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;
...
```

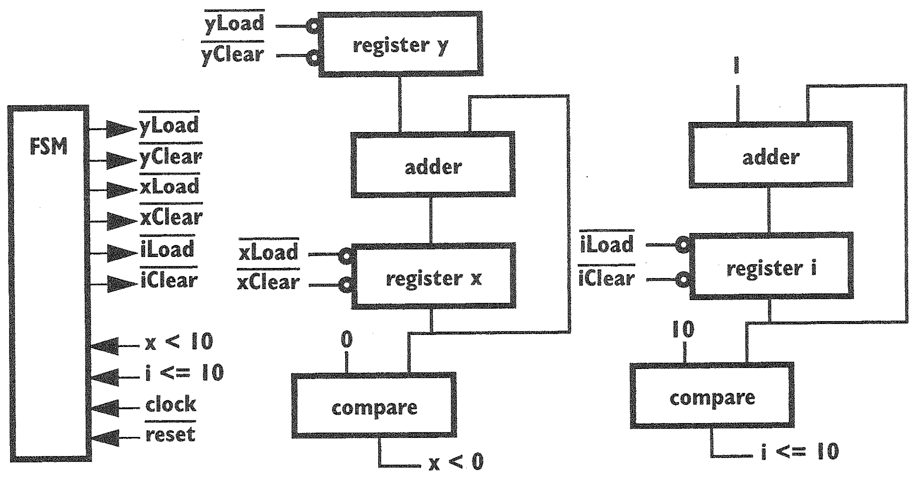
The computation starts off by clearing x and i to 0. Then, while i is less than or equal to 10, x is assigned the sum of x and y, and i is incremented. When the loop is exited, if x is less than zero, y is assigned the value 0. Otherwise, x is assigned the value 0. Although simple, this example will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



$\overline{\text{yLoad}}$
$\overline{\text{yClear}}$
register y
FSM
$\overline{\text{yLoad}}$
$\overline{\text{yClear}}$
$\overline{\text{xLoad}}$
$\overline{\text{xClear}}$
$\overline{\text{iLoad}}$
$\overline{\text{iClear}}$
x < 10
i <= 10
clock
reset
adder
$\overline{\text{xLoad}}$
$\overline{\text{xClear}}$
register x
0
compare
x < 0
1
adder
$\overline{\text{iLoad}}$
$\overline{\text{iClear}}$
register i
10
compare
i <= 10

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

```
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

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.

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.

```
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 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 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

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;
```

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

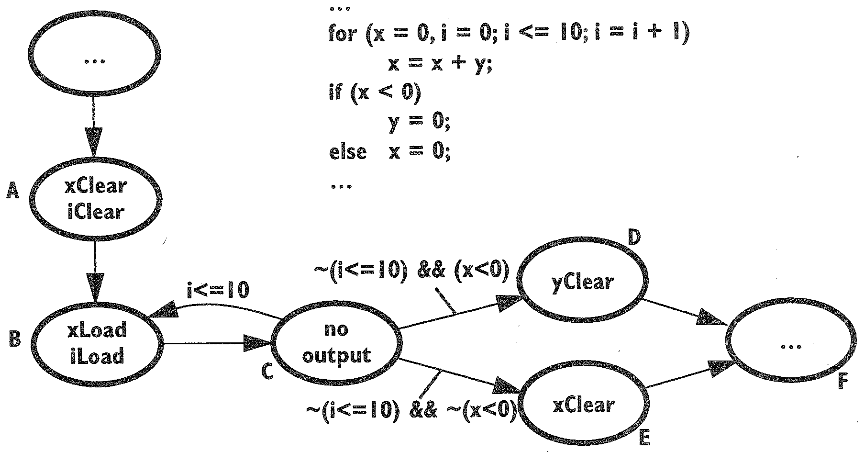
```
register      #(Width)     I     (i, addiOut, iClear, iLoad, ck),
```

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 true 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.



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

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.

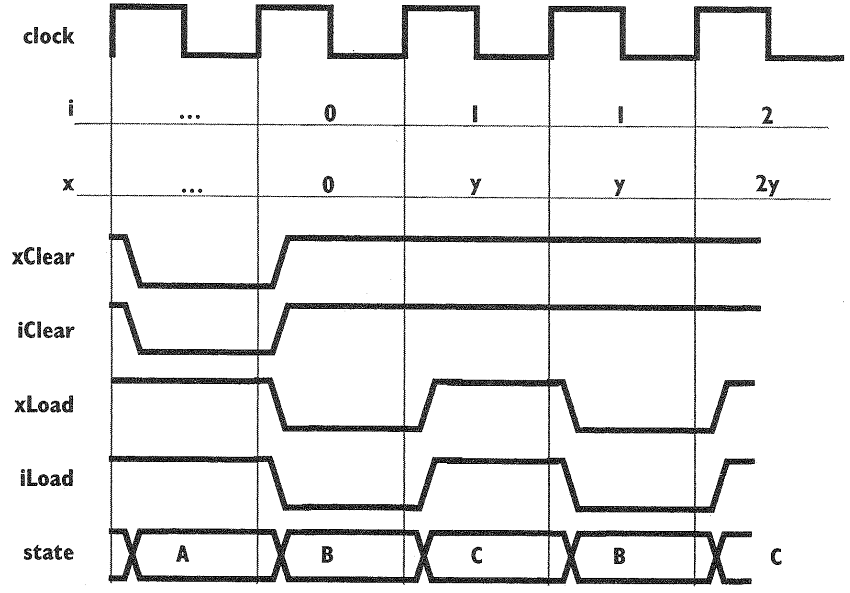


Figure 2.6 Timing Diagram For States A, B, and C.

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

```
        if ((m[4] - m[5]) < 0)
                m[4] = 17;
        else    m[4] = -10;
```

4.8 Add a third stage to the pipeline of Example 4.10. The second stage should only fetch operands; values read from memory should be put in a memory data register (mdr). The third stage will execute the instructions, loading the resulting values (mdr) in **acc**, **pctemp**, or **m**. Assume that the memory has multiple read and write ports. Handle any interstage conflicts that may arise.

5 Module Hierarchy

A *structural model* of a digital system uses Verilog *module* definitions to describe arbitrarily complex elements composed of other modules and gate primitives. As we have seen in earlier examples, a structural module may contain a combination of behavioral modeling statements (an always statement), continuous assignment statements (an assign statement), or module instantiations referencing other modules or gate level primitives. By using module definitions to describe complex modules, the designer can better manage the complexity of a design. In this chapter we explore module hierarchy and how it is specified as we cover instantiation, parameterized modules, and iterative generation.

5.1 Module Instantiation and Port Specifications

A port of a module can be viewed as providing a link or connection between two items, one internal to the module instance and one external to it. We have seen numerous examples of the specification of module ports.

An input port specifies the internal name for a vector or scalar that is driven by an external entity. An output port specifies the internal name for a vector or scalar which is driven by an internal entity and is available external to the module. An inout port specifies the internal name for a vector or scalar that can be driven either by an internal or external entity.

It is useful to recap some of the do's and don't's in their specification. First, an input or inout port cannot be declared to be of type register. Either of these port types may be read into a register using a procedural assignment statement, used on the right-hand side of a continuous assignment, or used as input to instantiated modules or gates. An inout port may only be driven through a gate with high impedance capabilities such as a bufif0 gate.

Secondly, each port connection is a continuous assignment of source to sink where one connected item is the signal source and the other is a signal sink. The output ports of a module are implicitly connected to signal source entities such as nets, registers, gate outputs, instantiated module outputs, and the left-hand side of continuous assignments internal to the module. Input ports are connected to gate inputs, instantiated module inputs, and the right-hand side of continuous and procedural assignments. Inout ports of a module are connected internally to gate outputs or inputs. Externally, only nets may be connected to a module's outputs.

```
module binaryToESeg
    (input   A, B, C, D,
     output  eSeg);

    nand #1
      g1 (p1, C, ~D),
      g2 (p2, A, B),
      g3 (p3, ~B, ~D),
      g4 (p4, A, C),
      g5 (eSeg, p1, p2, p3, p4);
endmodule
```

Finally, a module's ports are normally connected at the instantiation site in the order in which they are defined. However, we may connect to a module's ports by naming the port and giving its connection. Given the definition of **binaryToESeg**, reprinted here from Example 1.3, we can instantiate it into another module and connect its ports by name as shown below. Example 1.11 instantiated this module using the statement:

```
        binaryToESeg   disp   m1 (eSeg, w3, w2, w1, w0);
```

where **eSeg**, **w3**, **w2**, **w1**, and **w0** were all declared as wires. (The module instance name **m1** has been added here to help the discussion.) Alternately, the ports could have been specified by listing their connections as shown below:

```
        binaryToESeg   disp   m1 (.eSeg(eSeg), .A(w3), .B(w2), .C(w1), .D(w0));
```

In this statement, we have specified that port **eSeg** of instance **m1** of module **binaryToESeg** will be connected to wire **eSeg**, port **A** to wire **w3**, port **B** to wire **w2**, port **C** to wire **w1**, and port **D** to wire **w0**. The period (".") introduces the port name as defined in the module being instantiated. Given that both names are specified together, the connections may be listed in any order. If a port is to be left unconnected, no value is specified in the parentheses — thus **.D()** would indicate that no connection is to be made to port **D** of instance **m1** of module **binaryToESeg**.

At this point we can formally specify the syntax needed to instantiate modules and connect their ports. Note that the following syntax specification includes both means of listing the module connections: ordered-port and named-port specifications.

```
module instantiation
      ::=  module _identifier [ parameter_value_assignment ] module_instance {,
           module_instance };

parameter_value_assignment
      ::=  # ( expression {, expression } )

module_instance
      ::=  name_of_instance ([list_of_module_connections])

name_of_instance
      ::=  module_instance_identifier [ range ]

list_of_module_connections
      ::=  ordered_port_connection {, ordered_port_connection }
       |   named_port_connection {, named_port_connection }

ordered_port_connection
      ::=  [ expression ]

named_port_connection
      ::=  .port_identifier ( [ expression ] )
```

5.2 Parameters

Parameters allow us to enter define names for values and expressions that will be used in a module's description. Some, for instance a localparam, allow for the specification of a constant, possibly through a compile-time expression. Others (parameter) allow us to define a generic module that can be parameterized for use in different situations. Not only does this allow us to reuse the same module definition in more situations, but it allows us to define generic information about the module that can be overridden when the module is instantiated.

```
module xor8
    (output [1:8]   xout,
     input  [1:8]   xin1, xin2);

    xor     (xout[8], xin1[8], xin2[8]),
            (xout[7], xin1[7], xin2[7]),
            (xout[6], xin1[6], xin2[6]),
            (xout[5], xin1[5], xin2[5]),
            (xout[4], xin1[4], xin2[4]),
            (xout[3], xin1[3], xin2[3]),
            (xout[2], xin1[2], xin2[2]),
            (xout[1], xin1[1], xin2[1]);
endmodule
```

Example 5.1 An 8-Bit Exclusive Or

Example 5.1 presents an 8-bit XOR module that instantiates eight XOR primitives and wires them to the external ports. The ports are 8-bit scalars; bit-selects are used to connect each primitive. In this section we develop a parameterized version of this module.

First, we replace the eight XOR gate instantiations with a single assign statement as shown in Example 5.2, making this module more generally useful with the parameter specification. Here we specify two parameters, the width of the module (4) and its delay (10). Parameter specification is part of module definition as seen in the following syntax specification:

```
module xorx
    #(parameter width = 4,
                delay = 10)
    (output [1:width]  xout,
     input  [1:width]  xin1, xin2);

    assign #(delay) xout = xin1 ^ xin2;
endmodule
```

Example 5.2 A Parameterized Module

module_declaration
 ::= module_keyword module_identifier [module_parameter_port_list]
  [list_of_ports] ;
  { module_item }
  endmodule
 | module_keyword module_identifier [module_parameter_port_list]
  [list_of_ports_declarations] ;
  { non_port_module_item }
  endmodule
 ::=

module_parameter_port_list
 ::= # (parameter_declaration { , parameter_declaration })

parameter_declaration ::=
 ::= **parameter** [**signed**] [range] list_of_param_assignments ;
 | **parameter integer** list_of_param_assignments ;
 | **parameter real** list_of_param_assignments ;
 | **parameter realtime** list_of_param_assignments ;
 | **parameter time** list_of_param_assignments ;

The module_parameter_port_list can be specified right after the module keyword and name; the types of parameters that can be specified include signed, sized (with a range) parameters, as well as parameter types integer, real, realtime, and time.

Local parameters have a similar declaration style except that the localparam keyword is used instead of parameter.

local_parameter_declaration ::=
 ::= **localparam** [**signed**] [range] list_of_param_assignments ;
 | **localparam integer** list_of_param_assignments ;
 | **localparam real** list_of_param_assignments ;
 | **localparam realtime** list_of_param_assignments ;
 | **localparam time** list_of_param_assignments ;

These cannot be directly overridden and thus are typically used for defining constants within a module. However, since a local parameter assignment expression can contain a parameter (which can be overridden), it can be indirectly overridden.

When module **xorx** is instantiated, the values specified in the parameter declaration are used. This is a *generic* instantiation of the module. However, an instantiation of this module may override these parameters as illustrated in Example 5.3. The "#(4, 0)" specifies that the value of the first parameter (width) is 8 for this instantiation, and the value of the second (delay) is 0. If the "#(4, 0)" was omitted, then the values specified in the module definition would be used instead. That is, we are able to override the parameter values on a per-module-instance basis.

```
module overriddenParameters
    (output [3:0]    a1, a2);

    reg[3:0]         b1, c1, b2, c2;

    xorx     #(4, 0) a(a1, b1, c1),
                     b(a2, b2, c2);
endmodule
```

Example 5.3 Overriding Generic Parameters

The order of the overriding values follows the order of the parameter specification in the module's definition. However, the parameters can also be explicitly overridden by naming the parameter at the instantiation site. Thus:

```
        xorx      #(.width(4), .delay(0)
                                        a(a1, b1, c1),
                                        b(a2, b2, c2);
```

would have result as the instantiation of **xorx** in Example 5.3. With the explicit approach, the parameters can be listed in any order. Those not listed at the instantiation will retain their generic values.

The general form of specifying parameter values at instantiation time is seen in the following syntax specification:

```
module instantiation
      ::=  module _identifier [ parameter_value_assignment ] module_instance { ,
           module_instance };

parameter_value_assignment
         # (list_of_parameter_assignments} )

list_of_parameter_assignments
      ::=  ordered_parameter_assignment { , ordered_parameter_assignment}
      |    named_parameter_assignment { , named_parameter_assignment}

ordered_parameter_assignment
      ::=  expression

named_parameter_assignment
      ::=  . parameter_identifier ([expression])
```

This shows the syntax for either the ordered or named parameter lists.

Another approach to overriding the parameters in a module definition is to use the *defparam* statement and the hierarchical naming conventions of Verilog. This approach is shown in Example 5.4.

Using the *defparam* statement, all of the respecifications of parameters can be grouped into one place within the description. In this example, the delay parameter of instance **b** of module **xorx** instantiated within module **xorsAreUs** has been changed so that its delay is five. Module annotate uses hierarchical naming to affect the change. Thus, the parameters may be respecified on an individual basis. The general form of the defparam statement is:

```
parameter_override
       ::=  defparam
            list_of_param_assignments ;
```

```
module xorsAreUs
      (output  [3:0]    a1, a2);

      reg [3:0]           b1, c1, b2, c2;

      xorx      a(a1, b1, c1),
                b(a2, b2, c2);
endmodule

module xorx
      #(parameter  width = 4,
                   delay = 10)
      (output  [1:width]  xout,
       input   [1:width]  xin1, xin2);

      assign #delay xout = xin1 ^ xin2;
endmodule

module annotate;
      defparam
         xorsAreUs.b.delay = 5;
endmodule
```

Example 5.4 Overriding Parameter Specification With defparam

The choice of using the defparam or module instance method of modifying parameters is a matter of personal style and modeling needs. Using the module instance method makes it clear at the instantiation site that new values are overriding defaults. Using the defparam method allows for grouping the respecifications in specific locations. Indeed, the defparams can be collected in a separate file and compiled with the rest of the simulation model. The system can be changed by compiling with a different defparam file rather than by re-editing the entire description. Further, a separate program could generate the defparam file for back annotation of delays.

5.3 Arrays of Instances

The definition of the **xor8** module in Example 5.1 was rather tedious because each XOR instance had to be individually numbered with the appropriate bit. Verilog has a shorthand method of specifying an array of instances where the bit numbering of each successive instance differ in a controlled way. Example 5.5 shows the equivalent redefinition of module **xor8** using arrays of instances. This is equivalent to the original module **xor8** in Example 5.1. The array of instances specification uses the optional range specifier to provide the numbering of the instance names.

```
module xor8
    (output [1:8]   xout,
     input  [1:8]   xin1, xin2);

    xor a[1:8] (xout, xin1, xin2);
endmodule
```

Example 5.5 Equivalent xor8 Using Array of Instances

There are no requirements on the absolute values or the relationship of the msb or lsb of the range specifier (the [1:8} in this example) — both must be integers and one is not required to be larger than the other. Indeed, they can be equal in which case only one instance will be generated. Given msb and lsb, 1 + abs(msb-lsb) instances will be generated.

This example showed the case where each instance was connected to a bit-select of the outputs and inputs. When the instances are generated and the connections are made, there must be an equal number of bits provided by the terminals (ports, wires, registers) and needed by the instances. In this, eight instances needed eight bits in each of the output and input ports. (It is an error if the numbers are not equal.) However, instances are not limited to bit-select connections. If a terminal has only one bit (it is scalar) but there are n instances, then each instance will be connected to the one-bit terminal. Example 5.6 shows D flip flops connected to form a register. The equivalent module with the instances expanded is shown at the bottom. Note that **clock** and **clear**, being one-bit scalars, are connected to each instance.

```
module reggae
    (output [7:0]   Q,
     input  [7:0]   D,
     input          clock, clear);

    dff     r[7:0]  (Q, D, clear, clock);
endmodule

module regExpanded
    (output [7:0]   Q,
     input  [7:0]   D,
     input          clock, clear);

    dff     r7  (Q[7], D[7], clear, clock),
            r6  (Q[6], D[6], clear, clock),
            r5  (Q[5], D[5], clear, clock),
            r4  (Q[4], D[4], clear, clock),
            r3  (Q[3], D[3], clear, clock),
            r2  (Q[2], D[2], clear, clock),
            r1  (Q[1], D[1], clear, clock),
            r0  (Q[0], D[0], clear, clock);
endmodule
```

Example 5.6 A Register Using Arrays of Instances

5.4 Generate Blocks

Using arrays of instances is limited to fairly simple repetitive structures. Generate blocks provide a far more powerful capability to create multiple instances of an object. The primary objects that can be generated are: module and primitive instances, initial and always procedural blocks, continuous assignments, net and variable declarations, task and function definitions, and parameter redefinitions.

Continuing with the **xorx** examples of the chapter, Example 5.7 illustrates using a generate statement to re-describe the module. The generate…endgenerate block specifies how an object is going to be repeated. Variables for use in specifying the repetition are defined to be genvars. Then a for loop is used to increment (or decrement) the genvars over a range. The use of the genvars in the object to be repeated then specify such information as bit-selects.

```
module xorGen
    #(parameter width = 4,
                delay = 10)
    (output [1:width]  xout,
     input  [1:width]  xin1, xin2);

    generate
       genvar i;
       for (i = 1; i <= width; i=i+1) begin: xi
           assign #delay
                  xout[i] = xin1[i] ^ xin2[i];
       end
    endgenerate
endmodule
```

Example 5.7 A Generate Block

In this example, the genvar is **i**.

The following four copies of the assign statement are generated:

```
assign #delay     xout[1] = xin1[1] ^ xin2[1];
assign #delay     xout[2] = xin1[2] ^ xin2[2];
assign #delay     xout[3] = xin1[3] ^ xin2[3];
assign #delay     xout[4] = xin1[4] ^ xin2[4];
```

Since the generate statement is executed at elaboration time, these four statements become part of module **xorGen** replacing the generate…endgenerate statement.

The statements controlling the generation of objects within a generate…endgenerate block are limited to for, if-then-else, and case. The index of the for loop must be a genvar and both assignments in the for must be to the same genvar. The genvar can only be assigned a value as part of the for loop and it can only take on the values of 0 and positive integers. The genvar declaration may be outside of the generate…endgenerate block, making it available to other generate blocks. A named begin…end block must be used to specify the contents of the for loop; this provides hierarchical naming to the generated items.

The generate block of Example 5.7 could also have been written to instantiate primitive gate instances or always blocks. Shown below is the gate primitive version. In this case four XOR gates would be instantiated. The gates would have hierarchical names of xi[1] … xi[4], assuming that the generic instantiation of the module was specified.

```
generate
    genvar i;
    for (i = 1; i <= width; i=i+1) begin: xi
        xor #delay a (xout[i], xin1[i], xin2[i]);
    end
endgenerate
```

The always block version is shown below. The result would be four always blocks with the indicies replaced by 1 through 4.

```
generate
    genvar i;
    for (i = 1; i <= width; i=i+1) begin: xi
        always @(*)
            xout[i] = xin1[i] ^ xin2[i];
    end
endgenerate
```

Consider modeling an n-bit adder (where n is greater than 1) that also has condition code outputs to indicate if the result was negative, produced a carry, or produced a two's complement overflow. In this case, not all generated instances of the adder are connected the same. if-then-else and case statements in the for loop may be used to generate these differences. Example 5.8 shows a module using a case statement in the generate to produce different adder logic depending on which bit is being generated.

Three different situations are broken out for separate handling. For most of the stages, the carry in of a stage is connect to the carry out of the previous stage. For the least significant bit (bit 0), the carry in is connected to the module's carry in (**cIn**). For the most significant bit (which is parameterized as **width**), the carry out (**cOut**), **overFlow** and negative (**neg**) outputs must be connected.

```
module adderWithConditionCodes
    #(parameter                 width = 1)
    (output  reg  [width-1:0]   sum,
    output   reg                cOut, neg, overFlow,
    input         [width-1:0]   a, b,
    input                       cIn);

    reg           [width -1:0]  c;

    generate
        genvar i;
        for (i = 0; 1<= width-1; i=i+1) begin: stage
            case(i)
            0: begin
                    always @(*) begin
                    sum[i] = a[i] ^ b[i] ^ cIn;
                    c[i] = a[i]&b[i] | b[i]&cIn | a[i] & cIn;
                end
            end
            width-1: begin
                always @(*) begin
                    sum[i] = a[i] ^ b[i] ^ c[i-1];
                    cOut = a[i]&b[i] | b[i]&c[i-1] | a[i] & c[i-1];
                    neg = sum[i];
                    overFlow = cOut ^ c[i-1];
                end
            end
            default: begin
                always @(*) begin
                    sum[i] = a[i] ^ b[i] ^ c[i-1];
                    c[i] = a[i]&b[i] | b[i]&c[i-1] | a[i] & c[i-1];
                end
            end
            endcase
        end
    endgenerate
endmodule
```

Example 5.8 Generating an Adder

5.5 Exercises

5.1 Write a module with the structure:

```
module progBidirect (ioA, ioB, selectA, selectB, enable);
    inout   [3:0]   ioA, ioB;
    input   [1:0]   selectA, selectB;
    input           enable;
      ...
endmodule
```

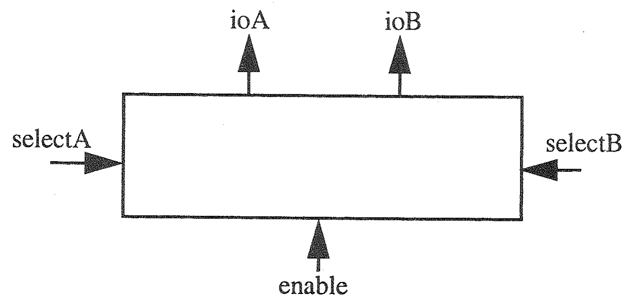
such that **selectA** controls the driving of **ioA** in the following way:

selectA	ioA
0	no drive
1	drive all 0's
2	drive all 1's
3	drive **ioB**

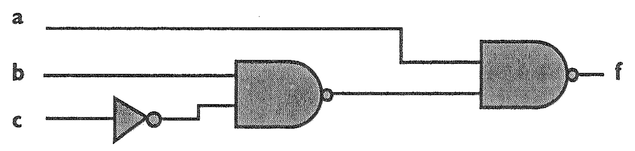
and **selectB** controls the driving of **ioB** in the same way. The drivers are only to be in effect if **enable** is 1. If **enable** is 0 the state of the **ioA** and **ioB** drivers must be high impedance.

A. Write this module using gate level primitives only.

B. Write this module using continuous assignments only.



5.2 The following combinational logic block has three inputs and an output. The circuit was built in some screwy technology and then analyzed. We now want to insert the correct input-to-output timing information into the circuit (internal node timings need not be correct).



Here are the circuit timings that must be represented in the circuit.

- The delay of a rising or falling edge on **a** or **b** to output **f**: 15 time units
- The delay of a rising or falling edge on **c** to output **f**: 10 time units

Yes, those times are rather strange given the logic diagram. However, this is a screwy technology and the transistor implementation made for some strange, but actual, time delays.

Assume **a**, **b**, and **c** are outputs of synchronously clocked flip flops. Write the structural Verilog description that will be correct in functionality and timing.