Preface

The Verilog language is a hardware description language that provides a means of specifying a digital system at a wide range of levels of abstraction. The language supports the early conceptual stages of design with its behavioral level of abstraction, and the later implementation stages with its structural abstractions. The language includes hierarchical constructs, allowing the designer to control a description's complexity.

Verilog was originally designed in the winter of 1983/84 as a proprietary verification/simulation product. Later, several other proprietary analysis tools were developed around the language, including a fault simulator and a timing analyzer. More recently, Verilog has also provided the input specification for logic and behavioral synthesis tools. The Verilog language has been instrumental in providing consistency across these tools. The language was originally standardized as IEEE standard #1364-1995. It has recently been revised and standardized as IEEE standard #1364-2001. This book presents this latest revision of the language, providing material for the beginning student and advanced user of the language.

It is sometimes difficult to separate the language from the simulator tool because the dynamic aspects of the language are defined by the way the simulator works. Further, it is difficult to separate it from a synthesis tool because the semantics of the language become limited by what a synthesis tool allows in its input specification and produces as an implementation. Where possible, we have stayed away from simulator- and synthesis-specific details and concentrated on design specification. But, we have included enough information to be able to write working executable models.

The book takes a tutorial approach to presenting the language. Indeed, we start with a tutorial introduction that presents, via examples, the major features of the language and the prevalent styles of describing systems. We follow this with a detailed presentation on using the language for synthesizing combinational and sequential systems. We then continue with a more complete discussion of the language constructs.

Our approach is to provide a means of learning by observing the examples and doing exercises. Numerous examples are provided to allow the reader to learn (and relearn!) easily by example. It is strongly recommended that you try the exercises as early as possible with the aid of a Verilog simulator. The examples shown in the book are available in electronic form on the enclosed CD. Also included on the CD is a simulator. The simulator is limited in the size of description it will handle.

The majority of the book assumes a knowledge of introductory logic design and software programming. As such, the book is of use to practicing integrated circuit design engineers, and undergraduate and graduate electrical or computer engineering students. The tutorial introduction is organized in a manner appropriate for use with a course in introductory logic design. A separate appendix, keyed into the tutorial introduction, provides solved exercises that discuss common errors. The book has also been used for courses in introductory and upper level logic and integrated circuit design, computer architecture, and computer-aided design (CAD). It provides complete coverage of the language for design courses, and how a simulator works for CAD courses. For those familiar with the language, we provide a preface that covers most of the new additions to the 2001 language standard.

The book is organized into eleven chapters and eight appendices. The first part of the book contains a tutorial introduction to the language which is followed by a chapter on its use for logic synthesis. The second part of the book, Chapters 3 through 6, provide a more rigorous presentation of the language's behavioral, hierarchical, and logic level modeling constructs. The third part of the book, Chapters 7 through 11, covers the more specialized topics of cycle-accurate modeling, timing and event driven simulation, user-defined primitives, and switch level modeling. Chapter 11 suggests two major Verilog projects for use in a university course. One appendix provides tutorial discussion for beginning students. The others are reserved for the dryer topics typically found in a language manual; read those at your own risk.

Have fun designing great systems...

always,

Donald E. Thomas
Philip R. Moorby
March 2002

2 Logic Synthesis

In this chapter, the use of the language as an input specification for synthesis is presented. The concern is developing a functionally correct specification while allowing a synthesis CAD tool to design the final gate level structure of the system. Care must be taken in writing a description so that it can be used in both simulation and synthesis.

2.1 Overview of Synthesis

The predominate synthesis technology in use today is *logic synthesis*. A system is specified at the register-transfer level of design; by using logic synthesis tools, a gate level implementation of the system can be obtained. The synthesis tools are capable of optimizing a design with respect to various constraints, including timing and/or area. They use a technology library file to specify the components to be used in the design.

2.1.1 Register-Transfer Level Systems

A register-transfer level description may contain parts that are purely combinational while others may specify sequential elements such as latches and flip flops. There may also be a finite state machine description, specifying a state transition graph.

A logic synthesis tool compiles a register-transfer level design using two main phases. The first is a technology independent phase where the design is read in and manipulated without regard to the final implementation technology. In this phase, major simplifications in the combinational logic may be made. The second phase is technology mapping where the design is transformed to match the components in a component library. If there are only two-input gates in the library, the design is transformed so that each logic function is implementable by a component in the library. Indeed, synthesis tools can transform one gate level description into another, providing the capability of redesigning a circuit when a new technology library is used.

The attraction of a logic synthesis CAD tool is that it aids in a very complex design process. (After all, did your logic design professor ever tell you what to do when the Karnaugh map had more than five or six variables!) These tools target large combinational design and different technology libraries, providing implementation trade-offs in time and area. Further, they promise functional equivalence of the initial specification and its resulting implementation. Given the complexity of this level of design, these tools improve the productivity of designers in many common design situations.

To obtain this increased productivity, we must specify our design in a way that it can be simulated for functional correctness and then synthesized. This chapter discusses methods of describing register-transfer level systems for input to logic synthesis tools.

2.1.2 Disclaimer

The first part of this chapter defines what a *synthesizable description* for logic synthesis is. There are behaviors that we can describe but that common logic synthesis tools will not be able to design. (Or they may design something you'd want your competitor to implement!) Since synthesis technology is still young, and the task of mapping an arbitrary behavior on to a set of library components is complex, arbitrary behavior specifications are not allowed as inputs to logic synthesis tools. Thus, only a subset of the language may be used for logic synthesis, and the style of writing a description using that subset is restricted. The first part of this chapter describes the subset and restrictions commonly found in logic synthesis specification today. Our discussion of logic synthesis is based on experience using current tools. If you use others, your mileage may vary. Read the synthesis tool manual closely.

2.2 Combinational Logic Using Gates and Continuous Assign

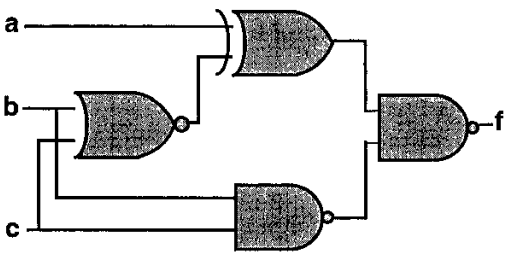
Using gate primitives and continuous assignment statements to specify a logic function for logic synthesis is quite straightforward. Examples 2.1 and 2.2 illustrate two synthesizable descriptions in this style. Both of the examples implement the same combinational function; the standard sum-of-products specification is:

$$f(a, b, c) = \Sigma\, m(a, b, c) = \Sigma\, m(1, 2, 3, 4, 7).$$

Essentially, logic synthesis tools read the logic functionality of the specification and try to optimize the final gate level design with respect to design constraints and library elements. Even though Example 2.1 specifies a gate level design, a logic synthesis tool is free, and possibly constrained, to implement the functionality using different gate primitives. The example shows a different, but functionally equivalent, gate level design. Here, the technology library only contained two-input gates; the synthesis tool transformed the design to the implementation on the right of the example. Other designs are possible with alternate libraries and performance constraints.

```
module synGate
     (output f,
      input  a, b, c);

     and    A (a1, a, b, c);
     and    B (a2, a, ~b, ~c);
     and    C (a3, ~a, o1);
     or     D (o1, b, c);
     or     E (f, a1, a2, a3);
endmodule
```



Example 2.1 A Description and Its Synthesized Implementation

The example does not contain delay (#) information, illustrating one of the key differences between writing Verilog descriptions for simulation and synthesis. In simulation, we normally provide detailed timing information to the simulator to help the designer with the task of timing verification. A logic synthesis tool will ignore these timing specifications, using only the functional specification provided in the description. Because timing specifications are ignored, having them in a description could give rise to differences in simulating a design being input to a logic synthesis tool versus simulating the resulting implementation.

Consider gate instance A in Example 2.1. If it had been specified as:

```
and     #5   A (a1, a, b, c);
```

then simulation of the description would have shown a 5 time unit delay between changes on the input to changes on the output of this gate. The implementation shown in Example 2.1 does not have a gate corresponding to A. Thus, the timing of the simulation of that implementation would be different. Logic synthesis does not try to meet such timing specifications. Rather, synthesis tools provide means of specifying timing requirements such as the clock period. The tool will then try to design the logic so that all set-up times are met within that clock period.

```
module synAssign
     (output  f,
      input   a, b, c);

     assign   f = (a & b & c) | (a & ~b & ~c) | (~a & (b | c));
endmodule
```

Example 2.2 A Synthesizable Description Using Continuous Assign

Using a continuous assign statement, as shown in Example 2.2, is similar to specifying logic in Boolean algebra, except Verilog has far more operators to use in the specification. The assign statement allows us to describe a combinational logic function without regard to its actual structural implementation — that is, there are no instantiated gates with wires and port connections. In a simulation of the circuit, the result of the logical expression on the right-hand side of the equal sign is evaluated anytime one of its values changes and the result drives the output f.

In this example, the same sum of products functionality from Example 2.1 is used but the assign statement is written combining products 1, 2, and 3 into the last product term. Of note is the fact that a continuous assign may call a function which contains procedural assignment statements. The use of procedural assignment statements to describe combinational logic will be discussed in section 2.3; thus we will limit the discussion here to continuous assigns without function calls.

Continuous assign statements are often used for describing datapath elements. These modules tend to have one-line specifications as compared to the logic specifications for next state and output logic in a finite state machine. In Example 2.3 both an adder and a multiplexor are described with continuous assign. The **addWithAssign** module is parameterized with the width of the words being added and include carry in (**Cin**) and carry out (**carry**) ports. Note that the sum generated on the right-hand side of the assign generates a result larger than output **sum**. The concatenation operator specifies that the topmost bit (the carry out) will drive the **carry** output and the rest of the bits will drive the **sum** output. The multiplexor is described using the conditional operator.

```
module addWithAssign
    #(parameter WIDTH = 4)
    (output                 carry,
     output [WIDTH-1:0]     sum,
     input  [WIDTH-1:0]     A, B,
     input                  Cin);

    assign  {carry, sum} = A + B + Cin;
endmodule

module muxWithAssign
    #(parameter WIDTH = 4)
     (output [WIDTH-1:0]    out,
     input  [WIDTH-1:0]     A, B,
     input                  sel);

    assign  out = (sel) ? A: B;
endmodule
```

Example 2.3 Datapath Elements Described With Continuous Assign

There are limits on the operators that may be used as well as the ways in which unknowns (x) are used. An unknown may be used in a synthesizable description but only in certain situations. The following fragment is not synthesizable because it compares a value to an unknown.

```
assign    y = (a === 1'bx)? c : 1 ;
```

An unknown used in this manner is a value in a simulator; it is useful in determining if the value of a has become unknown. But we do not build digital hardware to compare with unknowns and thus this construct is not synthesizable. However, the following fragment, using an unknown in a non-comparison fashion, is allowable:

```
assign    y = (a == b) ? 1'bx : c ;
```

In this case, we are specifying a don't-care situation to the logic synthesizer. That is, when a equals b, we don't care what value is assigned to y. If they are not equal, the value c is assigned. In the hardware synthesized from this assign statement, either 1 or 0 will be assigned to y (after all, there are no unknowns in real hardware). A don't-care specification used in this manner allows the synthesizer additional freedom in optimizing a logic circuit. The best implementation of this specification is just y = c.

References: assign 6.3; primitive gates 6.2; parameters 5.2.

2.3 Procedural Statements to Specify Combinational Logic

In addition to using continuous assign statements and primitive gate instantiations to specify combinational logic, procedural statements may be used. The procedural statements are specified in an always statement, within a task called from an always statement, or within a function called from an always statement or a continuous assign. In spite of the fact that a description using procedural statements appears sequential, combinational logic may be specified with them. Section 1.2 introduced this approach to specifying combinational logic. This section covers the topic in more detail.

2.3.1 The Basics

The basic form of a procedural description of combinational logic is shown in Example 2.4. It includes an always statement with an event statement containing all of the input variables to the combinational function. The example shows a multiplexor described procedurally. In this case, input a selects between passing inputs b or c to output f. Even though f is defined to be a register, a synthesis tool will treat this module as a specification of combinational logic.

```
module synCombinationalAlways
    (output reg f,
     input      a, b, c);

    always @ (a, b, c)
      if (a == 1)
            f = b;
      else
            f = c;
endmodule
```

Example 2.4 Combinational Logic Described With Procedural Statements

A few definitions will clarify the rules on how to read and write such descriptions. Let's define the *input set* of the always block to be the set of all registers, wires, and inputs used on the right-hand side of the procedural statements in the always block. In Example 2.4, the input set contains a, b, and c. Further, let's define the *sensitivity list* of an always block to be the list of names appearing in the event statement ("@"). In this example, the sensitivity list contains a, b, and c. When describing combinational logic using procedural statements, every element of the always block's input set must appear without any edge specifiers (e.g., posedge) in the sensitivity list of the event statement. This follows from the very definition of combinational logic — *any* change of *any* input value may have an immediate effect on the resulting output. If an element of the input set is not in the sensitivity list, or only one edge-change is specified, then it cannot have an immediate effect. Rather, it must always wait for some other input to change; this is not true of combinational circuits.

Considering Example 2.4 further, we note that the combinational output f is assigned in every branch of the always block. A *control path* is defined to be a sequence of operations performed when executing an always loop. There may be many different control paths in an always block due to the fact that conditional statements (e.g. case and if) may be used. The output of the combinational function must be assigned in each and every one of the possible control paths. Thus, for every conceivable input change, the combinational output will be calculated anew; this is a characteristic of combinational logic.

The above example and discussion essentially outline the rules for specifying combinational hardware using procedural statements: the sensitivity list must be the input set and contain no edge-sensitive specifiers, and the combinational output(s) must be assigned to in every control path.

A common error in specifying combinational circuits with procedural statements is to incorrectly specify the sensitivity list. Example 2.4 is revised to use the @(*) construct as shown in Example 2.5 — the two examples will simulate and synthesize identically. Essentially, @(*) is shorthand for "all the signals on the right-hand side of the statement or in a conditional expression."

```
module synAutoSensitivity
    (output reg f,
     input      a, b, c);

    always @ (*)
      if (a == 1)
          f = b;
      else
          f = c;
endmodule
```

Example 2.5 Automatically Determining the Sensitivity List

The basic form of the "@" event statement is:

@ (sensitivity_list) statement;

When using the construct @(*) — or @* which is equivalent — only the statement's right-hand side or conditional expression is included. Thus, if several procedural statements are needed to specify the combinational function, a begin-end block must be used to group them into a compound statement. The "@(*) begin-end" will then include the registers and nets from the right-hand sides and conditionals of all of the statements in the compound statement.

Although this relieves the problem of correctly specifying the sensitivity list for combinational functions, the rule concerning assigning to the combinational output(s) during any execution of the always block must still be followed. An approach to organizing descriptions so that an assignment is always made is shown in Example 2.6. This module has the same multiplexor functionality as Example 2.5. However, here the output f is assigned to first. In a complex description, this approach ensures that a latch will not be inferred because of a forgotten output assignment.

```
module synAssignOutputFirst
    (output reg f,
     input      a, b, c);

    always @ (*) begin
        f = c;
        if (a == 1)
            f = b;
    end
endmodule
```

Example 2.6 Automatically Determining the Sensitivity List

References: always 3.1; sensitivity list 8.1; @ 4.2; edge specifications 4.2; input set 7.2.1, functions and tasks 3.5.

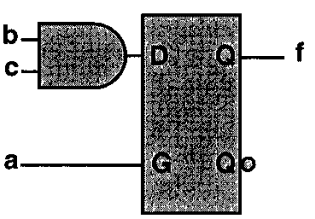
2.3.2 Complications — Inferred Latches

If there exists a control path that does not assign to the output, then the previous output value needs to be remembered. This is not a characteristic of combinational hardware. Rather it is indicative of a sequential system where the previous state is remembered in a latch and gated to the output when the inputs specify this control path. A logic synthesis tool will recognize this situation and infer that a latch is needed in the circuit. Assuming that we are trying to describe combinational hardware, we want to insure that this *inferred* latch is not added to our design. Assigning to the combinational output in every control path will insure this.

An example of a situation that infers a latch is shown in Example 2.7. If we follow the control paths in this example, we see that if **a** is equal to one, then **f** is assigned the value of **b** & **c**. However, if **a** is equal to zero, then **f** is not assigned to in the execution of the always block. Thus, there is a control path in which f is not assigned to. In this case a latch is inferred and the circuit shown on the right of the example is synthesized. The latch is actually a gated latch — a level-sensitive device

```
module synInferredLatch
    (output reg f,
     input      a, b, c);

    always @(*)
        if (a == 1)
            f = b & c;
endmodule
```



Example 2.7 An Inferred Latch

that passes the value on its input D when the latch's gate input (G which is connected to a) is one, and holds the value when the latch's gate input is zero.

2.3.3 Using Case Statements

Example 2.8 illustrates using a case statement to specify a combinational function in a truth table form. (This example specifies the same logic function as Examples 2.1 and 2.2.) The example illustrates and follows the rules for specifying combinational logic using procedural statements: all members of the always' input set are contained in the always' sensitivity list — the @(*) insures this, the combinational output is assigned to in every control path, and there are no edge specifications in the sensitivity list.

```
module synCase
    (output reg f,
     input      a, b, c);

    always @(*)
      case ({a, b, c})
            3'b000:   f = 1'b0;
            3'b001:   f = 1'b1;
            3'b010:   f = 1'b1;
            3'b011:   f = 1'b1;
            3'b100:   f = 1'b1;
            3'b101:   f = 1'b0;
            3'b110:   f = 1'b0;
            3'b111:   f = 1'b1;
      endcase
endmodule
```

Example 2.8 Combinational Logic Specified With a Case Statement

The first line of the case specifies the concatenation of inputs a, b, and c as the means to select a case item to execute. The line following the case keyword specifies a numeric value followed by a colon. The number 3'b000 is the Verilog notation for a 3-bit number, specified here in binary as 000. The b indicates binary. The right-hand sides of the assignments to f need not be constants. Other expressions may be used that include other names. The @(*) will include them in the sensitivity list.

Of course, when using a case statement it is possible to incompletely specify the case. If there are n bits in the case's controlling expression, then a synthesis tool will know that there are 2^n possible control paths through the case. If not all of them are specified, then there will be a control path in which the output is not assigned to; a latch will be inferred. The default case item can be used to define the remaining unspecified case items. Thus Example 2.8 could also be written as shown in Example 2.9. Here, we explicitly list all of the zeros of the function using

```
module synCaseWithDefault
    (output reg f,
     input      a, b, c);

    always @(a, b, c)
      case ({a, b, c})
            3'b000:   f = 1'b0;
            3'b101:   f = 1'b0;
            3'b110:   f = 1'b0;
            default:  f = 1'b1;
      endcase
endmodule
```

Example 2.9 Using Default to Fully Specify a Case Statement

separate case items. If the input does not match one of these items, then by default f is assigned the value one.

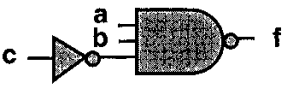
References: case 3.4, numbers B.3.

2.3.4 Specifying Don't Care Situations

Logic synthesis tools make great use of logical don't care situations to optimize a logic circuit. Example 2.10 illustrates specifying a logic function that contains a don't care. Often these can be specified in the default statement of a case. As shown, assigning the value **x** to the output is interpreted in this example as specifying input cases 3'b000 and 3'b101 to be don't cares. An optimized implementation of this function is shown on the right; only the single zero of the function (input case 3'b110) is implemented and inverted. In general, specifying an input to be **x** allows the synthesis tool to treat it as a logic don't care specification.

```
module synCaseWithDC
    (output reg f,
     input      a, b, c);

    always @(*)
        case ({a, b, c})
            3'b001:   f = 1'b1;
            3'b010:   f = 1'b1;
            3'b011:   f = 1'b1;
            3'b100:   f = 1'b1;
            3'b110:   f = 1'b0;
            3'b111:   f = 1'b1;
            default:  f = 1'bx;
        endcase
endmodule
```



Example 2.10 A Logic Function With a Don't Care

Two *attributes* are often used to help synthesis tools optimize a function. These are the *full_case* and *parallel_case* attributes illustrated in Example 2.11. The case statement in Example 2.10 is full by definition because all of the case items are specified either explicitly or by using a default. Thus all of the control paths are also specified. Synthesis tools look for the full_case attribute specified on a case statement indicating that the case is to be considered full even though all case items are not specified. In this situation, the unspecified cases are considered to be don't cares for synthesis, and a latch is not inferred.

An attribute is specified as shown on line 6 of Example 2.11. Attributes are declared as a prefix to the statement to which they refer; in this situation it is on the line before the case statement it refers to. (Attributes are not comments, nor are their names defined by the language. Rather, other tools that use the language, such as a synthesis tool, define their names and meanings. Consult their user manuals for details and examples of usage.)

```
module synAttributes
    (output reg f,
     input      a, b, c);

    always @(*)
      (* full_case, parallel_case *)
      case ({a, b, c})
            3'b001:   f = 1'b1;
            3'b010:   f = 1'b1;
            3'b011:   f = 1'b1;
            3'b100:   f = 1'b1;
            3'b110:   f = 1'b0;
            3'b111:   f = 1'b1;
      endcase
endmodule
```

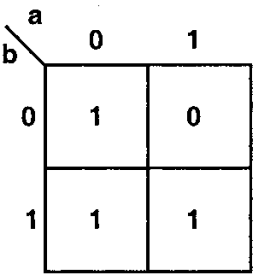
Example 2.11 Case Attributes

Also shown in the example is a parallel case attribute. A Verilog case statement is allowed to have overlapping case items. In this situation, the statements for the matching items are executed in the order specified. This can result in some complex logic because a priority among the case items is specified. A parallel case is a case statement where there is no overlap among the case items. That is, only one of the case items can be true at any time. If the case is parallel (and full), it can be regarded as a sum-of-products specification which could be implemented by a multiplexor. Specifying the parallel case attribute enables this interpretation and generally simplifies the logic generated.

A casex statement, which allows for the use of x, z, or ? in the controlling expression or in a case-item expression, can be used for specifying don't cares for synthesis. However, x, z, or ? may only be specified in a case item expression for synthesis.

```
module synUsingDC
    (output reg f,
     input      a, b);

    always @(*)
      casex ({a, b})
            2'b0?:  f = 1;
            2'b10:  f = 0;
            2'b11:  f = 1;
      endcase
endmodule
```



Example 2.12 Specifying a Logical Function Using a

Consider the module shown in Example 2.12. The first case item specifies that if a is zero, then the output f is one. The use of the ? in this statement specifies that the value of b does not matter in this situation. Thus this case item covers the first column of the Karnough map. Although we could have specified the two case items (2'b00 and 2'b01) and assigned f to be x in both situations, the approach shown is more compact. Since this first case item covers

two of the four possible case-items, it along with the other two case-items make this a full casex.

When using the ? in the case-item expressions, the case items can overlap. Example 2.13 illustrates how a one-hot state assignment could lead to overlapping case items. Specify the case with the full and parallel_case attributes; the synthesizer will then treat each case as exclusive and generate more optimized logic.

```
module oneHotEncoding
    (output reg [2:0] state,
     input            in, ck);

    always @(posedge ck)
      (* full_case, parallel_case *)
      casex (state)
          3'b1??: state <= 3'b010;
          3'b?1?: state <= in ? 3'b010: 3'b001;
          3'b??1: state <= in ? 3'b100: 3'b001;
      endcase
endmodule
```

Example 2.13 Use of Full and Parallel Case

The casez statement can also be used to specify logical don't cares. In this situation, only z or ? are used for the don't care in the case-item expression.

Care should be taken when using don't cares in a specification because they give rise to differences between simulation and synthesis. In a simulator, an x is one of the four defined logic values that will be printed when tracing values. However, in the synthesized circuit, the value printed for the same situation will either be 1 or 0. Further, comparing to an x makes sense in a simulation but not in synthesis. To reduce the differences between simulation and synthesis, a synthesizable description does not compare with x or z.

References: casex and casez 3.4.

2.3.5 Procedural Loop Constructs

A reading of the above examples might suggest that the only means to specify logic functions is through if and case statements. The for loop in Verilog may be used to specify combinational logic. The while and forever loops are used for synthesizing sequential systems. The repeat loop is not allowed in any synthesizable specifications.

For loops allow for a repetitive specification as shown in Example 2.14 (Generate loops are discussed in more detail in Section 5.4.) In this example, each iteration of the loop specifies a different logic element indexed by the loop variable **i**. Thus, eight xor gates are connected between the inputs and the outputs. Since this is a specification of combinational logic, **i** does not appear as a register in the final implementation.

```
module synXor8
    (output reg [1:8] xout,
     input      [1:8] xin1, xin2);

    reg         [1:8] i;

    always @(*)
        for (i = 1; i <= 8; i = i + 1)
            xout[i] = xin1[i] ^ xin2[i];
endmodule
```

Example 2.14 Using for to Specify an Array

The example illustrates several points about using for statements for specifying logic. The for loop is highly structured, clearly specifying the step variable and its limits. It will have an index **i** that must either start with a low limit and step up to a high limit, or start with a high limit and step down to a low limit. The comparison for end of loop may be <, >, <=, or >=, and the step size need not be one. The general form shown below illustrates the count down version:

```
for (i = highLimit; i >= lowLimit; i = i - step);
    ...
```

Example 2.15 shows a more complex design. The design is of a digital correlator which takes two inputs (**message** and **pattern**) and counts the number of bits that match. If **message** was 8'b00001111 and pattern was 8'b01010101, then the number of bits that match is four. At first glance this module appears to be a sequential algorithm. However, the for loop specifies a cascade of adders summing up the correlations of each bit-pair; a combinational circuit results.

The bitwidth of the inputs and outputs are parameterized. Starting with bit position zero, the two inputs are XNOR'd together producing their correlation — **1** if the input bits are the same, else **0**. The next iteration of the for loop specifies another correlation, this time of bit one of **message** and **pattern**; this correlation is added with the previous result. The result of all iterations of the for loop is to specify **dataWidth** levels of adders. A logic synthesizer can work hard on optimizing that! When simulated, the initialization to **matchCount** starts it at zero.

References: Unallowed constructs 2.8, parameters 5.2, generate 5.4.

```
module DigitalCorrelator
    #(parameter                          dataWidth = 40,
                                         countWidth= 6,)
    (output reg  [countWidth-1:0] matchCount = 0,
     input       [dataWidth-1:0]  message, pattern);

    int      i;

    always @(*) begin
       for (i = 0; i < dataWidth; i = i + 1)
            matchCount = matchCount + ~(message[i] ^ pattern[i]);
    end
endmodule
```

Example 2.15 Digital Correlator

2.4 Inferring Sequential Elements

Sequential elements are the latches and flip flops that make up the storage elements of a register-transfer level system. Although they are a fundamental component of a digital system, they are difficult to describe to a synthesis tool; the main reason being that their behavior can be quite intricate. The form of the description of some of these elements (especially flip flops) are almost prescribed so that the synthesis tool will know which library element to map the behavior to.

2.4.1 Latch Inferences

Latches are *level sensitive* storage devices. Typically, their behavior is controlled by a system wide clock that is connected to a gate input (G). While the gate is asserted (either high or low), the output Q of the latch follows the input D — it is a combinational function of D. When the gate is unasserted, the output Q remembers the last value of the D input. Sometimes these devices have asynchronous set and/or reset inputs. As we have seen in section 2.3.2, latches are not explicitly specified. Rather, they arise by inference from the way in which a description is written. We say that latches are *inferred*. One example of an inferred latch was shown in Example 2.7.

Latches are inferred using the always statement as a basis. Within an always statement, we define a *control path* to be a sequence of operations performed when executing an always loop. There may be many different control paths in an always block due to the fact that conditional statements (e.g. case and if) may be used. To produce a combinational circuit using procedural statements, the output of the combinational

function must be assigned in each and every one of the different control paths. Thus, for every conceivable input change, the combinational output will be calculated anew.

To infer a latch, two situations must exist in the always statement: at least one control path must exist that does not assign to an output, and the sensitivity list must not contain any edge-sensitive specifications. The first gives rise to the fact that the previous output value needs to be remembered. The second leads to the use of level-sensitive latches (as opposed to edge-sensitive flip flops). The requirement for memory is indicative of a sequential element where the previous state is remembered in a latch when the inputs specify this control path. A logic synthesis tool will recognize this situation and infer that a latch is needed in the circuit. Assuming that we are trying to describe a sequential element, leaving the output variable unassigned in at least one path will cause a latch to be inferred.

```
module synLatchReset
    (output reg Q,
     input      g, d, reset);

    always @(*)
      if (~reset)
          Q = 0;
      else if (g)
          Q = d;
endmodule
```

Example 2.16 Latch With Reset

Example 2.16 shows a latch with a reset input. Although we have specified output Q to be a register, that alone does not cause a latch to be inferred. To see how the latch inference arises, note that in the control flow of the always statement, not all of the possible input combinations of g and **reset** are specified. The specification says that if there is a change on either g, d or **reset**, the always loop is executed. If **reset** is zero, then Q is set to zero. If that is not the case, then if g is one, then Q is set to the d input. However, because there is no specification for what happens when **reset** is one and g is zero, a latch is needed to remember the previous value of Q. This is, in fact, the behavior of a level sensitive latch with reset. The latch behavior could also have been inferred using case or other statements.

```
module synALUwithLatchedOutput
    #(parameter     Width = 4)
    (output reg [Width-1:0] Q,
     input      [Width-1:0] a, b,
     input      g, addsub);

    always @(*) begin
      if (g) begin
          if (addsub)
              Q = a + b;
          else Q = a - b;
      end
    end
endmodule
```

Example 2.17 ALU With Latched Output

The latch synthesized does not need to be a simple gated latch; other functionality can be included as shown in Example 2.17. Here an ALU capable of adding and subtracting is synthesized with an output latch. The module's width is parameterized.

While gate g is TRUE, the output Q will follow the inputs, producing either the sum or difference on the output. Input **addsub** selects between the two functions. When g is not TRUE, the latch holds the last result.

2.4.2 Flip Flop Inferences

Flip flops are *edge-triggered* storage devices. Typically, their behavior is controlled by a positive or negative edge that occurs on a special input, called the clock. When the edge event occurs, the input d is remembered and gated to the output Q. They often have set and/or reset inputs that may change the flip flop state either synchronously or asynchronously with respect to the clock. At no time is the output Q a combinational function of the input d. These flip flops are not explicitly specified. Rather, they are inferred from the behavior. Since some of their behavior can be rather complex, there is essentially a template for how to specify it. Indeed some synthesis tools provide special compiler directives for specifying the flip flop type.

Example 2.18 shows a synthesizable model of a flip flop. The main characteristic of a flip flop description is that the event expression on the always statement specifies an edge. It is this edge event that infers a flip flop in the final design (as opposed to a level sensitive latch). As we will see, an always block with an edge-triggered event expression will cause flip flops to be inferred for all of the registers assigned to in procedural assignments in the always block. (Thus, an always block with an edge-triggered event expression cannot be used to define a fully combinational function.)

```
module synDFF
    (output reg q,
     input      clock, d);

    always @(negedge clock)
      q <= d;
endmodule
```

Example 2.18 A Synthesizable D Flip Flop

Typically flip flops include reset signals to initialize their state at system start-up. The means for specifying these signals is very stylized so that the synthesis tool can determine the behavior of the device to synthesize. Example 2.19 shows a D flip flop with asynchronous set and reset capabilities. In this example, the **reset** signal is asserted low, the **set** signal is asserted high, and the clock event occurs on the positive edge of **clock**.

Examples 2.18 and 2.19 both use non-blocking assignments in their specification. This specification allows for correct simulation if multiple instances of these modules are connected together.

Although the Example 2.19 appears straight-forward, the format is quite strict and semantic meaning is inferred from the order of the statements and the expressions within the statements. The form of the description must follow these rules:

```
module synDFFwithSetReset
      (output  reg  q,
       input        d, reset, set, clock);

      always @(posedge clock, negedge reset, posedge set) begin
         if (~reset)
               q <= 0;
         else if (set)
               q <= 1;
         else  q <= d;
      end
endmodule
```

Example 2.19 A Synthesizable D Flip Flop With Set and Reset

- The always statement must specify the edges for each signal. Even though asynchronous reset and set signals are not edge triggered they must be specified this way. (They are not edge triggered because **q** will be held at zero as long as **reset** is zero — not just when the negative edge occurs.)
- The first statement following the always must be an if.
- The tests for the set and reset conditions are done first in the always statement using else-if constructs. The expressions for set and reset cannot be indexed; they must be one-bit variables. The tests for their value must be simple and must be done in the order specified in the event expression.
- If a negative edge was specified as in **reset** above, then the test should be:
  `if (~reset) ...`

  or
  `if (reset == 1'b0) ...`
- If a positive edge was specified as in **set** above, then the test should be:
  `if (set) ...`

  or
  `if (set == 1'b1) ...`
- After all of the set and resets are specified, the final statement specifies the action that occurs on the clock edge. In the above example, **q** is loaded with input **d**. Thus, "clock" is not a reserved word. Rather, the synthesis tools infer the special clock input from assignment's position in the control path; it is the action that occurs when none of the set or reset actions occur.
- All procedural assignments in an always block must either be blocking or non-blocking assignments. They cannot be mixed within an always block. Non-blocking assignments ("<=") are the assignment operator of choice when specifying the edge-sensitive behavior of a circuit. The "<=" states that all the transfers in the whole system that are specified to occur on the edge in the sensitivity list should

occur concurrently. Although descriptions using the regular "=" will synthesize properly, they may not simulate properly. Since both simulation and synthesis are generally of importance, use "<=" for edge sensitive circuits.

- The sensitivity list of the always block includes only the edges for the clock, reset and preset conditions.

  These are the only inputs that can cause a state change. For instance, if we are describing a D flip flop, a change on D will not change the flip flop state. So the D input is not included in the sensitivity list.

- Any register assigned to in the sequential always block will be implemented using flip flops in the resulting synthesized circuit. Thus you cannot describe purely combinational logic in the same always block where you describe sequential logic. You can write a combinational expression, but the result of that expression will be evaluated at a clock edge and loaded into a register.

References: non-blocking versus blocking assignment 8.4.

2.4.3 Summary

Latches and flip flops are fundamental components of register-transfer level systems. Their complex behavior requires that a strict format be used in their specification. We have only covered the basics of their specification. Most synthesis tools provide compiler directives to aid in making sure the proper library element is selected to implement the specified behavior. Read the synthesis tool manual closely.

2.5 Inferring Tri-State Devices

Tri-state devices are combinational logic circuits that have three output values: one, zero, and high impedance (z). Having special, non-typical capabilities, these devices must be inferred from the description. Example 2.20 illustrates a tri-state inference.

```
module synTriState
    (output reg bus,
     input      in, driveEnable);

    always @(*)
        if (driveEnable)
            bus = in;
        else bus = 1'bz;
endmodule
```

Example 2.20 Inferring a Tri-State Device

The always statement in this module follows the form for describing a combinational logic function. The special situation here is that a condition (in this case, **driveEnable**) specifies a case where the output will be high impedance. Synthesis tools infer that this condition will be the tri-state enable in the final implementation.

2.6 Describing Finite State Machines

We have seen how to specify combinational logic and sequential elements to a synthesis tool. In this section we will combine these into the specification of a finite state machine. The standard form of a finite state machine is shown in Figure 2.1. The machine has inputs x_i, outputs z_i, and flip flops Q_i holding the current state. The outputs can either be a function solely of the current state, in which case this is a Moore machine. Or, they can be a function of the current state and input, in which case this is a Mealy machine. The input to the flip flops is the next state; this is a combinational function of the current state and inputs.

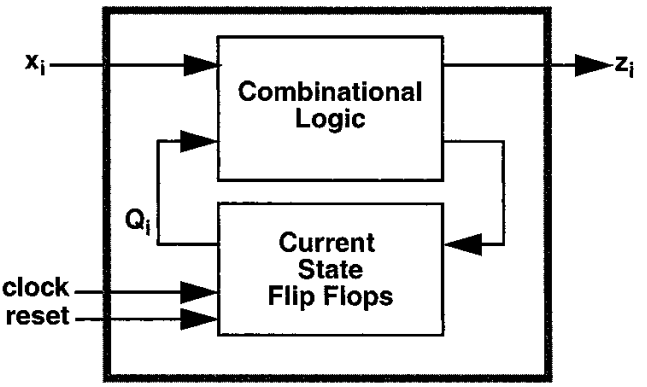


Figure 2.1 Standard Model of a Finite State Machine

The Verilog description of a finite state machine (FSM) follows this model closely. The outer box of Figure 2.1 will be the FSM module. The two inner boxes will be two separate always statements. One will describe the combinational logic functions of the next state and output. The other will describe the state register.

2.6.1 An Example of a Finite State Machine

An example of an FSM description will be presented using the *explicit* style of FSM description. In this style, a case statement is used to specify the actions in each of the machine's states and the transitions between states. Consider the state transition diagram shown in Figure 2.2. Six states and their state transitions are shown with one input and three output bits specified. Example 2.21 is the Verilog description of this FSM.

The first always statement is a description of the combinational output (**out**) and next state (**nextState**) functions. The input set for these functions contains the input **i** and the register **currentState**. Any change on either of these will cause the always statement to be re-evaluated. The single statement within the always is a case state-

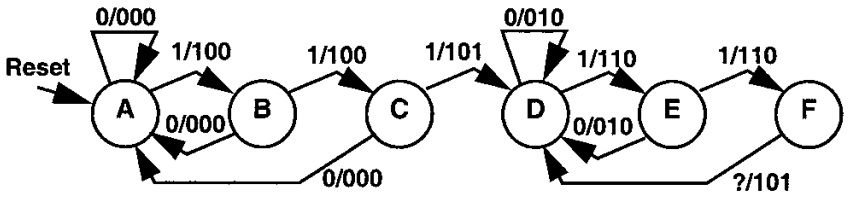


Figure 2.2 State Transition Diagram for Example 2.21

ment indicating the actions to be performed in each state. The controlling expression for the case is the state variable (**currentState**). Thus, depending on what state the machine is in, only the specified actions occur. Note that in each case item, the two combinational functions being computed (**out** and **nextState**) are assigned to. In addition, a default case item is listed representing the remaining unassigned states. The default sends the machine to state A which is equivalent to a reset. By arbitrary choice, **out** is set to don't care in the unassigned states.

This always statement will result in combinational logic because: the sensitivity list contains all of the input set, there are no edge specifiers in the sensitivity list, and for every control path, both of the combinational outputs have been assigned to. This includes every possible case item. Thus, there will be no inferred latches. Note that a default case item was used here instead of specifying that this is a full case. This allows us to specify the reset state as the next state in case there is an error in operation — for instance, the logic circuit somehow gets into an undefined state. Although we specified that the output in this situation is a don't care, we could have made a specification here too.

The second always statement infers the state register with its reset condition. In this case, **reset** is asserted low and will cause the machine to go into state **A**. If **reset** is not asserted, then the normal action of the always will be to load **currentState** with the value of **nextState**, changing the state of the FSM on the positive edge of **clock**.

Notice that **currentState** is assigned to in every control path of the always — so why is a flip flop inferred? The reason is that the edge specifications in the event expression cause any register assigned to in the block to be implemented using flip flops. You cannot specify combinational logic in an always block with edge triggers in the sensitivity list. This is why we need two always blocks to specify an FSM: one for the state register, and the other for the combinational logic.

The localparam statement specifies the state assignment for the system. Since these are treated as constants, they cannot be directly overridden by instantiation.

Together, these two always statements work together to implement the functionality of a finite state machine. The output of the second always is the current state of the

```
module fsm
    (input          i, clock, reset,
     output reg [2:0] out);

    reg     [2:0]   currentState, nextState;

    localparam [2:0] A = 3'b000,// The state labels and their assignments
                     B = 3'b001,
                     C = 3'b010,
                     D = 3'b011,
                     E = 3'b100,
                     F = 3'b101;

    always @(*)                 // The combinational logic
      case (currentState)
          A: begin
               nextState = (i == 0) ? A : B;
               out = (i == 0) ? 3'b000 : 3'b100;
             end
          B: begin
               nextState = (i == 0) ? A : C;
               out = (i == 0) ? 3'b000 : 3'b100;
             end
          C: begin
               nextState = (i == 0) ? A : D;
               out = (i == 0) ? 3'b000 : 3'b101;
             end
          D: begin
               nextState = (i == 0) ? D : E;
               out = (i == 0) ? 3'b010 : 3'b110;
             end
          E: begin
               nextState = (i == 0) ? D : F;
               out = (i == 0) ? 3'b010 : 3'b110;
             end
          F: begin
               nextState = D;
               out = (i == 0) ? 3'b000 : 3'b101;
             end
          default: begin // oops, undefined states. Go to state A
               nextState = A;
               out = (i == 0) ? 3'bxxx : 3'bxxx;
             end
      endcase
```

```
    always @(posedge clock or negedge reset) // The state register
            if (~reset)
                    currentState <= A;// the reset state
            else
                    currentState <= nextState;
endmodule
```

Example 2.21 A Simple Finite State Machine

FSM and it is in the input set of the first always statement. The first always statement is a description of combinational logic that produces the output and the next state functions.

References: parameters 5.2; non-blocking assignment 8.4; implicit style 2.6.2.

2.6.2 An Alternate Approach to FSM Specification

The above explicit approach for specifying FSMs is quite general, allowing for arbitrary state machines to be specified. If an FSM is a single loop without any conditional next states, an *implicit* style of specification may be used.

The basic form of an implicit FSM specification is illustrated in Example 2.22. The single always statement lists several clock events, all based on the same edge (positive or negative). Since the always specifies a sequential loop, each state is executed in order and the loop executes continuously. Thus, there is no next state function to be specified.

In this particular example, a flow of data is described. Each state computes an output (**temp** and **dataOut**) that is used in later states. The output of the final state (**dataOut**) is the output of the FSM. Thus, a new result is produced every third clock period in **dataOut**.

```
module synImplicit
    (input       [7:0]  dataIn, c1, c2,
     input              clock,
     output reg  [7:0]  dataOut);

    reg      [7:0]      temp;

    always begin
       @ (posedge clock)
            temp = dataIn + c1;
       @ (posedge clock)
            temp = temp & c2;
       @ (posedge clock)
            dataOut = temp - c1;
    end
endmodule
```

Example 2.22 An Implicit FSM

Another example of a flow of data is a pipeline, illustrated in Example 2.23 using a slightly different calculation. Here a result is produced every clock period in **dataOut**. In this case, three FSMs are specified; one for each stage of the pipe. At every clock event, each stage computes a new output (**stageOne**, **stageTwo**, and **dataOut**). Since these variables are used on the left-hand side of a procedural statement in an always block with an edge specifier, there are implemented with registers. The nonblocking assignment (<=) must be used here so that the simulation results will be correct. Figure 2.3 shows a simplified form of the implementation of module **synPipe**.

```
module synPipe
    (input       [7:0]  dataIn, c1, c2,
    input               clock,
    output  reg  [7:0]  dataOut);

    reg      [7:0]   stageOne;
    reg      [7:0]   stageTwo;

    always  @ (posedge clock)
      stageOne <= dataIn + c1;

    always  @ (posedge clock)
      stageTwo <= stageOne & c2;

    always  @ (posedge clock)
      dataOut <= stageTwo + stageOne;
endmodule
```

Example 2.23 A Pipeline

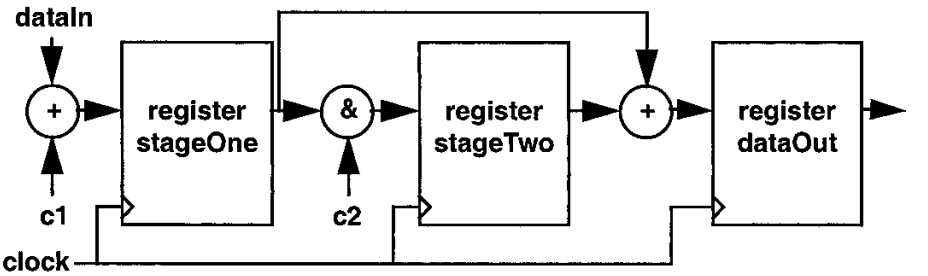


Figure 2.3 The Data Path of Example 2.23

References: explicit style 2.6.1

2.7 Finite State Machine and Datapath

We've used the language to specify combinational logic and finite state machines. Now we'll move up to specifying register transfer level systems. We'll use a method of specification known as finite state machine and datapath, or FSM-D. Our system will be made up of two parts: a datapath that can do computations and store results in registers, and a finite state machine that will control the datapath.

2.7.1 A Simple Computation

We begin with a simple computation and show how to specify the logic hardware using Verilog. The computation is shown below in a C-like syntax:

```
...
for (x = 0, i = 0; i <= 10; i = i + 1)
        x = x + y;
if (x < 0)
        y = 0;
else    x = 0;
...
```

The computation starts off by clearing **x** and **i** to 0. Then, while **i** is less than or equal to 10, **x** is assigned the sum of **x** and **y**, and **i** is incremented. When the loop is exited, if **x** is less than zero, **y** is assigned the value 0. Otherwise, **x** is assigned the value 0. Although simple, this example will illustrate building larger systems.

We'll assume that these are to be 8-bit computations and thus all registers in the system will be 8-bit.

2.7.2 A Datapath For Our System

There are many ways to implement this computation in hardware and we will focus on only one of them. A datapath for this system must have registers for **x**, **i**, and **y**. It needs to be able to increment **i**, add **x** and **y**, and clear **i**, **x**, and **y**. It also needs to be able to compare **i** with 10 and **x** with 0. Figure 2.4 illustrates a datapath that could execute these register transfers.

The name in each box in the figure suggests its functionality. Names with overbars are control signals that are asserted low. Looking at the block labeled **register i**, we see that its output (coming from the bottom) is connected back to the input of an adder whose other input is connected to 1. The output of that adder (coming from the bottom) is connected to the input of **register i**. Given that the register stores a value and the adder is a combinational circuit, the input to **register i** will always be one greater than the current value of **register i**. The register also has two control inputs: **iLoad** and **iClear**. When one of these inputs is asserted, the specified function will occur at

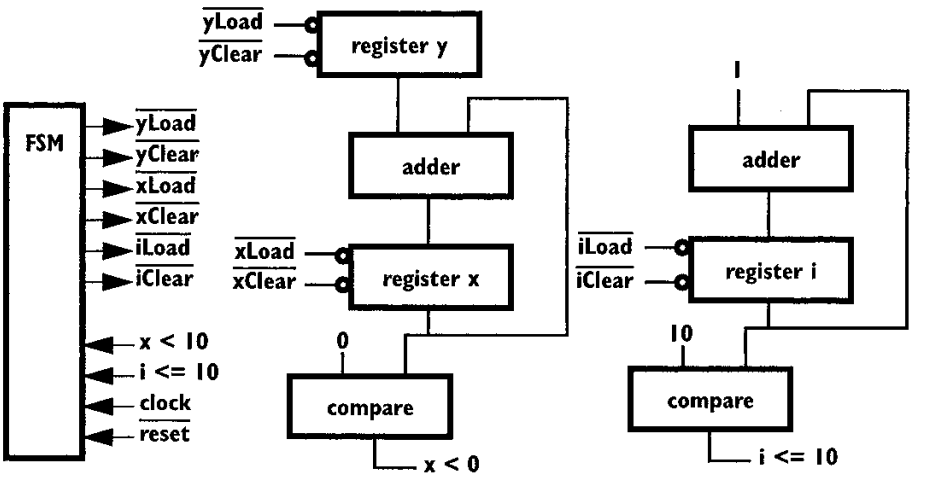


Figure 2.4 Finite State Machine and Datapath

the next clock edge. If we assert **iLoad**, then after the next clock edge **register i** will load and store its input, incrementing **i**. Alternately, **iClear** will load a zero into **register i**. The compare modules are also combinational and produce the Boolean result indicated.

The register transfers shown in our computation are x = 0, i = 0, y = 0, i = i + 1, and x = x + y. From the above description of how the datapath works, we can see that all of the register transfers in our computation can be executed on this datapath. Further, all of the conditional values needed for branching in the FSM are generated in the datapath.

The FSM shown on the left sequences through a series of states to cause the computation to occur. The FSM's outputs are **yLoad**, **yClear**, **xLoad**, **xClear**, **iLoad**, and **iClear**. Its inputs are x<0 and i<=10. A master **clock** drives the state registers in the FSM as well as the datapath registers. A **reset** signal is also connected.

2.7.3 Details of the Functional Datapath Modules

The datapath is made up of three basic modules: registers, adders, and comparators. The register module definition is shown in Example 2.24. Looking first at the always block, we see that it is very similar to those we've seen in sequential circuit descriptions so far. The register is positive edge triggered but does not have an asynchronous reset. To go along with the register modules defined for our datapath, it has two control points: clear and load. These control points, when asserted, cause the register to perform the specified function. If input **clear** is asserted, it will load 0 at the **clock** edge. If **load** is asserted, it will load input **in** into register **out** at the **clock** edge. If both are asserted, then the register will perform the clear function.

```
module register
    #(parameter                 Width = 8)
    (output reg [Width-1:0]     out,
     input      [Width-1:0]     in,
     input                      clear, load, clock);

    always @(posedge clock)
       if (~clear)
            out <= 0;
       else if (~load)
            out <= in;
endmodule
```

Example 2.24 Register Module

This example introduces a new statement, the parameter statement. The parameter defines a name to have a constant value; in this case **Width** has the value 8. This name is known within the module and can be used in any of the statements. Here we see it being used to define the default value for the left-most bit number in the vector definitions of the output and register **out** and the input **in**. Given that **Width** is defined to be 8, the left-most bit is numbered 7 (i.e., 8-1) and **out** and **in** both have a bitwidth of eight (i.e., bits 7 through 0). What is interesting about a parameter is that the default value can be overridden at instantiation time; however it cannot be changed during the simulation. Thus, this module definition can be used to instantiate registers of different bitwidth. We will see how shortly.

The adder module is shown in Example 2.25. It is parameterized to have a default bitwidth of eight. The assign statement in this example shows a means of generating our "adder" function. The output **sum** is assigned the arithmetic sum of inputs **a** and **b** using the "+" operator. The assign statement is discussed further in Chapter 6.

```
module adder
    #(parameter Width = 8)
    (input  [Width-1:0] a, b,
     output [Width-1:0] sum);

    assign sum = a + b;
endmodule
```

Example 2.25 The Adder Module

The **compareLT** and **compareLEQ** modules are shown in Example 2.26, again using the continuous assign statement. In the **compareLT** module, **a** is compared to **b**. If **a** is less than **b**, then **out** is set to TRUE. Otherwise it is set to FALSE. The **compareLEQ** module for comparing **i** with 10 in our computation is similar to this module except with the "<=" operator instead of the "<" operator. The width of these modules are also parameterized. Don't be confused by the second assign statement, namely:

```
assign out = a <= b;
```

```
module compareLT // compares a < b
    #(parameter Width = 8)
    (input      [Width-1:0] a, b,
     output                 out);

    assign out = a < b;
endmodule

module compareLEQ // compares a <= b
    #(parameter Width = 8)
    (input      [Width-1:0] a, b,
     output                 out);

    assign out = a <= b;
endmodule
```

Example 2.26 The CompareLT and CompareLEQ Modules

This does not assign **b** to **a** with a non-blocking assignment, and then assign **a** to **out** with a blocking assignment. Only one assignment is allowed in a statement. Thus by their position in the statement, we know that the first is an assignment and the second is a less than or equal comparison.

The **adder**, **compareLEQ**, and **compareLT** modules could have written using the combinational version of the always block. As used in these examples, the two forms are equivalent. Typically, the continuous assign approach is used when a combinational function can be described in a simple statement. More complex combinational functions, including ones with don't care specifications, are typically easier to describe with a combinational always statement.

References: continuous assign 6.3

2.7.4 Wiring the Datapath Together

Now we build a module to instantiate all of the necessary FSM and datapath modules and wire them together. This module, shown in Example 2.27, begins by declaring the 8-bit wires needed to connect the datapath modules together, followed by the 1-bit wires to connect the control lines to the FSM. Following the wire definitions, the module instantiations specify the interconnection shown in Figure 2.4.

Note that this module also defines a **Width** parameter, uses it in the wire definitions, and also in the module instantiations. Consider the module instantiation for the register **I** from Example 2.27.

```
module sillyComputation
    #(parameter Width = 8)
    (input              ck, reset,
     input  [Width-1:0] yIn,
     output [Width-1:0] y, x);
    wire   [Width-1:0] i, addiOut, addxOut;
    wire               yLoad, yClear, xLoad, xClear, iLoad, iClear;

    register    #(Width)    I     (i, addiOut, iClear, iLoad, ck),
                            Y     (y, yIn, yClear, yLoad, ck),
                            X     (x, addxOut, xClear, xLoad, ck);

    adder       #(Width)    addI  (addiOut, 'b1, i),
                            addX  (addxOut, y, x);

    compareLT   #(Width)      cmpX (x, 'b0, xLT0);
    compareLEQ  #(Width)      cmpI  (i, 'd10, iLEQ10);

    fsm             ctl
    (xLT0, iLEQ10, yLoad, yClear, xLoad, xClear, iLoad, iClear, ck, reset);
endmodule
```

Example 2.27 Combining the FSM and Datapath

```
register    #(Width)    I    (i, addiOut, iClear, iLoad, ck),
```

What is new here is the second item on the line, "#(Width)". This value is substituted in the module instantiation for its parameter. Thus, by changing the parameter **Width** in module **sillyComputation** to, say 23, then all of the module instantiations for the datapath would be 23 bits wide. Parameterizing modules allows us to reuse a generic module definition in more places, making a description easier to write. If #(Width) had not been specified in the module instantiation statement, then the default value of 8, specified in module **register**, would be used. The example also illustrates the use of unsized constants. The constant 1 specification (given as **'b1** in the port list of **adder** instance **addI**) specifies that regardless of the parameterized **Width** of the module, the value 1 will be input to the adder. That is the least significant bit will be 1 with as many 0s padded to the left as needed to fill out the parameterized width. This is also grue of unsized constants **'b0** and **'d10** in the compare module instantiations.

2.7.5 Specifying the FSM

Now that the datapath has been specified, a finite state machine is needed to evoke the register transfers in the order and under the conditions specified by the original computation. We first present a state transition diagram for this system and then describe the Verilog **fsm** module to implement it.

The state transition diagram is shown in Figure 2.5 along with the specification for the computation. The states marked "..." represent the computation before and after the portion of interest to us. Each state "bubble" indicates the FSM outputs that are to be asserted during that state; all others will be unasserted. The arrows indicate the next state; a conditional expression beside an arrow indicates the condition in which that state transition is taken. The diagram is shown as a Moore machine, where the outputs are a function only of the current state. Finally, the states are labeled A through F for discussion purposes.

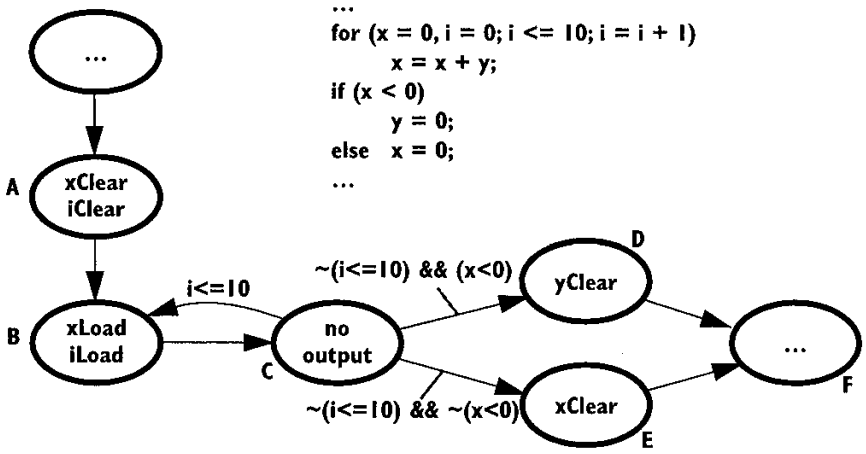


Figure 2.5 State Transition Diagram

Following through the computation and the state transition diagram, we see that the first action is to clear both the x and i registers in state A. This means that while the machine is in state A, **xClear** and **iClear** are asserted (low). Note though that the registers i and x will not become zero until after the positive clock edge and we're in the next state (B). State B then asserts the load signals for x and i. The datapath in Figure 2.4 shows us what values are actually being loaded: x + y and i + 1 respectively. Thus, state B executes both the loop body and the loop update. From state B the system goes to state C where there is no FSM output asserted. However, from state C there are three possible next states depending on whether we are staying in the loop (going to state B), exiting the loop and going to the then part of the conditional (state D), or exiting the loop and going to the else part of the conditional (state E). The next state after D or E is state F, the rest of the computation.

It is useful to understand why state C is needed in this implementation of the system. After all, couldn't the conditional transitions from state C have come from state B where x and i are loaded? The answer is no. The timing diagram in Figure 2.6 illustrates the transitions between states A, B, and C. During the time when the system is in state B, the asserted outputs of the finite state machine are **xLoad** and **iLoad**, meaning that the x and i registers are enabled to load from their inputs. But they will not be loaded until the next clock edge, the same clock edge that will transit the finite state machine into state C. Thus the values of i, on which the end of loop condition is based, and x, on which the if-then-else condition is based, are not available for comparison until the system is in state C. In the timing diagram, we see that since i is less than or equal to 10, the next state after C is B.

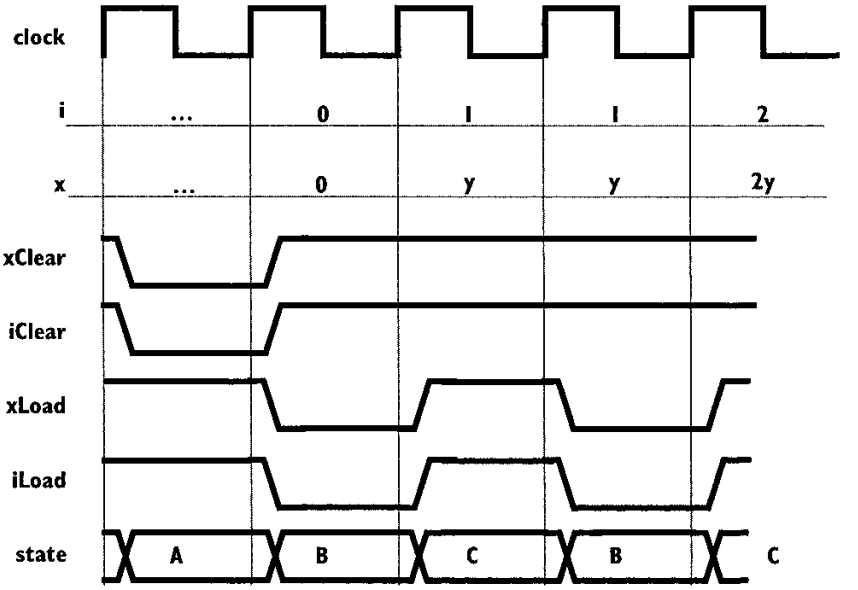


Figure 2.6 Timing Diagram For States A, B, and C.

It is interesting to note that in this implementation of the system, the exit condition of the for loop is not checked before entering the loop. However, given that we just cleared i before entering the loop, it is not necessary to check that is less than or equal to 10. Further, with a different datapath, state C might not be necessary. For instance, the comparisons with i and x could be based on the input value to these registers, thus comparing with the future value. Or the constants with which the comparisons are made could be changed. Of course, these are all at the discretion of the designer.

Now consider the Verilog model of the finite state machine for this system shown in Example 2.28. The machine's inputs are the two conditions, x < 0 and i <= 10. Internal to the **fsm** module, they are called **LT** and **LEQ** respectively. Module **fsm** also has a **reset** input and a clock (**ck**) input. The module outputs are the control points on the registers (**yLoad, yClear, xLoad, xClear, iLoad, iClear**). Like our previous fsm examples, there are two always blocks, one for the sequential state change and the other to implement the next state and output combinational logic. Registers are declared for all of the combinational outputs.

Our state machine will only implement the states shown in the state transition diagram, even though there would be many more states in the rest of the computation. Thus, the width of the state register (**cState**) was chosen to be three bits. Further, the reset state is shown to be state 0 although in the full system it would be some other state. A very simple state assignment has been chosen, with state **A** encoded by 0, **B** encoded by 1, and so on.

The first always block is very similar to our previous state machine examples. If **reset** is asserted, then the reset state is entered. Otherwise, the combinational value **nState** is loaded into **cState** at the positive **clock** edge.

The second always block implements the next state and output combinational logic. The inputs to this combinational logic are the current state (**cState**) and **fsm** inputs (**LT** and **LEQ**). The body of the always block is organized around the value of **cState**. A case statement, essentially a multiway branch, is used to specify what is to happen given each possible value of **cState**. The value of the expression in parentheses, in this case **cState**, is compared to the values listed on each line. The line with the matching value is executed.

If the current state changes to state **A**, then the value of **cState** is 0 given our encoding. Thus, when this change occurs, the always block will execute, and the statement on the right side of the 3'b000: will execute. This statement specifies that all of the outputs are unasserted (1) except **iClear** and **xClear**, and the next state is 3'b001 (which is state **B**). If the current state is **B**, then the second case item (3'b001) is executed, asserting **iLoad** and **xLoad**, and unasserting all of the other outputs. The next state from state **B** is **C**, encoded as 3'b010. State **C** shows a more complex next state calculation; the three if statements specify the possible next states from state **C** and the conditions when each would be selected.

The last case item specifies the **default** situation. This is the statement that is executed if none of the other items match the value of **cState**. For simulation purposes, you might want to have a $display statement to print out an error warning that you've reached an illegal state. The $display prints a message on the screen during simulation, acting much like a print statement in a programming language. This one displays the message "Oops, unknown state: %b" with the binary representation of **cState** substituted for %b.

To make this always block a combinational synthesizable function, the default is required. Consider what happens if we didn't have the default statement and the value of **cState** was something other than one of the five values specified. In this situation, the case statement would execute, but none of the specified actions would be executed. And thus, the outputs would not be assigned to. This breaks the combinational synthesis rule that states that every possible path through the always block must assign to every combinational output. Thus, although it is optional to have the default case for debugging a description through simulation, the default is required for this always block to synthesize to a combinational circuit. Of course a default is not required for synthesis if all known value cases have been specified or **cState** was assigned a value before the case statement.

Consider now how the whole FSM-Datapath system works together. Assume that the current state is state **C** and the values of **i** and **x** are 1 and **y** respectively, as shown in the timing diagram of Figure 2.6. Assume further that the clock edge that caused the system to enter state **C** has just happened and **cState** has been loaded with value 3'b010 (the encoding for state **C**). Not only has **cState** changed, but registers **x** and **i** were also loaded as a result of coming from state **B**.

In our description, several always blocks are were waiting for changes to **cState**, **x**, and **i**. These include the **fsm**'s combinational always block, the adders, and the compare modules. Because of the change to **cState**, **x**, and **i**, these always blocks are now enabled to execute. The simulator will execute them, in arbitrary order. Indeed, the simulator may execute some of them several times. (Consider the situation where the fsm's combinational always block executes first. Then after the compare modules execute, it will have to execute again.) Eventually, new values will be generated for the outputs of the comparators. Changes in **LT** and **LEQ** in the **fsm** module will cause its combinational always block to execute, generating a value for **nState**. At the next positive clock edge, this value will be loaded into **cState** and another state will be entered.

References: case 3.4; number representation B.3

2.8 Summary on Logic Synthesis

We have seen that descriptions used for logic synthesis are very stylized and that some of the constructs are overloaded with semantic meaning for synthesis. In addition, there are several constructs that are not allowed in a synthesizable description. Because these can vary by vendor and version of the tool, we chose not to include a table of such constructs. Consult the user manual for the synthesis tool you are using.

Table 2.1 summarizes some of the basic rules of using procedural statements to describe combinational logic and how to infer sequential elements in a description.

```
module fsm
    (input          LT, LEQ, ck, reset,
     output reg     yLoad, yClear, xLoad, xClear, iLoad, iClear);

    reg     [2:0]   cState, nState;

always @(posedge ck, negedge reset)
    if (~reset)
            cState <= 0;
    else    cState <= nState;

always @(cState, LT, LEQ)
    case (cState)
       3'b000 : begin      // state A
                   yLoad = 1; yClear = 1; xLoad = 1; xClear = 0;
                   iLoad = 1; iClear = 0; nState = 3'b001;
                end
       3'b001 : begin      // state B
                   yLoad = 1; yClear = 1; xLoad = 0; xClear = 1;
                   iLoad = 0; iClear = 1; nState = 3'b010;
                end
       3'b010 : begin      // state C
                   yLoad = 1; yClear = 1; xLoad = 1; xClear = 1;
                   iLoad = 1; iClear = 1;
                   if (LEQ) nState = 3'b001;
                   if (~LEQ & LT) nState = 3'b011;
                   if (~LEQ & ~LT) nState = 3'b100;
                end
       3'b011 : begin      // state D
                   yLoad = 1; yClear = 0; xLoad = 1; xClear = 1;
                   iLoad = 1; iClear = 1; nState = 3'b101;
                end
       3'b100 : begin      // state E
                   yLoad = 1; yClear = 1; xLoad = 1; xClear = 0;
                   iLoad = 1; iClear = 1; nState = 3'b101;
                end
       default : begin // required to satisfy combinational synthesis rules
                   yLoad = 1; yClear = 1; xLoad = 1; xClear = 1;
                   iLoad = 1; iClear = 1; nState = 3'b000;
                   $display ("Oops, unknown state: %b", cState);
                end
    endcase
endmodule
```

Example 2.28 FSM For the Datapath

Table 2.1 Basic Rules for Using Procedural Statements in Logic Synthesis

Type of Logic	Output Assigned To	Edge Specifiers in Sensitivity List
Combinational	An output must be assigned to in all control paths.	Not allowed. The whole input set must be in the sensitivity list. The construct @(*) assures this.
Inferred latch	There must exist at least one control path where an output is not assigned to. From this "omission," the tool infers a latch.	Not allowed.
Inferred flip flop	No affect	Required — from the presence of an edge specifier, the tool infers a flip flop. All registers in the always block are clocked by the specified edge.

2.9 Exercises

2.1 In section 2.2 on page 37, we state that a synthesis tool is capable, and possibly constrained, to implement the functionality using different gate primitives. Explain why it might be "constrained" to produce an alternate implementation.

2.2 Alter the description of Example 2.7 so that there is no longer an inferred latch. When **a** is not one, **b** and **c** should be OR-d together to produce the output.

2.3 Alter the description of Example 2.16. Use a case statement to infer the latch.

2.4 Why can't while and forever loops be used to specify combinational hardware?

2.5 Rewrite Example 2.21 as a Moore machine. An extra state will have to be added.

2.6 Rewrite Example 2.21 using a one-hot state encoding. Change the description to be fully parameterized so that any state encoding may be used.

2.7 Write a description for the FSM shown in Figure 2.7 with inputs **Ain**, **Bin**, **Cin**, **clock**, and **reset**, and output **Y**.

A. A single always block

B. Two always blocks; one for the combinational logic and the other for the sequential.

C. Oops, this circuit is too slow. We can't have three gate delays between the flip flop outputs and inputs; rather only two. Change part B so that Y is a combinational output. i.e. Move the gate generating d2 to the other side of the flip flops.

D. Simulate all of the above to show that they are all functionally equivalent.

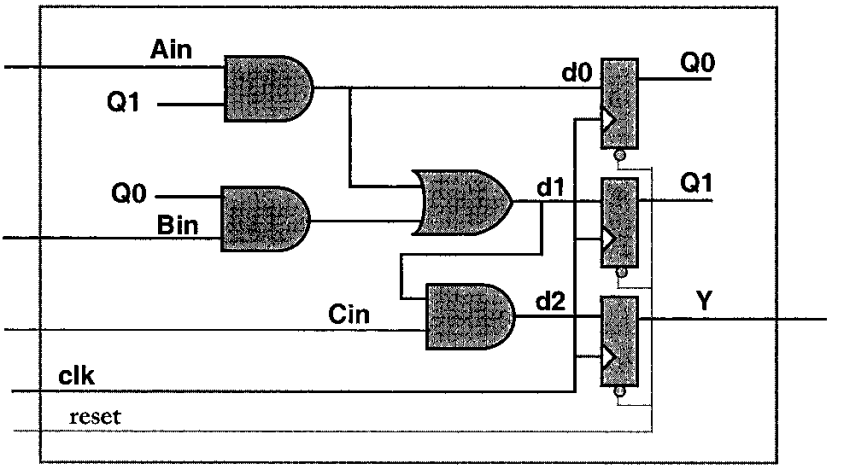


Figure 2.7 Control Over Synthesis

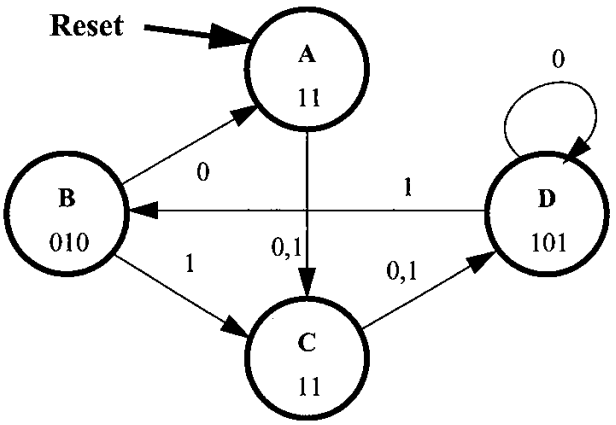
2.8 Design, implement, and simulate a 4-bit two's complement adder/subtractor circuit to compute (A+B), (A-B), sat(A+B), or sat(A-B) when a two-bit **add_sel** bit is 00, 01, 10, or 11, respectively. sat(x) is saturating addition. Saturating arithmetic is analogous to other kinds of saturation — numbers can only get so big (or so small) and no bigger (or smaller), but they don't wrap, unlike in modulo arithmetic.

For example, for a three-bit saturating adder, 010 + 001 = 011 (2+1=3, OK, fine), but 011 + 001 = 011 (3+1=3, huh?), i.e, instead of "wrapping" to obtain a negative result, let's just represent the result as close as we can, given our limited number of bits.

Similarly for negative results, 111+101 = 100 (-1+(-3) = -4), but 100 + 111 = 100 (-4 + (-1) = -4, i.e, the smallest, or most negative representation in 3-bit, two's complement).

Assume complemented inputs for **B** are *not* available. Write your description entirely in a synthesizable procedural Verilog style. Simulate and show *enough* test cases to verify your design (that does not mean all possible input cases this time!).

2.9 Consider the state transition diagram shown to the right. The output of the FSM is three bits, but states A and C only use two bits for the output. For *each* of the following encoding styles, (*binary encoding*, *output encoding*, *one-hot encoding*:):



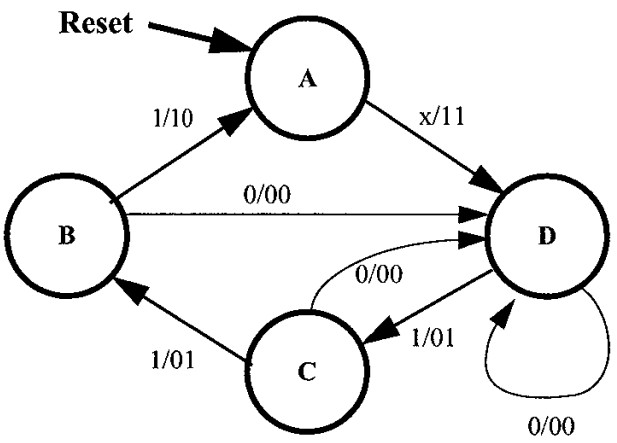
A. Show the state assignment

B. Derive boolean equations for the next state logic and output logic in minimized SOP. Show all your work and reasoning for full credit.

C. Design (draw) the circuit using positive edge-triggered D flip-flops with negative-logic preset and reset signals. Show the reset logic of the FSM.

D. Write a synthesizable Verilog simulation of your design. Write a test module to test your module with input sequence (read left to right) 11010001100. Use decimal format for the FSM outputs. Hand in the source code and output.

2.10 For the Mealy finite state machine shown to the right,



A. Write a procedural Verilog implementation for your mealy machine.

B. Simulate and test your circuit for an input sequence (read left to right) of 100101101111. Display the output sequence of your machine in decimal (i.e., 0,1,2,3) and relate it back to your input sequence. Your simulation

results must show that your machine behaves like a Mealy machine. In a Mealy, if you change inputs between block events, the output should follow, no matter how many times you change the inputs. Your simulation results must show this!

2.11 Design a Mealy finite state machine with one input, X, and one output, Y. With this state machine, you get an output of Y=1 every time the input sequence has exactly two or exactly four 1's in a row. Y=0 otherwise. Make sure your machine does this:

```
Input: 0110111011110111110
Output:0001000000001000000
```

Notice how you can't tell if you get *exactly* two or *exactly* four 1's until you see the next input. It sort of makes the timing look like a Moore machine, but it isn't. It's a Mealy. And you must design the machine as a Mealy! Write a Verilog implementation for your mealy machine. Simulate and test your circuit using the input sequence given above.