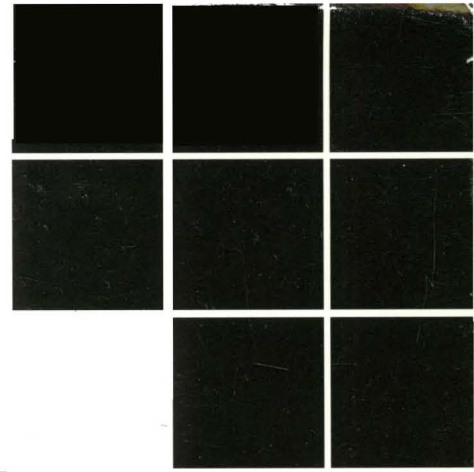
VHDL



For Programmable Logic

CYPRESS

VHDL for Programmable Logic

Kevin Skahill
Cypress Semiconductor

Cypress Semiconductor, 3901 North First Street, San Jose, CA 95134 (408) 943-2600
Telex: 821032 CYPRESS SNJ UD, TWX: 910 997 0753, FAX: (408) 943-2741
Fax-on-Demand: (800) 213-5120, http://www.cypress.com

Published August 14, 1995.

Contributing Authors

Jay Legenhausen

Ron Wade

Corey Wilner

Blair Wilson

Acknowledgements

Greg Somer for providing materials on the network repeater and patiently describing its operation time and time again. Haneef Mohammed, Alan Copola, and Jeff Freedman for consultation on VHDL, synthesis, and fitting. Garett Choy for endless work on drawing and redrawing figures, as well as formatting the document. David Johnson for help with figures and managing many issues involved with the production of this book. Chris Jones for providing valuable feedback, corrections, and suggested exercises. Steve Klinger for reviewing the text and generating the index. Krishna Rangasayee for assistance in the development of exercises. Caleb Chan for assistance with the quick-reference guide. Rich Kapusta for assistance with the glossary. Terri Fusco for coordinating the copy-editing, printing, and design of the book jacket. Nan Schwieger for copy-editing a large amount of material in a very short time. Al Graf, David Barringer, and Cypress Semiconductor for providing the opportunity and time to pursue this endeavor. Last, and most important, the many Cypress customers who call daily with questions regarding VHDL and programmable logic.

Table of Contents

1 Introduction

The VHSIC (Very High Speed Integrated Circuit) hardware description language, VHDL, is a product of the VHSIC program funded by the Department of Defense in the 1970's and 1980's. When the language was first developed, it was intended to be primarily a standard means to document complex circuits so that a design documented by one contractor could be understood by another. Today "VHDL" is a bit of a misnomer because it is used not only for integrated circuits or as a hardware description language, it is also used for the modeling and synthesis of integrated circuits, programmable logic, board-level designs, and systems. VHDL was established as the IEEE 1076 standard in 1987. In 1988, MilStd454 required that all ASICs delivered to the Department of Defense be described in VHDL, and in 1993, the IEEE 1076 standard was updated while an additional VHDL standard, IEEE 1164, was adopted.

Today VHDL is an industry standard for the description, modeling, and synthesis of circuits and systems. In 1993, the market for VHDL simulators exceeded the Verilog simulator market (*Electronic Engineering Times,* 10/3/94). The entire synthesis market has reached approximately $100 million, with a growth rate of 20%-30% per year.[1] This explosion in the use of HDLs in general, and VHDL in particular, has created the need for engineers in all facets of the electronics industry—from ASIC designers to system level designers alike—to learn VHDL.

VHDL is particularly well suited as a language for designing with programmable logic, and it is gaining in popularity. Designing with larger capacity CPLDs (complex programmable logic devices) and FPGAs (field programmable gate arrays) of 600 gates to 20K gates, engineers can no longer use Boolean equations or gate-level descriptions to quickly and efficiently finish a design. VHDL enables designers to describe large circuits rapidly and bring products to market quickly. In addition to providing high-level design constructs and efficiency, VHDL delivers portability of code between synthesis and simulation tools, device independent design, and easy ASIC migration. VHDL is an open, standard language, not a proprietary language. Because VHDL is a standard, VHDL code can be ported from one synthesis tool (or simulation tool) to another with little or no modification to the code. Because VHDL is a standard, VHDL design descriptions are device independent, allowing the designer to easily benchmark design performance in multiple device architectures. The same code used for designing with programmable logic can be used by an ASIC vendor to produce an ASIC when production volumes warrant a conversion.

VHDL serves the needs of designers at many levels and as such is a complex language. For this reason, this book (unlike other VHDL books presently on the market) limits its scope to the discussion of VHDL as a language for the synthesis of design descriptions into logic to be implemented in programmable logic devices. Rather than exploring numerous possible uses of VHDL on different levels—the description, modeling, and synthesis of circuits ranging from ASICs to systems—we will explore a number of applications of VHDL for designing with programmable logic. Much of what you will learn from this book will be applicable to other uses of VHDL, and at times our discussion will draw on modeling concepts in order to clarify what it means to a design engineer to write VHDL code for synthesis versus simulation. So, although the focus of this book is VHDL synthesis as it pertains to programmable logic, you will be able to easily apply what you will learn in our discussion to other aspects of VHDL.

1. Private correspondence, In-Stat.

Intended Audience

This book is intended to meet the needs of the professional engineer interested in updating his or her design methodologies to include both VHDL and programmable logic. The focus is not so much on teaching logic design, but rather on using VHDL and programmable logic to solve design problems. To this end, the text explains the architectures, features, and technologies of programmable logic and teaches how to write VHDL code that is intended for synthesis. Code constructs are evaluated to determine how synthesis software will interpret the code and produce logic equations or netlists. The approach is to use many practical design examples in order to encourage coding practices that will result in code that can be synthesized efficiently. Thus, examples focus on state machine design, counters, shifters, arithmetic circuits, control logic, FIFOs, and other "glue logic" that designers typically implement in programmable logic. This approach is in contrast to teaching modeling techniques, which tend to focus on abstract and theoretical designs but which cannot be synthesized either because the designs are physically meaningless or use constructs that are not particularly well suited for synthesis. A programmable logic primer is presented as the first technical chapter. It starts with the basics—discrete TTL logic components—so that no knowledge of programmable logic is assumed, but quickly reaches a detailed discussion of the architectures of CPLDs and FPGAs. The engineer already versed in the available programmable logic devices and their features may wish to skip or skim this chapter. A later chapter is devoted to case studies of one CPLD (FLASH370™) and one FPGA (pASIC380) to illustrate common issues with optimizing designs for these types of architectures. The chapter frames the studies around a discussion of the synthesis and fitting (place and route) processes. It is intended to equip the reader to effectively use device resources and achieve the maximum performance. Finally, a chapter is devoted to writing test benches with which to simulate source-level VHDL code. The same test benches can be used to simulate post-fit models of the design implementation that are produced by fitters and place and route tools.

The content is also appropriate for a junior/senior undergraduate or first-year graduate course with a focus on using VHDL to design with programmable logic. Although the text does not teach logic design, it does describe the design process for several design examples, including the design of the core logic for a 100BASE-T4 network repeater. This is a topical design example that illustrates the use of programmable logic in today's fast-growing network communications market. Traditional design methodologies are reviewed in that they are compared to designing with VHDL and programmable logic.

Why Use VHDL?

There's one crucial reason why you should learn and use VHDL: *It's become an industry standard.* Every design engineer in the electronics industry should soon learn and use a hardware description language to keep pace with the productivity of competitors. With VHDL, you can quickly describe and synthesize circuits of five, ten, twenty, or more thousand gates. Equivalent designs described with schematics or Boolean equations at the register transfer level would likely require several months of work by one person. Above and beyond that, VHDL gives you a language that provides the capabilities described below.

Power and Flexibility

VHDL not only gives you the power to describe circuits quickly using powerful language constructs but also permits other levels of design description including Boolean equations and structural netlists. *Figure 1-1* illustrates four ways to describe a 2-bit comparator.

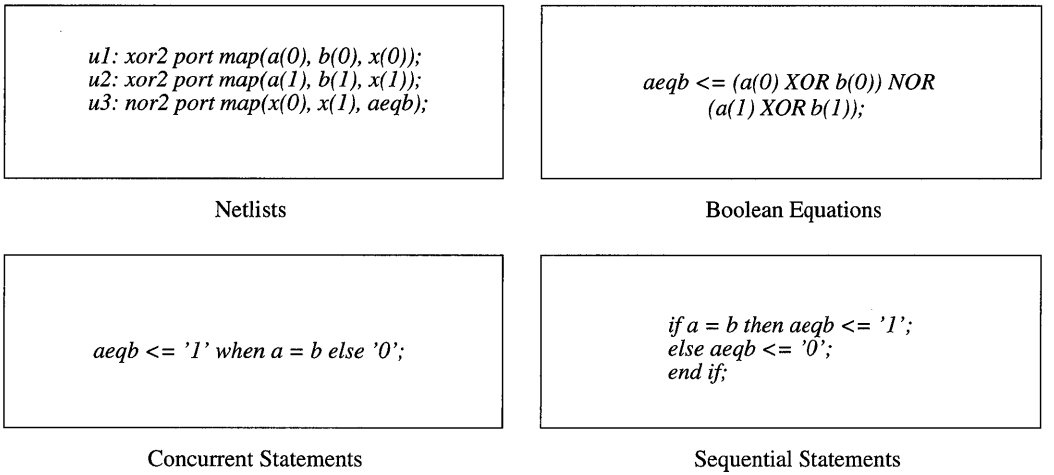


Figure 1-1 VHDL permits several classes of design description.

Device-Independent Design

VHDL permits you to create your design without having to first choose a device for implementation. With one design description, you can target many device architectures: You do not have to become intimately familiar with a device's architecture in order to optimize your design for resource utilization or performance. Instead, you can concentrate on creating your design.

Portability

VHDL's portability permits you to take the same design description that you used for synthesis and use it for simulation. For designs that require thousands of gates, being able to simulate the design description before synthesis and fitting (or place and route) can save you valuable time. Because VHDL is a standard, your design description can be taken from one simulator to another, one synthesis tool to another, and one platform to another. *Figure 1-2* illustrates that the source code for a design can be used with any synthesis tool and that the design can be implemented in any device that is supported by the chosen synthesis tool.

Benchmarking Capabilities

Device independent design and portability give you the ability to benchmark your design using different architectures and different synthesis tools. You no longer have to know before you start your design which device you are going to use or whether it's going to be a CPLD or an FPGA. You can take your completed design description and synthesize it, creating logic for an architecture of your choice. You can then evaluate the results and choose the device that best fits your design requirements. The same can be done for synthesis tools in order to measure the quality of the synthesis.

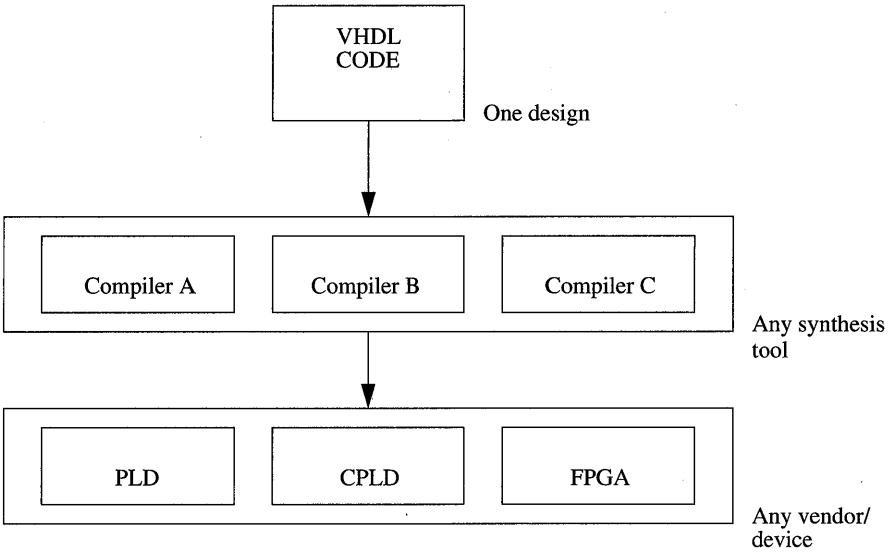


Figure 1-2 VHDL provides portability between compilers and device independent design.

ASIC Migration

The same design description used to synthesize logic for a programmable logic device can be used for an ASIC when production volumes ramp. VHDL permits your product to hit the market quickly in programmable logic by synthesizing your design description for an FPGA. When production volumes reach appropriate levels, the same VHDL code can be used in the development of an ASIC.

Fast Time-to-Market and Low Cost

VHDL and programmable logic pair well together to facilitate a speedy design process. VHDL permits designs to be described quickly. Programmable logic eliminates nonrecurring expenses (NREs) and facilitates quick design iterations. Synthesis makes it all possible. VHDL and programmable logic combine as a powerful vehicle to bring your products to market in record time.

Shortcomings

There are three common concerns expressed by design engineers about VHDL: (1) You give up control of defining the gate-level implementation of circuits that are described with high-level, abstract constructs, (2) the logic implementations created by synthesis tools are inefficient, and (3) the quality of synthesis varies from tool to tool.

There's no way around the first "shortcoming." In fact, the intent of using VHDL as a language for synthesis is to free the engineer from having to specify gate-level circuit implementation. However, if you understand how the compiler synthesizes logic, you're likely to realize that many (if not most) constructs will be implemented optimally by the compiler and you will have little need—or desire—to dictate implementation policy. Many synthesis tools do allow designers to specify technology-

specific, gate-level implementations. Descriptions of these types, however, are neither high level nor device independent.

The concern that logic synthesis is inefficient is not without warrant. Admittedly, VHDL compilers will not always produce optimal implementations. Compilers use algorithms to decide upon logic implementations, following standard design methodologies. An algorithm cannot look at a design problem in a unique way. Sometimes, there is no substitute for your human creativity, and in such cases you will want to code your design in such a way as to control the design implementation. The complaint that synthesis provides poor implementations is also commonly the result of sloppy coding. Similar to sloppy C coding, which results in slow execution times or poor memory utilization, sloppy VHDL coding will result in unneeded, repetitive, or non-optimal logic.

The third shortcoming (some synthesizers are better than others) is being addressed by the marketplace. Fortunately VHDL synthesis for CPLDs and FPGAs is emerging from its infancy and the competition is growing. Only the strong vendors will survive.

Even if you are the most cynical of engineers, believing that these shortcomings are too large to make it worthwhile to use VHDL at this point, take heed of the larger picture. As an engineer you should resist the temptation to believe that the details of a design's implementation are as important as the design objectives: fulfilling the design requirements, meeting the price-point, and getting the product to market quickly. Meeting design objectives is essential, the implementation is secondary.

Using VHDL for Design Synthesis

The design process can be broken into the six steps enumerated below. We'll explain each briefly in this section.The remainder of the text will focus primarily on steps 2 and 4. The last chapter will focus on steps 3 and 5.

1. Define the Design Requirements
2. Describe the Design in VHDL (Formulate and Code the Design)
3. Simulate the Source Code
4. Synthesize, Optimize, and Fit (Place and Route) the Design
5. Simulate the Post-fit (layout) Design Implementation
6. Program the Device

Define the Design Requirements

Before launching into writing code for your design, you should have a clear idea of the design objectives and requirements. What is the function of the design? What are the required setup and clock-to-out times, maximum frequency of operation, and critical paths? Having a clear idea of the requirements may help you to choose a design methodology and perhaps help you to choose a device architecture to which you will initially synthesize your design.

Describe the Design in VHDL

Formulate the Design

Having defined the design requirements, you may be tempted to jump right into coding, but until you become a seasoned VHDL coder, we suggest that you first decide upon a design methodology. By having an idea of how your design will be described, you will tend to write more efficient code that

does what you intended. Spending a few moments deciding upon a design methodology can prove well worth the time.

You're probably already familiar with the methodologies: top-down, bottom-up, or flat. The first two methods involve creating design hierarchies, the latter involves describing the circuit as one monolithic design.

The top-down approach requires that you divide your design into functional blocks, each block having specific (but not always unique) inputs and outputs and performing a particular function. A netlist is then created to tie the functional blocks together, after which the blocks themselves are designed. The bottom-up approach involves just the opposite: defining and designing the individual blocks of a design, then bringing the different pieces together to form the overall design. A flat design is one in which the details of functional blocks are defined at the same level as the interconnection of those functional blocks.

Flat designs work well for small designs, where having details of the underlying definition of a functional block does not distract from understanding the functionality of the chip-level design. For many of the smaller designs in this text, a flat design approach is used. The design of the core logic for a network repeater (chapter 6, "The Design of a 100BASE-T4 Network Repeater"), however, is divided into its constituent functional blocks. Hierarchical designs can prove useful in large designs consisting of multiple, complex functional blocks. Levels of hierarchy can clarify the interconnection of blocks and the design objectives. Avoid using too many levels of hierarchy, however, because excessive levels make it difficult to understand the interconnection of design elements and the relevance to the overall design.

Code the Design

After deciding upon a design methodology, you can launch into coding your design, being careful of syntax and semantics. If you're like many engineers, you will simply follow or edit an existing example to meet your particular needs. The key to writing good VHDL code is to think in terms of hardware. More specifically, think like the synthesis software (or compiler, as it is sometimes called) "thinks" so that you will have an understanding of how your design will be realized. A good portion of this book is devoted to helping you think like synthesis software so that you will write efficient code.

Simulate the Source Code

For large designs, simulating the design source code with a VHDL simulator will prove time efficient. The process of concurrent engineering (performing tasks in parallel rather than in series) brings circuit simulation (once normally a downstream task) to the early stages of design. Preempted at the earliest possible point in the design cycle, you will be able to make design corrections with the least possible impact to your schedule. If the design does not simulate as expected, then you can check your code and make the appropriate corrections before proceeding. Otherwise you can simulate your design after fitting or place and route. For small designs, you lose little time by first synthesizing and fitting your design. (In fact, you will eliminate the time it takes for the pre-synthesis simulation.) However, for larger designs, for which synthesis and place and route can take a couple of hours, you can significantly reduce your design iteration/debugging cycle time by simulating your design source code before synthesis. Also, large designs are usually hierarchical, consisting of several sub-designs or modules. This modularity allows you to verify and debug the functionality of each sub-design before assembling the hierarchy, potentially saving considerable time over verifying and debugging one monolithic design.

Synthesize, Optimize, and Fit (Place and Route) the Design

Synthesis

We have already used the term "synthesis" two dozen times or more, and you probably have at least a vague idea of what synthesis is, but if you have been waiting for an explanation, here's one: Synthesis is the realization of design descriptions into circuits. In other words, synthesis is the process by which logic circuits are created from design descriptions. It is a process that should be described as taking something as input (a design description) and producing another thing as output (logic equations or netlists). For brevity, however, we may at times drop either the input or output, and use phrases such as "synthesize logic descriptions," "synthesize logic," and "logic synthesis."

VHDL synthesis software tools convert VHDL descriptions to technology-specific netlists or sets of equations. Synthesis tools allow designers to design logic circuits by creating design descriptions without having to perform all of the Boolean algebra or create technology-specific, optimized net-lists. A designer presents only an abstract description of his or her design, specifying the way that the design is expected to "behave," and the synthesis software produces a set of equations to be fitted to a PLD/CPLD or a netlist to be placed and routed.

Synthesis should be technology specific. *Figure 1-3* illustrates the synthesis and optimization processes. Before synthesizing a design, a software tool must read the design and parse it, checking for syntax (and possibly a few semantic) errors. The synthesis process then converts the design to internal data structures, allowing the "behavior" of a design to be translated to a register transfer level (RTL) description. RTL descriptions specify registers, signal inputs, signal outputs, and the combinational logic between them. Other RTL elements depend on the device-specific library. For example, some programmable logic devices contain XOR gates. At this point, the combinational logic is still represented by internal data structures. Some synthesis tools will search the data structures for identifiable operators and their operands, replacing these portions of logic with technology-specific, optimized components. These operators can be as simple as identifying the use of an XOR gate for an architecture that has one, or as complicated as inferring a 16-bit add operation. Other portions of logic that are not identified are then converted to Boolean expressions that are not yet optimized.

Optimization

The optimization process depends on three things: the form of the Boolean expressions, the type of resources available, and automatic or user-applied directives (sometimes called constraints). Some forms of expressions may be mapped to logic resources more efficiently than others. For example, whereas a minimal sum of products can be implemented efficiently in a PAL, a canonical sum of products can be more efficiently mapped to a multiplexer or RAM. Sometimes a canonical or minimal product of sums may be the best representation for the target technology. Other user or automatic constraints may be applied to optimize expressions for the available resources. These constraints may be to limit the number of appearances of a literal in an expression (to reduce signal loading), limit the number of literals in an expression (to reduce fan-in), or limit the number of terms in an expression (to limit the number of product terms).

Optimizing for CPLDs usually involves reducing the logic to a minimal sum of products, which is then further optimized for a minimal literal count. This reduces the product term utilization and number of logic block inputs required for any given expression. These equations are then passed to the fitter for further device-specific optimization. Optimizing for FPGAs typically requires that the logic be expressed in forms other than a sum of products. Instead, systems of equations may be

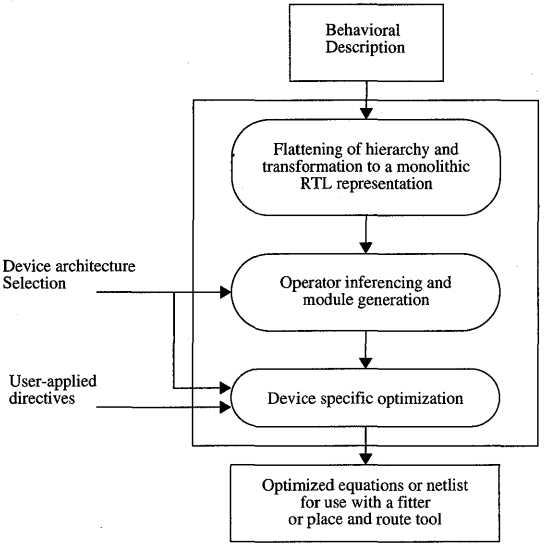


Figure 1-3 Synthesis process

factored based on device-specific resources and directive-driven optimization goals. The factors can be evaluated for efficiency of implementation. Criteria can be used to decide when to factor the system of equations differently or whether to keep the current factors. Among these criteria is usually the ability to share common factors, the set of which can be cached to compare with any newly created factors.

Another method for optimization does not involve as much manipulation of Boolean expressions. It involves the use of a binary decision diagram to map a given equation to a specific logic implementation.

Fitting

"Fitting" is the process of taking the logic produced by the synthesis and optimization processes, and placing it into a logic device, massaging the logic (if necessary) to obtain the best fit. Fitting is a term typically used to describe the process of allocating resources for CPLD-type architectures. Placing and routing differs from fitting and is used for FPGAs. Placing and routing is the process of taking logic produced by synthesis and optimization, packing the logic (massaging it if necessary) into the FPGA logic structures (logic cells), placing the logic cells in optimal locations, and routing signals from logic cell to logic cell or I/O. For brevity, we will use the terms "fit" and "place and route" interchangeably, leaving you to discern when we mean to use one or both terms.

Fitting designs in a CPLD can be a complicated process because of the numerous ways in which to start placing logic in the device. Before any placement, however, the logic equations are further optimized, again depending upon the available resources. For example, some macrocells permit the

configuration of a flip-flop to be D-type or T-type as well as allow the output polarity to be selected. In such a case, logic expressions for the true and complement of both the D-type and T-type should be generated. The optimal implementation can then be chosen from the four versions for each expression. After all equation transformations, expressions that share scarce resources (perhaps resets and presets, clocks, output enables, and input macrocells) can be grouped together. Expressions are also grouped together based upon user constraints such as pin assignment. The groups can then be examined to verify that they can be placed together in a logic block. Thus the collective resources of the group must be evaluated: number of required macrocells, independent resets and presets, output enables, unique product terms, clocks (and clock polarity), total input signals, and total output signals. If any of the groups cannot fit within the physical limitations of a logic block, then the groups are adjusted. Next, an initial placement is attempted. If a legal placement is not found, the grouping process may start over, possibly initially based on a different scarce resource. Once locations are found for all of the logic elements, routing between inputs, outputs, and logic cells is attempted. If routing cannot be completed with the given placement, then the router may suggest a new grouping to achieve a successful routing. Once a routing solution has been found, a fit has been achieved.

Place and Route

Place and route tools have a large impact on the performance of FPGA designs. Propagation delays can depend significantly on routing delays. A "good" placement and route will place critical portions of a circuit close together to eliminate routing delays. Place and route tools use algorithms, directives, user-applied constraints, and performance estimates to choose an initial placement. Algorithms can then iterate on small changes to the placement to approach a layout that is expected to meet performance requirements. Routing may then begin, with global routing structures used first for high-fanout signals or signals that must route over large distances. Local routing structures may then be used to route local inter-logic cell and I/O signal paths.

Figure 1-4 shows the process of synthesizing, optimizing, and fitting the description

```
aeqb <= '1' when (a=b) else '0';
```

(an equality comparator) into a CPLD and an FPGA.

There are many synthesis tool vendors on the market today. Among the most popular are Exemplar, Synopsys, Data I/O, IST, ViewLogic, and several programmable logic vendors that provide their own device-specific synthesis. Each synthesis vendor uses different, proprietary algorithms for synthesizing logic, so resulting logic implementations from one vendor to another will vary (but be functionally equivalent), just as the results of compiling C code with different compilers do. Nonetheless, there are some fundamental synthesis policies that apply across synthesis tools, making a general discussion of synthesis possible.

Simulate the Post-Fit (Layout) Design Implementation

Even if you have performed a pre-synthesis simulation, you will want to simulate your design after it has been fitted (or placed and routed). A post-layout simulation will enable you to verify not only the functionality of your design but also the timing, such as setup, clock-to-output, and register-to-register times. If you are unable to meet your design objectives, then you will need either to resynthesize and fit your design to a new logic device, massage (using compiler directives) any combination of the synthesis or fitting processes, or choose a different speed grade device. You may

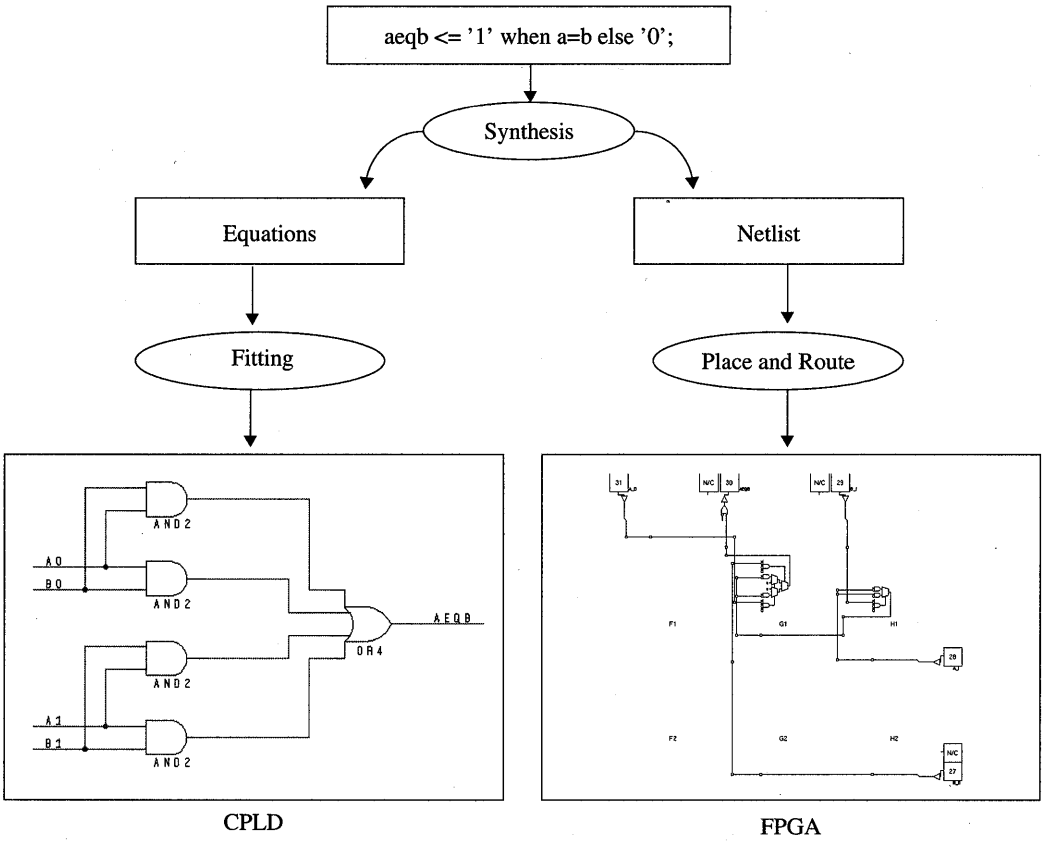


Figure 1-4 The process of synthesis to design implementation

also want to revisit your VHDL code to ensure that it has been described efficiently and in such a way as to achieve a synthesis and fitting result that meets your design objectives.

Program the Device

After completing the design description, and synthesizing, optimizing, fitting, and successfully simulating your design, you are ready to program your device and continue work on the rest of your system design. The synthesis, optimization, and fitting software will produce a file for use in programming the device.

Our System

We will be using Cypress's *Warp*™ VHDL synthesis software (on both the PC and Sun) to synthesize, optimize, and fit designs to CPLD and FPGA devices. *Warp* is an inexpensive software tool useful for learning and producing real designs for PLDs and FPGAs. The *Warp2*™ software

contains a VHDL synthesis tool and device-level simulator. The *Warp3*™ software contains the same VHDL synthesis tool as well as a VHDL simulator.

Summary

In this chapter we outlined the scope of this text, described some of the benefits of using VHDL and why the VHDL market is growing rapidly, and enumerated the design process steps, providing brief explanations of each. In the next chapter, we discuss programmable logic devices and their architectures.

Exercises

1. What are the advantages and benefits of using VHDL?

2. Give two examples where not doing source code simulation would be appropriate.

3. Give two design examples where doing source code simulation is appropriate in addition to post-layout simulation.

4. What are some concerns that designers voice when designing with VHDL?

5. How would the descriptions in Figure 1-1 change if the output is *aneqb* (a not equal to b)? What about *altb* (a lesser than) or *agtb* (a greater than b)?

2 Programmable Logic Primer

In this chapter, we explain the motivation for using programmable logic. We start at the beginning, without assuming that you already know what programmable logic is. Simple PLDs are explained using the 16L8,16R8, and 22V10 as examples. If you are already familiar with these architectures, you may wish to skip to the sections on CPLDs and FPGAs, where the architectural features of many popular devices are explained.

Why Use Programmable Logic?

Not long ago, Texas Instruments' TTL Series 54/74 logic circuits were the mainstay of digital logic design for implementing "glue logic": combinational and sequential logic for multiplexing, encoding, decoding, selecting, registering, and designing state machines and other control logic. These MSI and LSI logic circuits include discrete logic gates, specific boolean transfer functions, and memory elements, as well as counters, shift registers, and arithmetic circuits. *Table 2–1* describes just a few of the members of the 54/74 Series.

Table 2–1 TTL Series 54/74 logic circuits

54/74 Series	Description
7400	Quadruple 2-input positive-NAND gates $Y = \overline{AB}$
7402	Quadruple 2-input positive-NOR gates $Y = \overline{A + B}$
7404	Hex inverters $Y = \overline{A}$
7408	Quadruple 2-input positive-AND gates $Y = AB$
7430	8-input positive-NAND gates $Y = \overline{ABCDEFGH}$
7432	Quadruple 2-input positive-OR gates $Y = A + B$
7451	Dual AND-OR-INVERT gates $Y = \overline{AB + CD}$
7474	Dual D-type positive-edge-triggered flip-flops with preset and clear
7483	4-bit binary full adder with fast carry
7486	Quadruple 2-input exclusive-OR gates Y = AB = AB
74109	Dual J-K positive-edge-triggered flip-flops with preset and clear
74157	Quadruple 2-to-1 multiplexers
74163	Synchronous 4-bit counter with synchronous clear
74180	9-bit odd/even parity generator/checker
74374	Octal D-type flip-flops

Designing with TTL Logic

Armed with this inventory of TTL logic, you or any other digital designer could attack a problem following the standard design flow of *Figure 2-1*. Suppose, for example, that you are designing a network repeater consisting of four communication ports (A, B, C, and D). A collision signal, X, must be asserted if more than one port's carrier sense is active at a time. Signal X is to be synchronized to the transmit clock.

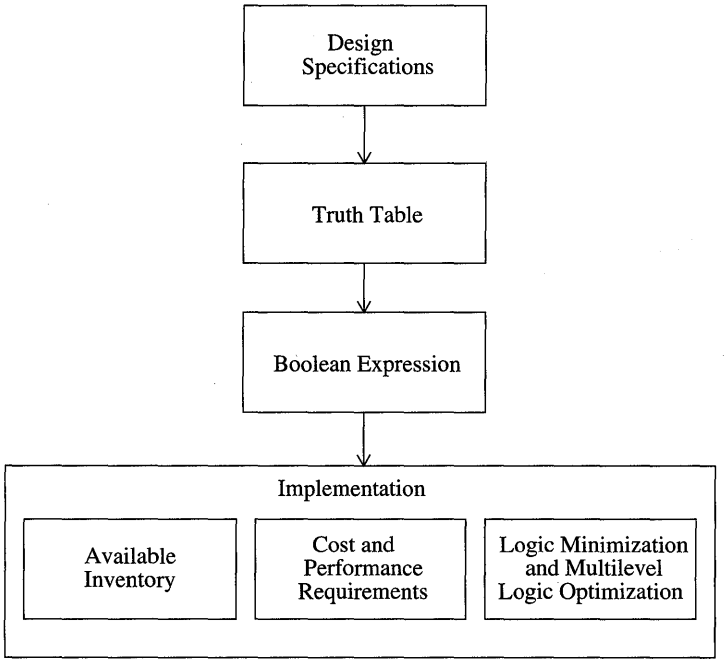


Figure 2-1 Design flow for designing with TTL logic

From the design specifications, we can generate a truth table (*Figure 2-2*), creating a sum of 11 product terms. Using boolean algebra or a Karnaugh map, we can reduce the expression for X to six product terms. An expression for the complement of X can be reduced to four product terms.

In determining how to implement the equation for X in TTL logic, we will want to minimize the number of resources (devices) required, minimize the number of levels of logic (each gate has an associated propagation delay) to minimize total propagation delay, or strike a balance between the two, depending on the performance and cost requirements of the design. Although the expression for the complement of X requires fewer product terms than the expression for X, the expression for the complement of X requires AND gates with a fan-in of three. Examining our inventory of TTL logic, we see that the AND functions in our inventory have a fan-in of only two. Creating wider AND gates would require cascading two-input AND gates. This cascading and the inverters that would be necessary to implement the complements of signals would unnecessarily increase the device count (increasing cost) and levels of logic (decreasing performance). If we implement the expression for X as a sum of products, this design will require two 7408s and two 7432s. In all, six levels of logic are

AB\CD	00	01	11	10
00	0	0	1	0
01	0	1	1	1
10	1	1	1	1
11	0	1	1	1

$$X = AB + CD + BD + BC + AD + AC$$

A	B	C	D	X
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

AB\CD	00	01	11	10
00	1	1	0	1
01	1	0	0	0
10	0	0	0	0
11	1	0	0	0

$$X = \bar{A}B\bar{C} + \bar{A}B\bar{D} + \bar{A}\bar{C}\bar{D} + B\bar{C}\bar{D}$$

Figure 2-2 Determining an expression for a collision signal

required: one level to produce each of the six product (AND) terms and five levels to produce the 6-input OR function by cascading the 2-input OR gates of the 7432. The product of sums implementation, $X = (A + B + C)\,(A + B + D)\,(A + C + D)\,(B + C + D)$, also requires several devices and levels of logic. An implementation that uses fewer devices (costs less) and requires fewer levels of logic (for better performance) is the NAND-NAND implementation. The equation $X = AB + CD + BD + BC + AD + AC$ may be rewritten in NAND-NAND form as $X = \left(\overline{\bar{A}B} \bullet \overline{\bar{C}\bar{D}} \bullet \overline{\bar{B}\bar{D}} \bullet \overline{\bar{B}\bar{C}} \bullet \overline{\bar{A}\bar{D}} \bullet \overline{\bar{A}\bar{C}}\right)$. This implementation requires two 7400s and one 7430, for a total of two levels of logic. Finally, to synchronize this signal to the transmit clock, we will use one 7474, wiring the circuit as in the circuit diagram of *Figure 2-3*. Depending on our current inventory of TTL devices, we could have implemented this design in one of several different ways.

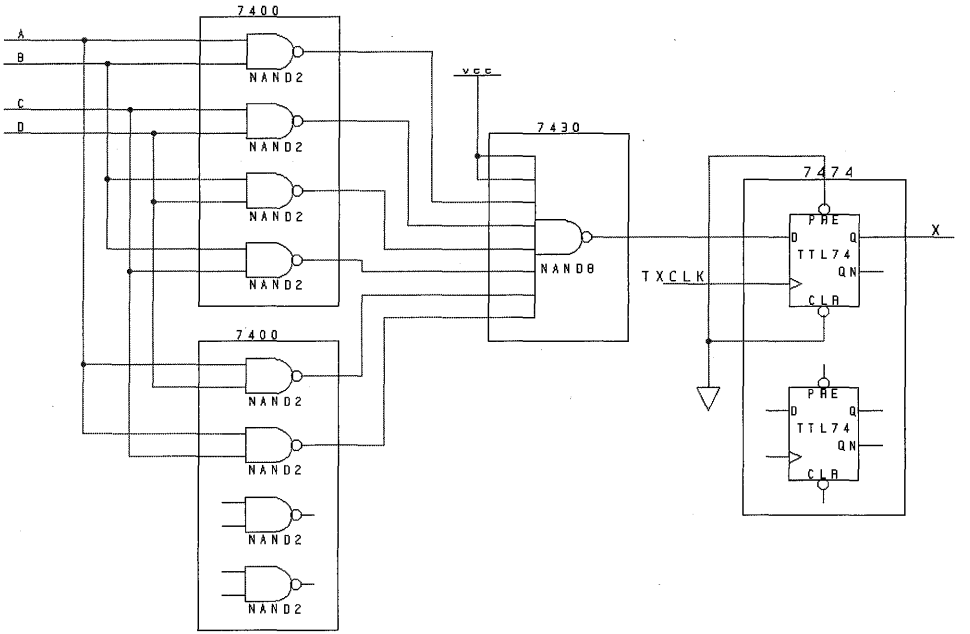


Figure 2-3 TTL logic implementation

How is implementing this design in programmable logic any different? Before we can answer that question, we need to understand what programmable logic is.

What Is a Programmable Logic Device?

PAL (Programmable Array Logic) devices, or PALs, are the simplest of programmable logic devices that are readily available in today's market. PALs consist of an array of AND gates and an array of OR gates in which the AND array is programmable and the OR array is fixed. To understand PAL architectures, consider the PAL 16L8 device architecture of *Figure 2-4*. (The 16L8 is available from several programmable logic vendors including Advanced Micro Devices, Cypress Semiconductor, Lattice Semiconductor, National Semiconductor, and Texas Instruments.)

The 16L8 derives its name from the fact that there are a total of 16 inputs into the AND array (the *logic* array, hence the L in 16*L*8) and 8 outputs. Eight of the inputs to the array are dedicated device inputs. Another eight inputs to the array are from the outputs of the three-state buffers (when enabled) or the I/O pins (when the three-state buffers are not enabled, the associated pins may be used as device inputs). The programmable AND array consists of 64 AND gates in which each AND gate may be used to create a product of any of the 16 inputs (the complements of the 16 logic array inputs may also be used). The OR array is fixed: Each of the eight OR gates sums seven products. The remaining eight product terms are used for each of the eight, three-state, inverting buffers.

16L8 Logic Diagram 20-Pin DIP/PLCC/LCC (28-Pin PLCC) Pinouts

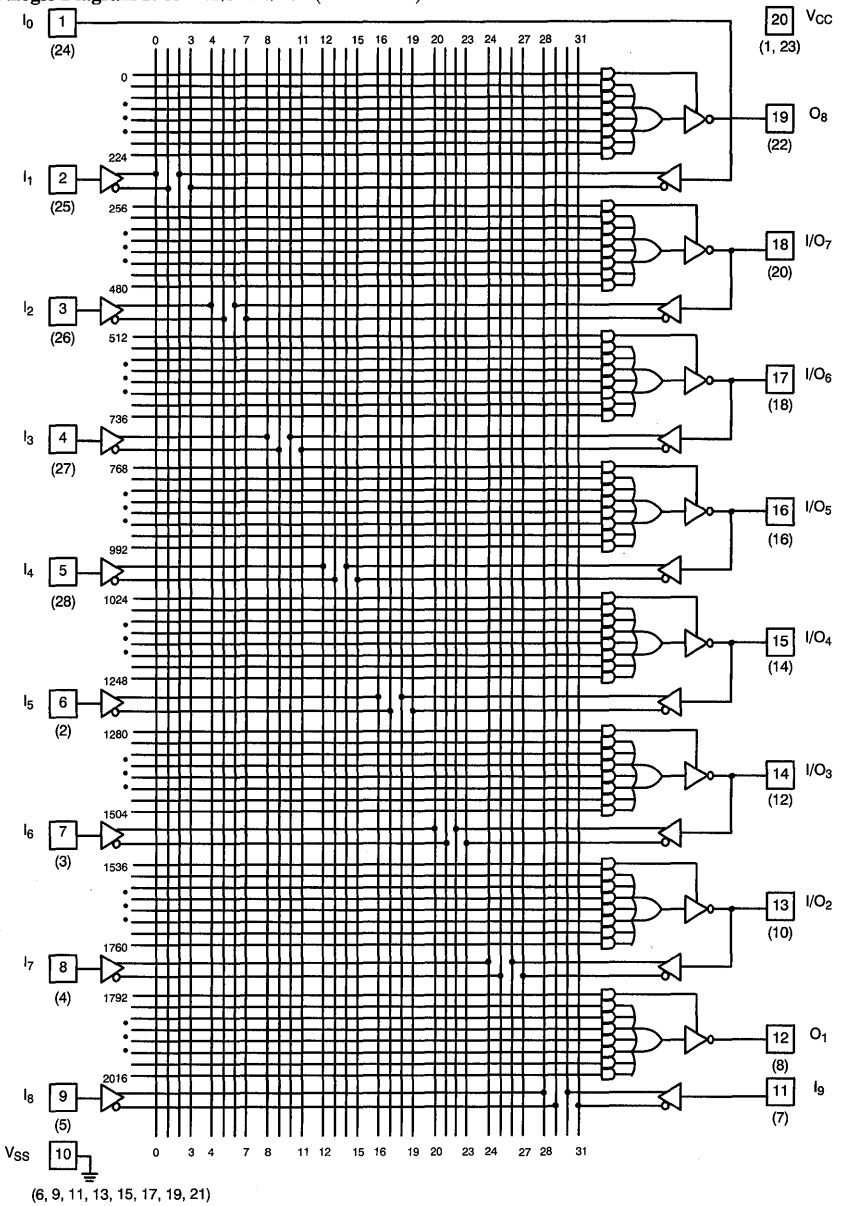


Figure 2-4 PAL 16L8 Device Architecture

Each of the three-state buffers is controlled by a product term. The product term may be programmed for a function of the inputs and their complements, or to ensure that the buffer is always on or always off. If always on, then the associated I/O pin functions as an output only. If always off, then the associated pin may function as an input only. If controlled by a product term, then the I/O pin can be used as a three-state output for use on an external bus or as a bidirectional signal in which the I/O pin may be driven externally and used internally in the logic array.

Next, consider the PAL 16R8 device architecture of *Figure 2-5*. Like the 16L8, this industry-standard architecture has 16 inputs to a logic array consisting of 64 product terms. In this architecture, eight of the logic array inputs are from device inputs and eight are from register feedbacks. Each of the OR gates sums eight product terms, and this sum is then registered by one of the output registers (the *R* in 16R8). The clock for the register is from a dedicated pin. The outputs cannot be configured as inputs, but may be configured as three-state outputs for use on an external bus. The three-state output is controlled by a dedicated pin.

Whereas the 16R8 provides registers, the 16L8 does not. Instead, the 16L8 provides combinatorial output, more flexible output enable control, and the capability to use more device pins as inputs. Two additional industry-standard architectures, the 16R6 and the 16R4, balance these differences. The 16R6 includes six registers, leaving two combinatorial output structures with individual three-state control. The 16R4 is the compromise between the 16L8 and 16R8 with four registered and four combinatorial output structures.

Designing with Programmable Logic

Consider, again, the design of the network repeater collision signal, X, for which an expression is given in *Figure 2-2*. Because this signal must be registered, of the PALs we have discussed, we can use the 16R8, 16R6, or 16R4 to implement this design. In order to obtain X as an output, the logic implementation for the boolean expression for the complement of X should be implemented in the AND-OR array (because of the inverting three-state buffer). This implementation will require six device inputs (A, B, C, D, TXCLK, and the enable connected to logic high) and one output (X). Only four of the eight product terms allocated to the register are required to implement the expression. There is considerably more logic left in the PAL to consolidate any additional functionality. Additional logic may be independent of A, B, C, and D, but does not need to be.

Advantages of Programmable Logic

Programmable logic provides several advantages, many of which have been illustrated through our simple example. Clearly, fewer devices are used: In our example, one 20-pin DIP (Dual In-Line Pin), PLCC (Plastic Leaded Chip Carrier), or LCC (Leadless Chip Carrier) 16R8 device replaced four 14-pin devices. The design utilized only a portion of the device, so one PAL could easily replace as many as ten or more TTL devices. While an individual TTL device may be less expensive than a PAL, the fact that multiple devices may be replaced by one PAL makes the PAL implementation more cost-effective. Additionally, the cost structure for producing many simple TTL logic devices today is inefficient, and the end user absorbs this cost. Today the largest portion of cost associated with TTL logic is packaging and testing. The manufacturing cost of a small PAL is close to that of many TTL logic devices because the amount of die area required to implement the TTL logic functions using today's technology is smaller than the minimum die area required for the pads (I/O buffers), rendering the designs of TTL logic pad-limited. That is, additional logic resources could be added to the die without increased manufacturing cost (because the number of dice per wafer would remain the same). In our example, it would be cost-effective to use a PAL over discrete TTL components even if the rest of the PAL went unused.

16R8 Logic Diagram 20-Pin DIP/PLCC/LCC (28-Pin PLCC) Pinouts

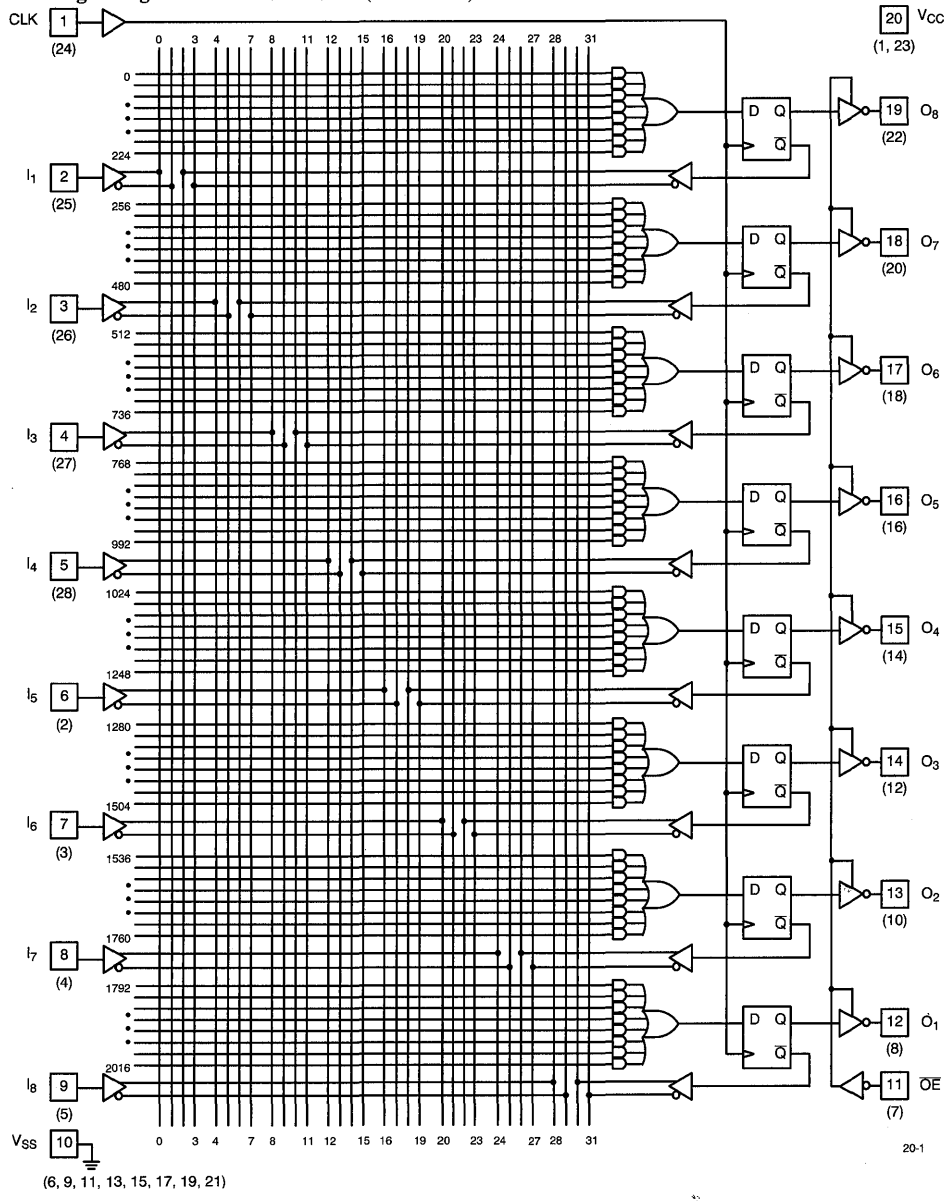


Figure 2-5 PAL 16R8 Device Architecture

Besides saving cost on parts, programmable logic saves valuable board space (*real estate*). The integration provided by the PALs results in lower power requirements, particularly because a PAL implementation requires fewer total device I/Os (a TTL implementation requires several TTL devices each with several inputs and outputs switching; a PAL implementation requires fewer total I/Os and fewer switching outputs, resulting in lower standby and switching currents). This savings can help the power budget or make it possible to use a smaller power supply. Integration increases design reliability because there are fewer dependencies on the interconnection of devices. Savings on debugging time again results in lower costs. Integration also increases performance by reducing the number of I/O delays and levels of logic. A feature of most PLDs is a security fuse that can be used to protect proprietary intellectual property; logic implemented with TTL devices is easily discernible.

Perhaps the greatest advantage of programmable logic is flexibility. With discrete logic components, this flexibility is absent: Once you have developed your board-level design, any design change will necessitate jumpers and/or drilling of boards. If the board is to be produced in high volume, then the board will have to be redesigned resulting in additional NREs and lost time-to-market. The same design with a PAL provides a fallback: The same boards can be used as is with a PAL programmed with the design change. Rather than making and breaking connections on the board, you can modify the connections inside the PAL. This concept will become more clear as we discuss other device architectures.

An additional advantage not to overlook is the ability to use design tools to help implement your design in programmable logic. With programmable logic devices, it is not usually necessary to arrange logic (perform logic optimization, minimization, and multilevel logic functions) based on the device architectures. Instead, you can enter your design equations, description (with hardware description languages such as VHDL), or schematics, and leave the logic synthesis, optimization, and fitting to software. For example, you would not need to determine an optimal implementation for the expression of X, as we did when implementing the design with TTL logic. Instead, you could choose any equivalent equation and leave it to logic minimization software to reduce the equation and implement the logic in a way in which it best fits in the 16R8 or any other PAL.

Simple PLDs

There are several popular industry standard PLDs, such as the 16V8, 20G10, and 20RA10, but we will limit our discussion of small PLDs to the 22V10 in order to limit the number of pages devoted to this chapter. Once you understand the 22V10 architecture described below, you should be able to quickly compare and contrast its architectural features with those of other PLDs by comparing architecture diagrams found in PLD vendor data sheets.

The 22V10

The 22V10 architecture of *Figure 2-6* represented a breakthrough in the architectures of it's time: it included a programmable *macrocell* and variable product term distribution.

Each macrocell (see *Figure 2-5*) may be individually configured (by using programmable configuration bits) according to the configuration table below (*Table 2-2*). This innovative idea allows for the macrocell input (a sum of products) to either pass through to the output buffer or be registered first, as well as to define the polarity of the signal being passed to the output buffer. With polarity control, the complement of an expression may be implemented. The complementary expression may save considerable logic resources over the other, or vice-versa. For example, the complement of a large sum of individual terms ($X = A + B + C + D + E + F$) may be expressed as one

product term ($X = \bar{A}B\bar{C}DEF$). The complement of that signal can then be obtained by inverting it at the output buffer ($X = \overline{(X)}$). The output polarity selection enables software to perform logic optimization (resulting in fewer device resources being used) based on a sum of products for the original equation or its complement. The programmable macrocell also allows feedback: The feedback (or input to the logic array) can be directed from the register or the I/O buffer depending on whether or not the register is bypassed (i.e., whether or not the signal is registered or combinational). An output enable product term can be programmed such that the output is always disabled, allowing the I/O to be used as a device input.

Another innovation in the 22V10 alluded to earlier is variable product term distribution: Recognizing that in typical applications, some logic expressions require more product terms per OR gate than others, the architects of the 22V10 (the *V* is for variable) varied the number of product terms per OR gate from 8 to 16. Looking at *Figure 2-6*, you can see that each macrocell is allocated 8, 10, 12, 14, or 16 product terms. The 22 in 22V10 is for the 22 inputs to the logic array; the 10 is for the 10 outputs.

Why did the architects of the 22V10 not allocate 16 product terms to every macrocell? The most likely reason is that doing so would increase the cost to manufacture the device (the increased die size would result in fewer dice per wafer and, therefore, higher unit costs). Because many applications do not require more product terms, the additional product terms per macrocell would usually go unused, in which case the additional cost would not provide additional functionality.

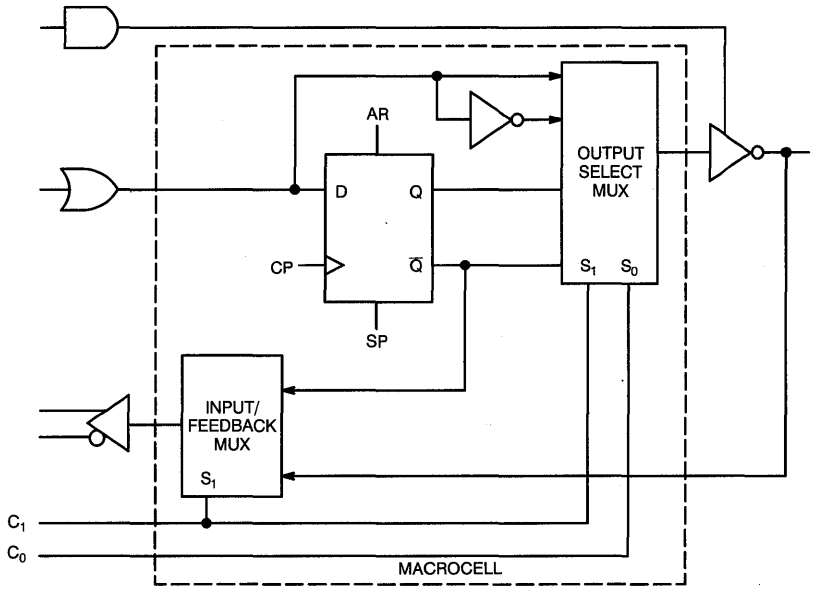


Figure 2-7 22V10 Macrocell

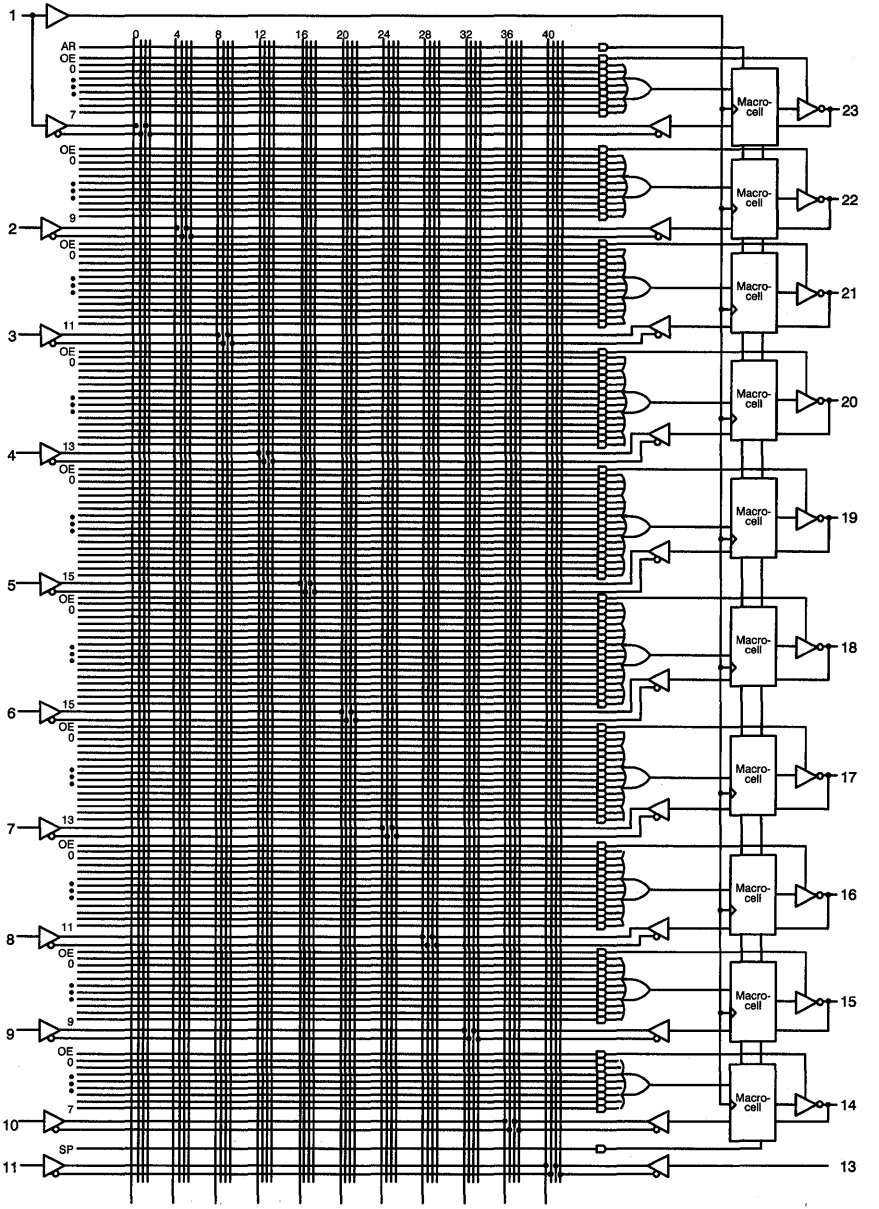


Figure 2-6 22V10 device architecture

Table 2–2 22V10 Macrocell configuration table

C_1	C_0	Description
0	0	Registered/Active Low
0	1	Registered/Active High
1	0	Combinational/Active Low
1	1	Combinational/Active High

Timing Parameters

Although there are additional data sheet timing parameters (such as minimum clock width, input to output enable delay, and asynchronous reset recovery time), the basic timing parameters most frequently referenced are propagation delay (t_{PD}), setup time (t_S), hold time (t_H), clock to output delay (t_{CO}), clock to output delay through the logic array (t_{CO2}), and system clock to system clock time (t_{SCS}), which is used to determine the maximum frequency of operation. The table below shows these basic parameters for a 4-nanosecond 22V10 (a 22V10 with a t_{PD} of 4 ns).

Table 2–3 Sample data sheet parameters for a 22V10

Parameter	Min.	Max.
t_{PD}		4 ns
t_S	2.5 ns	
t_H	0	
t_{CO}		3.5 ns
t_{CO2}		7 ns
t_{SCS}		5.5 ns

The propagation delay is the amount of time it takes for a combinational output to be valid after inputs are asserted at the device pins. The setup time is the amount of time for which the input to a flip-flop must be stable before the flip-flop is clocked. Hold time is the amount of time for which the input to a flip-flop must be held stable after the flip-flop is clocked. The designer must ensure that the setup time and hold time requirements are not violated—in this case, that data must be valid at least 2.5 ns before the clock. Violating the setup and hold time requirements may cause a metastable event: the flip-flop may not transition properly in that it may transition to the wrong value, remain indeterminate, or oscillate for an indeterminate (but statistically predictable) period of time. Clock to output delay is the amount of time after which the clock input is asserted at a device pin that the

output becomes valid at another device pin. These timing parameters are illustrated in the timing diagram of *Figure 2-8*.

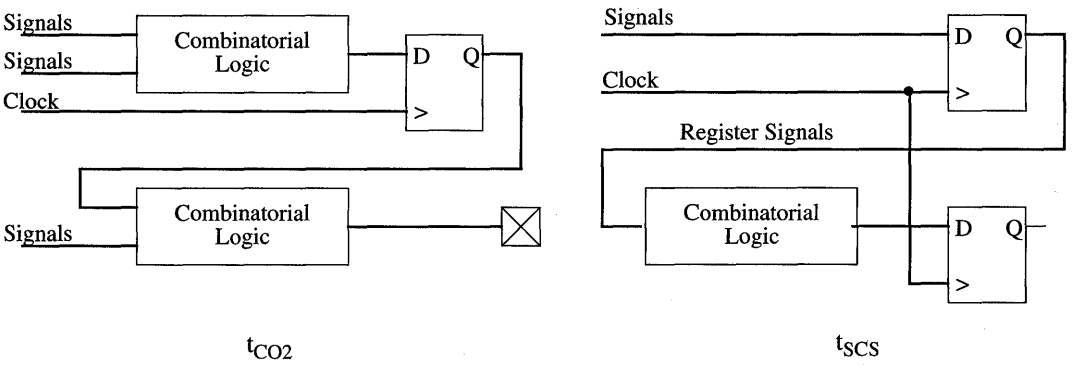


Figure 2-8 Timing parameters illustrated

The parameter t_{CO2} represents the clock to output delay for an output that does not route directly from a register to its associated output pin; rather, it represents the clock to output delay for a signal that is fed back from the register through the logic array, through a macrocell configured in combinational mode, and to a device pin (*Figure 2-8*). This configuration is often used to decode registers (to produce a state machine output, or to generate a terminal count output from the logical AND of several counter bits stored in registers, for examples). The parameter t_{SCS} indicates the minimum clock period if register-to-register operation is required and accounts for the amount of time from when the clock is asserted at the device pin until the output of one register is valid at the input of another (or the same) register, in addition to the setup time for that register. t_{SCS} is used to calculate the maximum frequency of operation, $f_{max} = 1/t_{SCS}$. A sequential circuit is also illustrated in *Figure 2-8*. The second bank of registers may be clocked t_{SCS} after the first bank.

Designing with the 22V10

Here, we will discuss a few design implementation issues (i.e., how resources are used and the resulting timing characteristics). We will not discuss how to use VHDL to create these designs. That is the discussion for the remainder of the text. The design equations that we will present will be determined by using standard design practices rather than software. Our discussion about which device resources must be used for a given design is intended to illustrate the task that a software *fitter* must perform.

Suppose, as an example, that you are asked to design a three-bit synchronous counter for which there are an additional two outputs—one that is asserted when the present count is greater than three, and one that is asserted when the count is equal to six. We can determine the expressions for each of the counter bits and outputs by creating a table with the present-state (PS), next-state (NS), and present-

state outputs, using Karnaugh maps to find the minimal sum of products for each expression, as illustrated in *Figure 2-9*. There are five total outputs (*a*, *b*, *c*, *x*, and *y*). If all macrocells are configured for active high logic, then the complement of the Q-output for each flip-flop is multiplexed to the output buffers. This allows the positive-logic implementations for A, B, and C. Signal *A* is the D-input to the flip-flop associated with output signal *a*. Likewise for *B* and *C*. *Figure 2-9* indicates that signal *A* requires three product terms, *B* two, and *C* one. The output *x* does not require any additional logic because it is equivalent to output *a*. (In fact, one output pin can be used for both signals *a* and *x*.) Signal *y* requires one product term. Because the product term requirement associated with each of the outputs is less than eight, the output signals can be placed on *any* of the I/O pins. A schematic of this design implementation is shown in *Figure 2-9*.

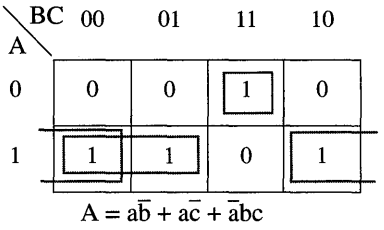
Present-State/Next-State Table

PS			NS				
a	b	c	A	B	C	x	y
0	0	0	0	0	1	0	0
0	0	1	0	1	0	0	0
0	1	0	0	1	1	0	0
0	1	1	1	0	0	0	0
1	0	0	1	0	1	1	0
1	0	1	1	1	0	1	0
1	1	0	1	1	1	1	1
1	1	1	0	0	0	1	0

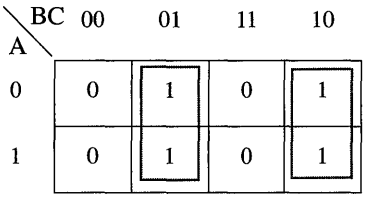
By inspection,

$x = a$

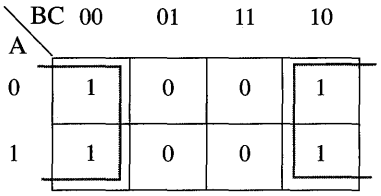
$y = ab\bar{c}$



$A = a\bar{b} + a\bar{c} + \bar{a}bc$



$B = b \oplus c = \bar{b}c + b\bar{c}$



$C = \bar{c}$

Figure 2-9 Expressions for a 3-bit counter

There are not setup or hold time requirements for this design because there are not any inputs that are registered. Outputs *a, b, c,* and *x* are valid at the output pins t_{CO}, 3.5 ns, after the clock input transitions. Output *y* is valid at its output pin t_{CO2}, 7 ns, after the clock input transitions. More time is required for *y* to be valid because the outputs of the counter must be decoded to produce *y*. The decoding is possible by feeding back the outputs of the counters into the logic array and using a product term associated with another macrocell. This decoding causes an additional delay of 4 ns over t_{CO}. The maximum clock frequency at which this circuit may be clocked (the frequency at which the registers may be clocked) is usually dictated by the amount of time it takes for the output of one register to propagate to the input of another register in addition to the setup time requirement for the second register. This frequency is the reciprocal of t_{SCS} (5.5 ns), or 180 MHz. In this case, however, the design cannot be clocked at this frequency because the output *y* takes 7 ns to propagate to an output pin. Clocking the circuit at 180 MHz would result in the output *y* never being valid. Theoretically, this circuit could be clocked at the rate of the reciprocal of 7 ns (143 MHz) and you would find a valid output at the device pin for *y*. However, unless there is a clock to output delay minimum time, the output is guaranteed to be valid only for an instant in time. In order to sample the output, you will need to include any trace delay and setup time of the sampling device in determining the maximum clock frequency for the system.

Using More Than 16 Product Terms

Although the largest number of product terms associated with any macrocell in the 22V10 is 16, you can implement expressions that require more than 16 product terms by using 2 passes through the logic array. For example, to sum 20 product terms, 2 macrocells can sum 10 product terms each and a third macrocell can sum the macrocell feedbacks of the first 2 macrocells. This third macrocell requires two additional product terms for positive output polarity, or one product term for negative output polarity. (If *a* and *b* are the outputs of the first 2 macrocells, and *y* is the output to be propagated to the device pin, then the expression for $\bar{y}$, $\bar{y} = \bar{a} \cdot \bar{b}$, can be implemented with the third macrocell using one product term. The complement of $\bar{y}$ can be obtained with the macrocell polarity control.) Alternatively, you could use one of the center macrocells of the 22V10 to sum 16 product terms, using another macrocell to sum the feedback of the first macrocell and the remaining 4 product terms (for a total of five product terms for this second macrocell). Either way, two passes through the logic array are required, and this will increase the propagation delay from input pins to output (if the function is combinational) or the setup time (if the function is to be registered). If the data sheet does not provide a specification for a propagation delay from input, through the logic array twice, and to an output pin, then the sum of $t_{PD} + t_{PD}$ is a safe worst-case estimate. Likewise, if a setup time for two passes through the logic array is not specified, then $t_{PD} + t_S$ provides a safe estimate.

Terminology

The 22V10 architecture is shown in block diagram form in *Figure 2-11* for the purpose of assigning terms to the different blocks or features of the 22V10 architecture. We can then use these terms in our discussion of CPLDs.

The term *logic array inputs*, or *inputs* for brevity, is used to indicate all of the signals that are inputs to the logic array. These inputs include dedicated device inputs, feedbacks from macrocells, and inputs from I/Os that are configured as device inputs. The term *product term array* will be used to describe the programmable AND gates (product terms). The *product term allocation scheme* is the

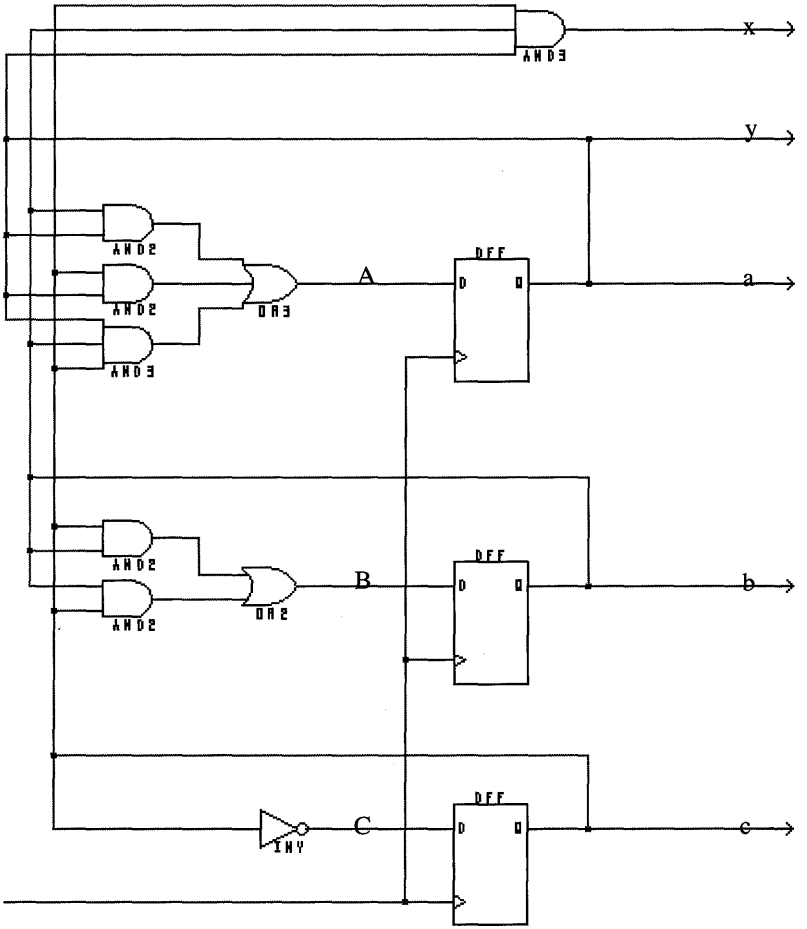


Figure 2-10 Schematic of a 3-bit counter

mechanism for distributing product terms to the macrocells. Macrocells typically contain a register and combinational path with polarity control and one or more feedback paths. *I/O cells* is a term used to describe the structure of the I/O buffers and flexibility of the output enable controls.

What is a CPLD?

Complex PLDs (CPLDs) extend the concept of the PLD to a higher level of integration to improve system performance, use less board space, improve reliability, and reduce cost. Instead of making the PLD larger with more inputs, product terms, and macrocells, a CPLD contains multiple logic blocks, each similar to a small PLD like the 22V10 that communicate with each other using signals routed via a programmable interconnect (*Figure 2-12*). This architectural arrangement makes more efficient

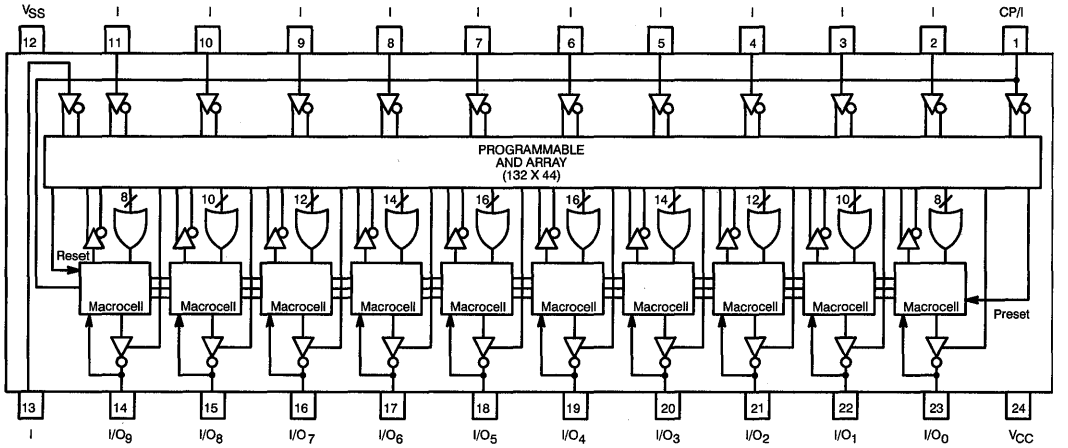


Figure 2-11 Block diagram of the 22V10

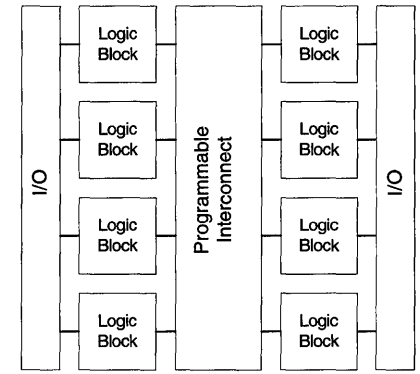
use (than one large PLD) of the available silicon die area, leading to better performance and reduced cost.

In this section, we present an overview of CPLDs: We examine their makeup and indicate the features of some specific families of popular CPLDs (and point out their differences). This presentation does not include a comprehensive examination of all of the CPLDs on the market today or an in-depth examination of all the features that are present in the architectures that we will discuss. To find out more information about the features of the architectures discussed in this text, and to compare them, we strongly recommend that you obtain the data sheets for these devices.

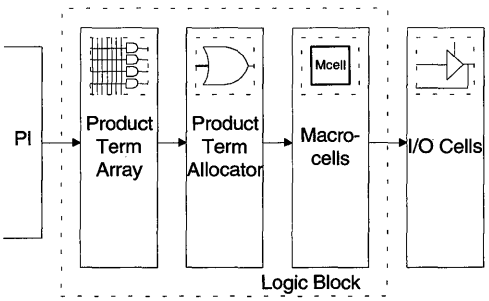
Programmable Interconnects

The programmable interconnect (PI) routes signals from I/Os to logic block inputs or from logic block outputs (macrocell outputs) to the inputs of the same or other logic blocks. (Some logic blocks have local feedback so that macrocell outputs used in the same logic block do not route through the global programmable interconnect; there are advantages and disadvantages—to be discussed later—of this approach.) As with a PLD such as the 22V10 which has a fixed number of logic array inputs, each logic block has a fixed number of logic block inputs.

Most CPLDs use one of two implementations for the programmable interconnect: *array-based interconnect* or *multiplexer-based interconnect.* Array based interconnect allows any signal in the PI to route to any logic block (see *Figure 2-13*). Each term in the PI is assigned as an input to a given logic block, so there is one PI term for each input to a logic block. An output of a logic block can



(a)



(b)

Figure 2-12 (a) Generic CPLD architecture (b) Generic logic block

connect to one of the PI terms through a memory element. This interconnect scheme is highly flexible in that it is fully *routable* (see glossary), but it may be at the expense of performance, power, and die area.

With multiplexer-based interconnect (*Figure 2-14*), signals in the PI are connected to the inputs of a number of multiplexers for each logic block. There is one multiplexer for each input to a logic block. The selection lines of these multiplexers are programmed to allow one input for each multiplexer to propagate into a logic block. Routability is increased by using wide multiplexers, allowing each signal in the PI to connect to the input of several multiplexers for each logic block. Wider multiplexers, however, increase the die area (and potentially reduce performance).

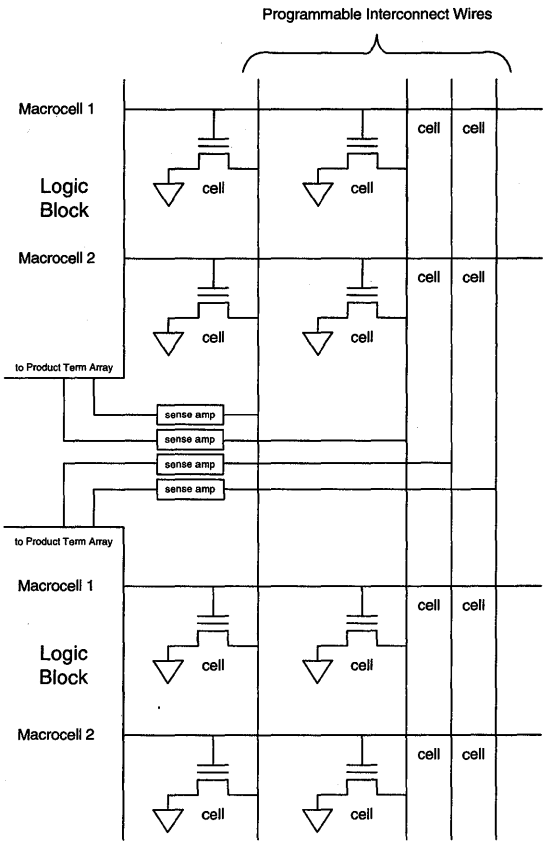


Figure 2-13 Array-based interconnect

Logic Blocks

A logic block is similar to a PLD such as the 22V10: Each logic block has a product term array, product term allocation scheme, macrocells and I/O cells. The size (used here as a measure of capacity—how much logic can be implemented) of a logic block is typically expressed in terms of the number of macrocells, but the number of inputs to the logic block, the number of product terms, and the product term allocation scheme are important as well. Logic blocks typically range in size from 4 to 20 macrocells. Sixteen or more macrocells permit 16-bit functions to be implemented in a single logic block, provided that enough inputs from the PI to the logic block exist. For example, a 16-bit free-running counter can fit in a logic block with 16 macrocells and 15 inputs (one for each bit less the most significant bit), assuming that the logic block outputs propagate through the PI and that T-type flip-flops are used to implement the counter. A 16-bit loadable counter with asynchronous reset requires a logic block with 16 macrocells and 33 inputs to the logic block (one for each counter

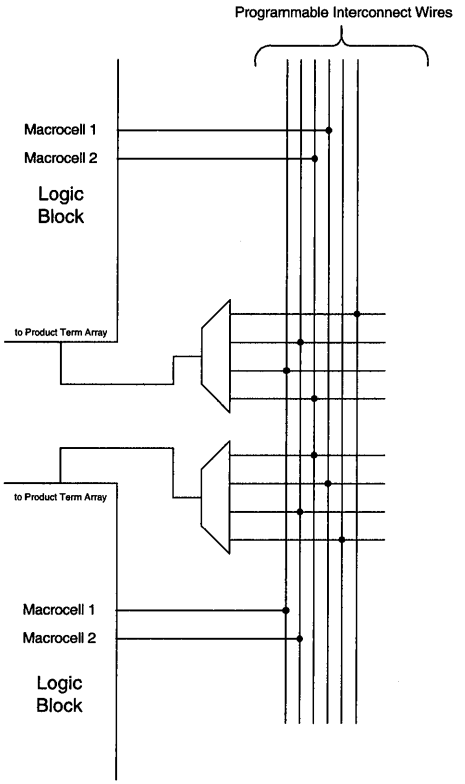


Figure 2-14 Multiplexer-based interconnect

bit, plus 16 for the load inputs, plus one for the asynchronous reset). This function could fit in a CPLD that has fewer macrocells or inputs per logic block only with multiple passes through the logic array, resulting in slower performance. Logic blocks with few inputs also tend to use logic block resources inefficiently. This is because if one expression using one macrocell requires all the available inputs to a logic block, then other expressions cannot be placed in other macrocells in that logic block unless these expressions use a subset of the signals required for the first expression.

Product Term Arrays

There is little difference between the product term arrays of the different CPLDs. Of course, the size of the array is important because it identifies the average number of product terms per macrocell and the maximum number of product terms per logic block.

Product Term Allocation

Different CPLD vendors have approached product term allocation with different schemes. The MAX family ("family" meaning several devices of the same architecture) of CPLDs, jointly developed by

the Altera Corporation (the market leader in CPLDs) and Cypress Semiconductor, was the first family of CPLDs on the market. (Altera named this family the MAX5000 family, Cypress named it the MAX340 family). Rather than using the variable product term distribution scheme of the 22V10 (which allocated a fixed but varied number of product terms—8, 10, 12, 14, or 16—per macrocell), the MAX family allocated four product terms per macrocell while allowing several *expander product terms* to be allocated individually to any macrocell (or multiple macrocells) of choice (*Figure 2-15*). With expander product terms, the additional product terms need be allocated to only those macrocells that can make use of them. The concept that a product term can be used by a macrocell of choice is termed *product term steering*, and the concept that the same product term may be used by multiple macrocells is termed *product term sharing*. An additional delay is incurred for signals that make use of the expander product terms because the output of the expander product term must pass through the logic array before propagating to a macrocell.

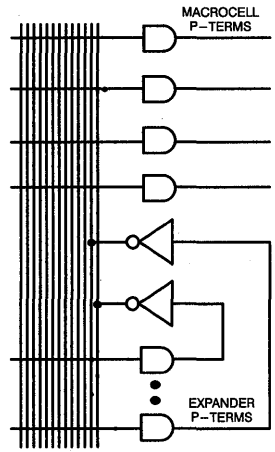


Figure 2-15 Product term allocation in the MAX340 and MAX5000 families of devices

The MACH 3 and 4 families offered by Advanced Micro Devices allow product terms to be steered (in groups of four product terms) and used in another macrocell without an additional delay (*Figure 2-16*). For each group of macrocells that must be steered, one macrocell is left unusable. This product term allocation scheme does not provide product term sharing.

The MAX7000 family offered by the Altera Corporation improved the product term allocation scheme of the MAX5000 family (*Figure 2-17*): In addition to the expander product terms (which may be individually steered or shared), the new architecture also includes a product term steering mechanism in which five product terms may be steered to a neighboring macrocell (which in turn can be steered to a neighboring macrocell). Steering in this way adds a significantly smaller incremental delay because the signal does not propagate through the product term array (as with an expander product term). This steering does prevent the product terms from being shared (unlike with the expander product terms).

The FLASH370™ family offered by Cypress Semiconductor provides yet a different innovation (*Figure 2-18*): Each macrocell is allocated from 0 to 16 product terms, depending on the needs of the

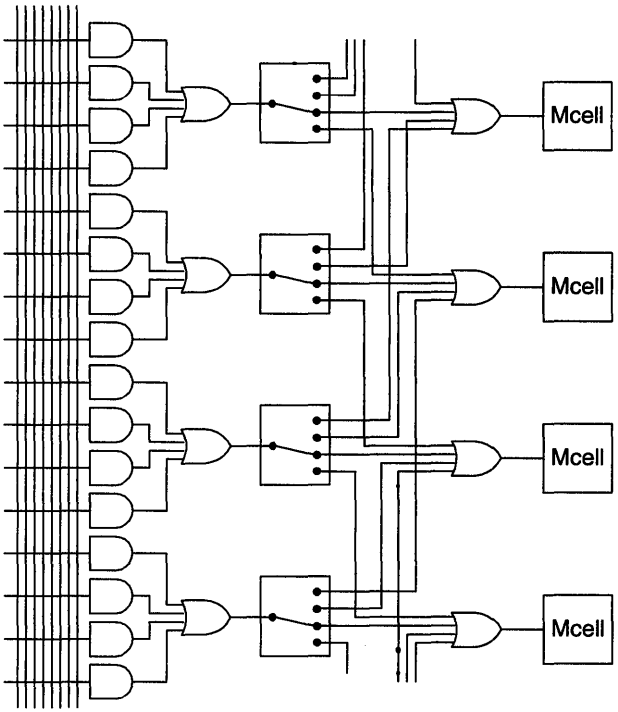


Figure 2-16 MACH3 product term allocation scheme

logic expression implemented for a given macrocell (however, adjacent macrocells cannot each be allocated 16 *unique* product terms). Each product term may be individually steered (i.e., a product term can be allocated to a particular macrocell). This steering scheme does not render a macrocell unusable or cause an incremental delay to be incurred. Most product terms (except for some of those for the first and last macrocell in a logic block) can also be shared with up to four neighboring macrocells without additional delay.

The product term allocation schemes of these architectures provide flexibility to the designer. More importantly, these schemes provide flexibility to software algorithms that will ultimately choose how to use logic resources. The design engineer should understand how logic resources may be used and the trade-offs between architectures, but engineering design automation tools (software) should automatically select the optimal implementation, leaving the designer to provide innovation to his or her system design.

Macrocells

CPLD macrocells offer more configurability than found in 22V10 macrocells. In addition, many CPLDs have I/O macrocells, input macrocells, and buried macrocells. A 22V10 has only what is considered an I/O macrocell (a macrocell associated with an I/O). An input macrocell, as you may

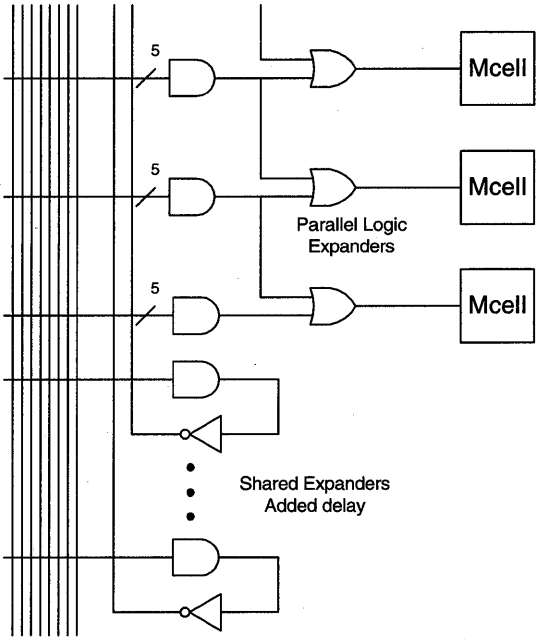


Figure 2-17 MAX7000 product term allocation

have deduced, is associated with an input pin. A buried macrocell is usually similar to an I/O macrocell except that its output cannot propagate directly to an I/O.

I/O and Buried Macrocells

Figure 2-19 illustrates the I/O macrocell of the MAX340 family of devices, which provides more configurability than the 22V10 macrocell. There are several inputs to this macrocell. The sum of products input is used as one input to the XOR gate. The other input to the XOR gate is an individual product term. The XOR gate can be used in arithmetic expressions (comparators and adders make good use of an XOR gate) or to complement the sum of products expression. If the array is programmed such that the individual product term is always deasserted, then the output of the XOR gate is the same as the sum of products expression on the other input of the XOR gate. If the individual product term is always asserted, then the output of the XOR gate is the complement of the sum of products, thereby allowing the expression (complemented or uncomplemented) that uses the fewest number of product terms to be implemented in the product term array. That is, the XOR gate serves the same function as the output polarity multiplexer found in the 22V10 macrocell, but it can also be used to implement logic, especially arithmetic logic, or to configure a flip-flop for T-, JK-, or SR-type operation. The preset and clear are individual NAND terms, and the clock can be either the system clock or a gated (product term) clock. The system clock provides the best performance, and the product term clock provides flexibility (product term clocks should be used carefully because they can easily cause timing problems; for example, race conditions can cause false clocks). The output of the macrocell can be configured as registered or combinational. Feedback is from either the

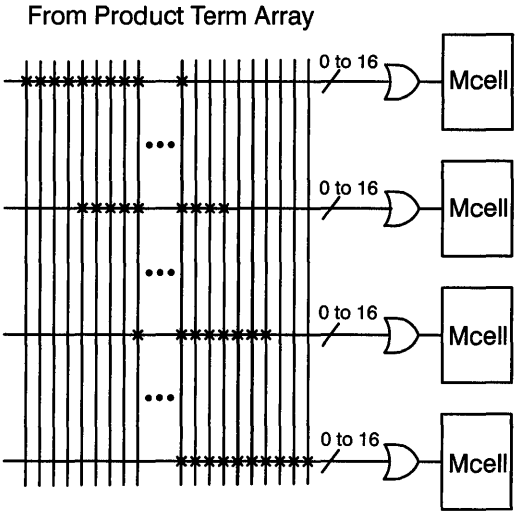


Figure 2-18 FLASH370 product term allocation

combinational or registered signal, depending on the macrocell configuration. This architecture provides both local feedback (i.e., the feedback does not use the PI and is not available to other logic blocks) and global feedback through the PI. The advantage of local feedback is quicker propagation to other macrocells in the logic block. The disadvantage of having both local and global is a more complicated timing model and redundant resources. Buried macrocells for this family are identical to the I/O macrocells except that the outputs of the buried macrocells do not feed the I/O cells.

Figure 2-20 represents the I/O and buried macrocells of the FLASH370 architecture. The macrocell input can be programmed for 0 to 16 product terms. This input can be registered, latched, or passed through as a combinational signal. If configured as a register, the register can be a D-type or T-type register. The clock for the register/latch can be one of four clocks available to a logic block (polarity of a clock is determined on a logic-block-by-logic-block basis). The output polarity control permits the optimization of product term utilization based on either the true or complement implementation of an expression in sum-of-products form. The macrocell has a feedback separate from the I/O cell to permit the I/O cell to be configured as a dedicated device input while still allowing the I/O macrocell to be used as an internal macrocell, in either a registered or combinational mode (contrast with the functionality of a 22V10 macrocell). The buried macrocells are nearly the same as the I/O macrocells, except that the output does not feed an I/O cell. Additionally, a buried macrocell can be configured to register the input associated with a neighboring I/O cell.

Input Macrocells

Input macrocells, such as that shown in *Figure 2-21*, are used to provide additional inputs other than those associated with I/O macrocells. The figure shows that for this architecture these inputs can be

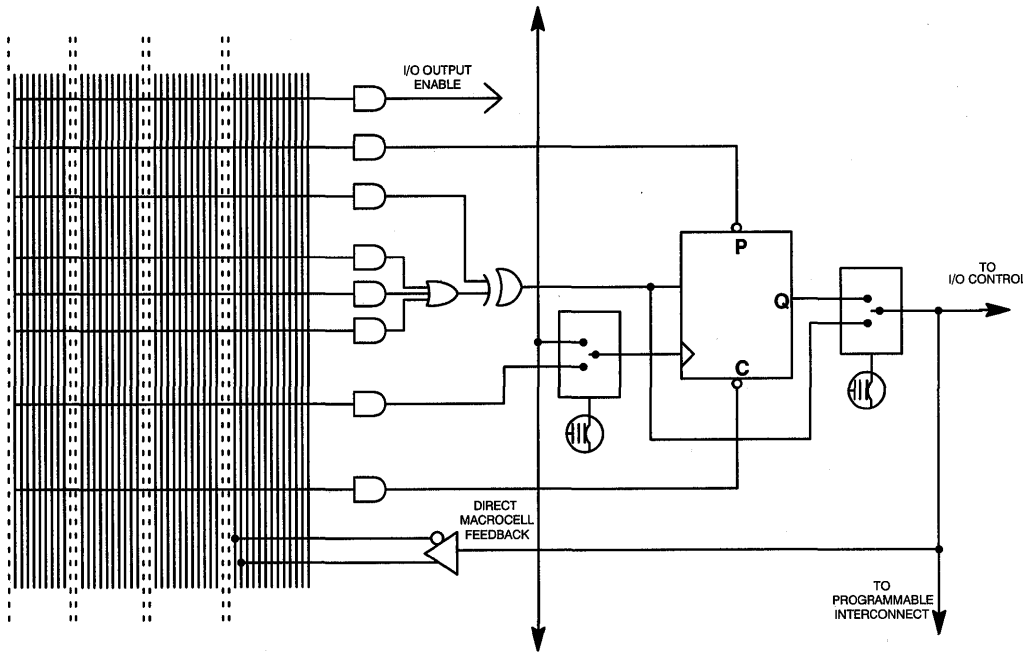


Figure 2-19 MAX340 macrocell

used as clocks or inputs to the PI or both. The inputs can be combinational, latched, registered, or twice registered. (A signal that is asynchronous to the system clock is sometimes twice registered to increase the MTBF—mean time between failure—for a metastable event.) If registered or latched, the register/latch can be clocked by another clock input (but not itself—this would surely lead to metastable events if delays were not carefully controlled!).

I/O Cells

Most I/O cells are used only to drive a signal off of the device, depending on the state of the output enable, and to provide a data path for incoming signals, as shown on the right-hand side of *Figure 2-20*. With some architectures like that of the MACH 3 family, however, I/O cells contain switch matrices or output routing pools in which a one-to-one connection is made between an I/O macrocell output and an I/O (see *Figure 2-22*). The advantage to this scheme is flexibility in determining where logic can be placed in a logic block in relation to where the I/O cell is located. The disadvantage is an incremental delay associated with the programmable routing structure and increased die size.

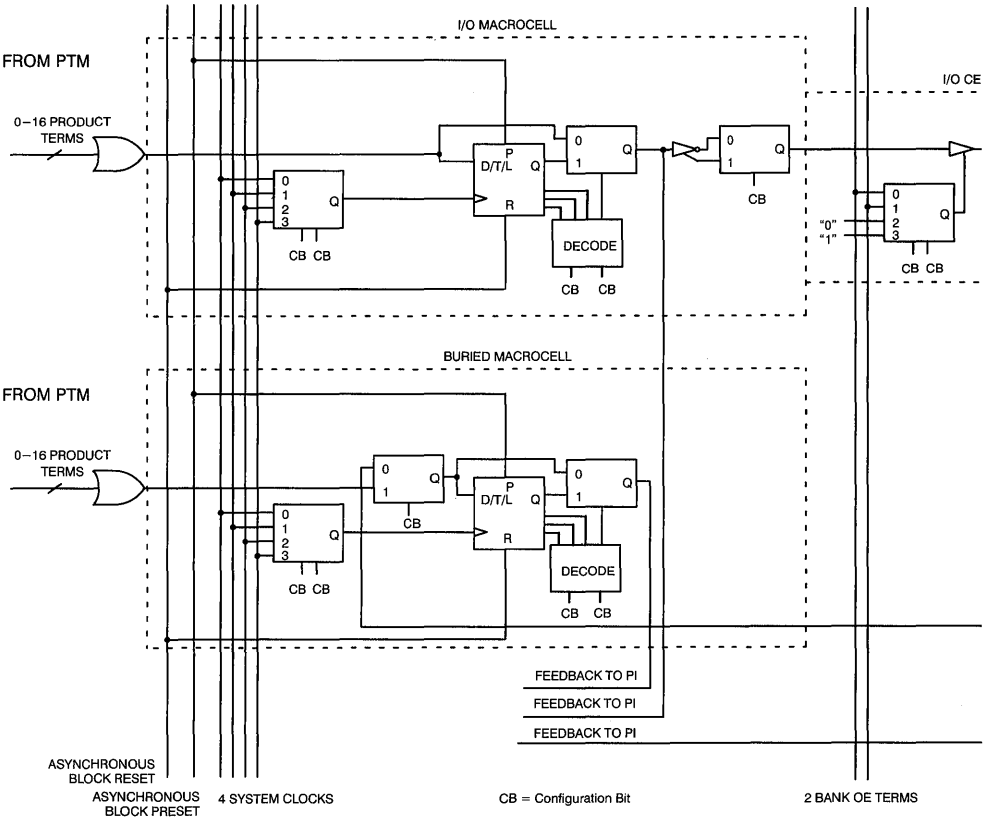


Figure 2-20 FLASH370 I/O and buried macrocells

Timing Parameters

The timing parameters of most interest are the same for those of a 22V10: propagation delay, setup, clock-to-output, and register-to-register times (illustrated in *Figure 2-23*).

Post-design implementation timing information is generally more predictable with CPLDs than with FPGAs (for reasons that will become clear later). For some designers, this is an advantage of CPLDs over FPGAs: that prior to beginning a design, the performance of that design can be estimated with good precision. These designers prefer to be able to select a device before doing any design work, and have the confidence that in the end the device will perform as predicted.

With a 22V10, the performance of a design can be estimated with good precision prior to implementing a design, provided that you have an accurate understanding of how many passes through the product term array that any logic expression will require. Any expression of more than 16 product terms will require additional passes. An expression of up to 112 product terms (16 + 16 + 14 + 14 + 12 + 12 + 10 + 10 +8 = 112) can be implemented in 2 passes. Certainly, it is easier to

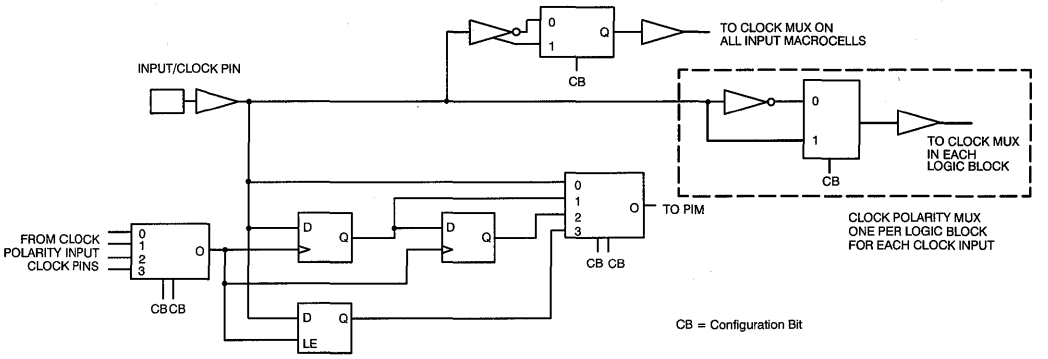


Figure 2-21 FLASH370 input macrocell

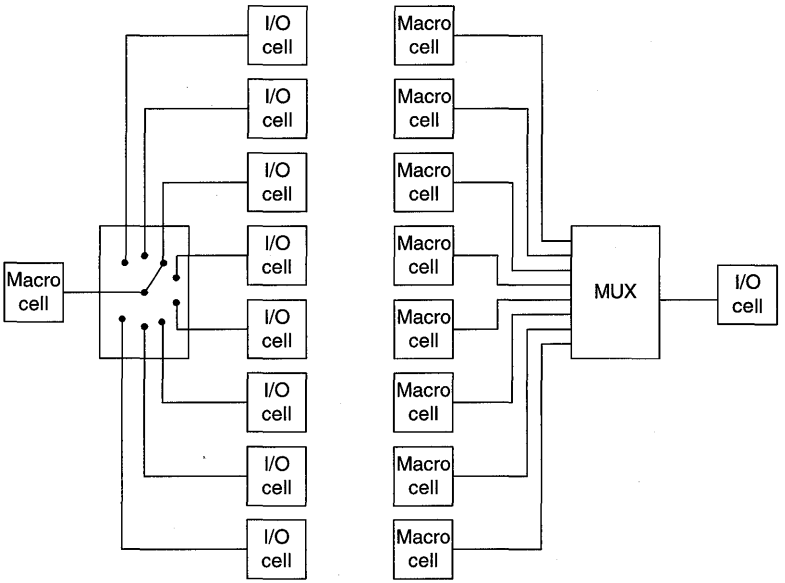


Figure 2-22 I/O cell with switch matrix

predict with some certainty, based on your knowledge of the design, whether a given expression will require a multiple of 16 product terms versus a multiple of one product term. For example, if you believe that the largest expression required for a state machine that you are designing may take 14 product terms, then you could safely estimate that it would take no more than a maximum of two passes (an additional pass was added to guard-band the estimate). If the number of passes for an expression can be accurately predicted, then the timing parameters can be predicted with precision.

Although estimating performance in some CPLDs is as easy as it is for a 22V10, it is not for many others. Compare the timing models for the MAX340 and FLASH370. Performance in the MAX340 is highly dependent upon resource utilization. It may be difficult to determine the achievable performance prior to implementing the design in a CPLD with a timing model similar to that of the MAX340; although, it is more predictable than with an FPGA.

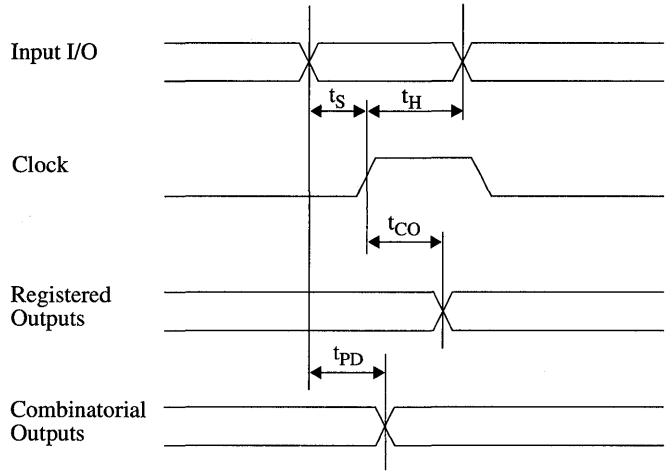


Figure 2-23 Key Timing Parameters

The CY7C371-143 CPLD has a simple timing model like the 22V10 and will be used as the target architecture for the synthesis and fitting of many design examples. Its timing parameters are listed in *Table 2–4*.

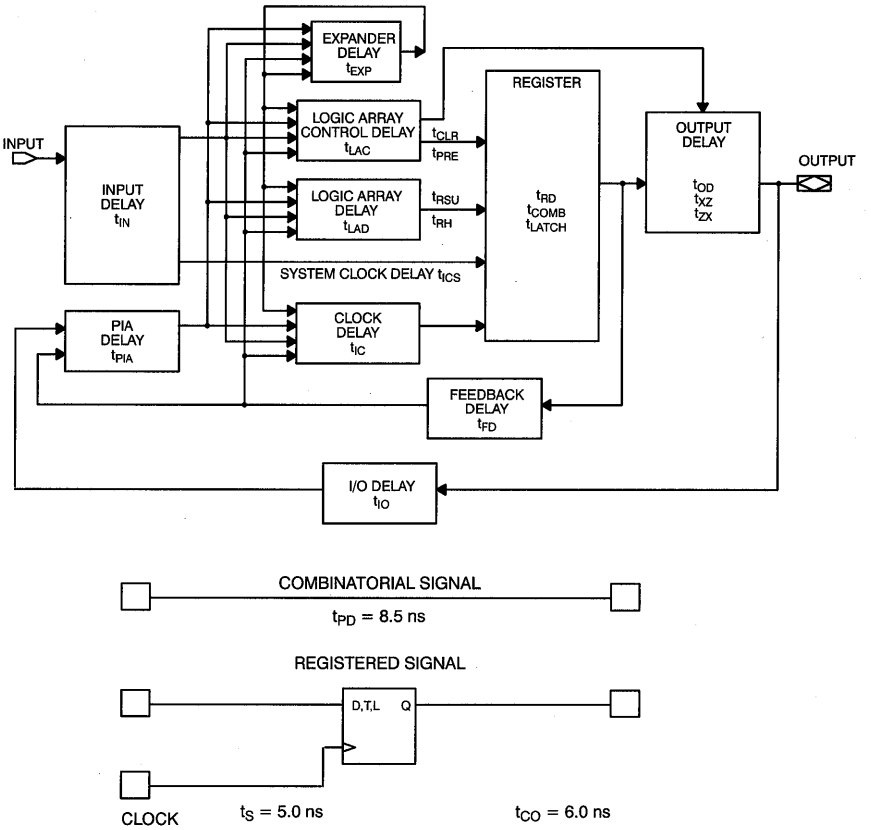


Figure 2-24 Timing Models for the MAX340 and FLASH370

Table 2–4

Parameters	Min.	Max.
t_{PD}	-	8.5 ns
t_S	5 ns	-
t_H	0 ns	-
t_{CO}	-	6 ns
t_{CO2}	-	12 ns
t_{SCS}	7 ns	-

Other Features

Besides logic resources, routing and product term allocation schemes, macrocell configurations, and timing models, a few other features set CPLDs apart from one another. These features include in-system programmability (ISP), in-circuit reconfigurability (ICR), 5V/3.3V operation, test access port and boundary scan capability that is IEEE 1149.1 (JTAG, or Joint Test Action Group) compliant, and input and output buffers that are PCI (Peripheral Component Interconnect) compliant. Devices are offered in a variety of packages.

In-system programmability is the ability to program a device while it is on the board. This mainstreams the manufacturing flow. Time may be saved because parts do not have to be handled for programming, and inventories of both programmed and unprogrammed parts do not need to be kept.

In-circuit reconfigurability is the ability to reprogram a device while it is in a circuit. In-circuit reconfigurability can be used for field upgrades or even to alter the functionality of the device during operation.

The JTAG specification defines a method for testing the integrity of a device and its connections to other devices on the board through a test access port and boundary scan. This capability enables stimuli to be applied to a particular device to verify its functionality, as well as data to be shifted through the boundary to verify device interconnections. JTAG may be used for testing and quality assurance or debugging. The large number of vectors that need to be clocked through to verify functionality has prompted the development of devices with BIST (built-in self-test). Devices with BIST can be placed in self-test mode. A device placed in this mode isolates the I/Os from other devices. The device generates pseudorandom test-vectors as stimuli, compares internal outputs against expected results, and indicates success or failure.

The PCI specification includes an electrical components checklist with requirements such as AC switching current. Because PCI has caught hold "by storm," vendors of programmable logic have rushed to characterize their devices and declare them PCI compliant.

What is an FPGA?

A field programmable gate array (FPGA) architecture is an array of *logic cells* that communicate with each other and I/O via *routing channels* (*Figure 2-25*). Like a semi-custom gate array, which consists of a sea of transistors, an FPGA consists of a sea of logic cells. In a gate array, however, routing is custom, without programmable elements. In an FPGA, existing wire resources that run in horizontal and vertical columns (*routing channels*) are connected together via programmable elements, with logic cells, and I/O. Logic cells have less functionality than the combined product terms and macrocells of CPLDs, but large functions can be created by cascading logic cells. Logic cell and routing architectures differ from one vendor to another.

As with all commercial products, FPGAs are developed to meet market needs. A market-driven vendor will survey not only what FPGAs are currently used for (predominately data-path, I/O-intensive, and register-intensive applications) but also what system designers would like to use FPGAs for or what they believe that they will use FPGAs for in the near future (high performance applications such as a PCI bus target interface operating at 33 or 66 MHz, a DRAM controller with a 3 ns setup time, a DMA controller with a 6 ns clock-to-output delay, and computer network applications involving Ethernet and ATM, among others). Among the top few m

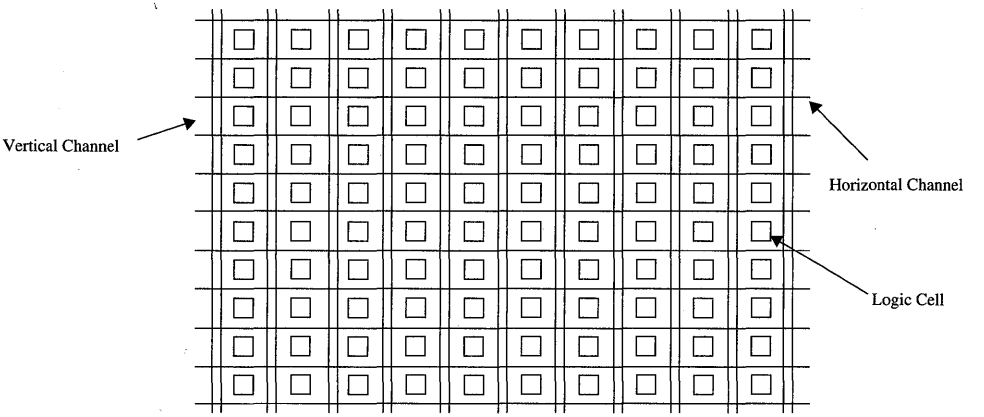


Figure 2-25 FPGA Architectures

FPGAs currently attempt to serve are (1) performance—the ability for real system designs to operate at higher and higher frequencies; (2) density and capacity—the ability to increase integration, to place more and more in a chip (*system in a chip*), as well as use all available gates within the FPGA, thereby providing a cost-effective solution; (3) ease of use—the ability for system designers to bring their products to market quickly, leveraging off of the availability of easy-to-use software tools for logic synthesis as well as place and route, in addition to architectures that enable late design changes that affect logic, routing, and I/O resources without significantly adversely affecting timing; and (4) in-system reprogrammability and in-circuit reconfigurability—the abilities to program or reprogram a device while it is in system, mainstreaming manufacturing and inventories as well as allowing for field upgrades and user configurability.

After completing the list of market needs, a vendor must choose or develop a technology that it believes will satisfy the most important market needs. As with much of product development, there are trade-offs. Presently, there are two technologies of choice for use in developing FPGAs—SRAM and antifuse—each of which can satisfy a subset of the market needs. SRAM technology is presently used by Altera, AT & T, Atmel, and Xilinx (the market leader in the sale of FPGAs). Antifuse technology is presently used by Actel, Cypress (also a market leader in the sale of high-performance SRAMs), and QuickLogic. Xilinx has recently announced the availability of an antifuse-based FPGA. We'll briefly explain each technology, the impact on device architectures, and summarize which market needs are best addressed with each technology.

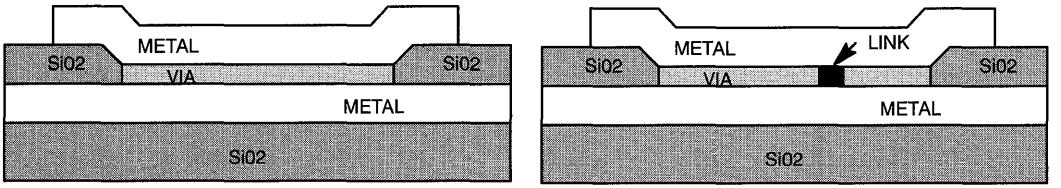
Technologies and Architecture Trade-offs

Once a technology has been selected, that technology influences choices of routing architectures. The routing architecture, in turn, influences the design of the logic cells.

Routing

The choice of technology has a significant impact on the routing architecture for one very simple reason: The physical dimensions of an SRAM cell are an order of magnitude larger than those of an antifuse element.

Amorphous-Silicon Antifuse. An amorphous-silicon antifuse can be deposited in the space (via) between two layers of metal, as shown in *Figure 2-26*. In a semi-custom gate array, the top and bottom layers of metal make direct contact through a metal interconnect via. In an amorphous-silicon-based FPGA, the two layers of metal are separated by amorphous silicon, which provides electrical insulation. A programming pulse of 10V to 12V and of necessary duration can be applied across the via, creating a bidirectional conductive link (with a resistance of about 50 ohms) connecting the top and bottom metal layers (also shown in *Figure 2-26*). Because the size of an amorphous-silicon antifuse element is the same as that of a standard metal interconnect via, the programmable elements can be placed very densely (limited only by the minimum dimensions of the metal-line pitch), as shown in the microphotograph of *Figure 2-27*. Once programmed, an antifuse element cannot be erased or reprogrammed.



(a) (b)

Figure 2-26 (a) An unprogrammed antifuse programmable element and (b) a programmed antifuse programmable element

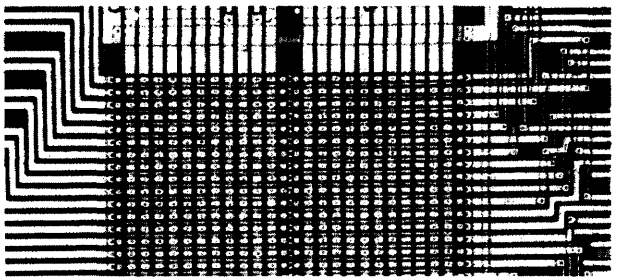


Figure 2-27 An array of amorphous-silicon antifuse elements

To program an antifuse element, a voltage differential must be applied across the antifuse element. Each antifuse element must be isolated by using pass transistors in order not to inadvertently

program other elements. These programming transistors, as well as the associated logic for addressing the antifuse locations, constitute the programming circuitry overhead.

Oxide-Nitride-Oxide (ONO) Antifuse. The Actel FPGA products make use of an ONO antifuse. This antifuse consists of three layers (*Figure 2-28*): the top layer, which is electrically connected to one layer of metal, is a conductor made of polysilicon; the middle layer has an oxide-nitride-oxide chemical composition and is an insulator; the bottom layer is a conductive layer of negatively doped diffusion. Unprogrammed, the ONO antifuse insulates the top layer of metal from the bottom layer. The fuse is programmed similar to that of an amorphous-silicon antifuse: A programming voltage is applied, allowing the insulator to be penetrated by the top and bottom layers and establishing an electrical connection of fairly low resistance (about 300 ohms).

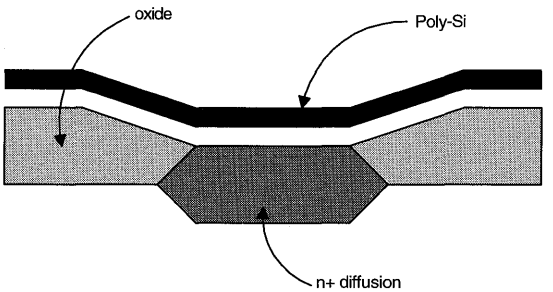


Figure 2-28 ONO Antifuse Element

Because antifuse elements can be placed very densely, FPGAs making use of this technology typically have flexible routing architectures, allowing the electrical connection of wires at nearly every intersection. *Figure 2-29* illustrates a routing architecture of an antifuse-based FPGA. The open boxes at the intersections of wires indicate a programmable antifuse. The inputs and outputs of the logic cell can connect to any vertical wire (except the clock structure, which in this figure is shown to connect only to the clock, set, and reset of the flip-flop). Wires within a vertical channel may connect with wires in a horizontal channel where the wires intersect. Some wires (segmented wires) extend the length of only one logic cell. These wires may connect to the segmented wires of the logic cells above and below but on the same layer of metal through a programmed antifuse. A routing architecture made up of entirely segmented wires would provide the greatest routing flexibility. However, using segmented wires for long routes would require several antifuse elements to be programmed, each adding an additional resistance (about 50 ohms for amorphous-silicon antifuse, about 300 ohms for ONO antifuse) to the signal path. Greater resistances will result in slower performance. Therefore, other wires extend further distances (two logic cells, four logic cells, or the entire length or width of the array), optimized for either local or global routing.

SRAM. Static RAM cells may be used to control the state of pass transistors, which can establish connections between horizontal and vertical wires (*Figure 2-30* shows six pass transistors allowing any combination of connections of the four wires). The source-to-drain resistance is about 1000 ohms. SRAM cells can also be used to drive the select inputs of multiplexers that are used to choose from one of several signals to route on a given wire source. A memory cell consists of five transistors (*Figure 2-31*): two each for the two inverters making up the latch and one for addressing (used to

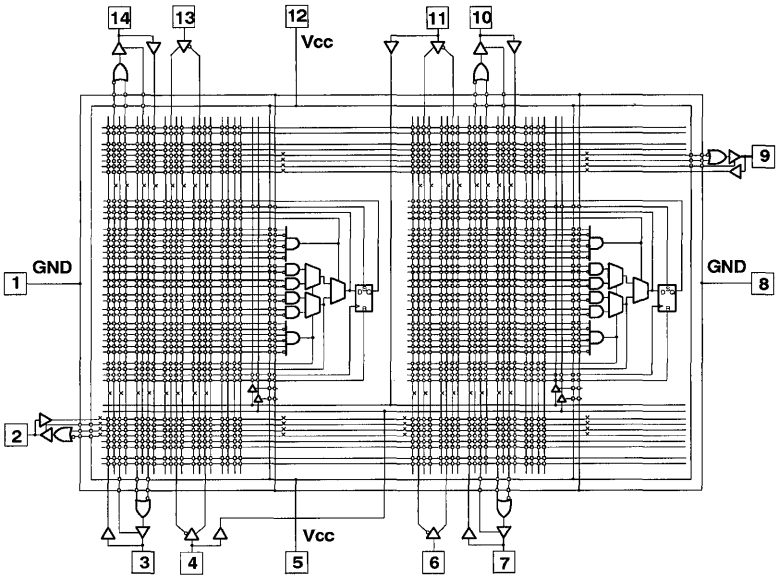


Figure 2-29 The Cypress pASIC380 Routing Structure

select the memory cell for programming). An SRAM cell is reprogrammable, unlike antifuse elements, which are physically altered when programmed. SRAM cells are volatile, however, meaning that the states of the memory cells are lost when power is not applied. SRAM-based FPGAs must be programmed (usually from a serial EPROM) each time the circuit is powered up. As for antifuse elements, the programming circuitry for SRAM elements must include the addressing and data registers.

The size of an SRAM cell and the associated pass transistor as compared to an antifuse element and the associated programming transistor is considerably larger. These programming elements cannot be placed as densely as the antifuse elements; SRAM FPGAs therefore do not have routing architectures for which there is a programmable element at nearly every intersection (doing so would increase the metal spacing and overall die size, limiting the density, increasing cost, and slowing performance). Instead, programmable elements are strategically placed to provide a trade-off between routability, density, and performance. As with antifuse FPGAs, some wires may extend the length of one logic cell, and others may extend further, again balancing routing flexibility with density and performance.

Figure 2-32 illustrates the Xilinx XC4000 interconnect. This figure shows only those wires that extend the length of one cell (named *single-length wires* for this architecture). Programmable elements exist at the intersection of the logic cell (named *configurable logic block (CLB)* by Xilinx) inputs and single-length wires. The outputs can connect to some of the single-length wires. For one CLB to communicate with another or with I/O via single-length wires, wires must connect through

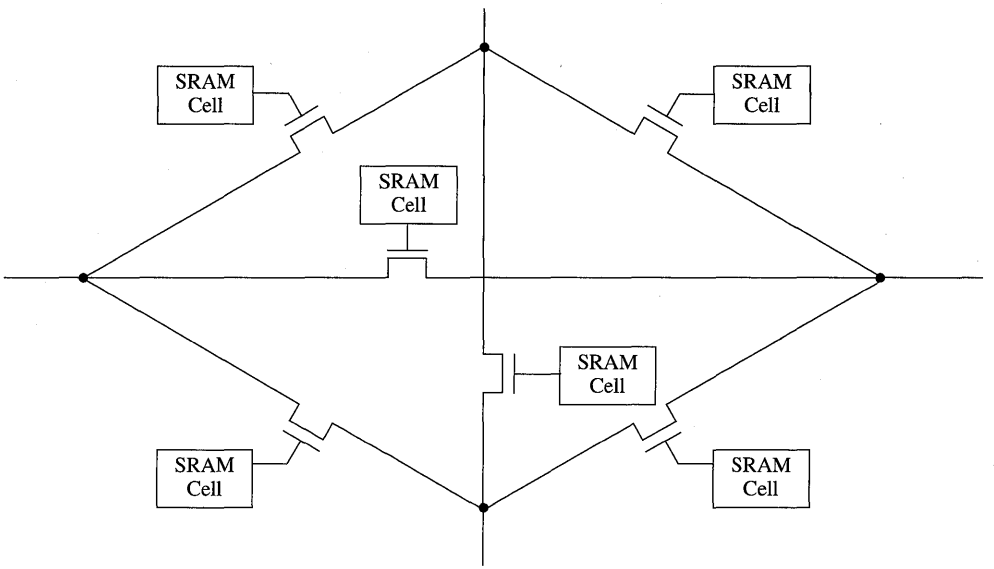


Figure 2-30 SRAM cells used to control state of pass transistors

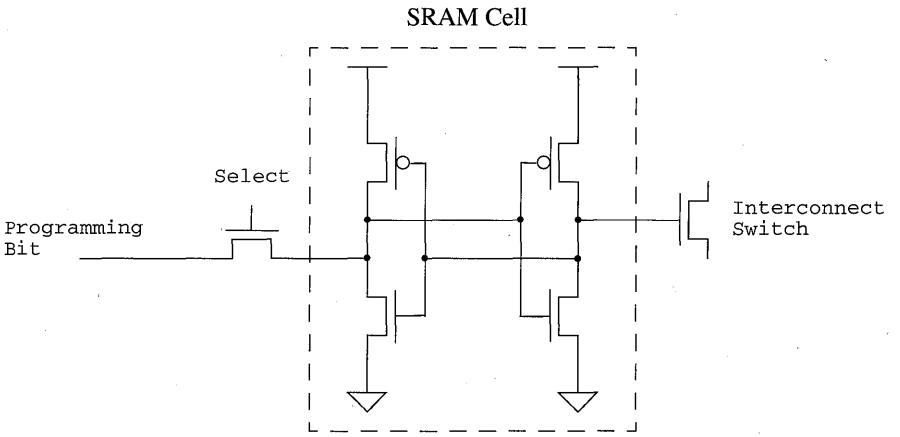


Figure 2-31 An SRAM cell

the switch matrices. Each wire on one side of a switch matrix can connect to one wire on the other side of the matrix, as illustrated by dots indicating where connections can be established.

Logic Cell Architecture

Logic cell architectures are influenced by routing structures: FPGAs that have routing structures that have many wires and flexible routing (in which a wire can be connected to any other wire) tend to

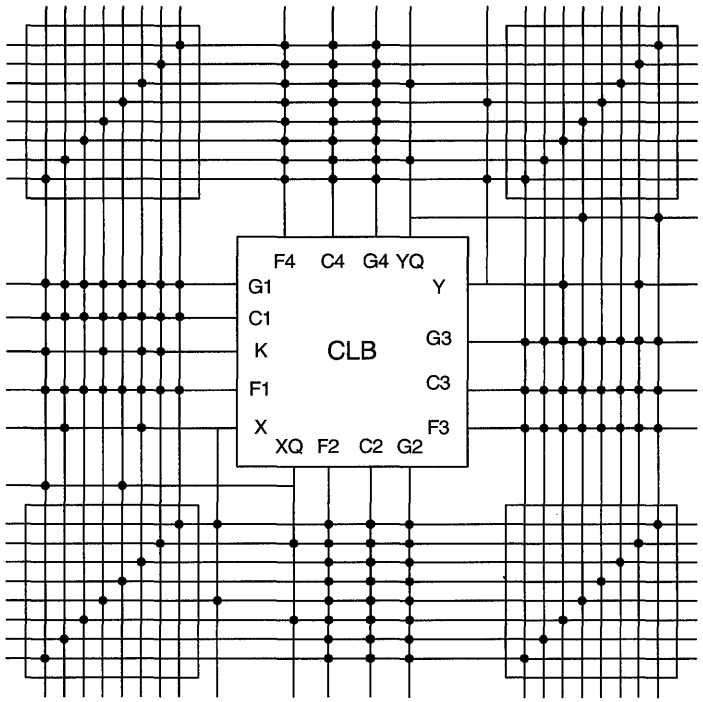


Figure 2-32 XC4000 Interconnect

have smaller logic cells with a larger fan-in and more outputs as a ratio of the number of gates in the logic cell. These are typically antifuse FPGAs. FPGAs that have routing structures with fewer wires and designated interconnections tend to have larger logic cells with less fan-in and fewer outputs as a ratio of the number of gates in the logic cell. These are typically SRAM FPGAs.

Antifuse FPGAs may use logic cells with large fan-in and relatively large fan-out because of the availability of wires to transport signals and the availability of fuses, which allow nearly any wire to connect to any other wire. Routing does not pose a problem or limitation. Antifuse FPGAs may use smaller logic cells to increase the efficiency of the logic cells. Small functions do not waste logic cell resources (e.g., a two-input AND gate will not consume a large amount of logic resources), and large functions can be built up from multiple logic cells. An architecture with small logic cells will enable the user to utilize the full capacity of the device. If a logic cell is too small, however, most functions will require multiple levels of logic cells, with each level incurring a propagation delay as well as a routing delay associated with the wire capacitance and fuse resistance and capacitance. To balance efficiency with performance, antifuse FPGAs may use slightly larger logic cells with multiple outputs that can implement multiple independent functions.

SRAM-based FPGAs typically use larger logic cells with fewer inputs and outputs. These logic cells can implement larger functions without incurring routing delays, which can be more significant because of the larger resistance and capacitance of the programmable element. However, because

they tend to have fewer outputs as a ratio of the number of gates in the logic cell, they tend to be less efficient for implementing small functions. Again, the trade-off is made between efficiency and performance and is closely tied to the routing architecture.

Figure 2-33 illustrates the logic cells of Actel's ACT3, AT & T's ORCA, Cypress's pASIC380, Xilinx's XC4000, and Altera's FLEX 8000 families of FPGAs. The first two are logic cells of antifuse FPGAs; the remainder are from SRAM FPGAs.

The ACT3 logic cell (*logic module,* or *LM*) has eight inputs and one output; there are two types of logic modules: combinational and sequential (one includes a flip-flop, and the other does not). The logic is based on multiplexers, which are universal logic modules (i.e., a 2^n to -1 multiplexer can implement any function of n + 1 or fewer variables, using 0, 1, and the true and complement of the variables as select lines and multiplexer inputs). As such, the modules can implement any of several hundred functions of the inputs. Larger functions can be built by cascading logic cells.

The ORCA logic cells, or *programmable function units (PFUs)*, have 14 inputs and 5 outputs. Each PFU can be configured as four 4-input LUTs (look-up tables), two 5-input LUTs, or one 6-input LUT, which can implement a function of up to 11 inputs. Each PFU has four flip-flops and can be configured for arithmetic circuits or read/write RAM.

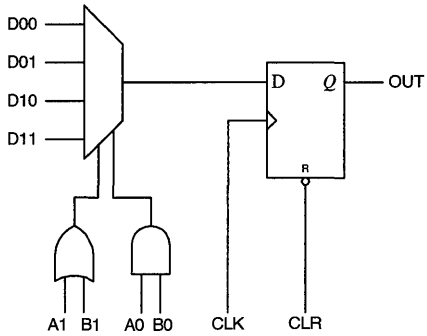
The pASIC380 logic cell has 23 inputs and 5 outputs and can implement multiple independent functions for efficiency. The 4-to-1 multiplexer ensures that the logic cell can implement any function of 3 variables, and the wide AND gates also allow gating functions of up to 14 inputs. Exclusive-OR gates, OR gates, and a sum of three small products, as well as counter macros can be implemented. Larger functions can be built by cascading logic cells. All logic cells include a flip-flop.

The XC4000 CLB has 13 inputs and 4 outputs. It is a complex cell: two 4-input lookup tables (LUTs) feed another 3-input LUT. Each CLB can implement any one function of four or five variables and some functions of up to nine variables. Alternatively, a CLB can be configured to implement two functions of four variables, or one of two variables and another of five. Each CLB has two flip-flops. The CLB can also be configured for special arithmetic circuits, such as a two-bit adder with carry-in and carry-out, or as a read/write RAM of 16 bits for storing data.

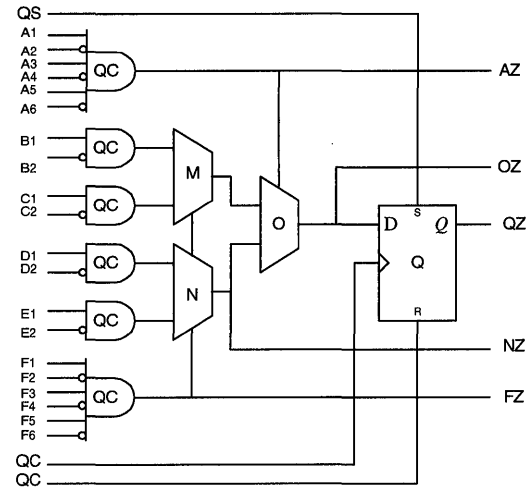
The FLEX 8000 architecture may be considered a hybrid CPLD/FPGA architecture. It addresses the SRAM routing issues differently. In this architecture, the logic cells (*logic array blocks,* or *LABs*) are made up of eight *logic elements (LEs)*. Each LAB has a local interconnect in which any LE can connect to any other LE. The local interconnect and the relatively large size of the LABs reduce the routing congestion on the inter-LAB and I/O routing channels. Each LE has a four-input LUT and can implement a single function of four variables. An LE also has carry circuitry for arithmetic circuits, as well as a flip-flop.

Timing

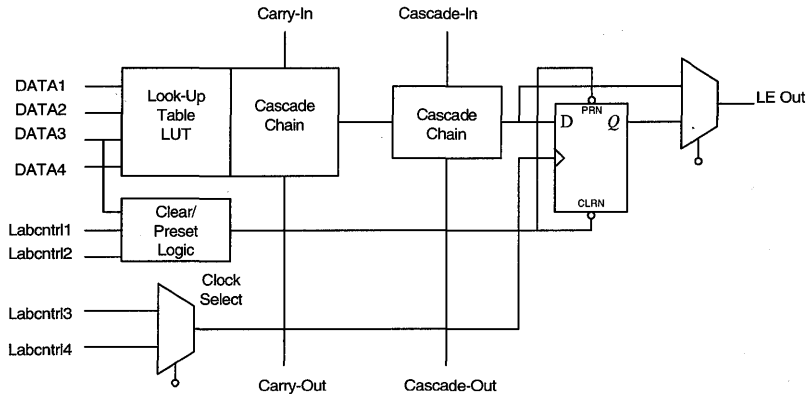
Timing for designs implemented in FPGAs cannot easily be predicted for any but the simplest of designs. Signal propagation delays are a function of the number of cascaded logic cells, the signal path in the logic cells, the number of programmable interconnects through which the signal propagates (as well as the technology, antifuse or SRAM), fan-out, and I/O cell delays. Without a priori knowledge of the value of each of these variables (how many logic cells, number of programmable interconnects, fan-out, etc.)—that is, without a knowledge of how the design will be placed and routed—the propagation delays and system performance cannot be predicted with precision. This is not unlike the dilemma faced when developing a semi-custom gate array. For



Actel ACT3

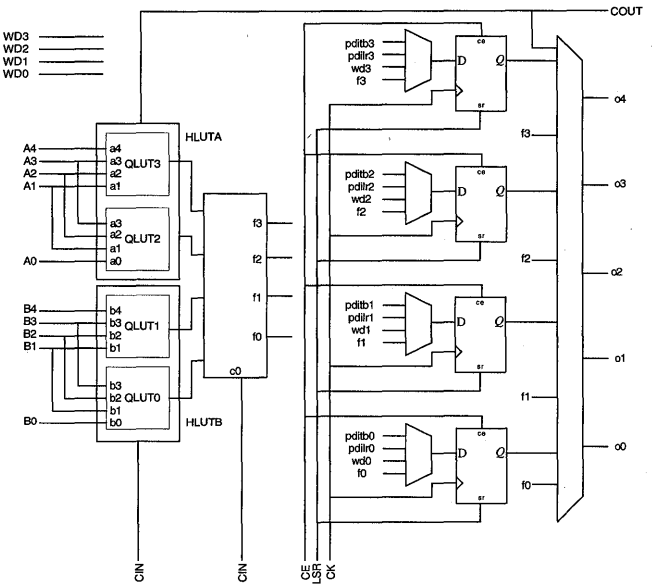


Cypress pASIC380

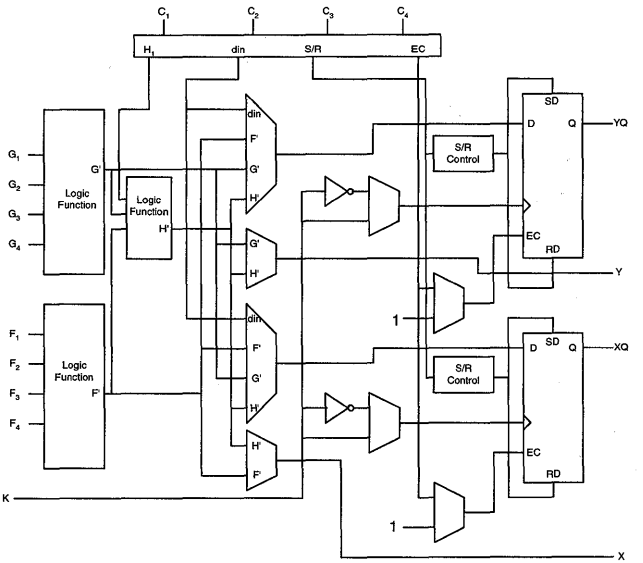


Altera FLEX8000

Figure 2-33 The logic cells of several FPGAs



AT&T ORCA



Xilinx XC4000

Figure 2-33 (continued)

systems that do not have high-performance requirements, any FPGA will potentially meet performance requirements.

Having an understanding of the technologies as well as the routing and logic cell architectures can help a designer choose an FPGA for a particular application. Additionally, HDLs like VHDL and Verilog allow relative design independence, permitting a designer to benchmark design performance from one architecture to the next without reentering a design.

Comparing SRAM to Antifuse

Vendors of both SRAM and antifuse FPGAs attempt to service the same market. However, the choice in technology forces a conscious and deliberate decision to focus on a segment of that market. We return now to our initial assumptions as to what those needs are and compare which product, an SRAM or antifuse-based FPGA, best services those needs.

Performance. Many designs push the limits of FPGA system performance, and designers often like to know where to start. Some generalizations may be made: (1) At present, peak system performance achievable from an 8,000 gate FPGA is less than 50 MHz (despite what advertisers will claim) and for most designs is around 35 MHz. Of course, design techniques such as pipe-lining can help achieve higher performance. (2) Presently, antifuse FPGAs offer the highest performance for most designs, due, in large part, to the smaller resistance of programmable links (50 ohms vs. 1,000 ohms) as well as the flexibility in routing, which does not prohibit signals from taking the most direct paths. However, some SRAM FPGAs have dedicated carry logic to provide better performance for some applications.

Density and capacity. Although the size of an antifuse FPGA logic cell is typically smaller and the number of outputs as a ratio of the number of logic cell gates is greater than those for its SRAM counterpart, allowing these logic cells to implement more user logic per available gate (greater capacity), SRAM FPGAs are presently available at higher densities. AT & T, for example, has a 40,000-gate FPGA. Most vendors of SRAM-based FPGAs have plans to achieve upwards of 50,000 usable gate densities in the next three years, whereas vendors of antifuse-based FPGAs have plans to reach upwards of 25,000 usable gates. For massive integration, SRAM FPGAs provide the solution.

Ease of use. This market need is for the designer to be able to develop a design quickly and easily, both because of the availability of easy-to-use software and "friendly, forgiving" device architectures. The availability of software varies from one silicon or software vendor to the next. Those that have been in the market longer tend to have the most support. Friendly, forgiving architectures make it easy to implement optimal structures (anything from simple AND gates to counters, arithmetic circuits, and state machines), have routing flexibility so that software tools can automatically place and route a design, and can accommodate design changes with the same pinout and achievable performance.

Antifuse-based architectures, for the most part, have the lead in this area: Routable architectures make it easy for software to be developed so that placing and routing can be done automatically and so that design changes can still fit in the same pinout by rerouting (the routing limitations of some SRAM FPGAs make it impossible or difficult—placing and routing must be done by hand—for a design to fit with the same pinout). Design changes usually can be accommodated with little impact on timing, provided that multiple logic cell delays are not added, because routing changes that require additional programmable links cause only incremental delays.

In-system programmability (ISP) and in-circuit reconfigurability (ICR) . Antifuse FPGAs are OTP (one time programmable) devices. While it is clearly not possible to reconfigure (reprogram)

such a device, it is possible to program such a device in-system, but the programming yield and programming times of antifuse FPGAs currently make ISP cost prohibitive. (Scrapping one device in a hundred is usually not a problem, particularly if the device can be replaced without charge, but scrapping one board or system per one hundred is usually too costly.). Because the ability to program an SRAM device can be verified before the device is delivered to the customer, SRAM FPGAs have significantly higher programming yields, allowing ISP for all and ICR for some. In-system programmability is used to mainstream the manufacturing flow, although in-system programmability usually requires an on-board serial EPROM or a card connector. With ISP, fewer parts and inventories need to be handled. In-circuit reconfigurability is also an emerging need: it is the ability to reconfigure an FPGA "in the field," either by a field technician or an end user, from a serial EPROM or perhaps data downloaded from a disk. Reconfiguring in the field is only feasible if the design can fit, route, and use the same pinout. For some device architectures, this requirement can pose a problem.

Other FPGA Features

Other features commonly advertised by FPGA vendors are low-skew clock buffers, lower power consumption, 5V/3.3V operation, JTAG-compliant boundary scan, on-chip RAM (memory), and PCI compliance.

Vendors of FPGAs and CPLDs specify power differently—there isn't a common measuring stick. It's important to read the fine print if using the least amount of power is critical. This can also be true of performance.

Vendors may use the term "PCI compliant" loosely: it usually means that the device meets the requirements of the PCI components electrical checklist for a specified range of temperature, but it often does not mean that a real-world PCI interface design can operate from dc to 33 MHz, as required by PCI.

Futures

We do not have a crystal ball with which to see where the future of programmable logic lies in the years to come, but that won't keep us from making predictions, some of which are obvious: (1) Performance and density requirements will increase. Three-layer metal technology as well as smaller processes such as 0.35 micron CMOS (most programmable logic is presently on 0.65 micron CMOS) will help both performance and density. As performance improves at higher densities, FPGAs will take some of the ASIC market because FPGAs provide several advantages: There are not any NREs (resulting in lower initial and low volume costs), designing is less risky (multiple cycles are acceptable), the design cycle is quicker (simulation can be less exhaustive; manufacturing is not part of the total design time), working with an ASIC vendor is eliminated, and fewer resources are required (one person rather than a team). (2) The 3.3V market will continue to grow. (3) Large devices may incorporate on-board PLLs (phase-locked loops) to control clock skew, and perhaps to purposely introduce clock skew in output flip-flops to achieve short clock-to-output delays. (4) In-system programming will continue to be used to mainstream manufacturing. (5) In-circuit reconfigurability will lead to innovative designs, which will fuel the need for more devices that are truly reconfigurable. Reconfigurability, however, requires robust routing resources and a reprogrammable technology. These two requirements may be at odds with each other and with end user performance requirements. It may take a few years for reconfigurability to be viable in a large percentage of systems, but if it takes hold, it has the potential to change design methodologies to include programmable and reprogrammable systems.

Exercises

1. *Noactivity* is a signal that must be asserted if none of the ports in a 4-port network repeater are active. (a) Implement the logic for *noactivity* using the TTL inventory listed at the beginning of the chapter. (b) Implement the logic for *noactivity* in a PAL 16R4. (c) Compare the implementations: How many TTL devices are required? What percentage of the 16R4 product terms is required? What percentage of the macrocells? Compare the levels of logic. Compare the total propagation delays and standby power requirements.

2. What percentage of the 16R4 resources (I/Os, product terms, and macrocells) would be required to implement the collision signal X (described at the beginning of the chapter) and *activity* (see exercise above) in the same 16R4?

3. After producing several production units, a design change is required: The logic for *collision* and *activity* must change. Describe the corrective action if TTL devices were used. Describe the corrective action if a PAL was used.

4. Determine the expressions required to implement a 10-bit counter in a 22V10. How can resources be allocated in order for this design to fit? What is the maximum frequency of operation, given the timing specifications of page 23? What if a LOAD input is provided along with 10 data inputs to the counter?

5. *Odd* is a registered signal that is the exclusive OR of A, B, C, D, E, and F. How can this be implemented in a 22V10 such that it will fit? What are the setup and hold times that must be met? What is the clock-to-output delay and maximum frequency of operation?

6. Implement a 4-bit counter with an enable in a 22V10. What are the setup and hold time requirements? What is the clock-to-output delay and maximum frequency of operation?

7. Describe the features of a 16V8 in relation to those of a 16L8 and 22V10.

8. Implement the following functions in each of the logic cells of *Figure 2-33* and compare efficiency: (a) two-input AND, (b) seven-input AND, (c) two-bit counter, and (d) two-bit adder, (e) two input OR, (f) eight input XOR (parity generator)

9. Implement a 4-bit adder with inputs A[3:0] and B[3:0] and outputs S[3:0] (Sum) and Cout (Carry-out) in a 22V10. What is the resulting performance? What is the implementation and performance in a FLASH370 CPLD? How do both of these change if the design is a 4-bit accumulator with input A[3:0] and outputs S[3:0] and Cout.

10. How would an 8-bit magnitude compare be implemented in a CPLD and in an FPGA? In both cases, are device resources being used evenly, or are some device resources being used more than others? Are any device resources left unused with this particular design?

11. Show how a 4-to-1 Multiplexor can implement any logic function of three input variables (A, B, and C). How many total logic functions are there for three inputs? *n* inputs?

12. List features that you would use to label a device as a PAL, CPLD, FPGA or an ASIC.

13. List major differences in the I/O and Buried Macrocell structures of the Cypress 340MAX and the Cypress FLASH370 family of CPLDs.

14. List major differences in the Logic cell structures of the X4000 and the Cypress pASIC380 family of FPGAs. Refer to the data books if necessary.

15. For a given application, how would you choose between an FPGA or a CPLD as your target architecture.

16. Create your own 256-macrocell CPLD. What are the issues you would be concerned with? What features would you put in? Justify your choices.

17. Create your own 20K gates Antifuse FPGA. What are the issues you would be concerned with? What features would you put in? Justify your choices. Justify why would you choose to use this device as opposed to an ASIC.

3 Entities and Architectures

This chapter discusses the basic building blocks of VHDL design, the *entity* and the *architecture*. We will make analogies to schematic design entry and high-level programming languages to help place VHDL concepts into a familiar framework. We'll be careful not to overstate the analogies, however, because coding VHDL is very different from coding in a programming language. An important concept to keep clear as you write VHDL code is that you are *designing* for hardware: Your descriptions in VHDL code will be synthesized into digital logic for a programmable logic device.

A Simple Design

The code example below is a VHDL description of a 4-bit equality comparator. It is divided into two sections: an *entity* and an *architecture*. In this example, the uppercase words are required by VHDL; the lowercase words are furnished by the designer. The keywords are in uppercase in this first example for readability only—VHDL does not require that the keywords be in uppercase. In fact, VHDL is not case sensitive. The line numbers are also *not* part of a VHDL design—they are used here to help us identify specific lines of code in our discussion. This 4-bit equality comparator serves to demonstrate the basic framework of entities and architectures. Following the code listing is a line-by-line explanation of the code, but first read through the listing; perhaps you can understand the code structure without an explanation.

```
1  -- eqcomp4 is a four bit equality comparator
2  ENTITY eqcomp4 IS
3      PORT (a, b: IN BIT_VECTOR(3 DOWNTO 0);
4          equals: OUT BIT);
5  END eqcomp4;
6
7  ARCHITECTURE dataflow OF eqcomp4 IS
8  BEGIN
9      equals <= '1' WHEN (a = b) ELSE '0';    -- equals is active high
10 END dataflow;
```

Listing 3-1 Dataflow design of a 4-bit comparator

The characters "--" introduce a comment. Line 1 is a comment line. Comments help to document your design; they are for the reader and are ignored by the compiler. The comment continues to the end of the line. To continue a comment on another line, you would need to start the line with a double-dash. Comments can also start anywhere in a line, as shown in line 9. Everything to the right of the double-dash is part of the comment. Lines 2 through 5 describe the I/O of a 4-bit equality comparator called *eqcomp4*. Lines 2 and 5 begin and end the declaration for the *eqcomp4* entity. Line 3 begins a port, or pin, declaration and the characters ");" (without the quotes) at the end of line 4 end the port declaration. Ports are points of communication of the entity with anything outside of the entity. On line 3 of this example, we declare two ports called *a* and *b*. These ports are inputs to the design that are 4-bit buses. Each member of the bus, a(0) for instance, is a BIT, which means it may have the value of '0' or '1'. Finally, *equals* is declared as an output bit in line 4. The entity has a schematic symbol equivalent, as shown in *Figure 3-1*.

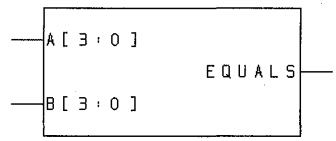


Figure 3-1 Schematic Symbol Equivalent of *eqcomp4* Entity

Lines 7 through 10 describe what our entity, *eqcomp4*, does. This is called the *architecture* of our entity; it begins on line 7 with the keyword ARCHITECTURE and ends with "END *dataflow*;" at line 10. In line 7, we give the architecture a name, *dataflow*, and identify the entity that it describes: "OF *eqcomp4*." (The name that we gave the architecture was our choice. We chose *dataflow* because the architecture falls into the class of dataflow descriptions. We'll explore this and other classes of architectural descriptions later in the chapter.) Line 8, obviously enough, begins the architecture description with the keyword BEGIN, and line 9 is where the digital logic is described. This simple architecture includes one equality comparator. Line 9 states that when the value of bus *a* is equal to the value of bus *b*, then *equals* gets '1', otherwise *equals* gets '0'. Read from left to right: "*equals* gets '1' when *a* equals *b*, else '0'." The <= symbol is an operator that can be read "gets" or "is assigned to." For brevity, we'll often use "gets," despite its being jargon. The comparison is bitwise from left to right (i.e., a(3) is compared to b(3), a(2) is compared to b(2), etc.). The most significant bits (MSB) for *a* and *b* are the leftmost bits *a(3)* and *b(3)*. For clarity, we will typically order the bits from "x downto 0" in order that the most significant bit is the one with the highest index.

Entity and Architecture Pairs

The design example above illustrates that a VHDL design consists of an entity and an architecture pair: The entity describes the design I/O and the architecture describes the contents of the design. Now that we have worked through an example to give you an idea of how designs are put together, we'll start looking at the details of VHDL design syntax and semantics. We'll begin by examining the syntax used to describe entities, after which we will describe the classes of architecture.

We could start with a detailed discussion of identifiers, data objects, and data types. However, we believe that you'll benefit more by first gaining a broad understanding of how VHDL designs are put together before delving into the details of data. If you prefer, read the section later in this chapter on identifiers, data objects, and data types before returning here.

Entities

A VHDL entity describes the inputs and outputs (I/O) of a design. This could be the I/O of a component in a larger, hierarchical design, or—if the VHDL entity is a device-level description—the I/O of a device. The entity is analogous to a schematic symbol, which describes a component's connections to the rest of a design. A schematic symbol for a 4-bit adder (*add4*) is shown in *Figure 3-2*. You can see that the 4-bit adder has a name (*add4*), two 4-bit inputs (*a* and *b*), a carry-in input (*ci*), a 4-bit output (*sum*), and a carry-out output (*co*). These items are also contained in an entity:

```
entity add4 is port(
            a, b:       in std_logic_vector(3 downto 0);
```

```
            ci:         in std_logic;
            sum:        out std_logic_vector(3 downto 0);
            co:         out std_logic);
end add4;
```

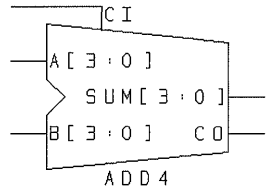


Figure 3-2 Symbol equivalent of entity *ADD4*

Ports

Each I/O signal in an entity is referred to as a port, which is analogous to a pin in a schematic symbol. (A port is a data object in VHDL. Like other data objects, it can be assigned values, which can be used in expressions. We'll investigate other data objects in the next section of the chapter.) The set of ports that you define in your entity is referred to as a port declaration. Each port that you declare must have a name, a direction (or mode), and a data type. The first part of the declaration, the port name, is self-explanatory. Legal VHDL identifiers (names) are described on page 70.

Modes

The mode describes the direction in which data is transferred through a port. The mode can be one of four values: IN, OUT, INOUT, or BUFFER. A port that is declared as mode IN describes a port in which data flows only into the entity. The driver for a port of mode IN is external to the entity. A port that is declared as mode OUT describes a port in which data flows only from its source to the output port (or "pin") of the entity. The driver for a port of mode OUT is from within the entity. Mode OUT does not allow for feedback within the associated architecture. For internal feedback (i.e., to use this port as a driver within the architecture), you'll need to declare a port as mode BUFFER or mode INOUT. A port that is declared as mode BUFFER is similar to a port that is declared as mode OUT, except that it *does* allow for internal feedback. Mode BUFFER does not allow for bidirectional ports, however, because it does not permit the signal to be driven from outside of the entity. An additional caveat for the use of the mode BUFFER will be explained in chapter 6, "The Design of a 100BASE-T4 Network Repeater." For bidirectional signals, you must declare a port as mode INOUT. This mode describes a port in which data can flow into or out of the entity. In other words, the signal driver can be inside or outside of the entity. Mode INOUT also allows for internal feedback. Mode INOUT can be used anywhere that mode BUFFER is used; that is, everywhere that mode BUFFER is used in a design could be replaced with mode INOUT. However, doing so may complicate the reading of large design listings, making it difficult to discern the source of the signals. If the mode of a port is not specified, then the port is of the default mode IN.

Mode IN is primarily used for clock inputs, control inputs (like load, reset, enable), or unidirectional data inputs. Mode OUT is used for outputs such as a terminal count output (a terminal count is asserted when the value of a counter reaches a predefined value). Mode BUFFER is used for ports

such as the counter itself (the present state of a counter must be used to determine its next state, so it's value must be in the feedback loop, thereby necessitating a mode other than just OUT). Mode INOUT could be used for all of the ports mentioned so far: the dedicated inputs, the terminal count, and the counter outputs. Although using one mode, INOUT, for all signals would be legal, it reduces the readability of the code. A more appropriate use for mode INOUT is for signals that require feedback (like the counter) or that are truly bi-directional such as the multiplexed address/data bus of a DMA controller. *Figure 3-3* illustrates the classification of modes.

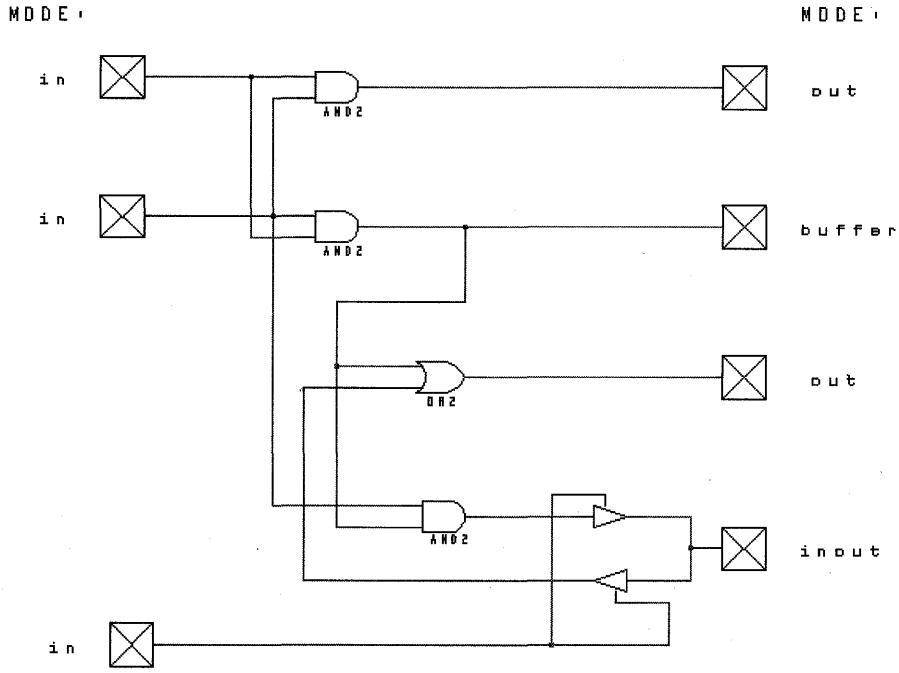


Figure 3-3 Modes and their signal sources

Types

In addition to specifying identifiers (names) and modes for ports, you must also declare the ports' data types. The most useful and well-supported types provided by the IEEE *std_logic_1164* package are the types std_ulogic, std_logic, and arrays of these types. As the names imply, "standard logic" is intended to be a standard type used to describe circuits for synthesis and simulation. For simulation and synthesis software to process these types, their declarations must be made visible to the entity by way of a USE clause. The most useful and well-supported types provided by the IEEE 1076/93 standard *and* which are applicable to synthesis are the data types boolean, bit, bit_vector, and integer. Many of the examples throughout this book will use the std_logic type to reinforce the idea that it is

a standard. However, you should be aware that you are not restricted from using other types. We'll defer a more detailed discussion of data types to page 72.

Architectures

Every architecture is associated with an entity. An architecture describes the contents of an entity; that is, it describes what an entity does. If the entity is viewed as the engineer's "black box" (for which the inputs and outputs are known, but not the details of what is inside the box), then the architecture is the internal view of the black box. VHDL allows you to write your designs using different "styles" of architecture and to mix and match these styles as you see fit. Depending on the style, an architecture is classified as a behavioral or structural description, or a combination of the two. The name given to the architecture classification is not important, and often the style or class of design description that you use for synthesis is not important. However, the terms "behavioral" and "structural" will give us a common vocabulary that we can use when discussing these different classes of design description.

Behavioral Descriptions

```
1  entity eqcomp4 is port(
2      a, b:   in std_logic_vector(3 downto 0);
3      equals:   out std_logic);
4  end eqcomp4;
5
6  architecture behavioral of eqcomp4 is
7  begin
8  comp:   process (a, b)
9      begin
10         if a = b then
11             equals <= '1';
12         else
13             equals <= '0';
14         end if;
15     end process comp;
16 end behavioral;
```

Listing 2-2 Behavioral architecture description for *eqcomp4*.

Listing 2-2 is an example of a behavioral description, as is *Listing 2-1*. Why is it "behavioral," or rather what makes it "behavioral?" After reading the code listing, you may already have an idea. Simply put, it's because of the algorithmic way in which the architecture is described. Behavioral descriptions are sometimes referred to as "high-" descriptions because of the resemblance to high-level programming languages. Rather than specifying the structure or netlist of a circuit, you specify signal assignments, or circuit "behavior." The advantage to high-level descriptions is that you don't need to focus on the gate-level implementation of a design; instead, you can focus your efforts on describing how the circuit is to "behave."

Lines 1 through 4 declare the entity and ports (I/O) for a 4-bit comparator. Line 6 declares the architecture that begins on line 7. Lines 8 through 15 embody the algorithm for this comparator. Processes are one of VHDL's design constructs for embodying algorithms. Lines 10 through 14 use a comparison operator and the IF-THEN-ELSE construct to indicate that *equals* should be asserted when *a* is equal to *b*.

If you've noticed that *Listing 2-2* is just another way to describe the 4-bit equality comparator of *Listing 2-1*, then you've discovered one of VHDL's greatest strengths: the ability to describe the same circuit using different styles. We'll consider three more ways to describe a 4-bit equality comparator as we continue to discuss behavioral and structural design descriptions.

Processes and Sequential Statements

Processes, like the one in the architecture of *Listing 2-2*, permit the description of circuits using sequential assignment statements, or algorithms. The architecture of *Listing 2-2* can be rewritten as:

```
1   architecture behavioral of eqcomp4 is
2   begin
3   comp:   process (a, b)
4     begin
5       equals <= '0';
6       if a = b then
7         equals <= '1';
8       end if;
9     end process comp;
10  end behavioral;
```

Listing 2-3 Alternative implementation of *eqcomp4*; *equals* has default value of '0'

The ordering of the statements in this process is important: It indicates that as a default *equals* should be assigned '0', but that if *a* is equivalent to *b,* then *equals* should be assigned '1'. If the statement "equals <= '0';" was placed after line 8, then this design would take on a completely different meaning: *equals* would always be '0'.

Modelling vs. Designing

Before proceeding, it will be instructive to explore briefly a few differences between writing VHDL code for modelling and designing. Understanding these differences will clarify the semantics (and motivation behind the semantics) for some VHDL statements. Having a firm grasp of these semantics will enable you to write accurate code. Without clarification, it is easy to misconstrue the concepts of concurrent and sequential statements, as well as event scheduling. Ignoring these misconceptions can lead to poor VHDL coding and frustration with synthesis and simulation software tools, as well as the VHDL language itself.

Modelling is the process of describing the behavior or structure of logic circuits that already exist. Designing is the process of describing the behavior or structure of logic circuits that have yet to be generated, or synthesized.

Because VHDL code that is written for synthesis must accurately reflect the behavior or structure of a piece of logic, part of the designer's challenge is to have a clear idea not only of the behavior or structure of the logic to be created but also of how to model that logic. The same code that is processed by synthesis software to produce logic must also be compatible with software that processes VHDL for simulation. The logic that is created for synthesis must match a functional simulation. Whereas all code that is written for synthesis can be simulated, the opposite—all code that is written for simulation can be synthesized—is not true, as you'll soon see.

Two key concepts in simulation and modelling that are often ignored by writers of VHDL code for synthesis are the concepts of simulation time and event scheduling. A signal assignment statement causes an event to be scheduled for the target of the signal assignment if the value of the calculation of the expression to the right of the signal assignment operator is different than the present value of the target signal. An event is a change in value of a signal and is scheduled to occur after a specified period of time, even if that period is zero. The signal will not assume its new value until the event occurs. In the case of a zero delay event, the signal will not assume its new value until the end of the current simulation time. The current simulation time is over upon the completion of execution of a concurrent statement (a process taken as a whole also constitutes a concurrent statement) or if simulation time is executed.

For instance, during the simulation of the following architecture,

```
architecture a_model of two_gates
  x <= a AND b after 5 ns;
  y <= not b;
end a_model;
```

suppose that *y* is '0' and *x* is '1' because both *a* and *b* have been '1' for a long period of time. But now, at time t = 100 ns, *b* has transitioned from a '1' to a '0.'

An event on *x* will be scheduled for 5 ns from now, at which time it will assume the value of '0', and an event on *y* will be scheduled for zero delay. At the end of the current simulation time, *y* assumes the value of '1'. The simulation software will show that immediately after both statements are executed, but still at t = 100 ns, *x* retains its value for 5 ns and *y* assumes the value of '1'.

How does synthesis software process this code as compared to simulation software? To start with, an AFTER clause is ignored by synthesis software because such a clause is typically used to model a propagation delay. In this text, only the test fixture designs of chapter 9, "Creating Test Fixtures", will use the AFTER clause. All other designs are intended specifically for synthesis. Also, synthesis software has the task of generating logic (equations for a fitter or a netlist for a place and route tool) such that a model of the generated logic will match the functionality the design description as simulated. For this description a simple AND gate and inverter are generated.

Understanding the make-up and interpretation of a PROCESS from a modelling and simulation point-of-view can help in writing processes that will be interpreted by synthesis software in generating logic. It also ensures that you will be able to use simulation software to test your design description.

As an example, we will evaluate *Listing 2-3* first from a simulation stand-point, then from a synthesis standpoint.

Like all processes, the process of *Listing 2-3* contains a *sensitivity list.* The sensitivity list is a list of signals for which a change in value of one of these signals will cause the process to become activated. Once activated, the statements between the BEGIN and END statements are executed in sequential order. After reaching the END PROCESS statement, the process becomes inactive once again.

This process will become active if the value of either signal *a* or signal *b* transitions. The statements within the process are executed in sequence. The first statement schedules an event on *equals* for zero delay. That is, at the end of the current simulation time (i.e., end of the process), *equals* will assume the value of '0,' provided that the currently scheduled event is not preempted before the end of the current simulation time (i.e., before the end of the process). If signals *a* and *b* are equivalent,

then the next statement—the IF statement—preempts the currently scheduled event, and a new event is scheduled on *equals* for zero delay. The end of the process is reached, indicating the end of the current simulation time, so the process becomes inactive again until the next transition in *a* or *b*, and *equals* assumes its scheduled value.

Synthesis software must produce logic that is accurately modelled by the design description. In this case, synthesis software will step through the code, find that '0' is the default value for *equals* and that *equals* should be asserted if *a* is equivalent to *b*.It will produce the logic of an equality comparator.

The following example is slightly more involved.

```
architecture behavioral of eqcomp4 is
    begin
    synth:   process (c, d)
        begin
            x <= '0'; y <= '1';
            if (c = '0' and d = '0') then
                        x <= '1';
            elsif (c = '0' and d = '1') then
                        x <= '1';
            elsif (c = '1' and d = '1') then
                        y <= '0';
            end if;
        end process synth;
    end behavioral;
```

Synthesis software will step through the code, find the default values and subsequent conditional assignments in order to generate the following equations for *x* and *y*:

$$x = \bar{c}\bar{d} + \bar{c}d \quad \text{and } \bar{y} = cd$$
$$= \bar{c}$$

Why does VHDL have the concept of event scheduling? The primary reason why signal assignments are delayed is to accurately model propagation delays. For example, the statement,

```
x <= a AND b after 5 ns;
```

can be used to model an AND gate. A change in *a*, *b*, or both that would cause a change in *x* will propagate in 5 ns. If *a*, *b*, or both change again within 5 ns (of the first change) such that *x* will be its original value, then *x* will not change value. The real circuit may also behave in this way, or a glitch may be produced.

Another reason for event scheduling is that a computer (with one CPU) that is used for simulation can only update one value at a time. Even if several signals are described as having zero delay, only one signal can be updated at a time by the computer. The simulation software must suspend the current simulation time while new values are calculated and scheduled.

We examine the following design to illustrate how someone unfamiliar with the concepts of event scheduling and simulation time may expect to write code that will logically AND all bits of a bus.

```
entity my_and is port(
   a_bus:    in bit_vector(7 downto 0);
```

```
   x:       buffer bit);
end my_and;
architecture wont_work of my_and is
begin
anding: process (a_bus)
   begin
      x <= '1';
      for i in 7 downto 0 loop
         x <= a_bus(i) AND x;
      end loop;
   end process;
end wont_work;
```

Listing 2-4 Inaccurate model of an 8-bit AND gate; initialization and scheduling causes output to always be '0'.

As the architecture name indicates, this process won't work as the designer desired. The designer wants for the output *x* to represent the logical AND of all of the bits in *a_bus*, but because of the way that the VHDL standard specifies initialization for signals like *x*, this code does not accurately model that the designer desires.

Let's step through the process. The initial value at the beginning of simulation for a signal of a data object of type bit is '0'; thus, *x* is initially '0.'The process is activated any time that *a_bus* changes value. The first statement schedules an event on *x* for zero delay. That is, *x* will assume the value of '1' *after* the current simulation time, provided that the currently scheduled event is not preempted. In fact, the currently scheduled event *is* preempted during the first iteration of the loop (when *i* is 7). In this iteration, an event on *x* is rescheduled for zero delay, and its new value will be the result of the expression *a_bus AND x*. The calculation of this value must be '0' because *x* is '0'. (The previously scheduled event was preempted, so it never changed value from it's initial value.) Thus, the iteration of $i = 7$ causes an event on *x* to be scheduled for zero delay unless the currently scheduled event is preempted. Further events on *x* do not occur because subsequent iterations of the loop do not change the currently scheduled value for *x*. The current simulation time is over when the simulation software reaches the END PROCESS statement, at which time *x* will assume the value of '0.'

Synthesis software must ensure that the logic it produces will match the functionality of simulation, so output *x* is hard-wired to '0,' which is not what the designer desired.

The design can easily be corrected to produce the desired effect by introducing a variable into the process. The scope of a variable is a process. When a process is active, its variables can be used in that process. When a process is not active, its variable cannot be used. An assignment to a variable differs from an assignment to a signal because it is immediate, not delayed or scheduled. So, a new architecture can be written for the *my_and* entity:

```
architecture will_work of my_and is
begin
anding: process (a_bus)
      variable tmp: bit := '1'
   begin
      for i in 7 downto 0 loop
         tmp := a_bus(i) AND tmp;
      end loop;
      x <= tmp;
```

```
   end process;
end will_work;
```

Listing 2-5 An 8-bit AND gate; assignments to variables are immediate

A variable, *tmp*, is declared and initialized to '1' in the process declarative region. Each time the process is activated, the variable is available and initialized to '1'. The variable must be initialized each time the process is activated, as it will not retain its previous value. The := operator is used here for the initialization expression and variable assignment, indicating immediate assignment. Thus, *tmp* is immediately assigned a value of '1,' and iterations of the loop of the implicitly declared variable *i* result in immediate, not scheduled, assignment to *tmp*. The last statement in the process is an assignment to *x*. Thus, an event on *x* is scheduled for zero delay. It will assume the value equivalent to the logical AND of all bits of *a_bus* after the current simulation time. The current simulation time is over once the process ends.

Synthesis software must generate logic that matches the design model—in this case, an 8-bit wide AND gate.

Because synthesis software interprets a process as describing either combinational or synchronous logic, some models (such as the following one) may be simulated but not synthesized.

```
entity neat_model is port(
   a, b, c: in bit;
   x:       buffer bit);
end neat_model;
architecture cant_synthesize of neat_model is
begin
proc1: process (a, b)
   begin
      x <= a and b and c;
   end process;
end cant_synthesize;
```

This model can be simulated easily. A change in *a* or *b* will cause the process to be activated and sequenced. An event on the signal *x* will be scheduled for zero delay. The value that it will assume is the logical AND of *a*, *b*, and *c*.

This circuit cannot be synthesized, however, because it is not at all clear how to build a circuit for which a transition in *c* does not cause a change in *x* but for which a change in *a* or *b* causes *x* to be the logical AND of *a*, *b*, and c.

If the signal *c* is added to the sensitivity list, then the model behaves differently, but in this case, synthesis software can generate logic—a 3-bit wide AND gate—that is accurately modelled by above description.

Logic design lends itself to descriptions using algorithms. As you review schematics, boolean equations, or a state machine diagram, you may hear yourself saying, "If signal *a* is asserted then the machine goes to the next state." VHDL provides a design methodology that enables you to capture designs with English language-like constructs in behavioral descriptions. You can capture a design in VHDL almost as if you were speaking to a colleague, "When the address is between 0000H and 4000H (inclusive), then *promsel* is asserted; when it's between 4001H and 4008H (inclusive), then *peripheral_1* is asserted..." In

behavioral descriptions, you can make use of control constructs such as IF-THEN-ELSE, CASE-WHEN, WHEN-ELSE, WITH-SELECT-WHEN, and loops, as you'll see in later chapters.

Dataflow and Concurrent Assignment

Our first example of the chapter was also a behavioral description. We reprint it again in *Listing 2-6*, changing the data types of the ports.

```
-- eqcomp4 is a four bit equality comparator
library ieee;
use ieee.std_logic_1164.all;
ENTITY eqcomp4 IS
    PORT (a, b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
        equals: OUT STD_LOGIC);
END eqcomp4;

ARCHITECTURE dataflow OF eqcomp4 IS
BEGIN
    equals <= '1' WHEN (a = b) ELSE '0';    -- equals is active high
END dataflow;
```

Listing 2-6 Dataflow architecture description for *eqcomp4*

This architecture can further be described as a dataflow architecture because it specifies how data will be transferred from signal to signal without the use of sequential statements. You'll likely want to use dataflow descriptions in those cases where it's more succinct to write simple equations or CASE-WHEN or WITH-SELECT-WHEN statements rather than a complete algorithm. Oftentimes, it's just a matter of style—what you're most comfortable with.

There is an important distinction between the signal assignment statements of *Listing 2-2* and *Listing 2-6*. The signal assignment is sequential in *Listing 2-2*, and the assignment in *Listing 2-6* is *concurrent*. Whereas the order of sequential signal assignment statements in a process can have a significant effect on the logic that is described, the order of concurrent signal assignment statements does not matter. From a simulation point of view, the order of execution of concurrent signal assignment is dependent upon which signals change. Take the following concurrent statements, for example:

```
architecture simple of example is
begin
   w <= x and y;
   x <= y OR c;
   y <= a AND b;
   z <= x AND d;
end simple;
```

Suppose that *a*, *b*, *c*, and d have been '1' for some time now. Therefore, *w*, *x*, *y*, and *z* also have steady-state values of '1.' Now suppose that *b* changes value from a '1' to a '0.' The signal assignment statement that will execute first is the one (or ones) that has *b* in its implied sensitivity list—a concurrent signal assignment statement is sensitive to transitions in any of the signals on the right-side of the <= signal assignment operator.

Thus, the signal assignment statement for *y* executes first, and an event on *y* is scheduled for zero delay. Once execution of the statement is complete, *y* assumes it's new value of '0'. This change in the value of *y* causes the signal assignment statements for *w* and *x* to execute because *y* is in their implied sensitivity lists. Simulation software will have to execute one statement before the other, even though the hardware that this describes is parallel. Either statement can be executed first. Suppose that the signal assignment statement for *w* is executed first: An event on *w* is scheduled for zero delay. Once execution of this statement is complete, *w* assumes it's new value of '0.' The signal assignment statement for *x* must now be executed: An event on *x* is scheduled for zero delay. Once execution of this statement is complete, *x* assumes the value of '0'. This change in value of *x* causes the signal assignment statements for *w* and *z* to be executed, in any order. Suppose that the signal assignment statement for *w* is executed. The evaluation of the expression on the right-side of the <= operator is not different from present value of *w*, so an event (remember, an event is a change in value of a signal) is not scheduled for *w*. Finally, the signal assignment statement for *z* is executed: an event on *z* is scheduled for zero delay. Once execution of this statement is complete, *z* assumes it's new value of '0.'

The bottom line is that the order of concurrent signal assignment statements is of no consequence.

If you're like most people, you'll develop a comfortable coding style as you first start to write VHDL code. As you gain familiarity with some constructs, you'll then explore other constructs and techniques.

Here is a style that many logic designers will already be comfortable with:

```
entity eqcomp4 is port(
    a, b:   in std_logic_vector(3 downto 0);
    equals:   out std_logic);
end eqcomp4;

ARCHITECTURE bool  OF  eqcomp4 IS
BEGIN
equals <=               NOT(a(0) XOR b(0))
            AND         NOT(a(1) XOR b(1)) ;
            AND         NOT(a(2) XOR b(2)) ;
            AND         NOT(a(3) XOR b(3)) ;
END  bool;
```

Listing 2-7 Dataflow architecture for *eqcomp* using boolean equations

Listing 2-7 is also a behavioral, dataflow description—behavioral because it does not describe the structure or netlist of the design, dataflow because it describes the way in which data flows from signal to signal. Writing boolean equations, particularly for the description of a comparator is unnecessarily cumbersome. Suppose that the size of ports *a* and *b* were increased. The architecture of *Listing 2-7* would require modification of the expression for *equals*, whereas the other architectures used to describe the comparator in this section are independent of the size of *a* and *b*. These architectures would not require modification. Nonetheless, there are times when boolean equations provide the most concise and clearly defined interaction of signals.

Structural Descriptions

Read through the *Listing 2-8* and observe what makes this description "structural":

```
entity eqcomp4 is port(
    a, b:   in std_logic_vector(3 downto 0);
    aeqb:   out std_logic);
end eqcomp4;

USE work.gatespkg.all;
ARCHITECTURE struct OF eqcomp4 IS
SIGNAL x : STD_LOGIC_VECTOR(0 to 3);
BEGIN
u0: xnor2 port map (a(0),b(0),x(0));
u1: xnor2 port map (a(1),b(1),x(1));
u2: xnor2 port map (a(2),b(2),x(2));
u3: xnor2 port map (a(3),b(3),x(3));
u4: and4 port map (x(0),x(1),x(2),x(3),equals);
END struct;
```

Listing 2-8 Structural description of *eqcomp4*

This design requires that *and4* and *xnor2* components be defined in a package. We have accessed these components by including a USE clause, which allows us to instantiate components from the *gatespkg* package found in the *work* library. (Libraries and packages are discussed in chapter 6, "The Design of a 100Base-T4 Network Repeater.")

Structural descriptions consist of VHDL netlists. These netlists are very much like schematic netlists: Components are *instantiated* and connected together with signals. (To *instantiate* a component is to place a component in a hierarchical design. An *instantiation* is therefore either (1) an act of instantiating (placing) a component or (2) an *instance* of a component—i.e., a particular occurrence of a component. If the word *instantiation* is new to you, then you may find the word to be awkward at first. As we continue to discuss structural descriptions in the coming chapters, you'll discover that the word "instantiation" is concise.)

Structural designs are hierarchical. In this example, separate entity and architecture pairs are created for the *and4*, *xnor2*, and *eqcomp4* designs. The *eqcomp4* design contains instances of the *xnor2* and *and4* components. *Figure 3-4* illustrates the hierarchy. The *xnor2* and *and4* components must each have associated entity and architecture pairs. The entity and architecture descriptions for the *xnor2* and *and4* are not contained in the design file for our *eqcomp4* component, rather they are accessed (included) by way of the USE clause.

A structural description for a 4-bit equality comparator is probably not an appropriate use of structural descriptions because it is more cumbersome than necessary. Large designs, however, are best decomposed into manageable subcomponents. Multiple levels of hierarchy may be called for, with the underlying components netlisted (connected) at each level of the hierarchy. Hierarchical design allows the logical decomposition of a design to be clearly defined. It also allows each of the subcomponents to be easily and individually simulated.

Comparing Architectural Descriptions

We've examined behavioral and structural architectures, and we created five different design descriptions for the same function (a 4-bit comparator). We can think of a couple of other descriptions for a 4-bit comparator, and you may be able to as well. All of these descriptions for the same function demonstrate the flexibility of VHDL, but this flexibility begs the question, "How does the synthesis and fitting of one design description differ from another?" That is, will different PLD

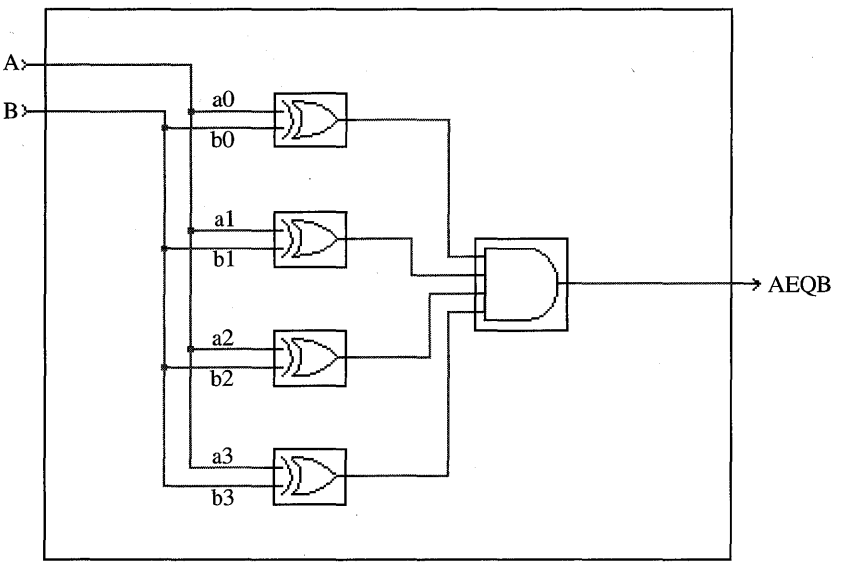


Figure 3-4 Hierarchical schematic representation of *Listing 2-8*

resources be used depending on which description is synthesized? For example, if you want the four-bit comparator to be realized in a 22V10, will the same 22V10 device resources be used regardless of which description is synthesized?

Fortunately, for simple design descriptions (such as those for our 4-bit comparator), almost any description will most assuredly be realized with the same device resources. However, this is not true of more complex design descriptions. The synthesis software must interpret complex design descriptions and create minimized logic equations for your circuit. The synthesis software as well as the place and route (or fitter) software must then determine how to implement those minimized logic equations to make the best use of available resources in the target logic device.

Different design descriptions for complex designs can result in different device resources being utilized for three reasons: (1) The VHDL code may not accurately describe an optimal function. For example, it is possible to describe a design that produces a functionally correct circuit, but which has unneeded logic that is accurately synthesized from the VHDL code. (2) The synthesis software may perform poorly in synthesizing a design description to make the best use of a device's architectural resources. For example, some synthesis software produces sum-of-products logic equations and passes those equations to the placer and router (or fitter) without having optimized the logic for the target architecture. Sum-of-products equations cannot easily be mapped to just any architecture (FPGAs in particular). Therefore, if the synthesis software does not present (to the place and route software) logic in a form that is representative of a device's architectural resources, then more resources than necessary may be used, unless the place and route software can compensate for the

lack of optimization in the synthesis tool. In this scenario, the VHDL description that more closely resembles a netlist of RTL (register transfer level) components representative of device resources will synthesize to a more optimal implementation (provided that the designer reduced the logic). (3) The place and route software (fitter) may not make the appropriate choices for using device resources. If the appropriate heuristics are not embedded within the fitter software, then the fitter may not find a solution for fitting a design as appropriate even though the synthesis software has presented optimized logic and there is a theoretical solution for fitting the design. In summary, different design descriptions can produce different, but functionally equivalent, design equations resulting in different circuit implementations. Mature VHDL synthesis tools should, however, produce nearly equivalent circuits. In the pages to follow, we will point out where there is danger in freely describing logic.

Used to implement hierarchical designs, a structural methodology is also sometimes used to instantiate device-specific resources. Most synthesis tools or silicon vendors provide libraries in which you can instantiate components that represent resources that are available in a device's architecture. Most synthesis tools provide directives or attributes to indicate that such structures are not to be optimized. In other words, software will implement these portions of a design exactly as the structure of the component and how it is connected. This provides the designer with the most control over design implementation.

This type of design methodology should be avoided for several reasons, unless it is the only way to access a required feature of a device. Instantiating device-specific components eliminates the device independence of the design. (The design can remain independent only if the device-specific component has an associated architecture that can be synthesized to logic for other architectures.) Using such components should be the exception—it should not be necessary to create an entire design from such components. Creating a design from such components requires an inordinate amount of time; it provides little, if any, benefit over schematic-based design entry, and it requires that you build functions from small-device resources. Additionally, you may inadvertently create logic that is not optimal for the architecture or that is in error. It can be an arduous task to find a logic error in a convoluted netlist. This is a design methodology that VHDL seeks to avoid and is precisely why the HDL synthesis markets are growing. Computers can run through algorithms much faster than any human. Carefully constructed software algorithms can produce optimal implementations.

We return to the trade-off discussed in the introductory chapter: meeting design requirements versus controlling a design's implementation. In the chapters ahead, we'll identify how circuits will be realized in logic devices to give you an idea as to when it will be most suitable to use behavioral or structural design styles. For the most part, you will want to start out with behavioral design descriptions because they are usually the quickest and easiest way to describe designs. If after synthesizing the behavioral design description, the design implementation meets your performance and cost requirements, then you have completed your design in the shortest possible design time. If at that point you have not met your requirements, then you will want to use the timing or constraint-driven directives available from the synthesis and place and route tools to help achieve the requirements. If the desired results are still not achieved, you can introduce RTL descriptions to optimize critical portions of your design. On your next design, you will be more cognizant of what portions of your design to describe with the different coding styles. As the state of the art in VHDL synthesis improves, you'll be able to describe a larger percentage of your designs completely behaviorally with the appropriate amount of structure.

Identifiers, Data Objects, Data Types, and Attributes

Identifiers

Basic identifiers are made up of alphabetic, numeric, and/or underscore characters:

- The first character must be a letter.
- The last character cannot be an underscore.
- Two underscores in succession are not allowed.

VHDL-reserved words (see Appendix) may not be used as identifiers. Uppercase and lowercase letters are equivalent when used in identifiers. The following are equivalent:

```
txclk, Txclk, TXCLK, TxClk
```

The following are all legal identifiers:

```
tx_clk
Three_State_Enable
sel7D
HIT_1124
```

The following are *not* legal identifiers:

```
_tx_clk         -- an identifier must start with a letter
8B10B           -- an identifier must start with a letter
large#number    -- letters, digits, and underscores only
link__bar       -- two underscores in succession are not allowed
select          -- keywords (reserved words) cannot be used as identifiers
rx_clk_         -- last character cannot be an underscore
```

Data Objects

Data objects are assigned types and hold values of the specified types. Data objects belong to one of three classes: constants, signals, or variables. Data objects must be declared before they are used.

Constants

A constant holds a specific value of a type that cannot be changed within the design description, and therefore is usually assigned upon declaration. Constants are generally used to improve the readability of code; also, it may be easier to modify code if a constant name is used rather than an actual value (if a value is repeated many times). For example, the following constant may represent the width of a FIFO buffer:

```
constant width: integer := 8;
```

The identifier *width* may be used at several points in the code. However, to change the width of the FIFO requires only that the constant declaration be changed and the code recompiled (resynthesized).

Constants must be declared in a declarative region such as the package, entity, architecture, or process declarative region. A constant defined in a process declarative region is only visible to that process; one defined in an architecture is visible only to that architecture; one defined in an entity can be referenced by any architecture of that entity; one defined in a package can be referenced by any entity or architecture for which the package is used.

Signals

Signals can represent wires, and they can therefore interconnect components (ports are signals; in fact, ports can be specifically declared as signals). As wires, signals can be inputs or outputs of logic gates. We have already seen signals used for such purposes in our examples of a 4-bit comparator. Signals can also represent the state of memory elements.

```
signal count: bit_vector(3 downto 0);
```

Count may represent the current state of a counter. As such, *count* represents memory elements, or at the least, wires attached to the outputs of those memory elements (see *Figure 3-5*). Initial values may be assigned to signals, but initial values are rarely meaningful for synthesis. It is a common misconception that by assigning an initial value to a memory element that the memory element will power-up in the initialized state. For example, the following initialization is meaningless for synthesis.

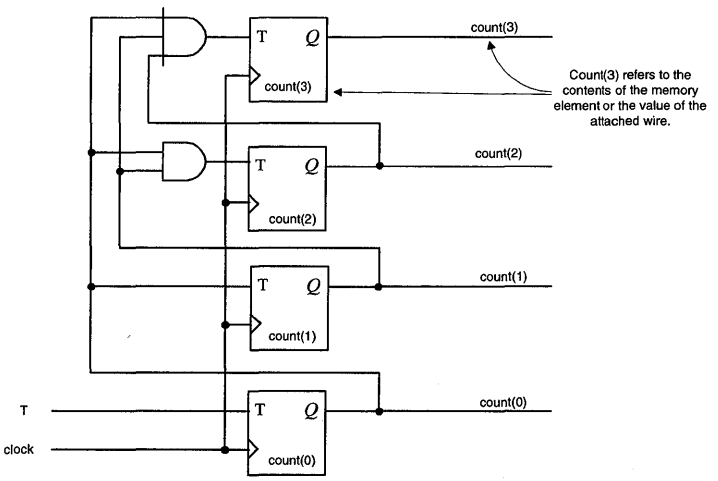


Figure 3-5 (a) A signal can refer to the memory elements or (b) the wires attached to the outputs of the memory elements.

```
signal count: bit_vector(3 downto 0) := "0101";
```

For simulation, this initialization does indeed ensure that the signal is initialized to "0101". To determine which state memory elements will power-up, refer to the data sheets of the target device. Many devices power-up with flip-flops in the reset state. If the state of a flip-flop is inverted before its associated pin, however, the pin will indicate a logic level of high. To ensure that memory elements are in the proper state, apply reset or preset during power-up. Alternatively, clock in known data after power-up.

The := operator in the signal declaration above is used for initialization of the signal. All other signal assignments within design descriptions are accomplished with the <= operator, as in

```
count <= "1010";
```

```
count <= data;
```

The := operator indicates immediate assignment. The <= operator indicates that the signal assignment is scheduled.

For designs that are to be synthesized, signals are most commonly declared in the entity and architecture declarative regions. Whereas signals declared as ports have modes, signals local to the architecture do not. Signals may also be declared in packages—packages contain declarations that can be used by other designs—as well as entities. To reference these signals, include a USE clause.

Variables

Variables are used only in processes and subprograms (functions and procedures), and must therefore be declared in the declarative region of a process or subprogram. Variables should be initialized before being used. In simulation, variables *do not* retain their values during the time that a process (or subprogram) is not active. For the value of a variable to be used outside of a process, it must be assigned to a signal of the same type. Following is an example of a variable declaration and initialization.

```
variable result: integer := 0;
```

Variable assignments are immediate, not scheduled, as with signal assignments. Refer to the discussions of *Listing 2-4* and *Listing 2-5* for a detailed explanation. The variable assignment and initialization operator is :=.

For synthesis, the most common use of variables is for index holders and the temporary storage of data.

Aliases

An alias is an alternate identifier for an existing object; it is not a new object. Referencing the alias is equivalent to referencing the original object. Making an assignment to the alias is equivalent to making an assignment to the original object. An alias is often used as a convenient method to identify a range of an array type. For example, to identify fields in an address:

```
signal stored_ad: std_logic_vector(31 downto 0);

alias top_ad: std_logic_vector(3 downto 0) is stored_ad(31 downto 28);
alias bank: std_logic_vector(3 downto 0) is stored_ad(27 downto 24);
alias row_ad: std_logic_vector(11 downto 0) is stored_ad(23 downto 12);
```

Data Types

A type has a set of values and a set of operations. Here, we discuss the categories of types and predefined types that are most useful for synthesis and some *scalar* and *composite* types.

VHDL is a *strongly-typed language*, meaning that data objects of different *base types* cannot be assigned to each other without the use of a type-conversion function (discussed in chapter 7, "Functions and Procedures"). A base type is either a type itself or the type assigned to a subtype. Thus, if *a* and *b* are both integer variables, then the following assignment

```
a := b + 2.0;
```

would illicit a compile-time error because *2.0* is a *real type* and cannot be used in an assignment to a data object of type integer.

Scalar types

Scalar types have an order that allows relational operators to be used with them. Scalar types comprise four classes: enumeration, integer, floating, and physical types.

Enumeration Types

An enumeration type is a list of values that an object of that type may hold. The list of values may be defined by you. Enumerated types are often defined for state machines:

```
type states is (idle, preamble, data, jam, nosfd, error);
```

A signal can then be declared to be of the enumerated type just defined:

```
signal current_state: states;
```

The physical implementation of an enumerated type is implementation specific. For example, *current_state* may represent a set of memory elements that hold the current state of a state machine. The state encoding can be user assigned (see chapter 5, "State Machine Design").

As a scalar type, an enumerated type is ordered. The order in which the values are listed in the type declaration defines their relation. The leftmost value is less than all other values. Each value is greater than the one to the left and less than the one to the right. Thus, if the type *sports* is defined as

```
type sports is (baseball,football, basketball, soccer, bicycling, running);
```

and *your_sport* is declared to be of type sports,

```
signal your_sport: sports;
```

then a comparison of *your_sport* to the value basketball,

```
better_than_bball <= '1' when your_sport >= basketball else '0';
```

reveals whether your preference in sports meets the author's definition of sport superiority. That is, the values basketball, soccer, bicycling, and running for *your_sport* would result in *better_than_bball* being assigned a '1.'

There are two enumeration types predefined by the IEEE 1076/1993 standard that are particularly useful for synthesis: bit and Boolean. They are defined as follows:

```
type BOOLEAN is (FALSE, TRUE);
type BIT is ('0', '1');
```

The IEEE 1164 standard defines an additional type, std_logic, and several subtypes that are consistently used as standards for both simulation and synthesis. The type std_ulogic defines a 9-value logic system. The enumeration of these values is

```
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
);
```

VHDL does not permit a signal to be driven by more than one source (driver) unless the signal is of a resolved type. A resolved type is one for which a resolution function defines a single value for a signal that is driven by more than one source. The type std_ulogic is the standard unresolved logic type. The type std_logic is defined as

```
SUBTYPE std_logic IS resolved std_ulogic;
```

It is the standard resolved logic type. The function *resolved* is in the *std_logic_1164* package found in Appendix C. The enumeration of values for the std_logic type is the same as it is for the std_ulogic. However, a signal of type std_logic may have more than one driver. If there is more than one driver for a signal, than the resolution function defined in the *std_logic_1164* package is used to determine the value of that signal. For example, suppose a signal *x* of type std_logic is driven by two signals:

```
signal a, b, x: std_logic;
...
x <= a;
x <= b;
```

If *a* and *b* are both '0,' then *x* will assume the value of '0.' If they are both '1' then *x* will assume the value of '1.' However, if *a* and *b* are opposite, then *x* assumes the value of 'U,' uninitialized. The resolution for the other combinations for *a* and *b* are defined in the *std_logic_1164* package.

The example above is useful in simulating models; however, it has little use for synthesis. Two drivers are not typically allowed in a logic device, and there isn't a standard policy for synthesizing such a construct.

The IEEE 1164 standard defines arrays of std_ulogics and std_logics as std_ulogic_vector and std_logic_vector. The standard defines several other subtypes (e.g., X01, X01Z,) operator overloading functions, conversion functions, and strength strippers. The *std_logic_11664* package will be better understood after chapter 7, "Functions and Procedures." To use these types, we simply add the following two lines to the before our entity declaration so that the scope of the declarations of the package will extend the entire entity/architecture pair:

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

Integer Types

The set of values and set of operations that characterize the type *integer* are integers and the relational and arithmetic operators (defined in the *IEEE Standard VHDL Language Reference Manual* as adding, sign, multiplying, and miscellaneous).

An integer type can be defined, as well as a data object declared, with or without specifying a range. If a range is not specified, then software tools that process VHDL must allow integers to have a

minimum range of -2,147,483,647, $-(2^{31}-1)$, to 2,147,483,647, $(2^{31}-1)$. A signal or variable that is an integer type and that is to be synthesized into logic should specify a range. For example,

```
variable a: integer range 0 to 255;
```

Floating Types

Floating point type values are used to approximate real numbers. Like integers, floating point types can be constrained. The only predefined floating type is REAL, which includes the range -1.0E38 to +1.0E38, inclusive, at a minimum. Floating point types are rarely used in code to be synthesized, but sometimes appear in computations.

Physical Types

Physical type values are used as measurement units. The only predefined physical type is TIME. Its range includes, as a minimum, the range of integers. Its primary unit is fs (femtoseconds) and is defined as follows (the range can exceed the minimum and is tool dependent).

```
TYPE time IS range -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;
```

Physical types do not carry meaning for synthesis; they are discussed here only to round out the discussion of scalar types, and because they are used frequently in simulation. We will use them in creating test benches (see chapter 9, "Creating Test Fixtures").

You could create another physical type based on another unit of measure such as meters, grams, ounces, etc. However, you can see that these units of measure have very little to do with logic design, but may bring to mind some interesting ideas of how VHDL can be used to simulate models of systems that do not represent logic circuits but some other type of "system."

Composite Types

Unlike scalar types, which do not have any elements and for which data objects of these types can only hold one value at a time, composite types define collections of values for which data objects of these types can hold multiple values at a time. Composite types consist of *array types* and *record types*.

Array Types

An object of an array type is an object consisting of multiple elements of the same type. The most commonly used array types are those predefined by the IEEE 1076 and 1164 standards:

```
type BIT_VECTOR is array (NATURAL range <>) of bit;
type STD_ULOGIC_VECTOR is array (NATURAL range <>) of std_ulogic;
type STD_LOGIC_VECTOR is array (NATURAL range <>) of std_logic;
```

These types are declared as unconstrained arrays: The number of bits, std_ulogics, or std_logics in the arrays are not specified (*range* <>); rather, the arrays are bounded only by NATURAL, the set of positive integers. These types are commonly used for buses, as in our previous code listings wherein these unconstrained arrays are constrained. For example,

```
signal a: std_logic_vector(3 downto 0);
```

However, a bus could also be defined with your own type:

```
type word is array(15 downto 0) of bit;
signal b: word;
```

Two-dimensional arrays are useful in creating a truth table:.

```
type table8x4 is array(0 to 7, 0 to 3) of bit;
constant exclusive_or: table8x4 := (
            "000_0",
            "001_1",
            "010_1",
            "011_0",
            "100_1",
            "101_0",
            "110_0",
            "111_1");
```

The entries are arranged vertically for readability, but of course do not have to be. An underline character was inserted to distinguish between sides of the "table": input and output sides. An underline character can be inserted between any two adjacent digits. Also, when using bit strings, a base specifier may be used to indicate whether the bit string is specified in binary, octal, or hexadecimal format. If the base specifier is octal, then the actual *value* of the bit string is obtained by converting the octal designator to its appropriate three-digit binary value. If the base specifier is hexadecimal, then the actual value of the bit string is obtained by converting the hexadecimal designator to its appropriate four-digit binary value. For example,

```
a  <= X"7A";
```

requires that *a* be eight bits wide, so that its value becomes "01111010".

Record Types

An object of a record type is an object that can consist of multiple elements of different types. Individual fields of a record can be referenced by element name. The following shows a record-type definition for *iocells*, objects declared as that type, and assignment of values:

```
type iocell is record
            buffer_inp: bit_vector(7 downto 0);
            enable:     bit;
            buffer_out: bit_vector(7 downto 0);
end record;
```

```
signal busa, busb, busc: iocell;
signal vec: bit_vector(7 downto 0);
busa.buffer_inp <= vec;              -- one bit_vector assigned to another
busb.buffer_inp <= busa.buffer_inp; -- assigning one field;
busb.enable <= '1';
busc <= busb;                        -- assigning entire object
```

Types and Subtypes

We have already created new types for enumeration types. Other types can also be created. Take for instance the type *byte_size*, which we define here:

```
type byte_size is range 0 to 255;
signal my_int: byte_size;
```

Byte_size is defined as a new type. Although this type is based on the integer type, it is its own type. Type checking rules often require that operands or ports be of a specific type. If an integer is expected for an operand, type *byte_size* will not be allowed. For example, suppose that signal *your_int* is defined as an integer:

```
signal your_int: integer range 0 to 255;
```

The following operation would produce a compile time (or analyzer) error:

```
if my_int = your_int then ...
```

The operands of this comparison operator are of type *byte_size* and *integer* resulting in a type mismatch.

A subtype is a type with a constraint. Subtypes are mostly used to define objects of a base type with a constraint. For example, *byte* below is defined as a type; objects can then be defined to be of this subtype. Compare

```
subtype byte is bit_vector(7 downto 0);
signal byte1, byte2: byte;
signal data1, data2: byte;
signal addr1, addr2: byte;
```

to the individual declaration of objects as constrained types:

```
signal byte1, byte2: bit_vector(7 downto 0);
signal data1, data2: bit_vector(7 downto 0);
signal addr1, addr2: bit_vector(7 downto 0);
```

Subtypes are also used to resolve a base type. A resolution function is defined by the IEEE 1164 standard for the type std_logic:

```
subtype std_logic is resolved std_ulogic;
```

Resolved is the name of a resolution function that is used to determine how multiple values from different sources (drivers) for a signal will be reduced to one value for that signal.

Four additional subtypes are declared in this standard:

```
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')
```

Whereas different types will often produce compile-time errors for type mismatches, subtypes of the same base type can legally be used interchanged. For example, if the subtype X01Z is expected for the port (a signal) of a given component, an instantiation of a component with a port of subtype UX01 will not create a compile time error. A simulation error or synthesis error will occur however, if you attempt to pass the value of 'U' (defined by type UX01 but not subtype X01Z) to the component. Also, the following comparison will not produce a compile time error:

```
signal mine: X01Z;
signal yours: UX01;
...
if yours = mine then ...
```

This is not a type mismatch because *yours* and *mine* are of the same base type.

Attributes

An attribute provides information about items such as entities, architectures, types, and signals. There are several predefined value, signal, and range attributes that are useful in synthesis.

Scalar types have value attributes. The value attributes are 'left, 'right, 'high, 'low, and 'length (pronounce ' as "tick," as in tick-left).

The attribute 'left yields the leftmost value of a type, and 'right the rightmost. The attribute 'high yields the greatest value of a type. For enumerated types, this value is the same as 'right. For integer ranges, the attribute 'high yields the greatest integer in the range. For other ranges, 'high yields the value to the right of the keyword TO or to the left of DOWNTO. The attribute 'low yields the value to the left of TO or the right of DOWNTO. The attribute 'length yields the number of elements in a constrained array. Some examples follow:

```
type count is integer range 0 to 127;
type states is (idle, decision, read, write);
type word is array(15 downto 0) of std_logic;

count'left = 0
states'left = idle
word'left = 15

count'right = 127
states'right = write
word'right = 0

count'high = 127
states'high = write
word'high = 15

count'low = 0
states'low = idle
```

```
word'low = 0

count'length = 128
states'length = 4
word'length = 16
```

An important signal attribute useful for both synthesis and simulation is the 'event attribute. This attribute yields a Boolean value of true if an event has just occurred on the signal for which the attribute is applied. It is used primarily to determine if a clock has transitioned.

A useful range attribute is the 'range attribute which yields the range of a constrained object. For example,

```
word'range = 15 downto 0
```

Below is an example that declares a signal, *my_vec*, to be a std_logic_vector constrained to the same size as another std_logic_vector, *your_vec*. Next, a loop is initiated. The loop will ascend from the lowest to the highest index of the vector:

```
signal my_vec:std_logic_vector(your_vec'range);
...
for i in my_vec'low to my_vec'high loop
            tally := tally + 1;
end loop;
```

Common Errors

There are several common errors that are worth mentioning. Identifying them early may prevent misconceptions. Some of the errors are with syntax, others with semantics. Following is a code example with several errors. See if you can identify the errors:

```
entity many_errors is port                          --line 1
            a: bit_vector(3 to 0);                  --line 2
            b: out std_logic_vector(0 to 3);        --line 3
            c: in bit_vector(6 downto 0);)          --line 4
end many_errors                                     --line 5
                                                        line 6
architecture not_so_good of many_errors             --line 7
begin                                               --line 8
my_label: process                                   --line 9
            begin                                   --line 10
            if c = x"F" then                        --line 11
                        b <= a                      --line 12
            else                                    --line 13
                        b <= '0101';                --line 14
            end if                                  --line 15
  end process;                                      --line 16
end not_so_good                                     --line 17
```

We'll take this design one line at a time because there are so many errors. The port declaration requires an "(" at the end of line 1 or beginning of line 2. In line 2, "to" should read "downto." The lack of the keyword IN to identify the mode is acceptable. If the mode is not explicitly declared, then the default of IN is assumed. Line 3 is ok. The semicolon on line 4 should appear after the second

")". The omission of a semicolon is one of the most common syntax errors. A semicolon is required at the end of line 5. There's even an error in line 6: The comment character "--" is required. Line 7 is missing the keyword "is" after the name of the entity. Line 8 is ok. The process sensitivity list is missing from line 9. Line 10 is ok. The comparison of line 11 will *always* evaluate to FALSE because x"F" represents "1111" not "001111"—six bits must be compared with six bits (this is not error, but is probably not what the designer wanted). In line 12, a signal of one type may be assigned to a signal only of the same base type, so we will have to change *a* or *b* to be of the same type. Line 13 is ok. The single quote marks (') of line 14 should be replaced with double quote marks ("). Line 15 requires a semicolon. Line 16 is ok. Line 17 requires a semicolon. The design is corrected and listed below.

```
entity many_errors is port(                          --line 1
            a: std_logic_vector(3 downto 0);         --line 2
            b: out std_logic_vector(0 to 3);         --line 3
            c: in bit_vector(6 downto 0));           --line 4
end many_errors;                                     --line 5
                                                     --line 6
architecture not_so_good of many_errors is           --line 7
begin                                                --line 8
my_label: process(c, a)                              --line 9
            begin                                    --line 10
            if c = "001111" then                     --line 11
                        b <= a;                      --line 12
            else                                     --line 13
                        b <= "0101";                 --line 14
            end if;                                  --line 15
  end process;                                       --line 16
end not_so_good;                                     --line 17
```

Exercises

1. Write an entity and architecture pair for each of the TTL devices in Table 2-1. For flip-flops, registers, or tri-state buffers, use structural descriptions with components from the pre-defined packages defined in /warp/lib/common/ directory in the *Warp* software.

2. Write an entity declaration for a 4-bit loadable counter. The counter may be enabled and asynchronously reset. The counter has three-state outputs, controlled by a common output enable.

3. Write the entity declaration for the following architecture, assuming that all signals in the architecture are ports:

```
architecture write_entity of exercise2 is
begin
mapper: process (addr) begin
            shadow_ram_sel <= '0';
            sram_sel <= '0';
            if addr >= x"0100" AND addr < x"4000" then
                        shadow_ram_sel <= '1';
            elsif addr >= x"8000" and addr < x"C000" then
                        sramsel <= '0';
            end if;

            promsel <= '0';
            if mem_mapped = '0' and bootup then
```

```
                        prom_sel <= '1';
            end if;
  end process mapper;

mem_mapped <= shadow_ram_sel OR sram_sel;
end write_entity;
```

4. Write the entity for a 2-bit equality comparator.

5. Create four architectures for the entity of exercise 4, one using an IF-THEN statement, one using a WHEN-ELSE statement, one using boolean equations, and one using instantiations of gates.

6. Using software, synthesize the designs of exercise 5. Compare the report files or physical place and route views. Are the equations the same, or do the place and routes utilize the same resources? What is the propagation delay for the selected device?

7. Which of the architectures of exercise 5 will require modification if the comparator is changed to a 4-bit comparator? Make the necessary changes and use software to synthesize the new designs and then compare results.

8. Identify errors in the following code:

```
entity 4to1_mux port(
            signal a, b, c, d: std_logic_vectors(3 downto 0);
            select: in std_logic_vector(1 downto 0);
            x: out bit_vector(3 downto 0);
end;
architecture of 4to1_mux
begin
p1: process begin
            if select = '00' then
                        x <= a;
            elsif select = '10'
                        x <= b;
            elsif select = '11'
                        x <= c;
            else
                        x <= d
            end if;
    end process;
end 4to1_mux;
```

9. What is the need for a sensitivity list to be associated with a process declaration? Can you declare a clocked process without a sensitivity list?

10. Write the VHDL code for a 4-bit wide register. Ensure that the input DATA[3:0] is stored only when the CLOCK signal is detected on its rising-edge.

11. Create a type declaration section that has 'ampere' declared as a physical type with a range from 0 to 1000. Declare 'Nanoamps' as your primary unit. Declare other units 'microamp', 'milliamp', 'amp', 'kiloamp' and 'megaamp'.

12. Extend the counter example in exercise 2, to do the following:

a) define 'count value' to an integer type with a range 0 to 15

b) Using pre-defined attributes:

1) Declare the highest value of the count to be '15'

2) Declare the lowest value of the count to be '0'

3) If the counter has reached the highest value, force the counter to be reset to its lowest value

13. Write a type declaration section that does the following:

a) declare 'MONTH' to be an enumerated type holding values of all 12 months of a year.

b) declare 'DATE' to be an integer type holding values from 1 to 31.

c) declare signals for any 5 national holidays. Detect the national holidays using information available in the type declarations.

d) declare signals for the 3 seasons. Detect the SUMMER, SPRING and WINTER seasons.

14. Repeat exercise 13 using Record types as an alternative.

15. List key differences between Signals and Variables.

4 Creating Combinational and Synchronous Logic

We begin this chapter with an example that brings together several concepts from the previous chapters and concepts that will be developed by this chapter's end. The design of *Listing 4-1* is an 8 by 9 FIFO (8-deep, 9-bit wide, first-in, first-out buffer). Two packages are used in this design: *std_logic_1164* and *std_math. Std_logic_1164* is included so that we can use the types *std_logic* and *std_logic_vector*; *std_math* is a package that we created to overload the + operator so that we can add integers to std_logic_vectors. The contents of this package are explained in chapter 7, "Functions and Procedures." Four processes are used to register the data, control the read and write pointers, and control the three-state outputs. Some of the logic is combinational, whereas other portions are synchronous. Several new constructs are introduced. Read through the code (*Listing 4-1*) to get a global idea of how the design is described, then continue to read the rest of the chapter for an explanation and elaboration of the syntax and semantics of this design.

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

entity fifo8x9 is port(
        clk, rst:                in std_logic;
        rd, wr, rdinc, wrinc:    in std_logic;
        rdptrclr, wrptrclr:      in std_logic;
        data_in:                 in std_logic_vector(8 downto 0);
        data_out:                out std_logic_vector(8 downto 0));
end fifo8x9;

architecture archfifo8x9 of fifo8x9 is
  type fifo_array is array(7 downto 0) of std_logic_vector(8 downto 0);

  signal fifo: fifo_array;
  signal wrptr, rdptr: std_logic_vector(2 downto 0);
  signal en: std_logic_vector(7 downto 0);
  signal dmuxout: std_logic_vector(8 downto 0);

begin

-- fifo register array:
reg_array: process (rst, clk)
  begin
        if rst = '1' then
                for i in 7 downto 0 loop
                        fifo(i) <= (others => '0');
                end loop;
        elsif (clk'event and clk = '1') then
                if wr = '1' then
                        for i in 7 downto 0 loop
                                if en(i) = '1' then
                                        fifo(i) <= data_in;
                                else
                                        fifo(i) <= fifo(i);
                                end if;
```

```
                        end loop;
                end if;
        end if;
  end process;

-- read pointer
read_count: process (rst, clk)
  begin
        if rst = '1' then
                rdptr <= (others => '0');
        elsif (clk'event and clk='1') then
                if rdptrclr = '1' then
                        rdptr <= (others => '0');
                elsif rdinc = '1' then
                        rdptr <= rdptr + 1;
                end if;
        end if;
  end process;

-- write pointer
write_count: process (rst, clk)
  begin
        if rst = '1' then
                wrptr <= (others => '0');
        elsif (clk'event and clk='1') then
                if wrptrclr = '1' then
                        wrptr <= (others => '0');
                elsif wrinc = '1' then
                        wrptr <= wrptr + 1;
                end if;
        end if;
  end process;

-- 8:1 output data mux
with rdptr select
        dmuxout <=      fifo(0) when "000",
                        fifo(1) when "001",
                        fifo(2) when "010",
                        fifo(3) when "011",
                        fifo(4) when "100",
                        fifo(5) when "101",
                        fifo(6) when "110",
                        fifo(7) when others;

-- FIFO register selector decoder
with wrptr select
        en <=   "00000001" when "000",
                "00000010" when "001",
                "00000100" when "010",
                "00001000" when "011",
                "00010000" when "100",
                "00100000" when "101",
                "01000000" when "110",
                "10000000" when others;
```

```
-- three-state control of outputs
three_state: process (rd, dmuxout)
  begin
        if rd = '1' then
                data_out <= dmuxout;
        else
                data_out <= (others => 'Z');
        end if;
  end process;

end archfifo8x9;
```

Listing 4-1 An 8 by 9 FIFO

Combinational Logic

Combinational logic can be described in several ways. Signals *dmuxout*, *en*, and the three-state buffers of *Listing 4-1* show combinational logic implemented using a dataflow construct (WITH-SELECT), and in terms of algorithms (IF-THEN-ELSE). In the following sections, we will examine how to write combinational logic with concurrent and sequential statements. Concurrent statements will make use of boolean equations, dataflow constructs, or component instantiations. Sequential statements will be embedded in the algorithms of processes.

Using Concurrent Statements

Concurrent signal assignments are differentiated from sequential signal assignments in that they are outside of any process. Concurrent signal assignments imply that the order of assignment is not important. The order of the concurrent signal assignments

```
b <= c;
a <= b;
h <= i;
i <= j XOR k;
```

does not matter. Signal *b* is assigned the value of *c* and is its equivalent; *a* is assigned the value of *b*, so *a* is the equivalent of both *b* and *c*. Signal *i* is assigned the exclusive-or of *j* and *k*. Signal *h* is equivalent to *i*, so it is assigned the *xor* of *j* and *k*. The fact that the assignment of *i* to *h* appears before the *xor* assignment to *i* does not affect the values of these signals because the signal assignments are concurrent. These four assignments could be placed in any order, anywhere outside of a process, and would describe the same logic.

Boolean Equations

Boolean equations can be used in concurrent signal assignments to describe combinational logic. The following example uses boolean equations to implement a four-to-one multiplexer that multiplexes 4-bit buses.

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

entity mux is port(
```

```
        a, b, c, d:     in std_logic_vector(3 downto 0);
        s:              in std_logic_vector(1 downto 0);
        x:              out std_logic_vector(3 downto 0));
end mux;

architecture archmux of mux is
begin
    x(3) <=     (a(3) and not(s(1)) and not(s(0)))
            OR  (b(3) and not(s(1)) and s(0))
            OR  (c(3) and s(1) and not(s(0)))
            OR  (d(3) and s(1) and s(0));

    x(2) <=     (a(2) and not(s(1)) and not(s(0)))
            OR  (b(2) and not(s(1)) and s(0))
            OR  (c(2) and s(1) and not(s(0)))
            OR  (d(2) and s(1) and s(0));

    x(1) <=     (a(1) and not(s(1)) and not(s(0)))
            OR  (b(1) and not(s(1)) and s(0))
            OR  (c(1) and s(1) and not(s(0)))
            OR  (d(1) and s(1) and s(0));

    x(0) <=     (a(0) and not(s(1)) and not(s(0)))
            OR  (b(0) and not(s(1)) and s(0))
            OR  (c(0) and s(1) and not(s(0)))
            OR  (d(0) and s(1) and s(0));

end archmux;
```

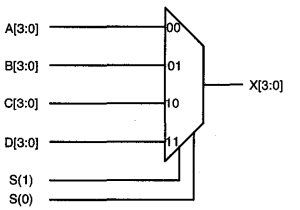


Figure 4-1 Block Diagram of mux

This description is cumbersome and provides little advantage over some proprietary, low-level languages that have been popular in the past for use with PALs. However, for those functions most easily thought of or described in boolean equations, VHDL provides for this class of description. For example,

```
entity my_design is port (
            mem_op, io_op: in bit;
            read, write             : in bit;
            memr, memw              : out bit;
```

```
            io_rd, io_wr            : out bit);
end my_design;

architecture control of my_design is
begin
            memw        <=  mem_op AND write;
            memr        <=  mem_op AND read;
            io_wr       <=  io_op AND write;
            io_rd       <=  io_op AND read;
end control;
```

Logical Operators

Logical operators are the cornerstone of Boolean equations. The logical operators AND, OR, NAND, XOR, XNOR, and NOT are predefined for the types bit, boolean, and one-dimensional arrays of bit and boolean. To use these operators (except **not**) with arrays, however, the two operands must be of the same length. The IEEE 1164 standard also defines these operators for the types std_ulogic, std_logic, and their one-dimensional arrays.

The logical operators do not have an order of precedence over each other. You may be accustomed to the precedence of operators in Boolean algebra: expressions in parentheses are evaluated first, followed by complements, AND expressions, and finally OR expressions. For example, with Boolean algebra, you expect the expression

```
A OR B AND C
```

to be evaluated as

```
A OR (B AND C)
```

However, in VHDL, one logical operator does not have precedence over another. Parentheses are required to differentiate the above expression from

```
(A OR B) AND C
```

In fact, the code

```
A OR B AND C
```

will result in a compile-time error.

Dataflow Constructs

Dataflow constructs are concurrent signal assignments that provide a level of abstraction that often enables you to write code more succinctly. The following two constructs provide selective and conditional signal assignments.

WITH-SELECT-WHEN

This selective assignment construct is best explained by using an example:

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

entity mux is port(
```

```
        a, b, c, d:     in std_logic_vector(3 downto 0);
        s:              in std_logic_vector(1 downto 0);
        x:              out std_logic_vector(3 downto 0));
end mux;

architecture archmux of mux is
begin
with s select
   x <= a when "00",
        b when "01",
        c when "10",
        d when others;
end archmux;
```

Based on the value of signal *s*, the signal *x* is assigned one of four values (*a*, *b*, *c*, or *d*). This construct enables a concise description of the four-to-one multiplexer. Four values of *s* are enumerated (00, 01, 10, and others). *Others* is specified instead of *"11"* because *s* is of type *std_logic*, and there are nine possible values for a signal of type *std_logic*. If *"11"* were specified instead of *others*, only 4 of the 81 values would be covered. For hardware and synthesis tools, *"11" is* the only other meaningful value, but the code should be made VHDL compliant so that it can also be simulated. For simulation software, there are indeed 77 other values that *s* can have. If you want, you *can* explicitly specify *"11"* as one of the values of *s*; however, *others* is still required to completely specify all possible values of *s*:

```
architecture archmux of mux is
begin
with s select
   x <= a when "00",
        b when "01",
        c when "10",
        d when "11",
        d when others;
end archmux;
```

The metalogical value "--" (two dashes) can also be used to assign the don't cares value.

A selective signal assignment allows an assignment to a signal based on mutually exclusive combinations of values of the selection signal.

WHEN-ELSE

An example of using the WHEN-ELSE conditional assignment appears below:

```
architecture archmux of mux is
begin
        x <= a when (s = "00") else
             b when (s = "01") else
             c when (s = "10") else
             d;
end archmux;
```

Whereas a selective signal assignment must specify mutually exclusive conditions for signal assignment, a conditional signal assignment does not have to. A conditional signal assignment gives highest priority to the first condition listed and priorities to subsequent conditions based on order of appearance.

This assignment above is slightly more verbose than the selective assignment construct for describing a multiplexer; however, this construct can also help you to succinctly describe constructs such as follows:

```
entity selection is port(
        a, b, c, v, w, x, y, z: in boolean;
        j: out boolean);
end selection;

architecture selection of selection is
begin

j <= w when a else
     x when b else
     y when c else
     z;

end;
```

This construct describes the logic of *Figure 4-2* results in the following equation for *j*:

```
j =   a * w
    + /a * b * x
    + /a * /b * c * y
    + /a * /b * /c * z
```

This AND (*) takes precendence over the OR (+) in the this equation and all other equations in this text that show the results of synthesis.

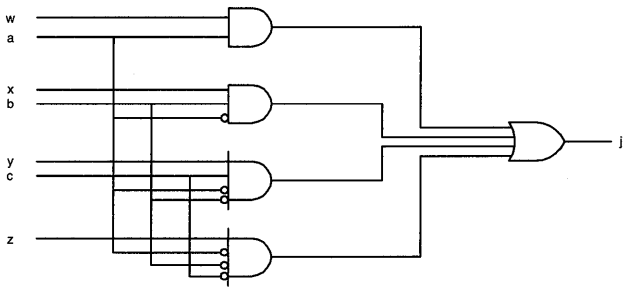


Figure 4-2 `When-else` Logic

This equation and the WHEN-ELSE construct indicates the priority that *j* is assigned the value of *w* when *a* is asserted, even if *b* or *c* is asserted. Signal *b* also holds priority over *c*. The conditions can also be expressions as in the following code fragments:

```
signal stream, instrm, oldstrm: std_logic_vector(3 downto 0);
signal state: states;
signal we: std_logic;
signal id: std_logic_vector(15 downto 0);;
...

   stream <= "0000" when (state=idle and start='0') else
             "0001" when (state=idle and start='1') else
             instrm when (state=incoming) else
             oldstrm;

   we <= '1' when (state=write and id < x"1FFF") else '0';
```

Relational Operators

Relational operators are used heavily in dataflow constructs. They are used for testing equality, inequality, or ordering. The equality and inequality operators (= and /=) are defined for all types discussed in this text. The magnitude operators (<, <=, >, and >=) are defined for scalar types or an array with a discrete range. Arrays are equivalent only if their length is equivalent and all elements of both arrays are equivalent. The result of any relational operation is Boolean (i.e., true or false).

The types of the operands in a relational operation must match. The following would produce an error because *a* is a std_logic_vector, and 123 is an integer:

```
   signal a: std_logic_vector(7 downto 0)
          ...
   if a = 123 then...
```

However, as with other operators, relational operators may be "overloaded." Overloaded operators permit you to use operators with multiple types (for which the operator is not predefined by the IEEE 1076 standard). At the time of this writing, an IEEE VHDL working group is developing a synthesis standard, which will include standard overloaded operators. These operators will allow the comparison shown above between a std_logic_vector and an integer. Operators are overloaded with functions, as you will see in the chapter on functions and procedures. These functions are contained in packages. In order to access these operators, a reference to the package must be included by way of a USE clause.

Component Instantiations

Component instantiations are also concurrent signal assignments that specify the interconnection of signals in the design. It is unlikely that you would use component instantiation to describe a 4-bit comparator as in the example below, but the code serves to illustrate that component instantiation can be used to implement combinational logic.

```
library ieee;
use work.std_logic_1164.all;
entity compare is port(
        a, b:   in std_logic_vector(3 downto 0);
        aeqb:   out std_logic);
```

```
end compare;

use work.gatespkg.all;
architecture archcompare of compare is
        signal c: std_logic_vector(3 downto 0);
begin
        x0: xor2 port map(a(0), b(0), c(0));
        x1: xor2 port map(a(1), b(1), c(1));
        x2: xor2 port map(a(2), b(2), c(2));
        x3: xor2 port map(a(3), b(3), c(3));

        n1: nor4 port map(c(0), c(1), c(2), c(3), aeqb);
end;
```

Gate components are not defined by the VHDL standard. This design requires that the gates be defined in another package (created by you, a synthesis tool vendor, or a PLD vendor). Sometimes, vendor-provided gates are technology specific (device dependent) and are provided so that you can access a particular feature. Using these components can reduce readability and eliminate the device independence of the code, unless there are behavioral descriptions of the supplied component for use in retargeting or simulating the design.

For example, a synthesis tool may provide an adder component for instantiating. This component may not have an underlying behavioral or structural description; rather, it may be recognized directly by the synthesis tool and directly mapped to the target architecture. Although this ensures that your code will produce the best possible implementation of the adder, it prevents the code from being used to target another device architecture. It also prevents the source code from being simulated, unless, of course, there are also behavioral models of the code.

Using Processes

The collection of statements that make up a process (that is, the process *itself*) constitutes a concurrent statement. If a design has multiple processes, then those processes are concurrent with respect to each other. Inside a process, however, signal assignment is sequential (from a simulation standpoint), and the order of signal assignment *does* affect how the logic gets synthesized. Processes and the sequential statements within the processes are used to describe signal assignments with algorithms. Following is a code fragment for a process that defines the dependence of signal *step* on the value of *addr*:

```
signal step: std_logic;
signal addr: std_logic_vector(7 downto 0);
...
proc_label: process (addr)
   begin
      step <= '0';
      if addr > x"0F" then
         step <= '1';
      end if;
   end process;
```

The value of *step* is assigned with an algorithm. It's given a default value of '0' at the beginning of the process. If *addr* is less than 0F hex, then *step* remains '0'; otherwise, it is assigned '1'. The

VHDL synthesis software must evaluate the entire process before creating the equation for *step*. To reinforce this idea, consider these two statements:

```
inc <= '0';
inc <= '1';
```

If these two signal assignments are concurrent signal assignments (i.e., if they are outside of a process), then one of two things can happen: (1) If *inc* is a signal of type bit, an error is issued, reporting that there is more than one driver for *inc* or (2) if *inc* is a signal of type std_logic, then *inc* is assigned the value of 'X' for simulation, and an error is produced for synthesis. (Although this could be implemented as is, allowing two drivers to drive the same wire to opposite states would damage a device.) If these two signal assignments are found in a process, then this is legal VHDL even if *inc* is of type bit. The synthesis software would simply sequence through the assignments and use the last value assigned to *inc*.

Do not confuse sequential statements with sequential (clocked) logic. Sequential statements are those statements found within a process that are evaluated sequentially before logic equations are created by the synthesis software. For example, to determine the equation for *step* in the process above, the synthesis software must realize that *step* is to be asserted for values of *addr* greater than 0F hex and deasserted for values of *addr* less than or equal to 0F hex. That is, the software cannot simply assign an equation to a signal upon its first occurrence in a process (as the software can do with concurrent signal assignments). Rather, the software must evaluate all the conditions within the process before assigning one concurrent equation that describes the logic for *step*:

```
step = addr(3) + addr(2) + addr(1) + addr(0)
```

IF-THEN-ELSE

The IF-THEN-ELSE construct is used to select a specific execution path based on a Boolean evaluation (true or false) of a condition or set of conditions. In the following example,

```
IF (condition) THEN
   do something;
ELSE
   do something different;
END IF;
```

if the *condition* specified evaluates true, the sequential statement or statements (do something) following the keyword THEN are executed. If the *condition* evaluates false, the sequential statement or statements after the END IF (do something else) are executed. The construct is closed with END IF spelled as two words. The process above, for which the value of *step* is assigned by first assigning a default value and then checking against a condition, can also be described with an IF-THEN-ELSE construct:

```
similar: process (addr)
   begin
      if addr > x"0F" then
         step <= '1';
      else
         step <= '0';
      end if;
   end process;
```

The process

```
not_similar: process (addr)
   begin
      if addr > x"0F" then
         step <= '1';
      end if;
   end process;
```

does not describe the same logic because neither a default value for *step* nor an ELSE condition is specified. The process *not_similar* above implies that *step* should retain its value (this is referred to as *implied memory*) if *addr* is less than or equal to 0F hex. Thus, once asserted, *step* will remain forever asserted as shown in *Figure 4-3* and defined by the following equation:

```
step = addr(3) * addr(2) * addr(1) * addr(0)
     + step
```

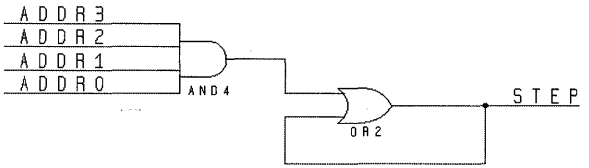


Figure 4-3 Implied memory in an IF-THEN construct

If you do not want the value of *step* to be "remembered," then be sure to include a default value or complete the IF-THEN with an ELSE.

The IF-THEN-ELSE can be expanded further to include an ELSIF (spelled without a second E and as one word) statement to allow for further conditions to be specified and prioritized. The syntax for this operation is

```
IF (condition1) THEN
   do something;
ELSIF (condition2) THEN
   do something different;
ELSE
   do something completely different:
END IF;
```

For each signal *x* that is assigned a *value* based on a *condition*, synthesis will produce an equation:

```
x =   condition1 * value1
    + /condition1 * condition2 * value2
    + /condition1 + /condition2 * condition3 * value3
    + ...
```

The IF-THEN-ELSIF-ELSE construct and the equation above clearly show that for x to be assigned *value3*, not only does *condition3* have to be true, but also *condition1* and *condition2* must be false. For x to be assigned *value1*, only *condition1* need be true, regardless of the evaluation of *condition2* and *condition3*. This indicates a clear order of precedence among the conditions.

The 4-bit wide four-to-one multiplexer can be described with an IF-THEN-ELSIF-ELSE construct as

```
architecture archmux of mux is
begin
mux4_1: process (a, b, c, d, s)
        begin
                if s = "00" then
                        x <= a;
                elsif s = "01" then
                        x <= b;
                elsif s = "10" then
                        x <= c;
                else
                        x <= d;
                end if;
        end process mux4_1;
end archmux;
```

Signals *a, b, c, d,* and *s* are included in the process sensitivity list because a change in any one of them should cause a change in (or evaluation of) x. If s were omitted from the sensitivity list, then x would not change with a change in s; only changes in *a, b, c,* or *d* would lead to a change in x, and this would not describe a multiplexer. The design equations produced by synthesizing this design description, or any of the other descriptions of the multiplexer found in this chapter, are the same for each bit of x: $x = \overline{s}_1\overline{s}_0 a + \overline{s}_1 s_0 b + s_1\overline{s}_0 c + s_1 s_0 d$. If mapped to a PLD-type architecture, each bit of x can easily be implemented with one macrocell and four product terms. If mapped to an FPGA, it is up to the synthesis and optimization software to optimally map the equation to the device architecture. For device architectures with multiplexers, this design should fit nicely.

The following code decodes a 16-bit address to address system memory elements and peripheral ports. This design uses IF-THEN-ELSIF constructs and relational operators to identify areas of

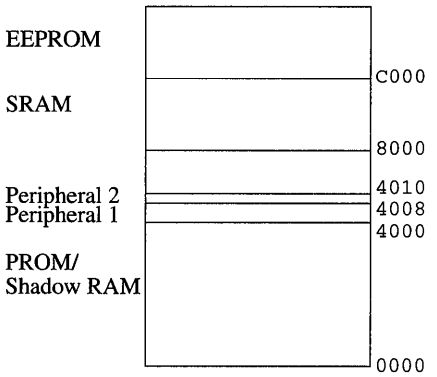


Figure 4-4 Memory Map

memory (see *Figure 4-4*)in order to assert the correct signal. Because the conditions are mutually exclusive, only one product term per output signal is required, as verified by the design equations that follow the code.

```
library ieee;
use ieee.std_logic_1164.all;
entity decode is port(
        address:        in std_logic_vector(15 downto 3);
        valid, boot_up: in std_logic;
        sram, prom, eeprom, shadow,
        periph1, periph2: out std_logic);
end decode;
architecture mem_decode of decode is
begin
mapper: process (address, valid, boot_up) begin
        shadow  <= '0';
        prom    <= '0';
        periph1 <= '0';
        periph2 <= '0';
        sram    <= '0';
        eeprom  <= '0';
        if valid = '1' then
                if address >=x"0000" and address < x"4000" then
                        if boot_up = '1' then
                                shadow <= '1';
                        else
                                prom <= '1';
                        end if;
                elsif address >=x"4000" and address < x"4008" then
                        periph1 <= '1';
                elsif address >=x"4008" and address < x"4010" then
                        periph2 <= '1';
                elsif address >=x"8000" and address < x"C000" then
```

```
                sram <= '1';
            elsif address >= x"C000" then
                eeprom <= '1';
            end if;
        end if;
end process;
end mem_decode;
```

The design equations indicate the need for one product term for each output. This maps well to a CPLD architecture. The equations for *periph1* and *periph2* contain as many as 14 literals. CPLD logic blocks typically have many more inputs than FPGA logic cells, and so there are still additional inputs to a logic block that can be used for other expressions. In an FPGA, these signals would likely require more than one logic cell (and therefore more than one level of logic) because of the wide fan-in and large AND gate required. The signals *valid*, *address(15)*, and *address(14)* would each have fanouts of six.

```
sram =
      valid * address_15 * /address_14
prom =
      valid * /address_15 * /address_14 * /boot_up
eeprom =
      valid * address_15 * address_14
shadow =
      valid * /address_15 * /address_14 * boot_up
periph1 =
      valid * /address_15 * address_14 * /address_13 * /address_12 *
      /address_11 * /address_10 * /address_9 * /address_8 * /address_7*
      /address_6 * /address_5 * /address_4 * /address_3
periph2 =
      valid * /address_15 * address_14 * /address_13 * /address_12 *
      /address_11 * /address_10 * /address_9 * /address_8 * /address_7*
      /address_6 * /address_5 * /address_4 * address_3
```

Because the address ranges for the different memory selections are mutually exclusive, six separate IF-THEN constructs could have, and perhaps should have, been used. Using separate constructs ensures that a condition for one signal does not create additional logic to exclude that condition for the next signal.

CASE-WHEN

Case statements are often used to perform decode operations or conditional tests on buses and other sets of input values. They can provide a more succinct description of the signal assignments and the conditions for assignment than a series of nested statements or logic gates. The following is an address decoder:

```
library ieee;
use ieee.std_logic_1164.all;
ENTITY test_case IS
   PORT (address : IN STD_LOGIC_VECTOR(2 DOWNTO 0);
      decode: OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
END test_case;
```

```
ARCHITECTURE design OF test_case IS
BEGIN
PROCESS (address)
   BEGIN
   CASE address IS
      WHEN "001" =>  decode  <= X"11";
      WHEN "111" =>  decode  <= X"42";
      WHEN "010" =>  decode  <= X"44";
      WHEN "101" =>  decode  <= X"88";
      WHEN OTHERS => decode  <= X"00";
   END CASE;
   END PROCESS;
END design;
```

The CASE-WHEN construct describes how *decode* is driven based on the value of the input address. *When others* is used to completely define the behavior of the output *decode* for all possible values of *address*. *When others* covers the address input combinations "000," "011," "100," and "110," as well as all of the metalogical values. The equations generated by synthesizing this design are

```
decoded_7 =
       address_2 * /address_1 * address_0
 decoded_5 =
       GND
 decoded_4 =
       /address_2 * /address_1 * address_0
 decoded_3 =
       address_2 * /address_1 * address_0
 decoded_2 =
       /address_2 * address_1 * /address_0
 decoded_1 =
       address_2 * address_1 * address_0
 decoded_0 =
       /address_2 * /address_1 * address_0
 decoded_6 =
       /address_2 * address_1 * /address_0
     + address_2 * address_1 * address_0
```

Many of these equations turn out to have common product terms. The fourth and zeroth bits are equivalent, as well as the third and the seventh. The first and sixth bits share a common product term, as well as the second and sixth. In a CPLD, these common product terms can be implemented with product terms that may be shared. (Some architectures produce an incremental delay when shared product terms are used, in which case, depending upon the performance requirements of this design, you may or may not want to have this use shared product terms.) An implementation in an FPGA could take advantage of equivalent signals—the output of one logic cell could drive two I/O cells. The shared gates between the first, second, and sixth bits would probably not provide an advantage in most FPGAs because an additional level of logic would likely be required; however, it does fit into the pASIC380 logic cell as shown in *Figure 4-5*. The equations for decode(1) and decode(2) are

placed on the select lines, each of which has its own unique output. If either select line is asserted, then the output for decode(6) should also be asserted.

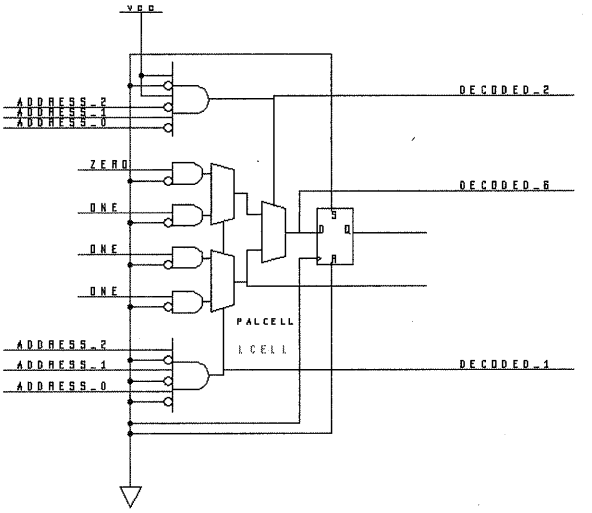


Figure 4-5 Implementing three equations simultaneously in one logic cell

Synchronous Logic

Programmable logic devices lend themselves well to synchronous applications. Most device architectures have blocks of combinational logic connected to the inputs of flip-flops as the basic building block. This section will show you how to write VHDL for synchronous logic, both with behavioral and structural descriptions.

The following code implements a simple D-type flip-flop (*Figure 4-6*):

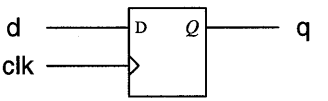


Figure 4-6 Block Diagram of DFF

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_logic is port (
   d, clk : in std_logic;
   q      : out std_logic
);
end dff_logic;
```

```
architecture example of dff_logic is
begin
   process (clk) begin
      if (clk'event and clk = '1') then
         q <= d;
      end if;
   end process;
end example;
```

This process is only sensitive to changes in *clk*. A VHDL simulator activates this process only when *clk* transitions; a transition in *d* does not cause a sequencing of this process.

The *if clk'event* condition is only true when there is a change in value—i.e., an event—of the signal *clk* ("tick event," *'event*, is a VHDL attribute that when combined with a signal forms a boolean expression that indicates when the signal transitions). Because this change in value can be either from a 0 to a 1 (a rising edge) or a 1 to a 0 (a falling edge), the additional condition *clk = '1'* is used to define a rising-edge event. The assignments inside of the *if* statement only occur when there is a change in the state of *clk* from a 0 to a 1—a synchronous event, allowing synthesis software to infer from the code a positive edge-triggered flip-flop. You can make this occur on the falling edge of the clock instead of the rising edge simply by changing

```
   if (clk'event and clk = '1')
```

to

```
   if  (clk'event and clk = '0')
```

Also, you can describe a level-sensitive latch (see *Figure 4-7*) instead of an edge-triggered flip-flop.

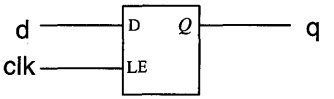


Figure 4-7 Block Diagram of `D-Latch`

To do this, all you have to do is take away the *clk'event* condition, leaving:

```
if (clk = '1') then
   q <= d;
end if;
```

In this case, whenever *clk* is at a logic high level, the output *q* is assigned the value of input *d*, thereby describing a latch.

In all of these cases (the rising-edge-triggered flip-flop, the falling-edge-triggered flip-flop, and the level-sensitive latch), there is not an ELSE condition indicating what assignments to make when the *if* conditions are not met. Without an ELSE, there is implied memory (i.e., *q* will keep its value), which is consistent with the operation of a flip-flop. In other words, writing:

```
if (clk'event and clk = '1') then
   q <= d;
```

```
   end if;
```

has the same effect as writing:

```
if (clk'event and clk = '1') then
   q <= d;
else
   q <= q;
end if;
```

This is exactly how a D-type flip-flop, and other synchronous logic, should operate. If there is a rising edge on the clock line, then the flip-flop output will get a new value based on the flip-flop input. If not, the flip-flop output stays the same. In fact, most synthesis tools will not handle an ELSE expression after (clk'event and clk='1') because it describes logic for which the implementation is ambiguous. For example, it is unclear how the following description should be synthesized:

```
if (clk'event and clk = '1') then
   q <= d;
else
   q <= a;
end if;
```

The following two examples show how a T-type flip-flop (a toggle flip-flop) and an 8-bit register can be described. First is the T-type flip-flop shown in *Figure 4-8*:

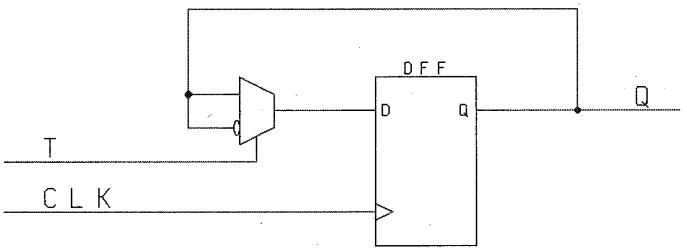


Figure 4-8 Implementation of a T-type flip-flop in a device that only has D-type flip-flops

```
library ieee;
use ieee.std_logic_1164.all;
entity tff_logic is port (
   t, clk : in std_logic;
   q      : inout std_logic
);
end dff_logic;

architecture t_example of dff_logic is
begin
   process (clk) begin
      if (clk'event and clk = '1') then
```

```
            if (t = '1') then
               q <= not(q);
            else
               q <= q;
            end if;
         end if;
      end process;
   end t_example;
```

All signal assignments in this process that occur after *if (clk'event and clk='1')* are synchronous to the signal *clk*. The signal assignments in the process above (*q <= not(q)* and *q <= q*) are synchronous to the clock. The first signal assignment, which occurs if *t* is asserted, indicates that on the rising edge of the clock, *q* will be assigned the opposite of its current value. The second signal assignment, which occurs if *t* is deasserted, indicates that on the rising edge of the clock, *q* will retain its value. Looking at *Figure 4-8*, you can see that combinational logic is described in this process (the multiplexer); however, all signal assignments (*q*) are synchronous. (This implementation is device specific. A 22V10, for example, does not have multiplexers, so it obviously would not be implemented as shown—it would be implemented with a sum of product terms. The CY7C371 device has a macrocell for which the register can be configured as a TFF, and so would not require any additional resources.)

Here is the 8-bit register:

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_logic is port (
   d   : in std_logic_vector(0 to 7);
   clk : in std_logic;
   q   : out std_logic_vector(0 to 7)
);
end reg_logic;

architecture r_example of reg_logic is
begin
   process (clk) begin
      if (clk'event and clk = '1') then
         q <= d;
      end if;
   end process;
end example;
```

You can also describe the behavior of this registered circuit by using the WAIT UNTIL construct instead of the *if (clk'event and clk='1')* construct:

```
architecture example2 of dff_logic is
begin
   process begin
      wait until (clk = '1');
         q  <=  d;
      end process;
end example2;
```

This process does not use a sensitivity list, but begins with a WAIT statement. A process that has a WAIT statement cannot have a sensitivity list (the WAIT statement implicitly defines the sensitivity list), and the WAIT statement must be the first statement in the process. Because of this, synchronous logic described with a WAIT statement cannot be asynchronously reset, as will be shown below.

If you interpret the code fragment above in terms of simulation, then you will see that this process is inactive until the condition following the WAIT UNTIL statement is true. Once true, the signal assignments that follow the WAIT statement are made, after which the process once again waits for the clock to be asserted (even if it is still presently asserted). So, in this case, once the *clk* signal becomes equal to 1 (i.e., on the rising edge of *clk*), *q* will be assigned the value of *d*, thereby describing a D-type flip-flop.

Resets and Synchronous Logic

None of the examples above make reference to resets or initial conditions. The VHDL standard does not require you to reset or initialize a circuit. The standard specifies that for simulation, unless a signal is explicitly initialized, it gets initialized to the 'LEFT value of its type. So a signal of type std_logic will get initialized to 'U', the uninitialized state, and a bit will get initialized to '0'. In the hardware world, however, this is not always true—not all devices power up in the reset state, and the uninitialized state is physically meaningless. Furthermore, you may wish to have global and local resets to place the logic in a known state. You can describe resets and presets (as shown in *Figure 4-9*) with simple modifications to the code, as shown here:

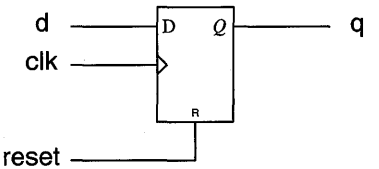


Figure 4-9 Block Diagram of DFF with Asynchronous Reset

```
architecture rexample of dff_logic is
begin
   process (clk, reset) begin
      if reset = '1' then
         q <= '0';
      elsif (clk'event and clk = '1') then
         q <= d;
      end if;
   end process;
end rexample;
```

The sensitivity list indicates that this process is sensitive to changes in *clk* and *reset*. A transition in either of these signals will cause a simulator to sequence through the process. Sequencing through the code, you'll find that this code accurately describes an asynchronously resettable D-type flip-flop: The process is activated only by a change in *clk* or *reset*. Upon activation of the process, if *reset* is asserted, then signal *q* will be assigned '0', regardless of the value of *clk*. Otherwise, if *reset* is not

asserted and the clock transition is a rising edge event, then the signal *q* will be assigned the value of signal *d*. This process template causes synthesis software to infer an asynchronous reset.

To describe a preset instead of a reset, you can simply write

```
if (preset = '1') then
   q < '1';
end if;
```

instead of:

```
if (reset = '1')
   then q <= '0'
end if;
```

You can also reset (or preset) your flip-flops synchronously by putting the reset (or preset) condition inside the logic controlled by the clock. For example,

```
architecture sync_rexample of dff_logic is
begin
   process (clk) begin
      if (clk'event and clk = '1') then
         if (reset = '1') then
            q <= '0';
         else
            q <= d;
         end if;
      end if;
   end process;
end sync_rexample;
```

This VHDL code describes a process that is sensitive only to changes in the clock, and synthesis produces a D-type flip-flop that is synchronously reset whenever the *reset* signal is asserted and is sampled by the rising edge of the clock. Because most flip-flops in PLDs do not have synchronous resets or sets (the 22V10, with a synchronous set, is a notable exception), implementing synchronous resets and sets requires using additional logic resources (product terms) (see *Figure 4-10*).

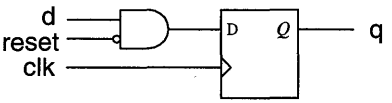


Figure 4-10 Additional logic resources are usually required for synchronous resets and sets.

You can also describe a combination synchronous/asynchronous reset and/or preset in VHDL. Suppose, for example, that you want an 8-bit register to be asynchronously reset to all 0's whenever the signal *reset* is asserted, but you want it to be synchronously preset to all 1's whenever the signal

init is asserted and sampled by the rising edge of the clock. The VHDL code to implement this function is shown in *Listing 4-2*:

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_logic is port (
   d                : in std_logic_vector(0 to 7);
   reset, init, clk : in std_logic;
   q                : out std_logic_vector(0 to 7)
);
end reg_logic;

architecture fancy_example of reg_logic is
begin
   process (clk, reset) begin
      if (reset = '1') then
         q <= b"00000000";
      elsif (clk'event and clk = '1') then
         if (init = '1') then
            q <= b"11111111";
         else
            q <= d;
         end if;
      end if;
   end process;
end fancy_example;
```

Listing 4-2 An 8-bit register with asynchronous reset and synchronous initialization

Arithmetic Operators

We will digress for a moment to briefly discuss arithmetic operators (adding, subtracting, concatenation, sign, multiplying, dividing, modulus, remainder, and absolute value). The arithmetic operators most commonly used in designs created for synthesis are addition and subtraction. All are defined for the types integer and floating.

The native VHDL + operator is not defined for the types bit or std_logic. Therefore, the following code can be written for use with synthesis:

```
ENTITY myadd IS PORT (
                        a, b : IN INTEGER RANGE 0 TO 3;
                        sum: OUT INTEGER RANGE 0 TO 6);
END myadd;

ARCHITECTURE archmyadd OF myadd IS
BEGIN
                        sum <=  a + b;
END archmyadd;
```

Here, the result of the addition is assigned to a data object of type integer. Although some synthesis tools handle this design, internally converting the integers to bits or std_logics, using integers as ports poses a problem: The same vectors used to simulate the source code cannot be used to simulate

the post-fit simulation model (see chapter 9, "Creating Test Fixtures"). You supply vectors with integers for the source code, whereas you need to supply vectors of type bit or std_logic to the back-annotated model.

Most synthesis tools provide overloaded operators (or you can write your own—see chapter 7, "Functions and Procedures") that allow the + operator, or any other operator, to be used with different types other than those predefined. For example, it is desirable to be able to add an integer to either a bit_vector or std_logic_vector.

The synthesis of arithmetic operators will be discussed in chapter 8, "Synthesis to Final Design Implementation". For now, we will simply use these operators to create components such as counters.

Asynchronous Resets and Presets

Our next code listing describes an 8-bit counter. This counter has one asynchronous signal, which places the counter at the value "00111010". The counter is also synchronously loadable and enableable.

```
library ieee;
use ieee.std_logic_1164.all;
entity cnt8 is port(
        txclk, grst:    in std_logic;
        enable, load:   in std_logic;
        data:           in std_logic_vector(7 downto 0);
        cnt:            inout std_logic_vector(7 downto 0));
end cnt8;

library work;
use work.std_math.all;
architecture archcnt8 of cnt8 is
begin
count: process (grst, txclk)
  begin
        if grst = '1' then
                cnt <= "00111010";
        elsif (txclk'event and txclk='1') then
                if load = '1' then
                        cnt <= data;
                elsif enable = '1' then
                        cnt <= cnt + 1;
                end if;
        end if;
  end process count;
end archcnt8;
```

Listing 4-3 An 8-bit counter with one asynchronous signal to reset some flip-flops and set other flip-flops

The process *count* is sensitive to transitions in *grst* and *txclk*. If *grst* is asserted, then some of the *cnt* flip-flops are asynchronously reset while others are asynchronously preset (see *Figure 4-11*). On the rising edge of *txclk,* the *cnt* registers are loaded if *load* is asserted, incremented if *enable* is asserted, or remain the same if neither *load* nor *enable* is asserted. The lack of an ELSE after *elsif enable*

implies that *cnt* will retain its value if neither of the previous conditions (*load* or *enable*) is true. Alternatively, you can explicitly include "else q <= q" (*Figure 4-11*).

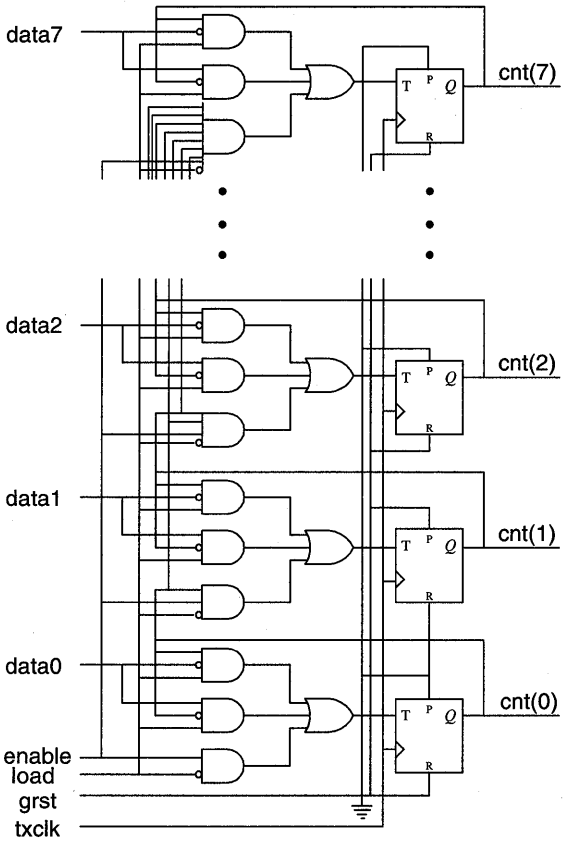


Figure 4-11 Schematic of the 8-bit counter of *Listing 4-3*, as implemented in the FLASH370 architecture

Occasionally, a design may require two asynchronous signals: both a reset signal and a preset. How are both a reset and a preset defined? *Listing 4-4* is our suggestion:

```
library ieee;
use ieee.std_logic_1164.all;
entity cnt8 is port(
        txclk, grst, gpst:      in std_logic;
        enable, load:           in std_logic;
        data:                   in std_logic_vector(7 downto 0);
        cnt:                    inout std_logic_vector(7 downto 0));
end cnt8;
```

```
library work;
use work.std_math.all;
architecture archcnt8 of cnt8 is
begin
count: process (grst, gpst, txclk)
  begin
        if grst = '1' then
                cnt <= (others => '0');
        elsif gpst = '1' then
                cnt <= (others => '1');
        elsif (txclk'event and txclk='1') then
                if load = '1' then
                        cnt <= data;
                elsif enable = '1' then
                        cnt <= cnt + 1;
                end if;
        end if;
  end process count;
end archcnt8;
```

Listing 4-4 A counter with asynchronous reset and preset.

This process is sensitive to changes in *grst, gpst,* and *txclk. Grst* and *gpst* are both used to asynchronously assign values to the *cnt* registers. The reset/preset combination of *Listing 4-4* poses a synthesis issue: As discussed earlier in the chapter, the IF-ELSIF construct implies a precedence—that *cnt* should be assigned the value of all 1's (others => '1') when *gpst* is asserted AND *grst* is not asserted. The logic in *Figure 4-12* assures that the counter will not be preset unless the reset signal is also low. This was not the intended effect. Some synthesis tools recognize that this is not the intended effect and that flip-flops by design are either reset- or preset-dominant. Therefore, depending on the synthesis policy of your software tool, the code of *Listing 4-4* produces the logic of either *Figure 4-12* or *Figure 4-13*. Many CPLDs with product term resets and presets are able to fit either implementation. Likewise, most FPGAs have the resources to implement product term resets and presets. However, while most FPGAs are set up to provide a high-performance, global, or near global, reset or preset, most do not have the resources to provide a high-performance product term reset or preset, in which case the implementation of *Figure 4-13* is preferred.

Occasionally, you may want a series of registers to be reset to all 0's when asynchronously reset, and preset to a predetermined value (some 0's, some 1's) when asynchronously preset. *Listing 4-5* is an example:

```
library ieee;
use ieee.std_logic_1164.all;
entity cnt8 is port(
        txclk, grst, gpst:      in std_logic;
        enable, load:           in std_logic;
        data:                   in std_logic_vector(7 downto 0);
        cnt:                    inout std_logic_vector(7 downto 0));
end cnt8;

library work;
use work.std_math.all;
```

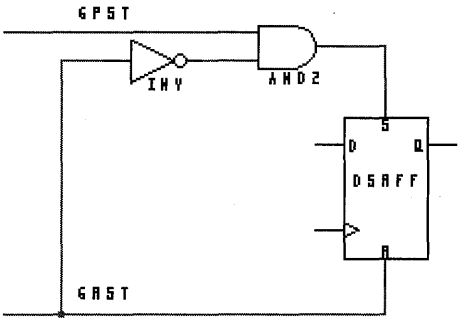


Figure 4-12 Logic assures that the reset is dominant.

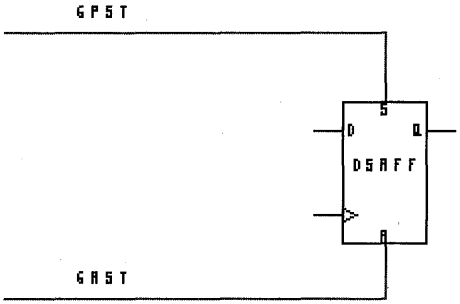


Figure 4-13 Synthesis result of *Listing 4-4*, assuming that the reset is dominant

```
architecture archcnt8 of cnt8 is
begin
count: process (grst, gpst, txclk)
  begin
        if grst = '1' then
                cnt <= "00000000";
        elsif gpst = '1' then
```

```
                cnt <= "00111010";
        elsif (txclk'event and txclk='1') then
                if load = '1' then
                        cnt <= data;
                elsif enable = '1' then
                        cnt <= cnt + 1;
                end if;
        end if;
  end process count;
end archcnt8;
```

Listing 4-5 A counter with asynchronous preset and product-term reset

Assuming reset dominance, synthesis will cause *cnt(7)* to be asynchronously set to '0' if *grst* OR *gpst* is asserted, provided that the device supports an OR term asynchronous reset. See *Figure 4-8*.

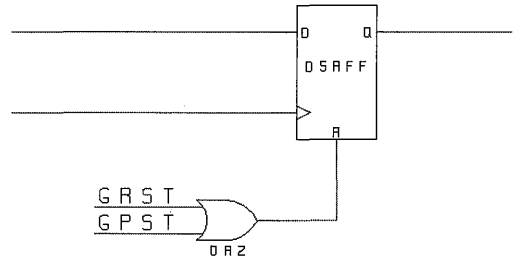


Figure 4-14 A flip-flop with an OR term asynchronous reset

All of the examples showing resets and presets in this chapter use the *if (signal_name'event and signal_name='1')* construct to describe synchronous logic. You can also use the *wait until* statement to describe synchronous logic with resets and presets, but these resets and presets must be synchronous. This is because the WAIT statement must be the first statement in the process, as mentioned above. Since no asynchronous statements can come before it, and since any statements that come after it are synchronous, resets and presets used with a WAIT statement are always synchronous.

Instantiation of Synchronous Logic Components

As with combinational logic, synchronous logic may be described in concurrent statements through component instantiation. Components may be created by you, a synthesis tool vendor, or a PLD vendor. Two advantages of instantiating components may exist: (1) A component that is repeated throughout the design need not be described multiple times—it can be described once and then instantiated and (2) vendor-supplied implementations of components may provide better implementations than those generated through synthesis of behavioral VHDL (chapter 8, "Synthesis to Final Design Implementation") discusses these issues in detail). For example, rather than using

behavioral code to redescribe a D-type flip-flop Each time you require the use of one, you can describe it once as a component and then instantiate it:

```
label: dsrff port map (d, s, r, clk, q);
```

Where *d* is the D input to the flip-flop; *s* is the preset input; *r* is the reset input; *clk* is the clock input; and *q* is the output. The following component may be a vendor-supplied component that the synthesis software recognizes in order that it can produce an optimal implementation:

```
mycount: cnt16 port map (cnt, r, en, clk);
```

Three-State Buffers and Bidirectional Signals

Most programmable logic devices have three-state outputs and I/O pins that can be used bidirectionally. Output buffers are placed in a high-Z (high-impedance) state in order not to drive a shared bus at the wrong time (i.e, to avoid bus contention), or so that bidirectional pins may be driven by off-chip signals. Additionally, some devices have internal three-state buffers. We will show how to describe three-state and bidirectional signals using both behavioral descriptions and structural instantiations of three-state and bidirectional I/O components.

Behavioral Three-States and Bidirectionals

The values that a three-state signal can have are '0', '1', and 'Z', all of which are supported by the type *std_logic*. We will modify the 8-bit counter example to have three-state outputs, and then discuss what code changes were necessary. *Listing 4-6* shows the modifications:

```
library ieee;
use ieee.std_logic_1164.all;
entity cnt8 is port(
    txclk, grst:    in std_logic;
    enable, load:   in std_logic;
    oe:             in std_logic;
    data:           in std_logic_vector(7 downto 0);
    cnt_out:        inout std_logic_vector(7 downto 0));
end cnt8;

library work;
use work.std_math.all;
architecture archcnt8 of cnt8 is
  signal cnt: std_logic_vector(7 downto 0);
begin
count: process (grst, txclk)
  begin
    if grst = '1' then
       cnt <= "00111010";
    elsif (txclk'event and txclk='1') then
        if load = '1' then
            cnt <= data;
        elsif enable = '1' then
            cnt <= cnt + 1;
        end if;
    end if;
  end process count;
```

```
oes: process (oe, cnt)
  begin
    if oe = '0' then
        cnt_out <= (others => 'Z');
    else
        cnt_out <= cnt;
    end if;
  end process oes;

end archcnt8;
```

Listing 4-6 A counter with three-state outputs

The process labeled *oes* is used to describe the three-state outputs for the counter. This process simply indicates that if *oe* is asserted, then the value of *cnt* is assigned to *cnt_out* (the output port of this design), and if *oe* is not asserted, then the outputs of this design are placed in the high-impedance state. The process is sensitive to changes in either *oe* or *cnt,* because a change in either signal causes a change in *cnt_out.* Two additional signals are used in this design compared to the design of *Listing 4-3. Oe* was added as the three-state control and *cnt_out* was added as the output port. The signal *cnt* was changed from a port of this design to a signal local to the architecture. This description is consistent with the functionality of a three-state buffer (*Figure 4-8*).

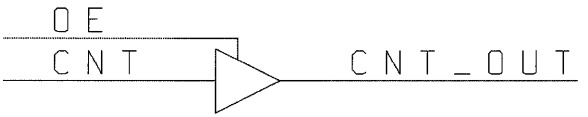


Figure 4-15 A three-state buffer

The three-state control can also be described with a WHEN-ELSE construct. In *Listing 4-7,* we add an additional output, *collision,* which is asserted when *load* and *enable* are asserted, provided that its output is enabled.

```
library ieee;
use ieee.std_logic_1164.all;
entity cnt8 is port(
    txclk, grst:    in std_logic;
    enable, load:   in std_logic;
    oe:             in std_logic;
    data:           in std_logic_vector(7 downto 0);
    collision:      out std_logic;
    cnt_out:        inout std_logic_vector(7 downto 0));
end cnt8;
```

```
library work;
use work.std_math.all;
architecture archcnt8 of cnt8 is
  signal cnt: std_logic_vector;
begin
count: process (grst, txclk)
  begin
    if grst = '1' then
        cnt <= "00111010";
    elsif (txclk'event and txclk='1') then
        if load = '1' then
            cnt <= data;
        elsif enable = '1' then
            cnt <= cnt + 1;
        end if;
    end if;
  end process count;

cnt_out <= (others => 'Z') when oe = '0' else cnt;
collision <= (enable AND load) when oe = '1' else 'Z';

end archcnt8;
```

Listing 4-7 Three-state outputs defined with a WHEN-ELSE construct

Bidirectional signals are described with little modification of *Listing 4-6* or *Listing 4-7*. Here, in *Lisiting 2-8,* the counter is loaded with the current value on the pins associated with the counter outputs, meaning that the value loaded when *load* is asserted may be the counter's previous value or a value driven from another device, depending upon the state of the output enable:

```
library ieee;
use ieee.std_logic_1164.all;
entity cnt8 is port(
    txclk, grst:    in std_logic;
    enable, load:   in std_logic;
    oe:         in std_logic;
    cnt_out:    inout std_logic_vector(7 downto 0));
end cnt8;

library work;
use work.std_math.all;
architecture archcnt8 of cnt8 is
  signal cnt: std_logic_vector(std_logic_vector);
begin
count: process (grst, txclk)
  begin
    if grst = '1' then
        cnt <= "00111010";
    elsif (txclk'event and txclk='1') then
        if load = '1' then
            cnt <= cnt_out;    -- cnt is now loaded from the cnt_out port
```

```
        elsif enable = '1' then
            cnt <= cnt + 1;
        end if;
    end if;
  end process count;

oes: process (oe, cnt)
  begin
    if oe = '0' then
        cnt_out <= (others => 'Z');
    else
        cnt_out <= cnt;
    end if;
  end process oes;

end archcnt8;
```

Listing 4-8 I/Os used bidirectionally

If you compare these listings closely, you'll find that the greatest difference is in the assignment of *cnt* when *load* is asserted. Other subtleties include the fact that *cnt_out* **must** be of mode INOUT in this example, whereas it can be of mode BUFFER in the other listings. The remaining difference is that *data* is not a required signal for this listing, for obvious reasons.

Here is an example for which the output enable of a three-state buffer is implicitly defined:

```
multiplexer: process (row_addr, col_addr, present_state)
   begin
      if (present_state = row_address or present_state = ras_assert) then
            dram <= row_addr;
      elsif (present_state = col_address or present_state= cas_assert) then
            dram <= col_addr;
      else
            dram <= (others => 'Z');
      end if;
   end process;
```

The three-state buffers for the signal dram are enabled if the value of *present_state* is *row_address*, *ras_assert*, *col_address*, or *cas_assert*. The output buffers are not asserted for any other values of the *present_state*.

Structural Three-States and Bidirectionals

IEEE 1076-compliant VHDL-synthesis vendors that have been slow to support the IEEE 1164 standard may not support the behavioral description of three-state or bidirectional components. Instead, their compilers will likely support vendor-supplied components for these purposes. In this case, you simply instantiate the component (here, we show the component name as *threestate*):

```
u0:threestate port map (cnt(0), oe, cnt_out(0));
```

Even if the compiler does support behavioral three-states, you may prefer to create a *threestate* component yourself and instantiate it if you find this method to be more clear.

For-Generate

If you use the *threestate* component to implement the three-state buffers for a 32-bit bus, it is cumbersome to instantiate the component 32 separate times. The FOR-GENERATE statement helps in this case:

```
gen_label:for i in 0 to 31 generate
            inst_label:threestate port map (value(i), read, value_out(i));
          end generate;
```

A generation scheme is implemented in the concurrent statement portion of an architecture, not within a process. A generation scheme can also include conditional instantiations. For instance, suppose you required 32 three-state signals, with each set of 8 having its own byte read, *byte_rd* (output enable) (*Figure 4-16*). The following generation scheme can employed:

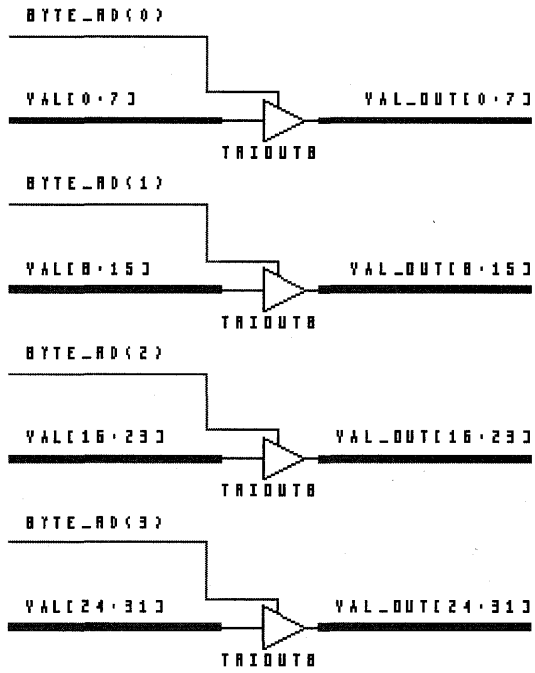


Figure 4-16 Three-state buffers with output enables for each byte

```
g1:   for i in 0 to 7 generate
         u0t7:  threestate port map (val(i), byte_rd(0), val_out(i));
         u8t15: threestate port map (val(i+8), byte_rd(1), val_out(i+8));
         u16t23: threestate port map (val(i+16), byte_rd(2), val_out(i+8));
         u24t31: threestate port map (val(i+24), byte_rd(3), val_out(i+24));
      end generated;
```

You can use more-complicated generation schemes. The generation scheme shown below is more complicated than need be (it describes the same logic as above), but it is helpful in demonstrating the flexibility of generation schemes. The name of the component has been abbreviated to *thrst*.

```
g1:   for i in 0 to 3 generate
   g2:   for j in 0 to 7 generate
            if i < 1 then generate
               ua: thrst port map(val(j), byte_rd(0), val_out(j));
            end generate;
            if i = 1 then generate
               ub: thrst port map (val(j+8), byte_rd(1), val_out(j+8));
            end generate;
            if i = 2 then generate
               uc: thrst port map (val(j+16), byte_rd(2), val_out(j+8));
            end generate;
            if i > 2 then generate
               ud: thrst port map (val(j+24), byte_rd(3), val_out(j+24));
            end generate;
         end generate;
      end generate;
```

Generation schemes can be used to instantiate any component, not just vendor supplied or user-written three-state and bidirectional components.

A return to the FIFO

Having covered many topics in this chapter, we return to our FIFO example of *Listing 4-1*. The entity declaration is simple enough and does not introduce any new concepts. The type declaration *fifo_array* is the first interesting construct encountered in this design. It is simply an array of eight std_logic_vectors wherein the std_logic_vectors are nine bits wide. Signal *fifo* is then declared to be of this type (one-dimensional array of std_logic_vectors). We can therefore access eight std_logic_vectors by indexing *fifo*, as in *fifo(3)*, *fifo(7)*, *fifo(1)*, etc. The next new concept that we run across is the loop.

Loops

Loop statements are used to implement repetitive operations and consist of either FOR loops or WHILE loops. The FOR statement will execute for a specific number of times based on a controlling value. The WHILE statement will continue to execute an operation as long as a controlling logical condition evaluates true. An extra step is required to initialize the controlling variable of a WHILE statement. Take, for instance, the loop used to asynchronously reset the FIFO array:

```
for i in 7 downto 0 loop
        fifo(i) <= (others => '0');
end loop;
```

This loop sequences through the eight std_logic vectors that make up the FIFO array, setting each element of the vectors to '0'. In a FOR loop, the loop variable is automatically declared. A WHILE loop can be used here but requires the additional overhead of declaring, initializing, and incrementing the loop variable. (The variable can be initialized upon declaration or within the process. Depending on how you use the variable, it may be required to reinitialize the variable within the process.)

```
reg_array: process (rst, clk)
    variable i: integer := 0;
  begin
        if rst = '1' then
                while i < 7 loop
                        fifo(i) <= (others => '0');
                        i := i + 1;
                end loop;
         ...
```

The NEXT statement is used to skip an operation based on specific conditions. Suppose, for example, that when *rst* is asserted, all of the *fifo* registers are reset except for the *fifo(4)* register:

```
reg_array: process (rst, clk)
  begin
        if rst = '1' then
                for i in 7 downto 0 loop
                  if i = 4 then
                        next;
                  else
                        fifo(i) <= (others => '0');
                end loop;
         ...
```

Or, as written with a while loop,

```
reg_array: process (rst, clk)
    variable i: integer := 0;
  begin
        if rst = '1' then
                while i < 8 loop
                  if i = 4 then
                        next;
                  else
                        fifo(i) <= (others => '0');
                        i := i + 1;
                end loop;
         ...
```

The EXIT statement is used to exit from a loop. This can be used to check an illegal condition. Suppose, for example, that FIFO is a component that is used for instantiating in a hierarchical design. Suppose further that the depth of the FIFO was defined by a generic, or parameter (these will be discussed in our chapter on hierarchy). You may wish to exit the loop when the depth of the FIFO is greater than a predetermined value. For instance,

```
reg_array: process (rst, clk)
  begin
        if rst = '1' then
         loop1: for i in deep downto 0 loop
                  if i > 20 then
                        exit loop1 when i > 3;
                  else
                        fifo(i) <= (others => '0');
                end loop;
```

You can see that for clarity, a loop label has been added. Loops have been used in a limited capacity here; however, they can be used to perform a myriad of functions. For example, the second loop used in the FIFO design is used to check which register is being written to (the *en* signal is decoded from *wrptr*):

```
                if wr = '1' then
                        for i in 7 downto 0 loop
                                if en(i) = '1' then
                                        fifo(i) <= data_in;
                                else
                                        fifo(i) <= fifo(i);
                                end if;
                        end loop;
                end if;
```

Our having discussed registered and combinational logic, including three-state and bidirectional signals, the remainder of the design provides no new challenges. The read and write pointers are simply three-bit counters that indicate which register in the *fifo* array to write to or read from. The dataflow constructs are used to decode the read and write counters, and the *three_state* process is used to control the three-state outputs.

Figure 4-11 shows a block diagram of the FIFO design.

Completing the FIFO

Now that you have mastered the basics, you are ready to move on to additional topics covered in the rest of this book, starting with state machines in the next chapter. But before we do that, we leave you with a modified version of our FIFO in which the width and depth are specified with a generic (parameter) and a constant. Generics are described in more detail in chapter 6, "The Design of a 100BASE-T4 Network Repeater." Additionally, the read and write pointers are not explicitly decoded because their values are of type integer. You'll see that this provides an even more concise way to describe the FIFO. There is one drawback to this implementation, however: the description of the counters using integers. A synthesis tool will internally convert the integers to a binary value such that when the counter reaches its maximum value, it will roll over on the next count. A VHDL simulator, however, will not convert the integers to a binary value. Consequently, when the counter reaches its maximum value, incrementing the counter will cause the simulator to issue an error indicating that the range for the counter signal (type integer) has been exceeded. This incompatibility between simulation and synthesis can be worked around in simulation by forcing the value of the counter to roll over at the appropriate time. Alternatively, you can explicitly specify in the VHDL code that the counter is to return to zero after it reaches its maximum value. This is the preferred solution as it maintains compatibility between the simulation and synthesis results and is unlikely to

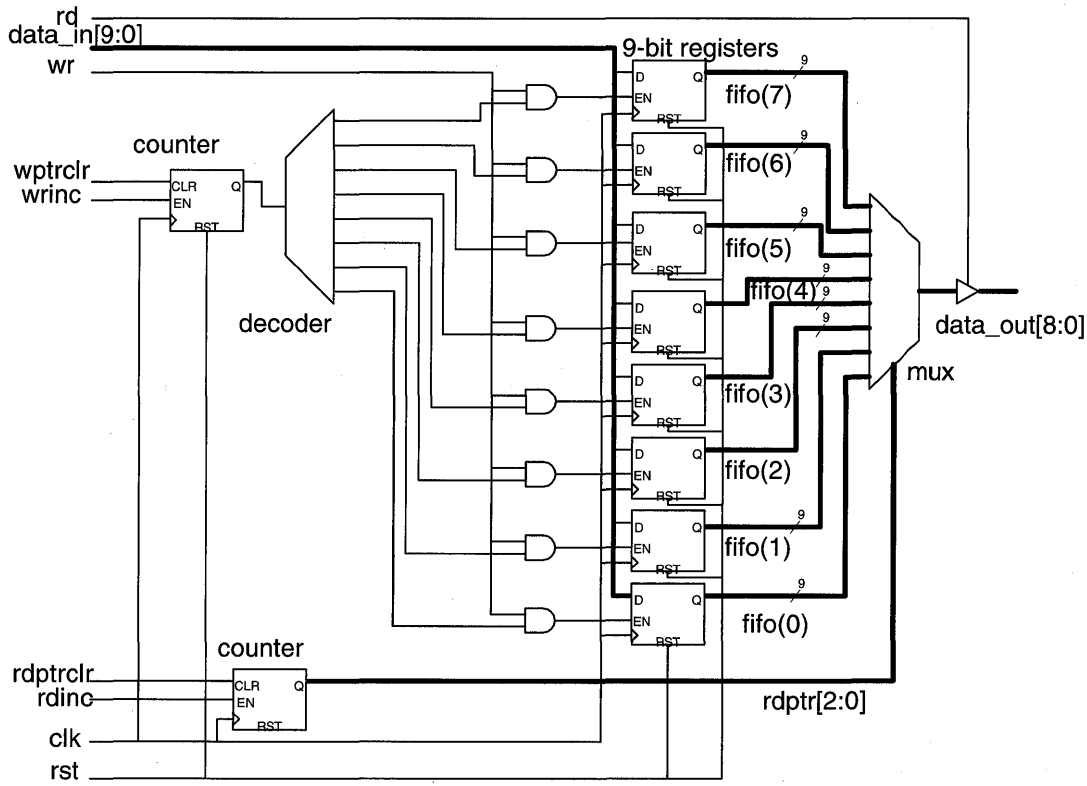


Figure 4-17 FIFO block diagram

cause additional resources to be used in the PLD. (This, however, is dependent on the compiler.) We leave you to make this modification.

Yet another approach is taken for implementation of a FIFO in chapter 6, "The Design of a 100BASE-T4 Network Repeater."

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

entity fifo8x9 is generic (wide : integer := 32);
    port(
    clk, rst, oe:           in std_logic;
    rd, wr, rdinc, wrinc:   in std_logic;
    rdptrclr, wrptrclr:     in std_logic;
    data_in:                in std_logic_vector(wide downto 0);
    data_out:               out std_logic_vector(wide downto 0));
end fifo8x9;
```

```
architecture archfifo8x9 of fifo8x9 is
  constant deep: integer := 20;
  type fifo_array is array(deep downto 0) of std_logic_vector(wide downto
0);

  signal fifo: fifo_array;
  signal wrptr, rdptr: integer range 0 to deep;
  signal en: std_logic_vector(deep downto 0);
  signal dmuxout: std_logic_vector(wide downto 0);

begin

-- fifo register array:
reg_array: process (rst, clk)
  begin
    if rst = '1' then
        for i in deep downto 0 loop
            fifo(i) <= (others => '0');
        end loop;
    elsif (clk'event and clk = '1') then
        if wr = '1' then
            fifo(wrptr) <= data_in;
        end if;
    end if;
  end process;

-- read pointer
read_count: process (rst, clk)
  begin
    if rst = '1' then
        rdptr <= 0;
    elsif (clk'event and clk='1') then
        if rdptrclr = '1' then
            rdptr <= 0;
        elsif rdinc = '1' then
            rdptr <= rdptr + 1;
        end if;
    end if;
  end process;

-- write pointer
write_count: process (rst, clk)
  begin
    if rst = '1' then
        wrptr <= 0;
    elsif (clk'event and clk='1') then
        if wrptrclr = '1' then
            wrptr <= 0;
        elsif wrinc = '1' then
            wrptr <= wrptr + 1;
        end if;
    end if;
  end process;
```

```
-- data output multiplexer
dmuxout <= fifo(wrptr);

-- three-state control of outputs
three_state: process (oe, dmuxout)
  begin
    if oe = '1' then
        data_out <= dmuxout;
    else
        data_out <= (others => 'Z');
    end if;
  end process;

end archfifo8x9;
```

Common Errors

Hidden Registers

Signal assignments can be synchronized to a clock by using one of several constructs. Remember that *all* signal assignments after the synchronizing statements, as in the example below, are synchronous to a clock.

```
seq: process (clk)
  begin
      if clk'event and clk='1' then
         b <= c;
         a <= b;
         h <= i;
         i <= j XOR k;
      end if;
  end process;
```

This code does not describe logic in which *b* is equivalent to *c* and *h* is equivalent to *i*. Rather, from this code, synthesis infers registers for each of the signal assignments, producing the logic of *Figure 4-8*. All assignments after the initial *if* statement are synchronous to the clock. If you want signals *h* and *i* not to be registered, then these sequential signal assignments should be removed from the process and made concurrent signal assignments, as shown below.

```
seq: process (clk)
  begin
      if clk'event and clk='1' then
         b <= c;
         i <= j XOR k;
      end if;
  end process;
      a <= b;
      h <= i;
```

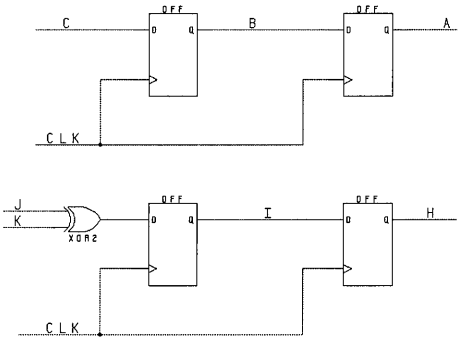


Figure 4-18 A process can describe interconnected flip-flops. The order of signal assignment is relevant

When simulators are used to process VHDL, signal assignments within processes are *scheduled*, not immediate. The signal assignments will only be made after termination of the process. Synthesis software will also generate logic based on this assumption. For this reason, care should be taken in using a signal that is assigned in a process also as a signal for the basis of comparison, as in the code below. Variable assignments within processes are immediate; however, variables are only visible in a process.

Careful evaluation of the code in *Listing 4-9* reveals that it describes the logic of *Figure 4-19.*

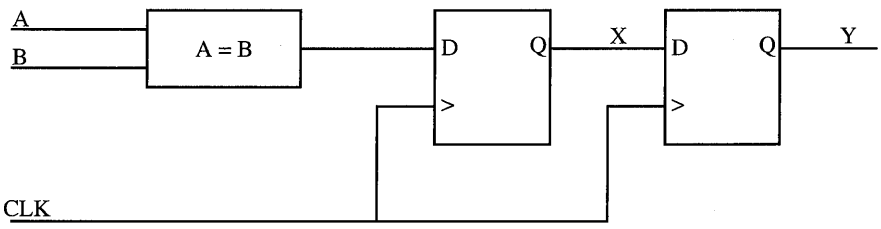


Figure 4-19 Logic of *Listing 4-9*

```
architecture careful of dangerous is
signal x: bit;
begin
p1: process begin
   wait until clk = '1';
   x <= '0';
   y <= '0';
   if a = b then
      x <= '1';
```

```
   end if;

   if x = '1' then
      y <= '1';
   end if;
end process p1;
end careful;
```

Listing 4-9 A signal in an assignment and as an operand

In this process, signal *x* is used as the object of an assignment and as an operand in a comparison. Because the assignment to *x* is scheduled and not immediate, the subsequent comparison, *if x = '1'*, compares the *present, not scheduled,* value of *x*. Thus, *y* is the registered version of *x*.

If x is a variable, rather than a signal, then the architecture can be written as follows, in which *y* is output of a combinational comparison of *a* and *b*.

```
architecture careful of dangerous is
begin
p1: process
  variable x: bit;
begin
   wait until clk = '1';
   x := '0';
   y <= '0';
   if a = b then
      x := '1';
   end if;

   if x = '1' then
      y <= '1';
   end if;
end process p1;
end careful;
```

Listing 4-10 A registered equality comparator.

Because *x* is a variable in this process, the assignment is immediate; hence, *y* is the output of an equality comparator (*Figure 4-20*). As a variable, *x* is only meaningful inside the process; to use the value outside of the process requires that its value be assigned to a signal as in the following code:

```
architecture pass_variable of to_signal
   signal vec: bit_vector(0 to 3);
   and_result: bit;
begin
proc: process (vec)
   variable result: bit := '1';
 begin
   for i in 0 to 3 loop
      result := result AND vec(i);
   end loop;
   and_result <= result;
 end process;
```

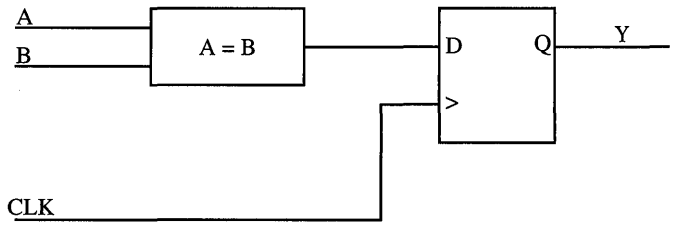


Figure 4-20 Logic described by *Listing 4-10*

```
 output_enable <= wmask AND and_result;
end pass_variable;
```

This code demonstrates that *result* can be used for immediate assignment, the value of which is passed to *and_result* at the end of the process. Placing the assignment statement for *and_result* at the beginning of the process would clearly produce a different result—it would always be assigned '1'.

Improper use of variables

In terms of processing VHDL for simulation, variables are only meaningful in a process and do not retain their values between periods of time when the processes are inactive. Variables must be reinitialized each time the process is activated. Synthesis creates logic based on these assumptions. Thus, the following code **does not** describe a counter:

```
architecture incorrect of bad_counter
begin
count: process (rst, clk)
   variable cnt: integer;
 begin
   if rst = '1' then
      cnt := 0;
   elsif (clk'event and clk='1') then
      cnt := cnt + 1;
   end if;
 end process;
end incorrect;
```

The variable *cnt* does not retain its value for the time during which the process is inactive.

Non-synthesizeable code

The following code is perfectly legal VHDL. It is a simulation model useful with simulation tools. This code cannot, however, be synthesized. Read through the code and see for yourself that this code accurately models simulation in the time domain. You'll also notice that this code doesn't describe a design for which synthesis software can create gates.

```
-- check for clk setup/hold violations
PROCESS (clk, d)
```

```
    VARIABLE dlastev        : TIME := 0 ns;
    VARIABLE clklastev      : TIME := 0 ns;
BEGIN
    -- Check only if the cell is registered.
    IF (registered = reg) THEN

        -- setup check
        IF (pchanging(clk) AND (clk = '1')) THEN
            ASSERT ((NOW = 0 ns) OR ((NOW - dlastev) >= Ts))
                REPORT
                "Setup ERROR ON D: Setting output to unknown"
                SEVERITY WARNING;
            IF NOT ((NOW = 0 ns) OR ((NOW - dlastev) >= Ts)) THEN
                q_ts <= '1';
                q_ts <= transport '0' after 1 ns;
            END IF;
            clklastev := NOW;
        END IF;

        -- hold check
        IF (pchanging(d)) then
            ASSERT ((NOW = 0 ns) OR ((NOW - clklastev) >= Th))
                REPORT
                "Hold ERROR ON D: Setting output to unknown"
                SEVERITY WARNING;
            IF NOT ((NOW = 0 ns) OR ((NOW - clklastev) >= Th)) THEN
                q_th <= '1';
                q_th <= transport '0' after 1 ns;
            END IF;
            dlastev := NOW;
        END IF;
    END IF;
END PROCESS;
```

Exercises

1. Write the VHDL code to describe an 8-bit wide, 4-to-1 mux using IF-THEN-ELSE statements within a process (include a sensitivity list).

2. Write the VHDL code to compare two 8-bit buses called a_data and b_data and drive a_grtr_b true when a_data is greater than b_data.

3. Decode a memory space between 02AD3hex and 07FFFF in a 24-bit address range using the CASE-WHEN statement.

4. Build a 4-bit magnitude comparator with 3-outputs (equals, lesser than and greater than) using:

a) logical operators

b) relational operators

c) structural instantiation

d) WHEN - ELSE construct

e) IF - THEN - ELSE construct

5. Write the VHDL code for a 32-bit register bank built using D flip-flops. The register bank has three-state outputs, controlled by a common output enable. Build this:

a) with structural Instantiation

b) structural instantiation using FOR-GENERATE and FOR-LOOPs

c) behaviorally

6. Build a 16-bit loadable down-counter with three-state outputs. The counter has two enable signals and can be asynchronously. The enable signals, control the lower 8-bits and the upper 8-bits respectively of the counter's three-state outputs.

7. Build a 16-bit up-down loadable Counter which does not include a sensitivity list along with the process declaration. Create two signals which can preset and reset the counter asynchronously . The counter has three state outputs, controlled by a common output enable.

8. Write VHDL code to compute the value of the temperature in Fahrenheit scale:

a) When given Celsius values

b) Automatically compute the values over a range of 0 to 200 degrees celsius, over 5 degree intervals. Hint : Use loop statements.

9. Build a 8-bit adder:

a) behaviorally using the '+' operator

b) behaviorally using logical operators

c) with structural instantion

d) using 2 ADD4 components from the *Warp* MATH library.

10. How do you handle logical operators on operands of unequal lengths?

5 State Machine Design

Introduction

One of the most common uses for programmable logic is for state machines. In this chapter, we show how you can easily describe state machines that meet your performance and resource utilization goals.

We begin with a simple example showing that writing a behavioral state machine description in VHDL is simply a matter of translating a state flow diagram to CASE-WHEN and IF-THEN-ELSE statements. Next, we examine how state machine logic is realized in a PLD after synthesis—i.e., we see which device resources are utilized—for the purpose of determining the resulting timing characteristics. Along the way, we explore variations of design descriptions that when synthesized may use different device resources and, hence, have different timing characteristics. From the various design descriptions, you'll be able to choose a style that best fits your design requirements.

Depending on your design requirements (time-to-market, performance, and cost-of-design), you may be concerned with more than just having a functionally accurate design description. That's why we investigate methods for optimizing designs for speed or area. Some of the optimization techniques are more applicable to CPLD architectures than FPGAs architectures (or vice versa); others are device independent. We point out which techniques are transitory—needed only to work around the state of the art of synthesis—and which are always applicable. All of the techniques discussed assist in understanding not only the synthesis process but also how to get the most of CPLD and FPGA architectures. With these techniques, you'll be able to converge on the optimal design solution quickly, even if time-to-market, performance, and cost-of-design are all important. We conclude this chapter with a discussion of fault tolerance and ways to avoid problems when designing complex state machines.

A Simple Design Example

We will use the following problem description to design a state machine, first using traditional design methodologies, then with VHDL.

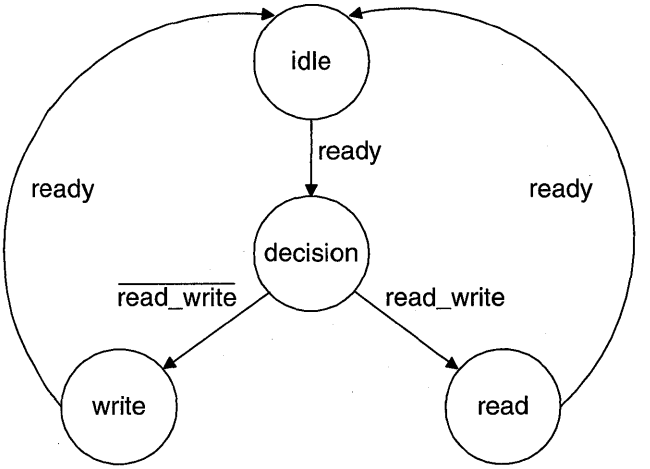
A controller is used to enable and disable the write enable *(we)* and output enable *(oe)* of a memory buffer during read and write transactions. The signals *ready* and *read_write* are microprocessor outputs that are inputs to the controller, and the signals *we* and *oe* are outputs of the controller. A new transaction begins with the assertion of *ready* following a completed transaction (or upon power-up for the initial transaction). One clock cycle after the commencement of the transaction, the value of *read_write* determines whether it is a read or write transaction. If *read_write* is asserted, then it is a read cycle; otherwise, it is a write cycle. A cycle is completed by the assertion of *ready*, after which a new transaction can begin. Write enable is asserted during a write cycle, and output enable is asserted during a read cycle.

Traditional Design Methodology

Traditional design methodology tells us that the first step is to construct a state diagram from which we can derive a state table. We can determine and eliminate equivalent states by matching rows of the state table and using an implication table, if necessary. We can then make state assignments and

create a state transition table from which we can generate next-state and output equations based on the types of flip-flops used for implementation.

From the design description, we can produce the state, or bubble, diagram of *Figure 5-1*. This diagram shows that a read or write transaction commences with the assertion of *ready*, in which case the state machine transitions from the *idle* state to the *decision* state. Depending upon the value of *read_write* during the next clock cycle, the transaction is either a read or write cycle, and the state machine transitions to the appropriate state. A transaction is completed when *ready* is asserted, placing the controller back in the *idle* state.



state	outputs oe	we
idle	0	0
decision	0	0
write	0	1
read	1	0

Figure 5-1 Simple State Machine

There are not any equivalent states in this machine: All states require different inputs to transition to the next state or have different outputs. We will combine the state assignment table with the state transition table (*Figure 5-2*); the state assignment is listed in the PS (present-state) column (we decided to use the fewest possible number of state registers, two). The NS (next-state) column shows transition from present state to next state based upon the present value of the two inputs, *read_write* and *ready*. The combinations of values for these inputs are shown in a row as 00, 01, 11, and 10. They are listed in this order for easy translation to a Karnaugh map. The outputs are in the rightmost column.

Next, we can determine the next-state equations for each of the two state bits (*Figure 5-2*). Q_1 and Q_0 represent the values of the next-state functions. The present-state values are represented in lowercase by q_1 and q_0. The Karnaugh maps can be generated easily from the state transition table: Each row corresponds to a state, and each column corresponds to a combination of the inputs; the entries in the Karnaugh maps correspond to the values of Q_1 and Q_0 in the transition table. The Karnaugh map can then be used to find the minimal equations assuming D-type flip-flops. The outputs are functions of the present state only.

PS state	q_1q_0	read_write, ready / NS Q_1Q_0 00	01	10	00	Outputs OE	WE
idle	00	00	01	01	00	0	0
decision	01	11	11	10	10	0	0
write	11	11	00	00	11	1	0
read	10	10	00	00	10	0	1

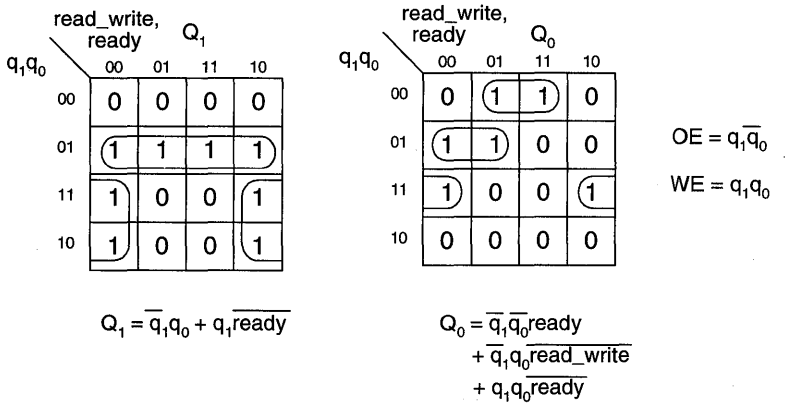


Figure 5-2 Determining next-state and output equations

This implementation can then be used in a PLD that has D-type flip-flops, such as the 22V10. To optimize the design for another type of flip-flop requires a different set of Karnaugh maps based on the excitation equations for the flip-flops and the transition table of the state machine.

State Machines in VHDL

The same state diagram can easily be translated to a high-level VHDL description without having to perform the state assignment, generate the state transition table, or determine the next-state equations based on the types of flip-flops available. In VHDL, each state can be translated to a case in a CASE-WHEN construct. The state transitions can then be specified in IF-THEN-ELSIF-ELSE constructs. For example, to translate the state flow diagram into VHDL, we begin by defining an enumerated type, consisting of the state names and two variables of that type:

```
type StateType is (idle, decision, read, write);
signal present_state, next_state : StateType;
```

Next, we create a process. *Next_state* is determined by a function of the *present_state* and the inputs (*ready* and *read_write*). We begin by writing a process that is sensitive to these signals:

```
state_comb: process (present_state, read_write, ready)
  begin
  ...
```

At this point, we describe the state machine transitions. We open the CASE-WHEN construct, specify the first case, and the state transitions from the *idle* state:

```
state_comb: process (present_state, read_write, ready)
 begin
   case present_state is
        when idle =>    if ready = '1' then
                          next_state <= decision;
                        else
                          next_state <= idle;
                        end if;
```

There are two options in this case (i.e., when *present_state* is *idle*): (1) to transition to *decision* if *ready* is asserted or (2) to remain in the *idle* state. The ELSE condition is not required. Without it, there is implied memory, and *next_state* remains the same. It is included to explicitly define state transitions. Coding of the remaining states requires following the same procedure: Create a case for each state (WHEN *state_name* =>) and indicate the state transitions with IF-THEN-ELSIF-ELSE constructs. Below is the complete definition of the state transitions:

```
state_comb:process(present_state, read_write, ready) begin
     case present_state is
          when idle     => if ready = '1' then
                               next_state <= decision;
                           else
                               next_state <= idle;
                           end if;
          when decision=> if (read_write = '1') then
                               next_state <= read;
                           else        --read_write='0'
                               next_state <= write;
                           end if;
          when read     => if (ready = '1') then
                               next_state <= idle;
                           else
                               next_state <= read;
                           end if;
          when write    => if (ready = '1') then
                               next_state <= idle;
                           else
                               next_state <= write;
                           end if;
    end case;
end process state_comb;
```

The above process indicates the next-state assignment but does not indicate when the next state becomes the present state. This happens synchronously, on the rising edge of a clock, as indicated by the following process:

```
state_clocked:process(clk) begin
     if (clk'event and clk = '1') then
          present_state <= next_state;
     end if;
end process state_clocked;
```

Finally, the state machine is completed by writing the equations for the state machine outputs, *oe* and *we*:

```
  oe <= '1' when present_state = read else '0';
  we <= '1' when present_state = write else '0';
```

The complete code follows.

```
entity example is port (
     read_write, ready, clk  : in bit;
     oe, we                  : out bit);
end example;

architecture state_machine of example is
     type StateType is (idle, decision, read, write);
     signal present_state, next_state : StateType;
begin
state_comb:process(present_state, read_write, ready) begin
     case present_state is
          when idle     => if ready = '1' then
                               next_state <= decision;
                           else
                               next_state <= idle;
                           end if;
          when decision=> if (read_write = '1') then
                               next_state <= read;
                           else        --read_write='0'
                               next_state <= write;
                           end if;
          when read     => if (ready = '1') then
                               next_state <= idle;
                           else
                               next_state <= read;
                           end if;
          when write    => if (ready = '1') then
                               next_state <= idle;
                           else
                               next_state <= write;
                           end if;
    end case;
end process state_comb;

state_clocked:process(clk) begin
     if (clk'event and clk = '1') then
          present_state <= next_state;
     end if;
end process state_clocked;

-- combinatorially decoded outputs
```

```
  oe <= '1' when present_state = read else '0';
  we <= '1' when present_state = write else '0';

end state_machine; --architecture
```

Listing 5-1 Design of a simple memory controller

Verifying Design Functionality

Now that we have designed the state machine, we can examine the description to see if it accurately models the state machine behavior. To do this, we determine how a simulation tool would treat the code. To do this, we will evaluate the concurrent statements (processes are also concurrent statements). A process does not become active until one of the signals in its sensitivity list changes value. The signals in the *state_comb* sensitivity list represent the present state and state machine inputs. If any of these signals change value, then the process is executed. The *state_clocked* process is executed any time that *clk* is changes value. This process is used for *present_state* to capture the present value of *next_state* on the rising edge of the clock. See *Figure 5-3* for example input stimuli and the corresponding changes in states and outputs.

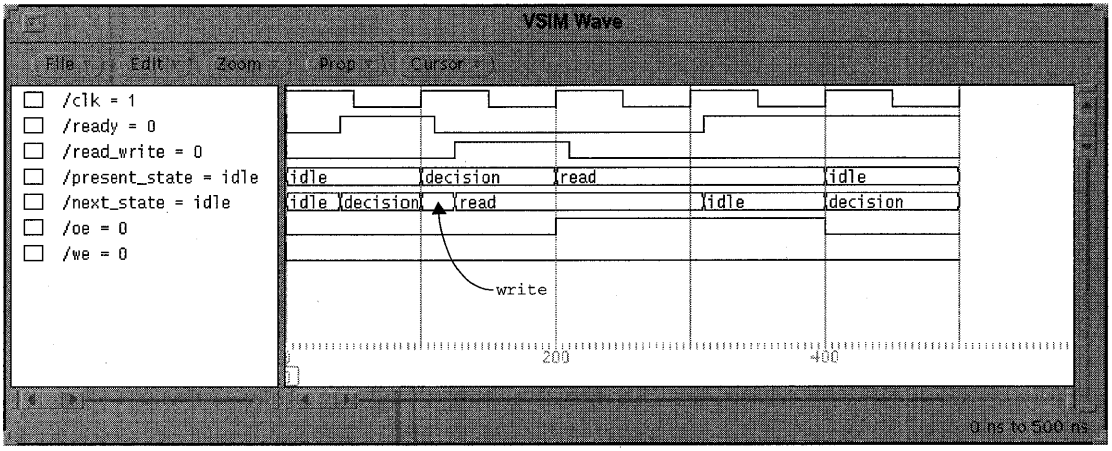


Figure 5-3 Functional verification of state machine

The LRM (Language Reference Manual, IEEE 1076) specifies that the initial value of a data object of an enumeration type is the 'left value, so all bits are initialized to '0,' and the signals *present_state* and *next_state* are initialized to *idle*. The LRM also specifies that all concurrent statements (including processes) are to be evaluated once after signals have been initialized. This means that at the beginning of clock period t_0, both processes are evaluated as well as the outputs.

When the *state_comb* process is evaluated, *present_state* is *idle*, and execution of the CASE statement jumps to the selection for when *present_state* is *idle*. The IF condition is evaluated: The input *ready* is '0,' so the statement "*next_state <= idle*" is executed (*next_state* retains it's initialized value). Execution jumps to the end of the CASE statement. No further assignments statements that

would schedule a new value for *next_state* are executed, so at the end of the process, *next_state* is assigned the value of *idle*. When the process *state_clocked* is evaluated as part of initialization, the IF condition evaluates false, and the process ends. A new value for *present_state* is not scheduled. The concurrent signal assignments are evaluated, and the outputs are both deasserted. After initialization, simulation time can be executed or inputs modified so as to effect a change in value of the circuit signals.

Shortly before the middle of clock period t_0, the input *ready* is asserted. This transition causes the *state_comb* process to execute. Execution of the CASE statement jumps to the selection for when *present_state* is *idle*. The IF condition is evaluated: The input *ready* is '1,' so the statement "*next_state <= write*" is executed (i.e., a new value for *next_state* is scheduled. If no further assignments to *next_state* are made before the end of the process, then *next_state* will assume the value of *decision*.) Execution jumps to the end of the CASE statement, and the process becomes inactive again.

In the middle of clock period t_0, *clk* transitions from a '1' to a '0.' This causes the *state_clocked* process to become active. The IF condition evaluates false and the process ends. A new value for *present_state* is not scheduled.

At the beginning of clock period t_0, *clk* transitions from a '0' to a '1.' This causes the *state_clocked* process to become active. The IF condition evaluates true, so *present_state* is scheduled to assume the value of *next_state*, which is *decision*.

The change in value of *present_state* from *idle* to *decision* causes the *state_comb* process to be executed. Execution of the CASE statement jumps to the selection for when *present_state* is *decision*. *Read_write* is deasserted, so *next_state* is scheduled to assume the value of *write*.

Shortly after the transition of *present_state* from *idle* to *decision*, *ready* is deasserted. This is a signal in the sensitivity list of process *state_comb*, so the process is activated and executed. Execution of the CASE statement jumps to the selection when *present_state* is *decision* (although *next_state* has changed value, *present_state* has not). *Read_write* has not changed value, and the *next_state* is presently *write*. Therefore, the signal assignment statement "*next_state <= write*" does not cause a new value to be scheduled for *next_state*.

About one-third of the way through clock period t_1, *read_write* is asserted, so the *state_comb* process is activated and executed. Execution of the CASE statement jumps to the selection when *present_state* is *decision* (*present_state* has not yet changed). The IF condition evaluates true (*read_write* is asserted). *Next_state* is scheduled to have the value of *read* upon termination of the process. The ELSE is not executed, and execution jumps to the end of the CASE statement. The process is inactive again.

The change in value of *present_state* from *decision* to *read* causes the *state_comb* process to be executed. Execution of the CASE statement jumps to the selection for when *present_state* is *read*. *Ready* is deasserted, so *next_state* retains its value, *read*.

Shortly after the transition of *present_state* from *decision* to *read*, *read_write* is deasserted. This is a signal in the sensitivity list of process *state_comb*, so the process is activated and executed. Execution of the CASE statement jumps to the selection when *present_state* is *read*. *Ready* is not asserted, so *next_state* retains its value.

The next rising edge of *clk* occurs: The *state_clocked* process is executed, and *present_state* retains its value, *read*.

About one-third of the way through clock period t_3, *ready* is asserted. Process *state_comb* is executed: Execution of the CASE statement jumps to the selection when *present_state* is *read*. *Ready* is asserted, so *next_state* is scheduled to assume the value *idle*.

On the next rising edge of *clk*, *present_state* assumes the value of *idle*.

The outputs, *we* and *oe*, are obtained through simple dataflow (concurrent) signal assignments.

We have partially verified that the design of our state machine is consistent with its behavioral model. It is a good idea to use a VHDL simulator to verify the functionality of your source code before synthesizing the design. In the end, this can reduce your total cycle time. The alternative is to go ahead with the synthesis and verify the functionality after fitting or placing and routing.

Results of Synthesis

With the traditional design methodology, you are expected to perform the logic synthesis from problem description to logic equations. Provided that you make the correct assumptions about device resources to be used, you can use VHDL to design a state machine at this level. In this case, you write Boolean equations in place of the *state_comb* process, as in the following code fragment, which illustrates the assignment of one state register (remember that the logical operators do not carry precedence over each other). Because VHDL is case insensitive, we cannot use *q* to represent the present state and *Q* the next state, as in the state transition table. Instead, we use *x* and *y*, respectively:

```
   y(0) <= ((NOT x(1)) AND (NOT x(0)) AND ready) OR
           (x(0) AND (NOT read_write) AND (NOT ready)) OR
           (x(1) AND x(0) AND (NOT ready));

   state_clocked:process(clk) begin
     if (clk'event and clk = '1') then
          x <= y;
     end if;
end process state_clocked;
```

Synthesis of the behavioral design description will produce the same logic. Following are the equations generated by *Warp* synthesis and documented in a report file.

```
    oe =
          present_stateSBV_0.Q * /present_stateSBV_1.Q

    we =
          present_stateSBV_0.Q * present_stateSBV_1.Q

    present_stateSBV_0.D =
          present_stateSBV_0.Q * /ready
        + /present_stateSBV_0.Q * present_stateSBV_1.Q

    present_stateSBV_1.D =
          /present_stateSBV_0.Q * present_stateSBV_1.Q * /read_write
        + present_stateSBV_0.Q * present_stateSBV_1.Q * /ready
        + /present_stateSBV_0.Q * /present_stateSBV_1.Q * ready
```

The equations in the report file do not make reference to *next_state*. The next state is a combinational function of the present state and the inputs. The present state registers are represented by *present_state*.Q*, and the next state is represented by the expression defined for *present_state*.D*.

Sometimes a Boolean description is more appropriate than a high-level behavioral description. However, state machines are clearly examples of the type of design that is best expressed with high-level constructs. A high-level description is easier to understand and maintain. Additionally, as the number of inputs to or states of a state machine increases, generating the transition table and next-state equations becomes increasingly tedious and difficult to manage, as our next example illustrates.

A Memory Controller

This memory controller is more practical and has more functionality than the one in our previous example. *Figure 5-4* shows a block diagram of a system that uses a state machine for a memory controller. All signals to the left of the *State Machine* block are inputs to that block, all signals to the right are outputs of that block and inputs to the *SRAM Memory Array*. *Address* is also an input to the memory array; *data* is bidirectional.

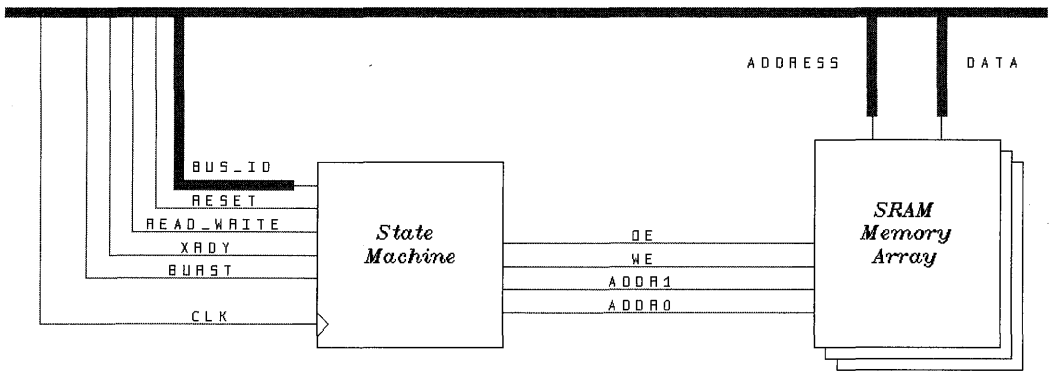


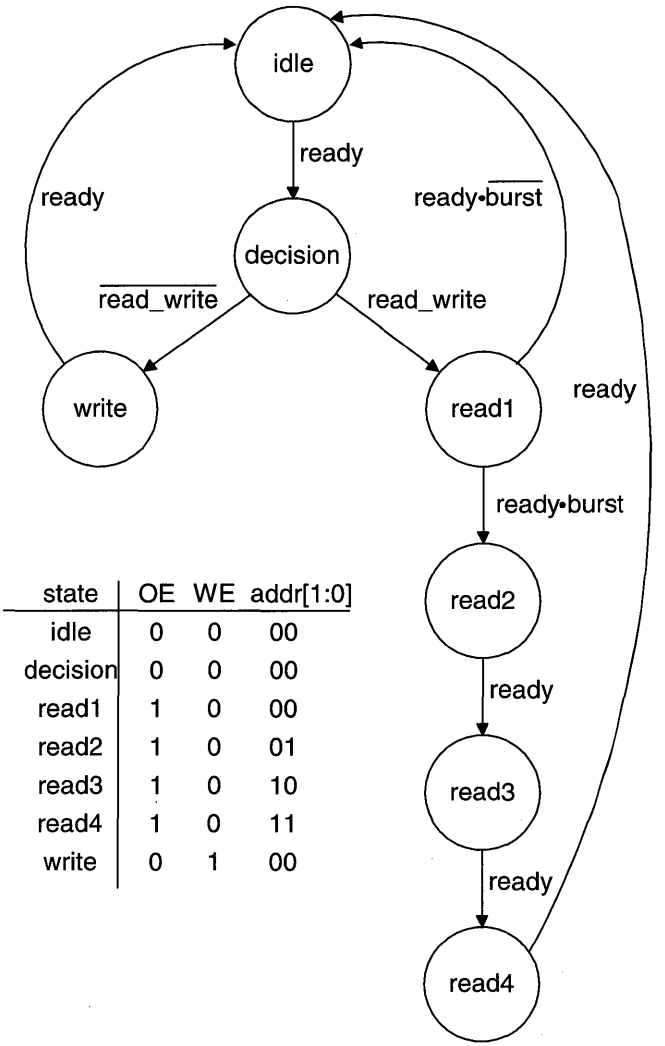
Figure 5-4 Memory Controller Block Diagram

The system works like this: Other devices on the bus initiate an access to the memory buffer by asserting the buffer's bus identification, F3 (hex). One cycle later, the *read_write* signal is asserted to indicate a read from the memory buffer; the signal is deasserted to indicate a write to the memory buffer. If the memory access is a read, the read may either be a single-word read or a four-word, burst read. A burst read is indicated by the assertion of *burst* during the first read cycle, following which the controller accesses four locations from the buffer. Consecutive locations are accessed following successive assertions of *ready*. The controller asserts *oe* (output enable) to the memory buffer during a read, and it increments the lowest two bits of the address during a burst.

A write to the buffer is always a single-word write, never a burst. During a write, *we* is asserted, allowing *data* to be written to the memory location specified by *address*. Read and write accesses are completed upon assertion of *ready*.

Figure 5-5 is the state diagram for this memory controller. This diagram shows that a synchronous reset places the state machine in the *idle* state. When the memory buffer is not being accessed, the controller remains in the *idle* state. If the *bus_id* is asserted as F3 (hex) while in *idle*, then the machine transitions to the *decision* state. The following transition is to either *read1* or *write*, depending on the value of *read_write*. If the access is a read, the controller branches to the read portion of the state machine. A single-word read is indicated by the assertion of *ready* without the assertion of *burst*. In this case, the controller returns to the *idle* state. A burst read is indicated by the assertion of both *ready* and *burst* while in the *decision* state. In this case, the machine transitions through each of the read states, advancing on *ready*. *Oe* is asserted during each of the read cycles. *Addr* is incremented in successive read cycles following the first.

If the access is a write, it can only be a single-word write. Therefore, after determining that the access is a write (*read_write* = 0) in the *decision* state, the controller branches to the write portion of the state machine. It simply asserts *we* (write enable) to the memory buffer, waits for the *ready* signal from the bus, and then returns directly to the *idle* state.



state	OE	WE	addr[1:0]
idle	0	0	00
decision	0	0	00
read1	1	0	00
read2	1	0	01
read3	1	0	10
read4	1	0	11
write	0	1	00

Figure 5-5 Memory Controller State Flow Diagram

Translating the State Flow Diagram to VHDL

The state flow diagram can be translated easily to a series of cases in a CASE-WHEN construct as follows (we are disregarding the synchronous reset for now):

```
case present_state is
     when idle     => if (bus_id = "11110011") then
                          next_state <= decision;
                      else
                          next_state <= idle;
                      end if;
     when decision=> if (read_write = '1') then
                          next_state <= read1;
                      else                        --read_write='0'
                          next_state <= write;
                              end if;
     when read1    => if (ready = '0') then
                          next_state <= read1;
                      elsif (burst = '0') then
                          next_state <= idle;
                      else
                          next_state <= read2;
                      end if;
     when read2    => if (ready = '1') then
                          next_state <= read3;
                      else
                          next_state <= read2;
                      end if;
     when read3    => if (ready = '1') then
                          next_state <= read4;
                      else
                          next_state <= read3;
                      end if;
     when read4    => if (ready = '1') then
                          next_state <= idle;
                      else
                          next_state <= read4;
                      end if;
     when write    => if (ready = '1') then
                          next_state <= idle;
                      else
                          next_state <= write;
                      end if;
end case;
```

As you can see, the section of code inside the *state_comb* process falls out directly from the bubble diagram: Each state is simply one of the cases in the CASE-WHEN construct, and all of the transitions from states are documented in IF-THEN statements. Take state *decision*, for example: Examining the state flow diagram, you see that there are two transitions from this state depending on the value of *read_write*. If *read_write* is asserted (a read operation), then *next_state* is *read1*, otherwise *next_state* is *write*. The remaining states are likewise coded.

This state machine requires a synchronous reset. Rather than specifying the reset condition in each of the transitions, we can include an IF-THEN construct at the beginning of the process in order to place

the machine in the *idle* state if *reset* is asserted. If *reset* is not asserted, then the normal state transitioning occurs. The code for this is as follows:

```
state_comb:process(reset, present_state, burst, read_write, ready) begin
  if (reset = '1') then
     next_state <= idle;
  else
     case present_state is
          ...
     end case;
  end if;
```

The complete code for the memory controller state machine is shown below as *Listing 5-2*.

```
library ieee;
use ieee.std_logic_1164.all;
entity memory_controller is port (
     reset, read_write, ready,
     burst, clk                    : in std_logic;
     bus_id                        : in std_logic_vector(7 downto 0);
     oe, we                        : out std_logic;
     addr                          : out std_logic_vector(1 downto 0)
);
end memory_controller;

architecture state_machine of memory_controller is
     type StateType is (idle, decision, read1, read2, read3, read4, write);
     signal present_state, next_state : StateType;
begin
state_comb:process(reset, bus_id, present_state, burst, read_write,
ready) begin

  if (reset = '1') then
     next_state <= idle;
  else
     case present_state is
          when idle     => if (bus_id = "11110011") then
                               next_state <= decision;
                           else
                               next_state <= idle;
                           end if;
          when decision=> if (read_write = '1') then
                               next_state <= read1;
                           else                          --read_write='0'
                               next_state <= write;
                           end if;
          when read1    => if (ready = '0') then
                               next_state <= read1;
                           elsif (burst = '0') then
                               next_state <= idle;
                           else
                               next_state <= read2;
                           end if;
          when read2    => if (ready = '1') then
```

```
                                next_state <= read3;
                            else
                                next_state <= read2;
                            end if;
          when read3    => if (ready = '1') then
                                next_state <= read4;
                            else
                                next_state <= read3;
                            end if;
          when read4    => if (ready = '1') then
                                next_state <= idle;
                            else
                                next_state <= read4;
                            end if;
          when write    => if (ready = '1') then
                                next_state <= idle;
                            else
                                next_state <= write;
                            end if;
    end case;
  end if;
end process state_comb;

state_clocked:process(clk) begin
     if (clk'event and clk = '1') then
          present_state <= next_state;
     end if;
end process state_clocked;

-- combinatorially decoded outputs
  with present_state select
        oe <= '1' when read1 | read2 | read3 | read4,
              '0' when others;

  we <= '1' when present_state = write else '0';

  with present_state select
        addr <= "01" when read2,
                "10" when read3,
                "11" when read4,
                "00" when others;
end state_machine; --architecture
```

Listing 5-2 Memory controller

Listing 5-2 illustrates a design that uses one process to define the combinational logic for state transitioning and a second process to synchronize the next-state assignment to the clock. This is a common and easy way to describe state machines, especially for machines that will be implemented in a PLD. A PLD architecture resembles this design structure because it consists of a product-term array feeding flip-flops. See *Figure 5-6* for an illustration of this concept. The decoding of *present_state* and inputs is performed in the combinatorial block of the PLD just as it is described in the combinatorial process in the code. Synchronization of *next_state* is described in the *state_clocked* process, referring to a bank of registers such as those in the PLD.

The state transitions defined in the *state_comb* process are fairly easy to follow, but you may be wondering about the state registers and state encoding. With this design, we chose to create a type called *StateType*, an enumerated type consisting of the state names: *idle, decision, read1, read2, read3*, and *read4,* and *write*. If you use an enumerated type, the state encoding is determined by the synthesis software unless you explicitly declare the state encoding by using an attribute, directive, command line switch, or GUI (graphical user interface) option. As a default, most synthesis tools use a sequential coding, in which case three bits are used for the seven states: *idle* is "000", *decision* is "001", *read1* is "010", etc. We'll show how to choose your own state encoding later in the chapter. Alternatively, we could have declared individual bits as state registers; however, using an enumerated type simplifies state transition coding and makes the code easier to comprehend and maintain.

Outputs for this design are defined with concurrent signal assignments using WITH-SELECT and WHEN-ELSE constructs. The outputs are functions of the present state only (a Moore machine) and can be defined quickly by making use of *StateType*.

If an asynchronous reset is desired instead of a synchronous reset, then you can use the asynchronous reset template discussed in the previous chapter by rewriting the *state_clocked* process of *Listing 5-2* as follows:

```
state_clocked:process(clk,reset) begin
     if reset= '1' then
        present_state <= idle;
     elsif (clk'event and clk = '1') then
          present_state <= next_state;
     end if;
end process state_clocked;
```

This state machine is one that would be difficult to design and maintain using a traditional design methodology. For instance, if the polarity of *ready*, *burst*, or *read_write* is reversed, updating the VHDL code is a simple task. Regenerating next-state design equations is not, because this design has many combinations of input. A comparison of the relative ease of design is left as an exercise for the reader!

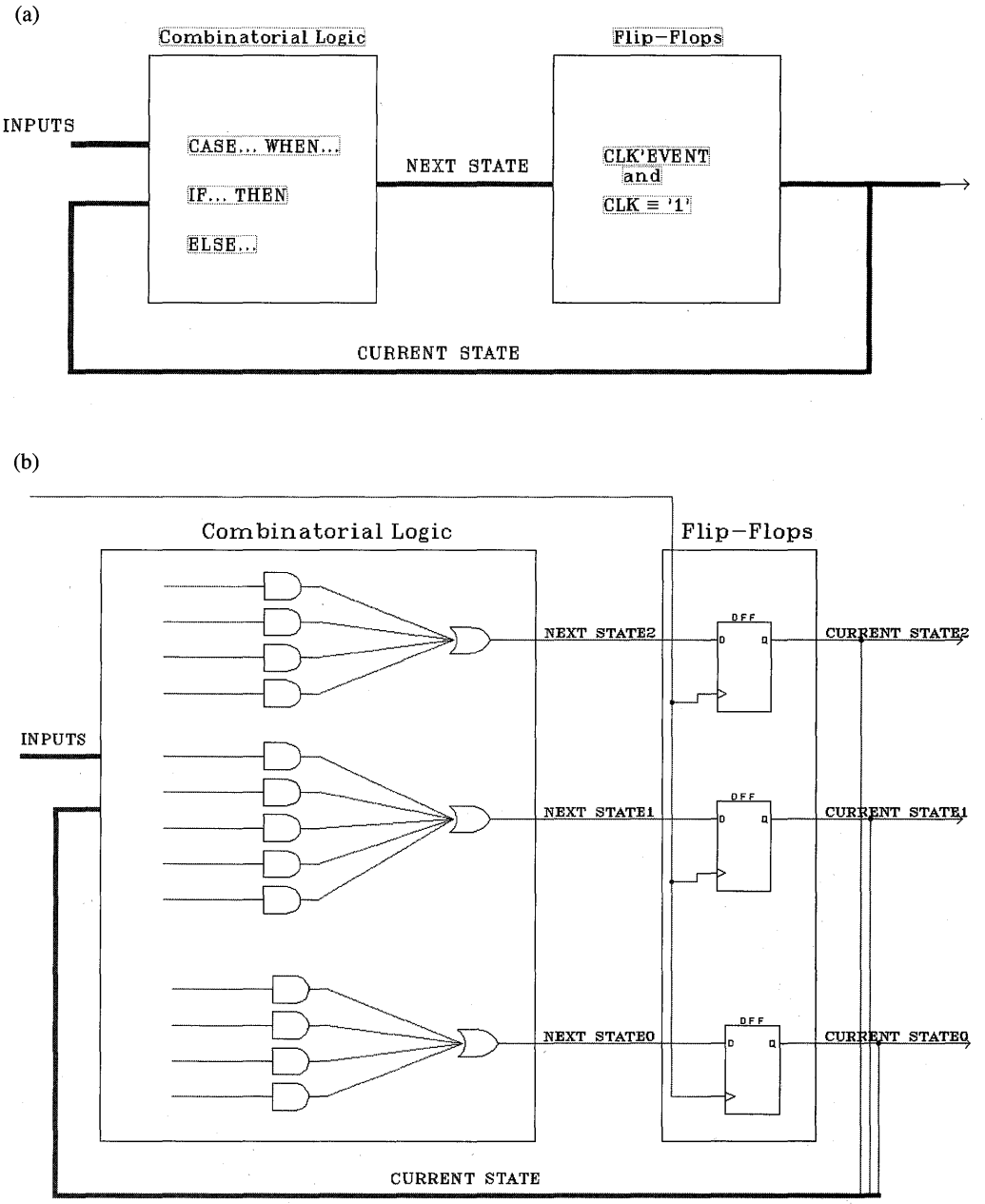


Figure 5-6 Comparing (a) the code structure of *Listing 5-2* to (b) the architecture of a PLD

An Alternative Implementation

The code in *Listing 5-3* below is functionally equivalent to the code in *Listing 5-2* above, and it requires the same device resources. We examine it here to illustrate a different coding style for the next-state decoding and output logic (the entity declaration is the same and is not reprinted):

```
architecture state_machine of memory_controller is
     type StateType is (idle, decision, read1, read2, read3, read4, write);
     signal state : StateType;
begin
state_tr:process(reset, clk) begin
  if (clk'event and clk='1') then
    if (reset = '1') then
       state <= idle;
    else
      case state is
          when idle     => if (bus_id = "11110011") then
                               state <= decision;
                           else
                               state <= idle;
                           end if;
          when decision=> if (read_write = '1') then
                               state <= read1;
                           else                        --read_write='0'
                               state <= write;
                           end if;
          when read1    => if (ready = '0') then
                               state <= read1;
                           elsif (burst = '0') then
                               state <= idle;
                           else
                               state <= read2;
                           end if;
          when read2    => if (ready = '1') then
                               state <= read3;
                           else
                               state <= read2;
                           end if;
          when read3    => if (ready = '1') then
                               state <= read4;
                           else
                               state <= read3;
                           end if;
          when read4    => if (ready = '1') then
                               state <= idle;
                           else
                               state <= read4;
                           end if;
          when write    => if (ready = '1') then
                               state <= idle;
                           else
                               state <= write;
                           end if;
      end case;
```

```
    end if;
  end if;
end process state_tr;
-- combinatorially decoded outputs
output_logic: process (state) begin
  if (state = read1 or state = read2 or state = read3 or state = read4)
then
  oe <= '1';
  else
  oe <= '0';
  end if;

  if state = write then we <= '1'; else we <= '0'; end if;

  if state = read2 then
   addr <= "01";
  elsif state = read3 then
   addr <= "10";
  elsif state = read4 then
   addr <= "11";
  else
   addr <= "00";
  end if;
end process output_logic;
end state_machine; --architecture
```

Listing 5-3 Memory controller with output decoding in a combinatorial process and state assignment in a clocked process

In this design description, the state transition process *state_tr* is used for the next-state logic *and* for clocking the state registers. Only one signal, *state*, of type *StateType* is required. The following statement implies that all subsequent signal assignments within the process occur on the rising edge of the clock:

```
if (clk'event and clk='1') then
```

That is, all signal assignments within this process are synchronized to the clock, and the following collection of statements implies that the assignment of *state* is synchronous:

```
state_tr:process(reset, clk) begin
  if (clk'event and clk='1') then
        ...
        case state is
         when idle    => if (bus_id = "11110011") then
                             state <= decision;
                         else
                             state <= idle;
                         end if;
```

Using this construct, *state* can have the present value *idle* and on the next clock edge be assigned a new value of *decision*. The code fragment above can be read, "In the case when the present state is *idle*, if *bus_id* is 11110011, then the new state will be *decision*; otherwise, the state will be *idle*." An advantage of *Listing 5-3* over *Listing 5-2* is its brevity—it requires neither the separate *state_clocked*

process nor the separate *StateType* signals, *present_state* and *next_state*, found in *Listing 5-2*. Later, however, you will see that using two state signals may provide a coding advantage when decoding outputs in parallel output registers. Either method is accurate—which one you choose is a matter of style.

The outputs in *Listing 5-3* are not described with concurrent statements, rather they are described using a process with IF-THEN-ELSE statements, which when synthesized results in combinational logic. To use this type of process along with the method of using two state signals for the present state and next state (as in *Listing 5-2*), you will need to replace *"state"* with *"present_state"* as we will show later in *Listing 5-4*. The results of synthesis will be the same; you can choose the style that you prefer. For compact code, choose the state transition structure found in *Listing 5-3* and the output logic description of *Listing 5-2*.

Both code listings indicate that the outputs are derived by using combinational logic to decode the current state registers. Likely, the combinatorial decode will add a level of logic physically between the flip-flop outputs and the output pins (*Figure 5-6*), affecting the timing of the output signals. A case in which an extra level of logic is not required for the output logic is one in which an output is equivalent to one of the state bits. Suppose, for example, that sequential encoding was chosen for a state assignment and that an output x must be asserted in any state for which the decimal equivalent is odd. In this case, x corresponds to the least significant bit of the state machine, and no decoding is required (*Figure 5-7*). The output of the flip-flop holding the least significant bit can simply be propagated to an output pin as x. We discuss timing issues next.

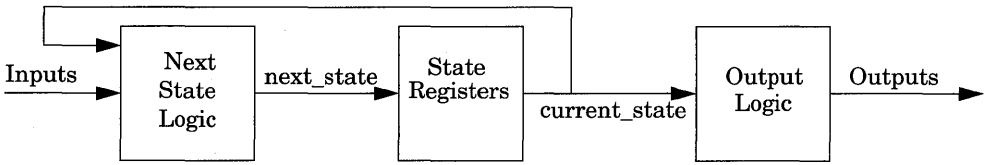


Figure 5-7 State machine with outputs decoded from state registers

Timing and Resource Usage

One state machine may synthesize differently than another, depending how it is described. Eventually, state of the art in synthesis will provide more optimization algorithms to optimize design descriptions based on user directives. In the future, coding will focus on describing a design in the clearest possible way in order to make the description easy to read, comprehend, and update. For now, state of the art in synthesis is largely based on RTL (register transfer level) optimization. Behavioral descriptions are broken down into explicitly declared or inferred registers and the logic between these registers. Optimization is then performed on the combinational logic, based on the types of registers available in the device architecture. In using synthesis tools, care must be taken to ensure that your descriptions provide the expected results. One description may produce a design that is faster or slower than another, while another may use more or less resources than another, and you may have to choose between design trade-offs or provide the appropriate directives.

Design trade-offs usually come in the form of trading one performance characteristic for another, or trading performance for "area" or device resources (how much logic a given design or portion of a design consumes). One design may have faster clock-to-output times (t_{CO}) and another may have faster setup times (t_{SU}). Both setup and clock-to-output times are important in determining the maximum frequency of operation, but one may be more important than another for a given system, depending on these timing parameters for interfacing devices. Area and performance often work against each other. Some implementations may require more flip-flops but fewer product terms (usually better for FPGAs), whereas others may require more product terms but fewer flip-flops (usually better for CPLDs).

Oftentimes, achieving the optimal implementation is not a design objective. Rather, time-to-market and having easy-to-read code that facilitates quick design cycles often supercede the need for the most area-efficient or highest-performance design. For these cases, the following discussion may not be necessary. For those cases that push the current limits of technology in terms of achievable silicon performance and the ability to use directives to influence synthesis choices, the following discussion will prove helpful.

Getting the optimal hardware implementation from a VHDL description can be achieved by understanding how specific VHDL implementations synthesize to logic in CPLDs and FPGAs from RTL-based synthesis. With this understanding, you are equipped to write code optimized for the particular timing and resource-utilization requirements of your design. In the following sections, we look at three techniques for generating state machine outputs for Moore machines: outputs decoded from state bits combinatorially, outputs decoded in parallel output registers, and outputs encoded within state bits. Each of these techniques produces a logic implementation with differing timing characteristics. We also investigate a technique called one-hot encoding that has implications not just for outputs and their timing, but for $t_{Q\text{-}Q}$ (register-to-register delays, useful for determining the maximum frequency of operation) and for gate utilization as well.

Outputs Decoded from State Bits Combinatorially

A third implementation of the memory controller uses the *state_comb* process of *Listing 5-2* (recall that *Listing 5-1* makes use of the signals *present_state* and *next_state*) and an *output_comb* process similar to that of *Listing 5-3* as shown in *Listing 5-4* below. This process uses the value of *present_state* to determine the value of the outputs.

```
-- combinatorially decoded outputs
output_logic: process (present_state) begin
  if (present_state = read1 or present_state = read2 or
        present_state = read3 or present_state = read4) then
        oe <= '1';
  else
        oe <= '0';
  end if;

  if present_state = write then we <= '1'; else we <= '0'; end if;

  if present_state = read2 then
        addr <= "01";
  elsif present_state = read3 then
        addr <= "10";
  elsif present_state = read4 then
        addr <= "11";
```

```
  else
        addr <= "00";
  end if;
end process output_logic;
```

Listing 5-4 Outputs decoded using a process

This design description (as with *Listing 5-2* and *Listing 5-3*), when synthesized by an RTL-based synthesis tool, results in a hardware implementation in which the state bits propagate from the state registers and through a level of combinational logic before propagating to the output pins of the PLD. *Figure 5-7* depicts this relationship between the state registers and outputs.

When the memory controller is synthesized and implemented in a CPLD such as the Cypress 32-macrocell CY7C371-143 CPLD, the resulting equations are similar to those shown below. These equations come from a report file produced by the *Warp* synthesis tool:

```
        idle :=         b"000";
        decision :=     b"001";
        read1 :=        b"010";
        read2 :=        b"011";
        read3 :=        b"100";
        read4 :=        b"101";
        write :=        b"110";

    we =
          present_stateSBV_1.Q * present_stateSBV_0.Q

    addr_1 =
          /present_stateSBV_1.Q * present_stateSBV_0.Q

    /oe =
          present_stateSBV_1.Q * /present_stateSBV_2.Q *
          present_stateSBV_0.Q
        + /present_stateSBV_1.Q * /present_stateSBV_0.Q

    addr_0 =
          present_stateSBV_2.Q * present_stateSBV_0.Q
        + present_stateSBV_1.Q * present_stateSBV_2.Q

    present_stateSBV_1.D =
          /reset * present_stateSBV_1.Q * /present_stateSBV_2.Q *
          /present_stateSBV_0.Q * burst
        + /reset * /present_stateSBV_1.Q * present_stateSBV_2.Q *
          /present_stateSBV_0.Q
        + /reset * /ready * present_stateSBV_1.Q

    present_stateSBV_1.C =
          clk

    present_stateSBV_2.D =
          /reset * /present_stateSBV_1.Q * /present_stateSBV_2.Q *
          /present_stateSBV_0.Q * bus_id_7 * bus_id_6 * bus_id_5 *
          bus_id_4 * /bus_id_3 * /bus_id_2 * bus_id_1 * bus_id_0
```

```
    + /reset * ready * present_stateSBV_1.Q * /present_stateSBV_2.Q *
      /present_stateSBV_0.Q * burst
    + /reset * ready * /present_stateSBV_1.Q * /present_stateSBV_2.Q *
      present_stateSBV_0.Q
    + /reset * /ready * present_stateSBV_2.Q * present_stateSBV_0.Q
    + /reset * /ready * present_stateSBV_1.Q * present_stateSBV_2.Q

present_stateSBV_2.C =
      clk
present_stateSBV_0.D =
      /reset * /read_write * /present_stateSBV_1.Q *
      present_stateSBV_2.Q * /present_stateSBV_0.Q
    + /reset * ready * present_stateSBV_1.Q * present_stateSBV_2.Q
    + /reset * /present_stateSBV_1.Q * /present_stateSBV_2.Q *
      present_stateSBV_0.Q
    + /reset * /ready * present_stateSBV_0.Q

present_stateSBV_0.C =
      clk
```

The equations from the report file do not make reference to *next_state*. Rather, the outputs of the state registers are represented by *present_state*.Q*, and the next state is represented by *present_state*.D*, combinational functions of the present state registers (See *Figure 5-7*).

The equations indicate that the state assignment is sequential and that the signal *present_state* is translated to a vector with a width of 3. The outputs are clearly combinatorial functions of the state registers. Outputs that must be decoded appear at the output pins 10.5 ns (t_{CO2}) after the rising edge of *clk*. All listings for the memory controller so far follow the logic implementation model of *Figure 5-7*, so each of these descriptions will also result in outputs available in 10.5 ns. Had any outputs propagated straight from the flip-flops to the output pins without going through the level of combinational logic, they would change 6.0 ns (t_{CO}) after the rising edge of *clk*, worst case. See xxx for a table of timing specifications for the CY7C371-143.

The VHDL code as it is written is easy to comprehend and maintain. However, there is a performance issue: The outputs of the state machine arrive t_{CO2} instead of t_{CO} after the rising edge of the clock. In many cases, this difference in delay will be acceptable in the overall system timing. In other cases, it may not be. Until the state of synthesis progresses, modification of the code will be required to achieve higher performance.

For example, let's revisit the design of the memory controller. Let's assume that for the data coming from the SRAM memory array to be ready in time for the device that is reading it, the *addr* outputs from the PLD must be available no more than 8 ns after the rising edge of the clock. In this particular implementation, they are not—they are available 10.5 ns after the rising edge of the clock. If we could find a way such that *addr* was not the result of decoding state outputs, which requires the additional pass through the logic array before going to the output pins, then *addr* would be available in 6.0 ns (t_{CO} instead of t_{CO2}), and the system design requirements would be satisfied.

Outputs Decoded in Parallel Output Registers

One way to ensure that the state machine outputs arrive at the device pins earlier is to decode the outputs from the state bits *before* the state bits are registered, and then store the decoded information in registers. In other words, instead of using the *present_state* information to determine the value for

addr, we use the *next_state* value to determine what *addr* should be in the next clock cycle. If the next state of the state machine is a state in which *addr(1)* is a '1', we store a '1' into a flip-flop at the rising edge of *clk*. If the *next_state* value indicates that the next state of the state machine is a state in which *addr(1)* is a '0', then we store a '0' into that flip-flop. The same idea can be used for the other outputs, but since there isn't a clock-to-output requirement for these outputs, we leave them as is. We illustrate the concept of storing the values of outputs based on the value of *next_state* in *Figure 5-8*.

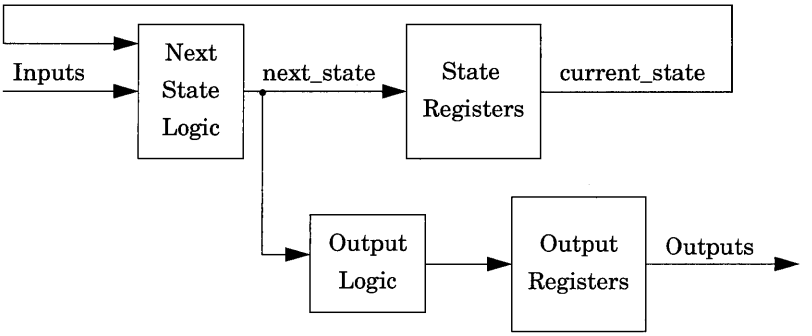


Figure 5-8 Moore machine with outputs decoded in parallel output registers

This implementation can be coded quickly in VHDL. Instead of using *present_state* in the equations for *addr*, we use *next_state*. We also register signals *addr* in flip-flops called *raddr*, for registered address. We do this by adding two lines of code to the process *state_clocked*, modifying the *output_logic* process to replace *present_state* with *next_state*, modifying the port declaration to replace *addr* with *raddr*, and including *addr* as a signal local to the architecture. That's it! Now, *raddr* has the same values in the same clock cycle as *addr* did in the previous implementations, but *raddr* is available t_{CO} after *clk* (6.0 ns in the CY7C371-143 CPLD we have chosen) instead of t_{CO2} (10.5 ns). The outputs are available in t_{CO} time because the value of the output address is held in flip-flops for which the outputs may propagate directly to the device pins rather than first propagating through the logic array. *Listing 5-5* shows the modified portion of the architecture.

```
-- combinatorially decoded outputs
output_logic: process (present_state) begin
  if (present_state = read1 or present_state = read2 or
        present_state = read3 or present_state = read4) then
        oe <= '1';
  else
        oe <= '0';
  end if;

  if present_state = write then we <= '1'; else we <= '0'; end if;

  if present_state = read2 then
        addr <= "01";
  elsif present_state = read3 then
        addr <= "10";
  elsif present_state = read4 then
```

```
        addr <= "11";
  else
        addr <= "00";
  end if;
end process output_logic;

state_clocked:process(clk) begin
     if (clk'event and clk = '1') then
          present_state <= next_state;
          raddr <= addr;
     end if;
end process state_clocked;
```

Listing 5-5 Moore machine with outputs from registers.

From the diagram of *Figure 5-8*, it may look, at first glance, as if this implementation may have two unintended side effects. First, it may look as though this implementation requires two more flip-flops than the previous version; second, it may look as if the propagation delay from flip-flop to flip-flop, $t_{Q\text{-}Q}$, between the state-bit flip-flops and the *raddr* flip-flops takes two passes through the combinational logic array (one for the next-state logic and one for the output logic), affecting the maximum frequency at which this design can operate. Both of these side effects may exist depending on the specific CPLD or FPGA chosen to implement the design. In the particular case of the CY7C371 and the Cypress *Warp* VHDL synthesis tool, however, they do not exist. The logic that determines the *next_state* signals and the logic that determines the outputs are combined and reduced into a single level of logic by the synthesis software. The resulting logic uses fewer than the 16 product terms available in a CY7C371 macrocell, so the output decoding logic requires only a single pass through the logic array. The equations produced by synthesis are shown below (equations for clock assignment are removed):

```
we =
      present_stateSBV_1.Q * present_stateSBV_0.Q

/oe =
      present_stateSBV_1.Q * /present_stateSBV_2.Q *
      present_stateSBV_0.Q
    + /present_stateSBV_1.Q * /present_stateSBV_0.Q

present_stateSBV_1.D =
      /reset * present_stateSBV_1.Q * /present_stateSBV_2.Q *
      /present_stateSBV_0.Q * burst
    + /reset * /present_stateSBV_1.Q * present_stateSBV_2.Q *
      /present_stateSBV_0.Q
    + /reset * /ready * present_stateSBV_1.Q

present_stateSBV_2.D =
      /reset * /present_stateSBV_1.Q * /present_stateSBV_2.Q *
      /present_stateSBV_0.Q * bus_id_7 * bus_id_6 * bus_id_5 *
      bus_id_4 * /bus_id_3 * /bus_id_2 * bus_id_1 * bus_id_0
    + /reset * ready * present_stateSBV_1.Q * /present_stateSBV_2.Q *
      /present_stateSBV_0.Q * burst
    + /reset * ready * /present_stateSBV_1.Q * /present_stateSBV_2.Q *
      present_stateSBV_0.Q
```

```
    + /reset * /ready * present_stateSBV_2.Q * present_stateSBV_0.Q
    + /reset * /ready * present_stateSBV_1.Q * present_stateSBV_2.Q


present_stateSBV_0.D =
      /reset * /read_write * /present_stateSBV_1.Q *
      present_stateSBV_2.Q * /present_stateSBV_0.Q
    + /reset * ready * present_stateSBV_1.Q * present_stateSBV_2.Q
    + /reset * /present_stateSBV_1.Q * /present_stateSBV_2.Q *
      present_stateSBV_0.Q
    + /reset * /ready * present_stateSBV_0.Q

raddr_1.D =
      /present_stateSBV_1.Q * present_stateSBV_0.Q

raddr_0.D =
      present_stateSBV_2.Q * present_stateSBV_0.Q
    + present_stateSBV_1.Q * present_stateSBV_2.Q
```

Addr required two macrocells in the previous design implementation; in this design implementation, *addr* is replaced by *raddr*. Thus, this design requires the same total number of macrocells as the first one. But, more product terms are required because the next state must essentially be decoded twice. Since the decoding is done in a single pass (single level) of logic, the $t_{Q\text{-}Q}$ is still at its maximum for this device, 7.5 ns.

Outputs Encoded within State Bits

An alternative way to acquire the state machine outputs in t_{CO} is to use the state bits themselves as outputs. A counter is an example of a state machine for which the outputs are also the state bits. This may work better than the previous method for some criteria, but it requires a slightly different implementation in which you must choose your state encoding carefully: You must choose a state encoding so that the outputs correspond to the values held by the state registers, as shown in *Figure 5-9*. This approach makes the design more difficult to comprehend and maintain, so it is only recommended for those cases that require specific area and performance optimization not provided by the current synthesis directives.

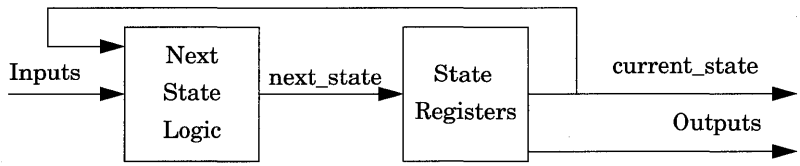


Figure 5-9 Moore machine with outputs encoded within state registers

We'll use the memory controller again to illustrate the concept of encoding the outputs within the state registers. With seven states in the machine, the fewest number of state bits that we can use is three. We also have two outputs, *addr(1)* and *addr(0)*, that we would like to have propagated to the output pins in t_{CO} time (the other two outputs are not critical and can take t_{CO2} time), so altogether we need at most seven macrocells (five for flip-flops and two for combinatorial outputs). Our task now

is to create a state encoding such that *addr(1)* and *addr(0)* are two of the state bits. To choose the encoding, we start by creating a table of the present state and outputs that we wish to encode. We will use this table as the starting point for our state encoding table.

State	addr(1)	addr(0)
idle	0	0
decision	0	0
read1	0	0
read2	0	1
read3	1	0
read4	1	1
write	0	0

Next, we examine the table, looking for the set of outputs that appears with the greatest frequency. The set of outputs "00" appears with the greatest frequency—four times. To create our state encoding table from here, we need to distinguish the state encoding for *idle*, *decision*, *read1*, and *write,* all of which have address outputs of "00". To create a unique encoding for each state, we need an additional two bits. For each of the four states with the same outputs, we must assign a unique combination of the additional encoding bits. We choose to order the bits sequentially as illustrated in the following table.

State	addr(1)	addr(0)	st1	st0
idle	0	0	0	0
decision	0	0	0	1
read1	0	0	1	0
read2	0	1		
read3	1	0		
read4	1	1		
write	0	0	1	1

The remaining states have unique outputs, so we choose to fill in the encoding table with "00" for these entries, although any arbitrary set of two bits will do. Our final encoding scheme follows.

State	addr(1)	addr(0)	st1	st0
idle	0	0	0	0
decision	0	0	0	1

State	addr(1)	addr(0)	st1	st0
read1	0	0	1	0
read2	0	1	0	0
read3	1	0	0	0
read4	1	1	0	0
write	0	0	1	1

We now have a unique state encoding for each state. You can see that with this scheme we had to use more than the fewest possible number of state bits to encode seven states. But now two of the outputs, *addr(1)* and *addr(0)*, are encoded in the state bits. If these outputs were not encoded, to implement the design, a total of seven macrocells would be required for both the state bits (three registers) and the outputs (two registers for *raddr(1)* and *raddr(0)*, and two macrocells for *oe* and *we*), as in the implementation of *Listing 5-5* in which two of the outputs were decoded in parallel with the state bits. Although in our new implementation we need to use more state bits than in the previous case, we need fewer total macrocells (six): four for the state encoding and *addr* outputs, and two for the *oe* and *we* outputs. This analysis is for a CPLD implementation; we will discuss an FPGA implementation in the following paragraph.

In an FPGA, the savings in logic resources is not as clear. Four registers and additional logic cells for the next-state logic as well as the output logic for *oe* and *we* are required. Without determining the complexity of the next-state and output logic, we can only make educated guesses about the number of logic cells required. Thus, for an FPGA, there is no clear advantage in using this implementation over the previous one in which the outputs are decoded in parallel output registers. The quickest way to find out is often to use software to synthesize the code and see first hand what the setup, internal flip-flop to flip-flop, and clock-to-out delays are. Later in the chapter, we'll choose a method that is more assured to help us achieve an efficient implementation in an FPGA. Nevertheless, the performance of this state machine is small enough that its implementation in an FPGA will not differ by much regardless of how it is described. For now, we'll proceed, assuming that we will target this design implementation to a CPLD.

Because we will be implementing this design in a CPLD, we will see if fewer total macrocells are required to encode all four state machine outputs in the state encoding. We start, as we did before, by creating a table of present state and outputs.

State	addr(1)	addr(0)	oe	we
idle	0	0	0	0
decision	0	0	0	0
read1	0	0	1	0
read2	0	1	1	0
read3	1	0	1	0
read4	1	1	1	0

State	addr(1)	addr(0)	oe	we
write	0	0	0	1

Next, we examine the table, looking for the set of outputs that appears with the greatest frequency. If all outputs were unique, then our state encoding would be complete. In this case, the set of outputs "0000" appears twice (once for state *idle* and once for state *decision*), so we must distinguish between *idle* and *decision* by adding an additional bit (one bit is sufficient to distinguish two values). We arbitrarily choose '0' for *idle* and '1' for *decision*. Next we arbitrarily choose to fill the remaining entries in the table with '0'. We could choose either '0' or '1' for any of these entries because the state encoding would remain unique regardless of which value we choose. Our final state encoding appears below.

State	addr(1)	addr(0)	oe	we	st0
idle	0	0	0	0	0
decision	0	0	0	0	1
read1	0	0	1	0	0
read2	0	1	1	0	0
read3	1	0	1	0	0
read4	1	1	1	0	0
write	0	0	0	1	0

With five bits, the state registers are now able to hold all the outputs. This is a savings of one macrocell over the case in which only *addr(1)* and *addr(0)* are encoded and a savings of two macrocells over the case in which the outputs are decoded either in parallel with or serially from the state bits. In general, macrocell savings depend upon the uniqueness of the state machine outputs on a state-by-state basis. Typically, the worst case will require you to use the same number of macrocells as when decoding the outputs in parallel registers. For state machines implemented in CPLDs, this technique will often produce the implementation that uses the fewest macrocells and achieves the best possible clock-to-out times.

We're now ready to design the state machine, and we'll start by coding the state assignments. Whereas before you simply indicated the states in an enumerated type, now you will have to explicitly declare the state encoding with constants. *Listing 5-6* illustrates this point, in which *present_state* and *next_state* are declared as std_logic_vectors, rather than the enumerated type, *StateType,* and the state encoding is specified with constants. (Some synthesis tools allow an enumerated type to be used in conjunction with an attribute or a directive as an alternative to defining constants to specify the state encoding.)

```
library ieee;
use ieee.std_logic_1164.all;
entity memory_controller is port (
     reset, read_write, ready,
     burst, clk                    : in std_logic;
     bus_id                        : in std_logic_vector(7 downto 0);
```

```
    oe, we                          : out std_logic;
    addr                            : out std_logic_vector(1 downto 0)
);
end memory_controller;

architecture state_machine of memory_controller is
  signal state : std_logic_vector(4 downto 0);
  constant idle    : std_logic_vector(4 downto 0) := "00000";
  constant decision: std_logic_vector(4 downto 0) := "00001";
  constant read1   : std_logic_vector(4 downto 0) := "00100";
  constant read2   : std_logic_vector(4 downto 0) := "01100";
  constant read3   : std_logic_vector(4 downto 0) := "10100";
  constant read4   : std_logic_vector(4 downto 0) := "11100";
  constant write   : std_logic_vector(4 downto 0) := "00010";
begin
state_tr:process(reset, clk) begin
  if (clk'event and clk='1') then
    if (reset = '1') then
       state <= idle;
    else
      case state is
          when idle     => if (bus_id = "11110011") then
                               state <= decision;
                           else
                               state <= idle;
                           end if;
          when decision=> if (read_write = '1') then
                               state <= read1;
                           else                        --read_write='0'
                               state <= write;
                           end if;
          when read1    => if (ready = '0') then
                               state <= read1;
                           elsif (burst = '0') then
                               state <= idle;
                           else
                               state <= read2;
                           end if;
          when read2    => if (ready = '1') then
                               state <= read3;
                           else
                               state <= read2;
                           end if;
          when read3    => if (ready = '1') then
                               state <= read4;
                           else
                               state <= read3;
                           end if;
          when read4    => if (ready = '1') then
                               state <= idle;
                           else
                               state <= read4;
                           end if;
          when write    => if (ready = '1') then
```

```
                        state <= idle;
                    else
                        state <= write;
                    end if;
            when others  => state <= "-----";
        end case;
      end if;
    end if;
end process state_tr;

-- outputs associated with register values
  we <= state(1);
  oe <= state(2);
  addr <= state(4 downto 3);
end state_machine; --architecture
```

Listing 5-6 State machine with outputs encoded in state registers

In this implementation, we used one process to describe and synchronize the state transitions. This process, *state_tr,* is identical to that of *Listing 5-3.* The procedure for translating the state diagram to a CASE-WHEN construct changes only slightly. All combinations of the vector *state* must be accounted for in the CASE-WHEN construct. Thus, the following code is used to indicate that illegal states are don't care conditions:

```
when others => state <= state "-----";
```

More will be said about illegal states and don't cares in our discussion of fault tolerance later in the chapter.

Because the state encoding was explicitly declared and this encoding was chosen such that it would contain present state outputs, the outputs can be assigned directly from the state variable, as follows:

```
-- outputs associated with register values
  we <= state(1);
  oe <= state(2);
  addr <= state(4 downto 3);
```

Accessing the outputs in this way (i.e., directly from the state flip-flops) means that they will be available t_{CO} after the rising edge of the clock, instead of going through the extra pass through the logic array and coming out in t_{CO2}, as was the case with the original implementation.

This new implementation ensures that an extra pass is not needed to decode several state bits in generating the state machine outputs. Consequently, the outputs are available at the device pins sooner. In addition, this method typically requires fewer macrocells than the method of decoding the outputs in parallel output registers. Equations produced from synthesis are below. Only five equations are required because none of the outputs need to be decoded. State bit names are replaced by the names of the outputs.

```
state_0.D =
      /addr_1.Q * /addr_0.Q * /oe.Q * /we.Q * /reset * /state_0.Q *
      bus_id_7 * bus_id_6 * bus_id_5 * bus_id_4 * /bus_id_3 *
      /bus_id_2 * bus_id_1 * bus_id_0
```

```
addr_1.D =
      /addr_1.Q * addr_0.Q * oe.Q * /we.Q * /reset * ready *
      /state_0.Q
    + addr_1.Q * oe.Q * /we.Q * /reset * /ready * /state_0.Q
    + addr_1.Q * /addr_0.Q * oe.Q * /we.Q * /reset * /state_0.Q

addr_0.D =
      /addr_0.Q * oe.Q * /we.Q * /reset * ready * burst * /state_0.Q
    + addr_1.Q * /addr_0.Q * oe.Q * /we.Q * /reset * ready *
      /state_0.Q
    + addr_0.Q * oe.Q * /we.Q * /reset * /ready * /state_0.Q

oe.D =
      /addr_1.Q * /addr_0.Q * /oe.Q * /we.Q * /reset * read_write *
      state_0.Q
    + addr_1.Q * /addr_0.Q * oe.Q * /we.Q * /reset * /state_0.Q
    + /addr_0.Q * oe.Q * /we.Q * /reset * burst * /state_0.Q
    + /addr_1.Q * addr_0.Q * oe.Q * /we.Q * /reset * /state_0.Q
    + oe.Q * /we.Q * /reset * /ready * /state_0.Q

we.D =
      /addr_1.Q * /addr_0.Q * /oe.Q * we.Q * /reset * /ready *
      /state_0.Q
    + /addr_1.Q * /addr_0.Q * /oe.Q * /we.Q * /reset * /read_write *
      state_0.Q
```

We've spent considerable time explaining this design technique and its benefits, so we'll reiterate the drawbacks. First is the extra time and effort required to choose a state encoding that maps well to your state machine's outputs. Second is the loss of readability (and therefore, ease of maintenance and debugging) of the original code. Also, more product terms are typically required. Eventually, synthesis optimization may allow you to use the code of *Listing 5-2* to achieve this implementation. Until then, you will have to decide when to use the different state machine design techniques based on your goals.

To this point, all of the equations that have been generated are for D-type flip-flops. Most CPLD macrocells can also be configured to implement T-type flip-flops. Using T-type flip-flops is transparent to the user. The report file would simply indicate their usage.

One-Hot Encoding

One-hot encoding is a technique that uses an n-bit-wide vector (i.e., *n* flip-flops) to represent a state machine with *n* different states. Each state has its own flip-flop, and only one flip-flop of the vector—the one corresponding to the current state—will be "hot" (hold a '1') at any given time. Decoding the current state is as simple as finding the flip-flop containing a '1', and changing states is as simple as changing the contents of the flip-flop for the new state from a '0' to a '1' and the flip-flop for the old state from a '1' to a '0'.

The primary advantage of one-hot-encoded state machines is that the number of gates required to decode state information for outputs and for next-state transitions is usually much less than the number of gates required for those purposes when the states are encoded in other fashions. This difference in complexity becomes more apparent as the number of states becomes larger. We'll use

an example to illustrate this point: Consider a state machine with 18 states that are encoded with 2 methods, sequential and one-hot, as shown below.

State	Sequential	One-Hot
state0	00000	000000000000000001
state1	00001	000000000000000010
state2	00010	000000000000000100
state3	00011	000000000000001000
state4	00100	000000000000010000
state5	00101	000000000000100000
state6	00110	000000000001000000
state7	00111	000000000010000000
state8	01000	000000000100000000
state9	01001	000000001000000000
state10	01010	000000010000000000
state11	01011	000000100000000000
state12	01100	000001000000000000
state13	01101	000010000000000000
state14	01110	000100000000000000
state15	01111	001000000000000000
state16	10000	010000000000000000
state17	10001	100000000000000000

Eighteen registers are required for one-hot encoding, whereas only five registers are required for the sequential encoding. With one-hot, only one register is asserted at a time. To continue our example, suppose that *Figure 5-10* represents a portion of the state flow diagram that depicts all possible transitions to state 15.

The code fragment that represents this state flow is

```
case current_state is
      when state2 =>
         if cond1 = '1' then next_state <= state15;
         else ...

      when state15 =>
         if ...
         elsif cond3 = '0' then next_state <= state15;
         else ...
```

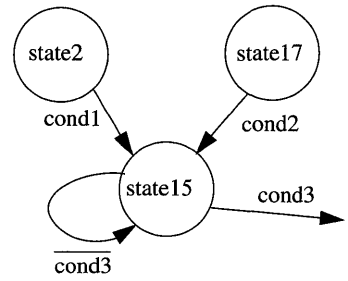


Figure 5-10 State flow diagram

```
when state17 =>
   if ...
   elsif cond2 = '1' then next_state <= state15;
   else ...
```

The IF-THEN constructs would be completed, of course, to specify the state transitions for conditions not shown in the state flow diagram above, but the code is suitable for our purpose of differentiating sequential and one-hot coding effects on state-transition logic.

We examine the next-state logic for sequential encoding first. The sequential encoding for *state15* is "01111". We refer to the state vector as *s*, so s_4 is 0; s_3, s_2, s_1, and s_0 are all 1 for *state15*. Using the state flow diagram of *Figure 5-10*, we can write an equation that represents the conditions in which s_i is asserted due to a transition to *state15*. In the equation, each state must be decoded (e.g., *state2* as "00010", *state17* as "10001", and *state15* as "01111").

$$\begin{aligned} s_{i,15} = & \overline{s_4} \cdot \overline{s_3} \cdot \overline{s_2} \cdot s_1 \cdot \overline{s_0} \cdot cond1 \\ & + s_4 \cdot \overline{s_3} \cdot \overline{s_2} \cdot \overline{s_1} \cdot s_0 \cdot cond2 \\ & + \overline{s_4} \cdot s_3 \cdot s_2 \cdot s_1 \cdot s_0 \cdot \overline{cond3} \end{aligned}$$

Although this equation defines s_i for transitions to *state15*, this equation is not sufficient to specify s_0, s_1, s_2, *and* s_3. Take s_0 for example: The equation for $s_{i,15}$ above only covers the cases in which s_0 is asserted due to transitions to *state15*. The state s_0 is asserted due to transitions to eight other states (all of the odd states). The logic equation associated with transitions to each of the other eight states may be of similar complexity to the equation for $s_{i,15}$ above. To obtain the complete equation for s_i, each of the $s_{i,x}$ equations (where x is an integer from 0 to 17, representing the 18 states) must be summed. You can imagine that the logic for s_0 can be quite complex, even for a relatively simple state flow diagram, simply because s_0 is 1 for nine states. The sequential encoding will create five complex equations for s_i.

Compare the amount of logic required to implement s_0, s_1, s_2, s_3, and s_4 for a sequentially encoded state machine to the logic required for a one-hot coded state machine in which the decoding logic for *state 15* is shown below. A state vector t is used so as not to confuse the equation with that of *s* above. The vector t is 15 bits wide.

$$t_{15} = t_2 \cdot cond1 + t_{17} \cdot cond2 + t_{15} \cdot \overline{cond3}$$

The equation for t_{15} can be derived easily from the state flow diagram. *Figure 5-12* shows the logic implementation for transitions to *state15*. Whereas the sequential encoding requires five complex

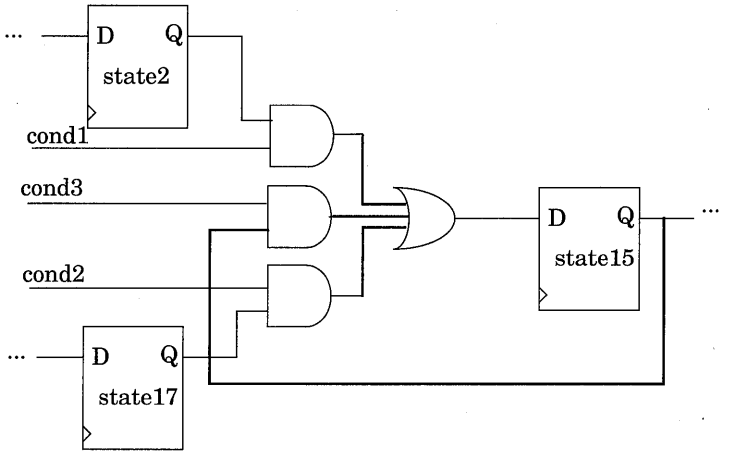


Figure 5-11 State transition logic for one-hot coded state machines is simple.

equations for the next-state logic, the one-hot encoding requires eighteen simple equations. Depending on the target logic device architecture, a one-hot coded state machine may require significantly fewer device resources for implementation of the design. Simple next-state logic may also require fewer levels of logic between the state registers, *allowing for higher frequency of operation.*

One-hot encoding is not always the best solution, however, mainly because it requires more flip-flops than a sequentially encoded state machine. In general, one-hot encoding is most useful when the architecture of the programmable logic device you want to use has relatively many registers and relatively little combinational logic between each register. For example, one-hot encoding is most useful for state machines implemented in FPGAs, which generally have much higher flip-flop densities than CPLDs, but which also have fewer gates per flip-flop. One-hot encoding may be the best choice even for CPLDs if the number of states and input conditions are such that the next-state logic for an encoded machine requires multiple passes through the logic array.

Up to now, we have discussed only what one-hot encoding is and the motivation for using it but not how to code one-hot state machines. Fortunately, using one-hot encoding requires little or no change to the source code but depends on the synthesis software tool you are using. Many synthesis tools allow you to use an enumerated type and specify the state encoding with an attribute, directive, command line switch, or GUI option. For example, we are using the *Warp* synthesis software to implement our memory controller as a one-hot state machine, so we can simply apply a state encoding attribute as shown below. The remaining code is the same as that of *Listing 5-2*:

```
type StateType is (idle, decision, read1, read2, read3, read4, write);
attribute state_encoding of StateType:type is one_hot_one;
signal current_state, next_state : StateType;
```

Generating outputs for one-hot encoded state machines is similar to generating outputs for machines in which the outputs are decoded from the state registers. The decoding is quite simple of course, because the states are just single bits, not an entire vector. This logic consists of an OR gate because Moore machines have outputs that are functions of the state and all states in a one-hot coded state machine are represented by one bit. The output decoding adds a level of combinational logic and the associated delay, just as it did before when the state bits were encoded. In an FPGA, the delay associated with the OR gate is typically acceptable and is an improvement upon decoding the outputs from an entire state vector. In a CPLD, the OR gate requires a second pass through the logic array, which means that the outputs will be available in t_{co2} time. The outputs can also be generated using parallel decoding registers, as was described earlier. This will eliminate the level of combinational logic and associated delay.

If any outputs are asserted in one state only, then these outputs are automatically encoded within the state bits. For example, *we* (write enable) is asserted only during the state *write*, expressed as

```
we <= '1' when present_state = write else '0';
```

There is a flip-flop directly associated with the state *write*, so *we* will be the value of that flip-flop. Consequently, *we* will be available at the device pins or for internal logic without the additional delay associated with decoding.

Mealy State Machines

So far we have discussed only Moore machines in which the state machine outputs are strictly functions of the current state. Mealy machines may have outputs that are functions of the present-state and present-input signals, as illustrated in *Figure 5-12*.

The additional task of describing Mealy machines versus Moore machines is minimal. To implement a Mealy machine, you simply have to describe an output as a function of both a state bit and an input. For example, if there is an additional input to the memory controller called *write_mask* that when asserted prevents *we* from being asserted, you can describe the logic for *we* as

```
if (current_state = s6) and (WRITE_MASK = '0') then
   WE <= '1';
else
   WE <= '0';
end if;
```

This now creates *we* as a Mealy output

The design techniques used previously to ensure that the output is available in t_{co} instead of t_{co2} cannot be used with a Mealy machine because the outputs are functions of the present inputs and the present state.

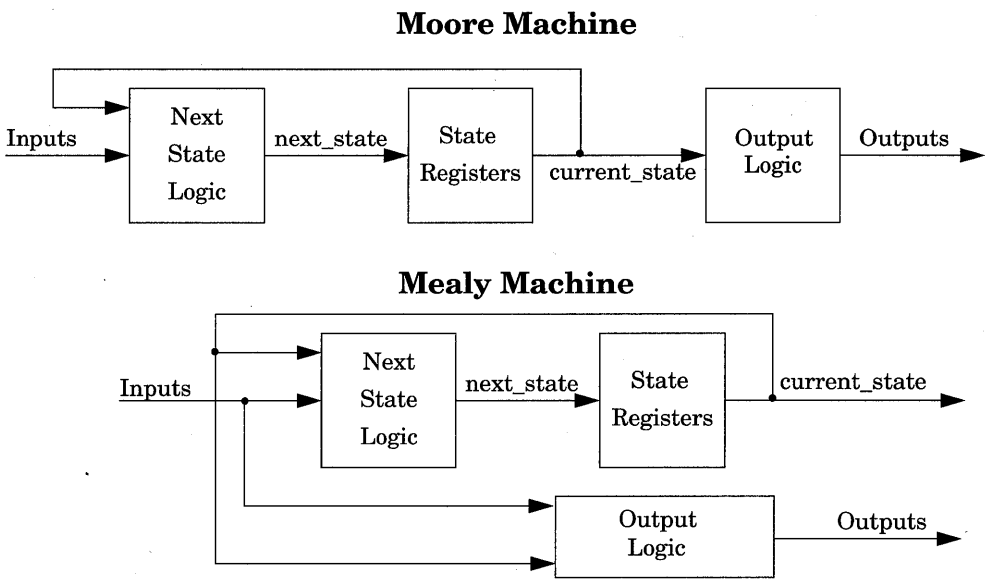


Figure 5-12 Moore and Mealy machines

Additional Design Considerations

State Encoding using Enumeration Types

Enumerated types provide an easy method to code state machines. When synthesized, state values must correspond to values held in state registers. For one-hot machines, each state value corresponds to one flip-flop. For other encodings, such as sequential, the minimum number of states required is the greatest integral value (the ceiling) of $\log_2 n$, $[\log_2 n]$, where n is the number of states. With these encodings, if $\log_2 n$ is not an integral value, then there will be undefined states. In our memory controller, we define seven states:

```
type StateType is (idle, decision, read1, read2, read3, read4, write);
signal state : StateType;
```

Using sequential encoding, *state* will require three flip-flops. Three flip-flops, however, can define eight unique states, so the sequential state encoding table for *state* is as shown in the table below. The vector representing the state encoding will not be referred to as *state* because *state* has already been defined above as an enumerated type. Instead, we will use *q* to represent the state vector.

State	q_0	q_1	q_2
idle	0	0	0
decision	0	0	1

State	q_0	q_1	q_2
read1	0	1	0
read2	0	1	1
read3	1	0	0
read4	1	0	1
write	1	1	0
undefined	1	1	1

Implicit Don't Cares

For the examples in this chapter that make use of an enumerated type, synthesis assumes that the value "111" for *q* is a don't care condition. No transition is defined for this state. Therefore, if the state machine were ever to get into this undefined, or "illegal" state, the state machine would not function in a predictable manner. The behavior of the state machine, once it is placed in an illegal state, will depend on the state transition equations. The advantage in having the illegal state as a don't care state is that additional logic is not required to ensure that the state machine will transition out of this state. This additional logic can require substantial device resources for implementation, especially if there are a large number of undefined states. The disadvantage of using these implied don't cares is that the state machine is less fault tolerant. The designer should make a conscious decision about whether this is acceptable for a particular design.

Fault Tolerance: Getting Out of Illegal States

The state "111" is an illegal state for our memory controller, but in hardware, glitches, ground bounce, noise, power up, or illegal input combinations may cause one of the flip-flop values to change, causing the state machine to enter an illegal state. If this happens, the state machine will not respond predictably, which may cause problems in your system. The machine may enter an illegal state and stay in that state permanently, or, among other possibilities, it may assert outputs that are illegal, or even harmful: Signals may cause bus-contention or a device to sink or source too much current, thereby destroying it.

State machines can be made more fault tolerant by adding code that ensures transitions out of illegal states. First, you will need to determine how many illegal states are possible. The number of illegal states is the number of states in the state machine subtracted from the number of bits (flip-flops) used to encode the state machine raised to the power of two. For our memory controller, there is one undefined state. Next, we will have to include a state name in the enumerated type for each undefined state. For example,

```
type StateType is (idle, decision, read1, read2, read3, read4,
                   write, undefined);
```

Finally, a state transition must be specified for the state machine to transition out of this state. This can be specified as

```
case state is
      ...
       when undefined  => state <= idle;
end case;
```

This state transition will require additional logic resources. Compare the equations below to those for *Listing 5-4*. Several additional product terms are required.

```
stateSBV_0.D =
      /stateSBV_0.Q * /stateSBV_1.Q * stateSBV_2.Q * /reset *
      /read_write
    + /stateSBV_0.Q * stateSBV_1.Q * stateSBV_2.Q * /reset * ready
    + stateSBV_0.Q * /stateSBV_2.Q * /reset * /ready
    + stateSBV_0.Q * /stateSBV_1.Q * /reset * /ready
    + stateSBV_0.Q * /stateSBV_1.Q * /stateSBV_2.Q * /reset

stateSBV_1.D =
      /stateSBV_0.Q * stateSBV_1.Q * /stateSBV_2.Q * /reset * burst
    + /stateSBV_0.Q * stateSBV_2.Q * /reset * /ready
    + stateSBV_1.Q * /stateSBV_2.Q * /reset * /ready
    + /stateSBV_0.Q * /stateSBV_1.Q * stateSBV_2.Q * /reset

stateSBV_2.D =
      /stateSBV_0.Q * /stateSBV_1.Q * /stateSBV_2.Q * /reset *
      bus_id_7 * bus_id_6 * bus_id_5 * bus_id_4 * /bus_id_3 *
      /bus_id_2 * bus_id_1 * bus_id_0
    + /stateSBV_0.Q * stateSBV_1.Q * /stateSBV_2.Q * /reset * ready *
      burst
    + stateSBV_0.Q * /stateSBV_1.Q * stateSBV_2.Q * /reset * /ready
    + /stateSBV_0.Q * stateSBV_1.Q * stateSBV_2.Q * /reset * /ready
    + stateSBV_0.Q * /stateSBV_1.Q * /stateSBV_2.Q * /reset * ready
```

If there are multiple states that are left undefined, then you can follow the same process:

```
type states is (s0, s1, s2, s3, s4, u1, u2, u3);
signal state: states;
...
case state is
   ...
   when others => state <= s0;
end case;
```

In this example, there are three undefined states (designated as *u1*, *u2*, and *u3*). Rather than specifying all three individual states, *when others* may be used to specify to transition to the same state.

Rather than returning to an idle or arbitrary state, you may want to create an error state or states to handle the fault, and have illegal states transition to the error state(s).

To have direct control over the state encoding and the enumeration of undefined states, you can use constants to define the state, as in *Listing 5-6* and described below.

Some synthesis tools provide the ability to define the state encoding with the use of attributes or directives. In this case, if you wish to make your design fault tolerant, you should make states for all possible values of the encoding bits. Some synthesis tools may also provide attributes or directives with which the designer can indicate that undefined states are don't cares or that there is a default state to transition to from undefined states.

Explicit State Encoding: Don't Cares and Fault Tolerance

State machine designs with explicit state encoding such as the one of *Listing 5-6*, in which constants are used to define states, must explicitly declare don't cares. The following state encoding defines seven states (however, *state* can have 32 unique values):

```
signal state : std_logic_vector(4 downto 0);
constant idle    : std_logic_vector(4 downto 0) := "00000";
constant decision: std_logic_vector(4 downto 0) := "00001";
constant read1   : std_logic_vector(4 downto 0) := "00100";
constant read2   : std_logic_vector(4 downto 0) := "01100";
constant read3   : std_logic_vector(4 downto 0) := "10100";
constant read4   : std_logic_vector(4 downto 0) := "11100";
constant write   : std_logic_vector(4 downto 0) := "00010";
```

To create a fault-tolerant state machine, then you must specify a state transition for the other 25 unique states:

```
when others => state <= idle;
```

Explicit Don't Cares

In specifying that illegal states will transition to a known state, additional logic is required. Depending on the cost of the solution versus the need for fault tolerance, the additional logic may not be worth the cost. In this case, rather than specifying that all other states should transition to a known state, you explicitly declare that the state transition is a don't care condition (i.e., you don't care what state it transitions to because you are not designing the state machine to be fault tolerant). You can define this as follows:

```
when others => state <= "-----";
```

Five don't care values are assigned to the signal *state* (*state* is a std_logic_vector(4 downto 0)). Whereas don't care conditions are implicit in state machine designs that make use of enumerated types that don't completely specify all possible combinations of values, the don't cares must be explicitly defined for state machines designed with explicit state encoding, because there are several combinations of metalogic values required to complete the case statement. Using constants allows you to explicitly define both the don't care conditions as well as the transitions from illegal states.

Fault Tolerance for One-Hot Machines

The potential for entering illegal states is magnified when you use outputs encoded within state bits or one-hot encoding. Both of these techniques can result in many more potential states than the ones you really use. With one-hot encoding, for example, there are 2^n possible values for the n-bit state vector. The state machine has only n states. Here we have a dilemma: One-hot encoding is usually chosen to achieve an efficient state machine implementation, but including logic that causes all of the illegal states to transition to a reset state (or other known state) creates an inefficient state machine implementation. To completely specify state transitions for a one-hot coded state machine that has 18 states, another 2^{18}-18 transitions would have to be decoded. This is an enormous amount of logic. Alternatively, rather than adding logic to transition out of all possible states, you can include logic that detects if more than one flip-flop is asserted at a time.

A collision signal can be generated to detect multiple flip-flops asserted at the same time. For eight states, the collision signal is as below. The same technique can be extended for any number of states. The states below are enumerated as state1, state2, etc.:

```
colin        <= (state1 AND (state2 OR state3 OR state4 OR
                  state5 OR state6 OR state7 OR state8)) OR

                 (state2 AND (state1 OR state3 OR state4 OR
                 state5 OR state6 OR state7 OR state8)) OR

                 (state3 AND (state1 OR state2 OR state4 OR
                 state5 OR state6 OR state7 OR state8)) OR

                 (state4 AND (state1 OR state2 OR state3 OR
                 state5 OR state6 OR state7 OR state8)) OR

                 (state5 AND (state1 OR state2 OR state3 OR
                 state4 OR state6 OR state7 OR state8)) OR

                 (state6 AND (state1 OR state2 OR state3 OR
                 state4 OR state5 OR state7 OR state8)) OR

                 (state7 AND (state1 OR state2 OR state3 OR
                 state4 OR state5 OR state6 OR state8)) OR

                 (state8 AND (state1 OR state2 OR state3 OR
                 state4 OR state5 OR state6 OR state7)) ;
```

It may not be necessary for the collision signal to be decoded in one clock cycle, in which case you may want to pipeline such a signal.

Entering an illegal state is usually a catastrophic fault in that it may require a system to be reset. Depending on the system requirements, it may not be necessary to reset the machine in one clock cycle, in which case the collision signal can be pipelined, if necessary, to maintain the maximum operating frequency. In all cases, you need to make a conscious design decision regarding fault tolerance. For one-hot encoding, fault tolerance is at odds with performance and resource utilization. You must decide how much speed or area (device resources) you can give up to make your design more fault tolerant, and if you decide not to include logic for fault tolerance, then you must be aware of the ramifications.

Incompletely specified IF-THEN-ELSE statements

In this section, we reiterate a point made in an early chapter about the use of IF-THEN-ELSE constructs. We have shown several methods for decoding state machine outputs. IF-THEN-ELSE statements may be used to decode state machine outputs or to create combinational logic in general. When using IF...THEN...ELSE statements, be careful to explicitly assign the value of signals for all conditions. Leaving ambiguities implies memory: If you do not specify a signal assignment, then you imply that the signal is to retain its value. For example, consider the following, perfectly legal VHDL code:

```
if (present_state = s0) then
   output_a <= '1';
elsif (present_state = s1) then
```

```
   output_b <= '1';
else      -- current_state = s3
   output_c <= '1';
end if;
```

This is how you may mistakenly write code if you intend *output_a* to be asserted only in state *s0*, *output_b* to be asserted only in state *s1*, and *output_c* to be asserted only in state *s2*. What this code really implies is that *output_a* is assigned '1' in state *s0*, and *it keeps whatever value it currently has in every other state. Figure 5-13* shows a logic implementation for this code. The logic shows that after *present_state* becomes *s0* once, the value of *output_a* is always a '1'.

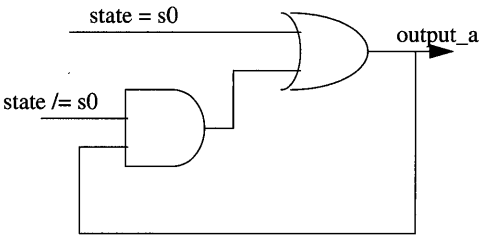


Figure 5-13 Implied memory

To avoid having a latch synthesized for *output_a*, *output_b*, and *output_c*, the above code should be rewritten, initializing the signals at the beginning of the process, as follows:

```
output_a <= '0';
output_b <= '0';
output_c <= '0';
if (current_state = s0) then
   output_a <= '1';
elsif (current_state = s1) then
   output_b <= '1';
else      -- current_state = s3
   output_c <= '1';
end if;
```

This will guarantee that the logic implementing these outputs is strictly a combinatorial decode of the state bits.

State Encoding for Reduced Logic

Earlier, we explained how to encode state machine outputs within state registers. Below, we reprint the partially completed state encoding table for one of the examples.

state	addr(1)	addr(0)	st1	st0
s0	0	0	0	0

state	addr(1)	addr(0)	st1	st0
s1	0	0	0	1
s2	0	0	1	0
s3	0	1		
s4	1	0		
s5	1	1		
s6	0	0	1	1

In explaining how to choose the state encoding, we arbitrarily chose to complete the table by filling in the empty entries with 0. This encoding may not be the optimal encoding. Another encoding may require fewer device resources. Choosing 0's may be adequate; at other times, you may need to experiment with the state encoding or use software that determines the optimal encoding for minimal logic (choosing an optimal state encoding is a nontrivial task). Some VHDL synthesis tools perform this function.

Summary

Writing VHDL code to implement a functionally accurate state machine is simple. You can translate state flow diagram using CASE-WHEN and IF-THEN-ELSE constructs. Often times, your initial design will satisfy all of your design requirements (time-to-market, performance, and cost-of-design). If the performance or cost-of-design requirements are not immediately met, you can quickly modify the state machine design to meet the timing requirements and to fit most efficiently into the target architecture.

Some of the techniques that we discussed in this chapter include

- decoding state machine outputs combinatorially from state registers.
- decoding state machine outputs in parallel output registers.
- encoding state machine outputs in the state registers.
- encoding state machines as one-hot.
- designing fault-tolerant state machines.

You are now cognizant of multiple coding styles, state encoding schemes, and design trade-offs, which will enable you to more quickly arrive at the optimal solution for your design.

Exercises

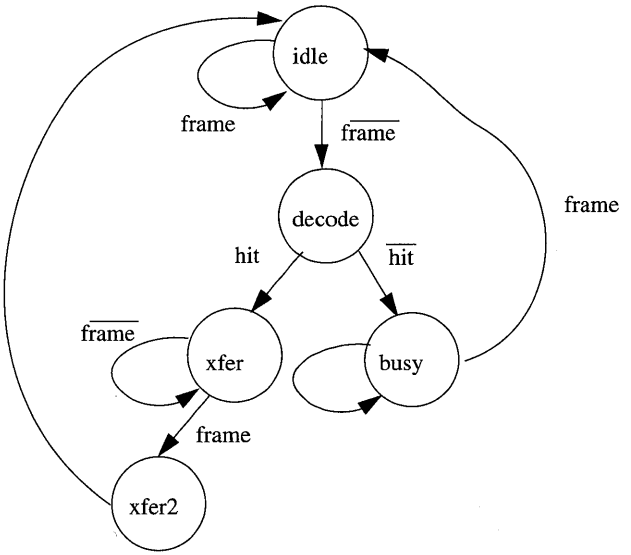
1. Design a state machine that detects the sequence "11010":
(a) Draw the state flow diagram.
(b) Code the design using sequential encoding.
(c) Compile, synthesize, and simulate the design.

2. Rewrite the code for Exercise 1, implementing the state machine as one-hot:
(a) Compile, synthesize, and simulate the design.
(b) Implement the design in an FPGA and a CPLD.

(c) Compare the performance and resource utilization for each device.
(d) Compare this implementation with the implementation of Exercise 1.

3. Design the state machine of *Figure 5-14*. Minimize the clock-to-output and internal register-to-register delays for the target architecture.

(a)



(b)

state	OE	GO	ACT
idle	0	0	0
decode	0	0	0
busy	0	0	1
xfer	1	1	1
xfer2	1	0	1

Figure 5-14 (a) State flow diagram and (b) state output table

4. The encoding for states *s2* through *s5* of *Listing 5-6* is such that the two least significant bits are arbitrarily "00". Implement the design of *Listing 5-6* with a different encoding for states *s2* through *s5* and compare the resource utilization.

5. Rewrite the VHDL code for the memory controller shown in page 137 to handle the synchronous reset in each state transition. Compare the merits and demerits of using this modified version instead of the original version, which used an asynchronous reset signal. Is there any advantage in using a signal both as a synchronous and asynchronous reset? Justify.

6. Rewrite exercise 3 with the first coding style presented on page 130. Repeat the exercise using the alternate implementation scheme presented in page 143.

7. Build a 16-state counter with terminal count using sequentially encoded states and one-hot encoded states. Draw the schematics for each, and compare to other known circuits (like shift registers).

8. Implement two versions of a 8-bit loadable counter with a terminal count output:

a) generates the value of the terminal count with tco

b) generates the value of the terminal count with tco2

Discuss performance and resources used in both schemes.

9. Discuss the differences between the following Moore machine implementations:

a) outputs decoded from state bits combinatorially

b) outputs decoded from state bits in parallel output resgisters

c) outputs encoded within state bits

Which do you think is the best implementation style for a 1) CPLD and 2) FPGA. Justify.

10. List atleast five different encoding strategies. What are the advantages and disadvantages of using one-hot encoding over the other strategies?

11. What are the differences between a one-hot-one and one-hot-zero encoding schemes?

12. Consider the state machine with 18 states illustrated in the listing in page 157. Encode the state machine using a gray-code encoding scheme. Discuss the merits of this scheme over sequential encoding.

13. List key differences between Moore and Mealy machines.

14. Determine the amount of logic for different assignments of the 'Don't Care' bits in listing shown.

State	addr(1)	addr(0)	st1	st0
idle	0	0	0	0
decision	0	0	0	1
read1	0	0	1	0
read2	0	1		
read3	1	0		
read4	1	1		

State	addr(1)	addr(0)	st1	st0
write	0	0	1	1

15. Discuss differences between Explicit and Implicit Don't Cares. Justify with an example the advantage of using Implicit Don't Cares.

16. Extend the collision detector shown on page 166 to include all 18 states. Rewrite the code using the FOR-GENERATE or the FOR-LOOP for the following intermediate signals:

colin1 = function(State0 .. .. State7)

colin2 = function(State4 .. .. State11)

colin3 = function(State9 .. .. State15)

colin4 = function(State13 .. .. State17)

and create colin_final = colin1 AND colin2 AND colin3 AND colin4

17. Write a detailed report on the different strategies adopted by VHDL EDA vendors to design fault tolerant state machines.

6 The Design of a 100BASE-T4 Network Repeater

In this chapter, we will work our way through the design of the core logic for a 100BASE-T4 network repeater. The design provides an excellent example of how programmable logic and VHDL can be used to quickly and efficiently design state-of-the-art networking equipment. At the time of this writing, 100BASE-T4 Ethernet is a standard that is gaining acceptance quickly.

Programmable logic allows for the "proof of concept" for designs like this one, as well as the fastest time-to-market and lowest initial cost. Time-to-market in the competitive electronics industry can mean the difference between success and failure. If a product is clearly successful, then production volumes may warrant a conversion to an ASIC, in which case the same VHDL code can be used.

The network repeater design will provide a practical example of design hierarchy. It also requires many of the design constructs that are frequently implemented with programmable logic: counters, state machines, multiplexers, small FIFOs, and control logic. This design serves to elaborate and put to use many of the concepts presented in previous chapters, and it does so with a "real-world" design. Because this is such a large design, we will use hierarchy, breaking the design into design units. We'll then build each of the design units from the bottom up. Along the way, we'll introduce a few new concepts such as generics and parameterized components.

Before discussing the design specifications for the network repeater, we will provide some background on Ethernet.

Background

Ethernet is the most popular Local Area Network (LAN) in use today for communication between work groups of computers, servers, peripherals, and networking equipment. Ethernet is the trade name for a set of networking standards adopted by the International Standards Organization (ISO) and IEEE 802.3 as the CSMA/CD Network Standard. The standard embodies specifications for network media, physical interfaces, signaling, and network access protocol. Initial work on Ethernet was pioneered in the early 1980's by researchers at Xerox, DEC, and Intel. Since then, Ethernet has evolved to meet market demands by including standards for new cabling types and faster data rates. A recent effort in the Ethernet standard working groups was to bring Ethernet to 100 megabits per second (Mb/s) over several media types. *Figure 6-1* illustrates a typical work group environment that is connected via several Ethernet media types.

Ethernet Networks

Ethernet allows the creation of flexible and inexpensive networks that accommodate different media types and hundreds of host station connections in a single network. Ethernet network architecture is based on a few simple concepts and configuration rules. The primary elements of the network are the medium, adaptor cards (transceivers), and hubs.

Architecture

Shared Medium

Ethernet is based on the idea that the network should behave as a shared medium. Every host station on the network listens to all network traffic and processes only data intended for it. Only one host

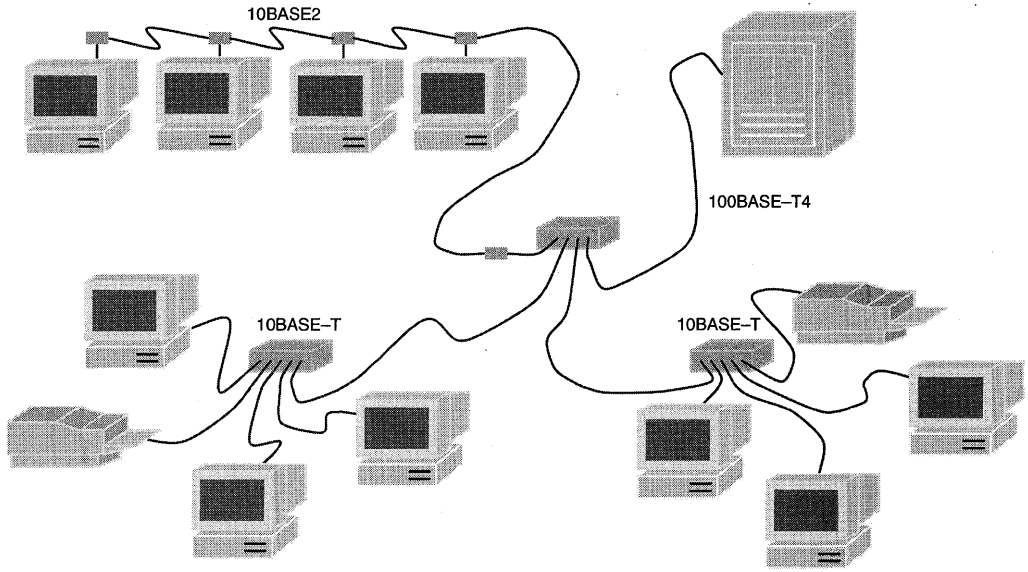


Figure 6-1 Typical Ethernet network

may transmit data at a time. Other hosts wishing to transmit must wait unit they detect that the medium is quiet before they can start transmitting data onto the network. If two hosts transmit at the same time, then a collision occurs and both hosts must back off of the network and try to resend their data after an arbitrary waiting period. This technique for accessing the network is called *Carrier Sense, Multiple Access with Collision Detection* (CSMA/CD) and is the distinguishing feature of Ethernet.

The concept of a shared medium is inherent in the coaxial implementations of Ethernet (10BASE5 and 10BASE2) for which a common coaxial cable connects to each host through a transceiver. In twisted-pair-based Ethernet (10BASE-T and 100BASE-T), each host attaches to a common repeater through a segment of twisted pair cable (see *Figure 6-1*). *The function of the repeater is to create the logical equivalent of a shared medium.*

Network Constraints

Several practical constraints limit the size of Ethernet networks. First is the number of hosts that may be connected to a single network, 1024, which is determined by the performance that can be achieved by that many hosts sharing the same transmission medium. In practice, networks are not this large.

The cabling type determines the maximum size of a network built from a single cable segment. For example, 10BASE2 limits the coaxial cable length to 185 meters. Repeaters may be used to connect network segments together to build a network that is larger in total diameter. Total network diameter is determined by the maximum tolerable round-trip delay that data can take through the network. This delay is equal to the minimum data-frame size that can be transmitted by any host (512 bits) and is called the slot time (For 10 Mb/s networks, the slot-time corresponds to approximately 2000

meters.) In a network that is large enough to violate the slot-time, it is impossible to determine if data is transferred collision free resulting in a broken network.

Adaptors and Transceivers

Adaptor cards interface computers, servers, or peripheral devices to the network medium. They are based on standard buses such as ISA, PCI, or PCMCIA. On an adaptor card, a Media Access Controller (MAC) builds data frames from user data, recognizes frames addressed to it, and performs media access management (CSMA/CD). Adaptor cards may have an RJ45 connector for twisted-pair- based Ethernet such as 10BASE-T or 100BASE-T4. They may also have an AUI (Attachment Unit Interface) connector for interfacing coaxial transceivers for 10BASE2 or 10BASE5; however, only one network connection may be active at a time. *Figure 6-2* shows a typical adaptor card.

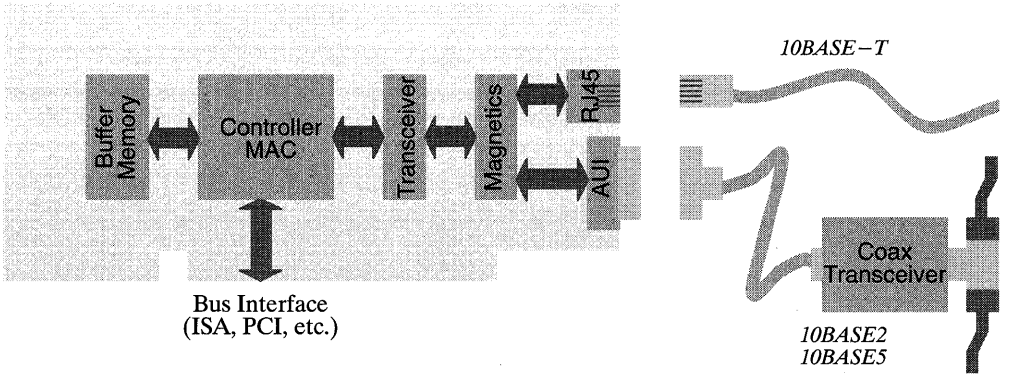


Figure 6-2 Typical Ethernet Adaptor Card

Hubs

Hub is a generic name for repeaters, bridges, and routers. Hubs are used to expand the size of the network and to allow interoperation with different media (coaxial cable, fiber optics, etc.) and different network types (Token Ring, FDDI, etc.). Smart hubs such as routers, learning bridges, and switches can route traffic between different networks based on addresses associated with the data. These devices ease network congestion and improve network bandwidth.

Ethernet over twisted pair requires the use of a hub (a repeater) in order to provide connectivity between all of the host cable segments.

Repeaters

Repeaters, the simplest type of hub, logically join cable segments to create a larger network while providing the equivalent of a shared medium. Repeaters improve reliability and performance because they isolate hosts with faulty network connections, keeping them from disrupting the network. Repeaters may have management functions so that remote hosts or a local terminal can configure the network or query network performance statistics. An Ethernet network built with repeaters is shown in *Figure 6-3*. The basic objectives for a repeater are listed below:

- Detect carrier activity on ports and receive Ethernet frames on active ports
- Restore the shape, amplitude, and timing of the received frame signals prior to retransmission
- Forward the Ethernet frame to each of the active ports
- Detect and signal a collision event throughout a network
- Extend a network's physical dimensions
- Protect network from failures of station, cable, port, etc.
- Allow installation (removal) of station without network disruption
- Support interoperability of different physical layers (10BASE2, 10BASE-T, etc.)
- Provide centralized management of network operation and statistics
- Provide for low-cost network installation, growth, and maintenance

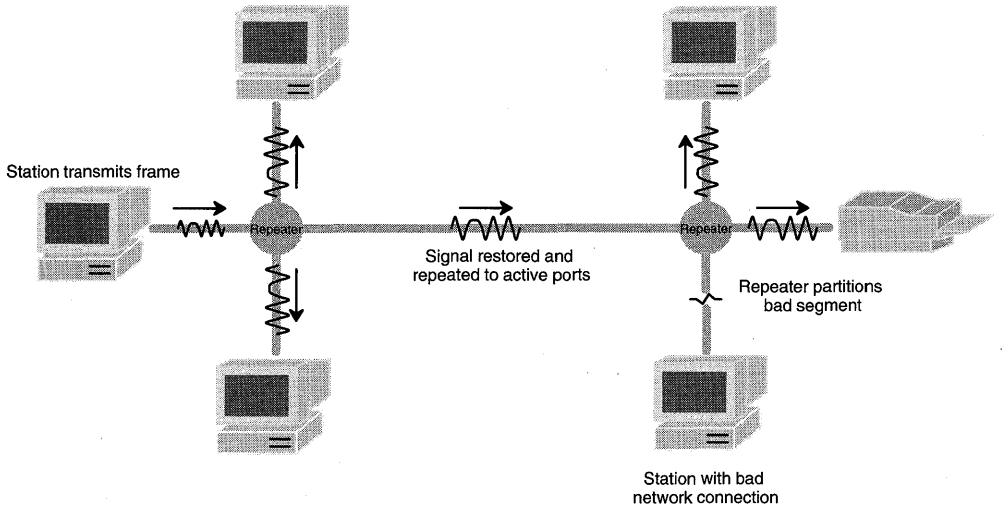


Figure 6-3 Ethernet Network Built with Repeaters

Although our focus will be on the functionality and design of a repeater, below we provide a brief description of bridges and routers to satisfy the curious reader. You may wish to proceed to "Design Specifications for the Core Logic of an 8-Port 100BASE-T4 Network Repeater" on page 177.

Bridges

A bridge is used to connect distinct networks (such as Token Ring, FDDI, and Ethernet). This requires that data be reframed for the standard data-frames of the appropriate networks. For example, an Ethernet network can be connected to a Token Ring network through a bridge that converts Ethernet frames to Token Ring frames. A bridge may also be used to connect distinct Ethernet networks to increase the overall network's physical dimension. A bridge used in this manner isolates the collisions between the attached networks and avoids the slot-time constraint. *Figure 6-4* shows a bridge used to connect dissimilar networks.

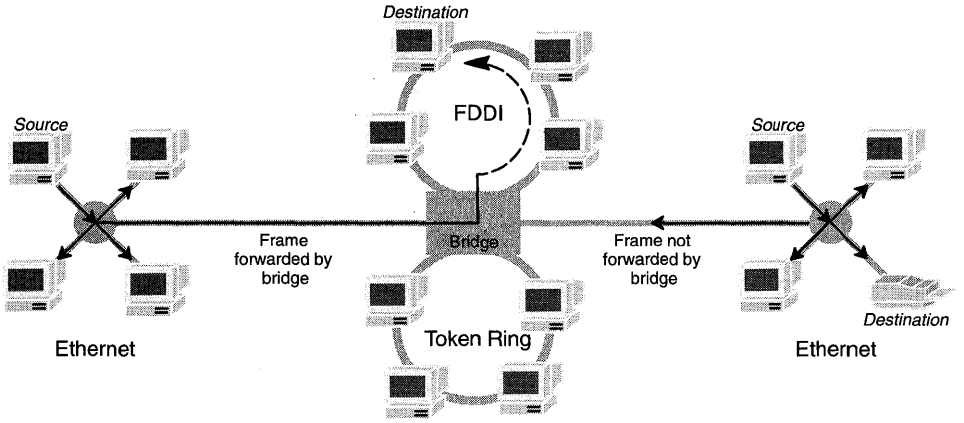


Figure 6-4 Network Connected with Bridges

Bridges may forward all frames between connected networks or they may selectively forward frames based on the frame's MAC address. To selectively forward frames, a bridge must have knowledge of the network addresses that can be accessed by each of the attached networks. *Learning* bridges monitor the network traffic in order to learn addresses dynamically.

Bridges also increase network bandwidth. A bridge will isolate Ethernet traffic inside a local work group so that bandwidth isn't wasted in attached networks. The other local work groups attached to the bridge do not have to wait for foreign traffic to clear before sending data. Only nonlocal traffic crosses the bridge en route to its destination.

Routers

Routers work much like bridges except that they *forward* a packet based on the Network Layer address. Routers are thus able to transfer packets between different network protocols and media types (e.g., TCP/IP -> X.25, Ethernet -> FDDI). *Figure 6-5* shows routers transferring data between different media types and network protocols.

Design Specifications for the Core Logic of an 8-Port 100BASE-T4 Network Repeater

Although we have made efforts to ensure the accuracy of the repeater core design, you should consult the IEEE 802.3 standard before attempting to use this design in a commercial product.

Interface

The basic function of a repeater is to retransmit data that is received from one port to all other ports. 100BASE-T4 is the 100 Mb/s Ethernet standard that operates over 4 pairs of category 3, 4, or 5 UTP (unshielded, twisted pair) cable. The architecture of a 100BASE-T4 repeater is that of *Figure 6-6*. Transceivers, such as the CY7C971, perform the electrical functions needed to interface the ports to

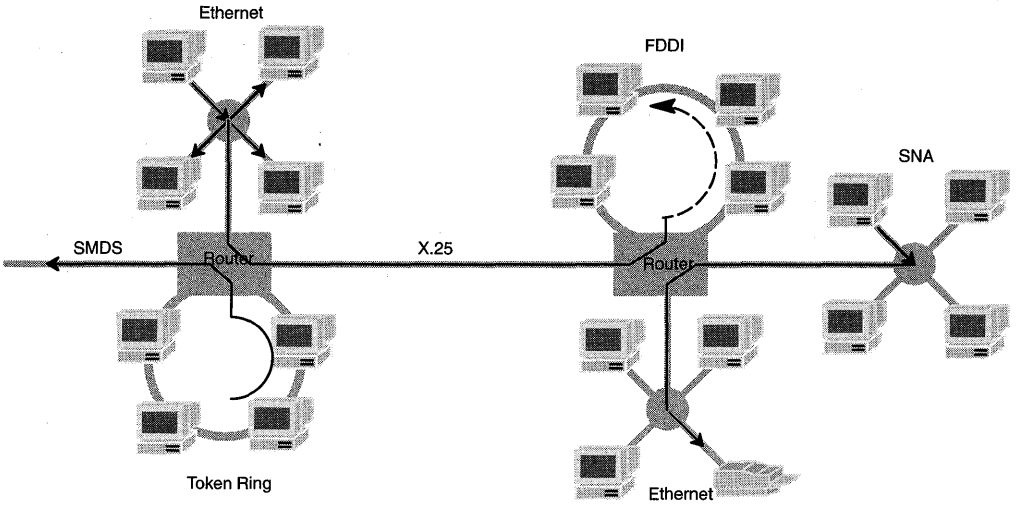


Figure 6-5 Network Connected with Routers

the repeater core logic. We will not be concerned with the functions and operation of the transceivers. Rather, we will focus on the functions of the repeater core. This requires that we understand the transceiver-repeater core interface.

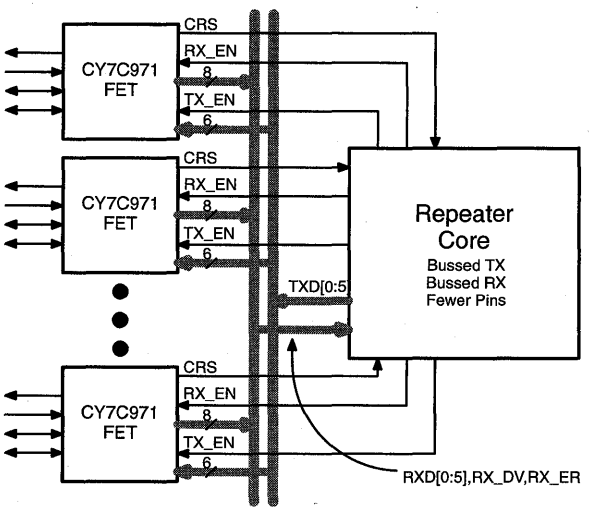


Figure 6-6 T4 repeater architecture

Figure 6-7 illustrates the transceiver interfaces to the repeater core. Each port is capable of receiving data and provides the following signals to the core logic:

- carrier sense (*crs*)
- receive clock (*rx_clk*)
- receive data valid (*rx_dv*)
- receive data error (*rx_er*)
- three pairs of data, *rxd0-rxd5*

Signal *crs* indicates that data is being received by the transceiver. *rx_clk* is the clock recovered from the incoming data by the transceiver; it is used to synchronize the incoming data, *rxd0-rxd5*. Signal *rx_dv* indicates that received data is valid. It is asserted at a data frame's start of frame delimiter (SFD) and is deasserted at the end of a frame. Signal *rx_er* indicates that the transceiver detected an error in the reception of data.

The core logic provides only one signal to the receive side of each transceiver: *rx_en (receive enable)* used to control which port is driving the bus. All ports share a common receive bus for *rxd0-rxd5, rx_dv,* and *rx_er* (*Figure 6-6*).

The core logic provides several signals to the transmit side of each port: *transmit clock (tx_clk), transmit enable (tx_en),* and *transmit data (txd0-txd5).*

An additional signal, *link integrity (link_bar),* indicates the integrity of the link between the repeater and a node on the network.

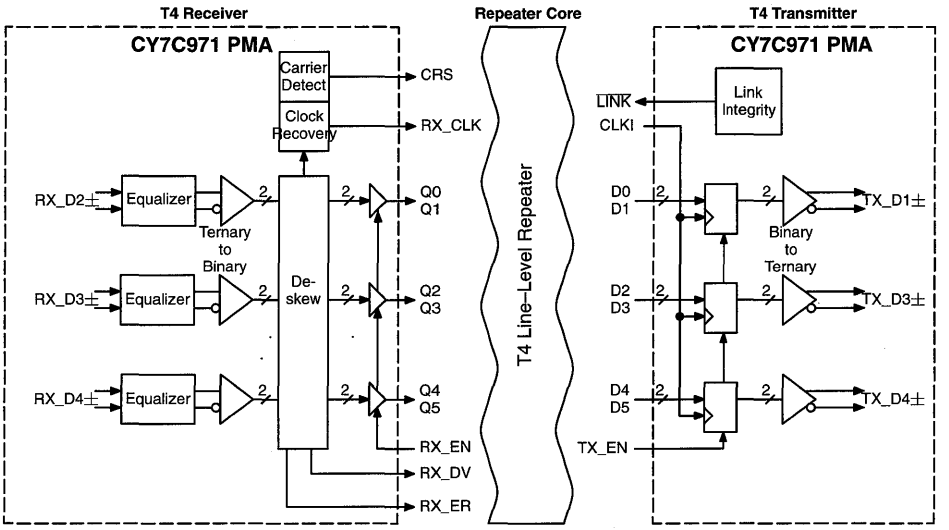


Figure 6-7 Transceiver interfaces to the repeater core

Protocol

Upon reception of a *carrier sense* (*crs*) from any port, the core logic must buffer the incoming data frame and retransmit this frame to all other functional ports, provided that (1) there is not a collision, (2) the port is not *jabbering*, and (3) the port is not partitioned.

A collision occurs if at any time more than one carrier sense becomes active, in which case a *jam* symbol must be generated and transmitted to all ports, including the one previously sending data. Those nodes that caused the collision will cease sending data, wait for an arbitrary length of time, and then attempt to resend data when the network is quiet.

If *crs* is not asserted by any of the ports, then idle symbols are generated and transmitted on all ports.

If the receiving port asserts *rx_er* while the repeater is retransmitting data to the other ports, then *bad* symbols (symbols that indicate that the transmission has gone bad) are generated and transmitted to all other ports until *crs* is deasserted or a collision occurs.

A port is *jabbering* if it continually transmits data for 40,000 to 75,000 bit times. If a port is jabbering, the repeater will inhibit *rx_en* (i.e., it will stop receiving data from this port). Therefore, it will not retransmit data from this port to the other ports. Instead, the repeater will free up the network for other ports to send data, retransmitting this data to all other ports except the one jabbering. The port will be considered to have ceased jabbering after *crs* is deasserted.

A port must be partitioned from the network if it causes 60 or more consecutive collisions, because continued collisions will bring to a halt all network communication. A broken cable or faulty connection is a likely source of these collisions. When a port is partitioned, the repeater will stop receiving data from that port. It will, however, continue to transmit data to this port. A partitioned port will be "reconnected" to the repeater if activity on another port occurs for 450 to 560 bit times without a collision.

The transceiver detects a carrier by identifying the preamble of the incoming data frame. By the time the carrier is detected and a clock recovered, some of the preamble is lost, but the data to be transmitted to the other ports has not been lost. Before retransmitting the data, however, the preamble must be regenerated so that receiving nodes can sense a carrier and recover the clock prior to receiving the actual data being transmitted. The frame structure is described below for the curious reader.

Data Frame Structure

Data transmitted on the Ethernet network is encapsulated in standard frames. The frame format is shown in *Figure 6-8*. A description of the frame components is given below:

- Preamble: The preamble is used by the receiving hosts to detect the presence of a carrier and initiate clock recovery.
- Start of Frame Delimiter: The start of frame delimiter (SFD) indicates to the receiving hosts that the next group of bits is the actual data to be transmitted.
- Destination Address: The destination address is a 48-bit address that uniquely identifies which host on the network should receive the frame. A host address is created by taking the 24-bit OUI (Organizationally Unique Identifier) assigned to each organization. The remaining 24 bits are determined internally by network administrators.
- Source Address: The source address is a 48-bit address that uniquely identifies which host is

sending the frame.

- Length: The length field is two bytes and determines how many bytes of data are in the data field.
- Data: The minimum data-field size is 46 bytes. If fewer than 46 bytes of data need to be sent, then additional characters are added to the end of the data field. The maximum data field size is 1500 bytes.
- Frame Check Sequence (FCS): The FCS is a 32-bit CRC (cyclic redundancy check) computed from a standard CRC polynomial. The FCS is computed over all fields except the preamble, SFD, and FCS. The receiving host computes a CRC from the bits it receives and compares the value to the FCS embedded in the frame in order to see if the data was received error free.

In order to transmit 100 Mb/s over 4 pairs, the frequency of operation must be 25 MHz. The repeater does not ensure that the minimum frame size is sent by a node but merely retransmits the data ("good data in, good data out; garbage in, garbage out").

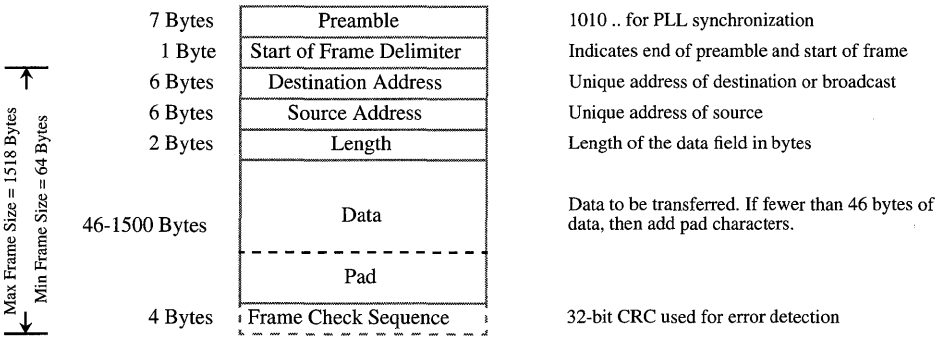


Figure 6-8 Ethernet MAC data frame

Block Diagram

We can summarize the functions of a repeater as this: (1) In general, the repeater receives data from one port and retransmits that data to the other port, (2) it detects collisions, no activity, and errors, generating and transmitting the appropriate symbols under these conditions, and (3) it detects jabbering and partition conditions, asserting *tx_en* and *rx_en* appropriately. To accomplish these functions, the incoming data must be buffered and the symbols generated. The buffered data must be multiplexed with other symbols, depending on which data should be transmitted to the active ports. From this summary, we can construct the block diagram of *Figure 6-9* to illustrate the top-level functionality of the core logic. At this point, we'll break this design down into manageable design units. Breaking a design into manageable units is the concept of creating a *hierarchy*. The advantages of hierarchy are that it allows you to (1) define the details of one portion of the design at a time (preferably, in parallel with other engineers), (2) focus on a manageable portion of a system, leading to fewer initial errors and quicker debugging time, and (3) verify small portions of the design at a time (if a VHDL simulator is available). The constituent pieces of the design can be interfaced at each level of the hierarchy and verified until the entire system is built up.

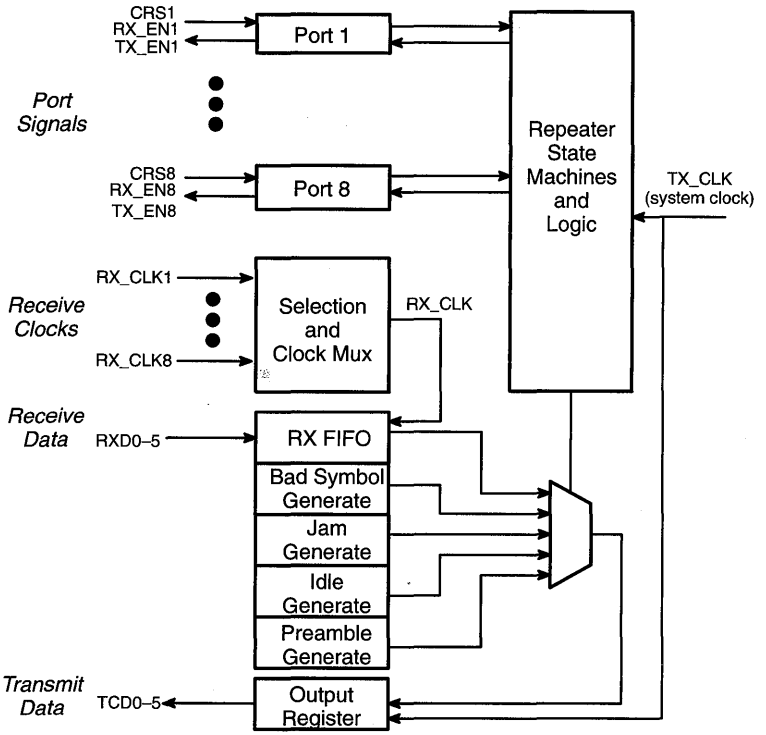


Figure 6-9 Block Diagram of the Repeater Core Logic

The block diagram of *Figure 6-9* and the design specifications give us some good clues as to how we may want to structure the hierarchy. Each port interfaces to a transceiver that, in turn, interfaces to the core logic. Thus, the logic for each port is clearly a good candidate for a design unit, as we do not want to redesign it seven times. We will also choose to make design units for the repeater state machines and the FIFO. The port selection and clock multiplexer will be divided into two design units, one for each task. The final design unit will be a symbol generator and output multiplexer. We can start to draw out our first level of hierarchy, but we first need to more clearly define the function of each unit so that we can define the interfaces between the units and top-level I/O. After reading through the descriptions of the following units, you will see that *Figure 6-10* represents the first level of hierarchy for the design of the core logic of this network repeater.

Port Controller

There will be eight port controllers, one for each port. On the receive side, each port controller will synchronize *crs, link_bar*, and *enable_bar* to *tx_clk*. Once a carrier has been detected and synchronized, *activity* is asserted. This signal is used by the arbiter to select a port from which to receive data. *Rx_en* is asserted by a port controller if its port is selected as the receiving port and

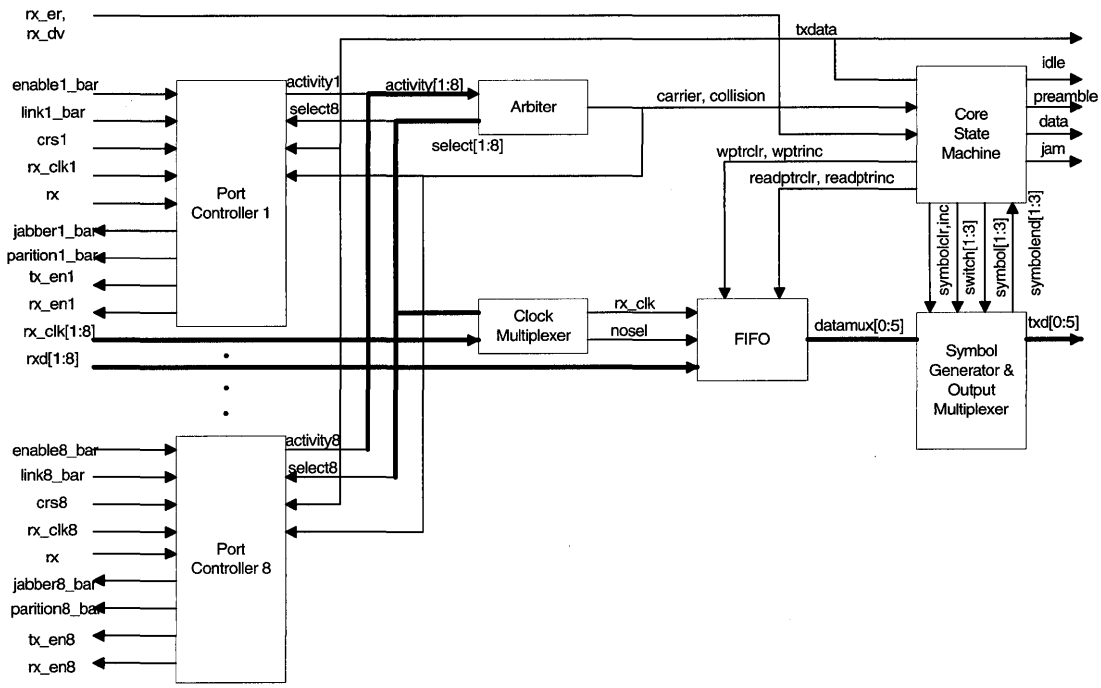


Figure 6-10 First level of hierarchy

there is not a collision. On the transmit side, each port will assert *tx_en* if the core controller indicates that *txdata* (transmit data) is ready, provided that this port is not the receiving port (*link_bar* must also be present and the port cannot be jabbering). *Tx_en* will also be asserted in the event of a collision so that *jam* characters may be transmitted to all hosts. *Jabber_bar* and *partition_bar* are driven by the port controller to indicate that the port is either jabbering or partitioned. These signals can be used to light LEDs. To determine if the port is jabbering, the port controller uses a timer to determine how long *crs* has been asserted. If *crs* has been asserted 40,000 to 75,000 bit times, then *jabber_bar* is asserted until *crs* is deasserted. To determine if the port should be partitioned, the port controller counts the number of consecutive collisions. If the number of collisions is greater than 60, then *partition_bar* is asserted to prevent the port from halting all network communication. To determine if the port should be reconnected after being partitioned, the port controller monitors port activity. If another port is active for more than 450 to 560 bit times without this port causing a collision, then the port controller deasserts *partition_bar*. These conditions will be determined by the state of *collision* and *carrier* as well as whether or not the port is *selected*. An additional input, *enable_bar*, will be usable by the implementers of the repeater core logic. In the event that eight ports are not required, *enable_bar* may be hard-wired as deasserted for the unused ports.

Arbiter

The arbiter will use the *activity* signals of each of the port controllers to supply eight *selected* signals to the port controllers and clock multiplexer. These signals will indicate which port is receiving data.

They will be used to gate the *rx_en* of that port and to choose the appropriate clock for writing to the FIFO. The arbiter also supplies *carrier* and *collision* for use by the port controllers and the core controller. *Nosel* is supplied to the clock multiplexer, indicating that no port is receiving a transmission.

Clock Multiplexer

The inputs to the clock multiplexer are the eight receive clocks *(rx_clk7-rx_clk0)*, the eight *selected* lines from the arbiter, and *nosel* from the arbiter. The *selected* and *nosel* signals are used to select one of the receive clocks as *the* receive clock, *rx_clk*, for use by the FIFO.

FIFO

The FIFO will capture incoming data on the receive side, storing six bits of data *(rxd5-rxd0)* on the rising edge of *rx_clk. Wptrclr* (write-pointer clear), *wptrinc* (write-pointer increment), *rptrclr* (read-pointer clear), and *rptrinc* (read-pointer increment) are used to advance or clear the FIFO and to indicate which register to read for the outputs *dmuxout5-dmuxout0.*

Symbol Generator and Output Multiplexer

The character symbol generator and output multiplexer will generate symbols. These symbols are the *bad* characters (transmitted to indicate a receive error), *jam* characters (transmitted to indicate a collision), *idle* characters (transmitted to indicate that there is no activity on the network), and *preamble* characters (transmitted to allow for carrier sensing and clock recovery by the receiving nodes). There are six output multiplexers (one for each of the transmit signals of the three transmit pairs). The multiplexers are paired to the transmit pairs, with each pair sharing the same select lines. There are five inputs to each multiplexer, so three select lines per pair are required (for a total of nine select lines for all multiplexers). The outputs are the transmit signals *txd5-txd0.*

Core Controller

The core controller controls the FIFO read and write pointers as well as symbol generation. It also asserts *tx_data* to indicate to the ports that data is ready. The core controller determines what data to transmit: data in the FIFO, or preamble, idle, jam, or bad characters. To do this, the core controller requires *carrier*, *collision*, *rx_dv*, and *rx_er* as inputs and asserts the FIFO and multiplexer control lines.

Before proceeding to design each of the individual units, we need to know how the signals will be synchronized. In this design, there are basically five input signals (*crs, rx_dv, rx_er, link_bar,* and *enable_bar*) that will need to be synchronized to the system transmit clock, *txclk.* These signals will be synchronized with two registers in order to increase the MTBF (mean time between failures). The incoming data is aligned to the incoming clock *rxclk* and stored in a FIFO, and the outgoing data is synchronized to the transmit clock.

Building a Library of Components

From the description of the design units, it's clear that several of them will require the use of some common components: flip-flops, enableable registers, counters, and synchronizers. In some of the design units, we will describe such logic with behavioral descriptions. In other units, we will instantiate components, if instantiating components and creating design hierarchy clarifies the design structure.

We will begin by designing some of the basic components that may be common to design units. We'll create D-type flip-flops, enableable registers, and counters, all of multiple widths. In addition, we will create a synchronization component. We'll keep common components in *packages*, collections of type or component declarations that can be used in other designs. We'll keep counters in a package for counters, registers in a package for registers, flip-flops in a package for flip-flops, etc. Once we have completed this design, we'll place these packages in a *library*, a place (usually a directory) to keep precompiled packages so that they may be used in other designs. For now, all design units—from the bottom of the hierarchy to the top—will be compiled and placed in the "work" library.

Generics and Parameterized Components

We begin with the design of D-type flip-flops with asynchronous resets. We will want to create flip-flops of multiple widths. We know that we will need a 1-bit-wide flip-flop, but we don't know a priori what other widths we may need in the design of the higher-level design units. This leads us to design *parameterized components,* components for which the sizes (widths) and feature sets may be defined by values of the instantiation parameters. (Here, we are using the term *component* loosely to identify any entity/architecture pair. A component is identified by a specific entity and architecture pair. Multiple architectures can describe the same entity, and a configuration statement can be used to specify which architecture is associated with an entity. If there is only one architecture for an entity, then the configuration is implicit. Because we are associating only one architecture to each entity, we will continue to use the term *component* loosely.)

The design of a parameterized component is similar to that of any other component, except that a parameter, or *generic*, is used to define the size (and optionally, the feature set). A parameterized entity must include generics in its declaration. Below, in *Listing 6-1*, we define two entity/architecture pairs: a 1-bit D-type flip-flop with asynchronous reset and an n-bit (where n is defined by the generic *size*) wide bank of D-type flip-flops with common clock and asynchronous reset. We will explain how these design units are placed in packages and used by higher-level design units later in the chapter.

```
--      Set of D-Type Flip-Flops
--
--      sizes: (1, size)
--
--  clk     -- posedge clock input
--  reset   -- asynchronous reset
--  d       -- register input
--  q     -- register output
---------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity rdff1 is port (
    clk, reset: in std_logic;
    d:      in std_logic;
    q:      out std_logic);
end rdff1;

architecture archrdff1 of rdff1 is
begin
p1: process (reset, clk) begin
    if reset = '1' then
```

```
        q <= '0';
    elsif (clk'event and clk='1') then
        q <= d;
    end if;
   end process;
end archrdff1;
--------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity rdff is
    generic (size: integer := 2);
    port (  clk, reset: in std_logic;
        d:      in std_logic_vector(size-1 downto 0);
        q:      out std_logic_vector(size-1 downto 0));
end rdff;

architecture archrdff of rdff is
begin
p1: process (reset, clk) begin
    if reset = '1' then
        q <= (others => '0');
    elsif (clk'event and clk='1') then
        q <= d;
    end if;
   end process;
end archrdff;
```

Listing 6-1 Set of D-type flip-flops

These entity/architecture pairs are placed in the same design file, although they do not need to be. The generic *size* is given a default value of 2. If an instantiation of the *rdff* component does not specify a *generic map*, then the component will have the default *size* of 2. *Size* is used to define the width of the std_logic_vectors for *d* and *q*. Because the *rdff* component is parameterized, we didn't need to create a separate component for *rdff1*; however, because we will be using a 1-bit wide D-type flip-flop with greater frequency than any other width, we have made it a separate component.

We will also design a set of registers (D-type flip-flops with enables) with asynchronous resets. Again, we know that we will need a 1-bit register but do not know a priori what other widths will be required. The designs of two components, a 1-bit register and an n-bit register with common enable, are defined shown in *Listing 6-2*.

```
--      Set of registers
--      sizes: (1,size)
--
--      clk     -- posedge clock input
--      reset   -- asynchronous reset
--      load    -- active high input loads rregister
--      d       -- register input
--      q       -- register output
-------------------------------------------------
```

```
library ieee;
use ieee.std_logic_1164.all;
entity rreg1 is port(
        clk, reset, load:       in std_logic;
        d:              in std_logic;
        q:              inout std_logic);
end rreg1;
architecture archrreg1 of rreg1 is
begin
        p1: process (reset, clk) begin
                if reset = '1' then
                        q <= '0';
                elsif (clk'event and clk='1') then
                        if load = '1' then
                                q <= d;
                        else
                                q <= q;
                        end if;
                end if;
        end process;
end archrreg1;

--------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity rreg is
        generic (size: integer := 2);
        port(   clk, reset, load:       in std_logic;
                d:              in std_logic_vector(size-1 downto 0);
                q:              inout std_logic_vector(size-1 downto 0));
end rreg;
architecture archrreg of rreg is
begin
        p1: process (reset,clk) begin
                if reset = '1' then
                        q <= (others => '0');
                elsif (clk'event and clk='1') then
                        if load = '1' then
                                q <= d;
                        else
                                q <= q;
                        end if;
                end if;
        end process;
end archrreg;
```

Listing 6-2 Set of registers with reset

Because the *rreg* component is parameterized, we didn't need to create a separate component for *rreg1*; however, because we will be using a 1-bit register with greater frequency than any other width of register, we have made it a separate component.

In the architecture of both the *rreg1* and *rreg* components, the signal *q* is used on the righthand side of a signal assignment. This requires that the signal be readable. To be readable, the signal must be of mode BUFFER or INOUT. We could have chosen mode BUFFER for *q*; however, if *q* is of mode BUFFER, then in any design unit in which one of these components is instantiated, the *actual* signal associated with *q* must be a signal without another source (e.g., a signal that is local to the

architecture) or a port of mode OUT. The actual can only be of mode BUFFER if it is not used elsewhere in the architecture as a signal source. The restrictions on the ports of mode BUFFER are great enough to have us use mode INOUT where BUFFER could potentially be used, despite the fact that mode INOUT may make the code less readable (because it makes it difficult to identify the source of ports of mode INOUT). Alternatively, we could remove the ELSE clause, allowing us to choose mode OUT for *q*.

Having designed both D-type flip-flops and registers with asynchronous reset, we would now like to create D-type flip-flops and registers with synchronous presets, calling these components *pdff1, pdff, preg1,* and *preg*. At this point, it is clear that rather than create two additional sets of components, we can create just one component, *reg*, as defined in *Listing 6-3*.

```
--       Register Set
--       sizes: (size) a generic
--
--       clk      -- posedge clock input
--       rst      -- asynchronous reset
--       pst      -- asynchronous preset
--       load     -- active high input loads register
--       d        -- register input
--       q        -- register output
-------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity reg is generic ( size: integer := 2);
   port(   clk, load:      in std_logic;
      rst, pst:       in std_logic;
      d:              in std_logic_vector(size-1 downto 0);
      q:              inout std_logic_vector(size-1 downto 0));
end reg;
architecture archreg of reg is
begin
   p1: process (clk) begin
      if rst = '1' then
         q <= (others => '0');
      elsif pst = '1' then
         q <= (others => '1');
      elsif (clk'event and clk='1') then
         if load = '1' then
            q <= d;
         else
            q <= q;
         end if;
      end if;
   end process;
end archreg;
```

Listing 6-3 Set of all-purpose registers

This component can be used for *pdff, rdff, rreg,* and *preg*. (It cannot, however, be used for the 1-bit versions of each component because q <= (others => '1') implies that *q* is an aggregate.—i.e., that it consists of more than one element, that it is an array) When instantiating the component, *size* is used to indicate the width of the register. Normally the actuals (i.e., the actual signals used in a component instantiation) associated with the *locals* (i.e., *rst, pst,* and *load,* the signals associated with the component definition) will act as controls to the reset, preset, and load logic. Alternatively, the actuals associated with the locals can be signals that never change value so that features may be permanently enabled (e.g., the load) or disabled (e.g., the reset). For example, if the signals *vdd* and

vss are permanently assigned the values '1' and '0', respectively, then synthesis of an instantiation of *reg* with the load permanently enabled and the preset permanently disabled (see below) results in a register without load and preset logic. The instantiation for this component is

```
u1: reg generic map(4) port map(myclk, vdd, reset, vss, data, mysig);
```

Likewise, synthesis of the following code will not result in any logic at all:

```
u1: reg generic map(4) port map(myclk, vss, reset, vss, data, mysig);
```

In this instantiation, the load is permanently disabled. If the register never loads data, then it doesn't serve any purpose.

The generic map is used to map an actual value to the generic of the component definition. The generic map may be ommitted if a default value is specified in the generic declaration of the component definition. In this case the generic map was used to specify the size of a register (4-bits wide).

Although the *reg* component is versatile, we will instantiate the other components in our design units. Doing so will improve the readability of the code because it will not require that the reader look for the details of the generic and port maps.

Next, we will create one more set of components, a set of counters. The counters will have enables and synchronous and asynchronous resets. We will not create separate sets of counters without enables, resets, or both. If we require the use of such counters in one of the higher-level design units, then we will simply assign the port maps appropriately. The VHDL description of the counters is in *Listing 6-4*.

```
--   Synchronous Counter of Generic Size
--
--   CounterSize -- size of counter
--
--   clk    -- posedge clock input
--   areset -- asynchronous reset
--   sreset -- active high input resets counter to 0
--   enable -- active high input enables counting
--   count  -- counter output
-------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity ascount is
   generic (CounterSize: integer := 2);
   port(clk, areset, sreset, enable: in std_logic;
      count:    inout std_logic_vector(counterSize-1 downto 0));
end ascount;
use work.std_math.all;
architecture archascount of ascount is
begin
p1: process (areset, clk) begin
   if areset = '1' then
      count <= (others => '0');
   elsif (clk'event and clk='1') then
      if sreset='1' then
         count <= (others => '0');
```

```
      elsif enable = '1' then
         count <= count + 1;
      else
         count <= count;
      end if;
   end if;
  end process;
end archascount;
```

Listing 6-4 Set of counters

Finally, we will create two synchronization components (*Listing 6-5*). These components consist of two D-type flip-flops in series that are used to synchronize an asynchronous input to the system clock. Two flip-flops are used to increase the MTBF for metastable events. One synchronizer component will have an asynchronous reset, the other will have an asynchronous preset.

```
--       Synchronizers
--
--
--       clk     -- posedge clock input
--       reset   -- asynchronous reset (rsynch)
--       preset  -- asynchronous preset (psynch)
--       d       -- signal to synchronize
--       q       -- synchronized
-----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
entity rsynch is port (
   clk, reset:      in std_logic;
   d:               in std_logic;
   q:               out std_logic);
end rsynch;

architecture archrsynch of rsynch is
  signal temp: std_logic;
begin
p1: process (reset, clk) begin
   if reset = '1' then
      q <= '0';
   elsif (clk'event and clk='1') then
      temp <= d;
      q <= temp;
   end if;
   end process;
end archrsynch;

-----------------------
library ieee;
use ieee.std_logic_1164.all;
entity psynch is port (
   clk, preset:     in std_logic;
   d:               in std_logic;
   q:               out std_logic);
```

```
end psynch;

architecture archpsynch of psynch is
  signal temp: std_logic;
begin
p1: process (preset, clk) begin
   if preset = '1' then
      q <= '1';
   elsif (clk'event and clk='1') then
      temp <= d;
      q <= temp;
   end if;
   end process;
end archpsynch;
```

Listing 6-5 Synchronizers

Because we have already defined *pdff* components, the architecture for *psynch* could be replaced by the following code, provided that the *pdff* component is declared and visible to this architecture description:

```
u1: pdff port map(clk, preset, d, temp);
u2: pdff port map(clk, preset, temp, q);
```

This netlist description is succinct and may be easier to understand for those coming from a logic design background. However, it does create an additional level of hierarchy, and someone not familiar with the function of a *pdff* component would need to find its description, which is likely in a different file.

Design Units

For reasons that will become more clear in the last chapter of the book, we will be defining the port map of the top-level entity one signal at a time, rather than busing common signals together. We are doing this to ensure the preservation of top-level signal names and to make the simulation of both the source code and implementation-specific model go smoothly. That is, we are doing this to avoid any problem with tool interfaces, as we will be using several different software tools: a VHDL synthesis tool, a place and route tool, and a VHDL simulator. We're certain that we could bus signals together and eventually get the interfaces to work (perhaps not seamlessly), but experience tells us that unless we test the flow out with a smaller design, for a large design like this one, we should stick with what we know will work to avoid any unnecessary headaches. Because the top-level port is specified signal-by-signal, this will tend to permeate through other levels of hierarchy. We will use buses (vectors) occasionally.

Port Controller

We begin our design of the core logic for the network repeater with the port controller. The entity declaration is shown below, with the name *porte*. "Controller" was removed to avoid a long or cumbersome name, and an "e" was added to *port* to avoid a collision with a keyword. Comments to the right may provide some additional information about the signals.

```
library ieee;
use ieee.std_logic_1164.all;
entity porte is port (
       txclk:                in std_logic; -- TX_CLK
       areset:               in std_logic; -- Asynch Reset
       crs:                  in std_logic; -- Carrier Sense
       enable_bar:           in std_logic; -- Port Enable
       link_bar:             in std_logic; -- PMA_Link_OK
       selected:             in std_logic; -- Arbiter Select
       carrier:              in std_logic; -- Arbiter Carrier
       collision:            in std_logic; -- Arbiter Collision
       jam:                  in std_logic; -- Control Jam
       txdata:               in std_logic; -- Control Transmit Data
       prescale:             in std_logic; -- counter prescale
       rx_en:                out std_logic;    -- RX_EN
       tx_en:                out std_logic;    -- TX_EN
       activity:             out std_logic;    -- Activity
       jabber_bar:           inout std_logic;  -- Jabber
       partition_bar:        inout std_logic); -- Partition
end porte;
```

Looking back at the description of the port controller, we see that *crs, link_bar*, and *enable_bar* must be synchronized to *tx_clk*, so we will use the synchronizer components. We will use the synchronizer with an asynchronous reset for the active high signal, *crs*, and will use the one with an asynchronous preset for the active low signals (those ending in *_bar*). Using the synchronizers requires that we create three signals (*crsdd, link_bardd*, and *enable_bardd*) for the outputs of the synchronizers. The suffix "*-dd*" indicates that these signals are twice-registered. Three components are instantiated:

```
u0: rsynch     port map (txclk, areset, crs, crsdd);
u1: psynch     port map (txclk, areset, link_bar, link_bardd);
u2: psynch     port map (txclk, areset, enable_bar, enable_bardd);
```

Instantiation of these components requires that they have been declared and are visible to the architecture. The components will be declared in a package named *portetop_pkg*, the contents of which will be described later, and will be made visible to the architecture by including the USE clause:

```
use work.portetop_pkg.all;
```

Next, we create an equation for *activity,* which is asserted if a carrier is present, the link is established, and the port is not partitioned or jabbering. The presence of a carrier will be gated by *enable_bar*. Carrier presence will be used for other port controller functions, so we will create a signal to hold its value.

```
carpres <= crsdd AND NOT enable_bardd;
activity <= carpres AND NOT link_bardd AND jabber_bar AND partition_bar;
```

Here, we chose to define *carpres* and *activity* with Boolean equations rather than a different dataflow construct because, in this case, other dataflow constructs do not lend to the readability of the code.

Likewise, *rx_en* and *tx_eni* will be defined with Boolean equations. *Tx_eni* is the value of *tx_en* before it is registered. It must be delayed one clock cycle to synchronize it with the output data. *Tx_eni* must be declared as a signal. The logic is described below.

```
rx_en   <= NOT enable_bardd and NOT link_bardd AND selected AND collision;
tx_eni  <= NOT enable_bardd and NOT link_bardd AND jabber_bar AND transmit;

u3: rdff1       port map (txclk, areset, tx_eni, tx_en);
```

Transmit is the logical AND of two quantities. The first signal is *txdata*, which indicates that the core controller state machine is ready to send data or *jam* characters. The second is the result of the logical OR of *collision* and a signal (*copyd*), which indicates that the arbiter has detected a *carrier* but that this port is not the *selected* port. All signals must be synchronized. *Txdata* is one clock cycle behind the other signals because it is triggered by them (see the core controller state machine later in this chapter). Therefore, the other signals must be delayed:

```
u4: rdff1      port map (txclk, areset, copyin, copyd);
u5: rdff1      port map (txclk, areset, collision, collisiond);

copyin <= carrier and NOT selected;
transmit <= txdata AND (copyd OR collisiond);
```

There are two remaining functions of the port controller that we must design: *jabber_bar* and *partition_bar*. *Jabber_bar* should be asserted if the carrier is present for anywhere between 40,000 and 75,000 bit times. 40,000 bit times over 4 pairs requires 10,000 clock cycles (75,000 bit times requires 18,750 clock cycles). We will use a counter to count while the carrier is present. If it counts for approximately 10,000 clock cycles, then *jabber_bar* will be asserted. The counter will be reset when the carrier is not present.

Counting to 10,000 requires a 14-bit counter. However, we don't want the counter to be replicated eight times, once for each port, because it would consume device resources unnecessarily. Therefore, we use one common counter, placing it in the core controller, to count to 1K. This counter runs continuously. In each of the port controllers, we place a 4-bit counter that is enabled each time the 10-bit counter rolls over. When the 4-bit counter reaches 12, then we know that the carrier has been present for 12K clock cycles plus or minus 1K clock cycles (10K < 12K +/- 1K < 18.75K). For the design of the port controller, this requires the instantiation of a 4-bit counter and the definition of the counter enable and clear signals as well as *jabber_bar*.

```
u6: ascount     generic map (4)
                port map (txclk, areset, jabberclr, jabberinc, jabcnt);

jabber_bar <=  NOT  (jabcnt(3) AND jabcnt(2));
jabberclr <= NOT carpres;
jabberinc <= carpres AND prescale AND jabber_bar;
```

The counter is not allowed to increment when *jabber_bar* is asserted, active low. *Jabber_bar* is asserted when *jabcnt* reaches 12 which is 1100 in binary, or *jabcnt(3)* AND *jabcnt(2)*. If the counter is still enabled and the port continues to jabber, then the counter rolls over, in which case *jabber_bar* is no longer asserted.

The conditions for which a port is partitioned or reconnected have been described previously and are defined by a state diagram in the IEEE 802.3 standard. This state diagram has been modified for our purposes and is illustrated in *Figure 6-11*. The states on the lefthand side of the diagram are used to determine if the port should be partitioned. The port is partitioned in the states on the righthand side, indicated by *partition_bar* being asserted (active low). The states on the righthand side are used to determine if the port should be reconnected. The clearing and enabling of two counters are controlled

by this state machine: one that counts collisions (the collision counter) and another that counts the number of consecutive clock cycles without a collision (the no-collision counter). Several signals are inputs to this state machine: *copyd* indicates that another port is active, *quietd* is the opposite of *copyd*—it indicates that another port is not active, *carpres* indicates that the carrier sense of this port is active, *collisiond* indicates a collision, *nocoldone* is asserted when the no-collision counter reaches 128 (128 clock cycles is between 450 and 560 bit times, the approximate size of the minimum data frame—each clock cycles represents four bit times, one bit time for each of the transmit and receive paris), and *cclimit* is asserted when 64 consecutive collisions have occurred.

The counters are cleared in the *CLEAR_STATE*. The state machine enters this state after a system reset or if a collision does not occur for approximately one minimum data-frame size while *carpres* is asserted. The *IDLE_STATE* resets the no-collision counter. In the collision watch state, *CWATCH_STATE*, if a collision occurs, then the collision counter is incremented and the state machine transitions to the collision count state, *CCOUNT_STATE*. As long as a collision does not occur, the state machine remains in the *CWATCH_STATE* until *nocoldone* is asserted, after which the state machine transitions to the *CLEAR_STATE* and both counters are reset. The *CCOUNT_STATE* is entered upon a collision. If the collision counter reaches its limit (64), then the port is to be partitioned and the state machine transitions to the *PWAIT_STATE*. If the collision counter does not reach its limit, then the machine waits for the offending nodes to back off of the network, transitioning to the *IDLE_STATE* when the network is quiet again. From this point, transitions through the states will continue; either additional collisions will be counted or the counters will be cleared.

The *PWAIT_STATE* indicates that the collision counter limit has been reached and that the port is partitioned. This state is used to wait until the collision ceases. Once the network is quiet again, the state machine transitions to the partition hold state, *PHOLD_STATE*. This state and the next, the partition collision watch state (*PCWATCH_STATE*), are used to count consecutive clock cycles during which there is not a collision and another port is active. If the port does not cause a collision for 450 to 560 bit times while another port is active, then the state machine transitions to the *WAIT_STATE*. The *WAIT_STATE* is used to wait until the active port is quiet before transitioning to the *IDLE_STATE*.

As with other state machines, we will code this one using an enumerated type and a CASE-WHEN construct. The complete architecture code for the port controller is found in *Listing 6-6*. There is one difference in the way that the partition and reconnect state machine was described as compared to other state machines in this text: The outputs are explicitly declared for each state. Although we could describe the outputs using WHEN-ELSE or WITH-SELECT-WHEN constructs in dataflow assignments, the way that they are described in this listing lends to the readability of the code. Using this method of describing state machine outputs is best done if the state clocking is described in a second process; otherwise, next-state (instead of present-state) outputs must be listed.

```
use work.portetop_pkg.all;
architecture archporte of porte is

  type states is (CLEAR_STATE, IDLE_STATE, CWATCH_STATE, CCOUNT_STATE,
      PWAIT_STATE, PHOLD_STATE, PCWATCH_STATE, WAIT_STATE);
  attribute state_encoding of states:type is one_hot_one;

  signal state, newstate: states;
```

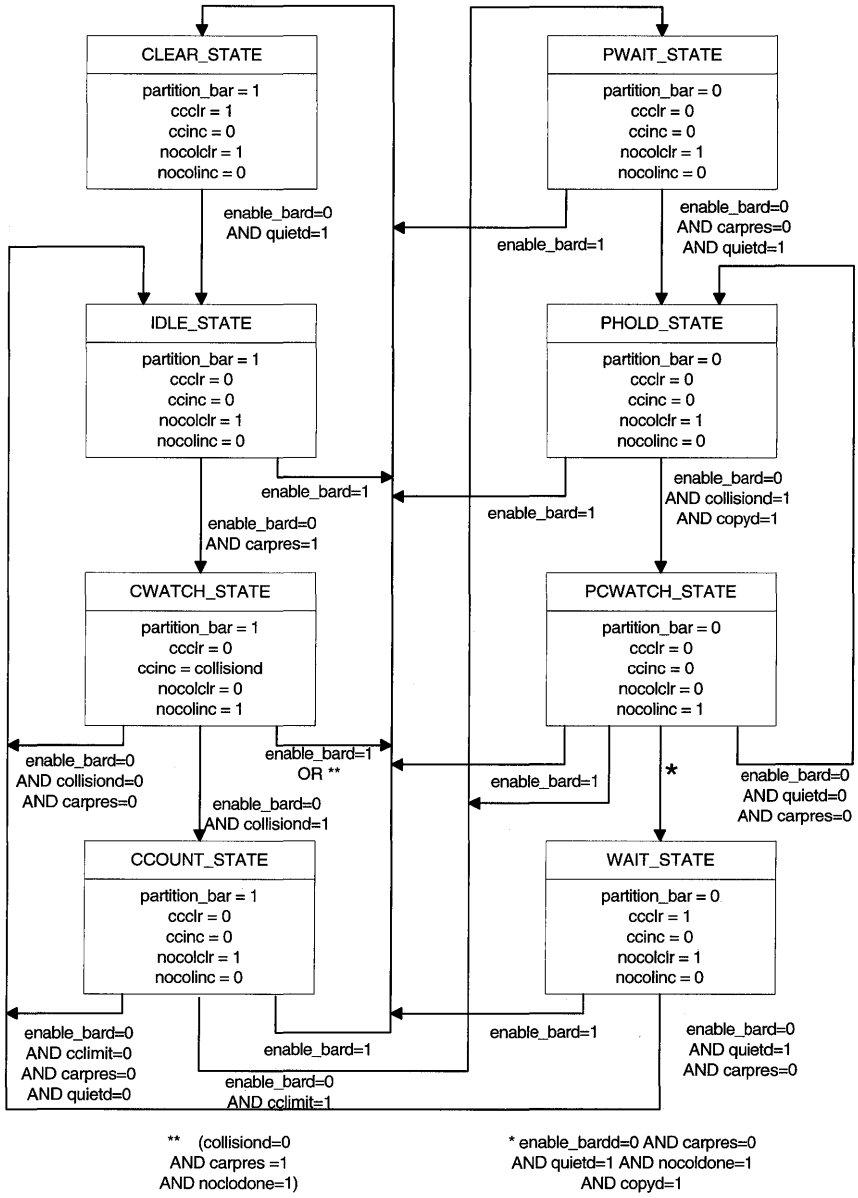


Figure 6-11 State diagram for paritioning and reconnecting

```
  signal  crsdd, link_bardd, enable_bardd: std_logic;
  signal  tx_eni, carpres, transmit, copyd, copyin, collisiond: std_logic;
  signal  jabcnt: std_logic_vector(3 downto 0);
  signal  jabberclr, jabberinc: std_logic;
  signal  quietd: std_logic;
  signal  cccnt: std_logic_vector(6 downto 0);
  signal  cclimit, nocoldone: std_logic;
  signal  nocolcnt: std_logic_vector(7 downto 0);
  signal  ccclr, ccinc, nocolclr, nocolinc: std_logic;

begin

--      Components

u0: rsynch      port map (txclk, areset, crs, crsdd);
u1: psynch      port map (txclk, areset, link_bar, link_bardd);
u2: psynch      port map (txclk, areset, enable_bar, enable_bardd);
u3: rdff1       port map (txclk, areset, tx_eni, tx_en);
u4: rdff1       port map (txclk, areset, copyin, copyd);
u5: rdff1       port map (txclk, areset, collision, collisiond);
u6: ascount     generic map (4) port map (txclk, areset, jabberclr,
jabberinc, jabcnt);
u7: ascount     generic map (7) port map (txclk, areset, ccclr, ccinc,
cccnt);
u8: ascount     generic map (8) port map (txclk, areset, nocolclr, nocolinc,
nocolcnt);


carpres <= crsdd AND NOT enable_bardd;
activity <= carpres AND NOT link_bardd AND jabber_bar AND partition_bar;
rx_en    <= NOT enable_bardd and NOT link_bardd AND selected AND collision;
tx_eni   <= NOT enable_bardd and NOT link_bardd AND jabber_bar and transmit;
copyin <= carrier and NOT selected;
transmit <= txdata AND (copyd OR collisiond);
jabber_bar <=  NOT  (jabcnt(3) AND jabcnt(2));
jabberclr <= NOT carpres;
jabberinc <= carpres AND prescale AND jabber_bar;
quietd <= NOT copyd;
cclimit <= cccnt(6);
nocoldone <= nocolcnt(7);


--      Partition State Machine

p1: process (state, carpres, collisiond, copyd, quietd,
      nocoldone, cclimit, enable_bardd)  begin
   case (state) is

   when CLEAR_STATE=>

         partition_bar    <= '1'  ;
         ccclr            <= '1'  ;
         ccinc            <= '0'  ;
         nocolclr         <= '1'  ;
```

```
      nocolinc        <= '0'  ;

      if (enable_bardd = '1') then
         newstate        <= CLEAR_STATE;
      elsif (quietd = '1')     then
         newstate        <= IDLE_STATE;
      else
         newstate        <= CLEAR_STATE;

      end if;

when IDLE_STATE=>

      partition_bar     <= '1'  ;
      ccclr           <= '0'  ;
      ccinc           <= '0'  ;
      nocolclr        <= '1'  ;
      nocolinc        <= '0'  ;

      if (enable_bardd = '1') then
         newstate        <= CLEAR_STATE;
      elsif (carpres = '1')    then
         newstate        <= CWATCH_STATE;
      else
         newstate        <= IDLE_STATE;

      end if;


when CWATCH_STATE=>

      partition_bar     <= '1'  ;
      ccclr           <= '0'  ;
      ccinc           <= collisiond;
      nocolclr        <= '0'  ;
      nocolinc        <= '1'  ;

      if (enable_bardd = '1') then
         newstate        <= CLEAR_STATE;
      elsif (collisiond = '1')         then
         newstate        <= CCOUNT_STATE;
      elsif (carpres = '0') then
         newstate        <= IDLE_STATE;
      elsif (nocoldone = '1') then
         newstate        <= CLEAR_STATE;
      else
         newstate        <= CWATCH_STATE;

      end if;


when CCOUNT_STATE=>

      partition_bar     <= '1'  ;
```

```
      ccclr           <= '0'  ;
      ccinc           <= '0'  ;
      nocolclr        <= '1'  ;
      nocolinc        <= '0'  ;


      if (enable_bardd = '1') then
         newstate        <= CLEAR_STATE;
      elsif (cclimit = '1')    then
         newstate        <= PWAIT_STATE;
      elsif (carpres = '0' AND quietd = '1') then
         newstate        <= IDLE_STATE;
      else
         newstate        <= CCOUNT_STATE;

      end if;


when PWAIT_STATE=>

      partition_bar     <= '0'  ;
      ccclr           <= '0'  ;
      ccinc           <= '0'  ;
      nocolclr        <= '1'  ;
      nocolinc        <= '0'  ;


      if (enable_bardd = '1') then
         newstate        <= CLEAR_STATE;
      elsif (carpres = '0' AND quietd = '1') then
         newstate        <= PHOLD_STATE;
      else
         newstate        <= PWAIT_STATE;

      end if;


when PHOLD_STATE=>

      partition_bar     <= '0'  ;
      ccclr           <= '0'  ;
      ccinc           <= '0'  ;
      nocolclr        <= '1'  ;
      nocolinc        <= '0'  ;


      if (enable_bardd = '1') then
         newstate        <= CLEAR_STATE;
      elsif (collisiond = '1' OR copyd = '1') then
         newstate        <= PCWATCH_STATE;
      else
         newstate        <= PHOLD_STATE;

      end if;
```

```
   when PCWATCH_STATE=>

         partition_bar     <= '0'  ;
         ccclr           <= '0'  ;
         ccinc           <= '0'  ;
         nocolclr        <= '0'  ;
         nocolinc        <= '1'  ;


         if (enable_bardd = '1') then
            newstate        <= CLEAR_STATE;
         elsif (carpres = '1') then
            newstate        <= PWAIT_STATE;
         elsif (quietd = '0') then
            newstate        <= PHOLD_STATE;
         elsif (nocoldone = '1' AND copyd = '1') then
            newstate        <= WAIT_STATE;
         else
            newstate        <= PCWATCH_STATE;

         end if;


   when WAIT_STATE=>

         partition_bar     <= '0'  ;
         ccclr           <= '1'  ;
         ccinc           <= '0'  ;
         nocolclr        <= '1'  ;
         nocolinc        <= '0'  ;


         if (enable_bardd = '1') then
            newstate        <= CLEAR_STATE;
         elsif (carpres = '0' AND quietd = '1') then
            newstate        <= IDLE_STATE;
         else
            newstate        <= WAIT_STATE;

         end if;


   end case;


end process;


--      State Flip-Flop for Synthesis
p1clk: process (txclk,areset)
   begin
      if areset = '1' then
```

```
            state <= clear_state;
         elsif (txclk'event and txclk = '1') then
            state <= newstate;
         end if;
      end process;
end archporte;
```

Listing 6-6 Port controller architecture

The state machine was quickly translated from the state diagram to VHDL code. Careful evaluation of the code shows that it models the description above.

Some designers would prefer to describe a design like this one by entering a bubble diagram, leaving the translation to VHDL to a software tool. There are tools that will perform this task. We find, however, that equal time (if not more) is spent in entering a bubble diagram than in entering the VHDL code directly.

We have described each part of the design except where the *rsynch*, *psynch*, *rdff1*, and *ascount* components come from. These components are declared in the *portetop_pkg* and made visible by the USE clause. The *portetop_pkg* code is shown in *Listing 6-7*.

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

package portetop_pkg is

component rsynch port (
   clk, reset:      in std_logic;
   d:               in std_logic;
   q:               out std_logic);
end component;

component psynch port (
   clk, preset:     in std_logic;
   d:               in std_logic;
   q:               out std_logic);
end component;

component rdff1 port (
   clk, reset:      in std_logic;
   d:               in std_logic;
   q:               out std_logic);
end component;

component ascount
   generic (CounterSize: integer := 2);
   port(    clk, areset, sreset, enable: in std_logic;
      count:     buffer std_logic_vector(CounterSize-1 downto 0));
end component;
```

```
end portetop_pkg;
```

Listing 6-7 Portetop_pkg includes components used in the design of the port controller

Without declaring the components and making them visible to the *porte* design, the entity/architecture pairs associated with these components can only be used as top-level designs. By declaring components for them, these components can be instantiated in a hierarchical design.

To synthesize the design of the *porte, Listing 6-1*, *Listing 6-4*, and *Listing 6-5* must be compiled first, then *Listing 6-7*, and finally *Listing 6-6*. It is assumed that the *std_math* package has already been compiled to the "work" library.

Arbiter

The entity declaration for the arbiter is shown below, with the name *arbiter8*. Comments to the right may provide some additional information about the signals. The *arbiter* ensures that only one port is selected as the active (receiving) port and identifies collisions as well as an absence of activity. The design does not present any surprises. The output signals are defined with Boolean equations. The *activityin* signals could have been described with an IF-THEN construct, and the *collision* signal could have been defined with an algorithm using loops, but the Boolean equations are clear and easy to follow.

A bank of registers is used to create a pipeline stage. The *carrier*, *collision*, and *nosel* will be used in multiple modules and will propagate through several levels of logic, generating the need for a pipeline stage to maintain a high frequency of operation. The other signals are pipelined to maintain synchronization. The complete code for *arbiter* is in *Listing 6-8*.

```
library ieee;
use ieee.std_logic_1164.all;
entity arbiter8 is port(
       txclk:                in std_logic;   -- TX_CLK
       areset:               in std_logic;   -- Asynch Reset
       activity1:            in std_logic;   -- Port Activity 1
       activity2:            in std_logic;   -- Port Activity 2
       activity3:            in std_logic;   -- Port Activity 3
       activity4:            in std_logic;   -- Port Activity 4
       activity5:            in std_logic;   -- Port Activity 5
       activity6:            in std_logic;   -- Port Activity 6
       activity7:            in std_logic;   -- Port Activity 7
       activity8:            in std_logic;   -- Port Activity 8
       sel1:                 out std_logic;  -- Port Select 1
       sel2:                 out std_logic;  -- Port Select 2
       sel3:                 out std_logic;  -- Port Select 3
       sel4:                 out std_logic;  -- Port Select 4
       sel5:                 out std_logic;  -- Port Select 5
       sel6:                 out std_logic;  -- Port Select 6
       sel7:                 out std_logic;  -- Port Select 7
       sel8:                 out std_logic;  -- Port Select 8
       nosel:                out std_logic;  -- No Port Selected
       carrier:              out std_logic;  -- Carrier Detected
       collision:            out std_logic); -- Collision Detected
end arbiter8;
```

```
use work.arbiter8top_pkg.all;
architecture archarbiter8 of arbiter8 is
--      Signals
signal  colin, carin: std_logic;
signal  activityin1, activityin2, activityin3, activityin4: std_logic;
signal  activityin5, activityin6, activityin7, activityin8: std_logic;
signal  noactivity: std_logic;

begin
--      Components

u1: rdff1 port map  (txclk, areset, activityin1, sel1);
u2: rdff1 port map  (txclk, areset, activityin2, sel2);
u3: rdff1 port map  (txclk, areset, activityin3, sel3);
u4: rdff1 port map  (txclk, areset, activityin4, sel4);
u5: rdff1 port map  (txclk, areset, activityin5, sel5);
u6: rdff1 port map  (txclk, areset, activityin6, sel6);
u7: rdff1 port map  (txclk, areset, activityin7, sel7);
u8: rdff1 port map  (txclk, areset, activityin8, sel8);

u9: pdff1 port map  (txclk, areset, noactivity, nosel);

u10: rdff1 port map (txclk, areset, colin, collision);
u11: rdff1 port map (txclk, areset, carin, carrier);

--      Arbitration Select Logic
   activityin1  <= activity1;

   activityin2  <= activity2
            AND NOT activity1;

   activityin3  <= activity3
            AND NOT(activity1 OR activity2);

   activityin4  <= activity4
            AND NOT(activity1 OR activity2 OR activity3);

   activityin5  <= activity5
            AND NOT(activity1 OR activity2 OR activity3 OR activity4);

   activityin6  <= activity6
            AND NOT(activity1 OR activity2 OR activity3 OR activity4 OR
           activity5);

   activityin7  <= activity7
            AND NOT(activity1 OR activity2 OR activity3 OR activity4 OR
           activity5 OR activity6);

   activityin8  <= activity8
            AND NOT(activity1 OR activity2 OR activity3 OR activity4 OR
           activity5 OR activity6 OR activity7);
```

```
   noactivity   <= NOT(activity1 OR activity2 OR activity3 OR activity4 OR
         activity5 OR activity6 OR activity7 OR activity8);

   colin        <= (activity1 AND (activity2 OR activity3 OR activity4 OR
            activity5 OR activity6 OR activity7 OR activity8)) OR

            (activity2 AND (activity1 OR activity3 OR activity4 OR
            activity5 OR activity6 OR activity7 OR activity8)) OR

            (activity3 AND (activity1 OR activity2 OR activity4 OR
            activity5 OR activity6 OR activity7 OR activity8)) OR

            (activity4 AND (activity1 OR activity2 OR activity3 OR
            activity5 OR activity6 OR activity7 OR activity8)) OR

            (activity5 AND (activity1 OR activity2 OR activity3 OR
            activity4 OR activity6 OR activity7 OR activity8)) OR

            (activity6 AND (activity1 OR activity2 OR activity3 OR
            activity4 OR activity5 OR activity7 OR activity8)) OR

            (activity7 AND (activity1 OR activity2 OR activity3 OR
            activity4 OR activity5 OR activity6 OR activity8)) OR

            (activity8 AND (activity1 OR activity2 OR activity3 OR
            activity4 OR activity5 OR activity6 OR activity7)) ;


   carin        <= activity1 OR activity2 OR activity3 OR activity4 OR
            activity5 OR activity6 OR activity7 OR activity8 ;
end archarbiter8;
```

Listing 6-8 Arbiter selects which port is active.

The *arbiter8top_pkg* package (not shown) serves the same purpose as the *portetop_pkg*: It declares components instantiated in the design.

Clock Multiplexer

The entity declaration for the clock multiplexer is shown below, with the name *clockmux8*. The design of the glitch-free clock-multiplexer circuit is left as an exercise for the reader. *Figure 6-12* illustrates a potential design solution.

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

entity clockmux8 is port (
   areset          : in std_logic;        -- Asynch Reset
   sreset          : in std_logic;        -- Synch  Reset
   clk1            : in std_logic;        -- Clock 1
   clk2            : in std_logic;        -- Clock 2
   clk3            : in std_logic;        -- Clock 3
   clk4            : in std_logic;        -- Clock 4
```

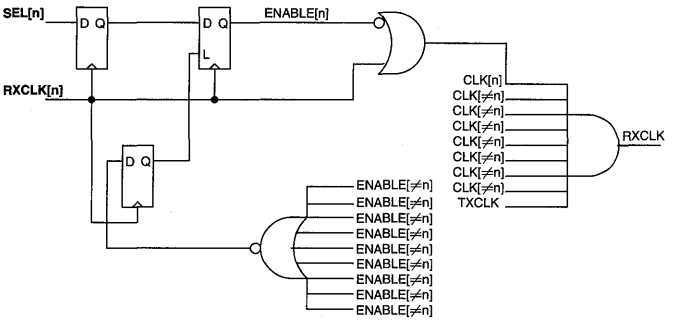


Figure 6-12 Glitch-free clock multiplexer circuit

```
    clk5            : in std_logic;         -- Clock 5
    clk6            : in std_logic;         -- Clock 6
    clk7            : in std_logic;         -- Clock 7
    clk8            : in std_logic;         -- Clock 8
    clk9            : in std_logic;         -- Clock 9 (txclk)
    sel1            : in std_logic;         -- Clock Select 1
    sel2            : in std_logic;         -- Clock Select 2
    sel3            : in std_logic;         -- Clock Select 3
    sel4            : in std_logic;         -- Clock Select 4
    sel5            : in std_logic;         -- Clock Select 5
    sel6            : in std_logic;         -- Clock Select 6
    sel7            : in std_logic;         -- Clock Select 7
    sel8            : in std_logic;         -- Clock Select 8
    sel9            : in std_logic;         -- Clock Select 9
    rxclk           : out std_logic);       -- RX Clock
end clockmux8;
```

FIFO

Our approach to the design of this FIFO differs from the design of the FIFO in chapter 4, "Creating Combinational and Synchronous Logic." In this design (*Listing 6-9*), rather than use an array of std_logic_vectors and a process to access the different vectors, we instantiate eight 6-bit-wide registers. Accessing of the registers with the read and write pointers is handled in the same way as with the FIFO in chapter 4, "Creating Combinational and Synchronous Logic."

The depth of the FIFO was chosen as eight to account for the latency between write cycles and read cycles. This latency is caused by the port selection, state transition, and datapath. Observation of the read and write pointers during simulation (simulation is discussed in chapter 9, "Creating Test Fixtures") indicates that the depth is sufficient. A worst-case analysis for the depth of the FIFO is left as an exercise for the reader.

```
library ieee;
use ieee.std_logic_1164.all;
entity fifo is port(
       rxclk:       in std_logic;    -- from Clock Mux Circuit
       txclk:       in std_logic;    -- Reference TX_CLK
```

```
        areset:     in std_logic;    -- Asynch Reset
        sreset:     in std_logic;    -- Synch Reset
        wptrclr:    in std_logic;    -- FIFO Write Pointer Clear
        wptrinc:    in std_logic;    -- FIFO Write Pointer Incr
        rptrclr:    in std_logic;    -- FIFO Read Pointer Clear
        rptrinc:    in std_logic;    -- FIFO Read Pointer Incr
        rxd5:       in std_logic;    -- FIFO Data Input
        rxd4:       in std_logic;    -- FIFO Data Input
        rxd3:       in std_logic;    -- FIFO Data Input
        rxd2:       in std_logic;    -- FIFO Data Input
        rxd1:       in std_logic;    -- FIFO Data Input
        rxd0:       in std_logic;    -- FIFO Data Input
        dmuxout:    out std_logic_vector(5 downto 0);  -- FIFO Mux output
        wptr2:      out std_logic;    -- Write Pointer
        wptr1:      out std_logic;    -- Write Pointer
        wptr0:      out std_logic;    -- Write Pointer
        rptr2:      out std_logic;    -- Read Pointer
        rptr1:      out std_logic;    -- Read Pointer
        rptr0:      out std_logic);   -- Read Pointer
end fifo;

use work.fifotop_pkg.all;
architecture archfifo of fifo is
-- signals
  signal rptr, wptr: std_logic_vector(2 downto 0);
  signal qout0, qout1, qout2, qout3, qout4, qout5,
       qout6, qout7, rxd: std_logic_vector(5 downto 0);
  signal en: std_logic_vector(7 downto 0);

begin
--       Components
      --FIFO array
u1: rreg generic map (6) port map (rxclk, areset, en(0), rxd, qout0);
u2: rreg generic map (6) port map (rxclk, areset, en(1), rxd, qout1);
u3: rreg generic map (6) port map (rxclk, areset, en(2), rxd, qout2);
u4: rreg generic map (6) port map (rxclk, areset, en(3), rxd, qout3);
u5: rreg generic map (6) port map (rxclk, areset, en(4), rxd, qout4);
u6: rreg generic map (6) port map (rxclk, areset, en(5), rxd, qout5);
u7: rreg generic map (6) port map (rxclk, areset, en(6), rxd, qout6);
u8: rreg generic map (6) port map (rxclk, areset, en(7), rxd, qout7);

-- Write Pointer
u10: ascount   generic map (3)
               port map (rxclk, areset, wptrclr, wptrinc, wptr);
-- Read Pointer
u11: ascount   generic map (3)
               port map (txclk,  areset, rptrclr, rptrinc, rptr);

  rxd <= (rxd5, rxd4, rxd3, rxd2, rxd1, rxd0);
  wptr2 <= wptr(2);
  wptr1 <= wptr(1);
  wptr0 <= wptr(0);
  rptr2 <= rptr(2);
  rptr1 <= rptr(1);
```

```
  rptr0 <= rptr(0);

--      8:1 Data Mux
with rptr select
   dmuxout <=
      qout0 when "000",
      qout1 when "001",
      qout2 when "010",
      qout3 when "011",
      qout4 when "100",
      qout5 when "101",
      qout6 when "110",
      qout7 when others;

--      FIFO Register Selector Decoder (wptr)
with wptr select
   en <=
      "00000001" when "000",
      "00000010" when "001",
      "00000100" when "010",
      "00001000" when "011",
      "00010000" when "100",
      "00100000" when "101",
      "01000000" when "110",
      "10000000" when others;

end archfifo;
```

Listing 6-9 FIFO design

The receive data is bused together internal to the architecture by concatenating the individual *rxd* bits:

```
  rxd <= (rxd5, rxd4, rxd3, rxd2, rxd1, rxd0);
```

Because the read and write pointers are propagated to the top level, they are sent as individual std_logics rather than as a std_logic_vector:

```
  wptr2 <= wptr(2);
  wptr1 <= wptr(1);
  wptr0 <= wptr(0);
  rptr2 <= rptr(2);
  rptr1 <= rptr(1);
  rptr0 <= rptr(0);
```

Core Controller

Before discussing the design of the core controller, we will digress to explain how a host on the network transmits data.

The MAC-to-transceiver interface is shown at a high level in *Figure 6-13*. The transceiver takes nibble-wide data from the MAC each clock cycle. Over two clock cycles, the transceiver stores the first nibble in one register and the second nibble in another. Eight bits of data are then encoded using the 8B6T ternary code of the IEEE 802.3 standard. Three shift registers used to serialize the data are

loaded with the 8B6T code groups (*Figure 6-14*). One shift register is loaded while another byte is converted to an 8B6T code group. Therefore, the second register is loaded two clock cycles after the first. The third register is loaded two clock cycles after the second. Two clock cycles later, data in the first register has been transmitted, so it can be loaded with a new 8B6T code group. This interface results in the data frame structure of *Figure 6-15*. The third transmit pair is the first to begin transmitting data, the first transmit pair is the second to transmit data, and the second pair is the last to transmit data. This transmission scheme is used to avoid latency and increase total system performance.

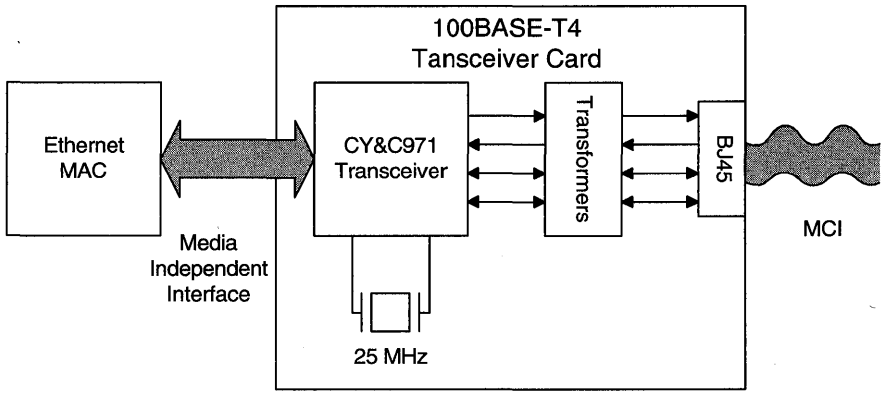


Figure 6-13 MAC-to-transceiver interface

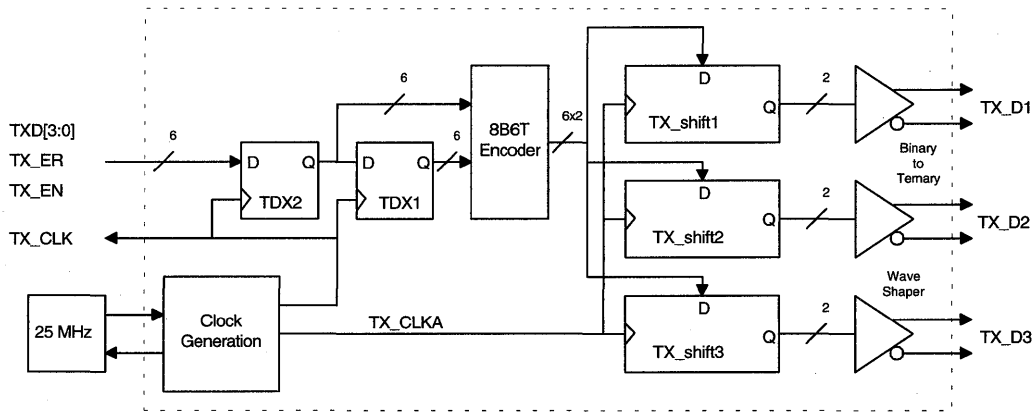


Figure 6-14 MAC-to-transceiver interface data frame structure

The repeater receives data with this structure and is responsible for retransmitting the data with the same structure. The preamble and SFD symbols (sosa and sosb) must be regenerated because the initial portion of the preamble is lost during the reception while clock recovery is performed. When the SFD symbol is received by the repeater, data is stored in the FIFO. Initial data is only valid, however, for the third pair at first, then the first and third pairs, and finally all pairs (*Figure 6-15*). All data is stored in the FIFO for all pairs, even if the data is not valid for all pairs. On the retransmission of data, only those pairs of data that are valid will be transmitted—the other transmit pairs will continue to transmit preamble, conforming to the data frame structure. Now, we will return to the design of the core controller.

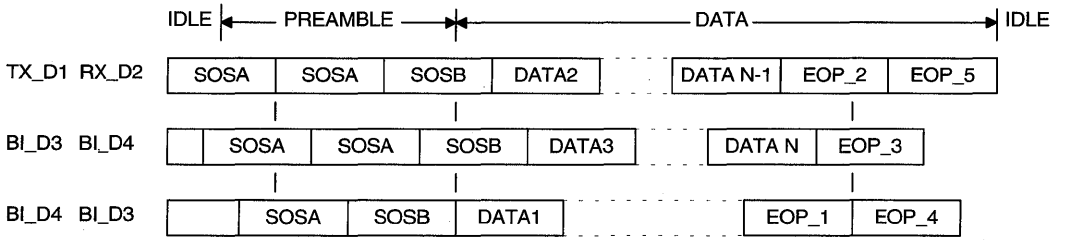


Figure 6-15 Data frame structure

The function of the core controller is illustrated in *Figure 6-16*. The state machines interact primarily with the symbol generator and output multiplexer. There is a state machine for each of the transmit pairs. Although the symbol generation is shown as three separate design units in *Figure 6-16*, it is designed as one unit in order to control the timing of transmit pairs (the multiplexers inside of this design unit are separate, however). A 3-bit counter is used to time the transmission of the SFD (start of frame delimiter) symbol and subsequent data on transmit pairs. One pair is transmitted when the counter reaches the decimal value of 1, the next pair begins at 3, and the next at 5. The counter continuously counts, rolling over after counting through 6, to indicate symbol boundaries and ensure that transmission of data is separated by two clock cycles.

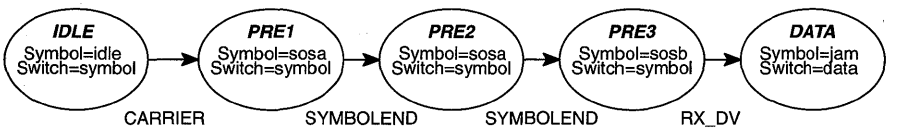


Figure 6-18 Preamble Generation for Pairs 1 and 3

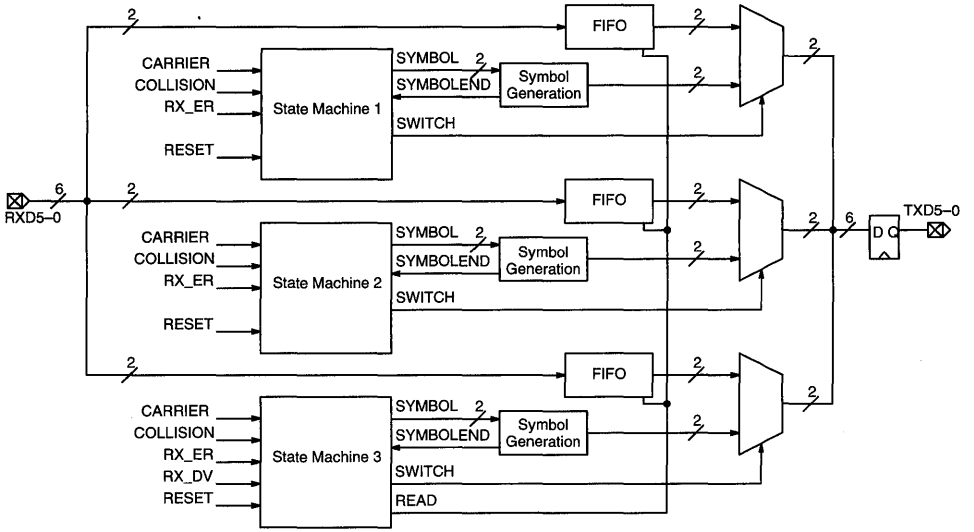


Figure 6-16 Core Controllers

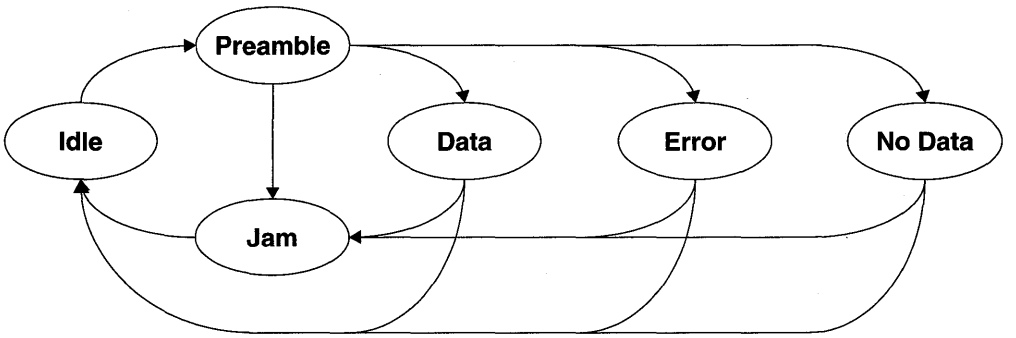


Figure 6-17 State Transition Diagram for Core State Machines

Each of the state machines follows the state transition diagram of *Figure 6-17*. The preamble state actually consists of three preamble substates for the first and third transmit pairs (*Figure 6-18*); the second transmit pair has four preamble states (three to transmit sosa and one to transmit sosb). Transitions from one preamble state to the next are based on the *symbolend* for each of the pairs. The *symbolend* is a symbol boundary that is generated from the 3-bit counter discussed in the previous

paragraph. *Symbolend1* indicates where the symbol boundaries are for the first transmit pair; *symbolend2* is for the second transmit pair, and *symbolend3* is for the third. The symbol boundaries are separated by two clock cycles, with *symbolend2* starting at 1, *symbolend3* starting at 3, and *symbolend1* starting at 5 to conform to the data frame structure of *Figure 6-15*. The symbolend counter will be placed in the symbol generator design unit because its value is required in the generation of symbols.

The state machines use *rx_dv* and *rx_er* as inputs. These signals transition from the high-impedance state and are gated with *carrierd* to filter out glitches. Because *rx_dv* and *rx_er* are asynchronous to the transmit clock, they must be synchronized to the rest of the system.

The FIFO read and write pointers and clears are all controlled by the core controller. The FIFO read and write pointers are continuously incremented. However, the write pointer is cleared until valid data is identified, and the read pointer is cleared until the state machine enters a data state.

Finally, the prescale counter shared by the port controllers is placed in this design unit. A prescale output is generated when the counter reaches its limit.

Listing 6-10 is a partial code listing of the core controller. It includes all logic except for two of the state machines. The designs of those state machines are similar to that of the other and are left as exercises for the reader.

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

entity control is port(
       txclk:        in std_logic;     -- Reference TX_CLK
       areset:       in std_logic;     -- Async Reset
       carrier:      in std_logic;     -- indicates carrier asserted
       collision:    in std_logic;     -- indicates collision condition
       rx_error:     in std_logic;     -- indicates RX PMA error
       rx_dv:        in std_logic;     -- indicates SFD found in data
       symbolend1:   in std_logic;     -- indicates end of symbol line 1
       symbolend2:   in std_logic;     -- indicates end of symbol line 2
       symbolend3:   in std_logic;     -- indicates end of symbol line 3
       symbolclr:    out std_logic;    -- resets symbol counter
       symbolinc:    out std_logic;    -- increments symbol counter
       symbol1:      out std_logic_vector(1 downto 0);--    selects
       symbol2:      out std_logic_vector(1 downto 0);--    special
       symbol3:      out std_logic_vector(1 downto 0);--    symbols
       switch1:      out std_logic;    -- selects special/data symbols
       switch2:      out std_logic;    -- selects special/data symbols
       switch3:      out std_logic;    -- selects special/data symbols
       wptrclr:      out std_logic;    -- FIFO write pointer clear
       wptrinc:      out std_logic;    -- FIFO write pointer increment
       rptrclr:      out std_logic;    -- FIFO read pointer clear
       rptrinc:      out std_logic;    -- FIFO read pointer increment
       txdata:      out std_logic;     -- txdata is ready
       idle:         out std_logic;    -- indicates idle generation
       preamble:     out std_logic;    -- indicates preamble generation
       data:         out std_logic;    -- indicates data generation
       col:          out std_logic;    -- indicates jam generation
       prescale:     out std_logic);   -- prescale output to port
```

```
end control;

use work.controltop_pkg.all;
architecture archcontrol of control is
type states1 is (IDLE_STATE1, PRE1_STATE1, PRE2_STATE1, PRE3_STATE1,
DATA_STATE1,
      JAM_STATE1, NOSFD_STATE1, ERROR_STATE1);
--attribute state_encoding of states1:type is one_hot_one;

type states2 is (IDLE_STATE2, PRE1_STATE2, PRE2_STATE2, PRE3_STATE2,
DATA_STATE2,
      JAM_STATE2, NOSFD_STATE2, ERROR_STATE2, PRE4_STATE2);
--attribute state_encoding of states2:type is one_hot_one;

type states3 is (IDLE_STATE3, PRE1_STATE3, PRE2_STATE3, PRE3_STATE3,
DATA_STATE3,
      JAM_STATE3, NOSFD_STATE3, ERROR_STATE3);
--attribute state_encoding of states3:type is one_hot_one;

signal state1, newstate1: states1;
signal state2, newstate2: states2;
signal state3, newstate3: states3;

signal  carrierd, carrierdd: std_logic;
signal  error, rx_dv_in, rx_error_in: std_logic;
signal  no_sfd, no_sfd_in, no_data, data_valid: std_logic;
signal  prescale_in: std_logic;
signal  pout: std_logic_vector(9 downto 0);

constant jam:   std_logic_vector(1 downto 0) := "00";
constant pre:   std_logic_vector(1 downto 0) := "00";
constant sosb:  std_logic_vector(1 downto 0) := "01";
constant bad:   std_logic_vector(1 downto 0) := "10";
constant zero:  std_logic_vector(1 downto 0) := "11";
constant fifodata:      std_logic := '1';
constant symboldata:    std_logic := '0';
signal vdd:std_logic := '1';
signal vss:   std_logic := '0';

begin
--      Components

u1: rsynch port map (txclk, areset, carrier, carrierdd);
u3: rsynch port map (txclk, areset, rx_error_in, error);
u5: rdff1 port map (txclk, areset, rx_dv_in, data_valid);
u7: rdff1 port map (txclk, areset, no_sfd_in, no_data);
u8: ascount generic map(10) port map (txclk, areset, vss, vdd, pout);
u9: rdff1 port map(txclk, areset, prescale_in, prescale);


rx_dv_in    <= carrierdd AND rx_dv;    -- filter out glitches
rx_error_in <= carrierdd AND rx_error; -- filter out glitches
wptrclr     <= NOT(rx_dv_in AND NOT collision);
no_sfd_in   <= (no_sfd OR no_data) AND carrier;
```

```
prescale_in <= '1' when pout = "1111111111" else '0';

wptrinc <= '1';
rptrinc <= '1';
symbolinc <= '1';

--      State Machine Controllers

--      State Machine Controller Line 3

p3: process (carrier, collision, symbolend3, data_valid, error, state3)
begin

   case (state3) is

   when IDLE_STATE3 =>

         symbol3<= zero;
         switch3 <= symboldata;
         symbolclr <= '1';
         rptrclr <= '1';
         preamble<= '0';
         data  <= '0';
         no_sfd<= '0';
         idle  <= '1';
         col   <= '0';
         txdata<= '0';

         if (collision = '1') then
            newstate3 <= JAM_STATE3;
         elsif (carrier = '1') then
            newstate3 <= PRE1_STATE3;
         else
            newstate3 <= IDLE_STATE3;

         end if;

   when PRE1_STATE3 =>

         symbol3<= pre;
         switch3 <= symboldata;
         symbolclr <= '0';
         rptrclr <= '1';
         preamble<= '1';
         data  <= '0';
         no_sfd<= '0';
         idle  <= '0';
         col   <= '0';
         txdata<= '1';

         if (carrier = '0') then
            newstate3 <= IDLE_STATE3;
         elsif (collision = '1') then
            newstate3 <= JAM_STATE3;
```

```
        elsif (symbolend3 = '1') then
           newstate3 <= PRE2_STATE3;
        else
           newstate3 <= PRE1_STATE3;

        end if;

when PRE2_STATE3 =>

        symbol3<= pre;
        switch3 <= symboldata;
        symbolclr <= '0';
        rptrclr <= '1';
        preamble<= '1';
        data  <= '0';
        no_sfd<= '0';
        idle  <= '0';
        col   <= '0';
        txdata<= '1';

        if (carrier = '0') then
           newstate3 <= IDLE_STATE3;
        elsif (collision = '1') then
           newstate3 <= JAM_STATE3;
        elsif (symbolend3 = '1') then
           newstate3 <= PRE3_STATE3;
        else
           newstate3 <= PRE2_STATE3;

        end if;

when PRE3_STATE3 =>

        symbol3<= sosb;
        switch3 <= symboldata;
        symbolclr <= '0';
        rptrclr <= '1';
        preamble<= '1';
        data  <= '0';
        no_sfd<= '0';
        idle  <= '0';
        col   <= '0';
        txdata<= '1';

        if (carrier = '0') then
           newstate3 <= IDLE_STATE3;
        elsif (collision = '1') then
           newstate3 <= JAM_STATE3;
        elsif (symbolend3 = '1' AND error = '1') then
           newstate3 <= ERROR_STATE3;
        elsif (symbolend3 = '1' AND data_valid = '0') then
           newstate3 <= NOSFD_STATE3;
        elsif (symbolend3 = '1' AND data_valid = '1') then
           newstate3 <= DATA_STATE3;
```

```
        else
           newstate3 <= PRE3_STATE3;

        end if;

when DATA_STATE3 =>

        symbol3<= jam;
        switch3 <= fifodata;
        symbolclr <= '0';
        rptrclr <= '0';
        preamble<= '0';
        data  <= '1';
        no_sfd<= '0';
        idle  <= '0';
        col   <= '0';
        txdata<= '1';

        if (carrier = '0') then
           newstate3 <= IDLE_STATE3;
        elsif (collision = '1') then
           newstate3 <= JAM_STATE3;
        elsif (symbolend3 = '1' AND error = '1') then
           newstate3 <= ERROR_STATE3;
        else
           newstate3 <= DATA_STATE3;

        end if;

when JAM_STATE3 =>

        symbol3<= jam;
        switch3 <= symboldata;
        symbolclr <= '0';
        rptrclr <= '1';
        preamble<= '0';
        data  <= '0';
        no_sfd<= '0';
        idle  <= '0';
        col   <= '1';
        txdata<= '1';

        if (carrier = '0') then
           newstate3 <= IDLE_STATE3;
        else
           newstate3 <= JAM_STATE3;

        end if;

when NOSFD_STATE3 =>

        symbol3<= jam;
        switch3 <= symboldata;
        symbolclr <= '0';
```

```
            rptrclr <= '0';
            preamble<= '0';
            data  <= '1';
            no_sfd<= '1';
            idle  <= '0';
            col   <= '0';
            txdata<= '1';

            if (carrier = '0') then
               newstate3 <= IDLE_STATE3;
            elsif (collision = '1') then
               newstate3 <= JAM_STATE3;
            elsif (symbolend3 = '1' AND error = '1') then
               newstate3 <= ERROR_STATE3;
            else
               newstate3 <= NOSFD_STATE3;

            end if;

      when ERROR_STATE3 =>

            symbol3<= bad;
            switch3 <= symboldata;
            symbolclr <= '0';
            rptrclr <= '0';
            preamble<= '0';
            data  <= '1';
            no_sfd<= '0';
            idle  <= '0';
            col   <= '0';
            txdata<= '1';

            if (carrier = '0') then
               newstate3 <= IDLE_STATE3;
            elsif (collision = '1') then
               newstate3 <= JAM_STATE3;
            else
               newstate3 <= ERROR_STATE3;

            end if;
      end case;
   end process;

   p3clk: process (txclk,areset)
      begin
         if areset = '1' then
            state3 <= idle_state3;
         elsif (txclk'event and txclk='1') then
            state3 <= newstate3;
         end if;
      end process;
   end archcontrol;
```

Listing 6-10 Core controller code

There are several outputs in each state. *Symbolclr* is used to clear the 3-bit counter that controls the symbol boundaries, *preamble* indicates that the core controller is generating preamble, *data* indicates that the core controller is transmitting data, *no_sfd* indicates that no data was found, *idle* indicates that the network is idle, and *col* indicates that there is a collision and that jam characters are being transmitted. All of these signals are propagated to the top level for observation. *Txdata* is used in each of the port controllers to assert *tx_en*. These signals do not need to be repeated in the design of the additional two state machines.

The *symbol* and *switch* outputs are used to control two output multiplexers for each transmit pair. *Symbol* indicates which symbol to generate. *Switch* indicates whether the symbol or FIFO data is selected to transmit to the outputs. The *symbol* and *switch* signals are separate for each transmit pair in order that one transmit pair may transmit symbols while another transmits data.

Symbol Generator and Output Multiplexer

Listing 6-11 below is the design of the symbol generator and output multiplexer. The jam, sosb2, and bad characters for each of the transmission pairs are defined by the IEEE 802.3 standard. The values of the symbol counter and multiplexers are used to generate these symbols. Two multiplexers for each transmit pair are used. The first multiplexer selects a symbol, and the second multiplexer selects either the symbol or data from the FIFO.

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

entity symbolmux is port(
       txclk:   in std_logic;-- Reference TX_CLK
       areset:  in std_logic;-- Async Reset
       symbolclr: in std_logic;-- Symbol Counter Clear
       symbolinc:  in std_logic;-- Symbol Counter Increment
       switch1: in std_logic;-- Line 1 D/S Switch Control
       switch2:  in std_logic;-- Line 2 D/S Switch Control
       switch3:  in std_logic;-- Line 3 D/S Switch Control
       symbol1: in std_logic_vector(1 downto 0);-- Line 1 Symbol Mux
Control
       symbol2:  in std_logic_vector(1 downto 0);-- Line 2 Symbol Mux
Control
       symbol3:  in std_logic_vector(1 downto 0);-- Line 3 Symbol Mux
Control
       dmuxout:  in std_logic_vector(5 downto 0);-- FIFO Data Input
       symbolend1: buffer std_logic; -- End of Line 1 Symbol
       symbolend2: out std_logic;-- End of Line 2 Symbol
       symbolend3:  out std_logic;-- End of Line 3 Symbol
       txd5:    out std_logic;-- Data
       txd4:    out std_logic;-- Data
       txd3:    out std_logic;-- Data
       txd2:    out std_logic;-- Data
       txd1:    out std_logic;-- Data
       txd0:    out std_logic);-- Data
end symbolmux;

use work.symboltop_pkg.all;
architecture archsymbolmux of symbolmux is
-- signals
```

```
signalclearcount: std_logic;
signalsymbolcount: std_logic_vector(2 downto 0);

signalsosb1, sosb2, sosb3, bad1, bad2, bad3, jam: std_logic_vector(1 downto
0);
signaltxd, muxout, smuxout: std_logic_vector(5 downto 0);

-- Constants
constant plus : std_logic_vector(1 downto 0) := "10";
constant zero : std_logic_vector(1 downto 0) := "00";
constant minus: std_logic_vector(1 downto 0) := "01";

begin
-- Components

u1: ascount generic map(CounterSize => 3)
      port map (txclk, areset, clearcount, symbolinc, symbolcount); --
Symbol Count

u2: rdff generic map (size => 6)
    port map (txclk, areset, muxout, txd);-- Output Latch


txd5 <= txd(5);
txd4 <= txd(4);
txd3 <= txd(3);
txd2 <= txd(2);
txd1 <= txd(1);
txd0 <= txd(0);


symbolend1<= symbolcount(0) AND NOT symbolcount(1) AND  symbolcount(2);
symbolend2<= symbolcount(0) AND NOT symbolcount(1) AND NOT symbolcount(2);
symbolend3<= symbolcount(0) AND  symbolcount(1) AND NOT symbolcount(2);
clearcount<= symbolend1 OR symbolclr;

-- Special Symbol Mux
with symbol1 select
   smuxout(1 downto 0) <=
      jam   when "00",
      sosb1 when "01",
      bad1  when "10",
      zero  when others;

-- Line 1 Switch Mux
with switch1 select
      muxout(1 downto 0) <=
         smuxout(1 downto 0) when '0',
         dmuxout(1 downto 0) when others;

-- Special Symbol Mux (Line 2)
with symbol2 select
   smuxout(3 downto 2) <=
      jam   when "00",
```

```
      sosb2 when "01",
      bad2  when "10",
      zero  when others;

-- Line 2 Switch Mux
with switch2 select
      muxout(3 downto 2) <=
         smuxout(3 downto 2) when '0',
         dmuxout(3 downto 2) when others;

-- Special Symbol Mux (Line 3)
with symbol3 select
   smuxout(5 downto 4) <=
      jam   when "00",
      sosb3 when "01",
      bad3  when "10",
      zero  when others;

-- Line 3 Switch Mux
with switch3 select
      muxout(5 downto 4) <=
         smuxout(5 downto 4) when '0',
         dmuxout(5 downto 4) when others;

-- Jam/Preamble Generation (All Lines)
with symbolcount(0) select
   jam <=
      plus  when '0',
      minus when others;

-- SOSB Generation (Line 1)
with symbolcount select
   sosb1 <=
      plus  when "000",
      minus when "001",
      plus  when "010",
      minus when "011",
      minus when "100",
      plus  when "101",
      zero  when others;

-- SOSB Generation (Line 2)
with symbolcount select
   sosb2 <=
      minus when "000",
      plus  when "001",
      plus  when "010",
      minus when "011",
      plus  when "100",
      minus when "101",
      zero  when others;

-- SOSB Generation (Line 3)
with symbolcount select
```

```
   sosb3 <=
      plus  when "000",
      minus when "001",
      minus when "010",
      plus  when "011",
      plus  when "100",
      minus when "101",
      zero  when others;

-- Bad Code Generation (Line 1)
with symbolcount select
   bad1 <=
      minus when "000",
      minus when "001",
      minus when "010",
      plus  when "011",
      plus  when "100",
      plus  when "101",
      zero  when others;

-- Bad Code Generation (Line 2)
with symbolcount select
   bad2 <=
      plus  when "000",
      plus  when "001",
      minus when "010",
      minus when "011",
      minus when "100",
      plus  when "101",
      zero  when others;

-- Bad Code Generation (Line 3)
with symbolcount select
   bad3 <=
      minus when "000",
      plus  when "001",
      plus  when "010",
      plus  when "011",
      minus when "100",
      minus when "101",
      zero  when others;
end archsymbolmux;
```

Listing 6-11 Symbol generator and output multiplexer code

Top-Level Design

Before the design units can be connected in the top-level of the design hierarchy, a package, *coretop_pkg*, must be created in which the design units are declared as components. Once the components are declared and made visible to the top-level design, they can be instantiated. *Listing 6-12* is the design of the repeater core logic in which the design units are interfaced to each other and the top-level I/O.

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

entity core is port(
   reset                    : in std_logic; -- Global Reset
   clk                      : in std_logic; -- to CKTPAD for TX_CLK
   rxd5                     : in std_logic; -- RXD5
   rxd4                     : in std_logic; -- RXD4
   rxd3                     : in std_logic; -- RXD3
   rxd2                     : in std_logic; -- RXD2
   rxd1                     : in std_logic; -- RXD1
   rxd0                     : in std_logic; -- RXD0
   rx_dv                    : in std_logic; -- RX_DV
   rx_er                    : in std_logic; -- RX_ER

   clk1                     : in std_logic; -- RX_CLK1
   crs1                     : in std_logic; -- CRS1
   enable1_bar              : in std_logic; -- ENABLE1
   link1_bar                : in std_logic; -- LINK1

   clk2                     : in std_logic; -- RX_CLK2
   crs2                     : in std_logic; -- CRS2
   enable2_bar              : in std_logic; -- ENABLE2
   link2_bar                : in std_logic; -- LINK2

   clk3                     : in std_logic; -- RX_CLK3
   crs3                     : in std_logic; -- CRS3
   enable3_bar              : in std_logic; -- ENABLE3
   link3_bar                : in std_logic; -- LINK3

   clk4                     : in std_logic; -- RX_CLK4
   crs4                     : in std_logic; -- CRS4
   enable4_bar              : in std_logic; -- ENABLE4
   link4_bar                : in std_logic; -- LINK4

   clk5                     : in std_logic; -- RX_CLK5
   crs5                     : in std_logic; -- CRS5
   enable5_bar              : in std_logic; -- ENABLE5
   link5_bar                : in std_logic; -- LINK5

   clk6                     : in std_logic; -- RX_CLK6
   crs6                     : in std_logic; -- CRS6
   enable6_bar              : in std_logic; -- ENABLE6
   link6_bar                : in std_logic; -- LINK6

   clk7                     : in std_logic; -- RX_CLK7
   crs7                     : in std_logic; -- CRS7
   enable7_bar              : in std_logic; -- ENABLE7
   link7_bar                : in std_logic; -- LINK7

   clk8                     : in std_logic; -- RX_CLK8
   crs8                     : in std_logic; -- CRS8
   enable8_bar              : in std_logic; -- ENABLE8
   link8_bar                : in std_logic; -- LINK8
```

```
rx_en1          : out std_logic;        -- RX_EN1
tx_en1          : out std_logic;        -- TX_EN1
partition1_bar  : inout std_logic;        -- PARTITION1
jabber1_bar     : inout std_logic;        -- JABBER1

rx_en2          : out std_logic;        -- RX_EN2
tx_en2          : out std_logic;        -- TX_EN2
partition2_bar  : inout std_logic;        -- PARTITION2
jabber2_bar     : inout std_logic;        -- JABBER2

rx_en3          : out std_logic;        -- RX_EN3
tx_en3          : out std_logic;        -- TX_EN3
partition3_bar  : inout std_logic;        -- PARTITION3
jabber3_bar     : inout std_logic;        -- JABBER3

rx_en4          : out std_logic;        -- RX_EN4
tx_en4          : out std_logic;        -- TX_EN4
partition4_bar  : inout std_logic;        -- PARTITION4
jabber4_bar     : inout std_logic;        -- JABBER4

rx_en5          : out std_logic;        -- RX_EN5
tx_en5          : out std_logic;        -- TX_EN5
partition5_bar  : inout std_logic;        -- PARTITION5
jabber5_bar     : inout std_logic;        -- JABBER5

rx_en6          : out std_logic;        -- RX_EN6
tx_en6          : out std_logic;        -- TX_EN6
partition6_bar  : inout std_logic;        -- PARTITION6
jabber6_bar     : inout std_logic;        -- JABBER6

rx_en7          : out std_logic;        -- RX_EN7
tx_en7          : out std_logic;        -- TX_EN7
partition7_bar  : inout std_logic;        -- PARTITION7
jabber7_bar     : inout std_logic;        -- JABBER7

rx_en8          : out std_logic;        -- RX_EN8
tx_en8          : out std_logic;        -- TX_EN8
partition8_bar  : inout std_logic;        -- PARTITION8
jabber8_bar     : inout std_logic;        -- JABBER8

txd5            : out std_logic;        -- TXD5
txd4            : out std_logic;        -- TXD4
txd3            : out std_logic;        -- TXD3
txd2            : out std_logic;        -- TXD2
txd1            : out std_logic;        -- TXD1
txd0            : out std_logic;        -- TXD0

txdata          : inout std_logic;        -- TX_ENall
idle            : out std_logic;        -- Idle Generation
preamble        : out std_logic;        -- Preamble Generation
data            : out std_logic;        -- Data Generation
jam             : inout std_logic;        -- Jam Generation
collision       : inout std_logic;        -- Collision Indication
```

```
   wptr2          : out std_logic;        -- Write Pointer2
   wptr1          : out std_logic;        -- Write Pointer1
   wptr0          : out std_logic;        -- Write Pointer0
   rptr2          : out std_logic;        -- Read Pointer2
   rptr1          : out std_logic;        -- Read Pointer1
   rptr0          : out std_logic);      -- Read Pointer0
end core;

--use work.rtlpkg.all;
use work.coretop_pkg.all;

architecture archcore of core is
   signal txclk1, nosel, areset, sel1, sel2, sel3, sel4: std_logic;
   signal sel5, sel6, sel7, sel8, rxclk, txclk: std_logic;
   signal activity1, activity2, activity3, activity4: std_logic;
   signal activity5, activity6, activity7, activity8: std_logic;
   signal carrier: std_logic;
   signal wptrclr, wptrinc, rptrclr, rptrinc,symbolinc:std_logic;
   signal switch1, switch2, switch3: std_logic;
   signal symbolend1, symbolend2, symbolend3: std_logic;
   signal symbolclr : std_logic;
   signal symbol1, symbol2, symbol3: std_logic_vector(1 downto 0);
   signal dmuxout: std_logic_vector(5 downto 0);
   signal prescale: std_logic;

begin
--      Components
u1: clockmux8 port map
   (areset,
   clk1, clk2, clk3, clk4, clk5, clk6, clk7, clk8, txclk1,
   sel1, sel2, sel3, sel4, sel5, sel6, sel7, sel8, nosel,
   rxclk);

u2: arbiter8 port map
   (txclk, areset,
   activity1, activity2, activity3, activity4,
   activity5, activity6, activity7, activity8,
   sel1, sel2, sel3, sel4, sel5, sel6, sel7, sel8,
   nosel, carrier, collision);

u3: fifo port map
   (rxclk, txclk, areset, wptrclr, wptrinc, rptrclr,
   rptrinc, rxd5, rxd4, rxd3, rxd2, rxd1, rxd0,
   dmuxout,  wptr2, wptr1, wptr0, rptr2, rptr1, rptr0);

u4: symbolmux port map
   (txclk, areset,
   symbolclr, symbolinc, switch1, switch2, switch3, symbol1,
   symbol2, symbol3, dmuxout, symbolend1, symbolend2,
   symbolend3, txd5, txd4, txd3, txd2, txd1, txd0);

u5: control port map
   (txclk, areset, carrier, collision, rx_er, rx_dv,
   symbolend1, symbolend2, symbolend3, symbolclr, symbolinc,
```

```
    symbol1, symbol2, symbol3, switch1, switch2, switch3,
    wptrclr, wptrinc, rptrclr, rptrinc, txdata, idle,
    preamble, data, jam, prescale);

u6: porte port map
    (txclk, areset,
    crs1, enable1_bar, link1_bar,
    sel1, carrier, collision, jam, txdata, prescale, rx_en1, tx_en1,
    activity1, jabber1_bar, partition1_bar);

u7: porte port map
    (txclk, areset,
    crs2, enable2_bar, link2_bar,
    sel2, carrier, collision, jam, txdata, prescale, rx_en2, tx_en2,
    activity2, jabber2_bar, partition2_bar);

u8: porte port map
    (txclk, areset,
    crs3, enable3_bar, link3_bar,
    sel3, carrier, collision, jam, txdata, prescale, rx_en3, tx_en3,
    activity3, jabber3_bar, partition3_bar);

u9: porte port map
    (txclk, areset,
    crs4, enable4_bar, link4_bar,
    sel4, carrier, collision, jam, txdata, prescale, rx_en4, tx_en4,
    activity4, jabber4_bar, partition4_bar);

u10: porte port map
    (txclk, areset,
    crs5, enable5_bar, link5_bar,
    sel5, carrier, collision, jam, txdata, prescale, rx_en5, tx_en5,
    activity5, jabber5_bar, partition5_bar);

u11: porte port map
    (txclk, areset,
    crs6, enable6_bar, link6_bar,
    sel6, carrier, collision, jam, txdata, prescale, rx_en6, tx_en6,
    activity6, jabber6_bar, partition6_bar);

u12: porte port map
    (txclk, areset,
    crs7, enable7_bar, link7_bar,
    sel7, carrier, collision, jam, txdata, prescale, rx_en7, tx_en7,
    activity7, jabber7_bar, partition7_bar);

u13: porte port map
    (txclk, areset,
    crs8, enable8_bar, link8_bar,
    sel8, carrier, collision, jam, txdata, prescale, rx_en8, tx_en8,
    activity8, jabber8_bar, partition8_bar);

txclk <= clk;
txclk1<= clk;
```

```
areset <= reset;

end archcore;
```

Listing 6-12 Top-level repeater core logic design code

Listing 6-12 completes the design of the repeater core logic. In chapter 8, "Synthesis to Final Design Implementation," we discuss issues concerning the synthesis of this design if it is to be implemented in an FPGA. In chapter 9, "Creating Test Fixtures," we explain how to create a VHDL test fixture to simulate both the source code and the post-synthesis and place and route model.

Exercises

1. The 14-bit Jabber counters described on page 193 are built with a 10-bit base counter and 4-bit counters. Build them with an 11-bit base counter and 3-bit counters. What is the accuracy now? Are any other combinations possible? How many resources are required for each? develop an algorithm for accuracy and resources based on using an n-bit base counter and m-bit counters for the remainder of the jabber counters, with $n+m = 14$.

2. Write the VHDL code for a 4-bit carry-look ahead adder. Create a package declaration for this code and use it to create a 32-bit version. Compile and synthesize this into a a)CPLD b) FPGA.

3. Study the components of the MATH library in the *Warp* tool. Create your own library of multipliers and shifters.

4. Rewrite the 'arbiter8' VHDL code in listing 6-8 using FOR-GENERATE and FOP-LOOP constructs

5. Design a glitch-free clock multiplexer circuit.

6. Do a worst-case analysis for the depth of the FIFO alluded to on page 204.

7. Complete the partial code presented in *Listing 6-10* by including the two state machine declarations.

8. Write a process to create a random number generator. Can you synthesize this procedure ? Justify.

9. What recommendations and changes would you make to improve the performance of the network repeater.

10. Complete the network repeater design. Compile, synthesize, place and route, and verify timing and functionality.

7 Functions and Procedures

Functions and procedures are high-level design constructs to compute values or define partial processes that you can use for type conversions, operator overloading, or as an alternative to component instantiation. Below is a type conversion function.

```
1  FUNCTION bl2bit(a:BOOLEAN) RETURN BIT IS
2    BEGIN
3      IF a THEN
4        RETURN '1';
5      ELSE
6        RETURN '0';
7      END IF;
8    END bl2bit;
```

Listing 7-1 A Boolean to bit type conversion function

As you can see, *Listing 7-1* describes a function that is used to convert the type Boolean to type bit, which are both predefined by the IEEE 1076 standard. Such functions are referred to as type conversion functions. Line 1 declares the function *bl2bit* and defines the input parameter as type Boolean and return type as bit. Lines 2 and 8 begin and end the function definition. All statements within the function definition must be sequential statements. Lines 3 through 7 are sequential statements that define the return value based on the Boolean condition *a*. If *a* is TRUE, then the return value is '1'; otherwise, the return value is '0' (return type is bit). Other often-used type conversion functions are bit to Boolean, bit to std_logic, and bit_vector to std_logic_vector. You'll have an opportunity to write a couple of these conversion functions in the exercises at the end of the chapter.

A bit to Boolean (or Boolean to bit) type conversion function can be helpful in writing Boolean equations or evaluation clauses. For example, if the signal *clk* is of type Boolean, then you can write

```
                        wait until clk;
instead of              wait until clk='1'
or                      if (clk'event and clk) then...
instead of              if (clk'event and clk='1') then...
Likewise                if ((A AND B) XOR (C AND D)) then...
can be substituted for  if (((A AND B) XOR (C AND D))='1') then ..
```

One way is not better than the other; however, VHDL provides the flexibility to meet different coding styles.

Functions

Function parameters can only be inputs; therefore, the parameters cannot be modified. The parameter *a* in *Listing 7-1* above is an input only. By default, all parameters are of mode IN, and the mode does not need to be explicitly declared. After all, it is the only legal mode for function parameters. Functions can return only one argument. (Procedures, as you'll see later, can return multiple arguments.) As mentioned in the explanation of the example above, all statements within the function definition are sequential. Because of this, no signals can be declared or assigned in

functions; however, variables may be declared in a function's declaration region and assigned values within the function definition. We'll take a look at a couple of examples to help you understand how to create your own functions. After that, we'll explain how to put these functions to use in your designs (i.e., where to declare and define functions in order to use them).

bv2i

Read through *Listing 7-2* to determine what this function does.

```
 1   -- bv2i
 2   --  Bit_vector to Integer.
 3   --  In:     bit_vector.
 4   --  Return: integer.
 5   --
 6   FUNCTION bv2I (bv : Bit_vector) RETURN integer IS
 7     VARIABLE result, abit : integer := 0;
 8     VARIABLE  count            : integer := 0;
 9   BEGIN  --  BV2I
10     bits : FOR I IN bv'low to bv'high LOOP
11     abit := 0;
12     IF ((bv(I) = '1')) THEN
13       abit := 2**(I - bv'low);
14     END IF;
15       result := result + abit;                --  Add in bit if '1'.
16     count := count + 1;
17     EXIT bits WHEN count = 32;          --  32 bits max.
18     END LOOP bits;
19     RETURN (result);
20   END bv2I;
```

Listing 7-2 A type conversion function

Lines 1 through 5 of *Listing 7-2* are comment lines that document the function name, describe the type conversion function, and indicate the input parameter and return type. This function takes as its input a bit_vector, performs a binary to decimal conversion, and returns an integer. Line 6 indicates that the input parameter is unconstrained because the widths of the bit_vectors that will be passed into this function are not known a priori. In fact, it is deliberately generic so that the size of the bit_vector is not constrained. Of course, when this function is called in a design, the bit_vector will have to be constrained. For signals to represent a collection of wires connected to gates, the widths of those signals must be defined at some level.

Lines 7 and 8 make up the function's declaration region where variables can be declared, as in a process's declaration region. In this *bv2i* function, three variables are declared as integers and initialized to zero. All of the integers could have been declared with one variable declaration, but the declarations are separate to emphasize the different purposes that the variables serve. The function definition is enclosed between the BEGIN and END statements of lines 9 and 20. Line 10 begins a loop that starts with the lowest order bit of the bit_vector *bv*. The attributes 'LOW and 'HIGH are predefined VHDL attributes that are used here to return the lowest and highest indices of the bit_vector that are passed into the function as a parameter. Therefore, regardless of the order of bit_vector *bv*—(x downto y) or (y to x)—y will be considered the least significant bit, and the integer value created for the bit_vector will reflect that y is the LSB and x is the MSB. For example, the two bit_vectors, *a* and *b,* may be defined as follows:

```
signal a: bit_vector(13 downto 6);
signal b: bit_vector(6 to 13);
```

For each of these bit_vectors, *a(6)* and *b(6)* will be considered the LSB. The function could have been written to always interpret the value on the left as the MSB, but that function is left as an exercise for the reader.

The loop is used to ascend the bit_vector from LSB to MSB, with the variable *i* used to index the bit_vector. Line 11 initializes the variable *abit* to zero for each iteration of the loop. For each bit of *bv* that is asserted, *abit* is assigned the appropriate power of two determined by its position in the bit_vector. The position is determined by subtracting the lowest index for the bit_vector ('LOW) from the current index. Consider our bit_vector *a(13 downto 6)*: If *a(8)* is a '1', then *abit* is assigned the integer value 4 because *i* is 8, *bv'LOW* is 6, *I - bv'LOW* is 2, and 2**2 is 4. This represents the binary number "100". The value of *abit* is added to *result* for each iteration of the loop. *Count* is used to determine the width of the bit_vector being converted to an integer. The range of integers that a VHDL tool must support extends to 2^{32} so *count* ensures that the integer returned is within the valid range. When the loop finishes, or is exited, *result* is returned (line 19), and the bit_vector-to-integer conversion is complete.

i2bv

Function *i2bv* performs just the opposite conversion: integer to bit_vector. Read through *Listing 7-3* to understand how this conversion is accomplished, then we'll provide a brief explanation.

```
-- i2bv
--  Integer to Bit_vector.
--  In:     integer, value and width.
--  Return: bit_vector, with right bit the most significant.
--
FUNCTION i2bv   (VAL, width : INTEGER) RETURN BIT_VECTOR IS
    VARIABLE result : BIT_VECTOR (0 to width-1) := (OTHERS=>'0');
    VARIABLE bits   : INTEGER := width;
BEGIN
    IF (bits > 32) THEN              --  Avoid overflow errors.
      bits := 32;
    ELSE
      ASSERT 2**bits > VAL REPORT
        "Value too big FOR BIT_VECTOR width"
        SEVERITY WARNING;
    END IF;

    FOR i IN 0 TO bits - 1 LOOP
        IF ((val/(2**i)) MOD 2 = 1) THEN
            result(i) := '1';
        END IF;
    END LOOP;

    RETURN (result);
END i2bv;
```

Listing 7-3 An integer to bit_vector type conversion function

This function takes as its inputs an integer value and the size (width) of the bit_vector that is to be returned. The function performs a decimal to binary conversion and returns the value of the integer as a bit_vector.

In the function declaration, *result* is declared as a variable of type bit_vector whose size is determined by the value of *width*. Variable *bits* is declared as an integer and is initialized to the value of *width*. Variable *bits* may be modified in the function, and therefore it is used rather than *width*. *Width* is a parameter to this function, must be mode IN, and cannot be modified.

The function definition begins by evaluating the size of the bit_vector to be returned. If it is greater than 32 (the largest integer that tools that process VHDL must handle is 2^{32}), then the size of the bit_vector is truncated to 32. Otherwise, the integer value *val* is evaluated to determine if it can be converted to a bit_vector of size *bits*. If the width of the bit_vector is too small to handle the integer, then the ASSERT condition is true and a severity warning is issued.

Next, *result* is computed by assigning a '1' to each bit of the bit_vector if *val* divided by 2^i (where *i* is the current index of the bit_vector) has a remainder of one. After all iterations of the loop, *result* is returned and the conversion is complete.

inc_bv

We'll look at one more function definition before discussing how to use functions such as these. Examine *Listing 7-4* to determine how it accomplishes an incrementing function.

```
-- inc_bv
--  Increment bit_vector.
--  In:     bit_vector.
--  Return: bit_vector.
--
FUNCTION inc_bv   (a      : BIT_VECTOR) RETURN BIT_VECTOR IS
    VARIABLE s      : BIT_VECTOR (a'RANGE);
    VARIABLE carry  : BIT;
BEGIN
    carry   := '1';

    FOR i IN a'LOW TO a'HIGH LOOP
        s(i)        := a(i) XOR carry;
        carry       := a(i) AND carry;
    END LOOP;

    RETURN  (s);
END inc_bv;
```

Listing 7-4 A function for incrementing bit_vectors

Function *inc_bv* takes as its input a bit_vector, increments the value of that bit_vector, and returns a bit_vector of the same size as the input bit_vector.

The attribute 'RANGE is a predefined VHDL attribute that returns the range of an array. It enables variable *s* to be declared as a bit_vector with the same range—(x downto y) or (y to x)—as the input vector *a*. *Carry* is defined as a bit.

The function definition sets the first carry input to be a '1' in order to increment the vector. The value of *a(i)* exclusive-or *carry* is assigned to *s(i)* for each bit of the input vector *a*. *Carry* is initially '1' and is recomputed for each bit of the vector. The result of adding one to the bit_vector *a* is therefore bit_vector *s*.

Now that we've looked at how functions are created, we'll explore how to use these functions within design entities and architectures.

Using Functions

Functions may be defined in architecture declaration regions, in which case the function definition also serves as the function declaration. Alternatively, a package may be used to declare a function with the definition of that function occurring in the associated package body. You may wish to create a collection of type conversion functions and place them in a package and library so that you can easily use them in any design. Also, if one function requires the use of another function, you will likely find it less cumbersome to have those function declarations and definitions in a package rather than in the architecture of the entity that you are describing. We'll take a look at both ways of declaring functions. You can decide which method meets your particular style and needs.

To begin with, let's use the *bl2bit* function of *Listing 7-1* in a design. This function will allow us to convert a Boolean value to type bit for use in a port map of a component whose input must be of type bit. First read through *Listing 7-5*. This listing simply defines a D-type flip-flop and includes its component declaration in a package called *flops*. The flip-flop requires that the input be of type bit. Next, read through *Listing 7-6*, to see how *bl2bit* is used to convert a Boolean signal to type bit so that it may be used in the port map of the *dff*. The function definition serves as the function declaration. It is located in the declaration region of the architecture and is used within the design architecture.

```
package flops is
component dff port(
        d,clk:  in bit;
        q:      out bit);
end component;
end flops;

entity dff is port(
        d,clk:  in bit;
        q:      out bit);
end dff;

architecture archdff of dff is
begin
process
begin
        wait until clk='1';
                q <= d;
end process;
end archdff;
```

Listing 7-5 Defining a D-type flip-flop in a package for which the port types must be bit

```
entity convert is port(
        a, b, c:        in boolean;
        clk:            in bit;
        x:              out bit);
end convert;

use work.flops.all;
architecture archconvert of convert is
        signal d: boolean;

  FUNCTION bl2bit(a:BOOLEAN) RETURN BIT IS
    BEGIN
      IF a THEN
        RETURN '1';
      ELSE
        RETURN '0';
      END IF;
    END bl2bit;

begin
        d <= ((A OR B) XOR C);
        u1: dff port map(bl2bit(d),clk,x);
end archconvert;
```

Listing 7-6 Defining a function in the architecture declaration region and using the function within a port map

The *bl2bit* conversion function was used in the instantiation statement itself to convert the value of *d* to its equivalent bit value. This enables you to write succinct code. Otherwise, you would need to create temporary signals to hold the conversion values before using those signals in the instantiation, as in *Listing 7-7* below.

```
use work.flops.all;
architecture archconvert of convert is
        signal d: boolean;
           signal dummy: bit;

  FUNCTION bl2bit(a:BOOLEAN) RETURN BIT IS
    BEGIN
      IF a THEN
        RETURN '1';
      ELSE
        RETURN '0';
      END IF;
    END bl2bit;

begin
        d <= ((A OR B) XOR C);
        dummy <= bl2bit(d);
        u1: dff port map(dummy,clk,x);
```

```
end archconvert;
```

Listing 7-7 An equivalent to *Listing 7-6* wherein the ports are first converted using local signals

In both of the above listings (*Listing 7-6* and *Listing 7-7*), the *bl2bit* function is defined (and, hence, declared) in the architecture declaration region. Alternatively, it can be a part of a type conversion package where many type conversion functions exist. In our case, we'll move the *bl2bit* function into a package named conversions that contains four type conversion functions (*Listing 7-8*).

```
package conversions is
  FUNCTION bl2bit(a:BOOLEAN) RETURN BIT;
  FUNCTION bit2bl(in1:BIT) RETURN BOOLEAN;
  FUNCTION bv2i (bv : Bit_vector) RETURN integer;
  FUNCTION i2bv   (VAL, width : INTEGER) RETURN BIT_VECTOR;
end conversions;

package body conversions is

    -- bl2bit
    --  Boolean to bit.
    --  In:     iBoolean.
    --  Return: Bit.
    --
  FUNCTION bl2bit(a:BOOLEAN) RETURN BIT IS
    BEGIN
      IF a THEN
        RETURN '1';
      ELSE
        RETURN '0';
      END IF;
    END bl2bit;

    -- bit2bl
    --  Bit to boolean.
    --  In: Bit.
    --  Return: Boolean
    --
   FUNCTION bit2bl(in1:BIT) RETURN BOOLEAN IS
     BEGIN
       IF (in1 = '1') THEN
         RETURN TRUE;
       ELSE
         RETURN FALSE;
       END IF;
     END bit2bl;

    -- bv2i
    --  Bit_vector to Integer.
    --  In:     bit_vector.
    --  Return: integer.
    --
    FUNCTION bv2i  (bv : Bit_vector) RETURN integer IS
```

```
    VARIABLE result, abit : integer := 0;
    VARIABLE  count             : integer := 0;
  BEGIN  --  bv2i
    bits : FOR I IN bv'low to bv'high LOOP
    abit := 0;
    IF ((bv(I) = '1')) THEN
      abit := 2**(I - bv'low);
    END IF;
      result := result + abit;                --  Add in bit if '1'.
    count := count + 1;
    EXIT bits WHEN count = 32;          --  32 bits max.
    END LOOP bits;
    RETURN (result);
  END bv2i;

  -- i2bv
  --  Integer to Bit_vector.
  --  In:     integer, value and width.
  --  Return: bit_vector, with right bit the most significant.
  --
  FUNCTION i2bv   (VAL, width : INTEGER)    RETURN BIT_VECTOR IS
      VARIABLE result : BIT_VECTOR (0 to width-1) := (OTHERS=>'0');
      VARIABLE bits   : INTEGER := width;
  BEGIN
      IF (bits > 32) THEN               --  Avoid overflow errors.
        bits := 32;
      ELSE
        ASSERT 2**bits > VAL REPORT
          "Value too big FOR BIT_VECTOR width"
          SEVERITY WARNING;
      END IF;

      FOR i IN 0 TO bits - 1 LOOP
          IF ((val/(2**i)) MOD 2 = 1) THEN
              result(i) := '1';
          END IF;
      END LOOP;

      RETURN (result);
  END i2bv;
end conversions;
```

Listing 7-8 Package containing four type conversion functions

Listing 7-8 declares the functions bl2bit, bit2bl, bv2i, and i2bv in a function declaration and defines the functions in the package body. *Listing 7-9* below then includes the package in order to use the function bl2bit.

```
entity convert is port(
       a, b, c:        in boolean;
       clk:            in bit;
       x:              out bit);
```

```
end convert;

use work.conversions.bl2bit;          --could have use ".all" but not needed
use work.flops.all;
architecture archconvert of convert is
        signal d: boolean;
begin
        d <= ((A OR B) XOR C);
        u1: dff port map(bl2bit(d),clk,x);
end archconvert;
```

Listing 7-9 Using functions declared in a package

Overloading Operators

A powerful use of functions is to overload operators. In previous chapters, you've seen how overloaded operators provided by a synthesis tool can be used. In this section, we'll discuss what an overloaded operator is and how it works.

An overloaded operator enables you to use an operator to operate on operands of types that are not supported by the native VHDL operator. For instance, the + operator is defined by the IEEE 1076 standard to operate on numeric types (integer, floating point, and physical types) but not with enumerated types like std_logic or bit_vector. To add a constant integer to a signal of type std_logic, an overloaded operator is required. The overloaded operator is a function declaration that defines the operator for the given types and a function definition that indicates how the operator is to work on the given types. *Listing 7-10* is a design in which an integer is added to a bit_vector. Other useful addition operations include, among others, addition of a bit_vector to an integer, a bit_vector to a bit, a std_logic_vector to an integer, or a std_logic_vector to a bit.

```
entity counter is port(
        clk, rst, pst, load,counten:    in bit;
        data:                           in bit_vector(3 downto 0);
        count:                          buffer bit_vector(3 downto 0));
end counter;

use work.myops.all;
architecture archcounter of counter is
begin
upcount: process (clk, rst, pst)
        begin
                if rst = '1' then
                        count <= "0000";
                elsif pst = '1' then
                        count <= "1111";
                elsif (clk'event and clk= '1') then
                        if load = '1' then
                                count <= data;
                        elsif counten = '1' then
                                count <= count + 1;
                        end if;
                end if;
        end process upcount;
```

```
end archcounter;
```

Listing 7-10 A counter in which the + operator has operands of types bit_vector and integer

The code in *Listing 7-10* makes use of the + operator for the statement "count <= count + 1;". The native VHDL operator will not handle this addition because the operands are bit_vector and integer. The overloaded operator must come from within the package *myops*.

Overloaded operators, such as the one used in *Listing 7-10*, allow you to perform operations on data types for which the function is not already defined. You can create several functions that define the same operation for different types. VHDL synthesis and simulation tools are required to look for the function (or operator in this case) that matches the parameters that are being used.

Listing 7-11 contains a package declaration and package body that declare and define two operator overloads for the + operator.

```
package myops is
    FUNCTION "+"        (a, b   : BIT_VECTOR)           RETURN BIT_VECTOR
    FUNCTION "+"    (a  : BIT_VECTOR; b : INTEGER)      RETURN BIT_VECTOR
end myops;

use work.conversions.all;
package body myops is
    -- "+"
    --  Add overload for:
    --  In:     two bit_vectors.
    --  Return: bit_vector.
    --
    FUNCTION "+" (a, b   : BIT_VECTOR)          RETURN BIT_VECTOR IS
        VARIABLE s       : BIT_VECTOR (a'RANGE);
        VARIABLE carry   : BIT;
        VARIABLE bi      : integer;       --  Indexes b.
    BEGIN
                carry     := '0';

        FOR i IN  a'LOW TO a'HIGH LOOP
            bi := b'low + (i - a'low);
            s(i)  := (a(i) XOR b(bi)) XOR carry;
            carry := ((a(i) OR b(bi)) AND carry) OR (a(i) AND b(bi));
        END LOOP;

        RETURN  (s);
    END "+";                            -- Two bit_vectors.

    -- "+"
    --  Overload "+" for bit_vector plus integer.
    --  In:     bit_vector and integer.
    --  Return: bit_vector.
    --
    FUNCTION "+" (a  : BIT_VECTOR; b : INTEGER)     RETURN BIT_VECTOR IS
    BEGIN
```

```
        RETURN (a + i2bv(b, a'LENGTH));
    END "+";
end myops;
```

Listing 7-11 Declaring and defining operator overloading functions

This package also makes use of the *conversions* package for the second + function. The expression for the return value makes use of the *i2bv* function found in the *conversions* package. The return expression first converts the bit_vector that is passed in as a parameter to an integer and then adds it to the integer that is passed in as a parameter.

The following line of code from the first overload function is used to ensure that the most significant bit of one vector is added to the most significant bit (not the least significant) of the other vector:

```
bi := b'low + (i - a'low);
```

The addition operator is enclosed in quotation marks to indicate that it is an operator. When the operator is used as in the counter example of *Listing 7-10*, the compiler must search for the addition function with matching operand types for the statement "count <= count + 1;" (where count is a bit_vector and 1 is an integer). If integers are being added, then the native VHDL addition operator is used. In the case of *Listing 7-10*, the second version of the + operator defined in the package *myops* is used.

Listing 7-10 can be written such that the counter uses the *inc_bv* function defined earlier. The line

```
count <= count + 1;
```

is then replaced by

```
count <= inc_bv(count);
```

The first implementation (the one using the + operator) provides more readable code. It is unambiguous without documentation what is accomplished with the statement "count <= count + 1;". What is accomplished with the statement "count <= inc_bv(count);" may be completely obvious to the original designer, but it may not be intuitive for someone else reading the code. The *inc_bv* function may require the reader to delve into the function definition or documentation (comments, if provided). If you use an overloaded operator, then there is no need to maintain and document yet another function, and if your design is transferred to another designer, it will easily be understood. As we'll talk about later in this chapter, using the native operators and vendor-supplied (but standardized) overloaded operators will also likely result in a more efficient implementation of your circuit.

Overloading Functions

Operators are not the only functions that can be overloaded. You can overload any function. Take, for example, the functions declared in the package *mygates* in *Listing 7-12* below.

```
package mygates is
 function and4(a,b,c,d: bit) return bit;
 function and4(a,b,c,d: boolean) return boolean;
 function and4(a,b,c,d: boolean) return bit;
 function and4(a,b,c,d: bit) return boolean;
end mygates;
```

```
use work.conversions.all;
package body mygates is

 FUNCTION and4(a,b,c,d: bit) RETURN BIT IS
    BEGIN
                return (a and b and c and d);
    end and4;

 function and4(a,b,c,d: boolean) return boolean is
    BEGIN
                return (a and b and c and d);
    end and4;

 function and4(a,b,c,d: boolean) return bit is
    BEGIN
                return (bl2bit(a and b and c and d));
    end and4;

 function and4(a,b,c,d: bit) return boolean is
    BEGIN
                return (bit2bl(a and b and c and d));
    end and4;
end mygates;
```

Listing 7-12 Overloading the AND operator for different operand types

Four functions of the same name have been declared and defined. These functions, however, are defined for input parameters and return values of different types. When used in the example below (*Listing 7-13*), the compiler must choose the appropriate function for the function call.

```
entity fourands is port(
        a1,b1,c1,d1:    in boolean;
        q1:             out boolean;
        a2,b2,c2,d2:    in bit;
        q2:             out bit;
        a3,b3,c3,d3:    in bit;
        q3:             out boolean;
        a4,b4,c4,d4:    in boolean;
        q4:             out bit);
end fourands;

use work.mygates.all;
architecture archfourands of fourands is
begin
        q1 <= and4(a1,b1,c1,d1);
        q2 <= and4(a2,b2,c2,d2);
        q3 <= and4(a3,b3,c3,d3);
        q4 <= and4(a4,b4,c4,d4);
end archfourands;
```

Listing 7-13 Using the overloaded AND operators

Listing 7-13 demonstrates that functions can be used as an alternative to certain types of component instantiations (particularly for combinational functions). The listing also demonstrates that the *and4* function can be overloaded to accept different types of input parameters and different types of return values.

It is not necessarily a good idea to overload all operators too handle several different types, because having the compiler find type mismatches may be useful in some cases.

The package that we have been using to overload the + operator for types std_logic and integer is found in Appendix D.

Standard Functions

Fortunately, some standard functions have been established since the first issue of the VHDL standard (1076) in 1987. Standard packages have been defined that include operator overloading for multiple types. This eliminates the need for each tool vendor to provide a proprietary package with unique function names. The standard provides a standard package name and standard function names. VHDL code that makes use of these standard packages is portable from one tool to another, provided the tool supports the standard. For example, the std_logic_1164 package provides a standard datatype system that, because it is supported by multiple tool vendors, enables you to use the data types defined in this package by including the following library and use clause:

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

For every synthesis or simulation tool vendor that supports this package, you will be able to use your code with that tool without any modifications to your code. If each tool vendor required you to use proprietary packages to have access to useful data types or operators, then you would not be able to easily port your code from one system to another.

At the time of this writing, the std_logic_1164 package is the most widely used and accepted standard package. It defines not only a standard datatype system but also standard overloaded operators and type conversion functions for use with that system.

Among other things, the std_logic_1164 package includes common subtypes of std_logic_1164: X01 and X01Z. It overloads the logical operators (and, or, etc.). It also provides some commonly used conversion functions:

```
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;
```

To_bitvector, *To_StdLogicVector*, and *To_StdULogicVector* are all overloaded for different input parameter types.

At the time of this writing, IEEE working group 1076.3 has developed a draft standard for two VHDL synthesis packages: NUMERIC_BIT and NUMERIC_STD. This standard specifies, "Two packages that define vector types for representing signed and unsigned arithmetic values, and that define arithmetic, shift, and type conversion operations on those types." An example of just one of the arithmetic operators that this standard defines is given below. (You won't likely want to read this in detail but rather in general to understand that the operators such as the + operator have been overloaded to accommodate the addition of data objects of many different types.) For a complete draft of the standard consult the World Wide Web (http:// www.vhdl.org).

From NUMERIC_BIT:

```
type UNSIGNED is array (NATURAL range <> ) of BIT;
type SIGNED is array (NATURAL range <> ) of BIT;

-- Id: A.3
function "+" (L, R: UNSIGNED) return UNSIGNED;
-- Result subtype: UNSIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
-- Result: Adds two UNSIGNED vectors that may be of different lengths.

-- Id: A.4
function "+" (L, R: SIGNED) return SIGNED;
-- Result subtype: SIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
-- Result: Adds two SIGNED vectors that may be of different lengths.

-- Id: A.5
function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
-- Result subtype: UNSIGNED(L'LENGTH-1 downto 0).
-- Result: Adds an UNSIGNED vector, L, with a non-negative INTEGER, R.

-- Id: A.6
function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
-- Result subtype: UNSIGNED(R'LENGTH-1 downto 0).
-- Result: Adds a non-negative INTEGER, L, with an UNSIGNED vector, R.

-- Id: A.7
function "+" (L: INTEGER; R: SIGNED) return SIGNED;
-- Result subtype: SIGNED(R'LENGTH-1 downto 0).
-- Result: Adds an INTEGER, L(may be positive or negative), to a SIGNED
--         vector, R.

-- Id: A.8
function "+" (L: SIGNED; R: INTEGER) return SIGNED;
```

```
  -- Result subtype: SIGNED(L'LENGTH-1 downto 0).
  -- Result: Adds a SIGNED vector, L, to an INTEGER, R.

From NUMERIC_STD:
  type UNSIGNED is array (NATURAL range <> ) of STD_LOGIC;
  type SIGNED is array (NATURAL range <> ) of STD_LOGIC;

  -- Id: A.3
  function "+" (L, R: UNSIGNED) return UNSIGNED;
  -- Result subtype: UNSIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
  -- Result: Adds two UNSIGNED vectors that may be of different lengths.

  -- Id: A.4
  function "+" (L, R: SIGNED) return SIGNED;
  -- Result subtype: SIGNED(MAX(L'LENGTH, R'LENGTH)-1 downto 0).
  -- Result: Adds two SIGNED vectors that may be of different lengths.

  -- Id: A.5
  function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
  -- Result subtype: UNSIGNED(L'LENGTH-1 downto 0).
  -- Result: Adds an UNSIGNED vector, L, with a non-negative INTEGER, R.

  -- Id: A.6
  function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
  -- Result subtype: UNSIGNED(R'LENGTH-1 downto 0).
  -- Result: Adds a non-negative INTEGER, L, with an UNSIGNED vector, R.

  -- Id: A.7
  function "+" (L: INTEGER; R: SIGNED) return SIGNED;
  -- Result subtype: SIGNED(R'LENGTH-1 downto 0).
  -- Result: Adds an INTEGER, L(may be positive or negative), to a SIGNED
  --         vector, R.

  -- Id: A.8
  function "+" (L: SIGNED; R: INTEGER) return SIGNED;
  -- Result subtype: SIGNED(L'LENGTH-1 downto 0).
  -- Result: Adds a SIGNED vector, L, to an INTEGER, R.
```

As you can imagine, having standard packages to define these overloaded operators greatly increases the power, flexibility, and portability of VHDL both for synthesis and simulation. It is usually best to use the standard packages rather than create your own, using them from the library specified by the standard or the vendor. Creating your own may provide functionally equivalent code; however, vendors may provide unique implementations that will provide better synthesis.

An important function that is defined in NUMERIC_STD is the function *std_match*. The function is overloaded for several types:

```
  -- Id: M.1
  function STD_MATCH (L, R: STD_ULOGIC) return BOOLEAN;
  -- Result subtype: BOOLEAN
  -- Result: terms compared per STD_LOGIC_1164 intent

  -- Id: M.2
  function STD_MATCH (L, R: UNSIGNED) return BOOLEAN;
```

```
-- Result subtype: BOOLEAN
-- Result: terms compared per STD_LOGIC_1164 intent

-- Id: M.3
function STD_MATCH (L, R: SIGNED) return BOOLEAN;
-- Result subtype: BOOLEAN
-- Result: terms compared per STD_LOGIC_1164 intent

-- Id: M.4
function STD_MATCH (L, R: STD_LOGIC_VECTOR) return BOOLEAN;
-- Result subtype: BOOLEAN
-- Result: terms compared per STD_LOGIC_1164 intent

-- Id: M.5
function STD_MATCH (L, R: STD_ULOGIC_VECTOR) return BOOLEAN;
-- Result subtype: BOOLEAN
-- Result: terms compared per STD_LOGIC_1164 intent
```

The definition for the first function for use with the type std_ulogic is defined below.

```
-- support constants for STD_MATCH:

type BOOLEAN_TABLE is array(STD_ULOGIC, STD_ULOGIC) of BOOLEAN;

constant MATCH_TABLE: BOOLEAN_TABLE := (
    ------------------------------------------------------------------------
    -- U      X      0      1      Z      W      L      H      -
    ------------------------------------------------------------------------
    (FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,  TRUE), -- | U |
    (FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,  TRUE), -- | X |
    (FALSE, FALSE,  TRUE, FALSE, FALSE, FALSE,  TRUE, FALSE,  TRUE), -- | 0 |
    (FALSE, FALSE, FALSE,  TRUE, FALSE, FALSE, FALSE,  TRUE,  TRUE), -- | 1 |
    (FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,  TRUE), -- | Z |
    (FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE,  TRUE), -- | W |
    (FALSE, FALSE,  TRUE, FALSE, FALSE, FALSE,  TRUE, FALSE,  TRUE), -- | L |
    (FALSE, FALSE, FALSE,  TRUE, FALSE, FALSE, FALSE,  TRUE,  TRUE), -- | H |
    ( TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  TRUE,  TRUE)  -- | - |
    );

-- Id: M.1
function STD_MATCH (L, R: STD_ULOGIC) return BOOLEAN is
  variable VALUE: STD_ULOGIC;
begin
  return MATCH_TABLE(L, R);
end STD_MATCH;

-- Id: M.2
function STD_MATCH (L, R: UNSIGNED) return BOOLEAN is
  alias LV: UNSIGNED(1 to L'LENGTH) is L;
  alias RV: UNSIGNED(1 to R'LENGTH) is R;
begin
  if ((L'LENGTH < 1) or (R'LENGTH < 1)) then
    assert NO_WARNING
        report "NUMERIC_STD.STD_MATCH: null detected, returning FALSE"
        severity WARNING;
```

```
      return FALSE;
    end if;
    if LV'LENGTH /= RV'LENGTH then
      assert NO_WARNING
          report "NUMERIC_STD.STD_MATCH: L'LENGTH /= R'LENGTH, returning
FALSE"
          severity WARNING;
      return FALSE;
    else
      for I in LV'LOW to LV'HIGH loop
        if not (MATCH_TABLE(LV(I), RV(I))) then
          return FALSE;
        end if;
      end loop;
      return TRUE;
    end if;
  end STD_MATCH;
```

This function returns a boolean value based on whether two data objects of type std_ulogic match according to don't care conditions. For example, the function as overloaded for std_ulogic_vectors, allows the comparison of `"1--1"` to a signal of type std_ulogic_vector (of width 4) to evaluate to true as long as the first and last elements of the vector are '1,' regardless of the value of the middle two.

The following comparison will evaluate to false, for all values of *a* except `"1--1"`:

```
if a = "1--1" then ...
```

Although '-' represents the don't care value, the = operator cannot be used to identify don't care conditions. The LRM defines the = operator for enumeration types such as std_logic to result in a Boolean evaluation of true only if the left hand-side expression is equivalent to the right-hand side expression. The don't care value, '-', is the 'high value of the std_ulogic type. A comparison of '-' to '0' or to '1' using the = operator evaluates false. The *std_match* function, on other hand, returns true for the comparison of '-' to '0' or '1'. The constant *match_table* above can be used to determine the result of comparing two std_ulogic values using the *std_match* function. Find one of the values you wish to compare in the comment line at the top of the table. Find the other value in the right-hand side column. The result of the comparison is the Boolean value listed at the intersection of the row and column of the two values that you are comparing. To use the function, you will need to include the appropriate USE clause, and write code similar to the following:

```
if std_match(a, "1--1") then ...
```

Procedures

Like functions, procedures are high-level design constructs to compute values or define partial processes that you can use for type conversions, operator overloading, or as an alternative to component instantiation.

Procedures differ from functions in a few ways. To begin with, a procedure can return more than one value. This is accomplished with parameters: If a parameter is declared as mode OUT or INOUT, then the parameter is returned to the actual parameter of the calling procedure. A parameter in a

function, however, can only be of mode IN. Another difference between a procedure and a function is that a procedure can have a WAIT statement whereas a function cannot.

As with functions, all statements within a procedure must be sequential statements. Because of this, procedures cannot declare signals (just as in a process). As with functions, variables can be declared in the declarative region and defined in the definition region.

Procedures and functions are declared and defined in the same way: either in the architecture's declaration region or in a package with the associated definition in the package body.

Now that we've defined the rules for procedures, let's take a look at how to put those rules to use.

```
package myflops is
 procedure dff8(signal d: bit_vector(7 downto 0);
                signal clk: bit;
                signal q: out bit_vector(7 downto 0));
end myflops;

package body myflops is
 procedure dff8(signal d: bit_vector(7 downto 0);
                signal clk: bit;
                signal q: out bit_vector(7 downto 0)) IS
    BEGIN
        wait until clk='1';
                q <= d;
    end dff8;
end myflops;
```

Listing 7-14 A procedure defining eight flip-flops

Listing 7-14 declares the procedure *dff8* in the package *myflops*. A function could not serve the purpose of this subprogram. A procedure is required in order to return more than one argument and to make use of the WAIT statement. A WAIT statement is not allowed in a function.

The procedure parameters were explicitly declared as signals. If the class of data object is not defined and the mode is OUT or INOUT, then the class is defaulted to variable. Using the procedure is quite easy, as demonstrated in *Listing 7-15*.

```
entity flop8 is port(
        clk:            in bit;
        data_in:        in bit_vector(7 downto 0);
        data:           out bit_vector(7 downto 0));
end flop8;

use work.myflops.all;
architecture archflop8 of flop8 is
begin
        dff8(data_in,clk,data);
end archflop8;
```

Listing 7-15 Using the procedure defined in *Listing 7-14*

Overloading Procedures

Procedures may be overloaded in the same way that functions may be overloaded. In *Listing 7-16*, four *dff8* procedure declarations and definitions are added to the *myflops* package. *Listing 7-17* then uses these procedures as appropriate.

```
package myflops is

type boolean_vector is array ( natural range <> ) of boolean;

 procedure dff8(signal d: bit_vector(7 downto 0);
                signal clk: bit;
                signal q: out bit_vector(7 downto 0));

 procedure dff8(signal d: boolean_vector(7 downto 0);
                signal clk: boolean;
                signal q: out boolean_vector(7 downto 0));

 procedure dff8(signal d: bit_vector(7 downto 0);
                signal clk: bit;
                signal q: out boolean_vector(7 downto 0));

 procedure dff8(signal d: boolean_vector(7 downto 0);
                signal clk: boolean;
                signal q: out bit_vector(7 downto 0));
end myflops;

use work.conversions.all;      --required to access bit2bl and bl2bit
package body myflops is
 procedure dff8(signal d: bit_vector(7 downto 0);
                signal clk: bit;
                signal q: out bit_vector(7 downto 0)) is
    Begin
        wait until clk='1';
                q <= d;
    end dff8;

 procedure dff8(signal d: boolean_vector(7 downto 0);
                signal clk: boolean;
                signal q: out boolean_vector(7 downto 0)) is
        begin
                wait until clk;
                        q <= d;
        end dff8;

 procedure dff8(signal d: bit_vector(7 downto 0);
                signal clk: bit;
                signal q: out boolean_vector(7 downto 0)) is
        begin
                wait until clk='1';
                for i in 7 downto 0 loop
                        q(i) <= bit2bl(d(i));
                end loop;
        end dff8;
```

```
 procedure dff8(signal d: boolean_vector(7 downto 0);
                signal clk: boolean;
                signal q: out bit_vector(7 downto 0)) is
        begin
                wait until clk;
                for i in 7 downto 0 loop
                        q(i) <= bl2bit(d(i));
                end loop;
        end dff8;

end myflops;
```

Listing 7-16 A package that overloads the *dff8* procedure for different parameter types

As you would expect, the additional *dff8* procedure declarations and definitions make use of different parameter types. Additionally, because we reference the *bl2bit* function in this package, we must include the *conversions* package by including the use clause.

```
use work.myflops.all;
entity flop8 is port(
        clk1:           in bit;
        data_in1:       in bit_vector(7 downto 0);
        data1:          out bit_vector(7 downto 0);
        clk2:           in boolean;
        data_in2:       in boolean_vector(7 downto 0);
        data2:          out boolean_vector(7 downto 0);
        clk3:           in bit;
        data_in3:       in bit_vector(7 downto 0);
        data3:          out boolean_vector(7 downto 0);
        clk4:           in boolean;
        data_in4:       in boolean_vector(7 downto 0);
        data4:          out bit_vector(7 downto 0));
end flop8;

architecture archflop8 of flop8 is
begin
        dff8(data_in1,clk1,data1);
        dff8(data_in2,clk2,data2);
        dff8(data_in3,clk3,data3);
        dff8(data_in4,clk4,data4);
end archflop8;
```

Listing 7-17 A design that uses overloaded procedures with different port types

There aren't any surprises in *Listing 7-17*. The use clause had to be moved from just above the architecture to just above the entity in order to make the type Boolean_vector available for use in the entity port declarations. The *dff8* procedural calls make use of different data types, but the overloaded operators enable the correct procedure to be called.

About Subprograms

Subprograms (functions and procedures) can greatly add to the readability of code, making VHDL both powerful and flexible. Use subprograms carefully, however, to ensure that the circuit you are describing will be implemented in such a way as to achieve your design objectives (performance and capacity).

Use vendor-supplied standard overloaded operators before defining your own. Often, these operators come in the form of standard packages such as the std_logic_1164, numeric_bit, or numeric_std packages. You can create your own overloaded operators, and the implementation will be logically correct. Nonetheless, synthesis and simulation tool vendors may have optimized package bodies for use with their tool.

Rest assured, however, that there are many more uses of subprograms—some we have explored, others we leave for you to discover—in which the function has not already been standardized or defined.

Exercises

1. What is the use and advantage of operator overloading? Are there any disadvantages? Justify.

2.Write function definitions for the following:

FUNCTION To_StdLogicVector (b : BIT_VECTOR) RETURN std_logic_vector;

FUNCTION To_bitvector (s : std_logic_vector ; xmap : BIT := '0') RETURN BIT_VECTOR;

3. Overload the following functions for the std_logic type:

a) and

b) or

c) - (unary negate)

4. Write a function that (a) defines a Boolean_vector as an array of Boolean and (b) includes two functions: a bit_vector to Boolean vector conversion function and a Boolean vector to bit_vector conversion function.

5. Overload the + operator for the addition of Boolean vector and integer.

6. Create an entity/architecture pair for a 12-bit counter whose ports are Boolean_vector. Use the overloaded operator created in Exercise 5 to perform the addition.

7. Create a package and package body to declare and define four XOR8 functions for inputs and return type combinations of bit-bit, bit-Boolean, Boolean-Boolean, and Boolean-bit.

8. Create an entity/architecture pair that uses each of the four XOR8 functions of Exercise 7.

9. Create a package and package body to declare and define four *tff16* (16-bit-wide T-flip-flops) procedures for input and output combinations of bit-bit, bit-Boolean, Boolean-Boolean, and Boolean-bit.

10. Create an entity/architecture pair that uses each of the four procedures in Exercise 9.

11. Create a Procedure for decrementing bit_vectors. Also, create an underflow output for the procedure.

12. What are the advantages of using a procedure over instantiating a component?

13. Rewrite the *bv2i* function shown in *Listing 7-2* to interpret the value on the left of the bit-vector as the MSB.

14. Write the procedure to perform a 16-bit even parity check. Synthesize this design into a CPLD.

15. Write a function to replace the synchronizer component of the network repeater design.

8 Synthesis to Final Design Implementation

Up to this point, we have discussed how to write VHDL code to create device-independent designs. In this chapter, we will explore issues of writing code for specific architectures. To do this, we will examine the processes of synthesis and fitting (place and route for FPGAs). We will show that the processes of synthesis and fitting produce the best results (in terms of resource utilization, achievable performance, and whether design objectives are met) when these processes are tightly coupled with the target architecture. To demonstrate these processes and their relationship to device architectures, we will use case studies of a CPLD architecture and an FPGA architecture.

We've already looked at a few synthesis issues such as creating memory elements, implicit memory, the effects of different state machine implementations, and writing efficient code. What we haven't yet discussed is how VHDL code will be synthesized and fitted to a limited resource. Three common mistakes that designers make when beginning to write VHDL code for programmable logic are that they forget that (1) the PLD, CPLD, or FPGA has limited resources, (2) the resources have specific features, and (3) not every design will fit in every architecture.

VHDL provides powerful language constructs that enable you to describe a large amount of logic quickly and easily, so you will have to choose a programmable logic device with the appropriate capacity and feature set. Choosing an appropriate device is made easier with VHDL because it allows you to benchmark designs in different devices: You can use the same code to target multiple architectures or devices. You can then easily compare the implementation results of the different architectures and choose the device that best meets design objectives. With that being said, however, it's helpful for you to have a good understanding of the process of synthesis and fitting so that you will understand the resource and feature set requirements, as well as the achievable performance of your design in various architectures.

Following is a simple design example that when synthesized requires specific device resources. It shows that even with a simple design, an appropriate device must be selected. In our discussion back in chapter xxx about creating registered elements using different templates for asynchronous reset and preset, we didn't discuss how a device's available resources affect the final realization of the described logic. Take, for instance, the code of *Listing 8-1*:

```
library ieee;
use ieee.std_logic_1164.all;
entity counter is port(
   clk, reset: in std_logic;
   count:      buffer std_logic_vector(3 downto 0));
end counter;

use work.std_math.all;
architecture archcounter of counter is
begin
upcount: process (clk, reset)
   begin
      if reset = '1' then
         count <= ("3261");
      elsif (clk'event and clk= '1') then
            count <= count + 1;
      end if;
   end process upcount;
```

```
end archcounter;
```

Listing 8-1 4-bit counter that resets to 1010

If we have a theoretical device consisting of one logic block as shown in *Figure 8-1* , then at first glance you may think that the counter in *Listing 8-1* cannot fit into this device. The code shows that this counter is asynchronously reset to 1010 when *rst* is asserted, but all registers in this device share a common asynchronous reset making it impossible for four registers to be asynchronously reset to 1010. That is, the *rst* signal cannot simply be tied to the asynchronous reset line because, as the figure illustrates, attaching *rst* to the asynchronous reset line of the logic block would cause the counter to reset to 0000 not 1010. *Figure 8-2* shows the implementation of a counter with a common reset line that resets the counter to 0000. But the design *can* still fit: The *rst* signal can be attached to

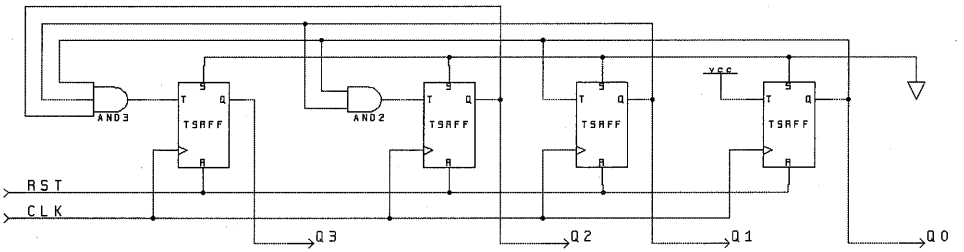


Figure 8-2 Counter that resets to 0000

the asynchronous reset line *if* the second and fourth bits of the count registers are implemented such that the outputs of these registers are inverted before the device pins. The registers would be reset to 0000, but the device pins would indicate 1010. If you read the fine pFirint of *Figure 8-1,* you will see that either polarity of each register may be propagated to a device pin. If an inverter is introduced between the *Q3* and *Q1* registers and their associated device pins, then the logic that causes these flip-flops to toggle (in the case of T-type flip-flops) must be modified so that the count at the output is sequential. The state transition table below illustrates the necessary values for the registers, and *Figure 8-3* illustrates the logic required for implementation.

count	$Q_3'Q_2Q_1'Q_0$
0000	1010
0001	1011
0010	1000
0011	1001
0100	1110
0101	1111

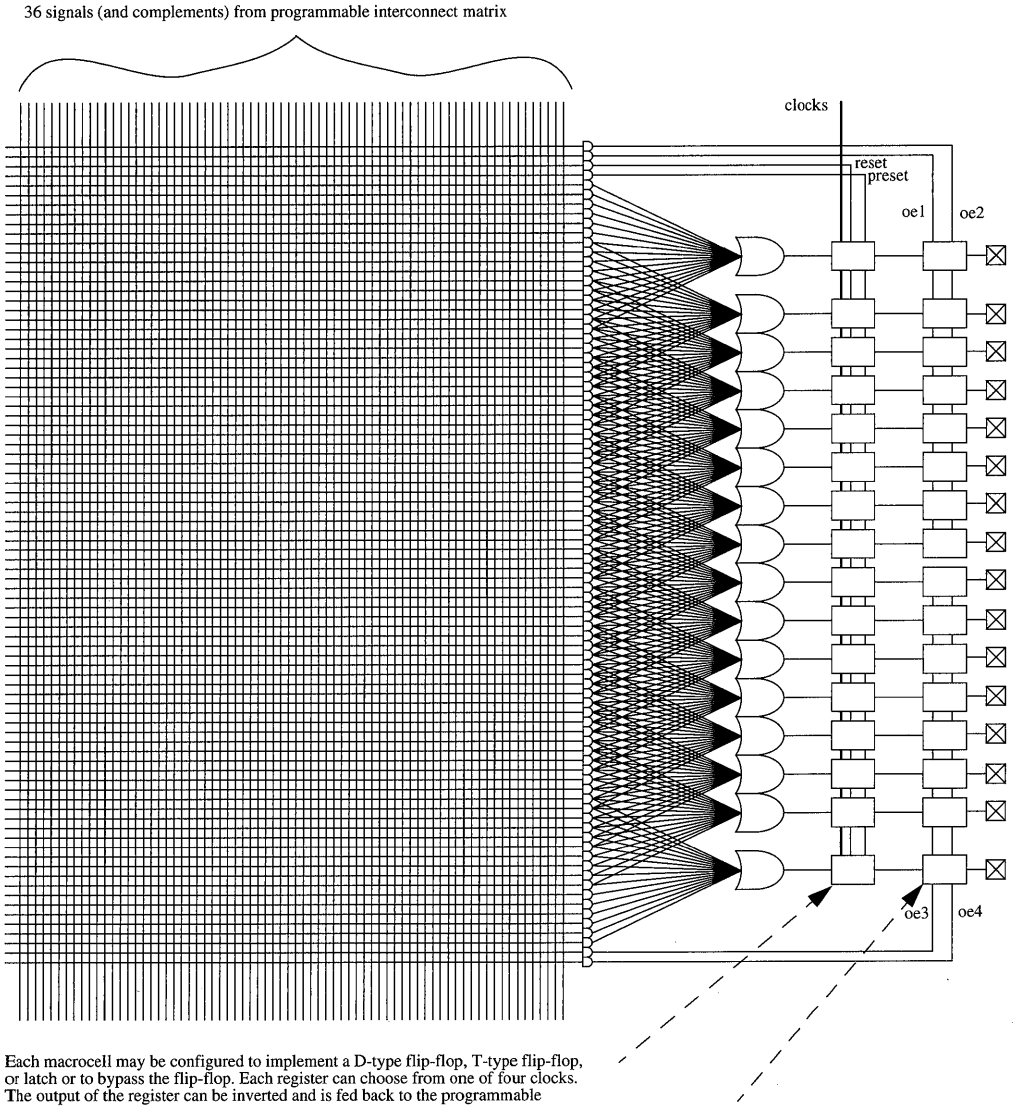


Figure 8-1 Logic block of the FLASH370 family of CPLDs

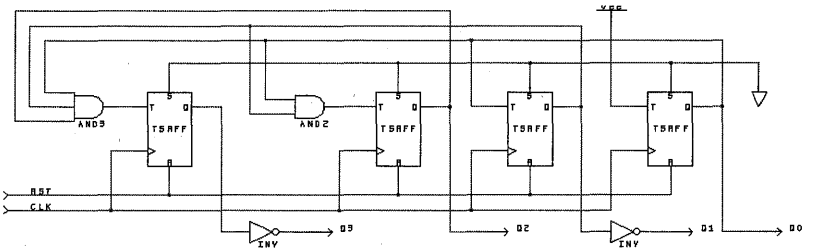


Figure 8-3 Counter that resets to "1010" at pins

count	$Q_3'Q_2Q_1'Q_0$
0110	1100
0111	1101
1000	0010
1001	0011
1010	0000
1011	0001
1100	0110
1101	0111
1110	0100
1111	0101

If we use a device that has two of the logic blocks shown in *Figure 8-2*, such as the CY7C371, then the design can be synthesized and fit by (1) the method described above, (2) working around the preset/reset conditions of the logic blocks (that is, partitioning the design into the two logic blocks by placing the first and third counter bits in one logic block—with a common reset—and placing the second and fourth counter bits in the other logic block—with a common preset), or (3) a combination of the two methods. The method that is chosen will affect the availability of resources for additional logic that may be implemented in the device. For example, with the second method, all registers in one logic block must share the same reset, and all registers in the other logic block must share the same preset. Logic that is added to this device must work around the placement of existing logic resources. All registers must conform to make use of the same preset and reset as the counter does. In most cases, the first method is preferred: All registers can be placed in the same logic block utilizing the same reset signal, leaving the second logic block's reset and preset signals available for additional logic.

This example demonstrates that a device architecture does affect how a design will finally be realized in the device. It also demonstrates that not every design can fit in every architecture. Fortunately,

state-of-the-art synthesis and fitting algorithms can try many options in a short time, finding an efficient implementation in most cases.

Having examined a simple example, we will move on to explore the task of synthesis and fitting for CPLDs and FPGAs. With an understanding of how VHDL designs are realized in devices, you will be equipped to optimize your designs for resource utilization and performance requirements. Your understanding will enable you to squeeze out the last macrocell, logic cell, or nanosecond from a design because you will be able to write efficient VHDL code and provide the human creativity that no synthesis or fitting tool can.

Synthesis and Fitting

Besides being tightly coupled to a device architecture, the synthesis and fitting processes must work closely together. Although these processes are two separate tasks from a software point of view, they are in effect on a continuum. Where one stops, the other must pick up. Whereas synthesis is the process of creating logic equations (or netlists) from the VHDL code, fitting is the process of taking those logic equations and fitting them into the programmable logic device. Device-specific optimization can occur in synthesis, fitting, or both (*Figure 8-4*).

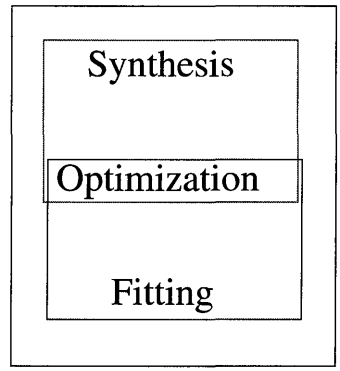


Figure 8-4 Optimization can occur in synthesis, fitting, or both. Ideally, the two processes are on a continuum.

The synthesis process can pass to the fitter a design's logic equations in a way that indicates precisely which resources should be used. Alternatively, the synthesis process can pass non-optimized equations to the fitter, leaving the optimization task to the fitter. From your (the designer's) point of view, you don't care where the device-specific optimization takes place, just as long as it does. What is important is that the synthesis and fitting processes interface well: that the fitter receives information from the synthesis process in such a way that enables the fitter to produce the best possible implementation. If the fitter does not perform any optimization, then the synthesis process should pass logic equations and information in such a way that the fitter simply places the logic. However, if the fitter does provide optimization, then information should be passed from the synthesis process in a way that does not restrict the fitter from performing the appropriate optimization.

CPLDs: A Case Study

In this section, we will examine the task of synthesizing and fitting several designs to the FLASH370 architecture shown in *Figure 8-5*. This examination will expose you to the scope of the task of synthesizing and fitting designs to CPLDs. With this information, you will be equipped to get the most from your VHDL designs for CPLD architectures, and you will come to appreciate the difficulty in designing a fitter. For a review of CPLD architectures, refer to the Programmable Logic Primer chapter.

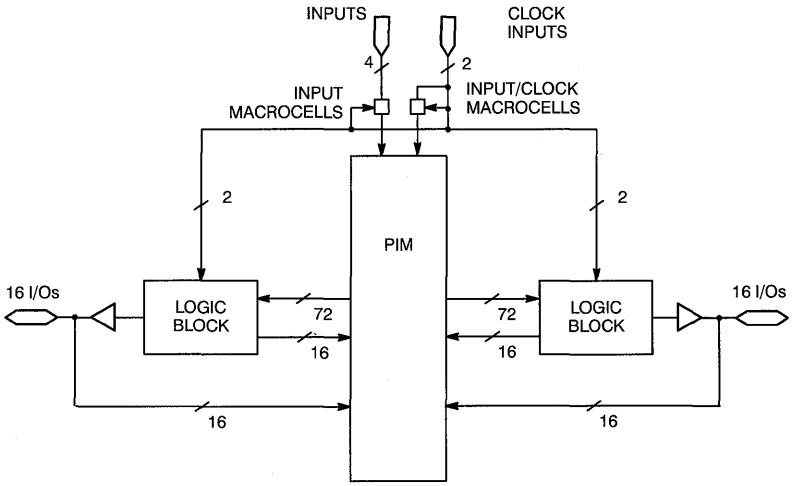


Figure 8-5 Block diagram of the 32-macrocell member of the FLASH370 family of CPLDs

The FLASH370 architecture consists of logic blocks that communicate with each other through a programmable interconnect matrix (PIM). All signals (except the clocks) route through the PIM. Each logic block can be configured to receive up to 36 inputs from the PIM. These signals and their complements are used to produce up to 86 product terms of which 80 are allocated to 16 macrocells and 6 are allocated to output enables, asynchronous preset, and asynchronous reset. All macrocell and I/O signals feed back to the PIM.

Each I/O macrocell (*Figure 2-20*) can sum from 0 to 16 product terms; product terms can be individually allocated from one macrocell to the next or shared among multiple macrocells. The sum of product terms at the macrocell can be used as a combinational output, combinational *buried node* (a node that is not an output of the device), registered output, registered buried node, latched output, or latched buried node. If the macrocell is configured for registering, the register can be a D-type register or T-type register with asynchronous reset and set lines. The output of the macrocell can then be fed back to the programmable interconnect. The macrocell output can also drive an I/O cell (I/O cells may be configured for input, output, or bidirectional signals), in which case the output may be inverted. Each macrocell can be configured to use one of several clocks that are allocated to the logic block.

Some of the pins function as dedicated inputs (i.e., these cells cannot be configured as output or bidirectional cells) or as either an input or clock pin. The input/clock macrocells are shown in *Figure* xxx {fig # from primer}. Those pins without clocking capabilities do not contain the clock logic in the upper half of the diagram. An input can feed to the PIM as a combinational input, latched input, a registered input, or twice-registered input. Twice-registered inputs are sometimes used by designers to increase the MTBF (mean time between failures) for asynchronous inputs that can cause metastable events. A registered input cannot be clocked by itself (doing so would surely decrease the MTBF!), but one of the other clock inputs can be configured as the clock for the two registers. Input/clock pins used for clocking feed clock multiplexers of the logic blocks. Each logic block can choose to have either the clock signal itself or its inverse (effectively allowing clocking on the falling edge of the clock).

We will use this architecture for the basis of the following discussion on synthesizing and fitting designs.

Synthesizing and Fitting Designs for the 370 Architecture

As the example at the beginning of the chapter demonstrated, fitting designs in one logic block presents a problem that may be difficult, but for which there are a manageable number of possibiliteis—designs either fit or they don't. Signals are assigned to macrocells based upon the number of product terms, how those product terms can be steered or shared, output enables, reset and preset conditions, and clocking requirements. With only one logic block, there are only a few ways that some designs can fit. With multiple logic blocks, designs can often fit in many different ways, but the design must be carefully partitioned among the logic blocks. Grouping signals in a logic block (partitioning) based on one condition may affect how another resource may be used. In the first part of the chapter, we looked at fitting the counter of *Listing 8-1* into a device with two logic blocks. We decided that the second method we considered restricted the way in which additional logic could fit in the device. Partitioning must satisfy the limits imposed by (1) the number of macrocells per logic block, (2) the number of product terms per logic block, (3) the preset/reset combinations for the logic within the logic block, (4) the output enable requirements for the logic within the logic block, (5) the number of inputs from the programmable interconnect to the logic block, and (6) clocking requirements for each logic block. Partitioning a design into groups of signals assigned to separate logic blocks must be based on a balance of these considerations; otherwise, the capacity of the device will be unnecessarily restricted.

Partitioning logic based on the number of macrocells is easily accomplished (each logic block can hold 16 macrocells), except when multiple passes are required to generate a signal (for now, we will only consider signals that are to be registered). Whether a signal requires multiple passes depends upon the polarity of the signal that is registered and the type of flip-flop used. Registering the complement of a signal or using a T-type flip-flop versus a D-type flip-flop can save product terms. However, using the complement of a signal may affect the number of required preset/reset combinations. If none of the signals require more than one level of logic even with the worst-case polarity selection, then as long as the total number of product terms required is 80 or fewer (and the product terms are distributed over the macrocells, see chapter 2, "Programmable Logic Primer"), then the macrocell and product term limits are met. If the worst-case polarity selection requires more than one level for a signal and choosing the opposite polarity upsets the preset/reset combinations, then a decision must be made based on the availability of resources: (1) Can the signal be moved to another logic block (or traded with a signal in another logic block) without upsetting that logic-block's preset/reset combinations, (2) can the polarity of every signal be reversed in order to maintain consistent preset/reset combinations while meeting the conditions listed above, or (3) can

the signal fit in the present logic block while taking multiple passes? We will explore a few of the more difficult partitioning choices next.

Satisfying Preset/Reset Conditions

All CPLDs have particular feature sets that affect the way in which designs may be implemented. In the next several examples, we look at the preset/reset resources of the CY7C371 CPLD and discuss how this resource can and cannot be used. The one preset product term and one reset product term (see top right of *Figure 8-1*) of each logic block usually provide enough flexibility because resets and presets tend to be global.

Listing 8-2 shows two 16-bit loadable counters that asynchronously reset to two different values based on the signals *rsta* and *rstb*. How can this design be partitioned into a CY7C371? Intuitively, you may want to partition the counters into separate logic blocks. In fact, this is the only way that these two counters can be partitioned into this device. *Rsta* must be used as an asynchronous reset for *cnta*, and therefore resets all of the registers within the logic block. The resetting of *cntb* is not controlled by *rsta*, and therefore all *cntb* registers must be in a separate logic block from the *cnta* registers.

```
library ieee;
use ieee.std_logic_1164.all;
entity counter is port(
        clk, rsta,rstb,lda,ldb: in std_logic;
        cnta, cntb:             buffer std_logic_vector(15 downto 0));
end counter;

use work.std_math.all;
architecture archcounter of counter is
begin

upcnta: process (clk, rsta)
        begin
                if rsta = '1' then
                        cnta <= x"3261";
                elsif (clk'event and clk= '1') then
                        if lda = '1' then
                                cnta <= cntb;
                        else
                                cnta <= cnta + 1;
                        end if;
                end if;
        end process upcnta;

upcntb: process (clk, rstb)
        begin
                if rstb = '1' then
                        cntb <= x"5732";
                elsif (clk'event and clk= '1') then
                        if ldb = '1' then
                                cntb <= cntb;
                        else
                                cntb <= cntb + 1;
                        end if;
```

```
                end if;
        end process upcntb;
end archcounter;
```

Listing 8-2 Two 16-bit loadable counters

The two counters do not need to be defined in separate processes. They are described this way for readability.

Read through *Listing 8-3* and determine how this design can fit into a CY7C371.

```
library ieee;
use ieee.std_logic_1164.all;
entity counter is port(
        clk, rsta, rstb, rstc:  in std_logic;
        cnta, cntb, cntc:       inout std_logic_vector(7 downto 0));
end counter;

use work.std_math.all;
architecture archcounter of counter is
begin

upcnta: process (clk, rsta)
        begin
                if rsta = '1' then
                        cnta <= (others => '0');
                elsif (clk'event and clk= '1') then
                        cnta <= cnta + 1;
                end if;
        end process upcnta;

upcntb: process (clk, rstb)
        begin
                if rstb = '1' then
                        cntb <= (others => '0');
                elsif (clk'event and clk= '1') then
                        cntb <= cntb + 1;
                end if;
        end process upcntb;

upcntc: process (clk, rstc)
        begin
                if rstc = '1' then
                        cntc <= (others => '0');
                elsif (clk'event and clk= '1') then
                        cntc <= cntc + 1;
                end if;
        end process upcntc;

end archcounter;
```

Listing 8-3 Three 8-bit counters with separate asynchronous resets

Even though there are enough macrocells to store all the register values, this design will not fit into two logic blocks because three asynchronous resets are required. Each asynchronous reset must reset all registers within a logic block. Therefore, this design requires three logic blocks and will not fit in a CY7C371. Modifying the design as in *Listing 8-4* below to use two synchronous resets permits the design to easily fit.

```
architecture archcounter of counter is
begin

upcnta: process (clk, rsta)
    begin
        if rsta = '1' then
            cnta <= (others => '0');
        elsif (clk'event and clk= '1') then
            cnta <= cnta + 1;
        end if;
    end process upcnta;

upcntb: process (clk, rstb)
    begin
        if (clk'event and clk= '1') then
            if rstb = '1' then
                cntb <= (others => '0');
            else
                cntb <= cntb + 1;
            end if;
        end if;
    end process upcntb;

upcntc: process (clk, rstc)
    begin
        if (clk'event and clk= '1') then
            if rstc = '1' then
                cntc <= (others => '0');
            else
                cntc <= cntc + 1;
            end if;
        end if;
    end process upcntc;

end archcounter;
```

Listing 8-4 Three 8-bit counters; one with asynchronous reset, two with synchronous reset

Listing 8-4 uses a synchronous reset for the second and third counter. The synchronous resets will use additional product terms but eliminate the need for three separate logic blocks.

The logic block of *Figure 8-1* indicates that the asynchronous reset and preset lines are product terms. The design in *Listing 8-5* will make use of a product term reset.

```
library ieee;
```

```
use ieee.std_logic_1164.all;
entity counter is port(
        clk, rsta, rstb:        in std_logic;
        cnta, cntb:     buffer std_logic_vector(15 downto 0));
end counter;

use work.std_math.all;
architecture archcounter of counter is
begin

upcnta: process (clk, rsta,rstb)
        begin
                if (rsta = '1' and rstb = '1') then
                        cnta <= x"0001";
                elsif (clk'event and clk= '1') then
                        cnta <= cnta + 1;
                end if;
        end process upcnta;

upcntb: process (clk, rsta, rstb)
        begin
                if (rsta = '1' and rstb = '1') then
                        cntb <= x"0002";
                elsif (clk'event and clk= '1') then
                        cntb <= cntb + 1;
                end if;
        end process upcntb;
end archcounter;
```

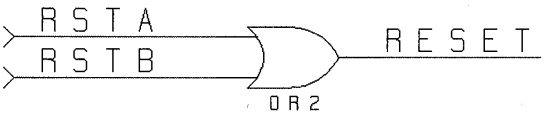
Listing 8-5 A design with product term asynchronous reset

The asynchronous set and reset product terms have polarity control, which allows the registers to be set or reset based on an AND expression (product term) or an OR expression (sum term). If the reset signal is the sum of two literals, such as *rsta OR rstb*, then the product term *((rsta)' AND (rstb)')'* may be used. Thus, the reset logic of the code in *Listing 8-6* may be implemented as in *Figure 8-6*.

```
library ieee;
use ieee.std_logic_1164.all;
entity counter is port(
        clk, rsta, rstb:        in std_logic;
        cnta, cntb:     buffer std_logic_vector(15 downto 0));
end counter;

use work.std_math.all;
architecture archcounter of counter is
begin

upcnta: process (clk, rsta, rstb)
        begin
                if (rsta = '1' or rstb = '1') then
                        cnta <= x"0001";
                elsif (clk'event and clk= '1') then
                        cnta <= cnta + 1;
```

```
                end if;
        end process upcnta;

upcntb: process (clk, rsta, rstb)
        begin
                if (rsta = '1' or rstb = '1') then
                        cntb <= x"0002";
                elsif (clk'event and clk= '1') then
                        cntb <= cntb + 1;
                end if;
        end process upcntb;
end archcounter;
```

Listing 8-6 A design with an OR term reset



The two input OR functions can be implemented in one pass through the product term array as a NAND function:

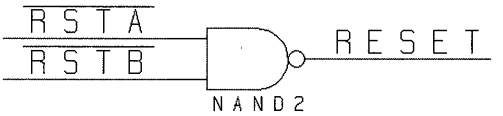


Figure 8-6 A design that uses an OR term for resetting

Forcing Signals to Macrocells

If the expression for the reset or preset requires more than one product term or a sum term, then an additional macrocell must be used to implement the required reset logic. That is, the reset and preset terms provide only enough logic for one AND or OR expression; reset logic that is more complex must make use of a macrocell for which an incremental delay on the order of several nanoseconds is incurred. *Listing 8-7* demonstrates such an example.

```
library ieee;
use ieee.std_logic_1164.all;
entity counter is port(
        clk, r1, r2, r3:        in std_logic;
        cnta:     buffer std_logic_vector(15 downto 0));
end counter;

use work.std_math.all;
architecture archcounter of counter is
      signal reset: std_logic;
begin
```

```
reset <= r1 and (r2 or r3);
upcnta: process (clk, reset)
        begin
                if (reset = '1') then
                        cnta <= x"0001";
                elsif (clk'event and clk= '1') then
                        cnta <= cnta + 1;
                end if;
        end process upcnta;
end archcounter;
```

Listing 8-7 Forcing the reset equation to a macrocell

Listing 8-7 illustrates a design for which the interface between the synthesis and fitting processes must be well defined. If the fitter process accepts the equation for *reset* as is, then it must determine how to break this equation up using a macrocell. This requires that (1) synthesis merely pass the equation to the fitter, and (2) optimization occur in the fitter. Otherwise, the synthesis tool will be responsible for passing logic equations to the fitter indicating that the equation for *reset* must use a macrocell. *Figure 8-7* shows that the reset must be implemented with more than one AND or OR term.

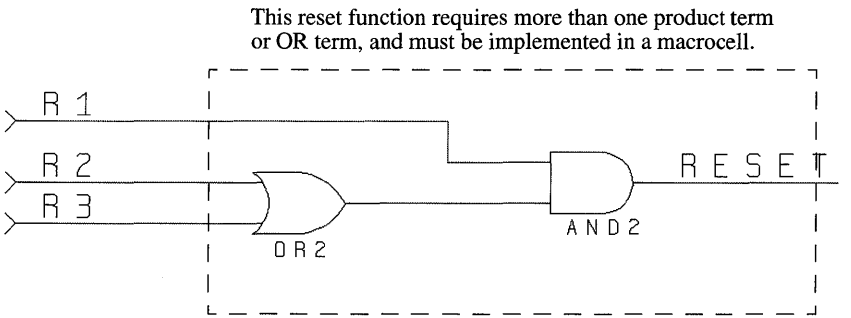


Figure 8-7 Forcing the reset equation to a macrocell

If neither tool performs the optimization necessary to place the reset equation in a macrocell, then you will need to intervene, indicating that *reset* should be forced to a macrocell (forced to a node). Synthesis tools differ in how to accomplish this. Oftentimes vendor attributes, as shown in *Listing 8-8*, are used.

```
library ieee;
use ieee.std_logic_1164.all;
entity counter is port(
        clk, r1, r2, r3:        in std_logic;
        cnta:     buffer std_logic_vector(15 downto 0));
end counter;
```

```
use work.std_math.all;
architecture archcounter of counter is
      signal reset: std_logic;
        attribute synthesis_off of reset:signal is true;
begin

reset <= r1 and (r2 or r3);
upcnta: process (clk, reset)
        begin
                if (reset = '1') then
                        cnta <= x"0001";
                elsif (clk'event and clk= '1') then
                        cnta <= cnta + 1;
                end if;
        end process upcnta;
end archcounter;
```

Listing 8-8 Using an attribute to force logic to a macrocell

Most designs require few resets, and these resets are usually global. For most designs, the preset/reset flexibility of the FLASH370 family of devices is more than sufficient and does not present a partitioning problem.

Preassigning signals to device pins

While concurrent engineering ideally enables multiple efforts to succeed in parallel, reducing costs and time-to-market, it can lead to rework, additional costs, and missed schedules if all critical requirements are not considered before work begins. Taking a careful look at the entire problem before beginning parallel efforts can avoid unnecessary rework.

One example of concurrent engineering is with board-level designs and programmable logic. Ideally, you would like to preassign signals for a programmable logic device before you actually design the logic for the device. If you already know which signals are inputs and outputs to the device, then you may want to assign those signals to actual pin numbers so that you can begin your board layout to manufacture a printed circuit board. At the same time that work begins on the board, you would like to start the design for your programmable logic device. Although these parallel efforts make sense at first glance, the examples in the previous section illustrate this pivotal point: assigning a *pinout* rather than allowing the fitter to choose how to place logic may remove the possibility of a design fit. Obviously, if the counter registers for *cnta* and *cntb* in *Listing 8-2* are preassigned to pins associated with the same logic block, then this design may fail to fit. The design *could* fit only if macrocells are available and a second pass for the outputs (resulting in a clock-to-output of t_{CO2}) is acceptable. This fitting solution is inefficient but may get you out of trouble if you are in a bind.

The routing and product term allocation schemes of CPLDs are another reason not to preassign signals to pins without a clear understanding of how these schemes work. Most CPLDs have *multiplexer-based interconnect* or *routing pools* (see chapter xxx, "Programmable Logic Primer") that route I/O signals and macrocell feedbacks to the logic blocks. These routing schemes are typically not functionally equivalent to cross-point switches: whether a signal can route to a particular logic block depends on which of the other signals must route to that logic block. That is, it may not always be possible to route a *particular set* of signals to a logic block. The larger a set is

with respect to the total number of logic block inputs from the programmable interconnect, the less chance that the set can route to the logic block. If you specify a pin assignment, you are in effect specifying a set of signals that must route to that logic block. Oftentimes, however, you may not know how many or which signals will be required to produce the logic for the signals for which you are specifying a pinout. The following paragraph clarifies this point.

Figure 8-8 shows the interconnect schemes for two different CPLDs that have 36 inputs to a logic

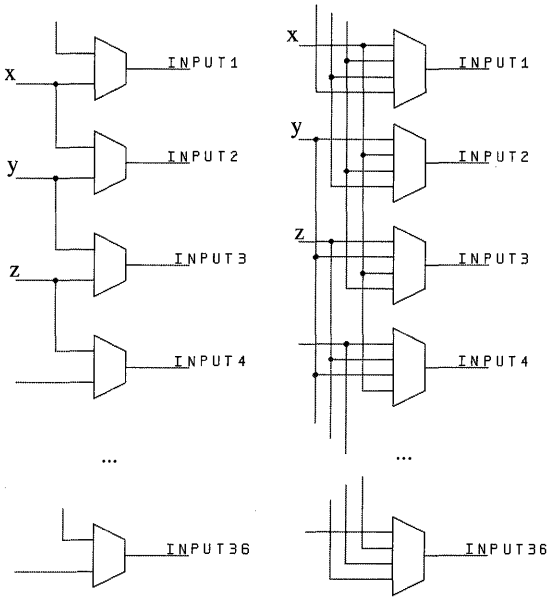


Figure 8-8 Interconnect schemes

block (one scheme is shown on the left, the other on the right). The diagram indicates how signals are routed into a logic block. Both devices use multiplexer-based interconnect schemes, but the width of the multiplexers differ for the two devices. The scheme on the left uses 36 two-to-one multiplexers; the scheme on the right uses 36 four-to-one multiplexers. Suppose *x, y,* and *z* must all route to a logic block. With the scheme on the left, *x, y,* and *z* are each an input to two multiplexers. That is, there are two paths or "chances" to route to a logic block, compared with four chances for the scheme on the right. With the scheme on the left, if *x* must route on input2 and *z* on input3, then there is no path for *y* in this design. With the scheme shown on the right, *y* can still route as input4 or input5 (not shown). Why don't all silicon vendors use wider multiplexers? Because wide multiplexers and more wires nearly always result in a larger die area that may also result in greater manufacturing costs for the device vendor, and because using wider multiplexers results ins a performance degradation.

Routing schemes can be differentiated not only by the width of the multiplexers but also in terms of *routability*, or the capability to route signals through the programmable interconnect and into a logic

block. We will investigate routability, aided by. *Figure 8-9*. Device A is different from the rest; in it,

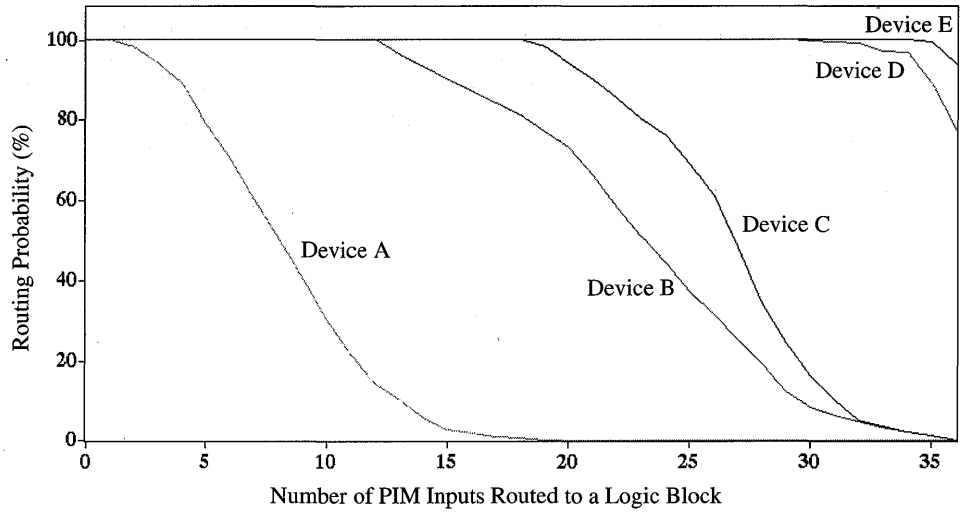


Figure 8-9 Routability of CPLDs

there are 22 inputs from the programmable interconnect to the logic block, whereas there are 36 inputs for each of the other devices. Because Device A has only 22 inputs to a logic block, it follows that the probability of routing any set of more than 22 signals into a logic block is zero. (Corollary: Functions of greater than 22 signals will require more than one level of logic in such devices.) If we assume that the devices compared in *Figure 8-9* use multiplexer-based interconnect schemes, then we can conclude that Devices D and E have wider multiplexers than Devices B and C. This accounts for the fact that a larger number of unique sets of signals (signals from I/Os or macrocells) can route to a logic block in Devices E and D. For example, if 35 signals are required to route to a logic block, upward of 90% of the unique sets of 35 signals can route in Devices D and E, whereas less than 5% of the unique sets of 35 signals can route in Devices B and C. This does not mean that less than 5% of designs requiring 35 signals per logic block will fit in devices B and C, but it does mean that if you specify which sets of 35 signals must be routed to the logic block (by specifying a pinout, for example), then there is less than a 5% chance of finding a fit. Because assigning a pinout defines a set of signals that must route to a logic block, designing a board layout prior to finishing the PLD design for devices B and C would be ill-advised. An additional design trap to be aware of is expecting design changes to fit with the same pinout in devices with a low routing probability for the required number of inputs to a logic block. Often, a design change is required after discovering a bug in testing or QA. Obviously, you'd like the design to fit with the same pinout so that you won't need to modify the board. If the design change is small, then the inputs to the logic block may not change at all. However, if a different set of signals is required and the probability of routing signals to the logic block is low, then you may have to make changes to the board.

The design in *Listing 8-9* consists of two 16-bit loadable, enableable counters that count by one or by two depending on the value of signal *by1*. If you attempt to fit this design into the CY7C371, you find that the two counters can be partitioned into logic blocks in only one way—each counter must

be in a separate logic block. This is because each counter requires all 36 inputs to the logic block: 16 for the counter bits (the present count value is required to determine the next count value), 16 for the load bits, and one each for *rst, by1, ld*, and the enable. Counter bits from one counter cannot be placed in the logic block of the other counter because that requires the other counter's enable signal (either *ena* or *enb*) to also route to that logic block. This design example demonstrates that although you are free to specify the pinout for the counters within each logic block (the CY7C371 is Device E above, so defining a set of signals to route to a logic block rarely presents a problem), you are not able to specify a pinout that requires different bits of a counter to be in separate logic blocks. Additionally, if you use a different 32-macrocell CPLD that is divided into two logic blocks that has fewer than 36 inputs, then the design does not fit at all.

```
library ieee;
use ieee.std_logic_1164.all;
entity counter is port(
    clk, rsta,rstb:     in std_logic;
    ld, en, by1:        in std_logic;
    cnta, cntb:     buffer std_logic_vector(15 downto 0));
end counter;

use work.std_math.all;
architecture archcounter of counter is
begin

upcnta: process (clk, rsta)
    begin
        if rsta = '1' then
            cnta <= x"0000";
        elsif (clk'event and clk= '1') then
            if ld = '1' then
                cnta <= cntb;
            elsif en = '1' then
                if by1 = '1' then
                    cnta <= cnta + 1;
                else
                    cnta <= cnta + 2;
                end if;
            else
                cnta <= cnta;
            end if;
        end if;
    end process upcnta;

upcntb: process (clk, rstb)
    begin
        if rstb = '1' then
            cntb <= x"0000";
        elsif (clk'event and clk= '1') then
            if ld = '1' then
                cntb <= cntb;
            elsif en = '1' then
                if by1 = '1' then
                    cntb <= cntb + 1;
                else
```

```
                cntb <= cntb + 2;
            end if;
        else
            cntb <= cntb;
        end if;
    end if;
  end process upcntb;
end archcounter;
```

Listing 8-9 Counter that counts *by1* or by 2

The product term allocation scheme can also affect the ability of a design to fit with a preassigned pinout. For example, suppose two signals, *a* and *b*, neighbor each other (in both pinout and macrocell location). If one signal, *a*, requires many product terms, then depending on the product term allocation scheme, neighboring macrocells (including *b*) may have to give up product terms, in which case it may not be possible to allocate product terms for *b (see Figure 8-10)*. Suppose that

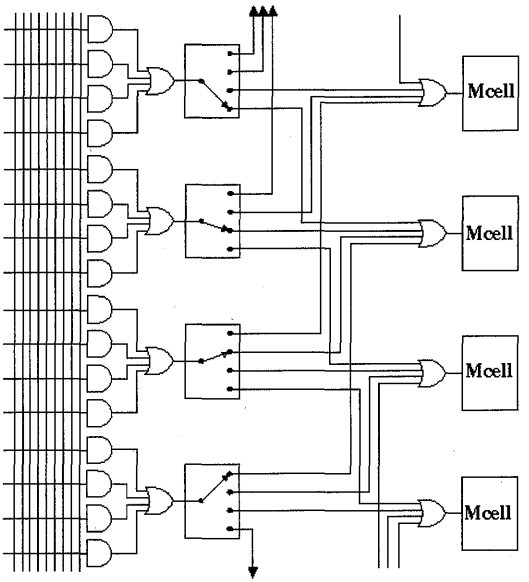


Figure 8-10 Product term allocation scheme #1

signal *a* requires five product terms. In this case, *a* will require the use of a product term from a neighboring macrocell. Because the product term allocation scheme steers product terms in groups of four, *b* must give up all of its available product terms. If macrocells neighboring *b* cannot forfeit their product terms for *b*, then *b* is left without any product terms, in which case the design will not fit with this preassignment of pins and macrocells. *Figure 8-11* shows another product term allocation scheme in which product terms may be steered (in groups of five) from one macrocell to a neighboring macrocell. This scheme also makes use of several additional expander product terms that may be used with any macrocell at the expense of an incremental delay. This scheme avoids the need for neighboring macrocells to give up all product terms in all cases in which greater than five product

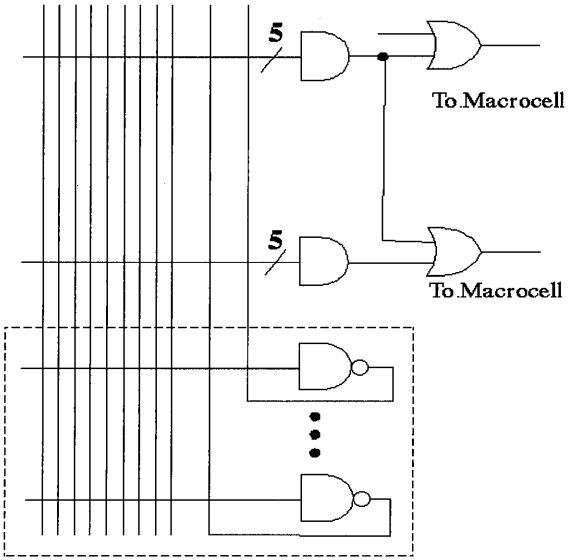


Figure 8-11 Product term allocation scheme #2

terms are needed on any given macrocell. If a macrocell has a high product-term requirement, however, then neighboring macrocells may have to forfeit their product terms, eliminating the possibility of a fit with a preassigned pinout. *Figure 8-12* illustrates a product term allocation scheme in which the product terms may be steered individually, permitting a fit with a preassigned pinout in the case where *a* and *b* each require more than 5 product terms (or up to a combined total of 20 product terms). What happens if 16 product terms are required for *a* and more than 4 product terms are required for *b* (or greater than 20 product terms for any pair of macrocells)? In such a scenario, this architecture could not permit signal *b* to be placed on a neighboring pin (unless *a* was at the top or bottom of the logic block where two adjacent macrocells are allocated 22 unique product terms).

Unless a CPLD specifically guarantees a cross-point switch for the interconnection of logic blocks and the interconnection of macrocells to I/O pins, preassigning a pinout or making a logic change after the fitter has chosen a pinout introduces a subsequent fitting constraint for which it may not be possible to find a solution given your design's resource and feature set requirements. Even with a cross-point switch, a fit will not be possible if a design change requires that more signals route to a logic block than there are inputs to that logic block. If you must preassign a pinout, be sure to understand all the issues involved with the architectural features that will affect the ability of your design to fit in the target architecture. Understanding the routability of signals through the programmable interconnect and to the logic blocks (as quantified in *Figure 8-9*) will also help you to judge whether it will be possible to make design changes and keep the same pinout.

Clocking

Most CPLDs have synchronous clocks—with dedicated pins—not only because synchronous clocks are inherently faster but also because asynchronous clocking is not a "standard design practice."

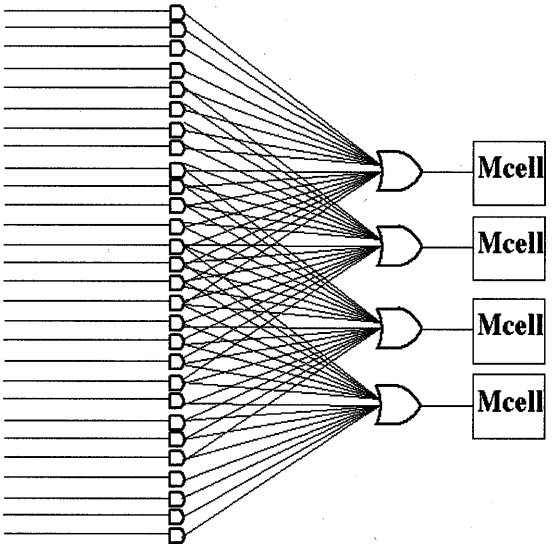


Figure 8-12 Product term allocation scheme #3

Some CPLDs include product term (gated) clocks for asynchronous clocking to accommodate designs that may require such clocking schemes. If your design uses gated clocks, then the programmable logic device that you choose must support this clocking scheme. The clock multiplexer circuit of the network repeater is an example of a design that uses product term clocking.

An interesting feature of the FLASH370 devices is the ability to select either polarity for a clock on a logic-block-by-logic-block basis. This feature allows state machines, counters, or other logic to run at twice the system frequency (but consumes twice the resources because the logic must be replicated in separate logic blocks).

Take for example a DRAM controller (*Figure 8-15*) that operates on a 20 MHz clock, for which the code is shown in *Listing 8-10*. The system address is captured on the rising edge of a clock during which the address strobe is asserted. The upper bits of the system address are examined to determine if the address is for a memory location. This address comparison is evaluated in the *address_detect* state. Subsequent states place the row and column addresses on the bus and assert *RAS* and *CAS* at the proper time. The timing diagram for interfacing to an asynchronous DRAM controller is shown if *Figure 8-13*.

```
library ieee;
use ieee.std_logic_1164.all;
library ieee;
use ieee.std_logic_1164.all;
package dram_pkg is
  component dram_controller port (
    addr: in std_logic_vector(31 downto 0);    -- system address
    clock,                                     -- clock 20MHz
    ads,                                       -- address strobe
```

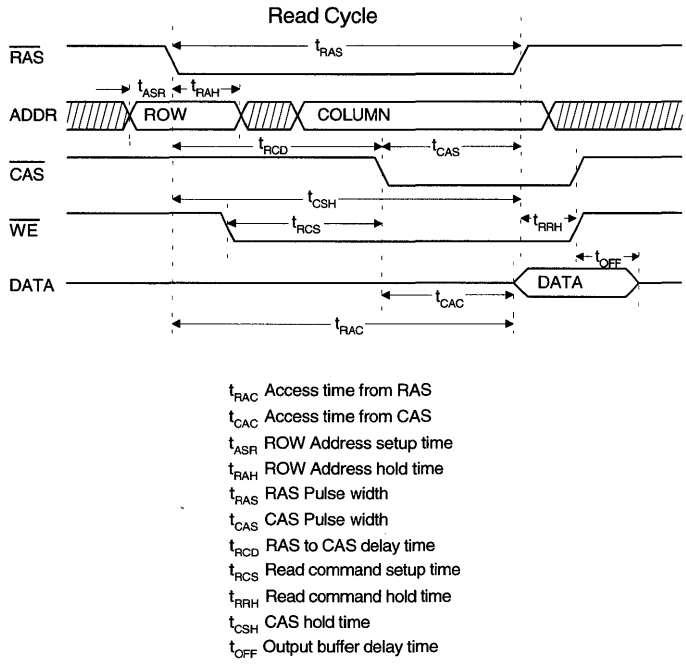


Figure 8-13 Timing diagram for DRAM interface

```
    read_write,                                  -- read/write
    reset: in std_logic;                         -- system reset

    ack: out std_logic;                          -- acknowledge
    we: out std_logic;                           -- write enable
    ready: out std_logic;       -- data ready for latching
    dram:out std_logic_vector (9 downto 0);      -- DRAM address
    ras: out std_logic_vector(1 downto 0);       -- row address strobe
    cas: out std_logic_vector(3 downto 0));      -- column address strobe
  end component;
end dram_pkg;

library ieee;
use ieee.std_logic_1164.all;
entity dram_controller is port (
    addr: in std_logic_vector(31 downto 0);      -- system address
    clock,                                       -- clock 20MHz
    ads,                                         -- address strobe
    read_write,       -- read/write
    reset: in std_logic;                         -- system reset

    ack: out std_logic;                          -- acknowledge
```

```
    we: out std_logic;       -- write enable
    ready: out std_logic;       -- data ready for latching
    dram:out std_logic_vector (9 downto 0);    -- DRAM address
    ras: out std_logic_vector(1 downto 0);     -- row address strobe
    cas: out std_logic_vector(3 downto 0));    -- column address strobe
end dram_controller;

use work.std_math.all;
architecture controller of dram_controller is
  type states is (idle, address_detect, row_address, ras_assert,
col_address,
        cas_assert, data_ready, wait_state, refresh0, refresh1);
  signal present_state, next_state: states;
  signal stored: std_logic_vector(31 downto 0);      -- latched addr
  signal ref_timer:std_logic_vector(8 downto 0);     -- refresh timer
  signal ref_request: std_logic;     -- refresh request
  signal match: std_logic;     -- address match
  signal read: std_logic;     -- latched read_write
   -- row and column address aliases
  alias row_addr: std_logic_vector(9 downto 0) is stored(19 downto 10);
  alias col_addr: std_logic_vector(9 downto 0) is stored(9 downto 0);

  --attribute synthesis_off of match,ref_request: signal is true;
begin
-------------------------------------------------------------
--                    Capture Address      --
-------------------------------------------------------------
capture: process (reset, clock)
  begin
   if reset = '1' then
      stored <= (others => '0');
      read <= '0';
   elsif (clock'event and clock='1') then
     if ads = '0' then
      stored <= addr;
      read <= read_write;
     end if;
   end if;
  end process;

-------------------------------------------------------------
--                    Address Comparator    --
-------------------------------------------------------------
-- The address comparator determines if memory is being accessed

  match <= '1' when stored(31 downto 21) = "00000000000" else '0';

-------------------------------------------------------------
--                    Address Multiplexer                --
-------------------------------------------------------------
-- The address multiplexer selects the row, column, or refresh
-- address depending on the current cycle

multiplexer: process (row_addr, col_addr, present_state)
```

```
  begin
   if ( present_state = row_address or present_state = ras_assert) then
      dram <= row_addr;
   else
      dram <= col_addr;
   end if;
  end process;

-------------------------------------------------------------
--         Refresh Counter & Refresh Timer                 --
-------------------------------------------------------------
-- The refresh timer is used to initiate refresh cycles. A
-- refresh cycle is required every 8ms. If the clock frequency
-- is 20MHz, then a refresh request must be generated every 312
-- clock cycles. Refresh_req is asserted until a refresh cycle
-- begins

synchronous: process (reset, clock)
  begin
    if reset = '1' then
   ref_timer <= (others => '0');
    elsif clock'event and clock = '1' then
   if (ref_timer = "100111000") then  -- start request at 312
      ref_timer <= (others => '0');
   else
      ref_timer <= ref_timer + 1;
   end if;
    end if;
  end process;

  ref_request <= '1' when (ref_timer = "100111000" or
            (ref_request = '1' and present_state /= refresh0))
       else '0';

-------------------------------------------------------------
--       DRAM State Machine                                --
-------------------------------------------------------------
-- The DRAM controller state machine controls the state of
-- the address multiplexer select lines as well as the
-- state of RAS and CAS

state_tr: process (present_state, ref_request, ads, match,
         stored(20), read)
  begin
    case present_state is
   when idle =>
      if ref_request = '1' then
         next_state <= refresh0;
      elsif ads = '0' then
         next_state <= address_detect;
      else
         next_state <= idle;
      end if;
  when address_detect =>
```

```
      if match = '1' then
         next_state <= row_address;
      else
         next_state <= idle;
      end if;
    when row_address =>
      next_state <= ras_assert;
    when ras_assert =>
      next_state <= col_address;
    when col_address =>
      next_state <= cas_assert;
    when cas_assert =>
      next_state <= data_ready;
    when data_ready =>
      next_state <= wait_state;
    when wait_state =>
      next_state <= idle;
    when refresh0 =>
      next_state <= refresh1;
    when refresh1 =>
      next_state <= idle;
     end case;
  end process;

clocked: process (reset, clock)
  begin
   if reset = '1' then
      present_state <= idle;
   elsif (clock'event and clock = '1') then
      present_state <= next_state;
   end if;
  end process;

   with present_state select
     cas <=  "0000" when cas_assert | data_ready | wait_state | refresh0
| refresh1,
        "1111" when others;

   ras <= "00" when (present_state = refresh1)
      else "01" when ((present_state = ras_assert or present_state =
col_address or
           present_state = cas_assert or present_state = data_ready or
           present_state = wait_state) and stored(20)='1')
      else "10" when ((present_state = ras_assert or present_state =
col_address or
           present_state = cas_assert or present_state = data_ready or
           present_state = wait_state) and stored(20)='0')
      else "11";

   we <= '0' when ((present_state = col_address or present_state =
cas_assert or
         present_state = data_ready) and read = '0')
      else '1';
```

```
   ack <= '0' when (present_state = address_detect and match = '1') else
'1';

   ready <= '0' when (read = '1' and (present_state = data_ready or
            present_state = wait_state)) else '1';

end controller;
```

Listing 8-10 DRAM controller

While the design is functionally accurate, the interface to the DRAM may be slower than the maximum specification of the DRAM. With a 20MHz clock, *RAS* and *CAS* are asserted 50 ns apart. What if the DRAMs are 60ns DRAMs (i.e., data is valid 60 ns after *RAS* is asserted)? To take advantage of the faster DRAM access times, we would need to run the state machine at twice the current frequency. One solution is to use a two-phase clock (possibly obtained by a programmable skew clock buffer such as the CY7B991). Another solution is to clock registers on the rising and falling edge, as allowed in some CPLDs. This requires that two state machines and outputs that are multiplexed based on the current state of the clock (*Figure 8-14*). *Listing 8-11* is this design. One state machine looks at the present state of the other to determine its next state. Two sets of outputs are produced, and are multiplexed based on *clock*. The Capture Address and Refresh Controller &

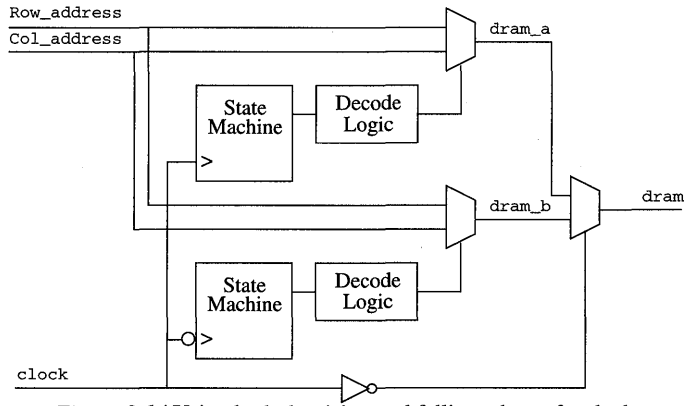


Figure 8-14 Using both the rising and falling edges of a clock

Refresh Timer processes do not change.

This design illustrates the concept of clocking a state machine on both the rising and falling edges of a clock, but would need to be modified before being used with an asynchronous DRAM. Care must be taken to ensure that there is not any glitching on any of the asynchronous interface signals. Glitching could cause erroneous accesses or unpredictable behavior of the DRAM.

```
use work.std_math.all;
library ieee;
use ieee.std_logic_1164.all;
package dram_pkg is
  component dram_controller port (
```

```
    addr: in std_logic_vector(31 downto 0);    -- system address
    clock,                                     -- clock 20MHz
    ads,                                       -- address strobe
    read_write,                                -- read/write
    reset: in std_logic;                       -- system reset

    ack: out std_logic;                        -- acknowledge
    we: out std_logic;                         -- write enable
    ready: out std_logic;       -- data ready for latching
    dram:out std_logic_vector (9 downto 0);    -- DRAM address
    ras: out std_logic_vector(1 downto 0);     -- row address strobe
    cas: out std_logic_vector(3 downto 0));    -- column address strobe
  end component;
end dram_pkg;

library ieee;
use ieee.std_logic_1164.all;
entity dram_controller is port (
    addr: in std_logic_vector(31 downto 0);    -- system address
    clock,                                     -- clock 20MHz
    ads,                                       -- address strobe
    read_write,       -- read/write
    reset: in std_logic;                       -- system reset

    ack: out std_logic;                        -- acknowledge
    we: out std_logic;       -- write enable
    ready: out std_logic;       -- data ready for latching
    dram:out std_logic_vector (9 downto 0);    -- DRAM address
    ras: out std_logic_vector(1 downto 0);     -- row address strobe
    cas: out std_logic_vector(3 downto 0));    -- column address strobe
end dram_controller;

use work.std_math.all;
architecture controller of dram_controller is
  type states is (idle, address_detect, row_address, ras_assert,
col_address,
        cas_assert, data_ready, wait_state, refresh0, refresh1);
  signal present_state_a, next_state_a: states;
  signal present_state_b, next_state_b: states;
  signal dram_a, dram_b: std_logic_vector(9 downto 0);
  signal ras_a, ras_b: std_logic_vector(1 downto 0);
  signal cas_a, cas_b: std_logic_vector(3 downto 0);
  signal we_a, we_b, ack_a, ack_b, ready_a, ready_b: std_logic;
  signal stored: std_logic_vector(31 downto 0);      -- latched addr
  signal ref_timer:std_logic_vector(8 downto 0);     -- refresh timer
  signal ref_request: std_logic;     -- refresh request
  signal match: std_logic;     -- address match
  signal read: std_logic;     -- latched read_write
   -- row and column address aliases
  alias row_addr: std_logic_vector(9 downto 0) is stored(19 downto 10);
  alias col_addr: std_logic_vector(9 downto 0) is stored(9 downto 0);

  --attribute synthesis_off of match,ref_request: signal is true;
begin
```

```
-------------------------------------------------------------
--                    Capture Address      --
-------------------------------------------------------------
capture: process (reset, clock)
  begin
   if reset = '1' then
      stored <= (others => '0');
      read <= '0';
   elsif (clock'event and clock='1') then
     if ads = '0' then
      stored <= addr;
      read <= read_write;
     end if;
   end if;
  end process;

-------------------------------------------------------------
--                    Address Comparator    --
-------------------------------------------------------------
-- The address comparator determines if memory is being accessed

  match <= '1' when stored(31 downto 21) = "00000000000" else '0';

-------------------------------------------------------------
--                    Address Multiplexer                  --
-------------------------------------------------------------
-- The address multiplexer selects the row, column, or refresh
-- address depending on the current cycle

multiplexer_a: process (row_addr, col_addr, present_state_a)
  begin
   if ( present_state_a = row_address or present_state_a = ras_assert) then
      dram_a <= row_addr;
   else
      dram_a <= col_addr;
   end if;
  end process;

multiplexer_b: process (row_addr, col_addr, present_state_b)
  begin
   if ( present_state_b = row_address or present_state_b = ras_assert) then
      dram_b <= row_addr;
   else
      dram_b <= col_addr;
   end if;
  end process;

-------------------------------------------------------------
--         Refresh Counter & Refresh Timer               --
-------------------------------------------------------------
-- The refresh timer is used to initiate refresh cycles. A
-- refresh cycle is required every 8ms. If the clock frequency
-- is 20MHz, then a refresh request must be generated every 312
-- clock cycles. Refresh_req is asserted until a refresh cycle
```

```
-- begins

synchronous: process (reset, clock)
  begin
    if reset = '1' then
   ref_timer <= (others => '0');
    elsif clock'event and clock = '1' then
   if (ref_timer = "100111000") then  -- start request at 312
      ref_timer <= (others => '0');
   else
      ref_timer <= ref_timer + 1;
   end if;
    end if;
  end process;

  ref_request <= '1' when (ref_timer = "100111000" or
            (ref_request = '1' and present_state_a /= refresh0))
       else '0';

-------------------------------------------------------------
--       DRAM State Machine                                --
-------------------------------------------------------------
-- The DRAM controller state machine controls the state of
-- the address multiplexer select lines as well as the
-- state of RAS and CAS

state_tr_a: process (present_state_b, ref_request, ads, match,
         stored(20), read)
  begin
    case present_state_b is
   when idle =>
      if ref_request = '1' then
         next_state_a <= refresh0;
      elsif ads = '0' then
         next_state_a <= address_detect;
      else
         next_state_a <= idle;
      end if;
   when address_detect =>
      if match = '1' then
         next_state_a <= row_address;
      else
         next_state_a <= idle;
      end if;
   when row_address =>
      next_state_a <= ras_assert;
   when ras_assert =>
      next_state_a <= col_address;
   when col_address =>
      next_state_a <= cas_assert;
   when cas_assert =>
      next_state_a <= data_ready;
   when data_ready =>
      next_state_a <= wait_state;
```

```
    when wait_state =>
       next_state_a <= idle;
    when refresh0 =>
       next_state_a <= refresh1;
    when refresh1 =>
       next_state_a <= idle;
     end case;
  end process;

clocked_a: process (reset, clock)
  begin
    if reset = '1' then
       present_state_a <= idle;
    elsif (clock'event and clock = '1') then
       present_state_a <= next_state_a;
    end if;
  end process;

  with present_state_a select
    cas_a <=  "0000" when cas_assert | data_ready | wait_state | refresh0 |
refresh1,
      "1111" when others;

  ras_a <= "00" when (present_state_a = refresh1)
    else "01" when ((present_state_a = ras_assert or present_state_a =
col_address or
        present_state_a = cas_assert or present_state_a = data_ready or
        present_state_a = wait_state) and stored(20)='1')
    else "10" when ((present_state_a = ras_assert or present_state_a =
col_address or
        present_state_a = cas_assert or present_state_a = data_ready or
        present_state_a = wait_state) and stored(20)='0')
    else "11";

  we_a <= '0' when ((present_state_a = col_address or present_state_a =
cas_assert or
      present_state_a = data_ready or present_state_a = wait_state) and
read = '0')
    else '1';

  ack_a <= '0' when (present_state_a = address_detect and match = '1') else
'1';

  ready_a <= '0' when (read = '1' and (present_state_a = data_ready or
           present_state_a = wait_state)) else '1';


state_tr_b: process (present_state_a, ref_request, ads, match,
         stored(20), read)
  begin
    case present_state_a is
   when idle =>
      if ref_request = '1' then
         next_state_b <= refresh0;
```

```
      elsif ads = '0' then
         next_state_b <= address_detect;
      else
         next_state_b <= idle;
      end if;
    when address_detect =>
      if match = '1' then
         next_state_b <= row_address;
      else
         next_state_b <= idle;
      end if;
    when row_address =>
      next_state_b <= ras_assert;
    when ras_assert =>
      next_state_b <= col_address;
    when col_address =>
      next_state_b <= cas_assert;
    when cas_assert =>
      next_state_b <= data_ready;
    when data_ready =>
      next_state_b <= wait_state;
    when wait_state =>
      next_state_b <= idle;
    when refresh0 =>
      next_state_b <= refresh1;
    when refresh1 =>
      next_state_b <= idle;
     end case;
  end process;

clocked_b: process (reset, clock)
  begin
   if reset = '1' then
      present_state_b <= idle;
   elsif (clock'event and clock = '0') then
      present_state_b <= next_state_b;
   end if;
  end process;

  with present_state_b select
    cas_b <=  "0000" when cas_assert | data_ready | wait_state | refresh0 |
refresh1,
     "1111" when others;

  ras_b <= "00" when (present_state_b = refresh1)
   else "01" when ((present_state_b = ras_assert or present_state_b =
col_address or
        present_state_b = cas_assert or present_state_b = data_ready or
        present_state_b = wait_state) and stored(20)='1')
   else "10" when ((present_state_b = ras_assert or present_state_b =
col_address or
        present_state_b = cas_assert or present_state_b = data_ready or
        present_state_b = wait_state) and stored(20)='0')
   else "11";
```

```
  we_b <= '0' when ((present_state_b = col_address or present_state_b =
cas_assert or
      present_state_b = data_ready) and read = '0')
   else '1';

  ack_b <= '0' when (present_state_b = address_detect and match = '1') else
'1';

  ready_b <= '0' when (read = '1' and (present_state_b = data_ready or
          present_state_b = wait_state)) else '1';



-------------------------------------------------------
-- Output Multiplexers;
-------------------------------------------------------
  cas   <= cas_a when clock = '1' else cas_b;
  ras   <= ras_a when clock = '1' else ras_b;
  we    <= we_a  when clock = '1' else we_b;
  ack   <= ack_a when clock = '1' else ack_b;
  dram  <= dram_a when clock = '1' else dram_b;
  ready <= ready_a when clock = '1' else ready_b;


end controller;
```

Listing 8-11 DRAM controller operating on both rising and falling edges of the clock.

Falling edge clocks can also be used to align data that is transferred between two buses operating at different speeds, or to work around a race condition between data and a buffered, among other possibilities.

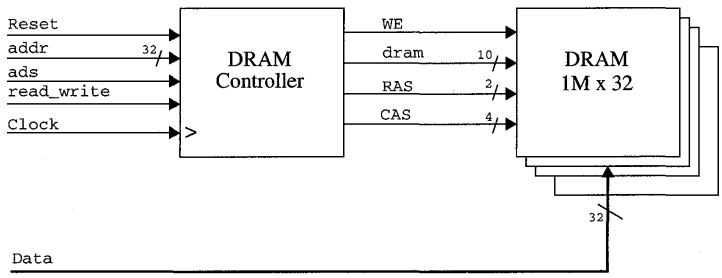


Figure 8-15 DRAM Controller interfacing to an asynchronous DRAM

Implementing Network Repeater Ports in a CY7C374

In this section, we will examine the implementation of three network repeater port controllers in the 128 macrocell member of the FLASH370 family of devices. We will begin by analyzing the design to determine how many and which resources the design will require. We will then synthesize the design and examine the report file to compare our expectations with the actual realization of the circuit. Listing 6-6 of chapter 6 defines one network port controller of the repeater core logic. The design of the network repeater port controllers was made modular so that repeater port controllers could easily be added or removed from the top-level design. In chapter 6, eight ports are used, and the implementation of the design in one device, an FPGA, is discussed later in that chapter. However, an alternate methodology for implementation is to place the port controllers in external devices, leaving the FIFO, core controller, clock multiplexer, and so forth, in a smaller FPGA. *Listing 8-12* is the code required to implement three repeater port controllers in one device.

```
library ieee;
use ieee.std_logic_1164.all;
entity port3 is port(
        txclk, areset:                          in std_logic;
        crs1, enable1_bar, link1_bar, sel1:     in std_logic;
        carrier, collision, jam, txdata,prescale: in std_logic;
        rx_en1, tx_en1, activity1:              inout std_logic;
        jabber1_bar, partition1_bar:            inout std_logic;
        crs2, enable2_bar, link2_bar, sel2:     in std_logic;
        rx_en2, tx_en2, activity2:              inout std_logic;
        jabber2_bar, partition2_bar:            inout std_logic;
        crs3, enable3_bar, link3_bar, sel3:     in std_logic;
        rx_en3, tx_en3, activity3:              inout std_logic;
        jabber3_bar, partition3_bar:            inout std_logic);
end port3;

use work.port3top_pkg.all;
architecture archport3 of port3 is
begin


u1: porte port map
        (txclk, areset,
        crs1, enable1_bar, link1_bar,
        sel1, carrier, collision, jam, txdata, prescale, rx_en1, tx_en1,
        activity1, jabber1_bar, partition1_bar);

u2: porte port map
        (txclk, areset,
        crs2, enable2_bar, link2_bar,
        sel2, carrier, collision, jam, txdata, prescale, rx_en2, tx_en2,
        activity2, jabber2_bar, partition2_bar);

u3: porte port map
        (txclk, areset,
        crs3, enable3_bar, link3_bar,
        sel3, carrier, collision, jam, txdata, prescale, rx_en3, tx_en3,
        activity3, jabber3_bar, partition3_bar);
```

```
end archport3;
```

Listing 8-12 Three network repeater ports to be implemented in a CPLD

Reviewing *Listing* 6-6 of chapter 6, we can determine the required number of macrocells to implement one repeater with fairly good precision. Determining the number of required macrocells requires that we understand how many registers and combinatorial outputs are needed, as well as whether any of the combinatorial logic exceeds 16 product terms (in which case, the logic would require two levels of logic and more than one macrocell). Each of the three repeater ports require

- 19 buried registers to hold the current state of the counters,
- 3 buried registers for an 8-state state machine,
- 6 input registers to synchronize *crs*, *link_bar*, and *enable_bar*,
- 2 registers for *copyin* and *collision*, and
- 5 output macrocells (one registered, four combinational) for outputs.

This is a total of 35 I/O macrocells for each port controller, or 105 for three controllers. There are 6 clock/input macrocells on the device. One will be used for clocking, leaving 5 available to replace 10 I/O macrocells for synchronizing external signals. One input macrocell can replace two I/O macrocells because the input macrocells, designed to synchronize asynchronous signals, contain two registers. Thus we expect that a minimum of 95 I/O macrocells and 10 input macrocells will be required. More than three macrocells may be required for the state machine if the state transition logic is sufficiently complex. The product term requirements are considerably more difficult to estimate, but following is our attempt:

- 57 product terms for all counter bits (3 product terms per counter bit, assuming an implementation with T-type flip-flops and a counter with enable and synchronous clear).
- 24 product terms for the state machine (average of 8). This is a guess.
- 8 product terms for all of the outputs. Based on the descriptions, most outputs (except *partition_bar*) are simple decodes of registers.

This is a total of 89 product terms for each port controller, or 267 total product terms. This is well below the 640 available product terms.

Below is a summary from a report file excerpt of the utilization of the realized circuit after synthesis and fitting:

```
                                  Required     Max (Available)
CLOCK/LATCH ENABLE signals             1              4
Input REG/LATCH signals                5              5
Input PIN signals                      0              0
Input PIN signals using I/O cells      7              7
Output PIN signals                    95            121


Total PIN signals                    108            134
Macrocells used                       95            128
Unique product terms                 254            640
```

The circuit realization meets our expectations for macrocell utilization (95 I/O macrocells and 5 input macrocells). Fewer product terms were required than expected, perhaps due to an overestimate on our part or the ability of some product terms to be shared among multiple macrocells. There is a substantial amount of resources available for additional logic or for significant design changes if required.

The diagram below is an excerpt from a report file and illustrates how the product terms in one logic block were assigned. The logic block's eighty product terms are numbered at the top from 0 to 79. The sixteen macrocells are listed on the left, numbered from 0 to 15 with the signal name just to the right of the macrocell number. Each macrocell's allotment of product terms (16) is shown in the row directly below the macrocell number and name. A + sign indicates an unused product term. An X represents a used product term. Most product terms may be shared by several macrocells. For example, product terms 18, 19, 20, and 21 can all be shared by macrocells 1, 2, 3, and 4. This does not mean, however, that because product term 18 is used by macrocell 4 that it *must* be used by macrocells 1, 2, and 3. The diagram illustrates that this is not the case. (An X is placed in product term location 18 for macrocell 4, and a + is placed in product term location 18 for the other macrocells.) This logic block was the most heavily used logic block in the device, yet there are still several product terms available. This placement of macrocells can easily accommodate a design change that requires the macrocells to utilize additional product terms.

```
         1111111111222222222233333333334444444444555555555566666666667777777777
01234567890123456789012345678901234567890123456789012345678901234567890123456789
--------------------------------------------------------------------------------
| 0 |(u2_stateSBV_2)
XXXXXXX+XX++++++.................................................................
| 1 |(u3_crsdd)
......+X++++++++++++++..........................................................
| 2 |partition2_bar
..........X+XX++++++++++++.......................................................
| 3 |(u2_cccnt_1)
..............X+XX++++++++++++...................................................
| 4 |(u2_cccnt_3)
..................X+XX++++++++++++.............................................
| 5 |(u2_cccnt_5)
......................X+XX++++++++++++.........................................
| 6 |(u2_jabcnt_0)
..........................X+XX++++++++++++.....................................
| 7 |(u2_jabcnt_3)
..............................X+XX++++++++++++.................................
| 8 |(u2_jabcnt_2)
..................................X+XX++++++++++++.............................
| 9 |(u2_jabcnt_1)
......................................X+XX++++++++++++.........................
|10 |(u2_cccnt_6)
..........................................X+XX++++++++++++.....................
|11 |(u2_cccnt_4)
..............................................X+XX++++++++++++.................
|12 |(u2_cccnt_2)
..................................................X+XX++++++++++++.............
|13 |(u2_cccnt_0)
......................................................X+XX++++++++++++.........
|14 |jabber2_bar
..........................................................X++++++++++++++......
|15 |(u2_stateSBV_1)
..................................................................XXXXXXX+XX++++++
```

```
Total product terms to be assigned    =  56
Max product terms used / available    =  56 / 80  = 70.1 %
```

The diagram below is an excerpt from the report file that illustrates the signals that were routed to the logic block (on the left) and the macrocell placements (on the right). Macrocells for which the outputs are not driven to I/O buffers are shown in parentheses. Those for which outputs propagate to the pins are shown without parentheses. Of the 36 inputs to the logic block, 26 were required. Design changes that require additional inputs to the logic block should not present a problem.

```
                              Logic Block 5

|    |= >jabber2_bar                          |    |
|    |= >prescale                             |    |
|    |= >u2_enable_b..                        |    |
|    |= >u2_jabcnt_2.Q                        | 63 |= (u2_stateSBV_2)
|    |= >u2_cccnt_0.Q                         |    |
|    |= >u2_cccnt_6.Q                         | 64 |= (u3_crsdd)
|    |= >areset                               |    |
|    |= >u2_jabcnt_3.Q                        | 65 |= partition2_bar
|    |= >u2_stateSBV..                        |    |
|    |= >u2_copyd.Q                           | 66 |= (u2_cccnt_1)
|    |= >partition2_..                        |    |
|    |= >u3_collisio..                        | 67 |= (u2_cccnt_3)
|    |= >u2_jabcnt_0.Q                        |    |
|    |= >u2_crsdd.Q                           | 68 |= (u2_cccnt_5)
|    |= >u2_cccnt_4.Q                         |    |
|    |> not used:333                          | 69 |= (u2_jabcnt_0)
|    |> not used:334                          |    |
|    |> not used:335                          | 70 |= (u2_jabcnt_3)
|    |= >u2_nocolcnt..                        |    |
|    |= >u2_cccnt_1.Q                         | 72 |= (u2_jabcnt_2)
|    |> not used:338                          |    |
|    |= >u2_stateSBV..                        | 73 |= (u2_jabcnt_1)
|    |> not used:340                          |    |
|    |= >u2_cccnt_3.Q                         | 74 |= (u2_cccnt_6)
|    |> not used:342                          |    |
|    |> not used:343                          | 75 |= (u2_cccnt_4)
|    |> not used:344                          |    |
|    |> not used:345                          | 76 |= (u2_cccnt_2)
|    |= >u2_jabcnt_1.Q                        |    |
|    |> not used:347                          | 77 |= (u2_cccnt_0)
|    |> not used:348                          |    |
|    |= >u2_cccnt_2.Q                         | 78 |= jabber2_bar
|    |> not used:350                          |    |
|    |> not used:351                          | 79 |= (u2_stateSBV_1)
|    |= >crs3.QI                              |    |
|    |= >u2_cccnt_5.Q                         |    |
```

Below is a summary of the worst-case performance metrics. The worst-case combinational propagation delay is 12.0 ns. This indicates that all combinational logic can be implemented in one level of logic (one pass through the product term array). The worst-case setup time with respect to the clock is 7.0 ns. The worst-case register-to-register delay is 10 ns (which supports 100MHz operation, well above the required 25 MHz). The worst-case clock-to-output delay is 15.0 ns. This clock-to-output delay (listed as tCO in the report file below) represents t_{CO2} rather than t_{CO} because the partition outputs are decoded from the state bits, requiring an additional level of logic.

```
Worst Case Path Summary
-----------------------
Worst case COMB,    tmax = 12.0 ns for activity1
Worst case PIN->D,  tS   = 7.0 ns for tx_en2.D
Worst case Q->Q,    tmax = 10.0 ns for tx_en2.D
Worst case CLK->Q,  tCO  = 15.0 ns for partition2_bar.C
```

Our case study of the FLASH370 has served to identify the relationship between synthesis and fitting, to illustrate how to take advantage of CPLD resources, to point out differences between CPLDs regarding their capabilities, and to enable resource utilization and performance estimations.

Having covered these issues, we will now turn our attention to the issues involved with designing with FPGAs. CPLDs will enter our discussion once more, later in the chapter, when arithmetic operators are examined for both CPLD and FPGA architectures.

FPGAs: A Case Study (pASIC 380 Architecture)

Designs, such as the counters examined in our CPLD study, do not usually present fitting problems when targeted to FPGAs because FPGAs typically have the resources to handle a variety of designs, making them more akin to semi-custom gate arrays than to CPLDs. The task of synthesizing and fitting designs to FPGAs does not center around algorithms to partition logic among logic blocks, route signals through the programmable interconnect, and steer or share logic block resources, as it does for CPLDs; rather, the task centers around optimizing logic and signal paths for the device architecture in order to achieve the appropriate balance (as directed by you, the designer) between density and speed trade-offs. In this section, we'll explore some of the issues involved with targeting designs to FPGAs. (You may wish to review some of the differences between CPLDs and FPGAs in the chapter titled, "Programmable Logic Primer.")

A block diagram of the FPGA architecture that we will be using for our discussion is shown in *Figure 8-16*. As with most other FPGAs, it consists of an array of logic cells that communicate with each other and the I/O through routing wires within the routing channels. The logic cell (see *Figure 8-17*) consists of a flip-flop, three two-to-one multiplexers (which may be cascaded as a four-to-one multiplexer), and AND gates for the multiplexer select lines and inputs The logic cell has multiple outputs that may be used at the same time. The flip-flop clock, asynchronous set, and asynchronous reset signals may be driven by any internal signal or by a signal from one of the dedicated low-skew, high-performance distribution trees. A theoretical architecture model consisting of two logic cells is shown in *Figure 8-18*. The smallest device in this family of devices, the CY7C381, has 96 logic cells (an array of eight by twelve), with 22 signal routing wires in each of the vertical routing channels and 12 routing wires in each of the horizontal routing channels.

Synthesizing and fitting designs for the 380 architecture

Determining the optimal design implementation for a design in an FPGA is not as easy as you might think. For one, there may be multiple solutions that are comparable in performance and capacity. Also, "optimal" can be different for two designers or two designs. A designer may be concerned with performance for one design, and with another design, for cost reasons, the same designer may be concerned only with fitting the design into the smallest FPGA. Oftentimes, the design must fit in a particular size device, but certain signal paths must meet specific performance criteria. This means that some logic should be packed in as tightly as possible with less concern for performance, while other portions of logic should be placed and routed to produce high performance with less concern for resource conservation. In achieving the density and performance requirements of a particular design, two of the most challenging tasks for FPGA synthesis and fitting tools are to (1) optimize logic for the FPGA logic cell architecture and (2) make the appropriate trade-offs while placing and routing the logic by prioritizing placement of critical portions of logic.

Optimizing logic for FPGA logic resources is more challenging than it is for most CPLDs because the architecture does not lend itself easily to a sum-of-products (SP) implementation. Most software

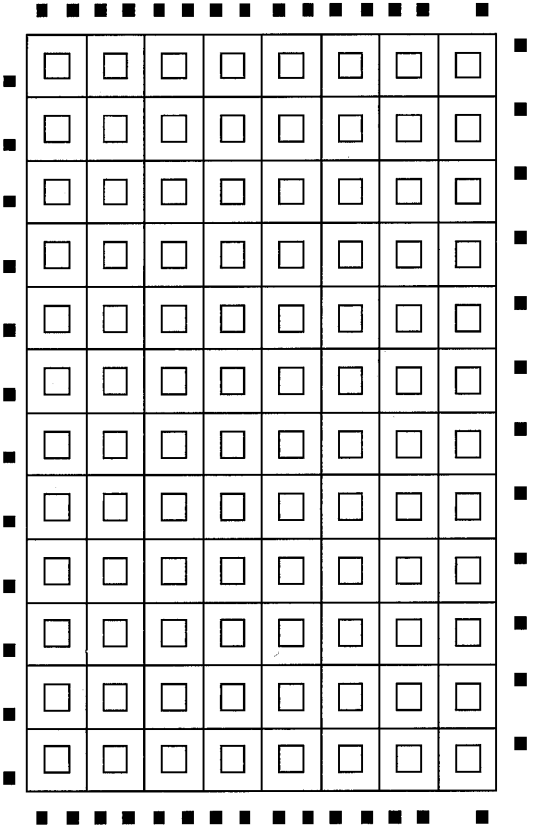


Figure 8-16 Block diagram of an FPGA

logic minimization algorithms begin with SP equations and then map this logic to a device's resources. (Other algorithms may use binary decision diagrams to map directly to logic cells.) With CPLDs, the task of mapping SP equations is easy—the products are mapped to the product term array, and these products are summed at each macrocell. Some CPLDs have an XOR gate included in the macrocell (one input to the XOR as a sum of a variable number of product terms and the other input as one product), but software can work with the logic to determine if the XOR can provide any logic reduction. If not, the XOR input with one product term is tied low. For FPGAs, software must have special mapping technologies to optimize logic for the device resources. Take, for instance, the logic cell of *Figure 8-17*. The three 2-to-1 multiplexers can be cascaded to create a 4-to-1 multiplexer. A 2^n-to-1 multiplexer is a universal logic module that can implement any function of (n + 1) variables, provided that the true and complement of each variable are available; thus, this logic cell can implement any function of three variables. But using an algorithm that assumes that this is the *most* logic that can be implemented in this logic cell would potentially waste many of its resources. After all, many other functions can be implemented: a seven-input AND gate, a seven-

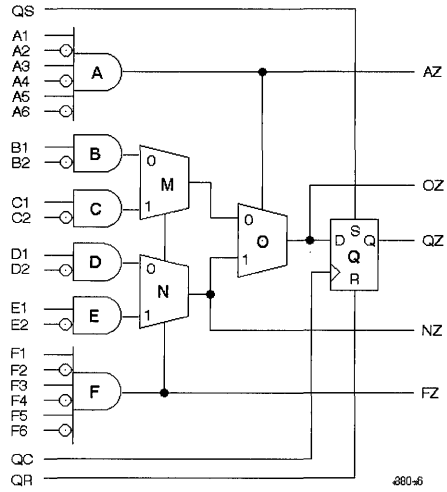


Figure 8-17 Logic cell architecture

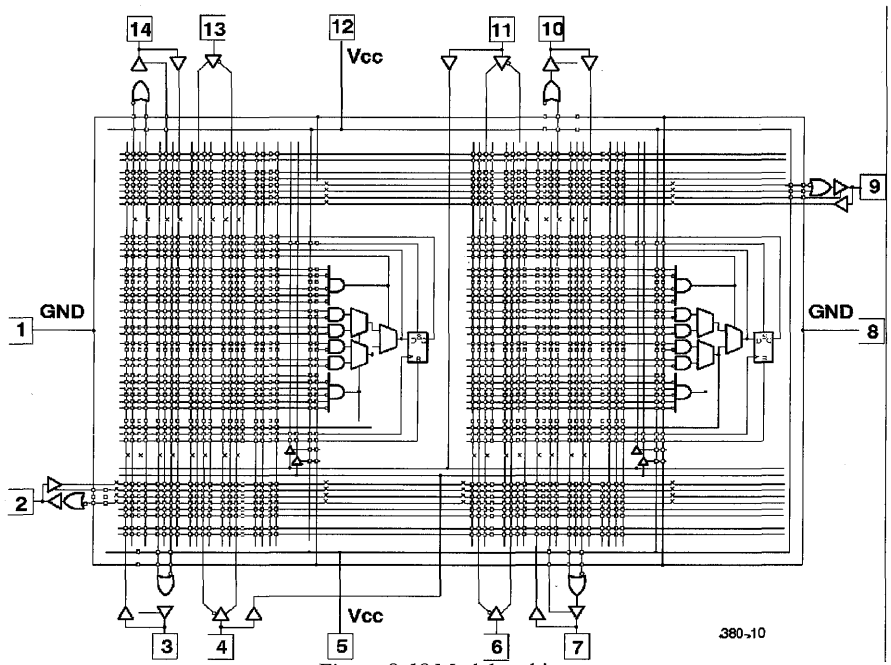


Figure 8-18 Model architecture

input OR gate, a fourteen-input AND gate (with half of the inputs inverted), a sum of three small products, a two-to-four decoder, or multiple functions at the same time, among other logic (see *Figure 8-19*). Each synthesis tool vendor or silicon vendor must create algorithms for mapping logic into FPGA device architectures.

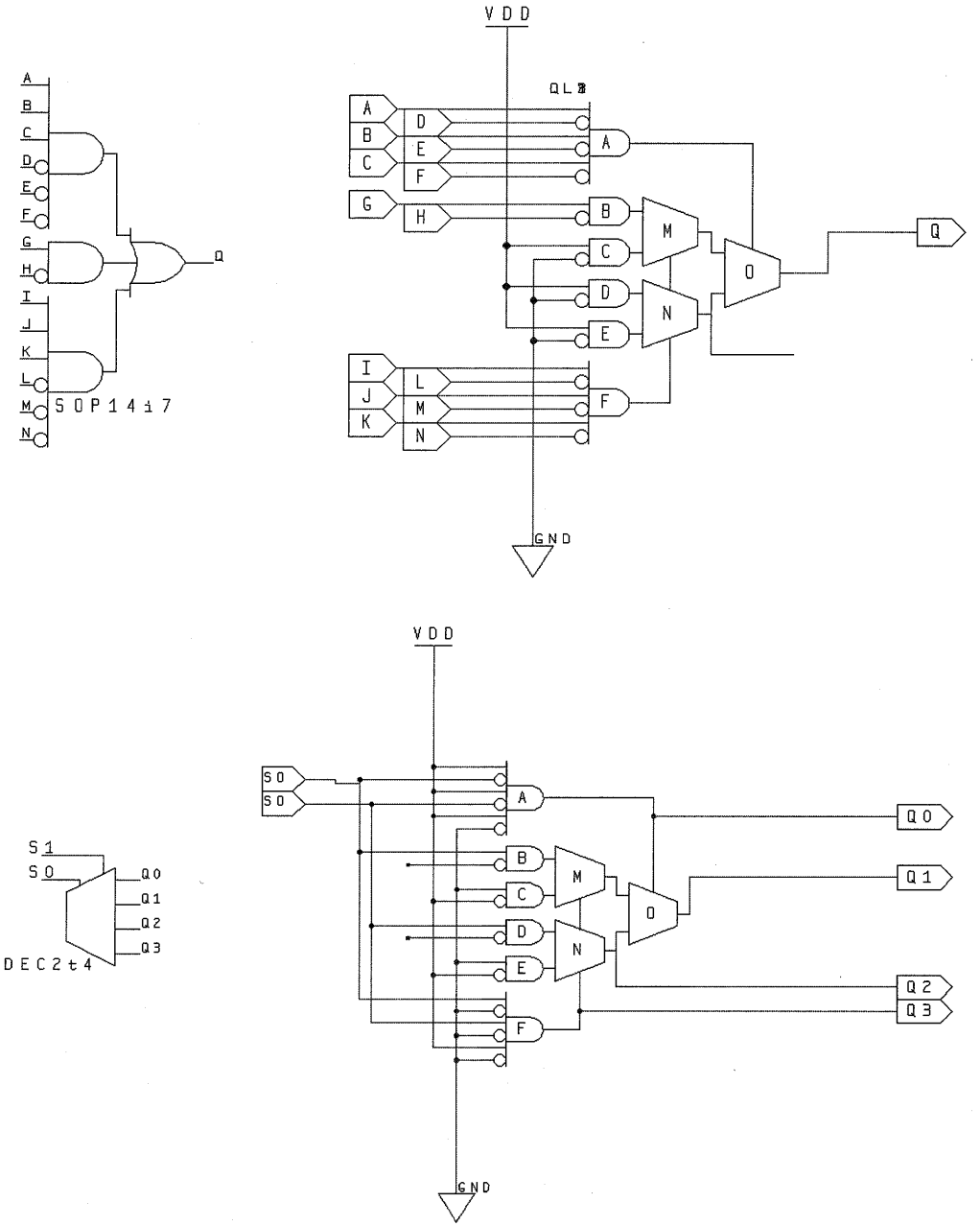


Figure 8-19 Sample logic functions implemented in a logic cell

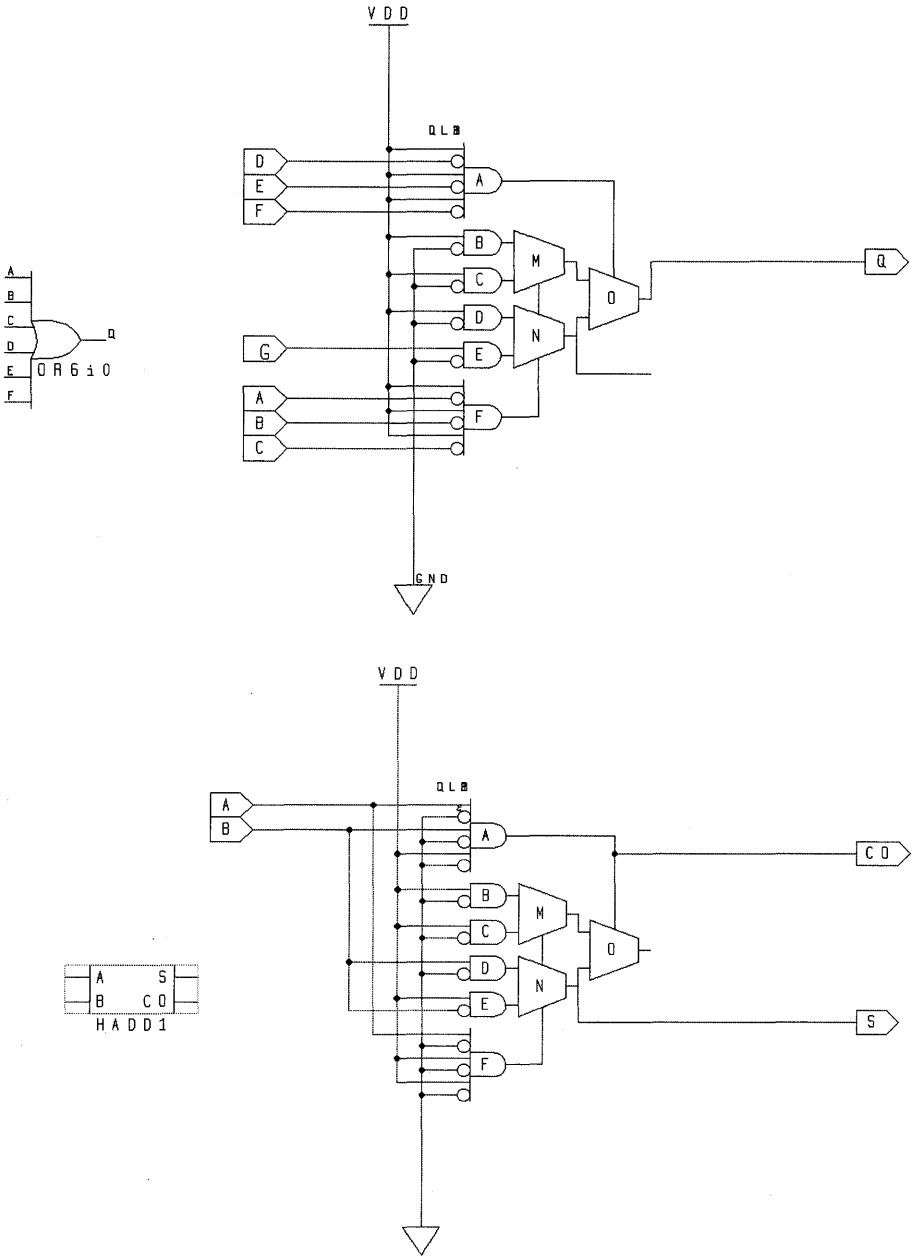


Figure 8-19 (continued)

Although one of the tasks of synthesis and place and route tools is to use logic resources efficiently, packing as much logic into the logic cells as possible may not produce the results that the designer wants. We pose the following question to make this point: Which of the implementations of a D-type flip-flop shown in *Figure 8-20* is optimal? (Obviously, we're assuming that there isn't any requirement for logic in front of the flip-flop.) The "optimal" implementation depends on the design requirements. Implementation (a) allows the six input AND gates to be used for other logic in the design (if the AND gates aren't required, then the inputs can be tied off to any value). Implementation (a) is the optimal choice unless you have a high performance requirement, in which case implementation (b) is optimal because the input to the flip-flop will be ready sooner. Implementation (a) requires that the flip-flop input fanin to four logic cell inputs. This greater load means that the signal rises and falls slower and is available at the input to the flip-flop later than it is with implementation (b).

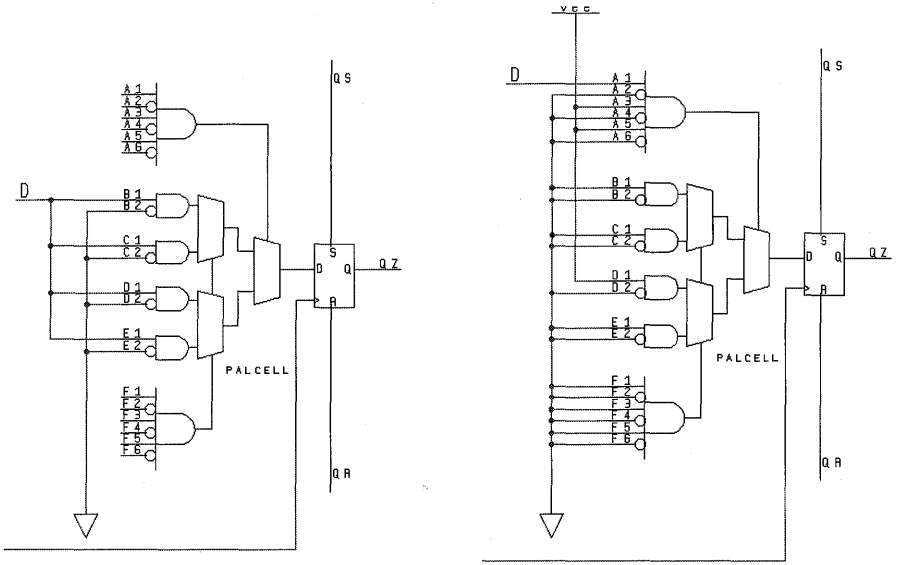


Figure 8-20 Multiple implementations of a D-type flip-flop

Propagation delays from logic cell to logic cell, I/O to logic cell, or logic cell to I/O depend not only on fanout (of the source) and the number of logic levels, but also on the available routing resources and how far the signals must route. This fact points to another difference between CPLDs and FPGAs: that predicting propagation delays before implementing a design in CPLDs is usually easier because delays are less dependent upon fanout, logic placement, and routing resources.

Design trade-offs

As a starting point, synthesis tools will often use just one algorithm to reduce logic so that it can be implemented in the fewest possible logic cells. This implementation is also usually the highest-performance solution—except where fanouts are excessively high or signals must travel a long distance—because there is a correlation between the total number of logic cells and the number of logic *levels* required to implement a function. (The number of logic levels refers to the number of

logic cells through which a signal must pass. A signal that must pass through many logic levels typically has a greater propagation delay than a signal that passes through few logic levels.) Adding levels of logic can, however, help to split the fanout through several buffers, but the additional propagation delay through a buffer must be made up by quicker rise times due to a smaller load per buffer. For example, consider 32 two-to-one multiplexers that have the same select line:

```
signal address: std_logic_vector(4 downto 0);
signal a, b, x: std_logic_vector(31 downto 0);
   ...
with address select
   x <= a when "10110",
         b when others;
```

Figure 8-21 shows five possible implementations for these 32 multiplexers. In *Figure 8-21* (a), the address lines are used in each logic cell where they are decoded to select one of the multiplexer inputs. In (b), the address lines are decoded once, and then this signal is used as the select line for the 32 multiplexers. In (c), the select line is decoded once, and then this signal is used as the select line for a couple of the multiplexers and as an input to several buffers that drive the select lines of the remaining multiplexers. In (d), the address lines are decoded multiple times from which the remaining select lines are driven. In (e), the select lines are decoded twice and these outputs are tied together to increase the drive of the multiplexer select lines. (This is a technique, called "double-buffering," that is allowed with the pASIC380 architecture, provided that the multiply driven signal is routed on an express or a quad wire; it is also a common technique used with gate arrays.)

You can see that even with such a simple example, there are several of ways to implement a design in an FPGA. Each implementation will produce different timing results, but many of them are comparable. Which is the optimal implementation? Well, that depends on what your design goals are for this piece of logic. Option (a) is probably not realistic or practical. It requires that the five address signals be routed to all of the logic cells. This doesn't gain anything. In fact, this implementation can eat up valuable routing resources that are better utilized by a critical signal. Option (b) is viable, but the select line will have a slow rise time and a fair amount of skew between the time it triggers the select input of the first and last multiplexers. If your design can operate under these conditions, then after considering the alternatives, you may choose to proceed with this implementation. Buffering the select line, as in options (c), (d), and (e), reduces the time it takes for all select lines to switch on transitions of *address*. Option (d) differs from (c) in that the select logic is replicated multiple times, and the times at which the select lines change relative to changes in *address* are closer together, whereas these times will be spread out in option (c). Option (d) is probably preferred over option (c), unless option (d) causes the lines to transition slower due to additional loading on *address*. Depending on whether or not the address signals must route elsewhere, using them as inputs to multiple buffers may increase the load on these lines such that the total propagation delay from the sources of the address lines to the multiplexer selection inputs is greater than the total propagation delay when the signals are buffered as in option (c). Option (e) may provide the best results, but it requires that a specific device resource be available.

With all of these options, which implementation does synthesis choose? Writing algorithms to try all the combinations of implementations for every unique netlist would be impossible. Fortunately, in architectures such as the pASIC380, many of the implementations are comparable, enabling synthesis software developers to reduce the number of algorithms for optimizing designs. Choosing the "correct" implementation is also a matter of understanding the design goals for a design or sub-design. Synthesis must be guided (by you) in these instances, as we'll discuss next. Because there are so many variables involved, one technique may prove more successful in one application. When you

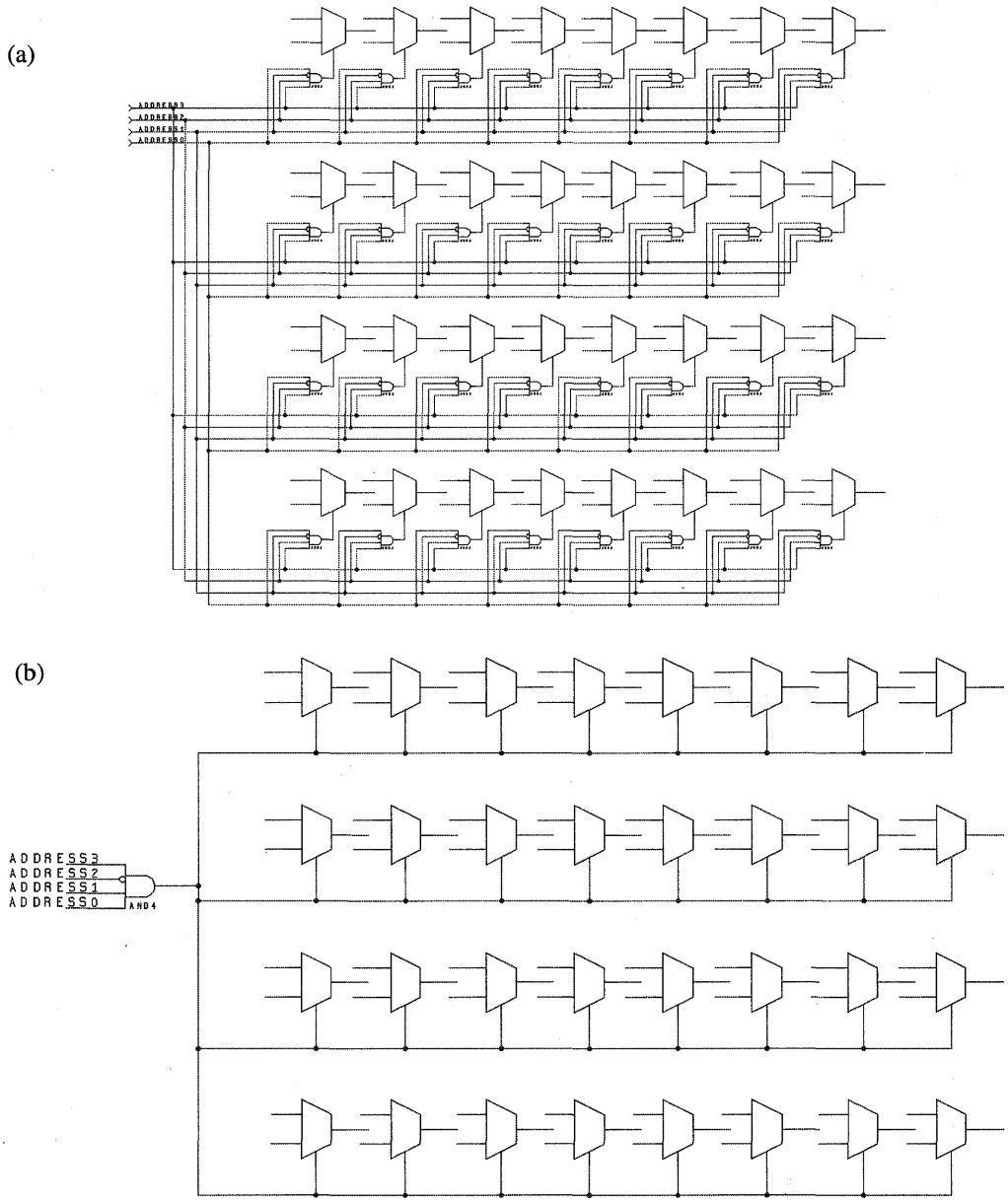


Figure 8-21 Managing signal loading with timing requirements

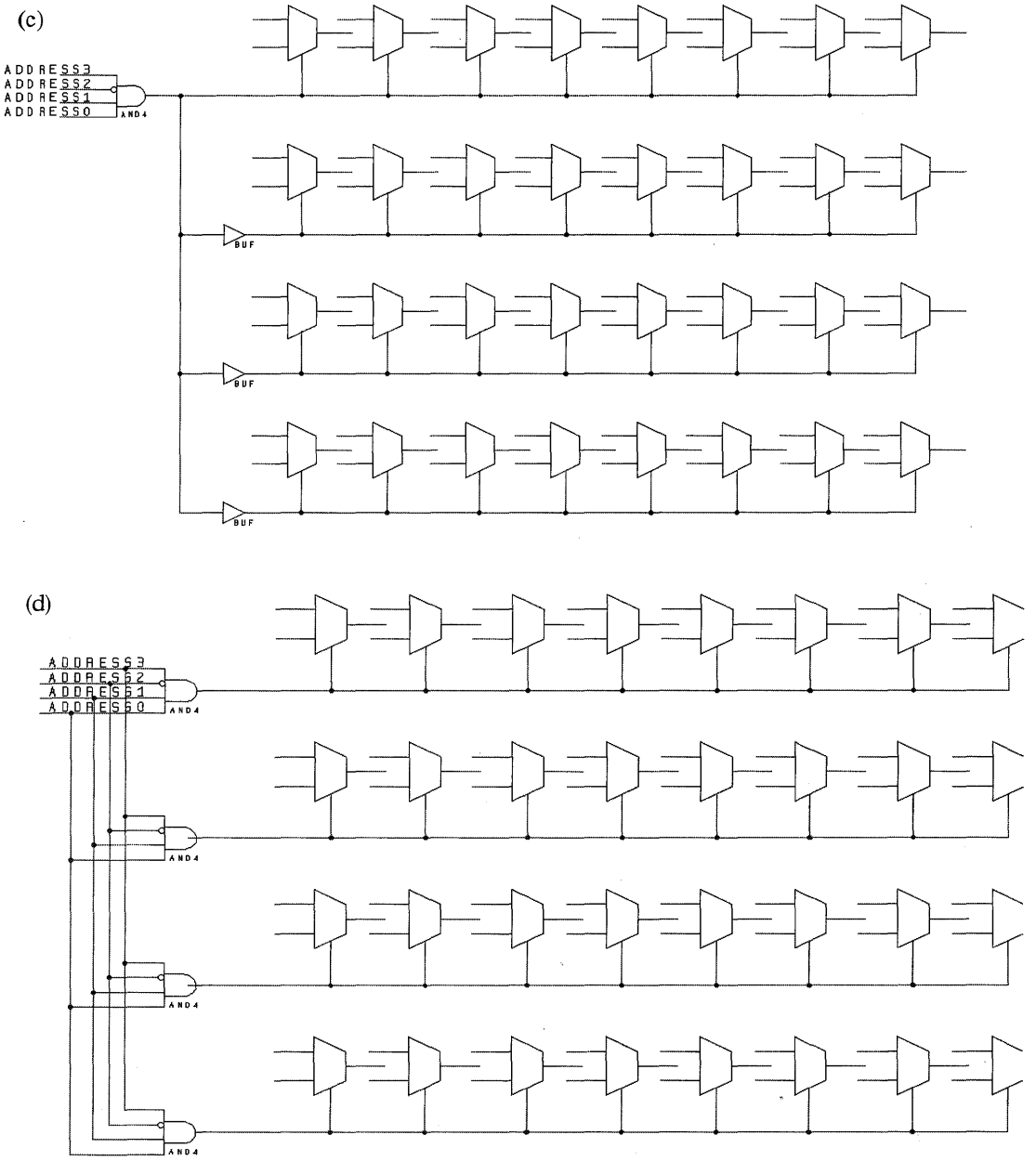


Figure 8-21 (continued)

(e)

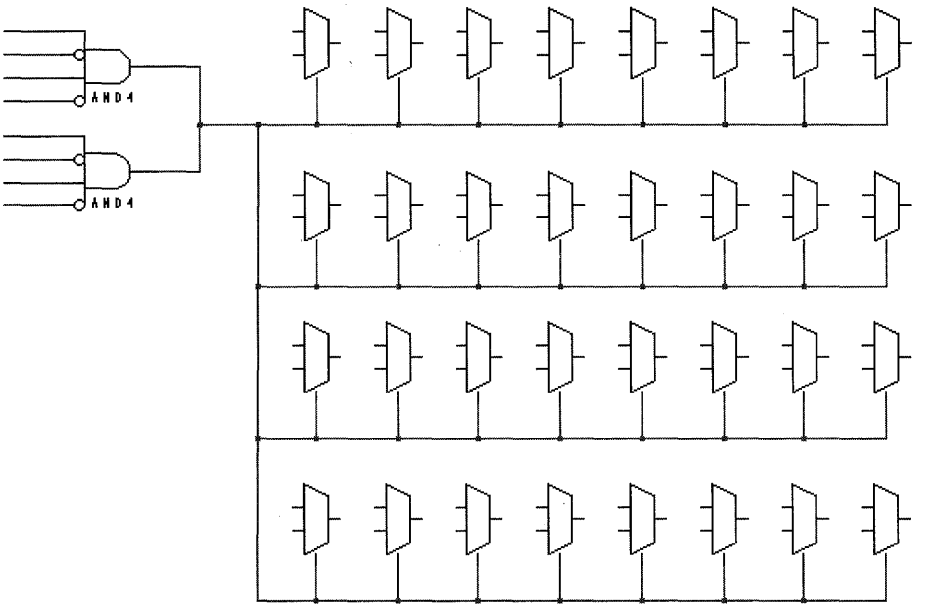


Figure 8-21 (continued)

push the upper limits of the technology, you may need to evaluate multiple implementations. Some of these implementations can be evaluated by the synthesis software and do not require user interaction. Other implementations may require you to change code, attributes, directives, command line options, or GUI switches.

Directive-driven synthesis

Mapping sum-of-products equations to the fewest number of logic cells may be a starting point for FPGA synthesis, but more sophisticated software will also allow you, the designer, to direct the synthesis process to achieve area or speed optimization globally or on particular portions of logic. Directives, often in the form of attributes, command line switches, GUI options, or a synthesis tool vendor's proprietary language, are most often used to control buffering of high fanout signals, automatic floor-planning, and operator inferencing (also referred to as module generation). We discussed fanout buffering above, and we will discuss automatic floor-planning and operator inferencing in the sections to follow. Some vendors include additional directives, but the directives that we will discuss will likely provide the greatest impact without convoluting the synthesis and place and route processes.

Automatic floor-planning

The "fitter" for an FPGA is a place and route tool. Whereas fitters for CPLDs often perform the design optimization and partitioning, place and route tools typically do little, if any, logic

optimization. The synthesis tool will convert the VHDL description into logic equations and map those equations into logic cells, specifying the interconnection of logic cells. That is, the synthesis tool will create a device-specific netlist (a netlist that can be directly mapped to the device architecture). The place and route tool must then place the logic cells and route them together and with the I/O to produce a design that meets the performance criteria. At this point, there is little that the place and route tool can do (besides pack unrelated logic into the same logic cells) to affect density. The synthesis tool, for the most part, dictates the number of logic cells that are required. One exception is the ability of place and route tools to automatically add buffers.

Placing and Routing

Place and route tools do have a large impact on performance, however, because propagation delays can depend significantly on how closely logic cells are placed to each other and which routing resources are used to connect the logic cells. This is because to route a signal a long distance requires a longer wire. This wire will have a larger total capacitance because of not only its length but also incremental fuse capacitance if additional wires must be connected.

Many place and route tools use a process called simulated annealing to determine how to place logic. In simulated annealing, the placer first places the logic cells (that are created and netlisted by the synthesis tool) semi-randomly within the array of logic cell locations. (With the CY7C381A, there are 96 logic cell locations in which the logic cells can be placed.) The placement is "semi-" random because logic cells used to capture inputs or propagate outputs are usually given preference for locations around the periphery of the logic cell matrix. The router determines the "cost" of routing with the semi-random placement. The cost is typically estimated by determining how far signals must travel and is a good determinant of speed. Next, the placer shuffles logic cells around, trying to reduce the overall cost. If an exchange or movement of logic cell placement increases the cost, then the logic cells are moved back to the original location. As long as the placer continues to make good progress in reducing the cost, the process goes on. At some point, the placer determines that it is asymptotically approaching an optimal placement. Usually depending on user settings for effort, the placer eventually settles on a solution and begins the routing process. Routers typically try to choose the type of routing resource that adds the least capacitance to a signal path.

Simulated annealing is quite successful with designs of 10K gates or fewer. However, with very large devices, the large number of logic cell locations and the exponential number of combinations of possible logic cell placements causes the simulated annealing approach to require too much time to settle on an appropriate solution. A better approach exists: floor planning.

Floor planning is based upon the assumption that large designs are typically broken up into functional units (e.g., a state machine, counter, comparator, controller, FIFO, etc.). An optimal placement is one that keeps these functional units close together (rather than shuffling them together and randomly placing the individual logic cells that make up these functional units). Functional units may be locally optimized if a bounding box is specified. The relative placement of the logic cells within these functional units may then be "frozen" and the functional units moved, as a whole, to the portion of the FPGA that makes most sense. Obviously, if the functional block interfaces with the chip I/O, then it should be placed near the periphery of the logic cell matrix. If it controls internal logic, then it will likely be best placed internal to the matrix. After the global optimization process places the functional units, routing may be performed.

With automatic floor-planning, units are identified as such (either because they are library components, as in a schematic, an inferred module, or because a user attribute has been added, indicating that certain signals are logically related and should therefore be placed in proximity to

each other). Automatic floor-planning can greatly reduce the randomness of results when placing and routing. It also enables the tool to come to optimal solutions quickly. Automatic floor-planning requires tight coupling between the synthesis and place and route processes.

Ideally, designers like to be able to specify the required operating frequency, setup time, and clock-to-output delay, and have the software tool synthesize, place, and route the design so that it meets those specifications. The processes are called timing-driven synthesis and timing-driven place and route. Floor planning is only an intermediate step in timing-driven place and route. It allows the designer to provide clues to the software based on information that the designer has that software algorithms may not be able to infer. As true timing-driven synthesis and place and route evolves, there will be less of a requirement for user intervention. Today's place and route technology is already at the state where placing and routing are usually fully automatic; at times, they require direction.

Operator inferencing

Operator inferencing is the process by which a VHDL synthesis tool infers an operation from a design description and produces an optimized component for that structure. This process is also referred to as module generation.

Two components can be inferred from the following VHDL code fragment:

```
if a=b then
   q <= q;
else
   q <= x + y;
end if;
```

The synthesis software can infer from this code that an equality comparator should be instantiated for "if a = b" and an adder for "q <= x + y;". Operator inferencing produces the logic shown in *Figure 8-22*. How is this different from not using operator inferencing to implement this design? Without module generation, the synthesis tool would create a boolean equation for each bit of q and then map that logic to the device architecture. Module generation enables arithmetic structures to be identified and hand-tuned, vendor-specific macros to be implemented. The implementation of a few arithmetic operators is examined later in this chapter.

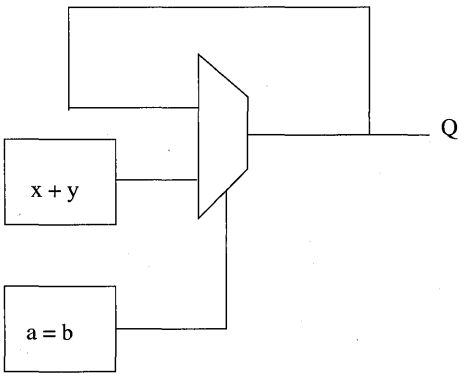


Figure 8-22 Example result of operator inferencing

Let's take another look at module generation. What logic do you suppose synthesis of the code in *Listing* 8-13 produces?

```
library ieee;
use ieee.std_logic_1164.all;
entity cnt16 is port(
   clk, rst:in std_logic;
   ld:    in std_logic;
   d:     in std_logic_vector(3 downto 0);
   c:     inout std_logic_vector(3 downto 0));
end cnt16;

architecture archcnt16 of cnt16 is
begin
counter: process (clk, rst)
   begin
      if rst = '1' then
         c <= (others => '0');
      elsif (clk'event and clk='1') then
         if ld = '1' then c <= d; else c <= c + 1; end if;
      end if;
   end process counter;
end archcnt16;
```

Listing 8-13 Counter design; how does operator inferencing help with the implementation of this design?

Does this code cause the compiler to infer, or identify, a 16-bit loadable counter? Most probably not. To identify this as a 16-bit loadable counter requires more than operator inferencing; it truly requires a robust module generator. We are not aware of any compilers or synthesis tools that recognize such code as a 16-bit counter (i.e., that provide this level of module inferencing). One reason for the lack of such tools is that VHDL permits a designer to write numerous possible constructs from which it is extremely difficult for a tool to pick out regular components. Certainly a tool can publish a list of recognizable constructs, but this limits VHDL coders to strict templates, defeating the purpose of a high-level description language. Such templates are not standard across tools. Instead of interpreting the code of *Listing 8-13* as a 16-bit loadable counter, most synthesis tools recognize the +1 adder (incrementer), implementing the logic as in *Figure 2-22*, and then optimizing the logic as necessary.

We'll use the code of Listing 8-14 to illustrate the difficulty of inferring the optimal logic from a high- level description:

```
library ieee;
use ieee.std_logic_1164.all;
entity wierdcnt16 is port(
   clk, rst:in std_logic;
   en,up,by2,by3,by5:in std_logic;
   d:       in std_logic_vector(3 downto 0);
   c:       inout std_logic_vector(3 downto 0));
end wierdcnt16;
```

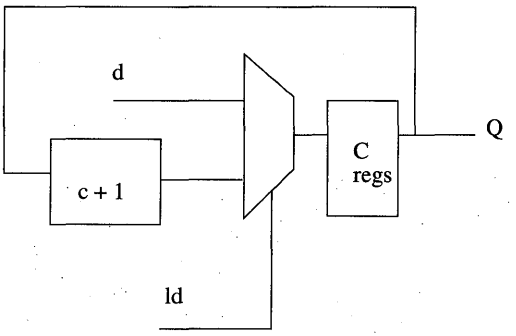


Figure 8-23 Implementation resulting from operator inferencing of a loadable counter

```
architecture archwierdcnt16 of wierdcnt16 is
begin
counter: process (clk, rst)
   begin
      if rst = '1' then
         c <= (others => '0');
      elsif (clk'event and clk='1') then
         if en = '1' then
            if up = '1' then
               if by2 = '1' then
                  c <= c + 2;
               elsif  by3 = '1' then
                  c <= c + 3;
               elseif by5 = '1' then
                  c <= c + 5;
               else
                  c <= c + d;
               end if;
            elsif by2 =  '1' then
               c <= c - 2;
            elsif by3 = '1' then
               c <= c - 3;
            elsif by5 = '1' then
               c <= c - 5;
            else
               c <= c - d;
            end if;
         else
            c <= c;
         end if;
      end if;
   end process counter;
```

```
end archwierdcnt16;
```

Listing 8-14 A unique counter: Operator inferencing can identify modules but not the optimal hand-tuned construct.

No compiler/synthesis tool today or in the near future will be able to produce the optimal implementation for the wierdcnt16 of *Listing 8-14* that is achievable by manual design. Instead, operator inferencing produces an implementation similar to that in the example above. That is, the *+2, +3, +5, +d, -2, -3, -4,* and *-d* adders and subtracters would be inferred and the results multiplexed based on the values of *en, up, by2, by3,* and *by5.*

Previously, we mentioned that some synthesis tools use directives with operator inferencing. These directives, sometimes in the form of attributes, are used to direct the implementation of a module. In other words, there may be multiple possible implementations for a 16-bit +5 adder. You may want this component to be optimized to be area efficient, speed efficient, or a balance of the two. Directives can provide that level of control, enabling you to control the critical and noncritical portions of a design. The code of *Listing 8-14*, as ludicrous as it may seem, is instructive: the 16-bit adder for "c + d" is the most complex logic, so the critical path in this circuit will be from the c registers through this adder and back to the input of the c registers. Paths through the other adders will not be nearly so complex. Therefore, while you may want to direct this adder to be speed optimized in order to reduce the delay of the critical path, you will likely want to optimize the other adders to be area efficient. Because they are not in the critical path, creating speed-optimized components would be of no advantage.

Arithmetic Operations

The optimal implementation of an arithmetic component such as an adder, subtracter, magnitude comparator, or multiplier is device dependent. An implementation strategy that produces an area efficient and high performance solution for one architecture may not work well for another because of the differences in macrocell or logic cell resources. We will examine 8-bit adder circuits for the FLASH370 and pASIC380 architectures to identify differences in implementation.

With both CPLD and FPGA architectures, there may be multiple implementations that trade off area efficiency for performance. Most synthesis tools use module generation in conjunction with directives to choose an area-efficient or high-performance implementation.

Module generation typically overrides operators defined in functions. For example, the following code is from *Listing* 7-11. This function overloads the + operator for bit_vector operands.

```
-- "+"
--   Add overload for:
--   In:     bit_vectors.
--   Return: bit_vector.
--
FUNCTION "+" (a, b   : BIT_VECTOR)       RETURN BIT_VECTOR IS
    VARIABLE s       : BIT_VECTOR (a'RANGE);
    VARIABLE carry   : BIT;
    VARIABLE bi      : integer;       --  Indexes b.
BEGIN
    carry     := '0';

    FOR i IN  a'LOW TO a'HIGH LOOP
```

```
    FOR i IN  a'LOW TO a'HIGH LOOP
        bi := b'low + (i - a'low);
        s(i)  := (a(i) XOR b(bi)) XOR carry;
        carry := ((a(i) OR b(bi)) AND carry) OR (a(i) AND b(bi));
    END LOOP;

    RETURN  (s);
END "+";                                -- Two bit_vectors.
```

An addition operation that uses this function for the implementation results in both an area-inefficient and low-performance solution for both CPLDs and FPGAs. Take, for example, the operation x <= a + b where *a* and *b* are bit_vectors. This implementation results in equations for x_0 and x_1 as follows:

$$x_0 = a_0 \oplus b_0$$
$$x_1 = a_1 \oplus b_1 \oplus a_0 b_0$$

For a CPLD with a macrocell that does not contain an XOR gate, the equations must be expanded as follows:

$$x_0 = \overline{a_0} b_0 + a_0 \overline{b_0}$$
$$x_1 = \overline{a_1}\,\overline{a_0} b_1 + a_1 \overline{a_0}\,\overline{b_1} + \overline{a_1} b_1 \overline{b_0} + a_1 \overline{b_1}\,\overline{b_0} + \overline{a_1} a_0 \overline{b_1} b_0 + a_1 a_0 b_1 b_0$$

As the size of *a* and *b* increases, the expansion of the XOR results in an exponential increase in product term requirements. The large number of product terms requires not only several levels of logic but also several macrocells in order to split product terms over multiple levels. At some point, it is better to implement the adder as a ripple-carry adder.

The full adder, as shown in *Figure 8-24*, is the basic building block of a ripple adder. The carry-out of one full-adder is used as the carry-in of the next stage of the adder (*Figure 8-25*). Rather than expanding the expression for the carry, the carry expression is forced to a macrocell. This implementation limits the number of product terms that are required (and therefore also the number of macrocells), but it increases the propagation delay. One level of logic is required for each bit of the adder. Further trade-offs between performance and area can be made with ripple adders by using 2-bit and 3-bit group adders (*Figure 8-26*). Group adders limit the number of carry terms, and, hence, increase the levels of logic and propagation delay (as long as the sum and carry terms do not require multiple passes).

An alternative to ripple-carry adders is carry look-ahead adders (*Figure 8-27*). Group adders are created with carry generate terms and carry propagate terms. A carry generate term indicates that the group has created a carry into the next group. A carry propagate term indicates that the group will produce a carry into the next group if there is a carry into the current group.

For the pASIC380 architecture, the ripple carry adder may be an area-efficient adder, but a carry select adder provides the best performance (see *Figure 8-28*).

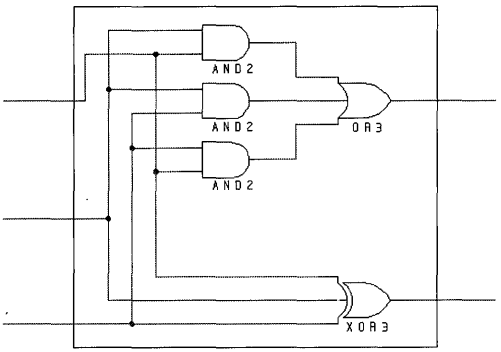


Figure 8-24 A full adder

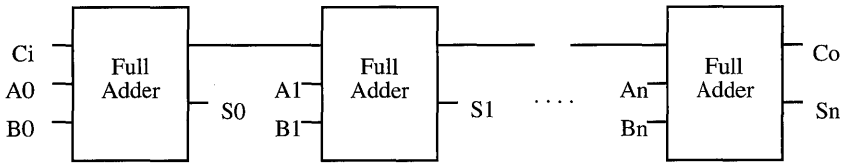


Figure 8-25 A ripple adder

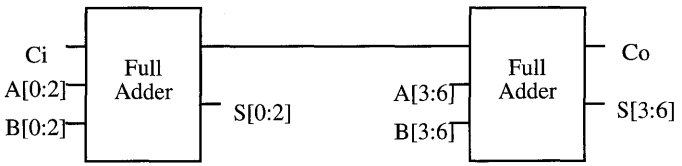


Figure 8-26 2-bit and 3-bit group adders

Although it may be obvious, it is worth reiterating that an optimal solution is based on an area or performance goal and is device dependent.

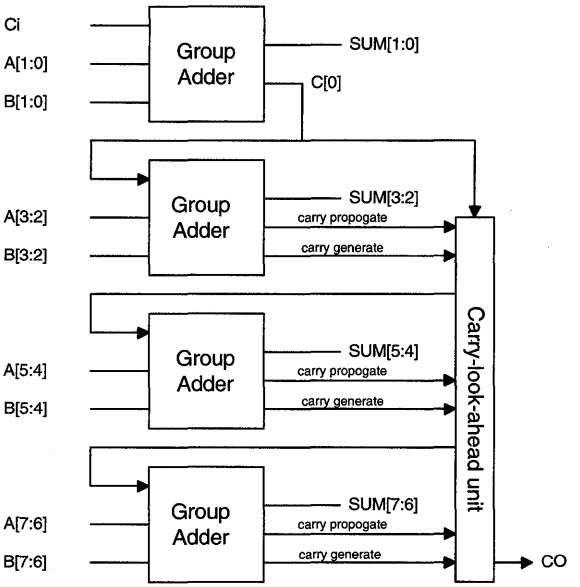


Figure 8-27 Carry look-ahead adder

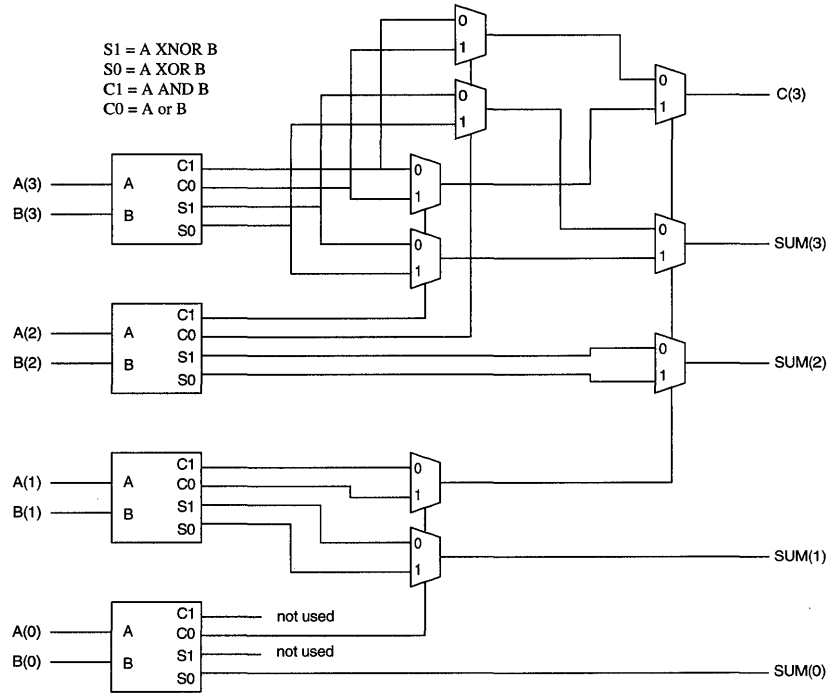


Figure 8-28 Carry select adder

Implementing the Network Repeater in an 8K FPGA

Load limiting is perhaps the most prevalent design issue for designers using any FPGA. In the design of the network repeater, several signals have high fanouts. Automatic or directive-driven buffer generation can alleviate some of the performance problems associated with fanout. Buffer generation can be a step in the synthesis or place and route process as long as the interface between the processes is well defined. The following is a list of buffers created for the repeater design if the maximum load is set at 13:

```
-----------------------------------------------------------
Begin Buffer Generation.
-----------------------------------------------------------
[max_load = 13, fanout = 51] Created 3 buffers [Duplicate] for 'rxclk'
[max_load = 13, fanout = 18] Created 1 buffers [Duplicate] for 'WFAC17'
[max_load = 13, fanout = 18] Created 1 buffers [Duplicate] for 'WFAC18'
[max_load = 13, fanout = 18] Created 1 buffers [Duplicate] for 'WFAC19'
[max_load = 13, fanout = 18] Created 1 buffers [Duplicate] for 'WFAC20'
[max_load = 13, fanout = 18] Created 1 buffers [Duplicate] for 'WFAC21'
[max_load = 13, fanout = 18] Created 1 buffers [Duplicate] for 'WFAC22'
[max_load = 13, fanout = 18] Created 1 buffers [Duplicate] for 'WFAC23'
[max_load = 13, fanout = 18] Created 1 buffers [Duplicate] for 'WFAC24'
[max_load = 13, fanout = 25] Created 2 buffers [Normal    ] for 'carrier'
[max_load = 13, fanout = 15] Created 2 buffers [Normal    ] for 'u4_symbolcount_0'
```

```
[max_load = 13, fanout = 19] Created 2 buffers [Normal   ] for 'u5_state3SBV_0'
[max_load = 13, fanout = 15] Created 2 buffers [Normal   ] for 'u5_state3SBV_1'
[max_load = 13, fanout = 16] Created 2 buffers [Normal   ] for 'u5_state3SBV_2'
[max_load = 13, fanout = 48] Created 4 buffers [Normal   ] for 'u6_collisiond'
[max_load = 13, fanout = 20] Created 2 buffers [Normal   ] for 'collision_OUT'
```

While buffering may solve the performance problem for many signals, there may be critical portions of a circuit that require further optimization to ensure that performance objectives are met. Introducing pipeline stages may be required to reduce the number of levels of logic between registers and maintain performance. The outputs of the arbiter have been pipelined because the collision and carrier signals are used in several design units and propagate through several levels of logic if the pipeline is absent. In a design such as that of the repeater, pipeline registers may be added for some signals (e.g., *jabber_bar, partition_bar*) without having to add pipeline registers for the others to keep synchronization if the synchronization is not important. In the case of *jabber_bar*, the signal can be asserted for a range of counter values, so adding a pipeline register will not affect the function of the design.

The common outputs (e.g., *txdata, idle, jam*) for the core controller state machines are all generated from the third state machine. This unnecessarily places a higher load on the state registers for the third state machine. Since the common outputs can be decoded from any of the state machines and all state machines must run at 25 MHz, it may make sense to balance the loading of all state machine registers. Of course, the state machines should be one-hot designed for maximum performance. Although, a performance comparison of a fault-tolerant, one-hot encoded state machine versus a sequentially encoded state machine may prove that one is not better than the other.

Special purpose pads can aid in the performance of a design. The pASIC380 has two clock pads and six high-drive pads. Signals assigned to a clock pad make use of a high-performance, low-skew clock distribution tree. Clocks on one of these trees can be propagated to all logic cells in about 5 ns with a skew of about 1 ns. A system-wide clock that does not use a clock distribution tree has a considerably larger distribution time. With this design, the system clock is the transmit clock. The clock distribution tree may also be used for the sets and resets of flip-flops. The reset in this design has a fanout of 343, so the clock distribution tree is the best choice.

High-drive inputs provide about twice the internal input driving current of normal I/Os configured as inputs. High fanout signals coming from off chip should make use of these special purpose pads. Following is the list of automatic pad selection:

```
------------------------------------------------------------
Begin PAD Generation.
------------------------------------------------------------
Created CLKPAD for signal 'reset'
    Above signal drives  0 Clocks, 343 Set/Resets. Total = 343
Created CLKPAD for signal 'clk'
    Above signal drives 294 Clocks,  0 Set/Resets. Total = 294
    And   1 other inputs (active high).
    Above signal consumed 1 express wire
Created HD1PAD  for signal 'rxd5'
    Above signal drives  0 Clocks,  0 Set/Resets,  8 other inputs. Total =  8
Created HD1PAD  for signal 'rxd4'
    Above signal drives  0 Clocks,  0 Set/Resets,  8 other inputs. Total =  8
Created HD1PAD  for signal 'rxd3'
    Above signal drives  0 Clocks,  0 Set/Resets,  8 other inputs. Total =  8
Created HD1PAD  for signal 'rxd2'
```

```
    Above signal drives  0 Clocks,  0 Set/Resets,  8 other inputs. Total =  8
Created HD1PAD  for signal 'rxd1'
    Above signal drives  0 Clocks,  0 Set/Resets,  8 other inputs. Total =  8
Created HD1PAD  for signal 'rxd0'
    Above signal drives  0 Clocks,  0 Set/Resets,  8 other inputs. Total =  8
```

Gated clocks are used with the FIFO. To ensure a small lock delay and clock skew, floor planning is likely required for the FIFO. Simulated annealing may not find the optimal placement for the FIFO registers. The optimal placement of the flip-flops making up the FIFO is one on top of the other (because outputs of the pASIC logic cell feed back to the left side of the logic cell where the vertical routing channel is). Placing each of the registers (six flip-flops) in a vertical column ensures that the gated clocks need to route to only one column per register.

Resource estimations are somewhat more difficult to make for FPGAs than for CPLDs because the logic cells contain less logic on average than a macrocell and its associated product terms. An estimation of the logic cell count follows:

- clockmux8 (37)
  - 27 for registers and the enable equations
  - 9 for the clock selection
  - 1 for rxclk
- arbiter8 (13)
  - 11 for registers
  - 1 for the *collision* signal
  - 1 for the *carryin* signal
- FIFO (110)
  - 48 for eight 6-bit registers for the FIFO
  - 6 for the counters (read and write pointers)
  - 48 for six 8-to-1 FIFO output multiplexers
  - 8 for the decoding of the write pointer
- symbolmux (24)
  - 3 for the symbol counter
  - 6 for the output registers
  - 12 for the output multiplexers (2 for each line)
  - 3 for *symbolend* signals
- core controller (47)
  - 4 for synchronization of inputs
  - 3 for synchronization of internal signals
  - 10 for the counter
  - 25 for the three state machines
  - 5 for the outputs
- port controller (8 x 40 = 320)
  - 6 for synchronization of inputs
  - 2 to synchronize internal signals
  - 19 for the three counters
  - 8 for the one-hot state machine
  - 5 for output signals

The estimation total is 551 logic cells, not accounting for additional logic cells needed for buffering. After place and route, the total logic cell count is listed as 604, which is within reach of our

estimation. Performance is difficult to estimate. The post-place-and-route, worst-case delays for register-to-register operation is shown in Figure 8-29. The worst-case delay of 36.5ns will support 40 MHz operation.

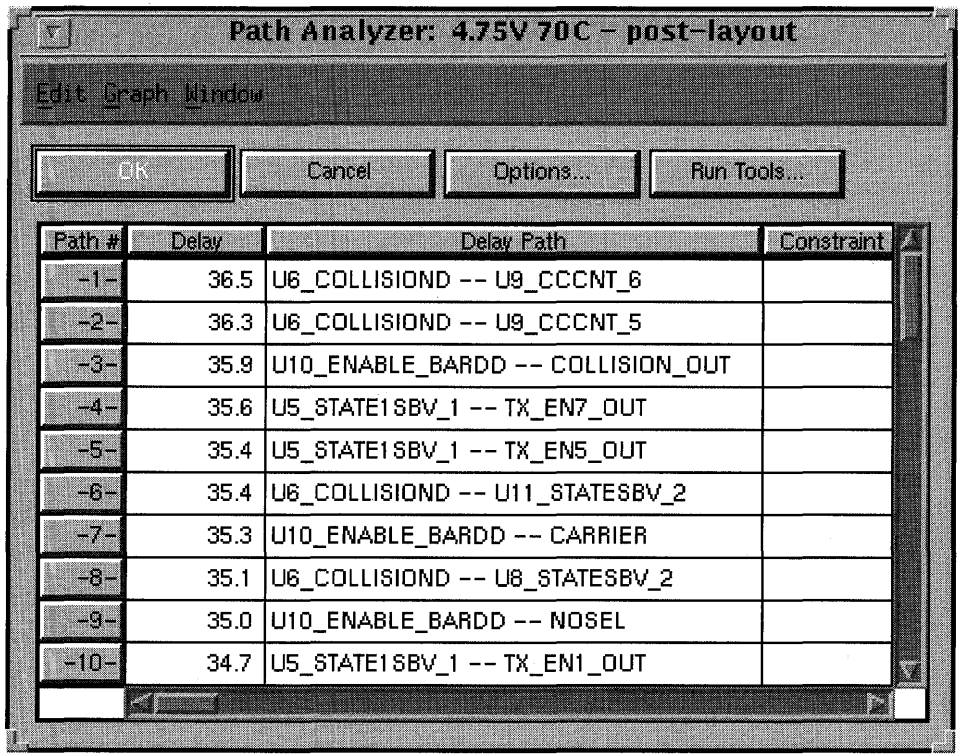


Figure 8-29 Worst-case register-to-register delays

In the next chapter, we will create a test bench to simulate a model of the circuit produced from the place and route software that models the design with timing information.

Preassigning pinouts

At the beginning of our FPGA case study, we implied that there are many fewer problems with fitting designs in FPGAs. While this is generally true because there are usually fewer restrictions on how device features are used, some FPGAs are better than others in this respect, and this affects whether or not you will want to assign the pinout so that you can begin your board layout to reap the benefits of concurrent engineering. Always check vendor claims before pursuing a board design before you have placed and routed your design in the FPGA. Antifuse FPGAs tend to have better routing ability because more routing wires can be added at a lower cost to the vendor. This results in devices that are inherently very routable. Refer to chapter 2, "Programmable Logic Primer" for a discussion of different FPGA technologies.

To use a CPLD or FPGA?

Different designs perform better in one device architecture over another. In general, register-intensive designs are best suited to FPGAs, but the best way to determine which device architecture works best is to synthesize the logic for both architectures and compare the results. The important thing to keep in mind is the available resources in a device and how these resources can be used.

Exercises

1. Determine if the following designs can fit ina 22V10:
   (a) an 8-bit loadable counter
   (b) an 11-bit counter
   (c) a 4-bit counter that asynchronously resets to 1010
   (d) the memory controller of chapter 5?

2. In *Figure 8-1*, if the second macrocell from the top is allocated eight product terms, how many unique product terms may be allocated to the macrocell directly above and below?
   (a) What if 10 product terms were allocated to the second macrocell?
   (b) 12 product terms? (c) 16 product terms?

3. The OE product terms in a CY7C371 do not have polarity control. How would (*a* AND *b*) and (*a* OR *b*) differ as enabling conditions when fitted? Compare resource usage and timing.

4. Implement a 12-bit adder in the CY7C371 and the CY7C381. In both cases, optimize for speed, and then optimize for area. Compare all of the results.

5. Design the 28-bit loadable counter described below in the CY7C371.

Implementing a 28-bit loadable counter in a 32-macrocell device that can have 36 inputs into each logic block can present a design obstacle. Typically, counters are implemented with T-type flip-flops. The least significant flip-flop toggles every clock cycle, and all subsequent bits toggle only when all of the lesser significant bits are a "1" (for example, the count transitions to "1000" only when all lesser significant bits are "1," as in "0111"). Thus, the most significant (the 28th) bit of a 28-bit counter will toggle only when all of the lesser significant bits are a "1." To determine if all are a "1," an AND gate is used. But you will not be able to simply place the 16 least significant counter bits in one logic block and the remaining twelve in the other logic block. This would require too many inputs to the second logic block: 28 for the counter bits, another 12 for the load inputs, and a few others for control—more than the allowed 36 inputs. However, the counter *can* still fit into the device, but (1) only the first 15 bits of the counter can be implemented in the first logic block and (2) a token must be passed from the first logic block to the second, indicating to the second logic block when the AND of the first 15 bits is a "1" (see *Figure 8-30*). This is, in effect, a cascading of AND gates wherein a 15-input AND gate in the first logic block is cascaded with other AND gates (a 12-input AND gate for the most significant bit) to enable the toggling of the upper 13 counter bits. This token passing scheme in which AND gates are cascaded requires two passes through the product term array. The delay associated with the second pass can be eliminated by producing a token that is registered and for which a '1' is on the output of the flip-flop one clock cycle before all bits are a '1.'

6. Discuss the merits and demerits of using the attribute 'synthesis_off'.

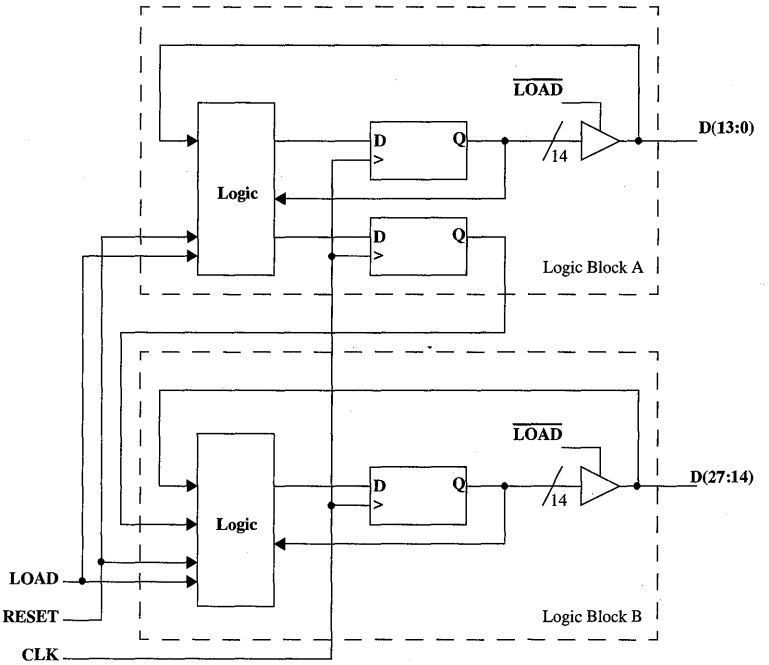


Figure 8-30 A 28-bit counter must use a token (cascaded AND gate) so as not to require more than 36 inputs to the logic block.

7. Between the three issues a designer has to keep in mind when targeting a VHDL code to a specific device, which do you think is of prime importance? Justify. Add some other concerns one should be aware of?

8. What are the typical resource constraints that a designer has to keep in mind while targeting CPLDs?

9. What are the typical resource constraints that a designer has to keep in while targeting FPGAs?

10. What are the typical issues that a designer has to keep track of, when migrating a design in a CPLD to a FPGA and vice versa?

11. How would you implement your counters in CPLDs to bring down your Product term count?

12. Discuss the tradeoffs you would consider in using a synchronous or an asychronous reset for state machines.

13. Would a CPLD fitter handle a design better if:

a) Signals are assigned to specific pins and specific buried macrocells

b) Signals are assigned to specific pins and buried macrocell assignments are floated

c) Signal assignments are floated and buried macrocell assignments are specified

d) Some of the signals are assigned to pins and buried macrocells and others are floated

e) Pin and buried macrocell assignments are floated

14. Given that the final implementation of a design is device specific, discuss the issues involved in the portability of VHDL designs over different FPGA and CPLD vendors. What all should you ensure to make your code portable across different vendors?

15. Create a scheme of your own to achieve product term (asynchronous) clocking in FLASH 370 CPLDs.

16. If you were to write a place and route tool for the pASIC FPGAs, what constraint swould you be dealing with? How would you prioritize them?

9 Creating Test Fixtures

The focus of this text has been to assist readers in writing VHDL code that can be efficiently synthesized for use with programmable logic. The intended audience has been those interested in VHDL as a *design* language, rather than those interested in VHDL as a language for modeling devices and systems. (However, much of the content of the previous chapters is directly applicable to modeling.) Nonetheless, the code that a designer writes *can* be used as a model—it is a model of the functionality of the design. Because the code is a model, it can be simulated with VHDL simulation software.

The ability to simulate a VHDL model can greatly increase design efficiency: It allows for the functional verification of a design before synthesis and place and route. The synthesis and place and route processes can consume anywhere from less than a minute to several hours, depending upon the size of the design and efficiency of the software. Because the initial synthesis and place and route may not yield the required performance and resource utilization goals, subsequent runs of the synthesis and place and route software may be required. If functional verification is not completed until after iterations of synthesis and place and route, then the time spent may have been wasted—a design error (especially one that significantly changes the design) may require further iterations of synthesis and place and route. Simulation of a VHDL model can bring out design errors at a much earlier stage of the design process, allowing design errors to be corrected *before* synthesis and place and route.

Most VHDL simulators (i.e., VHDL simulation software) allow real-time interaction—values of inputs can be assigned, simulation time can be executed, and the values of outputs can be inspected by looking at waveforms. This cycle can be repeated until the designer is satisfied that the model functions as expected. Alternatively, a test fixture (test bench) can be used to verify the design's functionality. A test fixture allows input test vectors to be applied to the design (unit under test) and output test vectors to be either observed (by waveform) or recorded in an output vector file. Test fixtures provide advantages over interactive simulation: (1) A test fixture allows the input and output test vectors to be easily documented. (2) This in turn provides a more methodical approach than relying on interactively entering and inspecting test vectors. (3) Once the test fixture has been built and the test vectors defined, the same functional tests can be repeated during iterations of design changes. That is, little time is required after a design change to rerun tests. (4) The same test fixture used to verify the functionality described in the VHDL source code can be used to verify the functionality and timing described by a post-fit model, as described further in the following paragraph.

Many fitters and place and route tools produce VHDL models of a device with a given fit or place and route. These models are representations of a design as fitted in the device architecture. The actual model bears little resemblance to the code written by the designer (the source code). Instead, these models typically consist of component instantiations of device architecture features and signals connecting the components. The models also usually have timing information so that a simulation run can detect setup violations, and outputs can be observed to propagate according to the device AC timing specifications. These models will have the same I/O as the original source code, so the same test fixture used to evaluate the functionality of the design source code can be used to verify both the functionality and the timing of the post-fit model.

Creating a Test Fixture

We will create a test fixture for use with the 3-bit counter of *Listing 9-1*.

```
library ieee;
use ieee.std_logic_1164.all;
package mycntpkg is
  component count port(clk, rst: in std_logic;
                       cnt:      inout std_logic_vector(2 downto 0));
  end component;
end mycntpkg;

library ieee;
use ieee.std_logic_1164.all;
entity count is port(clk, rst:  in std_logic;
                     cnt:       inout std_logic_vector(2 downto 0));
end count;

use work.std_math.all;
architecture archcount of count is
begin
counter: process (clk, rst)
begin
        if rst = '1' then
                cnt <= (others => '0');
        elsif (clk'event and clk= '1') then
                cnt <= cnt + 1;
        end if;
end process;
end archcount;
```

Listing 9-1 Source code of a 3-bit counter

The entity/architecture pair for *count* must be declared as a component so that it can be instantiated as the unit under test in the test fixture.

Listing 9-2 is the text fixture that can be used with the source code and with the post-fit model. In this listing, we include the test vectors for the unit under test in the source code of the test fixture.

The entity declaration for *testcnt* does not include any ports because there are not any inputs or outputs to the test fixture—it is self-contained. Signals are declared for each port of the unit under test. We choose signal names that match the formal signal names of the component. The type *test_vector* is defined as a record. Each *test_vector* has elements for the *clk*, *rst*, and *cnt* signals. The type *test_vector_array* is defined as an array of *test_vector*. A constant, *test_vectors*, is defined to be of the type *test_vector_array*, and its value defines the set of test vectors to be applied to the unit under test as well as the expected output.

The *count* component is instantiated as the unit under test. Inside a process, a loop is used to sequence through the *test_vectors* array. For each vector in the array, the clock and reset stimulus are assigned to the *clk* and *rst* signals. Because the assignments are to signals, the assignments are not immediate. Rather, the signal assignments are scheduled. The signals assume the values of the present value of the variables only at the end of the process or if any simulation time transpires. In this case, the very next statement calls for 20 ns of simulation time to elapse. But before any

simulation time elapses, all signals that are not explicitly initialized are by default initialized to the 'LEFT value (for std_logic that's the 'U', uninitialized, value), and signals are evaluated. Thus, the *counter* process is executed once before any simulation time: Because neither the IF nor ELSIF condition is true, the value of *cnt* remains "UUU" (all array elements uninitialized). The statement, "wait for 20 ns;" causes the *clk* and *rst* signals to assume the values that were scheduled after 0 ns of simulation time elapses. The changes in values for *clk* and *rst* cause the *counter* process to execute once again. This time *cnt* is assigned "000" as a result of *rst* being '1'. Twenty nanoseconds elapse, and then the value of *cnt* is compared with the expected result (*vector.cnt*). If *cnt* is not its expected value, then the assertion statement causes the simulation software to issue a report: "cnt is wrong value". The report is issued if the assertion statement is false. The assertion is hard-coded as false, because the comparison of *cnt* to the test vector was accomplished with an IF-THEN construct. An unexpected value of *cnt* also causes the variable *errors* to be asserted. The loop continues for the remainder of the vectors following the sequence of events described above with the exception of initialization. Once the loop has ended, two mutually exclusive assertion statements are evaluated. If the value of *errors* is false, then the report "Test vectors passed." is issued; otherwise, the report "Test vectors failed." is issued.

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

entity testcnt is
end testcnt;

use work.mycntpkg.all;
architecture mytest of testcnt is
  signal clk, rst: std_logic;
  signal cnt: std_logic_vector(2 downto 0);
  type test_vector is record
        clk: std_logic;
        rst: std_logic;
        cnt: std_logic_vector(2 downto 0);
  end record;
  type test_vector_array is array(natural range <>) of test_vector;
  constant test_vectors: test_vector_array := (
    -- reset the counter
        (clk => '0', rst => '1', cnt => "000"),
        (clk => '1', rst => '1', cnt => "000"),
        (clk => '0', rst => '0', cnt => "000"),
    -- clock the counter several times
        (clk => '1', rst => '0', cnt => "001"),
        (clk => '0', rst => '0', cnt => "001"),
        (clk => '1', rst => '0', cnt => "010"),
        (clk => '0', rst => '0', cnt => "010"),
        (clk => '1', rst => '0', cnt => "011"),
        (clk => '0', rst => '0', cnt => "011"),
        (clk => '1', rst => '0', cnt => "100"),
        (clk => '0', rst => '0', cnt => "100"),
        (clk => '1', rst => '0', cnt => "101"),
        (clk => '0', rst => '0', cnt => "101"),
        (clk => '1', rst => '0', cnt => "110"),
        (clk => '0', rst => '0', cnt => "110"),
        (clk => '1', rst => '0', cnt => "111"),
        (clk => '0', rst => '0', cnt => "111"),
```

```
        (clk => '1', rst => '0', cnt => "000"),
        (clk => '0', rst => '0', cnt => "000"),
        (clk => '1', rst => '0', cnt => "001"),
        (clk => '0', rst => '0', cnt => "001"),
        (clk => '1', rst => '0', cnt => "010"),
    -- reset the counter
        (clk => '0', rst => '1', cnt => "000"),
        (clk => '1', rst => '1', cnt => "000"),
        (clk => '0', rst => '0', cnt => "000"),
    -- clock the counter several times
        (clk => '1', rst => '0', cnt => "001"),
        (clk => '0', rst => '0', cnt => "001"),
        (clk => '1', rst => '0', cnt => "010"),
        (clk => '0', rst => '0', cnt => "010")
  );
begin
  -- instantiate unit under test
  uut: count port map(clk => clk, rst => rst, cnt => cnt);

  -- apply test vectors and check results
  verify: process
        variable vector: test_record;
        variable errors: boolean := false;
  begin
        for i in test_vectors'range loop
                -- get vector i
                vector := test_vectors(i);

                -- schedule vector i
                clk <= vector.clk;
                rst <= vector.rst;

                -- wait for circuit to settle
                wait for 20 ns;

                -- check output vectors
                if cnt /= vector.cnt then
                        assert false
                                report "cnt is wrong value ";
                        errors := true;
                end if;
        end loop;

        -- assert reports on false
        assert not errors
          report "Test vectors failed."
          severity note;
        assert errors
          report "Test vectors passed."
          severity note;
        wait;
  end process;
```

```
end;
```

Listing 9-2 Design used to run test vectors

The unit under test was instantiated with the actuals associated with the formals by using named association rather than by positional association. This was done so that the component of the post-fit model, which will not likely have the ports listed in the same order, can be instantiated in the same test fixture.

Sometimes, it will be easier to simply apply the input test vectors and observe the output vectors in a waveform. However, you may want to record these output vectors and compare the results of simulating the source code with those of the post-fit model. Also, if you will need to apply a large number of vectors, then you will likely not want to list those vectors in the source code. Instead, you may want to list the vectors in a file and read them into the source code. This approach also allows you to use the same test fixture to run multiple tests simply by changing the file.

Eight procedure calls will be used to read in a line from a file, read std_logics or std_logic_vectors from a line, write std_logics or std_logic_vectors to a line, and write a line to a file. The procedure declarations for these functions are:

```
procedure Readline (F:in Text; L:out Line);
procedure Writeline(F:out Text; L:in Line);
procedure Read (L: inout Line; Value: out std_logic; Good: out boolean);
procedure Read (L: inout Line; Value: out std_logic);
procedure Read (L: inout Line; Value: out std_logic_vector;
                      Good: out boolean);
procedure Read (L: inout Line; Value: out std_logic_vector);
procedure Write (L: inout Line; Value: in std_logic;
                      Justified:in Side := Right; Field: in Width := 0);
procedure Write (L: inout Line; Value: in std_logic_vector;
                      Justified:in Side := Right; Field: in Width := 0);
```

The first procedure declaration is for a procedure that reads in a line from a file. The second is for a procedure that writes a line to a file. These two procedures are from the TEXTIO package that is defined by the IEEE 1076 standard.

The third procedure declaration is for a procedure that attempts to read a std_logic value from the beginning of the line. If the value is a std_logic_value, then *good* returns the value true; otherwise, it returns the value false. The fourth procedure declaration is for a procedure that reads a std_logic value from a line. The next two procedure declarations are for procedures that perform the same jobs for std_logic_vectors. The two write functions are for writing std_logics and std_logic_vectors to a line.

Unfortunately, whereas there are standard *read* and *write* procedures for characters, strings, and bits, there are not (as of yet) standard procedures for the last of the six procedure declarations listed above. We have overloaded the standard procedure calls with ones for std_logics and std_logic_vectors. The content of these procedures is listed at the end of the chapter. For now, we can proceed with generating a test fixture for the state machine of *listing 5-2* from chapter 5. We will reference the above procedures in the design of our test fixture by including a USE clause.

We would like to place the following test vectors in a file and apply them to the *memory_controller*. Some of the inputs are the don't care value '-'.

```
-- test vectors for memory controller

reset    r/w    ready    burst     bus_id
  1       0       0        0       00000000  -- reset
  1       0       0        0       00000000  -- reset
  1       0       0        0       00000000  -- reset
  0       0       0        0       00000000  -- wrong address
  0       0       0        0       10101010  -- wrong address
  0       1       0        0       10101010  -- wrong address
  0       0       1        1       10101010  -- wrong address
  0       0       0        0       10101010  -- wrong address
  0       0       1        0       10101010  -- wrong address
  0       0       1        0       10101010  -- wrong address
  0       0       1        0       10101010  -- wrong address
  0       0       0        0       11110011  -- right address, go to decision
  0       1       0        0       --------  -- go to read1
  0       0       0        0       --------  -- stay in read1, not ready
  0       0       1        1       --------  -- go to read2, it's a burst
  0       0       1        0       --------  -- go to read3
  0       0       0        0       --------  -- stay in read3, not ready
  0       0       1        0       --------  -- go to read4
  0       0       1        0       --------  -- go to idle
  0       0       0        0       00111011  -- wrong address
  0       0       0        0       00111011  -- wrong address
  0       0       0        0       00111011  -- wrong address
  0       0       0        0       11110011  -- right address, go to decision
  0       1       0        0       --------  -- go to read1
  0       0       0        0       --------  -- stay in read1, not ready
  0       0       0        0       --------  -- stay in read1, not ready
  0       0       1        0       --------  -- go to idle
  0       0       0        0       11110011  -- right address, go to decision
  0       0       0        0       --------  -- go to write
  0       0       1        0       --------  -- go to idle
  0       0       0        0       11110011  -- right address, go to decision
  0       1       0        0       --------  -- go to read1
  0       0       1        1       --------  -- go to read2, it's a burst
  0       0       1        0       --------  -- go to read3
  1       0       1        0       --------  -- reset -- go to idle
  0       0       0        0       11110011  -- right address, go to decision
  0       1       0        0       --------  -- go to read1
  0       0       1        1       --------  -- go to read2, it's a burst
  0       0       1        0       --------  -- go to read3
  0       0       1        0       --------  -- go to read4
  0       0       1        0       --------  -- go to idle
  0       0       1        0       00111011  -- wrong address
  0       0       1        0       00111011  -- wrong address
  0       0       1        0       00111011  -- wrong address
  0       0       1        0       00111011  -- wrong address
  0       0       0        0       11110011  -- right address, go to decision
  0       0       0        0       11110011  -- go to write
  0       0       0        0       11110011  -- stay in write, not ready
  0       0       0        0       11110011  -- stay in write, not ready
  0       0       0        0       11110011  -- stay in write, not ready
  0       0       0        0       11110011  -- stay in write, not ready
  0       0       1        0       11110011  -- go to idle
```

The test fixture for the memory controller is shown in *Listing 9-3*. Signals are declared to match the ports of the unit under test. This time, the clock is initialized to 0 because we will be controlling the

clock from within the test fixture rather than from vectors. The unit under test is instantiated, and a process is used to read in vectors, apply the vectors, and control clocking. The procedure calls expect variables in the port maps, so variables are declared to match the signal names of the component ports. As long as the file is not empty, a line is read from the input vector file. A character is taken from the beginning of the line and placed in the variable *ch*. If the attempt to read the first value of the line does not return a valid character, or if the character is not a tab, then the next iteration of the loop starts. If the line starts with a tab character, then the next value is read. It is expected to be a std_logic value, so it is taken from the line and placed into the variable *vreset*. If the value is not a std_logic, then the next iteration of the loop begins. If it is a std_logic, then we are confident that we have found a line containing a test vector. Tab characters and vector elements are read, according to the test vectors listed above.

After reading in the test vector, we force 10 ns of simulation time before scheduling the inputs. We want to apply the vectors 10 ns before the rising edge of the clock. If the clock has a 40 ns clock period, and the clock starts out at 0, then we want to wait 10 ns, apply the vectors, wait 10 ns, clock the circuit, wait for 20 ns (for a 50% duty cycle clock), and transition the clock back to 0 before reading in the next vector. We will record the present outputs 10 ns before the rising edge of the clock.

```
--------------------------------------------------------------------------
-- test fixture for memory controller
-- reads file "memory.inp" ; writes file "memory.out"
--------------------------------------------------------------------------

entity test_fixture_of_memory is end;

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.myio.all;
use work.memory_pkg.all;
architecture testmemory of test_fixture_of_memory is
   signal clk: std_logic := '0';
   signal reset, read_write, ready, burst, oe, we : std_logic;
   signal bus_id: std_logic_vector(7 downto 0);
   signal addr: std_logic_vector(1 downto 0);
begin
    -- instantiate the unit under test
    uut: memory_controller port map(
      reset => reset,
      read_write => read_write,
      ready => ready,
      burst => burst,
      clk => clk,
      bus_id => bus_id,
      oe => oe,
      we => we,
      addr => addr );

test: process
  file vector_file : text is in "memory.inp";
  file output_file : text is out "memory.out";
  variable invecs, outvecs : line;
```

```
  variable good : boolean;
  variable ch : character;
  variable vbus_id: std_logic_vector(7 downto 0);
  variable vreset, vread_write, vready, vburst : std_logic;
  variable out_vec: std_logic_vector(3 downto 0);
begin
    while not endfile(vector_file) loop
        -- read a line from the file
        readline(vector_file, invecs);

   --skip line if it does not begin with a tab
   read(invecs, ch, good);
   if not good or ch /= HT then next; end if;

        -- skip line if next value is not a std_logic
   read(invecs, vreset, good);
   next when not good;

   -- found a vector
   -- read vreset, vread_write, vready, vburst, vbus_id
   -- with tabs in between
   read(invecs,ch);
   read(invecs,vread_write);
   read(invecs,ch);
   read(invecs,vready);
   read(invecs,ch);
   read(invecs,vburst);
   read(invecs,ch);
   read(invecs,vbus_id);

   -- wait for 10 ns before scheduling the vector (we want
   -- to introduce skew between the vectors and clock edges)
   wait for 10 ns;

   -- schedule vectors
   reset <= vreset;
   read_write <= vread_write;
   ready <= vready;
   burst <= vburst;
   bus_id <= vbus_id;

   -- apply vectors with plenty of setup time
   -- also, record the current output vector
   -- we will record output vectors 10 ns before rising
   -- edge of clock for each clock cycle
   wait for 10 ns;
   out_vec := oe & we & addr;
   write(outvecs, out_vec); writeline(output_file, outvecs);

   -- schedule and execute clock transition
   clk <= not (clk);
   wait for 20 ns;

   -- schedule and ensure execution of next clock transition
```

```
    clk <= not (clk) after 0 ns;

     end loop;

     assert false REPORT "Test complete";
     end process;
end;
```

Listing 9-3 Test fixture for memory controller

The signals could have been scheduled by appending the words "after 10 ns" to the assignment statement, but since we need to record the outputs at same time that we are scheduling the inputs, we used the simple WAIT statement.

The output vectors are stored in a file called "memory.out"; presynthesis simulation results can be compared with post-layout simulation results by comparing the output files. On a UNIX system, the "diff" command can be used to ensure that the contents of two files are the same .

Overloaded Read and Write Procedures

The following are the overloaded *read* and *write* procedures used in the test fixture above. The procedures are overloaded for the type std_logic. Keep an eye on the World Wide Web (http://www.vhdl.org) for upcoming standard *read* and *write* procedures.

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

package myio is
  procedure Read (L: inout Line; Value: out std_logic; Good: out boolean);
  procedure Read (L: inout Line; Value: out std_logic);
  procedure Read (L: inout Line; Value: out std_logic_vector; Good: out
boolean);
  procedure Read (L: inout Line; Value: out std_logic_vector);
  procedure Write (L: inout Line; Value: in std_logic;
         Justified:in Side := Right; Field: in Width := 0);
  procedure Write (L: inout Line; Value: in std_logic_vector;
         Justified:in Side := Right; Field: in Width := 0);
end myio;

package body myio is

  procedure Read (L: inout Line; Value: out std_logic; Good: out boolean)
is
    variable temp: character;
    variable good_character: boolean;
  begin
       read(L, temp, good_character);
       if good_character = true then
      good := true;
           case temp is
               when 'U' => value := 'U';
               when 'X' => value := 'X';
```

```
                when '0' => value := '0';
                when '1' => value := '1';
                when 'Z' => value := 'Z';
                when 'W' => value := 'W';
                when 'L' => value := 'L';
                when 'H' => value := 'H';
                when '-' => value := '-';
                when others => good := false;
            end case;
        else
          good := false;
        end if;
    end Read;

    procedure Read (L: inout Line; Value: out std_logic) is
      variable temp: character;
      variable good_character: boolean;
    begin
        read(L, temp, good_character);
        if good_character = true then
            case temp is
                when 'U' => value := 'U';
                when 'X' => value := 'X';
                when '0' => value := '0';
                when '1' => value := '1';
                when 'Z' => value := 'Z';
                when 'W' => value := 'W';
                when 'L' => value := 'L';
                when 'H' => value := 'H';
                when '-' => value := '-';
      when others => value := 'U';
            end case;
        end if;
    end Read;

    procedure Read (L: inout Line; Value: out std_logic_vector; Good: out
  boolean) is
      variable temp: string(value'range);
      variable good_string: boolean;
    begin
     read(L, temp, good_string);
     if good_string = true then
       good := true;
       for i in temp'range loop
         case temp(i) is
        when 'U' => value(i) := 'U';
        when 'X' => value(i) := 'X';
        when '0' => value(i) := '0';
        when '1' => value(i) := '1';
        when 'Z' => value(i) := 'Z';
        when 'W' => value(i) := 'W';
        when 'L' => value(i) := 'L';
        when 'H' => value(i) := 'H';
        when '-' => value(i) := '-';
```

```
      when others => good := false;
       end case;
     end loop;
   else
     good := false;
   end if;
  end Read;

  procedure Read (L: inout Line; Value: out std_logic_vector) is
    variable temp: string(value'range);
    variable good_string: boolean;
  begin
   read(L, temp, good_string);
   if good_string = true then
     for i in temp'range loop
       case temp(i) is
      when 'U' => value(i) := 'U';
      when 'X' => value(i) := 'X';
      when '0' => value(i) := '0';
      when '1' => value(i) := '1';
      when 'Z' => value(i) := 'Z';
      when 'W' => value(i) := 'W';
      when 'L' => value(i) := 'L';
      when 'H' => value(i) := 'H';
      when '-' => value(i) := '-';
      when others => exit;
       end case;
     end loop;
   end if;
  end Read;

 procedure Write (L: inout Line; Value: in std_logic;
                        Justified:in Side := Right; Field: in Width := 0)
is
    variable write_value: character;
  begin
          case value is
                when 'U' => write_value := 'U';
                when 'X' => write_value := 'X';
                when '0' => write_value := '0';
                when '1' => write_value := '1';
                when 'Z' => write_value := 'Z';
                when 'W' => write_value := 'W';
                when 'L' => write_value := 'L';
                when 'H' => write_value := 'H';
                when '-' => write_value := '-';
             end case;
        write(L, write_value, Justified, Field);
  end Write;

 procedure Write (L: inout Line; Value: in std_logic_vector;
         Justified:in Side := Right; Field: in Width := 0) is
    variable write_value: string(value'range);
  begin
```

```
    for i in value'range loop
      case value(i) is
                when 'U' => write_value(i) := 'U';
                when 'X' => write_value(i) := 'X';
                when '0' => write_value(i) := '0';
                when '1' => write_value(i) := '1';
                when 'Z' => write_value(i) := 'Z';
                when 'W' => write_value(i) := 'W';
                when 'L' => write_value(i) := 'L';
                when 'H' => write_value(i) := 'H';
                when '-' => write_value(i) := '-';
              end case;
            end loop;
     write(L, write_value, Justified, Field);
    end Write;

end myio;
```

Exercises

1. List the advantages of using test fixtures.

2. Create a comprehensive test fixture for a 16-bit carry-lookahead adder.

3. Create a comprehensive test fixture for a 8-bit updown counter with three-state outputs. Verify the functionality of the counter with your test fixture. Ensure that your test fixture has the ability to evaluate setup time violations.

4. Survey the EDA industry for companies that support test fixtures for VHDL designs.

5. If you were to write an automated test fixture software to verify the functionality of a design, how would you go about it? What basic features would it include?

Appendix A—Glossary

actual –a signal name in the port map of a component instantiation that has been declared at the current level of hierarchy. The signal that is being mapped to a port of a component. Actuals are required in every component instantiation. *See* formal.

antifuse – whereas a fuse provides an electrical connection of wires that is initially intact, broken only after a programming voltage is applied across the fuse, an antifuse is an interconnection between wires that is initially broken and formed only after a programming voltage is applied across the antifuse.

architecture – the section of a VHDL description that describes the behavior or structure of the circuit that is defined by the entity. An entity must be paired with at least one architecture for a complete circuit description. *See* entity.

ASIC – stands for application specific integrated circuit. A semiconductor device that is custom made to perform a dedicated function. ASICs are not field programmable and must be manufactured specifically for a given application. Development of an ASIC typically requires an NRE, and the manufacturing of prototype devices can take multiple weeks. *See* NRE.

component – a declaration of an entity/architecture pair as a unit which can be instantiated in other designs, or another portion of a design. *See* entity, architecture.

constant declaration – a VHDL construct that defines a data object to hold a constant value.

die – the physical semiconductor silicon device found inside a device's packaging. Multiple silicon devices are formed on a single silicon wafer that is then cut into individual die for packaging.

die size – The area of the rectangular piece of silicon which makes up the semiconductor device.

entity – the section of a VHDL description that describes the communication or interface of a circuit to the outside world. An entity describes the external ports of communication, their type, and their mode or direction. An entity can also include generic declarations. *See* architecture, generic.

fault tolerance – the design of a system that is resistant to failure. A fault tolerant system is one that is designed to correct itself and continue to operate under many circumstances. The space shuttle computer is an example of a fault tolerant system. The term is also used to describe a state machine that forces itself into a known state whenever an unforeseen condition or hardware glitch is encountered. *See* glitch.

FIFO – stands for First In First Out. A FIFO is a buffer, a semiconductor memory device which stores information in the order in which it was received and then releases that information starting with the first element it received and progressing sequentially. FIFOs typically have a depth and width associated with them related to the size and amount of information that can be stored within.

fitting – the process of transforming a gate level or sum-of-products representation of a circuit into a file used for programming a PLD or CPLD device. This process typically occurs after synthesis, and the resulting file is typically a JEDEC file. *See* synthesis, place and route.

formal – the signal name of a component that is defined in that component's entity declaration. The signal that is being mapped to a component instantiation. Formals are optional when using positional signal associations. *See* actual.

generic – a VHDL construct that is used with an entity declaration to allow the communication of parameters between levels of hierarchy. A generic is typically used to define parameterized components wherein the size or other configuration are specified during the instantiation of the component. *See* entity.

Glitch – a voltage spike on a data line that should otherwise remain in a stable state. A glitch can occur on a signal that is at a logic high or logic low level.

hold time – the minimum time an input to a digital logic storage element must remain stable after the triggering edge of a clock has occurred. *See* metastability, setup time.

library – a collection of previously analyzed VHDL design units. A library can consist of multiple packages. *See* package.

logic block – one of multiple blocks of logic within a complex programmable logic device that are interconnected together via a global interconnect. A logic block in a CPLD is similar in nature and capability to a small PLD such as the 22V10. A logic block typically consists of a product term array, a product term distribution scheme, and a set of macrocells.

logic cell – a replicated element of logic within an FPGA device that typically contains a register and additional logic that forms the basic building block for implementing logic in the device.

macrocell – a replicated element of logic in PLD and CPLD architectures that typically contains a configurable memory element, polarity control, and one or more feedback paths to the global interconnect.

Mealy machine – a state machine for which outputs may change asynchronously with respect to the clock. *See* Moore machine.

metalogical value – a value of 'U', 'X', 'W', or '-' as defined in the IEEE 1164 standard of the type STD_LOGIC. These values are intended for simulation of VHDL models and represent signal logic values of uninitialized, forcing unknown, weak unknown, and don't care respectively.

metastability – a term meaning "in between." An undesirable output condition of digital logic storage elements caused by violation of the basic timing parameters associated with that storage element such as setup or hold time. Metastability can be seen as an output with a voltage level between the logic high and logic low states. *See* set-up time, hold time.

mode – associated with signals defined in a VHDL entity's port declaration. A mode defines the direction of communication a signal can have with other levels of hierarchy.

Moore machine – a state machine in which outputs change synchronously with respect to the clock. *See* Mealy machine.

NRE – stands for Non Recurring Engineering and refers to the costs associated with a design of a system that are incurred once. The term is often used when referring to contracting an ASIC (applications specific integrated circuit) vendor to whom a large fee must be paid to produce the first devices and incremental fees are required for additional devices.

one-hot-one encoding – a method of state encoding which uses one register for each state. Only one bit is asserted, or "hot", for each state. The "hot" state can be asserted as a logic value of zero or one, but is typically a logic one.

one-hot-zero encoding – exactly like one-hot-one encoding except that the reset or idle state is encoded with all logic zero levels. This is typically done to allow the use of dedicated hardware register resets to easily place the state machine in a known reset or idle state.

package – a collection of declarations, including component, type, subtype, and constant declarations, that are intended for use by other design units. *See* library.

performance – the maximum clock frequency or slowest propagation delay of a design as implemented in a particular programmable logic device. Performance is typically measured in nanoseconds for propagation delay, or megahertz for clock frequency.

place and route – the process of transforming a gate level representation of a circuit into a programming file that may be used to program an FPGA device. This process requires two steps: one to place the required logic into logic cells, and one to route the logic cells via the horizontal and vertical routing channels to each other and the I/O pins. *See* synthesis, fitting, logic cell.

product term allocation – the method of distributing logic in the form of AND-terms (product terms) to macrocells, which contain OR gates, to complete the AND-OR logic equation. Each PLD/CPLD has a different method of distributing product terms to macrocells.

programmable interconnect – generically refers to any two wires which can be connected together via a programmable fuse (or anti-fuse). Also refers to the communication network within a CPLD architecture that provides communication between logic blocks and I/O pins.

routability – a measure of probability that a signal will successfully be connected from one location within a device to another location. Routability within FPGA devices is affected by the number of horizontal and vertical routing channels and the amount of circuitry that can be carried through those channels. Routability within CPLD devices refers to the probability that a set of logic signals will be successfully routed through the global interconnect and into a logic block.

setup time – the minimum time an input to a digital logic storage element must remain stable before the triggering edge of a clock will occur. *See* metastability, hold time.

synthesis – the process of converting a high-level design description into a gate level or sum-of-products representation. *See* fitting, place and route.

test bench – an HDL description which is used to apply simulation vectors to a another HDL design description. A test bench is used for simulation only and can be applied to either a pre-synthesis or a post-fitting (place and route) design description.

type – an attribute of a VHDL data object that determines the values that the object can hold. Examples of types are bit and std_logic. Objects of type bit can hold values '0' or '1'. Objects of type std_logic can hold values of 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', or '-'.

Appendix B—Quick Reference Guide

This quick reference guide is not meant to be an exhaustive guide to the VHDL language. Rather, it is intended to be used as a reference to help you quickly build VHDL descriptions for use with synthesis tools.

The right hand side columns of all tables contain brief examples. Constructs that are simplified and modified versions of the BNF (Backus-Naur form) syntax categories found in Annex A of the IEEE Std. 1076-1993 *Language Refererence Manual* (LRM) are contained in the left hand side column for the first three major headings, "Building Blocks," "Language Constructs," and "Describing Synchronous Logic." The BNF syntax categories are simplified and modified so as to present only those constructs most useful in creating VHDL designs for use with synthesis tools, and to combine a syntax category used in another syntax category. In BNF, **boldface** words are reserved words and lower case words represent syntax categories. A vertical bar, |, represents a choice between items, square brackets, [ ], enclose optional items, and braces, { }, contain items that may be optionally repeated (except for **boldface** items immediately following an opening brace—these items must be included).

The left hand side column for the last two major headings, "Data Objects, Types, and Modes" and "Operators" contain a brief description of each item. Some editorial comments are also made to indicate the usefulness of the item in writing simple designs.

Building Blocks

Entities

Creating an entity declaration	

entity *entity*_name **is**
  port (
  [**signal**] identifier {, identifier}: [mode] type_mark
  {; [**signal**] identifier {, identifier}: [mode] type_mark});
end [*entity*_name];
	entity register8 **is**
	  port (
	  clk, rst,en: **in** *std_logic*;
	  data: **in** *std_logic_vector*(7 **downto** 0);
	  q: **out** *std_logic_vector*(7 **downto** 0));
	end register8;

Creating an entity declaration with generics	

entity *entity*_name **is**
  generic (
  [**signal**] identifier {, identifier}: [mode] type_mark
  [:=*static*_expression]
  {; [**signal**] identifier {, identifier}: [mode] type_mark
  [:=*static*_expression]});
  port (
  [**signal**] identifier {, identifier}: [mode] type_mark
  {; [**signal**] identifier {, identifier}: [mode] type_mark});
end [*entity*_name];
	entity register_n **is**
	  generic (
	  width: *integer* := 8);
	  port (
	  clk, rst, en: **in** *std_logic*;
	  data: **in** *std_logic_vector*(width-1 **downto** 0);
	  q: **out** *std_logic_vector*(width-1 **downto** 0));
	end register_n;

Architectures

Creating an architecture

```
architecture architecture_name of entity_name is
    type_declaration
    | signal_declaration
    | constant_declaration
    | component_declaration
    | alias_declaration
    | attribute_specification
    | subprogram_body
begin
    {process_statement
    | concurrent_signal_assignment_statement
    | component_instantiation_statement
    | generate_statement}
end [architecture_name];
```

```
architecture archregister8 of register8 is
begin
    process (rst, clk)
    begin
        if (rst = '1') then
            q <= (others => '0');
        elseif (clk'event and clk = '1') then
            if (en = '1') then
                q <= data;
            else
                q <= q;
            end if;
        end if;
    end process;
end archregister8;

architecture archfsm of fsm is
    type state_type is (st0, st1, st2);
    signal state:   state_type;
    signal y, z:    std_logic;
begin
    process (rst, clk)
    begin
      wait until clk'event = '1';
        case state is
            when st0 =>
                state <= st1;
                y <= '1';
            when st1 =>
                state <= st2;
                z <= '1';
            when others =>
                state <= st3;
                y <= '0';
                z <= '0';
            end case;
    end process;
end archfsm;
```

Components

Declaring a component

```
component component_name is
    port (
        [signal] identifier {, identifier}: [mode] type_mark
        {; [signal] identifier {, identifier}: [mode] type_mark});
end component;
```

```
component register8 is
    port (
        clk, rst, en:   in std_logic;
        data:           in std_logic_vector(7 downto 0);
        q:              out std_logic_vector(7 downto 0));
end component;
```

Declaring a component with generics

```
component compnent_name is
    generic (
        [signal] identifier {, identifier}: [mode] type_mark
            [:=static_expression]
        {; [signal] identifier {, identifier}: [mode] type_mark
            [:=static_expression]});
    port (
        [signal] identifier {, identifier}: [mode] type_mark
        {; [signal] identifier {, identifier}: [mode] type_mark});
end component;
```

```
component register8 is
    generic (
        width: integer := 8);
    port (
        clk, rst, en:   in std_logic;
        data:           in std_logic_vector(width-1 downto 0);
        q:              out std_logic_vector(width-1 downto 0));
end component;
```

Instantiating a component (named association)

```
instantiation_label:
    component_name
        port map (
            port_name =>        signal_name
                                | expression
                                | variable_name
                                | open
            {, port_name =>     signal_name
                                | expression
                                | variable_name
                                | open});
```

```
architecture archreg8 of reg8 is
    signal clock, reset, enable:    std_logic;
    signal data_in, data_out:       std_logic_vector(7 downto 0);
begin
  First_reg8:
    register8
        port map (
            clk  => clock,
            rst  => reset,
            en   => enable,
            data => data_in,
            q    => data_out);
end archreg8;
```

Instantiating a component with generics (named association)

```
instantiation_label:
    component_name
        generic map(
            generic_name =>   signal_name
                              | expression
                              | variable_name
                              | open
            {, generic_name =>signal_name
                              | expression
                              | variable_name
                              | open})
        port map (
            port_name =>      signal_name
                              | expression
                              | variable_name
                              | open
            {, port_name =>   signal_name
                              | expression
                              | variable_name
                              | open});
```

```
architecture archreg5 of reg5 is
    signal clock, reset, enable:    std_logic;
    signal data_in, data_out:       std_logic_vector(7 downto 0);
begin
  First_reg5:
    register_n
        generic map (5)
        port map (
            clk  => clock,
            rst  => reset,
            en   => enable,
            data => data_in,
            q    => data_out);
end archreg5;
```

Instantiating a component (positional association)

```
instantiation_label:
    component_name
        port map ( signal_name | expression
                 | variable_name | open
            {, signal_name | expression
                 | variable_name | open});
```

```
architecture archreg8 of reg8 is
    signal clock, reset, enable:    std_logic;
    signal data_in, data_out:       std_logic_vector(7 downto 0);
begin
  First_reg8:
    register8
        port map (clock, reset, enable, data_in, data_out);
end archreg8;
```

Instantiating a component with generics (positional association)

```
instantiation_label:
    component_name
        generic map ( signal_name | expression
                 | variable_name | open
            {, signal_name | expression
                 | variable_name | open})
        port map ( signal_name | expression
                 | variable_name | open
            {, signal_name | expression
                 | variable_name | open});
```

```
architecture archreg5 of reg5 is
    signal clock, reset, enable:    std_logic;
    signal data_in, data_out:       std_logic_vector(7 downto 0);
begin
First_reg5:
    register_n
        generic map (5)
        port map (clock, reset, enable, data_in, data_out);
end archreg5;
```

Language Constructs

Concurrent Statements

Boolean equations

```
relation { and relation}
| relation { or relation }
| relation { xor relation }
| relation { nand relation }
| relation { nor relation }
```

```
v <= a and b and c;
w <= a or b or c;
x <= a xor b xor c;
y <= a nand b nand c;
z <= a nor b nor c;
```

Conditional signal assignment (WHEN-ELSE)

```
{expression when condition else} expression;
```

```
a <= '1' when b = c else '0';
```

Selected signal assignment (WITH-SELECT-WHEN)

```
with selection_expression select
    {identifier <= expression when
            identifier | expression | discrete_range | others,}
    identifier <= expression when
            identifier | expression | discrete_range | others;
```

```
architecture archfsm of fsm is
    type state_type is (st0, st1, st2, st3);
    signal state:   state_type;
    signal y, z:    std_logic;
begin
    with state select
        x <=    "0000" when st0 | st1;
                "0010" when st2 | st3;
                y and z when others;
end archfsm;
```

Component instantiation — see above

Generate scheme for component instantiation or generation of equations

```
generate_label: (for identifier in discrete_range)
            | (if condition) generate
                {concurrent_statement}
            end generate [generate_label] ;
```

```
g1:  for i in 0 to 7 generate
        reg1: register8 port map (clock, reset, enable,
                data_in(i), data_out(i));
     end generate g1;

g2:  for j in 0 to 2 generate
        a(j) <= b(j) xor c(j);
     end generate g2;
```

ential Statements

cess statement

*ocess*_label:]
rocess (sensitivity list)
{type_declaration
| constant_declaration
| variable_declaration
| alias_declaration}
begin
{wait_statement
| signal_assignment_statement
| variable_assignment_statement
| if_statement
| case_statement
| loop_statement
end process [*process*_label];

```
my_process:
process (rst, clk)
  constant zilch : std_logic_vector(7 downto 0) := "0000_0000";
begin
    wait until clk = '1';
        if (rst = '1') then
            q <= zilch;
        elsif (en = '1') then
            q <= data;
        else
            q <= q;
        end if;
end my_process;
```

IF-THEN-ELSE-ELSIF statement

if condition **then** sequence_of_statements
{**elsif** condition **then** sequence_of_statements}
[**else** sequence_of_statements]
end if;

```
if (count = "00") then
    a <= b;
elsif (count = "10") then
    a <= c;
else
    a <= d;
end if;
```

CASE-WHEN statement

case expression **is**
{**when** identifier | expression | discrete_range | **others =>**
sequence_of_statements}
end case;

```
case count is
    when "00" =>
        a <= b;
    when "10" =>
        a <= c;
    when others =>
        a <= d;
end case;
```

FOR LOOP statement

[*loop*_label:]
for identifier **in** discrete_range **loop**
sequence_of_statements
end loop [*loop*_label];

```
my_for_loop:
for i in 3 downto 0 loop
    if reset(i) = '1' then
        data_out(i) := '0';
    end if;
end loop my_for_loop;
```

WHILE LOOP statement

[*loop*_label:]
while condition **loop**
sequence_of_statements
end loop [*loop*_label];

```
my_while_loop:
    while (count > 0) loop
        count := count - 1;
        result <= result + data_in;
    end loop my_while_loop;
```

Describing Synchronous Logic Using Processes

No reset (assume clock is of type std_logic)

```
[process_label:]
process (clock)
begin
    if clock'event and clock = '1' then
        synchronous_signal_assignment_statement;
    end if;
end process [process_label];
```

or

```
[process_label:]
process
begin
    wait until clock = '1';
        synchronous_signal_assignment_statement;
end process [process_label];
```

```
reg8_no_reset:
process (clk)
  begin
    if clk'event and clk = '1' then
        q <= data;
    end if;
end process reg8_no_reset;
```

or

```
reg8_no_reset:
process (clk)
  begin
    wait until clock = '1';
        q <= data;
end process reg8_no_reset;
```

Synchronous reset

```
[process_label:]
process (clock)
begin
    if clock'event and clock = '1' then
        if synch_reset_signal = '1' then
            synchronous_signal_assignment_statement;
        else
            different_synchronous_signal_assignment_statement;
        end if;
    end if;
end process [process_label];
```

```
reg8_sync_reset:
process (clk)
  begin
    if clk'event and clk = '1' then
        if sync_reset = '1' then
            q <= "0000_0000";
        else
            q <= data;
        end if;
    end if;
end process;
```

Asynchronous reset or preset

```
[process_label:]
process (reset, clock)
begin
    if reset = '1' then
        asynchronous_signal_assignment_statement;
    elsif clock'event and clock = '1' then
        synchronous_signal_assignment_statement;
    end if;
end process [process_label];
```

```
reg8_async_reset:
process (asyn_reset, clk)
  begin
    if async_reset = '1' then
        q <= "0000_0000";
    elsif clk'event and clk = '1' then
        q <= data;
    end if;
end process reg8_async_reset;
```

Asynchronous reset and preset

```
[process_label:]
process (reset, preset, clock)
begin
    if reset = '1' then
        asynchronous_signal_assignment_statement;
    elsif preset = '1' then
        different_asynchronous_signal_assignment_statement;
    elsif clock'event and clock = '1' then
        synchronous_signal_assignment_statement;
    end if;
end process [process_label];
```

```
reg8_async:
process (asyn_reset, async_preset, clk)
begin
    if async_reset = '1' then
        q <= "0000_0000";
    elsif async_reset = '1' then
        q <= "1111_1111";
    elsif clk'event and clk = '1' then
        q <= data;
    end if;
end process reg8_async;
```

Conditional Synchronous Assignment (enables)

```
[process_label:]
process (reset, clock)
begin
    if reset = '1' then
        asynchronous_signal_assignment_statement;
    elsif clock'event and clock = '1' then
        if enable = '1' then
            synchronous_signal_assignment_statement;
        else
            different_synchronous_signal_assignment_statement;
        end if;
    end if;
end process [process_label];
```

```
reg8_sync_assign:
process (rst, clk)
begin
    if rst = '1' then
        q <= "0000_0000";
    elsif clk'event and clk = '1' then
        if enable = '1' then
            q <= data;
        else
            q <= q;
        end if;
    end if;
end process reg8_sync_assign;
```

Data Objects, Types, and Modes

Data Objects

Signals

Signals are the most commonly used data object in VHDL designs. Nearly all basic designs, and many large designs as well, can be fully described using signals as the only data object.

```
architecture archinternal_counter of internal_counter is
    signal count, data: std_logic_vector(7 downto 0);
begin
process (clk)
  begin
    if (clk'event and clk = '1') then
        if en = '1' then
            count <= data;
        else
            count <= count + 1;
        end if;
    end if;
end process;
end archinternal_counter;
```

Constants

Constants are used to hold a static value; they are typically used to improve the readability and maintainence of code.

```
my_process:
process (rst, clk)
  constant zilch : std_logic_vector(7 downto 0) := "0000_0000";
begin
    wait until clk = '1';
        if (rst = '1') then
            q <= zilch;
        elsif (en = '1') then
            q <= data;
        else
            q <= q;
        end if;
end my_process;
```

Variables

+ Variables can be used in processes and subprograms (sequential areas) only.
+ The scope of a variable is the process or subprogram, and the variable does not retain it's value when a process or subprogram becomes inactive.
+ Variables are most commonly used as the indices of loops, calculation of intermediate values, or for immediate assignment.
+ To use the value of a variable outside of the process or subprogram in which it was declared, the value of the variable must be assigned to a signal.

```
architecture archloopstuff of loopstuff is
    signal data:   std_logic_vector (3 downto 0);
    signal result: std_logic;
begin
process (data)
  variable tmp:    std_logic;
begin
    for i in a'range downto 0 loop
        tmp := tmp and data(i);
    end loop;
    result <= tmp;
end process;
end archloopstuff;
```

<table>
<tr><td colspan="2">std_logic</td></tr>
<tr><td>+ Values are:
'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 standard multivalue logic system for VHDL model interoperability.
+ A resolved type (i.e., a resolution function is used to determine the value of a signal with more than one driver).
+ Along with its subtypes, std_logic should be used over user defined types to ensure interoperability of VHDL models among synthesis and simulation tools.
+ To use must include the following two lines:
library ieee;
use ieee.std_logic_1164.all;</td><td>signal x: std_logic;
. . .
x <= data when enable = '1' else 'Z';</td></tr>
<tr><td colspan="2">std_ulogic</td></tr>
<tr><td>+ Values are:
'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
+ An unresolved type (i.e., a signal of this type may have only one driver).
+ To use must include the following two lines:
library ieee;
use ieee.std_logic_1164.all;</td><td>signal x, data, enable: std_ulogic;
. . .
x <= data when enable = '1' else 'Z';</td></tr>
<tr><td colspan="2">std_logic_vector and std_ulogic_vector</td></tr>
</table>

<table>
<tr><td>+ Are arrays of types std_logic and std_ulogic
+ Along with its subtypes, std_logic_vector should be used over user defined types to ensure interoperability of VHDL models among synthesis and simulation tools.
+ To use must include the following two lines:
library ieee;
use ieee.std_logic_1164.all;</td><td>signal mux: std_logic_vector(7 downto 0);
. . .
if state = address or state = ras then
mux <= dram_a;
else
mux <= (others => 'Z');
end if;</td></tr>
<tr><td colspan="2">bit and bit_vector</td></tr>
<tr><td>+ Bit values are: '0' and '1'.
+ Bit_vector is an array of bits.
+ Pre-defined by the IEEE 1076 standard.
+ This type was used extensively prior to the introduction and tool vendor support of std_logic_1164.
+ Useful when metalogic values not required.</td><td>signal x: bit;
. . .
if x = '1' then
state <= idle;
else
state <= start;
end if;</td></tr>
<tr><td colspan="2">boolean</td></tr>
<tr><td>+ Values are: TRUE and FALSE
+ Often used as return value of function.
+ Otherwise, not often used.</td><td>signal a: boolean;
. . .
if a = '1' then
state <= idle;
else
state <= start;
end if;</td></tr>
<tr><td colspan="2">integer</td></tr>
<tr><td>+ Values are the set of integers.
+ Data objects of this type are primarily constant, used for defining widths of signals or as an operand in an addition or subtraction.</td><td>entity counter_n is
generic (
width: integer := 8);
port (
clk, rst: in std_logic;
count: out std_logic_vector(width-1 downto 0));
end counter_n;
. . .
process(clk)
begin
if (rst = '1') then
count <= 0;
elsif (clk'event and clk='1') then
count <= count + 1;
end if;
end process;</td></tr>
</table>

enumeration types	

+ Values are user defined.
+ Commonly used to define states for a state machine
	architecture archfsm **of** fsm **is**
	  type state_type **is** (st0, st1, st2);
	  signal state: state_type;
	  signal y, z: *std_logic*;
	begin
	  process (clk)
	  begin
	  wait until clk'**event** = '1';
	  case state **is**
	  when st0 =>
	  state <= st2;
	  y <= '1';
	  when st1 =>
	  state <= st3;
	  z <= '1';
	  when others =>
	  state <= st0;
	  y <= '0';
	  z <= '0';
	  end case;
	  end process;
	end archfsm;

Modes

In	
+ Used for signals (ports are signals) that are inputs-only to an entity.	
Out	
+ Used for signals that are outputs-only and for which the values are not required internal to the entity.	
Buffer	
+ Used for signals that are outputs, but for which the values are required internal to the given entity. + Caveat with usage: The formal associated with an acutal that is of mode buffer must also be of mode buffer (i.e., if the source of a port of mode buffer is the output of another component, the mode of the port of that component must also be mode buffer.)	
Inout	
+ Used for signals that are truly bidirectional signals + May also be used for signals that are inputs-only or outputs-only, at the expense of readability of the code.	

```
entity counter_4 is
    port (
        clk, rst, ld:    in std_logic;
        term_cnt:        buffer std_logic;
        count:           inout std_logic_vector(3 downto 0));
end counter_4;

architecture archcounter_4 of counter_4 is
  signal int_rst:    std_logic;
  signal int_count:  std_logic_vector(3 downto 0);
begin
  process(int_rst, clk)
      begin
          if (int_rst = '1') then
              int_count <= "0000";
          elsif (clk'event and clk='1') then
              if (ld = '1') then
                  int_count <= count;
              else
                  int_count <= int_count + 1;
              end if;
          end if;
  end process;

  term_cnt <= count(2) and count(0);     -- term_cnt is 3
  int_rst <= term_cnt or rst;            -- resets at term_cnt
  count <= int_count when ld = '0' else "ZZZZ";
                                         -- count is bidirectional
end archcounter_4;
```

Operators

All operators of the same class hve the same level of precedence. The classes of operators are listed here in order of decreasing precedence. Many of the operators are overloaded in the NUMERIC_BIT and NUMERIC_STD packages; consult the World Wide Web (http://www.vhdl.org) for most recent updates to these standards. The IEEE working group 1076.3 is working on these standards.

Miscellaneous operators	
+ Operators: **, abs, not + The not operator is used frequently, the other two are rarely used for designs to be synthesized. + Predefined for any integer type (**), any numeric type (abs), and either bit and boolean (not).	**signal** a, b, c: *bit*; . . . a <= **not** (b **and** c);
Multiplying operators	
+ Operators: *, /, mod, rem + The * operator is occassionally used for multipliers; the other three are rarely used. + Predefined for any integer type (*, /, mod, rem), and any floating point type (*,/)	**variable** a, b: *integer* **range** 0 **to** 255; . . . a <= b * 2;
Sign	
+ Operators: +, - + Rarely used for synthesis + Predefined for any numeric type (floating point or integer)	**variable** a, b, c: *integer* **range** 0 **to** 255; . . . a <= - (b + 2);
Adding operators	
+ Operators: +, - + Used frequently to describe incrementers, decrementers, adders, and subtractors. + Predefined for any numeric type	**signal** count: *integer* **range** 0 **to** 255; . . . count <= count + 1;
Shift operators	
+ Operators: sll, srl, sla, sra, rol, ror + Used occasionally + Predefined for any one-dimensional array with elements of type bit or boolean.	**signal** a, b: *bit_vector*(4 **downto** 0); **signal** c: *integer* **range** 0 **to** 4; . . . a <= b **sll** c;
Relational operators	

+ Operators: =. /=, < , <=, >, >=
+ Used frequently for comparisons
+ Predefined for any type (both operands must be of same type)
	signal a, b: *integer* **range** 0 **to** 255;
	signal agtb: *std_logic*;
	. . .
	if a >= b **then**
	agtb <= '1';
	else
	agtb <= '0';

Logical Operators	
+ Operators: and, or, nand, nor, xor, xnor + Used frequently to generate boolean equations + Predefined for types bit and boolean. Std_logic_1164 overloads these operators for std_ulogic and its subtypes.	**signal** a, b, c: *std_logic*; . . . a <= b **and** c;

Appendix C—Std_logic_1164

Below is a copy of the *std_logic_1164* package. It is used in many of the examples throughout the text. The package declaration declares several data types and functions. The package body defines those functions declared in the package declaration.

```
-- ------------------------------------------------------------------
--
--   Title     :  std_logic_1164 multi-value logic system
--   Library   :  This package shall be compiled into a library
--             :  symbolically named IEEE.
--             :
--   Developers:  IEEE model standards group (par 1164)
--   Purpose   :  This packages defines a standard for designers
--             :  to use in describing the interconnection data types
--             :  used in vhdl modeling.
--             :
--   Limitation:  The logic system defined in this package may
--             :  be insufficient for modeling switched transistors,
--             :  since such a requirement is out of the scope of this
--             :  effort. Furthermore, mathematics, primitives,
--             :  timing standards, etc. are considered orthogonal
--             :  issues as it relates to this package and are therefore
--             :  beyond the scope of this effort.
--             :
--   Note      :  No declarations or definitions shall be included in,
--             :  or excluded from this package. The "package declaration"
--             :  defines the types, subtypes and declarations of
--             :  std_logic_1164. The std_logic_1164 package body shall be
--             :  considered the formal definition of the semantics of
--             :  this package. Tool developers may choose to implement
--             :  the package body in the most efficient manner available
--             :  to them.
--             :
-- ------------------------------------------------------------------
--   modification history :
-- ------------------------------------------------------------------
--  version | mod. date:|
--   v4.200 | 01/02/92  |
-- ------------------------------------------------------------------
--
-- $Id: stdlogic.vhd,v 1.3 1994/04/06 18:02:24 hemmert Exp $
--

PACKAGE std_logic_1164 IS

    ------------------------------------------------------------------
    -- logic state system  (unresolved)
    ------------------------------------------------------------------
    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
                      );
    ------------------------------------------------------------------
    -- unconstrained array of std_ulogic for use with the resolution
function
    ------------------------------------------------------------------
    TYPE std_ulogic_vector IS ARRAY ( NATURAL RANGE <> ) OF std_ulogic;

    ------------------------------------------------------------------
    -- resolution function
    ------------------------------------------------------------------
    FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic;


    ------------------------------------------------------------------
    -- *** industry standard logic type ***
    ------------------------------------------------------------------
    SUBTYPE std_logic IS resolved std_ulogic;

    ------------------------------------------------------------------
    -- unconstrained array of std_logic for use in declaring signal arrays
    ------------------------------------------------------------------
    TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;

    ------------------------------------------------------------------
    -- common 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')

    ------------------------------------------------------------------
    -- overloaded logical operators
    ------------------------------------------------------------------

    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;
    FUNCTION "not"  ( l : std_ulogic                 ) RETURN UX01;

    ------------------------------------------------------------------
    -- vectorized overloaded logical operators
```

```
    ------------------------------------------------------------------
    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;

--  ------------------------------------------------------------------
--  Note : The declaration and implementation of the "xnor" function is
--  specifically commented until at which time the VHDL language has been
--  officially adopted as containing such a function. At such a point,
--  the following comments may be removed along with this notice without
--  further "official" ballotting of this std_logic_1164 package. It is
--  the intent of this effort to provide such a function once it becomes
--  available in the VHDL standard.
--  ------------------------------------------------------------------
--  function "xnor" ( l, r : std_logic_vector  ) return std_logic_vector;
--  function "xnor" ( l, r : std_ulogic_vector ) return std_ulogic_vector;

    FUNCTION "not"  ( l : std_logic_vector  ) RETURN std_logic_vector;
    FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector;

    ATTRIBUTE no_op OF "and", "or", "nand", "nor", "xor", "not"
        : FUNCTION IS TRUE;              --  For CYPRESS synthesis.
    ------------------------------------------------------------------
    -- conversion functions
    ------------------------------------------------------------------
    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;
```

```
    ------------------------------------------------------------------
    -- strength strippers and type convertors
    ------------------------------------------------------------------

    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;

    ------------------------------------------------------------------
    -- edge detection
    ------------------------------------------------------------------
    FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN;
    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN;

    ------------------------------------------------------------------
    -- object contains an unknown
    ------------------------------------------------------------------
    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;

END std_logic_1164;

PACKAGE BODY std_logic_1164 IS
    ------------------------------------------------------------------
    -- local types
    ------------------------------------------------------------------
    TYPE stdlogic_1d IS ARRAY (std_ulogic) OF std_ulogic;
    TYPE stdlogic_table IS ARRAY(std_ulogic, std_ulogic) OF std_ulogic;

    ------------------------------------------------------------------
    -- resolution function
    ------------------------------------------------------------------
    CONSTANT resolution_table : stdlogic_table := (
    --      ---------------------------------------------------------
    --      |  U    X    0    1    Z    W    L    H    -         |    |
    --      ---------------------------------------------------------
```

```
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
        ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
        ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
        ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
        ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
        ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
        ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
    );

FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
    VARIABLE result : std_ulogic := 'Z';  -- weakest state default
BEGIN
    -- the test for a single driver is essential otherwise the
    -- loop would return 'X' for a single driver of '-' and that
    -- would conflict with the value of a single driver unresolved
    -- signal.
    IF    (s'LENGTH = 1) THEN    RETURN s(s'LOW);
    ELSE
        FOR i IN s'RANGE LOOP
            result := resolution_table(result, s(i));
        END LOOP;
    END IF;
    RETURN result;
END resolved;

-------------------------------------------------------------------
-- tables for logical operations
-------------------------------------------------------------------

-- truth table for "and" function
CONSTANT and_table : stdlogic_table := (
--      ---------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -      |   |
--      ---------------------------------------------------
        ( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ),  -- | U |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | X |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | 0 |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 1 |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),  -- | W |
        ( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),  -- | L |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | H |
        ( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )   -- | - |
);

-- truth table for "or" function
CONSTANT or_table : stdlogic_table := (
--      ---------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -      |   |
--      ---------------------------------------------------
        ( 'U', 'U', 'U', '1', 'U', 'U', 'U', '1', 'U' ),  -- | U |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | X |
```

```
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 0 |
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ),  -- | 1 |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | Z |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' ),  -- | W |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | L |
        ( '1', '1', '1', '1', '1', '1', '1', '1', '1' ),  -- | H |
        ( 'U', 'X', 'X', '1', 'X', 'X', 'X', '1', 'X' )   -- | - |
);

-- truth table for "xor" function
CONSTANT xor_table : stdlogic_table := (
--      ----------------------------------------------------
--      |  U    X    0    1    Z    W    L    H    -       |   |
--      ----------------------------------------------------
        ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ),  -- | U |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | X |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | 0 |
        ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ),  -- | 1 |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | Z |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  -- | W |
        ( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),  -- | L |
        ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' ),  -- | H |
        ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )   -- | - |
);

-- truth table for "not" function
CONSTANT not_table: stdlogic_1d :=
--  ---------------------------------------------
--  |   U    X    0    1    Z    W    L    H    -   |
--  ---------------------------------------------
     ( 'U', 'X', '1', '0', 'X', 'X', '1', '0', 'X' );

-------------------------------------------------------------------
-- overloaded logical operators ( with optimizing hints )
-------------------------------------------------------------------

FUNCTION "and"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
BEGIN
    RETURN (and_table(l, r));
END "and";

FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
BEGIN
    RETURN  (not_table ( and_table(l, r)));
END "nand";

FUNCTION "or"   ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
BEGIN
    RETURN (or_table(l, r));
END "or";

FUNCTION "nor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
BEGIN
    RETURN  (not_table ( or_table( l, r )));
```

```
    END "nor";

    FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01 IS
    BEGIN
        RETURN (xor_table(l, r));
    END "xor";

--  function "xnor"  ( l : std_ulogic; r : std_ulogic ) return ux01 is
--  begin
--      return not_table(xor_table(l, r));
--  end "xnor";

    FUNCTION "not"  ( l : std_ulogic ) RETURN UX01 IS
    BEGIN
        RETURN (not_table(l));
    END "not";

    ------------------------------------------------------------------
    -- and
    ------------------------------------------------------------------
    FUNCTION "and"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
        ALIAS lv : std_logic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_logic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'and' operator are not of the
same length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := and_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "and";
    ------------------------------------------------------------------
    FUNCTION "and"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'and' operator are not of the
same length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := and_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
```

```
    END "and";
    ---------------------------------------------------------------------
    -- nand
    ---------------------------------------------------------------------
    FUNCTION "nand"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
        ALIAS lv : std_logic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_logic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'nand' operator are not of
the same length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := not_table(and_table (lv(i), rv(i)));
            END LOOP;
        END IF;
        RETURN result;
    END "nand";
    ---------------------------------------------------------------------
    FUNCTION "nand"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector
IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'nand' operator are not of
the same length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := not_table(and_table (lv(i), rv(i)));
            END LOOP;
        END IF;
        RETURN result;
    END "nand";
    ---------------------------------------------------------------------
    -- or
    ---------------------------------------------------------------------
    FUNCTION "or"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
        ALIAS lv : std_logic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_logic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'or' operator are not of the
same length"
            SEVERITY FAILURE;
        ELSE
```

```
            FOR i IN result'RANGE LOOP
                result(i) := or_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "or";
    ------------------------------------------------------------------
    FUNCTION "or"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'or' operator are not of the
same length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := or_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "or";
    ------------------------------------------------------------------
    -- nor
    ------------------------------------------------------------------
    FUNCTION "nor"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
        ALIAS lv : std_logic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_logic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'nor' operator are not of the
same length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := not_table(or_table (lv(i), rv(i)));
            END LOOP;
        END IF;
        RETURN result;
    END "nor";
    ------------------------------------------------------------------
    FUNCTION "nor"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'nor' operator are not of the
same length"
```

```
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := not_table(or_table (lv(i), rv(i)));
            END LOOP;
        END IF;
        RETURN result;
    END "nor";
    ------------------------------------------------------------------
    -- xor
    ------------------------------------------------------------------
    FUNCTION "xor"  ( l,r : std_logic_vector ) RETURN std_logic_vector IS
        ALIAS lv : std_logic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_logic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'xor' operator are not of the
same length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := xor_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "xor";
    ------------------------------------------------------------------
    FUNCTION "xor"  ( l,r : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
        ALIAS rv : std_ulogic_vector ( 1 TO r'LENGTH ) IS r;
        VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH );
    BEGIN
        IF ( l'LENGTH /= r'LENGTH ) THEN
            ASSERT FALSE
            REPORT "arguments of overloaded 'xor' operator are not of the
same length"
            SEVERITY FAILURE;
        ELSE
            FOR i IN result'RANGE LOOP
                result(i) := xor_table (lv(i), rv(i));
            END LOOP;
        END IF;
        RETURN result;
    END "xor";
--  ------------------------------------------------------------------
--  -- xnor
--  ------------------------------------------------------------------
--  ------------------------------------------------------------------
--  Note : The declaration and implementation of the "xnor" function is
--  specifically commented until at which time the VHDL language has been
--  officially adopted as containing such a function. At such a point,
--  the following comments may be removed along with this notice without
```

```
--  further "official" ballotting of this std_logic_1164 package. It is
--  the intent of this effort to provide such a function once it becomes
--  available in the VHDL standard.
--  ---------------------------------------------------------------------
--  function "xnor"  ( l,r : std_logic_vector ) return std_logic_vector is
--      alias lv : std_logic_vector ( 1 to l'length ) is l;
--      alias rv : std_logic_vector ( 1 to r'length ) is r;
--      variable result : std_logic_vector ( 1 to l'length );
--  begin
--      if ( l'length /= r'length ) then
--          assert false
--          report "arguments of overloaded 'xnor' operator are not of
the same length"
--          severity failure;
--      else
--          for i in result'range loop
--              result(i) := not_table(xor_table (lv(i), rv(i)));
--          end loop;
--      end if;
--      return result;
--  end "xnor";
--  ---------------------------------------------------------------------
--  function "xnor"  ( l,r : std_ulogic_vector ) return std_ulogic_vector
is
--      alias lv : std_ulogic_vector ( 1 to l'length ) is l;
--      alias rv : std_ulogic_vector ( 1 to r'length ) is r;
--      variable result : std_ulogic_vector ( 1 to l'length );
--  begin
--      if ( l'length /= r'length ) then
--          assert false
--          report "arguments of overloaded 'xnor' operator are not of
the same length"
--          severity failure;
--      else
--          for i in result'range loop
--              result(i) := not_table(xor_table (lv(i), rv(i)));
--          end loop;
--      end if;
--      return result;
--  end "xnor";


    ---------------------------------------------------------------------
    -- not
    ---------------------------------------------------------------------
    FUNCTION "not"  ( l : std_logic_vector ) RETURN std_logic_vector IS
        ALIAS lv : std_logic_vector ( 1 TO l'LENGTH ) IS l;
        VARIABLE result : std_logic_vector ( 1 TO l'LENGTH ) := (OTHERS
=> 'X');
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := not_table( lv(i) );
        END LOOP;
        RETURN result;
```

```
    END;
    -------------------------------------------------------------------
    FUNCTION "not"  ( l : std_ulogic_vector ) RETURN std_ulogic_vector IS
        ALIAS lv : std_ulogic_vector ( 1 TO l'LENGTH ) IS l;
       VARIABLE result : std_ulogic_vector ( 1 TO l'LENGTH ) := (OTHERS =>
'X');
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := not_table( lv(i) );
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    -- conversion tables
    -------------------------------------------------------------------
    TYPE logic_x01_table IS ARRAY (std_ulogic'LOW TO std_ulogic'HIGH) OF
X01;
    TYPE logic_x01z_table IS ARRAY (std_ulogic'LOW TO std_ulogic'HIGH) OF
X01Z;
    TYPE logic_ux01_table IS ARRAY (std_ulogic'LOW TO std_ulogic'HIGH) OF
UX01;
    ---------------------------------------------------------
    -- table name : cvt_to_x01
    --
    -- parameters :
    --        in   :  std_ulogic  -- some logic value
    -- returns     :  x01         -- state value of logic value
    -- purpose     :  to convert state-strength to state only
    --
    -- example     : if (cvt_to_x01 (input_signal) = '1' ) then ...
    --
    ---------------------------------------------------------
    CONSTANT cvt_to_x01 : logic_x01_table := (
                          'X',  -- 'U'
                          'X',  -- 'X'
                          '0',  -- '0'
                          '1',  -- '1'
                          'X',  -- 'Z'
                          'X',  -- 'W'
                          '0',  -- 'L'
                          '1',  -- 'H'
                          'X'   -- '-'
                         );

    ---------------------------------------------------------
    -- table name : cvt_to_x01z
    --
    -- parameters :
    --        in   :  std_ulogic  -- some logic value
    -- returns     :  x01z        -- state value of logic value
    -- purpose     :  to convert state-strength to state only
    --
    -- example     : if (cvt_to_x01z (input_signal) = '1' ) then ...
    --
```

```
    ------------------------------------------------------
    CONSTANT cvt_to_x01z : logic_x01z_table := (
                          'X',  -- 'U'
                          'X',  -- 'X'
                          '0',  -- '0'
                          '1',  -- '1'
                          'Z',  -- 'Z'
                          'X',  -- 'W'
                          '0',  -- 'L'
                          '1',  -- 'H'
                          'X'   -- '-'
                         );

    ------------------------------------------------------
    -- table name : cvt_to_ux01
    --
    -- parameters :
    --        in   :  std_ulogic  -- some logic value
    -- returns     :  ux01         -- state value of logic value
    -- purpose     :  to convert state-strength to state only
    --
    -- example     : if (cvt_to_ux01 (input_signal) = '1' ) then ...
    --
    ------------------------------------------------------
    CONSTANT cvt_to_ux01 : logic_ux01_table := (
                          'U',  -- 'U'
                          'X',  -- 'X'
                          '0',  -- '0'
                          '1',  -- '1'
                          'X',  -- 'Z'
                          'X',  -- 'W'
                          '0',  -- 'L'
                          '1',  -- 'H'
                          'X'   -- '-'
                         );

    ------------------------------------------------------------------
    -- conversion functions
    ------------------------------------------------------------------
    FUNCTION To_bit       ( s : std_ulogic;        xmap : BIT := '0')
RETURN BIT IS
    BEGIN
            CASE s IS
                WHEN '0' | 'L' => RETURN ('0');
                WHEN '1' | 'H' => RETURN ('1');
                WHEN OTHERS => RETURN xmap;
            END CASE;
    END;
    ------------------------------------------------------------------
    FUNCTION To_bitvector ( s : std_logic_vector ; xmap : BIT := '0')
RETURN BIT_VECTOR IS
        ALIAS sv : std_logic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
```

```
        FOR i IN result'RANGE LOOP
            CASE sv(i) IS
                WHEN '0' | 'L' => result(i) := '0';
                WHEN '1' | 'H' => result(i) := '1';
                WHEN OTHERS => result(i) := xmap;
            END CASE;
        END LOOP;
        RETURN result;
    END;
    ---------------------------------------------------------------------
    FUNCTION To_bitvector ( s : std_ulogic_vector; xmap : BIT := '0')
RETURN BIT_VECTOR IS
        ALIAS sv : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : BIT_VECTOR ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE sv(i) IS
                WHEN '0' | 'L' => result(i) := '0';
                WHEN '1' | 'H' => result(i) := '1';
                WHEN OTHERS => result(i) := xmap;
            END CASE;
        END LOOP;
        RETURN result;
    END;
    ---------------------------------------------------------------------
    FUNCTION To_StdULogic       ( b : BIT              ) RETURN std_ulogic
IS
    BEGIN
        CASE b IS
            WHEN '0' => RETURN '0';
            WHEN '1' => RETURN '1';
        END CASE;
    END;
    ---------------------------------------------------------------------
    FUNCTION To_StdLogicVector  ( b : BIT_VECTOR        ) RETURN
std_logic_vector IS
        ALIAS bv : BIT_VECTOR ( b'LENGTH-1 DOWNTO 0 ) IS b;
        VARIABLE result : std_logic_vector ( b'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
    ---------------------------------------------------------------------
    FUNCTION To_StdLogicVector  ( s : std_ulogic_vector ) RETURN
std_logic_vector IS
        ALIAS sv : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : std_logic_vector ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
```

```
            result(i) := sv(i);
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_StdULogicVector ( b : BIT_VECTOR        ) RETURN
std_ulogic_vector IS
        ALIAS bv : BIT_VECTOR ( b'LENGTH-1 DOWNTO 0 ) IS b;
        VARIABLE result : std_ulogic_vector ( b'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_StdULogicVector ( s : std_logic_vector ) RETURN
std_ulogic_vector IS
        ALIAS sv : std_logic_vector ( s'LENGTH-1 DOWNTO 0 ) IS s;
        VARIABLE result : std_ulogic_vector ( s'LENGTH-1 DOWNTO 0 );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := sv(i);
        END LOOP;
        RETURN result;
    END;

    -------------------------------------------------------------------
    -- strength strippers and type convertors
    -------------------------------------------------------------------
    -- to_x01
    -------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_logic_vector ) RETURN  std_logic_vector IS
        ALIAS sv : std_logic_vector ( 1 TO s'LENGTH ) IS s;
        VARIABLE result : std_logic_vector ( 1 TO s'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := cvt_to_x01 (sv(i));
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector IS
        ALIAS sv : std_ulogic_vector ( 1 TO s'LENGTH ) IS s;
        VARIABLE result : std_ulogic_vector ( 1 TO s'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := cvt_to_x01 (sv(i));
        END LOOP;
        RETURN result;
    END;
```

```
    ---------------------------------------------------------------------
    FUNCTION To_X01  ( s : std_ulogic ) RETURN  X01 IS
    BEGIN
        RETURN (cvt_to_x01(s));
    END;
    ---------------------------------------------------------------------
    FUNCTION To_X01  ( b : BIT_VECTOR ) RETURN  std_logic_vector IS
        ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
        VARIABLE result : std_logic_vector ( 1 TO b'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
    ---------------------------------------------------------------------
    FUNCTION To_X01  ( b : BIT_VECTOR ) RETURN  std_ulogic_vector IS
        ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
        VARIABLE result : std_ulogic_vector ( 1 TO b'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
    ---------------------------------------------------------------------
    FUNCTION To_X01  ( b : BIT ) RETURN  X01 IS
    BEGIN
            CASE b IS
                WHEN '0' => RETURN('0');
                WHEN '1' => RETURN('1');
            END CASE;
    END;
    ---------------------------------------------------------------------
    -- to_x01z
    ---------------------------------------------------------------------
    FUNCTION To_X01Z  ( s : std_logic_vector ) RETURN  std_logic_vector IS
        ALIAS sv : std_logic_vector ( 1 TO s'LENGTH ) IS s;
        VARIABLE result : std_logic_vector ( 1 TO s'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := cvt_to_x01z (sv(i));
        END LOOP;
        RETURN result;
    END;
    ---------------------------------------------------------------------
    FUNCTION To_X01Z  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector
IS
```

```
    ALIAS sv : std_ulogic_vector ( 1 TO s'LENGTH ) IS s;
    VARIABLE result : std_ulogic_vector ( 1 TO s'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        result(i) := cvt_to_x01z (sv(i));
    END LOOP;
    RETURN result;
END;
--------------------------------------------------------------------
FUNCTION To_X01Z  ( s : std_ulogic ) RETURN  X01Z IS
BEGIN
    RETURN (cvt_to_x01z(s));
END;
--------------------------------------------------------------------
FUNCTION To_X01Z  ( b : BIT_VECTOR ) RETURN  std_logic_vector IS
    ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
    VARIABLE result : std_logic_vector ( 1 TO b'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
END;
--------------------------------------------------------------------
FUNCTION To_X01Z  ( b : BIT_VECTOR ) RETURN  std_ulogic_vector IS
    ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
    VARIABLE result : std_ulogic_vector ( 1 TO b'LENGTH );
BEGIN
    FOR i IN result'RANGE LOOP
        CASE bv(i) IS
            WHEN '0' => result(i) := '0';
            WHEN '1' => result(i) := '1';
        END CASE;
    END LOOP;
    RETURN result;
END;
--------------------------------------------------------------------
FUNCTION To_X01Z  ( b : BIT ) RETURN  X01Z IS
BEGIN
        CASE b IS
            WHEN '0' => RETURN('0');
            WHEN '1' => RETURN('1');
        END CASE;
END;
--------------------------------------------------------------------
-- to_ux01
--------------------------------------------------------------------
FUNCTION To_UX01  ( s : std_logic_vector ) RETURN  std_logic_vector IS
    ALIAS sv : std_logic_vector ( 1 TO s'LENGTH ) IS s;
    VARIABLE result : std_logic_vector ( 1 TO s'LENGTH );
BEGIN
```

```
        FOR i IN result'RANGE LOOP
            result(i) := cvt_to_ux01 (sv(i));
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_UX01  ( s : std_ulogic_vector ) RETURN  std_ulogic_vector
IS
        ALIAS sv : std_ulogic_vector ( 1 TO s'LENGTH ) IS s;
        VARIABLE result : std_ulogic_vector ( 1 TO s'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            result(i) := cvt_to_ux01 (sv(i));
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_UX01  ( s : std_ulogic ) RETURN  UX01 IS
    BEGIN
        RETURN (cvt_to_ux01(s));
    END;
    -------------------------------------------------------------------
    FUNCTION To_UX01  ( b : BIT_VECTOR ) RETURN  std_logic_vector IS
        ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
        VARIABLE result : std_logic_vector ( 1 TO b'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_UX01  ( b : BIT_VECTOR ) RETURN  std_ulogic_vector IS
        ALIAS bv : BIT_VECTOR ( 1 TO b'LENGTH ) IS b;
        VARIABLE result : std_ulogic_vector ( 1 TO b'LENGTH );
    BEGIN
        FOR i IN result'RANGE LOOP
            CASE bv(i) IS
                WHEN '0' => result(i) := '0';
                WHEN '1' => result(i) := '1';
            END CASE;
        END LOOP;
        RETURN result;
    END;
    -------------------------------------------------------------------
    FUNCTION To_UX01  ( b : BIT ) RETURN  UX01 IS
    BEGIN
            CASE b IS
                WHEN '0' => RETURN('0');
                WHEN '1' => RETURN('1');
            END CASE;
```

```
    END;

    -------------------------------------------------------------------
    -- edge detection
    -------------------------------------------------------------------
    FUNCTION rising_edge  (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
    BEGIN
        RETURN (s'EVENT AND (To_X01(s) = '1') AND
                            (To_X01(s'LAST_VALUE) = '0'));
    END;

    FUNCTION falling_edge (SIGNAL s : std_ulogic) RETURN BOOLEAN IS
    BEGIN
        RETURN (s'EVENT AND (To_X01(s) = '0') AND
                            (To_X01(s'LAST_VALUE) = '1'));
    END;

    -------------------------------------------------------------------
    -- object contains an unknown
    -------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic_vector ) RETURN  BOOLEAN IS
    BEGIN
        FOR i IN s'RANGE LOOP
            CASE s(i) IS
                WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
                WHEN OTHERS => NULL;
            END CASE;
        END LOOP;
        RETURN FALSE;
    END;
    -------------------------------------------------------------------
    FUNCTION Is_X ( s : std_logic_vector  ) RETURN  BOOLEAN IS
    BEGIN
        FOR i IN s'RANGE LOOP
            CASE s(i) IS
                WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
                WHEN OTHERS => NULL;
            END CASE;
        END LOOP;
        RETURN FALSE;
    END;
    -------------------------------------------------------------------
    FUNCTION Is_X ( s : std_ulogic        ) RETURN  BOOLEAN IS
    BEGIN
        CASE s IS
            WHEN 'U' | 'X' | 'Z' | 'W' | '-' => RETURN TRUE;
            WHEN OTHERS => NULL;
        END CASE;
        RETURN FALSE;
    END;

END std_logic_1164;
```

Appendix D—Std_math Package

Below is a copy of the *std_math* package that is used in several of the designs in this text. This package is used to overload the + operator for operands of std_logic. This is not the the NUMERIC_STD package being worked on by the IEEE 1076.3 working group. For a draft copy of that standard, consult the World Wide Web (http://www.vhdl.org).

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

package std_math is
    FUNCTION inc_std(a      : STD_LOGIC_VECTOR)          RETURN
STD_LOGIC_VECTOR;
    FUNCTION std2i  (std : STD_LOGIC_VECTOR)              RETURN INTEGER;
    FUNCTION i2std  (VAL, width : INTEGER)               RETURN
STD_LOGIC_VECTOR;
    FUNCTION "+"    (a, b   : STD_LOGIC_VECTOR)          RETURN
STD_LOGIC_VECTOR;
    FUNCTION "+"    (a  : STD_LOGIC_VECTOR; b : INTEGER) RETURN
STD_LOGIC_VECTOR;
end std_math;

PACKAGE BODY std_math IS
    --   inc_std
    --  Increment std_logic_vector
    --  In:     std_logic_vector.
    --  Return: std_logic_vector.
    --
    FUNCTION inc_std     (a      : STD_LOGIC_VECTOR)          RETURN
STD_LOGIC_VECTOR IS
        VARIABLE s      : STD_LOGIC_VECTOR (a'RANGE);
        VARIABLE carry  : std_logic;
    BEGIN
        carry   := '1';

        FOR i IN a'LOW TO a'HIGH LOOP
            s(i)        := a(i) XOR carry;
            carry       := a(i) AND carry;
        END LOOP;

        RETURN  (s);
    END inc_std;

    -- "+"
    --  Add overload for:
    --  In:     two std_logic_vectors.
    --  Return: std_logic_vector.
    --
    FUNCTION "+"        (a, b   : STD_LOGIC_VECTOR)          RETURN
STD_LOGIC_VECTOR IS
        VARIABLE s      : STD_LOGIC_VECTOR (a'RANGE);
        VARIABLE carry  : STD_LOGIC;
        VARIABLE bi     : integer;       --  Indexes b.
```

```
    BEGIN
        carry     := '0';

        FOR i IN  a'LOW TO a'HIGH LOOP
            bi := b'low + (i - a'low);
            s(i)  := (a(i) XOR b(bi)) XOR carry;
            carry := ((a(i) OR b(bi)) AND carry) OR (a(i) AND b(bi));
        END LOOP;

        RETURN  (s);
    END "+";                              -- Two std_logic_vectors.

    -- std2i
    --  std_logic_vector to Integer.
    --  In:     std_logic_vector.
    --  Return: integer.
    --
    FUNCTION STD2I (std : std_logic_vector) RETURN integer IS
      VARIABLE result, abit : integer range 0 to 2**std'length - 1 := 0;
      VARIABLE  count            : integer := 0;
    BEGIN  --  STD2I
      bits : FOR I IN std'low to std'high LOOP
      abit := 0;
      IF ((std(I) = '1')) THEN
        abit := 2**(I - std'low);
      END IF;
        result := result + abit;                --  Add in bit if '1'.
      count := count + 1;
      EXIT bits WHEN count = 32;          --  32 bits max.
      END LOOP bits;
      RETURN (result);

    END STD2I;

    -- i2std
    --  Integer to Bit_vector.
    --  In:     integer, value and width.
    --  Return: std_logic_vector, with right bit the most significant.
    --
    FUNCTION i2std   (VAL, width : INTEGER)              RETURN
STD_LOGIC_VECTOR IS
        VARIABLE result : STD_LOGIC_VECTOR (0 to width-1) := (OTHERS=>'0');
        VARIABLE bits   : INTEGER := width;
    BEGIN
        IF (bits > 31) THEN               --  Avoid overflow errors.
          bits := 31;
        ELSE
          ASSERT 2**bits > VAL REPORT
            "Value too big FOR STD_LOGIC_VECTOR width"
            SEVERITY WARNING;
        END IF;

        FOR i IN 0 TO bits - 1 LOOP
            IF ((val/(2**i)) MOD 2 = 1) THEN
```

```
                result(i) := '1';
            END IF;

        END LOOP;

        RETURN (result);
    END i2std;

    -- "+"
    --  Overload "+" for std_logic_vector plus integer.
    --  In:     std_logic_vector and integer.
    --  Return: std_logic_vector.
    --
    FUNCTION "+"    (a  : STD_LOGIC_VECTOR; b : INTEGER)      RETURN
STD_LOGIC_VECTOR IS
    BEGIN
        RETURN (a + i2std(b, a'LENGTH));
    END "+";

end std_math;
```

Bibliography

The following list of books and articles on VHDL programmable logic, and computer networks is not intended to be comprehensive, but merely to point you to further reading that can help you become knowledgeable of programmable logic and Ethernet, and skilled in the use of VHDL simulation and synthesis tools.

Books:

Applications Handbook, Cypess Semiconductor Corporation, 1994.

Armstrong, J.R., *Chip-Level Modeling with VHDL*, Englewood cliffs, NJ: Prentice-Hall, 1988.

Barton, D., *A First Course in VHDL*, VLSI Systems Design, January 1988.

Bhasker, J., *A VHDL Primer*, Englewood Cliffs, NJ: Prentice Hall, 1992.

Brown, S., *Field-Programmable Devices*. 2nd ed. Stan Baker Associates

Carlson, S., *Introduction to HDL-Based Design Using VHDL*, Synopsys Inc., 1991.

Coelho, D., *The VHDL Handbook*, Boston: Kluwer Academic, 1988.

Data Book, Altera Corporation , San Jose, 1993.

Harr, R. E., et al., *Applications of VHDL to Circuit Design*, Boston: Kluwer Academic, 1991.

IEEE Standard VHDL Language Reference Manual, Std 1076-1993, IEEE, NY, 1993.

IEEE Standard 1076 VHDL Tutorial, CLSI, Maryland, March 1989.

IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164), Std IEEE 1164-1993, IEEE, NY, 1993.

Leung, S., *ASIC System Design With VHDL*, Boston: Kluwer Academic, 1989.

Leung, S., and M. Shanblatt, *ASIC System Design With VHDL: A Paradigm*, Boston: Kluwer Academic, 1989.

Lipsett, R., C. Shaefer, and C. Ussery, *VHDL: Hardware Description and Design*, Boston: Kluwer Academic, 1989.

MAC Parameters, Physical Layer, Medium Attachment Units and Repeater for 100 Mb/s Operation, Supplement to Std IEEE 802.3-1993, IEEE, NY, 1995.

Mazor, S., and P. Langstraat, *A Guide to VHDL*, Boston: Kluwer Academic, 1992.

IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164), Std IEEE 1164-1993

PAL Device Data Book and Design Guide, Advanced Micro Devices, Inc., 1995.

Perry, D., *VHDL*, New York: McGraw-Hill, 1991.

Programmable Logic Data Book, Cypress Semiconductor Corporation, 1994.

Programmable Logic Data Book, Xilinx, Inc. 1994.

Schoen, J.M., *Performance and Fault Modeling with VHDL*, Englewood Cliffs, NJ: Prentice Hall, 1992.

Tanenbaum, Andrew, *Computer Networks*, 2nd ed. Prentice Hall, 1988.

Sternheim, Eli *et al.Digital Design and Synthesis*, Automa Publishing Company, San Jose, 1993.

Articles:

Armstrong, J.R., et al., "The VHDL Validation Suite," *Proc. 27th Design Automation Conference*, June 1990, pp. 2-7.

Barton, D., "Behavioral Descriptions in VHDL," *VLSI Systems Design*, June 1988.

Bhasker, J., "Process-Graph Analyzer: A Front-End Tool for VHDL Behavioral Synthesis," *Software Practice and Experience*, vol. 18, no. 5, May 1988.

Bhasker, J., "An Algorithm for Microcode Compaction of VHDL Behavioral Descriptions," *Proc. 20th Microprogramming Workshop*, December 1987.

Coelho, D., "VHDL: A Call for Standards," *Proc. 25th Design Automation Conference*, June 1988.

Coppola, A., and J. Lewis, "VHDL for Programmable Logic Devices," *PLDCON'93*, Santa Clara, March 29-31, 1993.

Coppola, A., J. Freedman, et al., "Tokenized State Machines for PLDs and FPGAs," *Proceedings of IFIP WG10.2/WG10.5 Workshop on Control-Dominated Synthesis from a Register-Transfer-Level Description*, Grenoble, France, 3-4 September, 1992, G. Saucier and J. Trilhe, editors, Elsevier Science Publishers.

Coppola, A., J. Freedman, et al., "VHDL Synthesis of Concurrent State Machines to a Programmable Logic Device," *Proc. of the IEEE VHDL International User's Forum*, May 3-6, 1992, Scottsdale, Arizona.

Coppola, A., and M. Perkowski, "A State Machine PLD and Associated Minimization Algorithms," *Proc. of the FPGA'92 ACM/SIGDA First International Workshop on Field-Programmable Gate Arrays*, Berkeley, California, Feb. 16-18, 1992, pp. 109-114.

Dillinger, T.E., et al., "A Logic Synthesis System for VHDL Design Description," *IEEE ICCAD-89*, Santa Clara, California.

Farrow, R., and A. Stanculescu, "A VHDL Compiler Based on Attribute Grammar Methodology," *SIGPLAN 1989*.

Gilman, A.S., "Logic Modeling in WAVES," *IEEE Design and Test of Computers*, June 1990, pp. 49-55.

Hands, J.P., "What Is VHDL?", *Computer-Aided Design*, vol. 22, no. 4, May 1990.

Hines, J., "Where VHDL Fits Within the CAD Environment," *Proc. 24th Design Automation Conference*, 1987.

Kim, K., and J. Trout, "Automatic Insertion of BIST Hardware Using VHDL," *Proc. 25th Design Automation Conference*, 1988.

Moughzail, M., et al., "Experience with the VHDL Environment," *Proc. 25th Design Automation Conference*, June 1988.

Saunders, L., "The IBM VHDL Design System," *Proc. 24th Design Automation Conference*, 1987.

Ward, P.C., and J. Armstrong, "Behavioral Fault Simulation in VHDL," *Proc. 27th Design Automation Conference*, June 1990, pp. 587-593.

Index

J

K

L

T

X

Z

Language Constructs

Concurrent Statements

<table>
<tr><td colspan="2">Boolean equations</td></tr>
<tr><td>relation { and relation}
| relation { or relation }
| relation { xor relation }
| relation { nand relation }
| relation { nor relation }</td><td>v <= a and b and c;
w <= a or b or c;
x <= a xor b xor c;
y <= a nand b nand c;
z <= a nor b nor c;</td></tr>
<tr><td colspan="2">Conditional signal assignment (WHEN-ELSE)</td></tr>
<tr><td>{expression when condition else} expression;</td><td>a <= '1' when b = c else '0';</td></tr>
<tr><td colspan="2">Selected signal assignment (WITH-SELECT-WHEN)</td></tr>
<tr><td>with selection_expression select
{identifier <= expression when
identifier | expression | discrete_range | others,}
identifier <= expression when
identifier | expression | discrete_range | others;</td><td>architecture archfsm of fsm is
type state_type is (st0, st1, st2, st3);
signal state: state_type;
signal y, z: std_logic;
begin
with state select
x <= "0000" when st0 | st1;
"0010" when st2 | st3;
y and z when others;
end archfsm;</td></tr>
<tr><td colspan="2">Component instantiation — see above</td></tr>
<tr><td colspan="2">Generate scheme for component instantiation or generation of equations</td></tr>
<tr><td>generate_label: (for identifier in discrete_range)
| (if condition) generate
{concurrent_statement}
end generate [generate_label] ;</td><td>g1: for i in 0 to 7 generate
reg1: register8 port map (clock, reset, enable,
data_in(i), data_out(i));
end generate g1;

g2: for j in 0 to 2 generate
a(j) <= b(j) xor c(j);
end generate g2;</td></tr>
</table>

Sequential Statements

Process statement

```
[process_label:]
process (sensitivity list)
    {type_declaration
    | constant_declaration
    | variable_declaration
    | alias_declaration}
begin
    {wait_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | if_statement
    | case_statement
    | loop_statement
end process [process_label];
```

```
my_process:
process (rst, clk)
  constant zilch : std_logic_vector(7 downto 0) := "0000_0000";
begin
    wait until clk = '1';
        if (rst = '1') then
            q <= zilch;
        elsif (en = '1') then
            q <= data;
        else
            q <= q;
        end if;
end my_process;
```

IF-THEN-ELSE-ELSIF statement

```
if condition then sequence_of_statements
{elsif condition then sequence_of statements}
[else sequence_of_statements]
end if;
```

```
if (count = "00") then
    a <= b;
elsif (count = "10") then
    a <= c;
else
    a <= d;
end if;
```

CASE-WHEN statement

```
case expression is
    {when identifier | expression | discrete_range | others =>
            sequence_of_statements}
end case;
```

```
case count is
    when "00" =>
        a <= b;
    when "10" =>
        a <= c;
    when others =>
        a <= d;
end case;
```

FOR LOOP statement

```
[loop_label:]
    for identifier in discrete_range loop
        sequence_of_statements
    end loop [loop_label];
```

```
my_for_loop:
for i in 3 downto 0 loop
    if reset(i) = '1' then
        data_out(i) := '0';
    end if;
end loop my_for_loop;
```

WHILE LOOP statement

```
[loop_label:]
    while condition loop
        sequence_of_statements
    end loop [loop_label];
```

```
my_while_loop:
    while (count > 0) loop
        count := count - 1;
        result <= result + data_in;
    end loop my_while_loop;
```

Cypress Semiconductor
3901 North First Street
San Jose, CA 95134
Tel: (408) 943-2600
Fax: (408) 943-2741

1-895VHDL 3000