:0] datain;
input [8:0] address;
input we, inclock, outclock;
lpm_ram_dq ramA (.q (dataout), .data (datain), .address
       (address), .we (we), .inclock (inclock),
       .outclock (outclock));

   defparam ramA.lpm_width = 8;
   defparam ramA.lpm_widthad = 9;

endmodule
```

The designer assigns values to all parameters in the logic function instance using Altera-specific defparam statement. Some parameters do not require user-defined value. If no value is specified for a parameter, the Compiler searches for a default value in the parameter value search order.

## 14.6 Questions and Problems

14.1 Under what conditions does a typical synthesizer generates a combinational circuit from the Verilog always block? When a sequential circuit will be synthesized from the Verilog always block?

14.2 Under what conditions does a typical synthesizer generates a combinational circuit from the VHDL process?

14.3 A 64K memory address space is divided into eight 8K large segments. Using Verilog describe an address decoder that decodes the segment from the 16-bit address.

14.4 The address space from the preceding example is divided into seven segments of equal length (8K) and the topmost segment is divided into four segments of 2K size. Using Verilog describe an address decoder that decodes the segment from the 16-bit address. Describe a decoder using always block and different procedural statements. Make at least two different descriptions of the decoder.

14.5 Describe a J-K and T flip-flops using the Verilog always block.

14.6 Using templates for Mealy and Moore-type FSMs describe flip-flops from the preceding problem.

14.7 Apply templates for Mealy and Moore-type FSMs to the example of a system that describes driving the car with four states: stop, slow, mid, and high, and two inputs representing acceleration and braking. The output is represented by a separate indicator for each of the states. The states are coded using a one-hot encoding scheme. What is the difference if you apply different state encodings scheme (do it for sequential binary, Johnson and Gray encoded states).

14.8 Describe in Verilog a generic synchronous n-bit up/down counter that counts up-by-p when in up-counting mode, and counts down-by-q when in down-counting mode. Using this model instantiate 8-bit up-by-one, down-by-two counter.

14.9Describe an asynchronous ripple counter that divides an input clock by 32. For the ripple stages the counter uses a D-type flip-flop whose output is connected back to its D input such that each stage divides its input clock by two. For description use behavioral-style modeling. How would you modify the counter to divide the input clock by a number which is between 17 and 31 and cannot be expressed as $2^k$ (k is an integer).

14.10Design a parameterized frequency divider that divides input clock frequency by N, and provides the duty cycle of the generated clock of duration M (M<N-1) cycles of the input clock.

14.11 Repeat all problems from Section 5.9 (problems 5.1 to 5.21). Instead of AHDL use Verilog. Compare your designs when using different hardware description languages. How solutions compare to those using VHDL?

# 15 A VERILOG EXAMPLE: PIPELINED SIMP

In this chapter we present an enhanced version of SimP microprocessor introduced in Chapter 7. In some applications custom computing machines implemented in FPLDs require high performance. By using pipelining as an architectural solution that employs instruction parallelism original SimP almost triples its performance with practically the same FPLD resources. This can bee achieved with relatively small modifications of original SimP. In this chapter we describe necessary modifications of the original processor and present most of the design descriptions using Verilog HDL.

## 15.1 SimP Pipelined Architecture

Original SimP is a 16-bit custom-configurable microprocessor. Its architecture is based on the traditional von Neumann model with a single address space and memory used to store both programs and data. The SimP core instructions are executed as sequences of micro-operations, each instruction cycle consisting of four machine cycles, which perform the three major steps: instruction fetch, instruction decode and instruction execution. Consequently, one instruction is completed after each four machine cycles resulting in a relatively low instruction throughput and low utilization of hardware resources.

The approach to achieve a speedup and enhance the performance of a processor can be to shorten the machine cycle time by using faster hardware elements and/or to reduce the number of cycles per instruction (increase instruction throughput) by using some more efficient processing algorithm. The basic way to reduce the number of cycles per instruction is to exploit instruction level parallelism.

Instruction pipelining is an implementation technique that achieves instruction parallelism by overlapping instruction fetching, decoding and execution. In this technique, the pipelined processor consists of a sequence of $m$ processing stages, through which a stream of instructions can be passed. Every instruction is broken down into $m$ partial steps for execution in the $m$ stage pipelining. Partial processing

of the instructions takes place in each stage. Final fully processed result is obtained only after an instruction has passed through the entire pipeline. The partial steps are executed within a single machine cycle; consequently one instruction result is available with each machine cycle except for the first couple and ending instructions. Figure 15.1 illustrates the difference between non-pipelined and pipelined instruction execution.

| Machine Cycle | Stage 1 Instruction Fetch | Stage 2 Instruction Decode | Stage 3 Instruction execution | Instruction completed |
|---|---|---|---|---|
| T | Instruction $I_i$ | Instruction $I_{i-1}$ | Instruction $I_{i-2}$ | Instruction $I_{i-2}$ |
| T+1 | Instruction $I_{i+1}$ | Instruction $I_i$ | Instruction $I_{i-1}$ | Instruction $I_{i-1}$ |
| T+2 | Instruction $I_{i+2}$ | Instruction $I_{i+1}$ | Instruction $I_i$ | Instruction $I_i$ |
| T+3 | Instruction $I_{i+3}$ | Instruction $I_{i+2}$ | Instruction $I_{i+1}$ | Instruction $I_{i+1}$ |

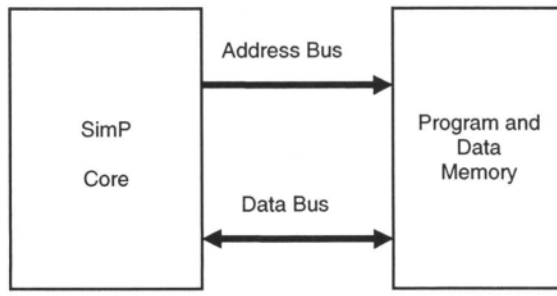
Figure 15.1 Instruction pipelining with 3 pipeline stages

When designing a pipelined processor the first task is finding a suitable multistage sequential algorithm for computing the target function. Due to SimP simple architecture, a three-stage instruction pipelining can be implemented by partitioning the instruction cycle into three stages: fetch stage, decode stage and execution stage as shown in Figure 1. However, as we will see in the following sections, there are some problems in implementing the instruction pipeline that require modifications of the original SimP data path and control mechanism.

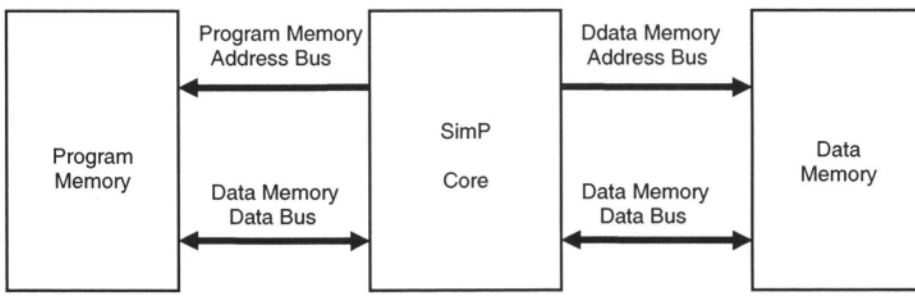
### *15.1.1 Memory conflicts*

Von Neumann architecture adopted in the original SimP requires both instructions and data to be stored in the same memory. Obviously it is difficult to implement instruction pipelining to a processor adopting this model since all pipeline stages are simultaneously active which may cause request for simultaneous access to the memory by two pipeline stages. For example the instruction fetch stage can request

reading and instruction execution stage reading or writing at the same time. This problem can be resolved by adopting Harvard architecture, which uses two separate memories for instructions and data. Therefore, a new program memory should be introduced to store instructions with separate data and address buses. Both architectures of original and pipelined SimP are illustrated in Figure 2.




a) Original SimP von Neumann architecture




b) Pipelined SimP Harvard architecture

Figure 2 Original and pipelined SimP architectures

### *15.1.2 Stage Registers*

Pipeline stages have to be separated by stage registers in order to store the intermediate result from one stage and make it available to the next stage. Figure 3 illustrates the connections between pipeline stages using the stage registers in pipelined SimP.

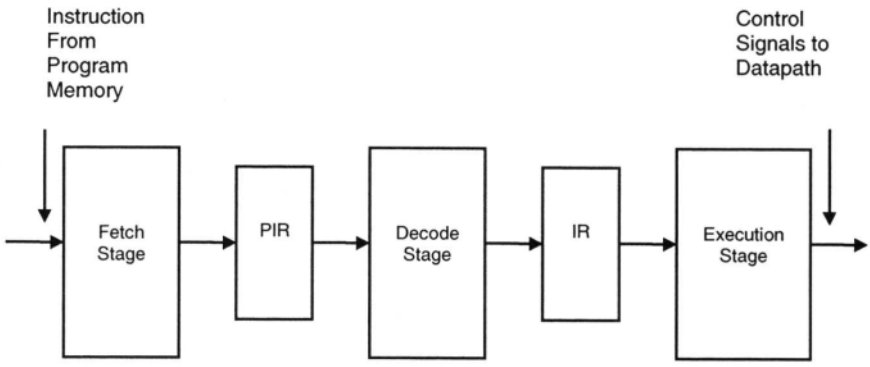



Figure 3 Pipeline stages and stage registers

### *15.1.3 Branching Instructions*

Although instruction pipelining can increase instruction throughput, there still remain some critical instruction sequences that can not be pipelined (overlapped or partitioned). These sequences usually consist of data and control dependencies.

An example of these instructions is a branching instruction. When an instruction $I_i$ with address $A_i$ is being executed by the execution stage, the instruction $I_{i+1}$ with the next consecutive address $A_{i+1}$ is being decoded in decode stage while the instruction $I_{i+2}$ with the next consecutive address $A_{i+2}$ is fetched from program memory by the fetch stage. Except when $I_i$ is a branch instruction causing a jump to an address $A_j \neq A_{i+2}$, $I_{i+1}$ is the next instruction required by the execution stage. If $I_i$ happens to be a branch to a nonconsecutive address, the instruction $I_{i+1}$ that has been predecoded and the instruction $I_{i+2}$ that has been prefetched during the execution of $I_i$ have to be discarded. As a result, decode and execution stages should be cleared, a new instruction fetch cycle must be initiated by fetch stage and therefore, the pipeline must be disabled.

## 15.2 Pipelined SimP Design

In this section we present all major design changes to the original SimP and show how they are implemented. In addition we present practically full implementation in Verilog. First we concentrate on SimP data path and then requirements on control mechanisms.

### *15.2.1 Data Path*

The pipelined SimP data path requires a number of modifications. The new data path is illustrated in Figure 15.4.

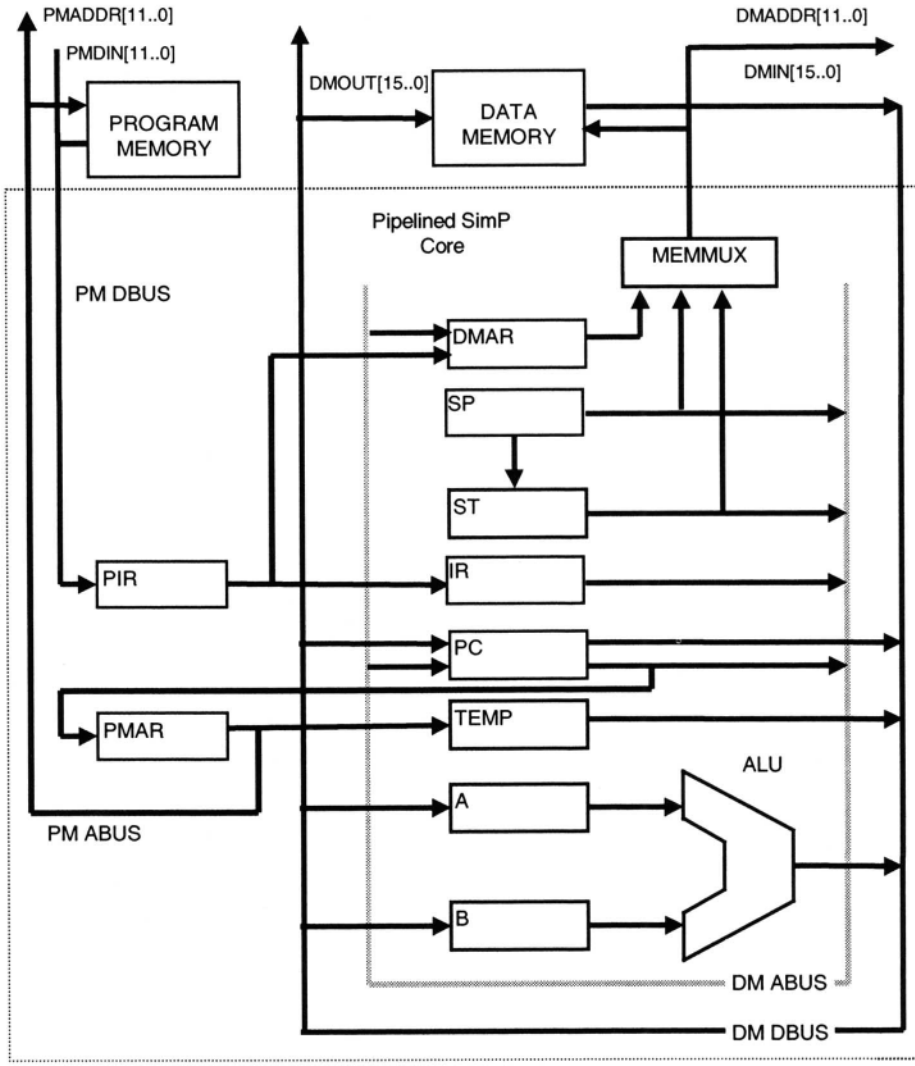



Figure 15.4 Pipelined SimP data path

Due to the adoption of Harvard architecture, additional memory is used as program memory in pipelined SimP. As this memory has as primary function to store programs (instructions), it is sufficient to provide a single program memory address register (PMAR) to hold an effective address, and instruction data register to store instruction read from program memory. As there are two separate physical memories, 4K 16-bit locations each,, the total memory capacity compared to the original SimP is doubled.

Looking at the entire data path, it is obvious that the changes in the data path are minimal and require small additional resources compared to the original SimP. These changes are discussed in the following paragraphs.

The role of the program counter (PC) and program counter temporary register (TEMP) is changed. PC is used to point to the next instruction to be fetched by fetch stage, while TEMP holds the address of the instruction being decoded in decode stage (next instruction to be executed) in order to be prepared for an interrupt request signal if it occurs.

Stack pointer (SP) and temporary stack pointer (ST) are used to implement stack and support related operations and mechanisms such as subroutine, interrupt and return mechanism, and instructions to push to and pull from the stack. SP always points to the next available (free) location on the stack, while ST is used in original SimP to hold a copy of SP value. SimP updates these values after each instruction cycle (four machine cycles). This is not allowed in pipeline SimP as everything should be done in one machine cycle. Therefore, ST function is changed to point to the last used location on the stack (always equal SP+1 as the stack grows towards lower addresses). Initial contents of these registers is implementation dependent and depends on the memory actually present in the pipelined SimP (for example SP is loaded with H"FEF", and ST with H"FEF" if full memory is present in the system). The detailed design of the ST register is shown in the following sections.

Two 12-bit address registers are required in pipelined SimP. They are called data memory address register (DMAR) and program memory address register (PMAR). The PMAR contains the address of the instruction that is ready to be fetched, while DMAR contains the data memory address in the case of instructions with direct addressing mode, or have no meaning in the case of instructions which are using stack addressing mode.

Two registers hold instruction codes of instructions that are in the instruction pipeline. Instruction register (IR) holds the instruction that is ready to be executed. It is connected to the operation decoder, which decodes operation codes and provides input signals to the control unit. Another prefetched instruction register (PIR) is a new stage register used to hold the prefetched instruction that is ready to be decoded in the next machine cycle.

The implementation of the pipelined SimP presented in this chapter is based on the use of internal FLEX10K FPLD memory, but also supports external memory connection. Two small internal memory blocks of 256 16-bit words are built into the processor data path. One of them serves as program memory ROM, and the other one as data RAM. Appropriate address decoders are used to differentiate accesses to internal memories from the accesses to the external memories by placing the internal memories to the lowest addresses of both program and data address space.

### *15.2.2 Control Unit*

Major changes have to be made in the SimP control mechanisms in order to support pipelining and also provide proper operation in the situations that can be considered as exceptions (initialization at power-up and reset, interrupts handling and program flow change when branch is taken in branch instructions). These changes are discussed in more details in the following paragraphs. Processor control flow is summarized by the flow diagram shown in Figure 15.5 that describes the global processor behavior and serves as the basic guideline for implementation of the control unit FSM.

#### Pulse Distributor

In the original SimP, pulse distributor generates four non-overlapping phases from the system clock. For its normal pipelined operation the pipelined SimP does not require this as all pipeline stages are simultaneously active and the instruction fetching, decoding and execution are overlapped. However the processor still requires four identifiable machine cycles to initialize pipeline stages at the system power-up, reset, or after some instructions that require to disable the pipeline processing such as branching and return instructions that can be considered as a kind of exceptions. Also it requires the same number of cycles to perform a jump on address specified in interrupt vector when interrupt cycle is carried out. To implement this, the pulse distributor can be preserved or the required actions can be built in into the control unit FSM. In the design presented below we decided to depart from the original SimP pulse distributor and build in control into the control unit FSM.

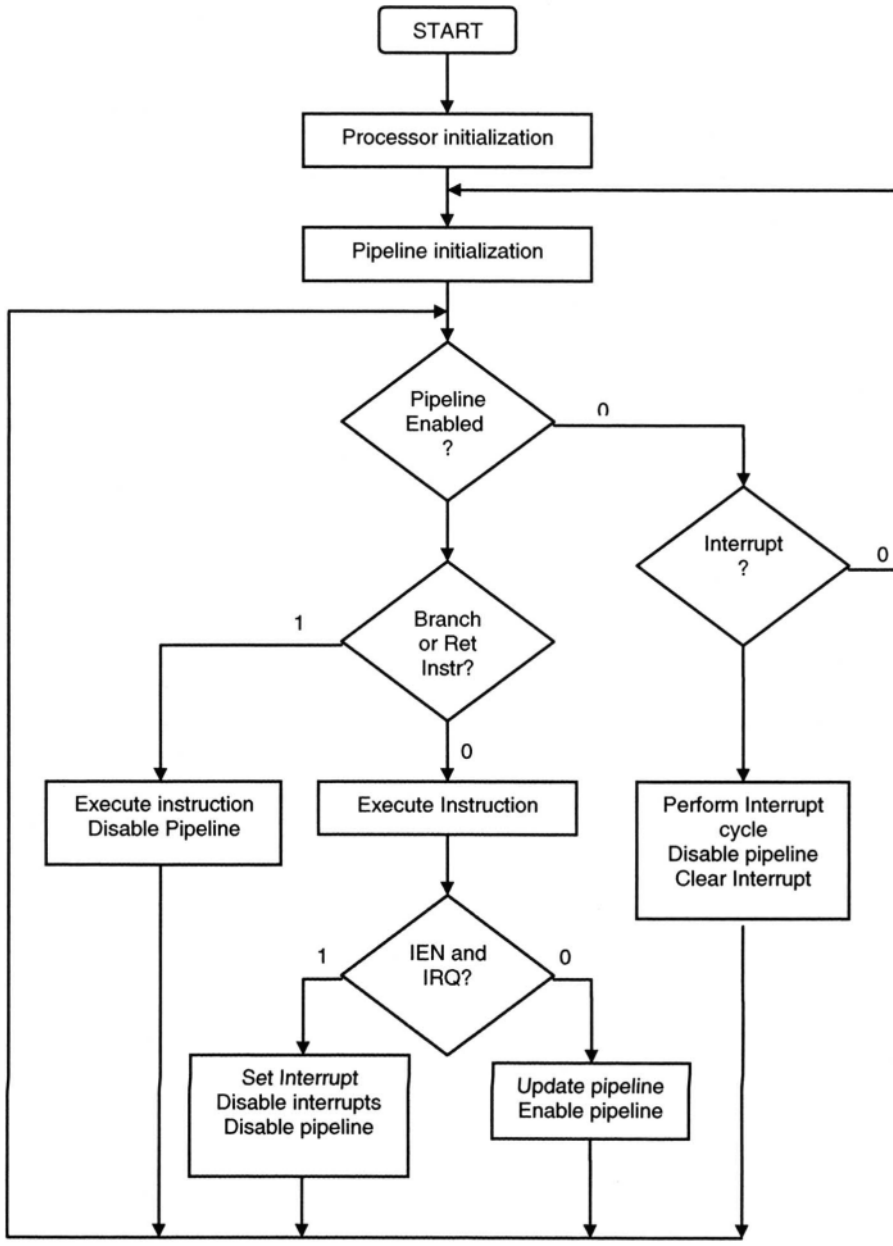



Figure 15.5 Pipelined SimP control flow

**Processor and Pipeline Initialization**

After power-up, the processor passes initialization steps in order to initialize program counter PC, stack pointer SP and temporary stack pointer ST. After processor initialization, the control is transferred to the pipeline operation that is presented by a number of separate states of the control unit FSM. The same action of processor initialization must be performed at the processor reset. Register transfers presented in Table 15.1 describe operations that take part in processor initialization. From the table we also see that interrupt control flip-flop (only interrupt enable IEN flip-flop is used), which has the same meaning as for the original SimP, must be initialized. The interrupt structure is immediately enabled, although different solutions are obviously possible. Initialization itself is presented by two FSM states to enable stabilization of signals before pipeline initialization starts.

Table 15.1 Processor initialization operations

| Operations |
| --- |
| PC ← "Initial program address"<br>SP ← "Initial value of the stack pointer – bottom of the stack"<br>ST ← "Initial value of ST, one higher than SP"<br>IFF ← 0; No interrupts to process<br>IEN ← 1; Interrupts enabled |

After the initialization, processor enters instruction execution cycle, which always starts with pipeline initialization. Pipeline initialization requires four machine cycles. The first cycle is used to initialize program memory address register to a value representing new current instruction and program counter to take value of address of the next instruction to fetch. Other three cycles are necessary to start feeding pipeline stages until pipeline becomes full. When the pipeline stages are full, the pipeline is enabled and pipeline starts to operate. The pipeline initialization operations are described in Table 15.2. Each cycle is assigned a single state in the control unit FSM. Pipeline initialization cycle is executed not only after processor initialization, but also after each event that is considered exceptional (branch instructions and interrupts).

Table 15.2 Pipeline initialization operations

| Cycle/ FSM state | Operations |
|---|---|
| $T_0$/plinit0 | PMAR ← $A_i$; $A_I$ is the address where instruction $I_I$ is stored<br>PC ← $A_{i+1}$; next instruction address in PC |
| $T_1$/plinit1 | IDR ← $I_i$; Instruction $I_i$ is fetched.<br>PMAR ← $A_{i+1}$; prepare fetching of the next instruction<br>PC ← $A_{i+1}$; next instruction address in PC |
| $T_3$/plinit2 | IR ← IDR; instruction $I_i$ being decoded |
| $T_4$/plinit3 | $I_i$ instruction is stored in DMAR (only lower 12 bits)<br>$I_{i+1}$ instruction being decoded.<br>$I_{i+2}$ instruction is fetched<br>TEMP ← $A_{i+2}$;<br>PMAR ← $A_{i+2}$<br>PC ← $A_{i+3}$<br>PLCR ← 1; request to enable pipeline |

**Pipelined Instruction Execution**

When the control unit detects that the pipeline initialization has been finished, it activates pipelined operation in order to start normal pipelined instruction execution. There is an instruction (in IR) ready to be executed and the twelve least significant bits from the current instruction (which represent an address in the case of instructions with direct addressing mode) are in data memory address register (DMAR). Also, there is a prefetched instruction (in PIR) ready to be decoded. Other registers contain values prepared for the next instruction execution. At the end of execution of instruction $\mathbf{I_i}$, a number of steps to update the pipeline stages, unless $\mathbf{I_i}$ is a branch instruction with a branch to be taken or if it is a return instruction, have to be undertaken and they are described in Table 15.3.

Table 15.3 Operations to update the pipeline

| Operations |
| --- |
| Twelve least significant bits of instruction $I_{i+1}$ are stored in DMAR<br>$I_{i+2}$ instruction is decoded<br>$I_{i+3}$ instruction is fetched<br>$A_{i+3}$ address is stored in PMAR<br>$A_{i+4}$ address is stored in PC<br>PLCR ← 1; pipeline enabled |

If the instruction in IR being executed is a branch (JUMP or JSR) or return (RET) instruction or an interrupt signal is detected, processor will execute current instruction and disable pipeline operation in order to return to the pipeline initialization.

### Branching Instructions

If the instruction in the execution stage happens to be a branch instruction (JMP, JSR, or RET), the pipeline must be disabled and the decode and execution stages cleared. A new instruction fetch cycle is initiated by fetching the instruction at the address specified by the branch destination address and the pipeline initialization has to be performed again. During JSR execution, the address of instruction in decode stage is stored on the stack.

The same happens if the instruction in execution stage is a return instruction. The only difference is that the new fetch cycle is initiated by fetching the instruction addressed by ST, i.e. control returns to the next address in the main program before JSR was executed or interrupt happened.

### Interrupt Handling

The original SimP checks for hardware interrupt at the end of each instruction cycle, i.e. after every four machine cycles. When an external device generates an interrupt request, and under the condition that interrupts are enabled, it will cause the interrupt cycle to be initiated instead of normal instruction execution cycle. The pipelined SimP does not have to wait all that time to respond since there is an instruction in each pipeline stage and one instruction is completed after every machine cycle.

The solution adopted for the processor that is presented below is to respond as fast as possible to an interrupt request. Therefore, it responds after the instruction in execution stage has been completed. The address of the instruction in the decode stage is transferred from TEMP to stack, decode and execution stages are cleared and a new instruction fetch cycle is initiated by fetching the first instruction of interrupt service routine specified by INTVEC memory location.

The only exception to the above reaction is if the instruction in the execution stage happens to be a branch instruction. In that case control must be transferred to the instruction specified by branch instruction first. In this case the next consecutive address (consecutive to the destination address) is transferred from TEMP to the stack.

Interrupt cycle is executed in a separate branch of the control unit FSM and requires four machine cycles.

### Conditional Branch Instructions

If the instruction in execution stage is a conditional branch (Skip on Condition, SZ or SC), the original SimP will examine the condition flag *(*Z or C*)*. If the value of flag is low (**Z=0** or **C=0**), it will do nothing, i.e. the next instruction will be executed otherwise if the flag is high (**Z=1** or **C=1**) the next instruction will be skipped. The behavior of the pipelined processor is similar, except for the case when the next instruction is to be skipped. In that case the control unit clears the decode stage so there will be nothing to execute in the next machine cycle. However, it inserts a "No operation" (NOP) instruction that is included into the pipelined SimP instruction set.

## 15.3 Pipelined SimP Implementation

The pipelined SimP implementation presented in this chapter should be considered just as one possible core design that can be easily changed and modified. However, it still presents a fully functional design that easily fits in an FLEX10K20 device. The design is divided into two modules, data path and control unit, that are integrated on a higher design hierarchical level. In this chapter we present those two modules separately, and leave the integration to the reader as an exercise.

### *15.3.1 Data Path Design*

The data path design is given with in the following Verilog description presented as Example 15.1. Some explanations are provided as comments in the description itself.

Example 15.1 Pipelined SimP data path

```
module datapath (clk, reset, irq,
    dm_datain, pm_datain,
    clr_a, ld_a,
    clr_b, ld_b, inc_b, dec_b, com_b,
    lda_dmar, ldd_dmar,
    ld_pir,
    clr_ir, ld_ir,
    ld_pmar,
    clr_pc,
    lda_pc, ldd_pc, inc_pc,
    inc_sp, dec_sp, init_sp,
    inc_st, ld_st, dec_st, intvec_st, init_st,
    ld_temp,
    clr_c, clr_z, ld_c, ld_z,
    set_ien, clr_ien,
    set_iack,clr_iack,
    clr_irq,

    alu_select,
    dmdbus_sel,
    dmabus_sel,
    dmaddr_sel,
    wr_dm,

    z, c,
    dm_dataout, dm_addrout, pm_addrout,
    irbus,
    irqa,iena,lack
       );

input  clk, reset, irq;
input  [15:0]  dm_datain, pm_datain; // from data and program
       // memory

input  clr_a, ld_a; //accumulator a
input  clr_b, ld_b, inc_b, dec_b, com_b; //accumulator b
input  lda_dmar, ldd_dmar; // dmar
input  ld_pir; // pir prefetch instruction register
input  clr_ir, ld_ir; // instruction register
```

```
input ld_pmar; // pm address register
input clr_pc, lda_pc, ldd_pc, inc_pc; // pc program counter
input inc_sp, dec_sp, init_sp; // sp stack pointer
input inc_st, ld_st, dec_st, intvec_st, init_st; // st shadow
      // stack pointer
input ld_temp; // temp shadow program counter
input clr_c, clr_z, ld_c, ld_z; // flags control
input  set_ien, clr_ien,   // interrupt control
   set_iack,clr_iack,
   clr_irq;

input [1:0] alu_select; // selsect ALU operation
input [1:0] dmdbus_sel; // dbusmux
input [1:0] dmabus_sel; // abusmux
input [1:0] dmaddr_sel; // dm address
input wr_dm;


// outputs
output z, c;
output [15:8] irbus;
output [15:0] dm_dataout;
output [11:0] dm_addrout, pm_addrout;
output irqa,iena,iack;

reg z, c;
reg [15:8] irbus;
reg [15:0] dm_dataout;
reg [11:0] dm_addrout, pm_addrout;
reg irqa,iena,iack;

//internal signals
reg [15:0] dm_dbus, pm_dbus;
reg [11:0] dm_abus, pm_abus;

reg [16:0] alu_out; // alu output
reg hold_c, hold_z;

reg [15:0] hold_a; // acca output
reg [15:0] hold_b; // accb output
reg [11:0] hold_dmar; // dmar output
reg [15:0] hold_pir; // pir output
reg [15:0] hold_ir; // ir output
reg [11:0] hold_pmar; // pmar
reg [11:0] hold_pc; // pc
reg [11:0] hold_sp; // sp
reg [11:0] hold_st; //st
reg [11:0] hold_temp; // temp
```

```
// internal memories connections
reg [15:0] dm_data;
reg [15:0] intdm_dataout; // from dm
reg [15:0] intpm_dataout; // from pm

// select signal values
parameter pc2dmdb = 2'B00, temp2dmdb=2'B01, alu2dmdb=2'B10,
       dm2dmdb=2'B11; // dm_dbusmux
parameter pc2dmab = 2'B00, sp2dmab = 2'B01, st2dmab = 2'B10,
       ir2dmab = 2'B11; // dm_abus select lines
parameter alu_add = 2'B00, alu_and = 2'B01, alu_passa = 2'B10,
       alu_passb = 2'B11; // alu operations
parameter st2adbus = 2'B10, sp2adbus = 2'B01, dmar2adbus =
       2'B11; // memory mux select


// Altera specific modules
lpm_ram_dq dm (.q (intdm_dataout), .data (dm_dbus),
       .address (dm_addrout[7:0]),.we (wr_dm),
       .inclock (!clk));
    defparam dm.lpm_width = 16;
       defparam dm.lpm_widthad = 8;
   defparam dm.lpm_outdata = "UNREGISTERED";
    defparam dm.lpm_address_control = "REGISTERED";

lpm_rom pm (.q (intpm_dataout),
       .address(pm_addrout[7:0]),.inclock (!clk),
       .memenab (1'b1)); // read permanently enabled
       defparam pm.lpm_width = 16;
       defparam pm.lpm_widthad = 8;
   defparam pm.lpm_file = "pm.mif";
   defparam pm.lpm_outdata = "UNREGISTERED";
   defparam pm.lpm_address_control = "REGISTERED";

// accumulator a
always @(posedge clk or posedge reset)
       if (reset)
    hold_a = 0;
   else
   begin
    if (clr_a)
       hold_a - 0;
    elseif(ld_a)
       hold_a = dm_dbus;
   end


// accumulator b
always @(posedge clk or posedge reset)
```

```
    if (reset)
     hold_b=0;
    else
    begin
     if (ld_b)
       hold_b = dm_dbus;
     else if (clr_b)
       hold_b = 0;
     else if (inc_b)
       hold_b = hold_b + 1;
     else if (dec_b)
       hold_b = hold_b - 1;
     else if (com_b)
       hold_b= !hold_b;
     else
       hold_b= hold_b;
    end


// data memory address register
always @(posedge clk or posedge reset)
    if (reset)
     hold_dmar = 0;
    else if (lda_dmar)
       hold_dmar = dm_abus;
    else if (ldd_dmar)
       hold_dmar = hold_pir;


// prefetch instruction register
always @(posedge clk or posedge reset)
    if (reset)
     hold_pir = pm_datain;
    else if (ld_pir & (pm_addrout > 255))
     hold_pir = pm_datain;
    else if (ld_pir & (pm_addrout <= 255))
     hold_pir = intpm_dataout;


// instruction register
always @(posedge clk or posedge reset)
    if (reset)
     hold_ir = 0;
    else if (clr_ir)
     hold_ir = 16'H7000; // NOP instruction
    elseif(ld_ir)
     hold_ir = hold_pir;
```

```
// pm address register
always @(posedge clk or posedge reset)
    if (reset)
     hold_pmar=0;
    else if (ld_pmar)
     hold_pmar = hold_pc;

// pc program counter
always @(posedge clk or posedge reset)
    if (reset)
     hold_pc = 0;
    else if (clr_pc)
     hold_pc = 0;
    else if(ldd_pc)
     hold_pc = dm_dbus;
    else if(lda_pc)
     hold_pc = hold_dmar;
     //hold_pc = hold_ir[11:0];
    else if(inc_pc)
     hold_pc = hold_pc+1;


// sp stack pointer
always @(posedge clk or posedge reset)
    if (reset)
     hold_sp = 12'H0fe;
    else if(init_sp)
     hold_sp = 12'H0fe; // initial value
    else if(dec_sp)
     hold_sp = hold_sp-1;
    else if (inc_sp)
     hold_sp = hold_sp+1;


// st register
always @(posedge clk or posedge reset)
    if (reset)
     hold_st = 12'H0ff;
    else if (init_st)
     hold_st = 12'H0ff; // initial value
    else if (intvec_st)
     hold_st = 12'Hff;
    else if (dec_st)
     hold_st = hold_st-1;
    else if (inc_st)
     hold_st = hold_st+1;
    else if (ld_st)
     hold_st = hold_sp;
```

```
// temp
always @(posedge clk or posedge reset)
    if (reset)
     hold_temp = 0;
    else if (ld_temp)
     hold_temp = hold_pmar;


// alu provisional
always @(posedge clk or posedge reset)
    case (alu_select)
     alu_add: alu_out = hold_a + hold_b;
     alu_and: alu_out = hold_a & hold_b;
     alu_passa: alu_out = hold_a;
     alu_passb: alu_out = hold_b;
    default: alu_out = hold_a;
    endcase


// Flags
always @(posedge clk or posedge reset)
    begin
    if ((alu_out == 16'H000) & ld_z)
     hold_z = 1;
    else if (clr_z)
     hold_z = 0;
    if (alu_out[16] == 1 & ld_c)
     hold_c = 1;
    else if (clr_c)
     hold_c = 0;
    end

//connect flags to the output
always @(hold_c or hold_z)
    begin
     c = hold_c;
     z = hold_z;
    end

// outputs from data path
always @(dm_dbus)
    dm_dataout = dm_dbus;

always @(hold_pmar)
    pm_addrout= hold_pmar;

always @(hold_ir)
    irbus = hold_ir[15:8];
```

```
// dm_dbusmux
always @(dmdbus_sel or hold_pc or hold_temp or alu_out or
       dm_data)
    case (dmdbus_sel)
     pc2dmdb: begin dm_dbus[11:0] = hold_pc;
           dm_dbus[15:12] = 4'H0;
        end
     temp2dmdb: begin dm_dbus[11:0] = hold_temp;
           dm_dbus[15:12] = 4'H0;
        end
     alu2dmdb:       dm_dbus = alu_out;
     dm2dmdb: dm_dbus = dm_data;
    default: dm_dbus = dm_data;
    endcase

//data memory select address decoder
always @(dm_addrout or dm_datain or intdm_dataout)
    if (dm_addrout > 255)
     dm_data = dm_datain;
    else if (dm_addrout <= 255)
     dm_data = intdm_dataout;


// dm_abusmux
always @(dmabus_sel or hold_sp or hold_pc or hold_ir or
       hold_st)
    case (dmabus_sel)
     sp2dmab: dm_abus = hold_sp;
     pc2dmdb: dm_abus = hold_pc;
     ir2dmab: dm_abus = hold_ir[11:0];
     st2dmab: dm_abus = hold_st;
    default: dm_abus = hold_ir[11:0];
    endcase

// dm address selector
always @(dmaddr_sel or hold_sp or hold_dmar or hold_st)
    case (dmaddr_sel)
     sp2adbus: dm_addrout = hold_sp;
     dmar2adbus: dm_addrout = hold_dmar;
     st2adbus: dm_addrout = hold_st;
    default: dm_addrout = hold_dmar;
    endcase


// interrupt circuitry
always @(posedge clk)
    begin
     if (set_ien)
        iena = 1;
```

```
    else if (clr_ien)
       iena = 0;

    if (set_iack)
       iack = 1;
    else if (clr_iack)
       iack = 0;

    if (irq)
       irqa = 1;
    else if (clr_irq)
       irqa = 0;

   end

endmodule
```

### *15.3.2 Control Unit Design*

Control unit design implements the FSM as it is described by the flowchart in Figure 15.5. The detailed behavior is presented by the design presented by Verilog description in Example 15.2.

Example 15.2 Pipelined SimP control unit

```
module controlunit (
       //outputs
   clr_a, ld_a,
   clr_b, ld_b, inc_b, dec_b, com_b,
   lda_dmar, ldd_dmar,
   ld_pir,
   clr_ir, ld_ir,
   ld_pmar,
   clr_pc,
   lda_pc, ldd_pc, inc_pc,
   inc_sp, dec_sp, init_sp,
   inc_st, ld_st, dec_st, intvec_st, init_st,
   ld_temp,

   clr_c, clr_z, ld_c, ld_z,
   set_ien, clr_ien,
   set_iack,clr_iack,
   clr_irq,
```

```
    alu_select,
    dmdbus_sel,
    dmabus_sel,
    dmaddr_sel,
    rd_dm, wr_dm, rd_pm,

    clk, reset,
    z,c,
    irbus,
    irqa, iena
    );

output clr_a, ld_a,
    clr_b, ld_b, inc_b, dec_b, com_b,
    lda_dmar, ldd_dmar,
    ld_pir,
    clr_ir, ld_ir,
    ld_pmar,
    clr_pc,
    lda_pc, ldd_pc, inc_pc,
    inc_sp, dec_sp, init_sp,
    inc_st, ld_st, dec_st, intvec_st, init_st,
    ld_temp,
    clr_c, clr_z, ld_c, ld_z;


output set_ien, clr_ien,
    set_iack,clr_iack,
    clr_irq;

output [1:0] alu_select;
output [1:0] dmdbus_sel;
output [1:0] dmabus_sel;
output [1:0] dmaddr_sel;
output rd_dm, wr_dm, rd_pm;

input clk, reset;
input z,c;
input [15:8] irbus;
input irqa, iena;

reg clr_a, ld_a,
    clr_b, ld_b, inc_b, dec_b, com_b,
    lda_dmar, ldd_dmar,
    ld_pir,
    clr_ir, ld_ir,
    ld_pmar,
    clr_pc,
    lda_pc, ldd_pc, inc_pc,
```

```
    inc_sp, dec_sp, init_sp,
    inc_st, ld_st, dec_st, intvec_st, init_st,
    ld_temp,
    clr_c, clr_z, ld_c, ld_z;


reg set_ien, clr_ien,
    set_iack,clr_iack,
    clr_irq;


reg [1:0] alu_select;
reg [1:0] dmdbus_sel;
reg [1:0] dmabus_sel;
reg [1:0] dmaddr_sel;
reg rd_dm, wr_dm, rd_pm;


// instruction opcodes
parameter lda = 8'h0x, ldb = 8'h1x, sta = 8'h2x, stb = 8'h3x,
       jmp = 8'h4x, jsr = 8'h8x, psha = 8'hA0, pula = 8'hC0,
       ret = 8'hE0;

parameter add = 8'h71, a_and_b = 8'h72, cla = 8'h73, clb =
       8'h74, cmb = 8'h75, incb = 8'h76, decb = 8'h77,
    clc = 8'h78, clz = 8'h79, ion = 8'h7A, iof = 8'h7B,
       sc = 8'h7C, sz = 8'h7D, nop = 8'h70;


// select signal values
parameter pc2dmdb = 2'B00, temp2dmdb=2'B01, alu2dmdb=2'B10,
       dm2dmdb=2'Bll; // dm_dbusmux
parameter pc2dmab = 2'B00, sp2dmab = 2'B01, dmar2dmab = 2'B10,
       ir2dmab = 2'B11; // dm_abus select lines
parameter alu_add = 2'B00, alu_and = 2'B01, alu_passa = 2'B10,
       alu_passb = 2'B11; // alu operations
parameter st2adbus = 2'B10, sp2adbus = 2'B01,
       dmar2adbus = 2'B11; // memory mux select


// control unit one-hot encoded states
parameter s_reset0 =             11'b000_0000_0000,
     s_reset1 =             11'b100_0000_0001,
     s_plinit0 =            11'b100_0000_0010,
     s_plinit1 =            11'b100_0000_0100,
     s_plinit2 =            11'b100_0000_1000,
     s_plinit3 =            11'b100_0001_0000,
     s_pipeline =           11'b100_0010_0000,
     s_interrupt0 =  11'b100_0100_0000,
```

```
    s_interrupt1 = 11'b100_1000_0000,
    s_interrupt2 = 11'b101_0000_0000,
    s_interrupt3 = 11'b110_0000_0000;

reg [10:0] state; // state variable register

always @(posedge clk or posedge reset)

if (reset) // asynchronous reset
   state = s_reset0;
else
      case (state)

   s_reset0: // after transition on reset line detected
   begin
    init_sp = 1;
    init_st = 1;
    clr_pc = 1;
    set_ien = 1;
    state = s_reset1;
   end

   s_reset1: // deactivate initialization signals
   begin
    state = s_plinit0;
    clr_pc = 0;
    init_sp = 0;
    init_st = 0;
    set_ien = 0;
   end

   s_plinit0: // start pipeline initialization
   begin
    ld_pmar=1;      // load pmar from pc
    state = s_plinit1;
    dec_sp = 0;
    dec_st = 0;
    inc_sp = 0;
    inc_st = 0;
    wr_dm=0;
    rd_dm =0;
    lda_pc = 0;
    ldd_pc = 0;
   end

   s_plinit1: // continue pipeline initialization
   begin
    ld_pir = 1;
    rd_pm = 1;
```

```
    inc_pc = 1; // pc to next instruction
    state = s_plinit2;
    ld_pmar=0;
  end

  s_plinit2: // continue pipeline initialization
  begin
    ld_ir = 1;
    dmabus_sel = ir2dmab;
    lda_dmar = 1; // dmar <-- ir
    ld_pmar = 1;
    inc_pc = 1;
    state = s_plinit3;
    ld_pir = 0;
    rd_pm = 0;

  end

  s_plinit3: // finish pipeline initialization
  begin
    dmabus_sel = ir2dmab; // dmar from ir12
    lda_dmar = 1;   // dmar <-- ir
    ld_temp = 1;    // temp<-pmar
    ld_pir = 1; // prefetch next instruction
    rd_pm = 1;
    ld_pmar = 1;
    inc_pc = 1;
    state = s_pipeline; // transfer to pipeline mode
    ld_ir = 0;
  end

  s_interrupt0: // start interrupt cycle
  begin
    dmdbus_sel = temp2dmdb;      // from temp
    dmaddr_sel = sp2adbus; // from sp
    wr_dm = 1;      // pmem[sp]<--temp
    rd_dm = 0;
    intvec_st = 1;  // st<-- INTVEC
    clr_irq = 1;
    state = s_interrupt1;
  end

  s_interrupt1: // continue interrupt cycle
  begin
    set_iack = 1;
    dec_sp = 1;
    dmaddr_sel = st2adbus; // from st
    dmdbus_sel = dm2dmdb; // from mem %
    rd_dm = 1;
```

```
 ldd_pc = 1;
 state = s_interrupt2;
 wr_dm = 0;      // pmem[sp] <--temp
 rd_dm = 0;
 intvec_st = 0 ;  // st<-- INTVEC
 clr_irq = 0;
end

s_interrupt2: // continue interrupt cycle
begin
 ld_st = 1;
 state = s_interrupt3;
 set_iack = 0;
 dec_sp = 0;
 rd_dm = 0;
 ldd_pc = 0;
end

s_interrupt3:
begin
 inc_st = 1;
 clr_iack = 1;
 state = s_plinit0; // pipeline initialize
 ld_st = 0;
end

s_pipeline:
begin
 // initialize control signals
 clr_a =0;
 ld_a =0;
 clr_b =0;
 ld_b =0;
 inc_b =0;
 dec_b =0;
 com_b =0;
 lda_dmar =0;
 ldd_dmar =0;
 ld_pir =0;
 clr_ir =0;
 ld_ir =0;
 ld_pmar =0;
 clr_pc =0;
 lda_pc =0;
 ldd_pc =0;
 inc_pc =0;
 inc_sp =0;
 dec_sp =0;
 init_sp =0;
```

```
inc_st =0;
ld_st =0;
dec_st =0;
intvec_st =0;
init_st =0;
ld_temp =0;
clr_c =0;
clr_z =0;
ld_c =0;
ld_z =0;
rd_dm =0;
wr_dm =0;
rd_pm =0;
clr_ien =0;

// instruction decoding and execution
case (irbus[15:8])
jmp:
begin
   lda_pc = 1;
   state = s_plinit0;
end

jsr:
begin
   dmdbus_sel = temp2dmdb;
   dmaddr_sel = sp2adbus;
   wr_dm = 1;
   rd_dm= 0;
   //dmabus_sel = ir2dmab;
   lda_pc = 1;
   dec_sp = 1;
   dec_st = 1;
   state = s_plinit0;
end

ret:
begin
   dmaddr_sel= st2adbus;
   dmdbus_sel = dm2dmdb;
   rd_dm = 1;
   ldd_pc = 1; // pc<-M[st]
   inc_sp = 1;
   inc_st = 1;
   state = s_plinit0;
end

lda:
begin
```

```
      ld_a = 1;
      rd_dm = 1;
      dmaddr_sel = dmar2adbus;
      dmdbus_sel = dm2dmdb;
      if (iena & irqa)
        begin
          clr_ien = 1;
          state = s_interrupt0;
        end
      else
        begin
          //  update pipeline

          lda_dmar = 1 ; // dmar<-pir
          ld_temp = 1; //temp<-pmar
                                  ld_ir = 1;
          ld_pir = 1;
          rd_pm = 1;
          ld_pmar = 1;
          inc_pc = 1;
          state = s_pipeline;
        end
   end

   ldb:
   begin
      ld_b = 1;
      rd_dm =       1;
      dmaddr_sel = dmar2adbus;
      dmdbus_sel = dm2dmdb
  if (iena & irqa)
        begin
          clr_ien = 1;
          state = s_interrupt0;
        end
      else
        begin
          //  update pipeline

          lda_dmar = 1 ; // dmar<-pir
          ld_temp = 1; //temp<-pmar
                                  ld_ir = 1;
          ld_pir = 1;
          rd_pm = 1;
          ld_pmar = 1;
          inc_pc = 1;
          state = s_pipeline;
        end
   end
```

```
sta:
begin
   alu_select = alu_passa;
   dmdbus_sel = alu2dmdb;
   dmaddr_sel = dmar2adbus;
   wr_dm = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       //  update pipeline

       lda_dmar = 1; // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
       ld_pir = 1;
       rd_pm = 1;
       ld_pmar = 1;
       inc_pc = 1;
       state = s_pipeline;
     end
end


stb:
begin
   alu_select = alu_passb;
   dmdbus_sel = alu2dmdb;
   dmaddr_sel = dmar2adbus;
   wr_dm = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       //  update pipeline

       lda_dmar =1 ;  // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
       ld_pir = 1;
       rd_pm = 1;
       ld_pmar = 1;
       inc_pc = 1;
```

```
        state = s_pipeline;
      end
end

psha:
begin
   dmaddr_sel = sp2adbus;
   wr_dm = 1;
   dmdbus_sel = alu2dmdb;
   alu_select = alu_passa; // M[dmar]<-a
   dec_sp = 1;
   dec_st = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       //  update pipeline

       lda_dmar = 1; // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
       ld_pir = 1;
       rd_pm = 1;
       ld_pmar = 1;
       inc_pc = 1;
       state = s_pipeline;
     end
end

pula:
begin
   ld_a = 1;
   dmaddr_sel = st2adbus;
   dmdbus_sel = dm2dmdb;
   rd_dm = 1;  // a<-M[st]
   inc_sp = 1;
   inc_st = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       //  update pipeline
```

```
        lda_dmar = 1; // dmar<-pir
        ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
        ld_pir = 1;
        rd_pm = 1;
        ld_pmar = 1;
        inc_pc = 1;
        state = s_pipeline;
      end
end

add:
begin
   ld_a = 1;
   dmdbus_sel = alu2dmdb;
   alu_select = alu_add;
   ld_c = 1;
   ld_z = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
    else
     begin
       //  update pipeline

       lda_dmar = 1; // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
       ld_pir = 1;
       rd_pm = 1;
       ld_pmar = 1;
       inc_pc = 1;
       state = s_pipeline;
     end
end

a_and_b:
begin
   ld_a = 1;
   dmdbus_sel = alu2dmdb;
   alu_select = alu_and;
   ld_z=1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
```

```
    else
      begin
        //  update pipeline

        lda_dmar = 1; // dmar<-pir
        ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
        ld_pir = 1;
        rd_pm = 1;
        ld_pmar = 1;
        inc_pc = 1;
        state = s_pipeline;
      end
end


cla:
begin
   clr_a = 1;
   if (iena & irqa)
      begin
        clr_ien = 1;
        state = s_interrupt0;
      end
   else
      begin
        //  update pipeline

        lda_dmar =1 ; // dmar<-pir
        ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
        ld_pir = 1;
        rd_pm = 1;
        ld_pmar = 1;
        inc_pc = 1;
        state = s_pipeline;
      end
end

clb:
begin
   clr_b = 1;
   if (iena & irqa)
      begin
        clr_ien = 1;
        state = s_interrupt0;
      end
   else
      begin
```

```
        //  update pipeline

        lda_dmar = 1; // dmar<-pir
        ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
        ld_pir = 1;
        rd_pm = 1;
        ld_pmar = 1;
        inc_pc = 1;
        state = s_pipeline;
      end
end

cmb:
begin
   com_b = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       //  update pipeline

       lda_dmar = 1; // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
       ld_pir = 1;
       rd_pm = 1;
       ld_pmar = 1;
       inc_pc = 1;
       state = s_pipeline;
     end
end

incb:
begin
   inc_b = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       lda_dmar = 1; // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
```

```
        ld_pir = 1;
        rd_pm = 1;
        ld_pmar = 1;
        inc_pc = 1;
        state = s_pipeline;
      end
end

decb:
begin
   dec_b = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       lda_dmar = 1; // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
       ld_pir = 1;
       rd_pm = 1;
       ld_pmar = 1;
       inc_pc = 1;
       state = s_pipeline;
     end
end

clc:
begin
   clr_c = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       lda_dmar = 1; // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
       ld_pir = 1;
       rd_pm = 1;
       ld_pmar = 1;
       inc_pc = 1;
       state = s_pipeline;
     end
end
```

```
clz:
begin
   clr_z = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       lda_dmar = 1; // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
       ld_pir = 1;
       rd_pm = 1;
       ld_pmar = 1;
       inc_pc = 1;
       state = s_pipeline;
     end
end

ion:
begin
   set_ien = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       lda_dmar = 1; // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
       ld_pir = 1;
       rd_pm = 1;
       ld_pmar = 1;
       inc_pc = 1;
       state = s_pipeline;
     end
end

iof:
begin
   clr_ien = 1;
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
```

```
      end
    else
      begin
        lda_dmar = 1 ; // dmar<-pir
        ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
        ld_pir = 1;
        rd_pm = 1;
        ld_pmar = 1;
        inc_pc = 1;
        state = s_pipeline;
      end
end

nop:
begin
   if (iena & irqa)
     begin
       clr_ien = 1;
       state = s_interrupt0;
     end
   else
     begin
       lda_dmar = 1; // dmar<-pir
       ld_temp = 1; //temp<-pmar
                                ld_ir = 1;
       ld_pir = 1;
       rd_pm = 1;
       ld_pmar = 1;
       inc_pc = 1;
       state = s_pipeline;
     end
end

sc:
begin
   if (c == 1)
     begin
       ld_pir = 1;
       rd_pm = 1;
       inc_pc = 1;
       ld_temp = 1;
       clr_ir = 1;//load NOP into ir
       if (iena & irqa)
          begin
          clr_ien = 1;
          state = s_interrupt0;
          end
       else
```

```
            begin
            //  update pipeline
 // dmar<-pir
            lda_dmar = 1;
            ld_pmar = 1;
            state = s_pipeline;
            end
        end
      else if (c == 0)
        begin
         if (iena & irqa)
            begin
            clr_ien = 1;
            state = s_interrupt0;
            end
         else
            begin
            //  update pipeline
 // dmar<-pir
            lda_dmar = 1;
 ld_temp = 1;
ld_ir = 1;
            ld_pir = 1;
            rd_pm = 1;
            ld_pmar = 1;
            inc_pc = 1;
            state = s_pipeline;
            end
        end
 end // sc

 sz:
 begin
    if (z == 1)
      begin
       ld_pir = 1;
       rd_pm = 1;
       inc_pc = 1;
       ld_temp = 1;
       clr_ir = 1;//load NOP into ir
       if (iena & irqa)
          begin
          clr_ien = 1;
          state = s_interrupt0;
          end
       else
          begin
          //  update pipeline
 // dmar<-pir
```

```
            lda_dmar = 1;
    ld_pmar = 1;
            state = s_pipeline;
            end
        end
      else if (z==0)
        begin
          if (iena & irqa)
            begin
            clr_ien = 1;
            state = s_interrupt0;
            end
          else
            begin
            //  update pipeline
    // dmar<-pir
            lda_dmar = 1;
    ld_temp = 1;
   ld_ir = 1;
            ld_pir = 1;
            rd_pm = 1;
            ld_pmar = 1;
            inc_pc = 1;
            state = s_pipeline;
            end
        end
     end // sz
   endcase //pipeline mode

   end //s_pipeline

endcase // control unit states

endmodule
```

## 15.4 Questions and Problems

15.1 Extend the pipelined SimP ALU with the following additional operations:

- Subtraction represented by **A ← A – B**
- Logical OR operation represented by **A ← A** or B (bit-wise OR)
- Logical XOR operation represented by **A ← A** xor B (bit-wise XOR)

15.2 Extend the pipelined SimP instruction set with instructions for arithmetic and logical shift for 1 bit left and right of the content of register A. Use carry bit C to receive a bit that is transferred out of A register.

15.3Complete pipelined SimP's design by connecting the data path and control unit shown in this chapter and carry out simulation using Max+Plus II simulator. Simulation should be extensive and show execution of all SimP instructions. For that purpose write a small program and store it into program memory. For data storage use data memory.

15.4 Modify pipelined SimP by introducing external memories to store additional programs and data. Each of these memories should have 1K 16-bit locations. What are the limitations of the pipelined SimP in terms of the type of program memory?

15.5 Assume that the pipelined SimP internal program memory is always treated as ROM that can be modified at device configuration time. What modifications to the processor architecture are needed to enable external program memory to be read-write memory to which programs can be downloaded using program already loaded into internal program memory. Program stored into internal memory should perform a function of the program loader.

15.6 Assume that pipelined SimP can change its programs by reconfiguring contents of the internal program memory. The new contents is stored in external memory device, e.g. ROM. Study what circuitry should be added to the pipelined SimP to enable change of the programs as requested by computation being carried out from the internal program memory.

15.7 Analyze solutions for the problems 7.6 – 7.12 applied to the pipelined SimP.

15.8 Using Verilog implement a serial asynchronous receiver/transmitter (SART) from Chapter 12. Add all necessary registers to enable SART's connection with pipelined SimP. Add SART to the SimP and make a full computer that can communicate with the external world using SART.

15.9Analyze additions to the pipelined SimP from problem 15.8 extended with external read/write program memory from problem 15.4 to enable downloading of new programs into external memory from another source connected to SimP using SART.

# GLOSSARY

**Access Type** A data type analogous to a pointer that provides a form of indirection.

**Active-high (-low) node** A node that is activated when it is assigned a value one (zero) or Vcc (Gnd). In AHDL design files, an active-low node should be assigned a default value of Vcc with the Defaults statement.

**Aggregate** A form of expression used to denote the value of a composite type. An aggregate value is specified by listing the value of each element of the aggregate using either positional or named notation.

**AHDL** Acronym for Altera Hardware Description Language. Design entry language which supports Boolean equation, state machine, conditional, and decode logic. It also provides access to all Altera and user-defined macrofunctions.

**Alias** Statement used to declare an alternate name for an object.

**Always block** A basic concurrent statement in Verilog represented by a collection of procedural statements that are executed whenever there is an event on any signal that appears in the sensitivity list.

**Antifuse** Any of the programmable interconnect technologies forming electrical connection between two circuit points rather than making open connections.

**Architecture** Describes the behaviour, dataflow, and/or structure of a VHDL entity. An architecture is created with an architecture body. A single entity can have more than one architecture. Configuration declarations are used to specify which architectures to use for each entity.

**Array** A collection of one or more elements of the same type that are accessed using one or more indices depending on dimension of array. Array data types are declared with an array range and array element type.

**ASIC** Acronym for Application-Specific Integrated Circuit. A circuit whose only final photographic mask process is design dependent.

**Assert** A statement that checks whether a specified condition is true. If the condition is not true, a report is generated during simulation.

**Assignment** In VHDL, assignment refers to the transfer of a value to a symbolic name or group, usually through a Boolean equation. The value on the right side of an assignment statement is assigned to the symbolic name or group on the left.

**Asynchronous input** An input signal that is not synchronized to the device Clock.

**Attribute** A special identifier used to return or specify information about a named entity. Predefined attributes are prefixed with ’character.

**Back annotation** Process of incorporating time delay values into a design netlist reflecting the interconnect capacitance obtained from a completed design. Also, in Altera’s case, the process of copying device and resource assignments made by the Compiler into Assignment and Configuration File for a project. This process preserves the current fit in future compilations.

**Block** A feature that allows partitioning of the design description within an architecture in VHDL.

**Block statements** Used in Verilog to group two or more statements together to act as a single statement. Synthesizable statements are delimited by **begin** and **end** keywords.

**Cell** A logic function. It may be a gate, a flip-flop, or some other structure. Usually, a cell is small compared to other circuit building blocks.

**Cell library** The collective name for a set of logic functions defined by the manufacturer of an FPLD or ASIC. Simulation and synthesis tools use cell library when simulating and synthesizing a model.

**CLB** Acronym for Configurable Logic Block. This element is the basic building block of the Xilinx LCA product family.

**Clock** A signal that triggers registers. In a flip-flop or state machine, the clock is an edge-sensitive signal. The output of the clock can change only on the clock edge.

**Clock enable** The level-sensitive signal on a flip-flop with E suffix, e.g., DFFE. When the Clock enable is low, clock transitions on the clock input of the flip-flop are ignored.

**Component** Specifies the ports of a primitive or macrofunction in VHDL. A component consists of the name of the primitive or macrofunction, and a list of its inputs and outputs. Components are specified in the Component declaration

**Component instantiation** A concurrent statement that references a declared component and creates one unique instance of that component.

**Composite type** A data type that includes more than one constituent element (for instance, array or record).

**Concurrent statements** Statements that are executed in parallel.

**Concurrent statements** Statements that are executed in parallel and their textual order within the model has no effect on the modeled behavior. Concurrent statements are basic element of all hardware description languages.

**Configuration** It maps instances of VHDL components to design entities and describes how design entities are combined to form a complete design. Configuration declarations are used to specify which architectures to use for each entity.

**Configuration scheme** The method used to load configuration data into an FPGA.

**Constant** An object that has a constant value and cannot be changed.

**Control path** The path of signals generated by control unit used to control data path.

**CPLD** Acronym for Complex Programmable Logic Device. CPLDs include an array of functionally complete or universal logic cells in an interconnection framework that has foldback connection to central programming regions.

**Data path** The path which provides processing and transfer of information in the circuit through the blocks of combinational and sequential logic.

**Design entity** The combination of an entity and its corresponding architecture.

**Design file** A file that contains description of the logic for a project and is compiled by the Compiler.

**Design library** Stores VHDL units that have already been compiled. These units can be referenced in VHDL designs. Design libraries can contain one or more of the following units:

- Entity declarations
- Architecture declarations
- Configuration declarations
- Package declarations
- Package body declarations

**Design unit** A section of VHDL description that can be compiled separately. Each design unit must have a unique name within the project.

**Driver** Contains the projected output waveform for a data object. Each scheduled value is a driver.

**Dual-purpose pins** Pins used to configure an FPGA device that can be used as I/O pins after initialization.

**Dynamic reconfigurability** Capability of an FPLD to change its function "on-the-fly" without interruption of system operation.

**EDIF** Acronym for Electronic Design Interchange Format. An industry-standard format for the transmission of design files.

**Entity** See Design entity.

**Enumeration type** A symbolic data type that is declared with an enumerated type name, and one or more enumeration values.

**EPLD** Acronym for EPROM Programmable Logic Devices. This is a PLD that uses EPROM cells to internally configure the logic function. Also, Erasable Programmable Logic Device.

**Event** The change of value of a signal. Usually refers to simulation.

**Event scheduling** The process of scheduling of signal values to occur at some simulated time.

**Excitation function** Boolean function that specifies logic that directs state transitions in a state machine.

**Exit condition** An expression that specifies a condition under which a loop should be terminated.

**Expander** Section in the MAX LAB containing an array of foldback NAND functions. The expander is used to increase the logical inputs to the LAB macrocell section or to make other logic and storage functions in the LAB.

**Fan-in** The number of input signals that feed all the input equations of a logic cell.

**Fan-out** The number of output signals that can be driven by the output of a logic cell.

**FastTrack interconnect** Dedicated connection paths that span the entire width and height of a FLEX 8000 device. These connection paths allow the signals to travel between all LABs in a device.

**Field name** An identifier that provides access to one element of a record data type.

**File type** A data type used to represent an arbitrary-length sequence of values of a given type.

**For loop** A loop construct in which an iteration scheme is a for statement.

**Finite state machine** The model of a sequential circuit that cycles through a predefined sequence of states.

**Fitting** Process of making a design fit into a specific architecture. Fitting involves technology mapping, placement, optimization, and partitioning among other operations.

**Flip-flop** An edge-sensitive memory device (cell) that stores a single bit of data.

**Floorplan** Physical arrangement of functions within a design relative to the other.

**FPGA** Acronym for Field Programmable Gate Array. A regular array of cells that is either functionally complete or universal within a connection framework of signal routing channels.

**FPLD** An integrated circuit used for implementing digital hardware that allows the end user to configure the chip to realize different designs. Configuring such a device is done using either a special programming unit or by doing it "in system".

**Function prototype** Specifies the ports of a primitive or macrofunction in AHDL. It consists of the name of the primitive or macrofunction, and a list of its inputs and outputs in exact order in which they are used. An instance of the primitive or macrofunction can be inserted with an Instance declaration or an in-line reference.

**Function** A subprogram common for both VHDL and Verilog used to model combinational logic. Function must have at least one input and returns a single value.

**Functional simulation** A simulation mode that allows to simulate the logical performance of a project without timing information.

**Functional test vector** The input stimulus used during simulation to verify a VHDL model operates functionally as intended.

**Functionally complete** Property of some Boolean logic functions permitting them to make any logic function by using only that function. The properties include making the AND function with an invert or the OR function with an invert.

**Fuse** A metallic interconnect point that can be electrically changed from short circuit to an open circuit by applying electrical current.

**Gate** An electronic structure, built from transistors, that performs a function.

**Gate array** Array of transistors interconnected to form gates. The gates in turn are configured to form larger functions.

**Gated clock** A clock configuration in which the output of an AND or OR gate drives a clock.

**Generic** A parameter passed to an entity, component or block that describes additional, instance-specific information about that entity, component or block.

**Glitch or spike** A signal value pulse that occurs when a logic level changes two or more times over a short period.

**Global signal** A signal from a dedicated input pin that does not pass through the logic array before performing its specified function. Clock, Preset, Clear, and Output Enable signals can be global signals.

**GND** A low-level input voltage. It is the default inactive node value.

**Hierarchy** The structure of a design description, expressed as a tree of related components.

**Identifier** A sequence of characters that uniquely identify a named entity in a design description.

**Index** A scalar value that specifies an element or range of elements within an array.

**Input vectors** Time-ordered binary numbers representing input values sequences to a simulation program.

**Instance** The use of a primitive or macrofunction in a design file.

**I/O cell register** A register on the periphery of a FLEX 8000 device or a fast input-type logic cell that is associated with an I/O pin.

**I/O feedback** Feedback from the output pin on an Altera device that allows an output pin to be also used as an input pin.

**LAB** Acronym for Logic Array Block. The LAB is the basic building block of the Altera MAX family. Each LAB contains at least one macrocell and an I/O block and an expander product term array.

**Latch** A level-sensitive clocked memory device (cell) that stores a single bit of data. A high-to-low transition on the Latch Enable signal fixes the contents of the latch at the value of the data input until the next low-to-high transition on Latch Enable.

**Latch enable** A level-sensitive signal that controls a latch. When it is high, the input flows through the output; when it is low, the output holds its last value.

**Library** In VHDL denotes facility to store analyzed design units.

**Literal** A value that can be applied to an object of some type.

**Logic element** A basic building block of an Altera FLEX 8000 device. It consists of a look-up table i.e., a function generator that quickly computes any function of four variables, and a programmable flip-flop to support sequential functions.

**Long line** Mechanism inside an LCA where a signal is passed through repeating amplifier to drive a larger interconnect line. Long lines are less sensitive to metal delays.

**LPM** Acronym for Library of Parametrized Modules. Denotes the library of design units that contain one or more changeable parts, parameters, that are used to customize design unit as application requires.

**Macro** When used with FPGAs, a cell configuration that can be repeated as needed. It can be Hard and Soft macro.

**Macrocell** In FPGAs, a portion of the FPGA that is smallest indivisible building block. In MAX devices it consists of two parts: combinatorial logic and a configurable register.

**MAX** Acronym for Multiple Array MatriX, which is an Altera product family. It is usually considered to be a CPLD.

**MAX+PLUS II** Acronym for Multiple Array Matrix Programmable Logic User System II. A set of tools that allow design and implementation of custom logic circuits with Altera's MAX and FLEX devices.

**Memory declaration** Used in Verilog to describe groups of registers or variables. It is used to model memories (RAM, ROM) or arrays of registers.

**Mode** A direction of signal (either in, out, inout or buffer) used as subprogram parameter or port.

**Model** A representation that behaves similarly to the operation of some digital circuit.

**Module** Basic Verilog design unit that encapsulates a design including input and output ports. It can be reused in subsequent designs as an entity at the lower hierarchical level.

**MPLD** Acronym for Mask-Programmed Logic Device.

**Net** Data type used in Verilog to represent the physical connection of hardware elements in a structural type of architecture.

**Netlist** A text file that describes a design. Minimal requirements are identification of function elements, inputs and outputs and connections.

**Netlist synthesis** Process of deriving a netlist from an abstract representation, usually from a hardware description language.

**NRE** Acronym for Non-Recurring Engineering expense. It reefers to one-time charge covering the use of design facilities, masks and overhead for test development.

**Object** A named entity of a specific type that can be assigned a value. Object in VHDL include signals, constants, variables and files.

**One Hot Encoding** A design technique used more with FPGAs than CPLDs. It assigns a single flip-flop to hold a logical one representing a state, with the rest of flip-flops being held at zeros.

**Package** A collection of commonly used VHDL constructs that can be shared by more than one design unit.

**PAL** (Programmable Array Logic) a relatively small FPLD containing a programmable AND plane followed by a fixed-OR plane.

**Parameter** An object or literal passed into a subprogram via that subprogram's parameter list.

**Parameter declaration** Used in Verilog to describe a constant.

**Partitioning** Setting boundaries within functions of a system.

**Physical types** A data type used to represents measurements.

**PLA** (Programmable Logic Array) a relatively small FPLD that contains two levels of programmable logic - an AND plane and an OR plane.

**Placement** Physical assignment of a logical function to a specific location within an FPGA. Once logic function is placed, its interconnection is made by routing.

**PLD** Acronym for Programmable Logic Device. This class of devices comprise PALs, PLAs, FPGAs and CPLDs.

**Port** A symbolic name that represents an input or output of a primitive or of a macrofunction design file.

**Primitive** One of the basic functional blocks used to design circuits with Max+Plus II software. Primitives include buffers, flip-flops, latch, logical operators, ports, etc. Functional prototypes for AHDL primitives are built into the Max+Plus II software. Component declarations for VHDL primitives are provided in the maxplus2 package.

**Process** A basic concurrent statement in VHDL represented by a collection of sequential statements that are executed whenever there is an event on any signal that appears in the process sensitivity list, or whenever an event occurs that satisfies condition of a wait statement within the process.

**Programmable switch** A user programmable switch that can connect a logic element or input /output element to an interconnect wire or one interconnect wire to another.

**Project** A project consists of all files that are associated with a particular design, including all subdesign files and ancillary files created by the user or by Max+Plus II software. The project name is the same as the name of the top-level design file without extension.

**Propagation delay** The time required for any signal transition to travel between pins and/or nodes in a device.

**Range** A subset of the possible values of a scalar type.

**Record** A composite data type that includes more than one of differing types. Record elements are identified by field names.

**Register** A memory device that contains more than one latch or flip-flop that are clocked from the same source clock signal.

**Register** (**reg**) Data type in Verilog used for the declaration of objects that preserve their value over simulation cycles. The objects of register type are assigned values using blocking and non-blocking procedural assignments.

**Resource** A resource is a portion of a device that performs a specific, user-defined task (e.g., pins, logic cells).

**Retargetting** A process of translating a design from one FPGA or other technology to another. Retargetting involves technology mapping and optimization.

**Routing** Process of interconnecting previously placed logic functions.

**RTL** Acronym for Register Transfer Level. The model of circuit described in VHDL that infers memory devices to store results of processing or data transfers. Sometimes it is referred to as dataflow-style model.

**Scalar** A data type that has a distinct order of its values, allowing two objects or literals of that type to be compared using relational operators.

**Semicustom** General category of integrated circuits that can be configured directly by the user of IC. It includes gate array, PLD, FPGA, PROM and EPROM devices.

**Signal** In VHDL a data object that has a current value and scheduled future values at simulation times. In RTL models signals denote direct hardware connections.

**Simulation** Process of modeling a logical design and its stimuli in which the simulator calculates output signal models.

**Slew rate** Time rate of change of voltage. Some FPGAs permit a fast or slow slew rate to be programmed for an output pin.

**Slice** A one-dimensional, contiguous array created as a result of constraining a larger one-dimensional array.

**Speed performance** The maximum speed of a circuit implemented in an FPLD. It is set by the longest delay through any path for combinational circuits, and by maximum clock frequency at which the circuit operates properly for sequential circuits.

**State transition diagram** A graphical representation of the operation of a finite state machine using directed graphs.

**Structural-type architecture** The level at which VHDL describes a circuit as an arrangement of interconnected components.

**Subprogram** A function or procedure. It can be declared globally or locally.
**Synthesis** The process of converting the model of a design described in VHDL from one level of abstraction to another, lower and more detailed level.

**Technology mapping** Process of translating the function of a design from one technology to another. All versions of the design would have the same function, but the cell used would be very different.

**Test bench** A VHDL model used to verify the correct behavior of another VHDL model, commonly known as unit under test.

**Type** A declared name and its corresponding set of declared values representing the possible values the type. Four general categories of types are used: scalar types, composite types, file types and access types.

**Type declaration** A declaration statement that creates a new data type. A type declaration must include a type name and a description of the entire set of possible values for that type.

**Universal logic cell** A logic cell capable of forming any combinational logic function of the number of inputs to the cell. RAM, ROM and multiplexers have been used to form universal logic cells. Sometimes they are also called look-up tables or function generators.

**Usable gates** Term used to denote the fact that not all gates on an FPLD may be accessible and used for application purposes.

**Variable** In VHDL a data object that has only current value that can be changed in variable assignment statement.

**VCC** A high-level input voltage represented as a high (1) logic level in binary group values. It is a default active node value in AHDL.

**Verilog** Hardware description language used for description of digital systems for simulation and synthesis purposes. Language reference is fully described in IEEE 1364-1995.

**VHDL** Acronym for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. VHDL is used to describe function, interconnect and modeling. Language reference is fully described in IEEE 1076-1993

# SELECTED READING

Due to the large amount of literature in the area of field-programmable logic, digital systems design, and hardware description languages we only suggest some of the very good further readings.


Ashenden, P. *The Designer's Guide to VHDL*, Morgan Kaufmann, 1996

Bashker, J. *A Guide to VHDL Syntax*, Prentice-Hall, 1995

Bolton, M. *Digital Systems Design with Programmable Logic*, Addison-Wesley Publishing Co., 1990.

Brown, S. et al., *Field-Programmable Gate Arrays*, Kluwer Academic Publishers, 1992.

Brown, S. and Rose, J.. "FPGA and CPLD Architectures: A Tutorial", IEEE Design and Test of Computers, Summer 1996.

Chang, K.C. Digital Systems Design with VHDL and Synthesis, IEEE Computer Society Press, 1999

Dewey, A. *Analysis and Design of Digital Systems with VHDL*, PWS Publishing Company, 1997

Gajski, D.D. *Principles of Digital Design*, Prentice Hall International, 1998
Jenkins, J. H. *Designing with FPGAs and CPLDs*, Prentice-Hall, 1994

Hamblen, J. and Furman, D. *Rapid prototyping of Digital Systems – A Tutorial Approach*, Kluwer Academic Publishers, 2000

Perry, D. *VHDL*, Second Edition, McGraw-Hill, 1994.

Rose J., El Gamal A., and Sangiovanni-Vincentelli A. "Architecture of Field-Programmable Gate Arrays", Proc. IEEE, Vol. 81, No.7, July 1993.

Roth, C.H. *Digital Systems Design Using VHDL*, PWS Publishing Co., 1998
Salcic, Z. "SimP- A Simple Custom-Configurable Processor Implemented in FPGA", Tech. Report no.567/96, Auckland University, Department of Electrical and Electronic Engineering, July 1996.

Salcic, Z. Maunder B., "SimP - a Core for FPLD-based Custom-Configurable Processors", Proceedings of International Conference on ASICS - ASICON '96, Shanghai, 1996.

Salcic, Z., Maunder B. "CCSimP - An Instruction-Level Custom-Configurable Processor for FPLDs", Field-Programmable Logic 96, R.Hartenstein, M.Glesner (Eds), Lecture Notes in Computer Science 1142, Springer, 1996.

Salcic, Z. *VHDL and FPLDs in Digital Systems Design, Prototyping and Customization,* Kluwer Academic Publishers, 1998

Shakill, K. and Cypress Semiconductor *VHDL for Programmable Logic*, Addison-Wesley, 1996

Smailagic, A., et. al. "Benchmarking an Interdisciplinary Concurrent Design Methodology for Electronic/Mechanical Systems" Proc. ACM/IEEE Design Automation Conference, June 1995 San Francisco, CA. 514-519.

Smailagic, A., Siewiorek, D.P. "A Case Study in Embedded System Design: The VuMan2 Wearable Computer", IEEE Design and Test of Computer, Vol. 10, No. 3, 1993; 56-67.

Smailagic, A., Siewiorek, D.P. "The CMU Mobile Computers and Their Application For Maintenance", Mobile Computing, Eds. T. Imielinski and H. Korth, Kluwer Academic Publishers, January 1996.

Smailagic,A., Siewiorek, D.P. "Interacting with CMU Wearable Computers", IEEE Personal Communications, Vol.3, No.l, Feb. 1996; 14-25.

Smailagic,A., Amon, C. H. et. al. "Concurrent Design and Analysis of the Navigator Wearable Computer System", IEEE Transactions on Components, Packaging, and Manufacturing Technology, Vol.18, No. 3, Sept. 1995, 567-577.

Trimberger, S., ed. *Field-Programmable Gate Array Technology*, Kluwer Academic Publishers, 1994.

WakerleyJ. F. *Digital Design Principles and Practices*, Prentice Hall, 1990

Proc. IEEE Symposium FPGAs for Custom-Computing Machines, IEEE Computer Society Press, Los Alamitos. 1993-1998.

Proc. Of Field-Programmable Logic, FPL, conferences held annualy in Europe, most of them printed as Lecture Notes in Computer Science by Springer-Verlag

*Max+PLUS II Programmable Logic Development System: AHDL*, Altera Corporation, 1995.

*VHDL - Language Reference*, IEEE Press, 1994.

Various data sheets, application notes and application briefs by Altera Co., Atmel Co. and Xilinx Co. that can be found on http://www.altera.com, http://www.atmel.com and http://www.xilinx.com, respectively.

# WEB RESOURCES

The following list of Web sites represents a starting list of useful Web links related to concrete FPLD families, hardware description languages and synthesis and simulation tools. Most of these sites contain many further useful links.

***www.altera.com***

Altera Corporation produces complex FPLD devices and design tools that include AHDL, Verilog and VHDL synthesis tools and simulation tools that support those devices. The readers can also find full data sheets and application notes related to Altera UP-1 prototyping board that was used to test most of examples in this book.

***www.atmel.com***

Atmel produces complex FPLD devices and design tools that support those devices.

***www.cadence.com***

Cadence Design Systems is a major vendor of electronic design tools that also include VHDL and Verilog-related products.

***www.cypress.com***

Cypress Semiconductor produces complex PLDs and FPGAs and related VHDL synthesis tools. It now provides a complete PLD design environment including both VHDL and Verilog synthesis.

***www.eda.org***

The Electronic Design Automation (EDA) and Electronic Computer-Aided Design (ECAD) one-stop standards resource on the World Wide Web!

***www.latticesemi.com***

Lattice Semiconductor Corporation produces FPGA devices and complex PLDs, and provides design tools that support those devices including VHDL and Verilog Synthesis tools.

***www.standards.ieee.org***

IEEE standards and related issues including VHDL and Verilog standardization documents and working groups.

***www.mentorg.com***

Mentor Graphics is a major vendor of electronic design tools that also include VHDL and Verilog-related products.

***www.orcad.com***

OrCad is one of the major vendors of personal computer based electronic design tools that inlcude hardware description languages.

***www.ovi.org***

Open Verilog International (OVI) drives worldwide development and use of standards required by systems, semiconductor and design tools companies, which enhance a language-based design automation process.

***www.syncad.com***

SynaptiCAD, an important source for timing analysis and VHDL & Verilog generation and simulation software.

***www.synopsys.com***

Synopsys is a major vendor of electronic design tools that also include VHDL and Verilog-related synthesis and simulation products.

***www.verilog.net***

A source of various useful Verilog-related information and links.

***www.vhdl.org***

VHDL International - an organization dedicate to cooperatively and proactively promoting VHDL as standard worldwide language for design and description of electronic systems.

***www.viewlogic.com***

Viewlogic is a vendor of electronic design tools that also include VHDL and Verilog-related synthesis and simulation products for both personal computers and workstations.

***www.xilinx.com***

Xilinx produces FPGAs and other types of FPLD devices and provides VHDL and Verilog design tools that support those devices.

# INDEX


***A***

Address bus 260
Address decoders 148,156
Addressing (modes, SimP) 257
AHDL 122, 143, 185
Alias (VHDL) 354
Always block
Altera FPLDs 43, 54, 75, 80
ALU (Arithmetic-logic unit)
Antifuse 12
Architecture (in VHDL) 324-328

Array
- AHDL (see group)
- Verilog
-VHDL 351
Assert 381
Assignment In
-AHDL 146
- Verilog 494, 507
-VHDL 339, 340, 376
Atmel FPLDs 107

***B***

Baud Rate Generator 477
BCD counter 466
Behavioral style architecture 324
Bit (in VHDL) 335
Bit_vector 335

***C***

Carry chain 55
Cascade chain 55
Cell (Logic) 17
CLB (Configurable Logic Block) 103
Combinatorial logic
-in AHDL 149,152
-inVerilog 530
-inVHDL 392

Component 209
Component instantiation 209
Concurrent statements 144, 317, 326, 383,398
Conditional logic 152,533
Conditionally generated logic 217
Configuration (in VHDL) 329
Configuration scheme (FPLD) 67
Control unit
-inSimP 262,276
- in Pipelined SimP 565,568
Counter
-in AHDL 162
-inVerilog 542
-inVHDL 421
CPLD 10
Custom instruction 264-265
Custom-Computing Machines 38

***D***

Data bus 259
Datapath 36
Data path
-SimP 259,267
- Pipelined SimP 563,571
De Morgan's inversion 44
Decoder 154
Dedicated I/O 65
Design entry 120
Design verification 128
Display circuitry 241
Dynamic reconfigurability 37

***E***

Electronic lock 223
Entity (in VHDL) 322
Enumeration Type (VHDL) 348
Expander (shareable) 46

***F***

Fitting 134
FLEX devices 54-90
Flip-flop 194
Floating gate programming technology 15
For loop
-AHDL 217
-Verilog 514
-VHDL 379
FPGA (see FPLD)
FPLD 1,7,13
Frequency divider 214
FSM (Finite State Machine)
-in AHDL 163
-inVerilog 548
-inVHDL 431
Function
-in AHDL 191, 204, 210
-inVerilog 517
-inVHDL 384
Function prototype (AHDL) 191, 199, 204, 210
Functional simulation 128
Functional unit (SimP) 264-265

***G***

Gate array 1-5
Global signal 65
Glue logic 34
Group or array (AHDL) 151

***H***

Hardware accelerator 35
HDL (Hardware Description Language)
Hierarchy (of Design Units)

***I***

Include file 125,136
Input vectors 131,132,139,140
Input/output block 8
Instruction set 256
Instruction execution
- SimP 262
- Pipelined SimP 565
Interrupt circuitry 287

***K***

Keypad encoder

***L***

LAB (Logic Array Block)
Latch 194, 541
Library
-AHDL
-VHDL
Logic cell
Logic element

LPM (Logic Parameterized Module)
LUT (Look-up Table)

*M*

Macrocell 46
Max + Plus II 130
Memory controller 298
Microprocessor
-SimP 255
- Pipelined SimP 559
Module (Verilog) 507

*N*

Net (Verilog) 495
Netlist 116
Node(AHDL) 150

*O*

Object
-inVerilog 494
-inVHDL 337
One-hot encoding 72
Operation decoder 276
Operators
-inAHDL 186
-inVerilog 501
-inVHDL 340

*P*

Package (inVHDL) 321
Parameters
-AHDL 213
-Verilog 497
Partitioning 134
Pipelining 72
Pipelining (SimP) 559
Placement 134
Primitives (Design) 191
Port
-AHDL
-Verilog 511
-VHDL 322
Procedural statements (Verilog) 513
Program counter (see SimP)
Programmable switch 13
Programming (FPLDs) 13
Pulse distributor (see SimP)

*R*

Rapid system prototyping (Vumn) 295
Reconfigurable hardware 37
Record (data type, VHDL) 354
Register
-AHDL 159
-Verilog 496,542
-VHDL 421
Reset circuitry
-SimP 285
- Pipelined SimP 565, 578
Routing 26

*S*

SART (Serial Receiver/Transmitter) 475
Schematic entry 121
Sequence recognizer 459
Sequential logic
-AHDL 159
-Verilog 540
-VHDL 415
Signal (VHDL) 340
SimP microprocessor 255
Simulation 128,137
SRAMFPGAs 11
SRAM programming technology 13
Stack pointer (see SimP)
Structural (model, VHDL) 328

*T*

Temperature controller 236

Truth table (AHDL) 152

*V*

Variable (VHDL) 339
Variable section (AHDL) 144
Verilog 493 - 496
VHDL 313-491
Virtual hardware 37
VuMan 295

*W*

Working register (see SimP)

*X*

Xilinx FPLDs 91

***Digital Systems Design and Prototyping: Using Field Programmable Logic and Hardware Description Languages, Second Edition*** includes a CD-ROM that contains Altera's MAX+PLUS II Student Edition programmable logic development software. MAX+PLUS II is a fully integrated design environment that offers unmatched flexibility and performance. The intuitive graphical interface is complemented by complete and instantly accessible on-line documentation, which makes learning and using MAX+PLUS II quick and easy. MAX+PLUS II version 9.23 Student Edition offers the following features:

- ✓ Operates on PCs running Windows 95/098, or Windows NT 4.0
- ✓ Graphical and text-based design entry, including the Altera Hardware Description Language (AHDL), VHDL and Verilog
- ✓ Design compilation for product-term (MAX 7000S) and look-up table (FLEX 10K) device architectures
- ✓ Design verification with functional and full timing simulation

The MAX+PLUS II Student Edition software is for students who are learning digital logic design. By entering the designs presented in the book or creating custom logic designs, students develop skills for prototyping digital systems using programmable logic devices.

**Registration and Additional Information**

To register and obtain an authorization code to use the MAX+PLUS II software, go to: ***http://www.altera.com/maxplus2-student.*** For complete installation instructions, refer to the ***read.me*** file on the CD-ROM or to the ***MAX+PLUS II Getting Started Manual,*** available on the Altera world-wide web site **(http://www.altera.com).**

This CD-ROM is distributed by Kluwer Academic Publishers with *ABSOLUTELY NO SUPPORT* and *NO WARRANTY* from Kluwer Academic Publishers.

Kluwer Academic Publishers shall not be liable for damages in connection with, or arising out of, the furnishing, performance or use of this CD-ROM.