Fundamentals of Digital Logic and Microcomputer Design. M. Rafiquzzaman

APPENDIX

I

VERILOG

I.1 Introduction to Verilog

Verilog describes a digital system as a set of modules. A module is a basic block in Verilog. A typical Verilog segment is given below:

```
module <module name>  // A typical Module
<port list>
<declarations>
<module items>
endmodule
```

In the above, the module is defined by the keyword `module` and endeded by the keyword `endmodule`. The <module name> identifies a module uniquely. This means that a name or an identifier is assigned to a module to identify it. This name must start with an alpha character rather than a number. The two slashes (//) shown in the above Verilog module is used before a single line comment. Verilog module, when invoked, creates a unique object containing its name, variables, parameters, and input/output interface. The objects are called instances and the process of obtaining objects from modules are known as instantiation. Each port in the <port list> is defined by keywords `input` and `output` based on the port directions. Verilog also supports bidirectional ports which can be defined by keyword `inout`. The ports are included in parentheses with commas separating them. A semicolon (;) is used to terminate the port statement. Ports provide the module with a means to connect to other modules. The wire declaration by keyword `wire` provides internal connection in Verilog. All port declarations in Verilog are inherently defined as `wire`. This means that a port is automatically declared as a wire if it is defined as `input` or `output`, or `inout`.

Verilog includes a set of built-in logic gates such as OR, AND, XOR, NOT, NOR, NAND, and XNOR. The outputs of these gates are one-bit data and are declared as `wire` in Verilog. The built-in gates are utilized to provide a structural design called *netlist*. The Netlist facilitates connections between one-bit wires and logic gates. Ports can be internal or external to a module. Certain rules for port connections must be followed for the Verilog simulator when modules are instantiated within other modules. Input ports must be of the type Net (for all) internally. On the other hand, the inputs can be connected externally to a variable which is `reg` or a `wire`. The output ports can be of the type `reg` or `wire` internally. Output must always be connected to a `wire` (not `reg`) externally. The `inout` ports must always be of type `wire`. `inout` ports must be connected to `wire` externally.

Nets mean connection between hardware elements. Nets are driven continuously

by the outputs of devices they are connected to. Nets are typically declared by the keyword `wire`. Net is a class of data that includes `wire` as one data type. Verilog registers (defined by keyword `reg`) typically retain their values until a new value is stored. Verilog registers are different from hardware registers which need a clock. Verilog register does not require a clock. Also, Verilog register does not need a driver like the net. Values of Verilog registers can be changed anytime during simulation by replacing with another value.

Keywords `reg` and `wire` are one-bit wide by default. To define a wider `reg` or `wire`, the left and right bit positions are defined in square brackets separated by a colon. For example, `reg [7:0] a,b;` declares two variables a and b as 8 bits with the most significant bit as bit 7 (a[7] or b[7]) and the least significant bit as bit 0 (a[0] or b[0]). Verilog contains approximately 100 keywords. Verilog keywords and identifiers are case sensitive. This means that Full_adder and full_adder are distinct variables. Also, Verilog keywords are reserved, and cannot be used as names.

The <declarations> define data objects as registers or `wires`. The <module items> for behavioral modeling (to be discussed later) may be initial block or always block. Verilog uses keywords `begin` and `end` like Pascal to define a block. A typical initial block is defined by using keyword `initial`. The statements are contained between keywords `begin` and `end` as in conventional programs. The always block is defined in a similar manner except that `always` instead of `initial` is written before `begin`. The `always` block is executed continuously and cannot be interrupted unless time control feature of Verilog utilizing symbols such as @ is used. Note that the output of a typical combinational logic circuit is altered with changes in input(s). The Verilog simulator can use `always` along with the symbol @ to stop execution of the `always` block continuously until changes in one or more inputs occur. For example, the statement `always @ (a or b or c)` means that a, b, and c are three inputs to be used in the always block that follows. The symbol @ allows the simulator to execute an `initial` block that may follow as long as there are no changes in the inputs; however, the always block will be executed whenever changes in inputs occur. Note that all procedural blocks are active concurrently. Constants in Verilog are decimal integers by default. However, the syntax 'b,'d, or 'h can be used before a number to define it as binary, decimal or hexadecimal. Furthermore, the total number of bits in a number can be represented by placing the number before the quote. For example, 4'b1111 and 4'hf will represent 15 in decimal.

Verilog provides a conditional operator denoted by the symbol ?. For example, consider the statement, `assign z = s ? x : y;` . This means that if s=1 then z=x, else z=y for s=0. Note that in this expression, s is the condition, z=x is the true expression while z=y is the false expression. Also, Verilog keyword `parameter` declares and assigns value to a constant. For example, `parameter x = 5;` will assign the value of integer 5 to x. Nesting of modules is not permitted in Verilog. That is, a module cannot be placed between `module` and `endmodule` of another `module`. However, modules can be instantiated within other modules. This provides hierarchical modeling of design in Verilog. The name of a Verilog module is not available outside the module unless hierarchical modeling is used. The instance names must be defined when modules are instantiated.

Verilog offers a feature called reduction operator for the logic operations and, nand, or, nor, xor and xnor. The reduction operation is performed bitwise from right to left on the bits of the same word. As an example, consider the reduction operation &x where x is a 4-bit number. In this case, the operation &x means x[3]&x[2]&x[1]&x[0].

To precisely model all logical conditions in a circuit, each bit in Verilog can be

one of the following: 1'b0, 1'b1, 1'bz (high impedance), or 1'bx (don't care). 1'b0 and 1'b1 respectively correspond to 0 and 1. Verilog includes 1'bz for the situation when the designer needs to define a high impedance state. Furthermore, Verilog includes 1'bx to specify a don't care condition. Sometimes, miswiring of gates may also result into an unknown value of the output in certain situation. For example, if the designer makes a mistake and connects outputs of two gates together. This output may want to assume a value of either 0 or 1. This may cause physical damage to certain logic families. In order for the simulator to detect such problems, 1'bx (don't care) definition can be used for the output.

A Verilog simulator includes a built-in system function called `$time` for representing simulated time. This means that `$time` provides a measure of actual time for the hardware to function when fabricated. `$time` is expressed as an integer value rather than by time units such as seconds. However, designers typically use one `time` unit as one nanosecond. Time control statements may be included in Behavioral Verilog. A statement will not be executed with the symbol # followed by a number until the specified number of `time` steps has elapsed. This allows Verilog to model propagation delays of logic gates. The symbol # when used in test programs generates a sequence of patterns at particular `times` that will behave like inputs to the hardware being designed. Also, if the symbol @ is used before a statement, the statement that follows will not be executed until the statement with @ is completed.

The test bench for the simulation is normally written by the designer. The test bench tests the Verilog design by applying stimulas and providing outputs during simulation. Test benches utilize procedural blocks which start with either the keywords `initial` or `always` for providing stimulas for the test circuit. An example of a simple `initial` block is provided below:

```
initial
        begin
              #0
                x=1'b0; y=1'b0; z=1'b0;
              #50
                x=1'b0; y=1'b0; z=1'b1;
              #50
                x=1'b0; y=1'b1; z=1'b0;
         end
```

In the above, keywords `begin` and `end` are used to define the block with the time units defined by the symbol #. At time = 0, x = 0, y = 0 and z = 0. At time = 50 ns, x = 0, y = 0 and z = 1. Finally, at time = 100 ns, x = 0, y = 1 and z = 0.

A simple test bench has the following structure:

<module name>
<reg and wire declarations>
<Instantiate the Verilog design>
<Generate stimulus using `initial` and `always` keywords>
<Produce the outputs using $monitor for verification>
endmodule

The inputs applied to the test (design) block for simulation are declared in the stimulus block as `reg` data type. The outputs (responses) of the test block that are to be monitored and verified are declared as `wire` data type. The test block has no inputs or outputs. The stimulus block produces inputs for the test block and verifies the output of the

test block. `initial` and `always` procedural blocks can be used to produce the output. The simulator can represent the output as waveforms or in tabular form using Verilog system tasks such as $monitor. The syntax for $monitor is provided below:

```
$monitor ( "time = %d x = %2d y = %3d z = %2b",
           $time, x, y, z);
```

Verilog system task, `$monitor` can be used to display the output of the design block under test. Verilog simulator allows the output to be represented in binary (%b or %B), octal (%o or %O), decimal (%d or %D) or hexadecimal (%h or %H). $time is a built-in function that provides the simulation time. In the above $monitor statement time, x, and y are displayed in decimal while z is represented in binary. Another way to display the output is by using system task `$display`. Note that `$display` is used to display one time value of variables. In contrast $monitor displays variables whenever changes in variables occur during simulation. The syntax for `$display` is `$display ("%b%d",x,y);` which will display x in binary and y in decimal. As mentioned before, there are three levels of abstractions in Verilog. These are Structural, dataflow, and behavioral modeling. They can be combined in an application. These abstractions are described along with Verilog programming examples.

Verilog provides primitives which can be defined by the user to represent truth table in a tabular form. These primitives are called User-Defined Primitives (UDP). UDP descriptions are enclosed by keywords `primitive` and `endprimitive` rather than keywords `module` and `endmodule`. There are two types of UDPs. These are Combinational UDPs used for combinational circuits and Sequential UDPs used for sequential circuits. As an example, a Verilog description using Combinational UDP for the 2-to1 multiplexer of Table 4.11 is provided below. The truth table for the 2-to-1 multiplexer from Table 4.11:

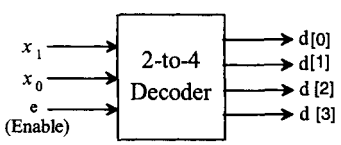
Select input, S	Output, Z
0	d_0
1	d_1

```
//2to1 multiplexer
primitive mux2to1 (z,d0,d1,s);
output z;
input d0,d1;
input s;
//Truth table is enclosed by keywords table and endtable
//The inputs are listed in order followed by colon(:)
//The output is always the last entry followed by semicolon(;)
//The symbol? in the table is used to represent don't care
//condition
 table
  //  d0  d1  s  :  z
      1   ?   0  :  1;
      0   ?   0  :  0;
      ?   1   1  :  1;
      ?   0   1  :  0;
 endtable
endprimitive
// stimulus for 2to1 mux using UDP
module mux_stimulus;
reg i0,i1;
reg s;
```

```
wire out;
mux2to1mux(out,i0,i1,s);
initial
begin
// set inputs
i0=1, i1=0;
#1 $display("i0=%b,i1=%b",i0,i1);
//select i0
s=0;
#1 $display("s=%b,out=%b",s,out);
//select i1
s=1;
#1 $display("s=%b,out=%b",s,out);
end
endmodule
//simulation outputs
i0=1,i1=0
s=0, out=1
s=1, out=0
```

I.1.1 Structural Modeling

The following Verilog structural description is provided for the 2-to-4 decoder of Figure 4.14. The figure is redrawn below for convenience:




```
// Structural description of a 2-to-4 decoder
module decoder2to4 (x1, x0, e, d);
    input x1, x0, e;
    output [0:3] d; //output vector d must be declared as wire.
    wire [0:3] d;   //if vector d is not declared as wire, Verilog
    wire x11, x00;  //will make vector d one bit by default.
    not
        inv1 (x11, x1),
        inv2 (x00, x0);
    and
        and1 (d[0], x11, x00,e),
        and2 (d[1], x11, x0, e),
        and3 (d[2], x1, x00, e),
        and4 (d[3], x1, x0, e);
endmodule
```

The above structural description for the 2-to-4 decoder contains three inputs (x1, x0, e), and four outputs (d[0] through d[3]). The wire declaration provides internal connections. Two NOT gates are used to obtain complements x11 and x00 of the inputs x1 and x0 respectively while the four AND gates are used for the outputs d[0] through d[3]. In the gate list such as `and1 (d[0], x11, x00,e);` , the output d[0] is always listed first followed by inputs x11, x00, and e. The keyword `and` is written once for all AND operators, and in this case, provides output d[0] by logically ANDing x11, x00, and e.

Note that the Verilog keywords and names are case sensitive. Also, Verilog keywords are reserved, and cannot be used as names. Note that if a Verilog operation is required several times in a program such as `not` requiring twice in the above , the Verilog code can be written in two ways. The two `not` operations, in the above, are written using the keyword `not` followed by two different labels inv1 and inv2 separated by commas, and terminated by ;. An alternate Verilog code for the two `not` operations can be written as follows:

```
not (x11, x1);
not (x00, x0);
```

Similarly, alternative codes for other logic operations in the above can be written. A module instantiation statement associates the signals in the module instantiation with the ports in a module definition. There are two ways to represent the association. These are positional association, and named association. These two methods cannot be mixed. In positional association, each signal in the module instantiation is mapped by position to the corresponding signal in the module definition.

In order to illustrate positional association, consider the following Verilog program:

```
module system;
    wire [3:0] d;
    subsystem f1 (d[3], d[1], d[2], d[0]);
endmodule
module subsystem (w, x, y, z);
    input x, y;
    output w, z;
endmodule
```

In the above program, the module system has an instance of the module subsystem inside it. The connections to the subsystem are made by placing the bit vectors of the identifier (d in this case) at the desired positions in the port definitions of the subsystem module. In the above, d[3] is associated with w, d[1] with x, d[2] with y, and d[0] with z. The ordering must be done properly. Therefore, in the positional association, the names of the connecting signals must be included at the appropriate positions in the module port list. Positional association is used for small systems while named association is used for large systems.

In the named association, Verilog connects external signals by the port names rather than by positions. The port connections can be specified in any order as long as the port names in the module definition precisely match the external signals. For example, the above Verilog program with positional association can be rewritten using named association as follows:

```
module system;
    wire [3:0] d;
    subsystem f1 (.w(d[0] ), .x(d[3] ), .y(d[2] ), .z(d[1] ));
endmodule
module subsystem (w, x, y, z);
    input x, y;
    output w, z;
endmodule
```

In the above, d[0] is associated with w, d[1] with z, d[2] with y, and d[3] with x. The ordering of the ports of instance f1 of subsystem module is not important because the signals are associated by names. Note that if an instance of a module contains an unconnected port, the position of the port in the instantiation is left empty. For example, consider a module representing a three-input OR gate with declaration as or3 (f, a, b, c); . If it is desired to keep the input at position b unconnected, an instance of or3 will be

or3 (f, a, , c); . Note that an unconnected module input is placed in high impedance state automatically, and unconnected outputs are not used.

I.1.2 Dataflow Modeling

Dataflow modeling in Verilog allows a digital system to be designed in terms of its function. Dataflow modeling utilizes Boolean equations, and uses a number of operators that can act on inputs to produce outputs. Some of the operators are listed in the table below:

Verilog operators

Operation	Symbol
Arithmetic addition	+
Subtract	--
NOT of a single bit	!
AND between two operands	&&
OR between two operands	||
Bit-by-bit NOT	~
Bit-by-bit logical AND	&
Bit-by-bit logical OR	|
Bit-by-bit XOR	^
Bit-by-bit XNOR	~^ or ^~
Logical Equality	==
Less than	<
Greater than	>
Conditional	?
Concatenation	{ }

All Boolean equations are executed concurrently whenever any one of the values on the right hand side of one or more equations changes. This is accomplished using Verilog's continuous assignment statement. This statement uses the keyword `assign`. A continuous assignment statement is used to assign a value to a net. A net is not a verilog keyword. It is used to specify the output (defined by `output` or `wire` using declaration statements) of a gate. For example, consider the following assignment statement:

```
assign e = (a ^ b) & (~ c | d);
```

The Boolean expression on the right hand side of the above equation is first evaluated, and the AND gate output is connected to wire e. In order to illustrate dataflow modeling in Verilog, consider the following program for a 2-to-4 decoder:

```
module decoder2to4 (e, a, b, d0, d1, d2, d3);
    input e, a, b;
    output d0, d1, d2, d3;
    assign d0 = (e & ~a & ~b);
    assign d1 = (e & ~ a & b);
    assign d2 = (e & a & ~b);
    assign d3 = (e & a & b);
endmodule
```

The above dataflow program uses Verilog keyword `assign` followed by Boolean equations using Boolean operators.

I.1.3 Behavioral Modeling

The Behavioral description in Verilog is used to describe the function of a design in an algorithmic manner. Behavioral modeling is used in the initial stages of a design process to determine design-related tradeoffs. Behavioral modeling in Verilog uses constructs similar

to C language constructs. Verilog provides two types of procedural blocks. They are represented using keywords `initial` (an `initial` block executes once), and `always` (an `always` block executes continuously until simulation ends). The designer typically uses "initial" procedural block to provide initializations for a simulation, and produce stimulus waveforms for a simulation test bench.

The "always" procedural block provides a cyclic activity flow from simulation time of zero. This means that the procedural statements in the always block are executed continuously until simulation ends. The procedural statements in behavioral modeling execute sequentially in the order they are listed in the source code. The outputs of the procedural statements must be declared by the keyword `reg`. Input ports cannot be declared as `reg` since they do not normally retain values, rather affect the changes in the external signals they are connected to. Note that a `reg` data type retains its value until a new value is assigned. As an illustration of behavioral modeling, Consider the following Verilog program written using Behavioral modeling for the 2-to-4 decoder:

```
module decoder2to4 (e, i, d);
output [3:0] d;
input [1:0]i;
input e;
reg [3:0] d;

always @ (i or e)
       if (e==1)
              begin
                     case (i)
                            0: d = 4'b 0001;
                            1: d = 4'b 0010;
                            2: d = 4'b 0100;
                            3: d = 4'b 1000;
                            default d = 4'b xxxx;
                     endcase
              end
      else
            d = 4'b 0000;
endmodule
```

In the above, i (2-bit) and e (1-bit) are declared as inputs while d is declared as 4-bit `reg` output. The conditional statement `if-else` allows execution of the `case` statements if e=logic 1. Note that the decoder is enabled when enable line, e equals logic 1. The logical operator == is used for logical equality in the if expression. If e= logic 1 , the statements (between `case` and `endcase`) are executed sequentially. The statement `if (e==1)` is executed as soon as any of the inputs after `@` in the `always` statement changes. The `case` statement is used for multiple branching. For example, `case(i)` determines the value of the 2-bit vector, i and compares it with the values with the list of the statements. The assignment statement associated with the first value that matches is executed. Since the vector i is a two-bit vector, it can be any of the four values from 0 to 3. For example, consider the statement 2: d= 4'b0100; . If i $= 10_2$(2 in decimal), then the `case` statement after executing 2: d= 4'b0100; will assign four-bit vector, d with the binary value 0100. This means that the line 2 of the decoder output is high while others are low. An optional default value can be used for the `case` statement. This is for assigning other values such as don't care (x) or high impedance (z). Also, in the above, if e= logic

0, the 4-bit output vector,d is assigned with low values. This is shown as part of the `else` statement. This means that the decoder is disabled.

I.2 Verilog descriptions of typical combinational logic circuits

In the following, Verilog descriptions of typical combinational logic circuits are provided.

i) Write a Verilog description for a full adder using two half adders and an OR gate as described in Section 4.5.1.

Solution

Assume x, y, z as three inputs and cout,sum as the two outputs of the full adder. x and y can be applied as the inputs to the first half adder generating sum, s1 = x ⊕ y and carry, c1 = xy. s1 can be applied as one of the inputs to the second half adder with z as the other input. The second half adder will produce a sum,
sum = x ⊕ y ⊕ z which is the desired sum of the full adder. The carry output, c2 of the second half adder will be (x ⊕ y) z. c1 and c2 can be logically ORed together to provide the carry output (cout) of the Full adder.
The Verilog description is given below:

```
// Half  Adder
module half_adder (s,c,x,y);
  output s,c;
  input x,y;
  xor  (s,x,y);
  and  (c,x,y);
endmodule
// Full adder is obtained by instantiating half adder twice
// (Hierarchical modeling)
module full_adder (sum,cout,x,y,z);
  output  sum,cout;
  input  x,y,z;
  wire  s1,c1,c2;
  half_adder B1(s1,c1,x,y);
  half_adder B2(sum,c2,s1,z);
  or(cout,c1,c2);
endmodule
```

ii) Write a Verilog description along with the test bench for a 4-bit ripple-carry adder using behavioral modeling.

Solution

Although the following program may not be an efficient one, it is included for illustrative purposes. As mentioned before, the test bench usually does not have any inputs and outputs. The inputs applied for simulation are declared as `reg` data type while the outputs to be obtained from the simulation are declared as `wire` data type. Therefore, in this test bench, the inputs (a, b, cin) to the design module are declared as `reg` data while outputs (s, cout) are declared as `wire` data type. The initial block specifies several values to be applied during simulation. The outputs are verified with the $monitor system task. The simulator displays time, inputs, and outputs in binary (since %b is used) as soon as there is a change in one or more input values. Note that the concatenate operator { } in {cout,s} is used to combine cout and s as a 5-bit output.

```
// 4 bit adder
module adder4 (cout,s,a,b,cin);
       output cout;
       output[3:0] s;
       input[3:0] a,b;
       input cin;
       reg[3,0] s;
       reg cout;
       always @ (a or b or  cin)
begin
       {cout,s}= a+b+cin;
end
endmodule

// Test bench
module adder_test;

// declare variables
  reg  [3:0]  a,b;
  reg  cin;
  wire  [3:0]  s;
  wire  cout;

// Instantiate
adder4 A1 (cout,s,a,b,cin);
  initial
  begin
    $monitor ($time, "a=%b, b=%b, cin=%b, cout=%b, s=%b",
              a, b, cin, cout,s);
  end
// Stimulus inputs
  initial
  begin
       a = 4'b0001; b = 4'b0010; cin = 1'b0;
     #10      a = 4'b0101; b = 4'b0010;
     #10      a = 4'b1000; b = 4'b1010;
     #10      a = 4'b1001; b = 4'b0111;
  end
endmodule
// Simulation outputs
       0 a = 0001,   b = 0010,  cin = 0,  cout = 0, s = 0011
       10 a = 0101,  b = 0010,  cin = 0,  cout = 0, s = 0111
       20 a = 1000,  b = 1010,  cin = 0,  cout = 1, s = 0010
       30 a = 1001,  b = 0111,  cin = 0,  cout = 1, s = 0000
```

iii) Write a Verilog description for a BCD to seven-segment code converter (Section 4.4) for driving a common-cathode display for displaying the decimal digits 2, 4, and 9. The converter will turn the display OFF for any other inputs.

Solution

```
module code_converter (bcd_in,seven_seg_out);
  input  [3:0]  bcd_in;
  output [6:0] seven_seg_out;
  reg    [6:0]   seven_seg_out;
 //       bcd_in  =    abcdefg
```

```
  parameter  two  = 7'b1101101;
  parameter  four = 7'b0110011;
  parameter  nine = 7'b1110011;
  parameter other = 7'b0000000;
   always @ (bcd_in)
       case  (bcd_in)
         2:   seven_seg_out = two;
         4:          seven_seg_out = four;
         9:          seven_seg_out = nine;
         default:  seven_seg_out = other;
       endcase
endmodule
```

EXAMPLE I.1

Write a Verilog description for $f = A + B\overline{C}$ (Section 3.6) using structural modeling.

Solution

```
// file name: func.v
//written using structural modeling
module func(a, b, c, f);
  input a, b, c;
  output f;
  wire y0, y1;
  not(y0, c);
  and(y1, b, y0);
  or(f, y1, a);
endmodule
```

EXAMPLE I.2

Write a Verilog description for a two-input exclusive-OR gate using structural modeling.

Solution

The program is written as follows:

```
// Exclusive OR operation
// file name: xor_1.v
module xor_1 (a, b, y);
  input a, b;
  output  y;
  xor (y, a, b);
endmodule
```

EXAMPLE I.3

Write a Verilog description for a 2 to 4 decoder with one high enable as described in section 4.5.3. Use (a) behavioral modeling (b) dataflow modeling .

Solution

(a) Using behavioral modeling:

Note that { } is concatenate operator in Verilog.

```
module decoder(Y3, Y2, Y1, Y0, A, B, en);
  // Define inputs and outputs
  output Y3, Y2, Y1, Y0;
  input A, B;
  input en;
  reg Y3, Y2, Y1, Y0;
```

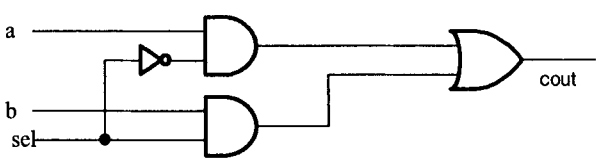
```
  always @(A or B or en)
begin
// Use behavioral  method for decoder
if (en == 1)
begin
       case ( {A,B} )
              2'b00: {Y3,Y2,Y1,Y0} = 4'b0001;
              2'b01: {Y3,Y2,Y1,Y0} = 4'b0010;
              2'b10: {Y3,Y2,Y1,Y0} = 4'b0100;
              2'b11: {Y3,Y2,Y1,Y0} = 4'b1000;
              default: {Y3,Y2,Y1,Y0} = 4'bxxxx;
       endcase
end
if (en == 0)
{Y3,Y2,Y1,Y0} = 4'b0000;
end
endmodule
```

(b) Using dataflow modeling:

```
// 2-to-4 decoder
// file name: decoder.v
module decoder(E, X, Y, Z0, Z1, Z2, Z3);
  output Z0, Z1, Z2, Z3;
  input E, X, Y;
  assign Z0 = E & ~X & ~Y;
  assign Z1 = E & ~X & Y;
  assign Z2 = E &  X & ~Y;
  assign Z3 = E &  X & Y;
endmodule
```

EXAMPLE I.4

Write a Verilog description for the 2-to-1 multiplexer of figure 4.21 using structural modeling. Figure 4.21 is redrawn below:



Solution

```
   // file name: mux2.v
module mux2(a, b, sel, cout);
   // I/O port declarations
  output cout;
  input a, b, sel;
   // Internal nets
  wire y0, y1, y2;
  // Instantiate logic gate primitives
  not(y0, sel);
  and(y1, a, y0);
  and(y2, b, sel);
  or(cout, y1, y2);
endmodule
```

EXAMPLE I.5

Write a verilog description for a four-bit binary adder using hierarchical modeling.

Solution

```
// Define a 1-bit full_adder
// file name: fulladd.v
module fulladd(sum, c_out, a, b, c_in);

// I/O port declarations
  output sum, c_out;
  input a, b, c_in;

// Internal nets
  wire s1, c1, c2;

// Instantiate logic gate primitives
  xor (s1, a, b);
  and (c1, a, b);

  xor (sum, s1, c_in);
  and (c2, s1, c_in);
  or (c_out, c2, c1);

endmodule

// Define a 4-bit binary adder
module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations
  output [3:0] sum;
  output c_out;
  input [3:0] a, b;
  input c_in;

// Internal nets
  wire c1, c2, c3;

// Instantiate four 1-bit full adders.
fulladd fa0(sum[0], c1, a[0], b[0], c_in);
fulladd fa1(sum[1], c2, a[1], b[1], c1);
fulladd fa2(sum[2], c3, a[2], b[2], c2);
fulladd fa3(sum[3], c_out, a[3], b[3], c3);
endmodule
```

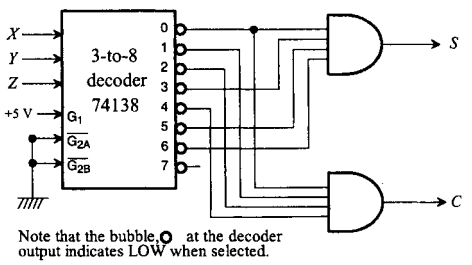
Note: In Verilog, nesting of modules is not permitted. That is, a module cannot be placed between module and endmodule of another module. However, modules can be instantiated within other modules. This provides hierarchical modeling of design in Verilog. In the above program, the full-adder is defined by instantiating primitive gates. The next module describes the 4-bit binary adder by instantiating four full-adders. The instantiation is done by using the name of the module that is instantiated with the same port names in this case.

EXAMPLE I.6

Write a Verilog description for a full-adder using 74138 decoder and gates (Figure 4.17).

Solution

This problem implements a full adder using a 3to8 decoder and two 4 input AND gates as shown in figure 4.17 in the text book. Behavioral modeling is used for implementation of 3to8 decoder and the 4 input AND gate while Structural modeling is used for the interconnection of the decoder with the AND gates using the schematic of figure 4.17 as follows:



The 74138 is a 3to8 decoder with an active low output when selected and the outputs are only driven if the chip enable lines are in a valid state (G1, $\overline{G2A}$, $\overline{G2B}$ = 100_2). If the decoder is not selected, the outputs are tristated.

For the 4 input AND gate, the inputs are ANDed using the bit-wise AND operator "&".

```
//Description: Full Adder Using 3-to-8 MUX with AND gates.
//implementation of a full adder using 2 four input
//AND gates and one 3to8 decoder-74138

//APPROACH:Behavioral for the implementation of the decoder and 4 input
//AND gates.
//Structural approach when combining the decoder and AND gates,
//decoder74138     3 to 8 decoder with active low outputs.

//INPUTS:  --X, Y, Z( select lines )
//         --G1, nG2A, nG2B ( enable lines)
//         Out[7:0] ( eight output lines)

//OUTPUTS: --high impendance "Z" outputs when chip not selected
//      --active low output on line selected. (if chip selected)
module decoder74138 (nout, G1, nG2A, nG2B, X , Y , Z);
    output [7:0] nOut;
    input G1, nG2A, nG2B, X, Y, Z;
    reg [7:0] nOut;
    always @(G1 or nG2A or nG2B or X or Y or Z)
begin
  if((G1, nG2A , nG2B) ==3'b100)
//  chip enabled
    begin
// select conditions for select lines w/ active low outputs
    case ( { X, Y, Z})
        0: nOut[7:0] = 8'b1111_1110;
        1: nOut[7:0] = 8'b1111_1101;
        2: nOut[7:0] = 8'b1111_1011;
        3: nOut[7:0] = 8'b1111_0111;
        4: nOut[7:0] = 8'b1110_1111;
        5: nOut[7:0] = 8'b1101_1111;
        6: nOut[7:0] = 8'b1011_1111;
        7: nOut[7:0] = 8'b0111_1111;
```

```
            default nOut [7:0] = 8'bx;   //this should never happen
        endcase
    end
  else
// chip disabled
        begin
                nOut [7:0] = 8'hzz;
        end
end
endmodule
//AND4: 4 input and gate

//INPUTS:  --A,B,  C,D

//OUTPUTS: --Out   AND output of all four inputs

module AND4 (Out,A,B,C,D);
    output Out;
    input A,B,C,D;
    reg Out;
    always@(A or B or C or D)
begin
    Out=A & B & C & D;
end
endmodule

//Full-Add:Full adder using 3to8 decoder 74138 and 2 four input AND gates
//INPUTS : -- X , Y , Z ( X bit to add, Y bit to add , Z carry to add )

//OUTPUTS:      --S = sum bit
//              --C = Carry out bit
module Full_Add (C,S,X,Y,Z);

    output C , S;
    input X , Y , Z;
    wire [7:0] decoder_out;

// 3 to 8 decoder enabled with bits to be added as inputs

decoder74138 decoder74138_0( decoder-out [7:0],1'b1,1'b0,1'b0, X , Y , Z);

// use 4 input AND gates to do final sum and carry

AND4AND4_0(S,decoder_out[0],decoder_out[3],decoder_out[5],decoder_out[6]);

AND4AND4_1(C,decoder_out[0],decoder_out[1],decoder_out[2],decoder_out[4]);
endmodule


//Full_Add_Test:  test bench for full adder implemented w/ 3to8 decoder
//and two 4 input AND gates

module Full_Add_Test;
    reg X , Y , Z;
    wire S , C ;
  Full_Add Full_Add_0 ( C,S,X,Y,Z);

initial
   $monitor("Time=%0d, X= %b, Y= %b, Z= %b, S= %b, C= %b",
             $time, X, Y, Z, S, C);
```

```
initial
begin
#0
X = 1'b0;Y = 1'b0;Z = 1'b0;
#50
X = 1'b0;Y = 1'b0;Z = 1'b1;
#50
X = 1'b0;Y = 1'b1;Z = 1'b0;
#50
X = 1'b1;Y = 1'b0;Z = 1'b0;
#50
X = 1'b1;Y = 1'b1;Z = 1'b0;
#50
X = 1'b0;Y = 1'b1;Z = 1'b1;
#50
X = 1'b1;Y = 1'b1;Z = 1'b1;
#50
X = 1'b0;Y = 1'b0;Z = 1'b0;
end
endmodule
```

Note: An alternative to Verilog code for the AND4 module in the above is provided below. The codes from `input` to `always` can be replaced by using the reduction operator & as follows:

```
input   [3:0] A;
reg     out;
assign  out = & A;
```

I.3 Verilog descriptions of typical synchronous sequential circuits

Sequential circuits are typically described in Verilog using behavioral modeling. Verilog utilizes two basic statements in behavioral modeling. They are represented using keywords `initial` and `always`. An `initial` block is created using an `initial` statement. The `initial` block executes once during simulation starting at time 0. For several blocks, each block executes concurrently at time 0. Each block completes its execution independent of the other blocks. Keywords `begin` and `end` are normally used to group multiple behavioral statements. Grouping is not required for a single behavioral statement. The `initial` blocks are typically used to provide initializations for a simulation and produce stimulus waveforms for a simulation test bench. An `always` block, on the other hand, is defined using an `always` statement. The `always` block executes the statements continuously starting at time 0 until simulation ends. Furthermore, Keywords `initial` and `always` can be used to generate a clock signal for simulating a sequential circuit. An example is provided below:

```
module clock;
reg clk;
initial
       clk=1'b0;
always
       #20 clk=~clk;
initial
```

```
        #2000 $finish;
endmodule
```

In the above, the `initial` statement starts the clock at time=0. The `always` statement complements the clock every 20 time units with a time period of 40 time units. The simulation is ended by the system task `$finish` at 2000 time units.

Verilog provides timing controls to specify the simulation at which procedural statements execute. Two such timing controls include delay- based timing control and event control. Delay-based timing control in an expression defines the time between start of execution of the statement and its completion. Symbol # is used to specify delays. An example is given below:

```
initial
begin
        #5  x=2; // Delay execution of x=2 by 5 time units
```

The event control expression, on the other hand, defines a condition based on the change in value in a register or a net to trigger execution of a statement or a block of statements. An event control is defined by the symbol @ along with the keyword `always`. Level-sensitive and edge-triggered events will be considered next. In synchronous sequential circuits, level-sensitive and edge-triggered flip-flops are encountered. The level-sensitive flip-flop can be accomplished by the following statement:

```
always @ (x or enable)
```

As soon as a change in x or enable occurs, the procedural statements in the `always` block will be executed. Verilog provides the keywords `posedge` and `negedge` to implement positive-edge triggered or negative-edge triggered clock. For example, the statements `always @ posedge clock` and `always @ negedge clock` will initiate execution of the procedural statements in the always block respectively for positive clock and negative clock. Since a sequential circuit is comprised of flip-flops and combinational circuits, it can be represented using behavioral and dataflow modeling. Flip-flops can be described with behavioral modeling using `always` keyword while the combinational circuit part can be assigned with dataflow modeling using `assign` keyword and Boolean equations.

Note that a behavioral model in Verilog is defined using the keyword `initial` or `always` followed by one or several procedural statements. The procedural statements in behavioral modeling execute sequentially in the order they are listed in the source code. The final output of these statements must be of the `reg` data type rather than `wire` (normally used for structural) data type. Note that `wire` continuously updates the output while the `reg` stores the value until a new value is provided.

Next, the meaning of "procedural statement" will be discussed. A procedural statement is an assignment in an `initial` or `always` statement. Also, procedural statement assigns value to a register (data objects of type `reg`). There are three types of procedural assignments. These are procedural assignment (uses = as the operator), continuous procedural assignment (uses keyword `assign` with = as the operator), and non-blocking procedural assignment (uses <= as the operator). The right hand side of a procedural assignment is an expression which must evaluate to a value while the left hand side is typically a `reg`. The procedural continuous assignment retains the last output (when a digital circuit is disabled) until it is enabled again. This is useful in modeling latches and flip-flops. The first two procedural assignments that use the = operator execute the statements sequentially. These statements are called blocking assignments. This means that in blocking assignment, the next procedural assignment must wait until the present

one is completed. In non-blocking procedural assignment, executions of the statements that follow are not blocked. This means that the right hand side of the expression is evaluated first, but assignment to the left hand side is not made until all expressions are evaluated. Next, consider an example of the following blocking assignments:

```
reg a, b, c;
reg [3:0] x, y;
//Must place Behavioral statements in initial or always block
initial
begin
       a=1; b=0; c=0;
       y= 4'b1111; x=y;
       #10  y[1]= 1'b0;
end
```

In the above, the statement `b=0` is executed only after `a=1` is executed. The statements in the `begin` and `end` block can only execute in sequence since blocking statements are used. All statements `a=1` through `x=y` are executed at time=0. However, statement `y[1]= 1'b0` is executed at time=10 since there is a delay of 10 time units in this statement.

As mentioned before, non-blocking assignments permit scheduling of assignments without blocking execution of the statements that follow. In order to illustrate non-blocking assignments, the previous example is modified as follows:

```
reg a, b, c;
reg [3:0] x, y;
//Must place Behavioral statements in initial or always block
initial
begin
       a=1; b=0; c=0;
       y= 4'b1111; x=y;
       y[1]  <= #10 1'b0;
       x[1:0]<= #5 2'b00
end
```

In the above, statements `a=1` through `x=y` are executed sequentially at time 0. Then, the two non-blocking assignments are executed simultaneously. The statement `y[1] =1'b0` is scheduled to execute after 10 time units while `x[1:0]= 2'b00` is scheduled to be executed after 5 time units. The simulator schedules execution of a non-blocking assignment , and then continues with the next statement in the block without waiting for completion of the present statement. When the two non-blocking statements in the above are executed, the right hand side expressions are evaluated first, and are stored in temporary locations. The assignments to the left hand side are made after both the expressions are completed. Non-blocking assignments are used in digital design where multiple concurrent data transfers such as in a register transfer, take place after a common event (positive or negative edge triggered clock).

For state machines, the inputs including clock, and outputs can be declared at the beginning of a Verilog program. The states can be defined using `parameter` keyword in Verilog which defines constants in a module. Statement using `always` along with `posedge` or `negedge` can be used for the clock. Statements using `case` and `if-else` can be used to implement various state transitions.

EXAMPLE I.7

Write a Verilog description for a D flip-flop (a) with a positive edge reset and a negative edge triggered clock. Use `if-else`.

(b) with a positive edge triggered clock and a negative edge clear input. Use `if-else`.

Solution

I.7 (a)

```
// D Flip-Flop
// Module DFF with synchronous reset
// file name: dfflop.v

module dfflop(q, d, clk, reset);
input d, clk, reset;
output q;
reg q;

//always do this when the reset is positive edge or clock is
//negative edge
always @(posedge reset or negedge clk)
// if it's reset q will equal to zero
if (reset)
        q = 1'b0;
// if it's clock q will equal to d
else
        q = d;
endmodule
```

I.7 (b)

```
// FileName: D.v
//description: D flipflop
module D_ff(Q, Q_bar, CLR, CLK, D);
output Q, Q_bar;
input CLR, CLK, D;

reg Q, Q_bar;
always @(posedge CLK or negedge CLR)
begin
//When CLR == 0 (neg logic) Q is always 0
//else @ rising edge of clock, Q <-- D
  if(!CLR)
    begin
      Q <= 1'b0;
      Q_bar <= 1'b1;
    end
  else
begin
      Q <= D;
      Q_bar <= !D;
    end
 //   Q_bar <= !D;

end

endmodule
```

EXAMPLE I.8

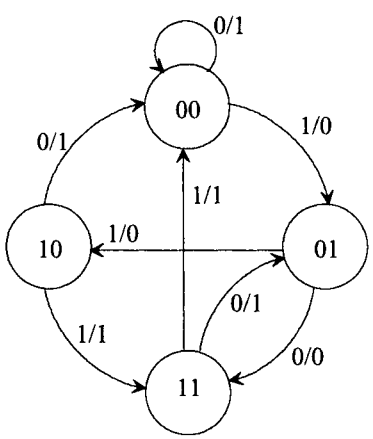
Write a Verilog description for a JK flip-flop with negative edge triggered clock. Use `case` statements.

Solution

```
// JK ff using case statements

// J=A and K=B as inputs

// Q and nQ are outputs

module jk_ff(A,B,clock,Q,nQ);

       input A,B,clock;
       output Q,nQ;
       reg Q;
       assign nQ=~Q
       always @ (negedge clock)
              case ({A,B})
               2'b00:Q=Q;
               2'b01:Q=1'b0;
               2'b10:Q=1'b1;
               2'b11:Q=~Q;
              endcase
endmodule
```

EXAMPLE I.9

Write a Verilog description for the state diagram of Figure 5.21. Use a reset input so that the hardware can be initialized. Figure 5.21 is redrawn below:



Solution

```
//Description:state machine of Example 5.2
//File Name:   fig5 21.v

//fig. 5.21       Implementation of state machine on figure 5.21
//APROACH : behavioral
```

```
module fig5-21( Z , state , A , clk , reset);
output Z ;
output [1:0] state;
reg      [1:0] currentstate , state;
reg    Z ;
input    A , clk , reset;
always @ ( posedge clk)
begin
if ( reset == 1) //need to reset to start from a known state at
//some point
currentstate = 0 ;
case (currentstate) //step thru all states per state table
      0:
      if(A == 1)
      begin
     state=1;
     Z =  0;
  end
else
  begin
     state=0;
     Z=1;
  end
1:

 if ( A==1)
   begin
     state=2;
     Z = 0;
   end
 else
   begin
     state=3;
     Z = 0;
   end
2:

if ( A == 1)
 begin
  state 3:
 Z = 1;
 end
 else



 begin
 state=0;
 Z=1;
 end

3:
 if ( A==1)
```

```
   begin
     state = 0;
     Z=1;
   end
else
begin
state=1;
Z=1;
end
 default
   if ( A == 1)
     begin
       state = 2'bxx;
       Z = 1'bx;
     end
else
     begin
       state = 2'bxx ;
       Z = 1'bx;
     end
 endcase
currentstate = state ;            //update state for next time
pass
end
endmodule
module fig5_21_0 test;
reg A , clk, reset;
wire [1:0] state;
wire Z ;
fig5_21            fig5_21_0(Z,state,A,clk,reset);

initial
   $monitor( "Time %0d, state=%b, A= %b, Z= %b, reset= %b",

              $time,    state,    A,     Z,     reset );
      initial
begin
     #0
     A=  1'b0;        //reset to state 0
     reset=1'b1;
     clk =1'b0;
     #20
     clk =1'b1;
     #20
     A=  1'b0;        //Input 1 to go to state 1
     reset=1'b0;
     clk =1'b0;
     #20
     clk =1'b1;
     #20
     A=  1'b0;        //Input 0 to go to state 3
     reset=1'b0;
     clk =1'b0;
```

```
    #20
    clk =1'b1;
    #20
    A=  1'b1;          //Input 1 to go to state 0
    reset=1'b0;
    clk =1'b0;
    #20
    clk =1'b1;
    #20
    A=  1'b0;          //Input 0 to stay at state 0
    reset=1'b0;
    clk =1'b0;
    #20
    clk =1'b1;
    #20
    A=  1'b0;          //Input 1 to go to state 1
    reset=1'b0;
    clk =1'b0;
    #20
    clk =1'b1;
    #20
    A=  1'b1;          //Input 1 to go to state 2
    reset=1'b0;
    clk =1'b0;
    #20
    clk =1'b1;
    #20
    A=  1'b1;          //Input 1 to go to state 3
    reset=1'b0;
    clk =1'b0;
    #20
    clk =1'b1;
    #20
    A=  1'b1;          //Input 1 to go to state 0
    reset=1'b0;

    clk =1'b0;
    #20
    clk =1'b1;
    #20
    A=  1'b1;          //done
    reset=1'b0;
    clk =1'b0;
    #20
    clk =1'b1;
    end
endmodule
```

EXAMPLE I.10

Write a Verilog description for the two-bit counter of example 5.5.

Solution

```
// exercise 5.5
 module counter2bit(clock, reset, state);
    input clock, reset;
    output [1:0] state;
    reg [1:0] state, next_state;
    parameter s00 = 2'b00,
              s01 = 2'b01,
              s10 = 2'b10,
              s11 = 2'b11;

    always @ (posedge clock or posedge reset)
      begin
          if (reset == 1)
             state <= s00;
          else
             state <= next_state;
      end

    always @ (state)
      begin
        case(state)
          s00 :  next_state <= s01;
          s01 :  next_state <= s10;
          s10 :  next_state <= s11;
          s11 :  next_state <= s00;
        endcase
      end
endmodule
  module test;
    reg clock, reset;
    wire [1:0] state;

    counter2bit  c2bit(clock, reset, state);
initial
  begin
$display(" clock     reset\tstate binary \tstate decimal");
   $monitor ( "   %b\t     %b\t   %b\t   %d ",
                    clock, reset,state,state);
  #0 reset = 0;
  #1 reset = 1;
  #1 reset = 0;
  end
   initial
       begin
       #0 clock = 0;
   #40 $finish;
       end
      always #1 clock = ~clock;
 endmodule
```

Note: In the above, inclusion of `\t` with statements for `$display and $monitor` provides horizontal tab.

```
clock     reset     state binary  state decimal
   0         0          xx             x
   1         1          00             0
   0         0          00             0
   1         0          01             1
   0         0          01             1
   1         0          10             2
   0         0          10             2
   1         0          11             3
   0         0          11             3
   1         0          00             0
   0         0          00             0
   1         0          01             1
   0         0          01             1
   1         0          10             2
   0         0          10             2
   1         0          11             3
   0         0          11             3
   1         0          00             0
   0         0          00             0
   1         0          01             1
   0         0          01             1
   1         0          10             2
   0         0          10             2
   1         0          11             3
   0         0          11             3
   1         0          00             0
   0         0          00             0
   1         0          01             1
   0         0          01             1
   1         0          10             2
   0         0          10             2
   1         0          11             3
   0         0          11             3
   1         0          00             0
   0         0          00             0
   1         0          01             1
   0         0          01             1
   1         0          10             2
   0         0          10             2
   1         0          11             3
```

EXAMPLE I.11

Write a Verilog description for the three-bit counter of Example 5.7.

Solution

```
// example 5.7
  module nonbinarycounter(clock, reset, state);
  input clock, reset;
  output [2:0] state;
  reg [2:0] state, next_state;
  parameter s0 = 3'b000, s1 = 3'b001,
            s2 = 3'b010, s3 = 3'b011,
            s4 = 3'b100, s5 = 3'b101,
            s6 = 3'b110, s7 = 3'b111;
  always @ (posedge clock or posedge reset)
    begin
        if (reset == 1)
           state <= s0;
        else
           state <= next_state;
    end

  always @ (state)
    begin
      case(state)
        s0 :  next_state <= s2;
        s1 :  next_state <= s3;
        s2 :  next_state <= s3;
        s3 :  next_state <= s5;
        s4 :  next_state <= s7;
        s5 :  next_state <= s6;
        s6 :  next_state <= s7;
        s7 :  next_state <= s0;
      endcase
    end
endmodule
module test;
  reg clock, reset;
  wire [2:0] state;
  nonbinarycounter nbc(clock, reset, state);
initial
  begin
  $display(" clock     reset\tstate binary \tstate decimal");
   $monitor ( "   %b\t      %b\t    %b\t     %d ",
                  clock,    reset,   state,   state);
  #0 reset = 0;
  #1 reset = 1;
  #1 reset = 0;
  end
   initial
       begin
        #0 clock = 0;
        #40 $finish;
        end
      always #1 clock = ~clock;
```

```
endmodule
```

Note: In the above, inclusion of `\t` with statements for `$display and $monitor` provides horizontal tab.

clock	reset	state binary	state decimal
0	0	xxx	x
1	1	000	0
0	0	000	0
1	0	010	2
0	0	010	2
1	0	011	3
0	0	011	3
1	0	101	5
0	0	101	5
1	0	110	6
0	0	110	6
1	0	111	7
0	0	111	7
1	0	000	0
0	0	000	0
1	0	010	2
0	0	010	2
1	0	011	3
0	0	011	3
1	0	101	5
0	0	101	5
1	0	110	6
0	0	110	6
1	0	111	7
0	0	111	7
1	0	000	0
0	0	000	0
1	0	010	2
0	0	010	2
1	0	011	3
0	0	011	3
1	0	101	5
0	0	101	5
1	0	110	6
0	0	110	6
1	0	111	7
0	0	111	7
1	0	000	0
0	0	000	0
1	0	010	2

EXAMPLE I.12

Write a Verilog description for the General Purpose register of figure 5.41.

Solution

```
/*************************************************************************
****
Description: Basic Cell
File Name: BasicCell.v
*************************************************************************
***/
module BasicCell( q, CLR, CLK, s, A );
output q;
input CLK, CLR;
input [1:0] s;
input [3:0] A;
wire data, q_bar;
mux4to1 M1( data, s, A );
D_ff D0( q, q_bar, CLR, CLK, data );
endmodule

/*************************************************************************
****Description: D Flip Flop
File Name: D.v
*************************************************************************
***/
module D_ff( Q, Q_bar, CLR, CLK, D );
output Q, Q_bar;
input CLR, CLK, D;

reg Q, Q_bar;
always @( posedge CLK or negedge CLR)
begin                           //When CLR == 0 (neg logic) Q is always 0
                                //else @ rising edge of clock, Q <-- D
        if(!CLR)
                begin
                        Q <= 1'b0;
                        Q_bar <= 1'b1;
                end
        else
                begin
                        Q <= D;
                        Q_bar <= !D;
                end
end

endmodule
// The code for the 4 to 1 multiplexer used in the Basic cell is:
// Filename : mux4to1.v
//description: 4 to 1 multiplexer

module mux4to1(X, s, A);
        output X;
        input [1:0] s;
        input [3:0] A;
        assign X =      (s == 2'b00)? A[0]:
                        (s == 2'b01)? A[1]:
                        (s == 2'b10)? A[1]: A[3];
endmodule

//description: General purpose register
```

```
module GPR (Q, CLR, CLK, S, X, r_in, l_in) ;
output [3:0] Q;
input CLR, CLK, r_in, l_in;
input [ 1: 0] S;
input [3:0] X;
wire [3:0] A;
BasicCell Cell3 (A[3] , CLR, CLK, S, {X[3] , A[2] , r_in , A[3]} );
BasicCell Cell2 (A[2] , CLR, CLK, S, {X[2] , A[1] , A[3] , A[2]} ) ;
BasicCell Cell1 (A[1] , CLR, CLK, S, {X[1] , A[0] , A[2] , A[1]} ) ;
BasicCell Cell0 (A[0] , CLR, CLK, S, {X[0] , l_in, A[1] , A[0] } ) ;
assign Q = A;
endmodule
```

I.4 Status register design using Verilog

In this section, the Verilog description of the Status register of Example 6.1 will be provided.

EXAMPLE I.13

Write a Verilog description of the Status register of Figure 6.1.

Solution

VeriLogger Program, Test Bench and Results

```
//  Status Register
module statsreg(stat,cfinal,cprev,clk,r);
input [3:0] r;
input cfinal,cprev,clk;
output [4:0] stat;
reg [4:0] stat;
/*  The status register is 5-bits. They will be latched and the
output is shown at a positive edge of the clock.
*/
  always@(posedge clk)
       begin
            stat[0] <= r[3]^r[2]^r[1]^r[0];       //Parity flag
            stat[1] <= cfinal^cprev;              //Overflow flag
            stat[2] <= ~(r[3]|r[2]|r[1]|r[0]);    //Zero flag
            stat[3] <= r[3];                      //MSB
            stat[4] <= cfinal;                    //Final carry

          end
endmodule
//  The following is a test bench to verify the results of our
module above.
module tbench;
reg [3:0] r_in;
reg cfinal_in,cprev_in,clock;
```

```
wire [4:0] stat_out;

//  module statsreg(stat,cfinal,cprev,clk,r);
  statsreg SReg1(stat_out,cfinal_in,cprev_in,clock,r_in);

    initial
    begin

       $monitor("Time=%0d clock=%b r_in=%b cfinal_in=%b cprev_in=%b
       stat_out=%b", $time,clock,r_in,cfinal_in,cprev_in,stat_out);
    end
    always
          begin
            #1  clock=0;
            #1  clock=1;
          end

    initial
       begin
          #0 r_in=0; cfinal_in=1; cprev_in=1;
          #2
          #3 r_in=6; cfinal_in=1; cprev_in=0;
          #2
          #3 r_in=15; cfinal_in=0; cprev_in=0;
          #2
          #1 $finish;
       end
endmodule
Time=0 clock=x r_in=0000 cfinal_in=1 cprev_in=1 stat_out=xxxxx
Time=1 clock=0 r_in=0000 cfinal_in=1 cprev_in=1 stat_out=xxxxx
Time=2 clock=1 r_in=0000 cfinal_in=1 cprev_in=1 stat_out=10100
Time=3 clock=0 r_in=0110 cfinal_in=1 cprev_in=0 stat_out=10100
Time=4 clock=1 r_in=0110 cfinal_in=1 cprev_in=0 stat_out=10010
Time=5 clock=0 r_in=0110 cfinal_in=1 cprev_in=0 stat_out=10010
Time=6 clock=1 r_in=1111 cfinal_in=0 cprev_in=0 stat_out=01000
Time=7 clock=0 r_in=1111 cfinal_in=0 cprev_in=0 stat_out=01000
Time=8 clock=1 r_in=1111 cfinal_in=0 cprev_in=0 stat_out=01000
```

I.5 CPU design using Verilog

Memory can be modeled in Verilog as an array of registers. The following are some of the typical examples of specifying memory in Verilog:

```
reg addr [0:2047];          // Memory with 2K 1-bit words (Addresses
```

```
                          // addr[0]
                          // through addr[2047]).
reg [15:0] addr [0:4095]; // Memory with 4K 16-bit words (Addresses
                          // addr[0] through addr[4095]).
reg [22:0] mem [52:0];    // Memory of size 53X23 bits (Addresses mem[0]
                          // through mem[52]).
data = mem[loc]           // Memory read operation. Read the contents of a
                          // memory
                          // location addressed by loc into a register
                          // called data.
mem[loc] = data           // Memory write operation. Write the contents of
                          // a register
                          // called data into a memory location addressed
                          // by loc.
```

Example I.14

Write a Verilog description for the ALU of Figure 7.24.

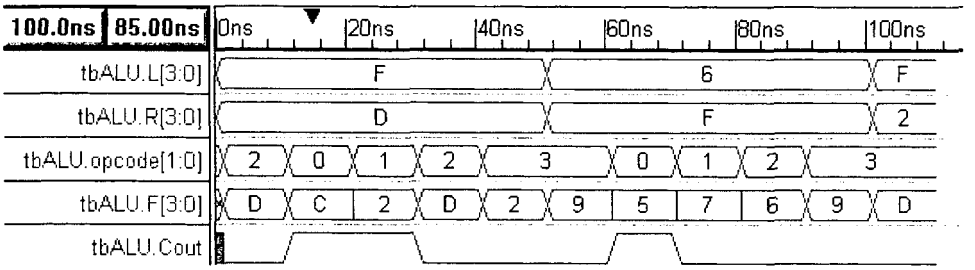
Solution

The verilog coding for 4-bit ripple carry adder is:

```
`include "FA.v"
module Add4(c_out, Sum, A, B, c_in);
//Add 2 4-bit numbers A & B with carry in
//output Sum and c_out
output c_out;
output [3:0] Sum;
input [3:0] A, B;
input c_in;
wire [2:0] carry;

//need 4 full adders

FA fa0(carry[0], Sum[0], A[0], B[0], c_in);
FA fa1(carry[1], Sum[1], A[1], B[1], carry[0]);
FA fa2(carry[2], Sum[2], A[2], B[2], carry[1]);
FA fa3(c_out, Sum[3], A[3], B[3], carry[2]);
endmodule

//The included code for full adder is:

module FA(c_out, sum, a, b, c_in);
//Full Adder
       input  a, b, c_in;
       output sum, c_out;
      assign{c_out, sum} = a + b + c_in;
endmodule
//The coding for multiplexer is:

module mux2to1(x, select, A0, A1);
output x;
input select, A0, A1;
assign x = (select)? A1: A0;
endmodule
```

```
//description: 4-bit ALU
module ALU(F, C_out, X, Y, fCode);
output [3:0] F;
output C_out;
input [3:0] X, Y;
input [1:0] fCode;
wire [3:0] B, Y_not, AU, LU, LU_0, LU_1;
wire carry;

//Structure of Arithmetic unit
//Prep inverted Y
not(Y_not[0], Y[0]);
not(Y_not[1], Y[1]);
not(Y_not[2], Y[2]);
not(Y_not[3], Y[3]);

//Prep input B to adder
mux2to1 B0( B[0], fCode[0], Y[0], Y_not[0]);
mux2to1 B1( B[1], fCode[0], Y[1], Y_not[1]);
mux2to1 B2( B[2], fCode[0], Y[2], Y_not[2]);
mux2to1 B3( B[3], fCode[0], Y[3], Y_not[3]);

//Feed signal to adder
Add4 Adder(carry, AU, X, B, fCode[0]);
//Only when S1 = 0, we need carry
//otherwise carry should be 0
and(C_out, carry, ~fCode[1]);

//Structure of logic unit;
//Input when S0 == 0
and(LU_0[0], X[0], Y[0]);
and(LU_0[1], X[1], Y[1]);
and(LU_0[2], X[2], Y[2]);
and(LU_0[3], X[3], Y[3]);
//Input when S0 == 1
xor(LU_1[0], X[0], Y[0]);
xor(LU_1[1], X[1], Y[1]);
xor(LU_1[2], X[2], Y[2]);
xor(LU_1[3], X[3], Y[3]);

//calc output of logic unit
mux2to1 G0(LU[0], fCode[0], LU_0[0], LU_1[0]);
mux2to1 G1(LU[1], fCode[0], LU_0[1], LU_1[1]);
mux2to1 G2(LU[2], fCode[0], LU_0[2], LU_1[2]);
mux2to1 G3(LU[3], fCode[0], LU_0[3], LU_1[3]);
//Connect arithmethic and logic unit together
mux2to1 F0(F[0], fCode[1], AU[0], LU[0]);
mux2to1 F1(F[1], fCode[1], AU[1], LU[1]);
mux2to1 F2(F[2], fCode[1], AU[2], LU[2]);

mux2to1 F3(F[3], fCode[1], AU[3], LU[3]);
endmodule
```

Waveform:



In the above, when opcode is 2, L is 15, R is 13. A Boolean AND operation between L and R is performed, and the answer is 13 (D_{16} as expected. For opcode 0, operation L plus R is performed generating an answer of 12 with 1 carry out as expected.

Example I.15

Write a Verilog description for the microprogrammed CPU of section 7.4.

Solution

Xlinix ModelSim simulator is used to simulate the Verilog program. A test bench is written to instantiate the CPU module and generate the clock.

Seven modules are created in the Verilog program to implement the microprogrammed CPU. The modules are **memcntrl, reg_8bit, alu_8bit, mux_8bit, ram, processor and cpu**. The design is created using hierarchical method. The **cpu** module is at the top of the hierarchy, **processor** and **memcntrl** are under **cpu** module, and finally the rest of the modules are under the **processor**.

The **memcntrol** contains the ROM, filled with a 23-bit value, which contains a 4-bit condition select, a 6-bit branch address, and 13-bit control input (C12 - C0) for the registers, ALU, and RAM. It also has the conditional statement that will make the Microprogram Counter (MPC) to count up by one if the load/$\overline{\text{increment}}$ is LOW, or will load the branch address passed by the control memory buffer if load/$\overline{\text{increment}}$ is HIGH. The **processor** module connects mux, alu, registers (regA, regIR, regMAR, regPC, regBUFF), and the RAM. It also includes the instruction decoder and performs the following (Figure 7.58) : If condition select field = 0, load/$\overline{\text{increment}}$ = 0, no branch. If condition select = 1 and Z = 1, branch. If condition select = 2 and C =1, branch. If condition select = 3 and I3 = 1, branch. If condition select = 4 and XC2 = 1, branch. If condition select = 5 and XC1 = 1, branch. If condition select = 6 and XC0 = 1, branch. If condition select = 7 and I0 = 1, branch.

The 256 x 8 RAM holds program instructions and data. The program is stored beginning at RAM address 0. This program tests two instructions (LOAD and ADD) of the CPU. The program will first load a value into register A from RAM address 100, add it to itself and store the result in register A.

The CPU module has only two inputs. These are reset and clock. It connects the **processor** module with the **memory control** module to complete the hierarchy of the microporgrammed CPU design.

Verilog code for the microprogrammed CPU is provided in the following:

```
// Microprogrammed Controller Module for the CPU
// Port declarations
```

```
module memcntrl (C_fn, Z, C, I3, XC2, XC1, XC0,I0, reset, clk);
input Z, C, I3, XC2, XC1, XC0, I0, reset, clk;
output [12:0] C_fn;
reg [22:0] mem [52:0];
reg [12:0] C_fn;
reg [22:0] regCMDB;
reg [5:0] regMPC;
reg ld_inc;
// Binary microprogram
// The size of the control memory is 53 x 23 bits.  The 23-bit
// control word consists of 13-bit control function containing C0
// through C12 with C0 as bit 12 and C12 as bit 0. The condition
// select field is 4-bit wide (bits 19-22). For example, consider
// the code for line 0 with the operation PC <- 0 in the
// following.  Since there is no condition in this operation,
// condition select field ( CS ) bits are 0's. The branch address
// field ( Brn )bits are assumed as don't cares arbitrarily.  To
// clear PC to 0, C0 = 1 (bit 12). To disable RAM,  C6 = 1.  C1,
// C2, C4, C7, C8 and C9 are initialized to 0's.  Other bits are
// arbitrarily initialized as don't cares.
initial
begin

// 23-bit value contains a 4-bit condition select, a 6-bit branch
// address, and 13-bit control. input ( C12 - C0 ) for the
// registers, ALU, and RAM.
//          22  19     12          0
//           cs    Brn   Cntrl Func
mem[0]  = 23'b0000xxxxxx100x0x1000xxx;
mem[1]  = 23'b0000xxxxxx00001x1000xxx;
mem[2]  = 23'b0000xxxxxx010x010010xxx;
mem[3]  = 23'b0011001110000x0x1000xxx;
mem[4]  = 23'b0110001000000x0x1000xxx;
mem[5]  = 23'b0101001010000x0x1000xxx;
mem[6]  = 23'b0100001100000x0x1000xxx;
mem[7]  = 23'b1000110100000x0x1000xxx;
mem[8]  = 23'b0000xxxxxx000x0x1001111;
mem[9]  = 23'b1000000001000x0x1000xxx;
mem[10] = 23'b0000xxxxxx000x0x1001100;
mem[11] = 23'b1000000001000x0x1000xxx;
mem[12] = 23'b0000xxxxxx000x0x1001101;
mem[13] = 23'b1000000001000x0x1000xxx;
mem[14] = 23'b0110010111000x0x1000xxx;
mem[15] = 23'b0101100000000x0x1000xxx;
mem[16] = 23'b0100101001000x0x1000xxx;
mem[17] = 23'b0000xxxxxx00001x1000xxx;
mem[18] = 23'b0000xxxxxx010x010100xxx;
mem[19] = 23'b0000xxxxxx00011x1000xxx;
mem[20] = 23'b0000xxxxxx000x010100xxx;
mem[21] = 23'b0000xxxxxx000x0x1001110;
mem[22] = 23'b1000000001000x0x1000xxx;
mem[23] = 23'b0000xxxxxx00001x1000xxx;
mem[24] = 23'b0000xxxxxx010x010100xxx;
```

```
mem[25] = 23'b0000xxxxxx00011x1000xxx;
mem[26] = 23'b0111011110000x0x1000xxx;
mem[27] = 23'b0000xxxxxx000x010100xxx;
mem[28] = 23'b0000xxxxxx000x0x1001001;
mem[29] = 23'b1000000001000x0x1000xxx;
mem[30] = 23'b0000xxxxxx000x000000xxx;
mem[31] = 23'b1000000001000x0x1000xxx;
mem[32] = 23'b0000xxxxxx00001x1000xxx;
mem[33] = 23'b0000xxxxxx010x010100xxx;
mem[34] = 23'b0000xxxxxx00011x1000xxx;
mem[35] = 23'b0000xxxxxx000x010100xxx;
mem[36] = 23'b0111100111000x0x1000xxx;
mem[37] = 23'b0000xxxxxx000x0x1001010;
mem[38] = 23'b1000000001000x0x1000xxx;
mem[39] = 23'b0000xxxxxx000x0x1001011;
mem[40] = 23'b1000000001000x0x1000xxx;
mem[41] = 23'b0000xxxxxx00001x1000xxx;
mem[42] = 23'b0000xxxxxx000x0x1000xxx;
mem[43] = 23'b0111101111000x110000xxx;
mem[44] = 23'b0001110010000x0x1000xxx;
mem[45] = 23'b0000xxxxxx010x0x1000xxx;
mem[46] = 23'b1000000001000x0x1000xxx;
mem[47] = 23'b0010110010000x0x1000xxx;
mem[48] = 23'b1000000001000x0x1000xxx;
mem[49] = 23'b0000xxxxxx010x0x1000xxx;
mem[50] = 23'b0000xxxxxx001x010000xxx;
mem[51] = 23'b1000000001000x0x1000xxx;
mem[52] = 23'b1000110100000x0x1000xxx;
end
always @( reset )
       if ( reset )
              begin  // when reset is active and reset is high
               regMPC = 6'b000000;// initialize MPC to zero
              end

//conditional statement that will make the Microprogram Counter
//(MPC) to count up by one if the load/increment is low, or will
//load the branch address passed by the control memory buffer.

always @ ( posedge clk )    // when clock is at positive edge
       begin
              regCMDB = mem[regMPC];
// register regCMDB contains 23-bit contents of memory addressed
// by regMPC
              C_fn = regCMDB [12:0];
       // control function equals to first 13 bits of  register CMDB

       // if condition select field = 0, load /increment = 0, no
       // branch.
       // if condition select = 1 and Z = 1, branch
       // if condition select = 2 and C =1, branch
       // if condition select = 3 and I3 = 1, branch
       // if condition select = 4 and XC2 = 1, branch
```

```
      // if condition select = 5 and XC1 = 1, branch
      // if condition select = 6 and XC0 = 1, branch
      // if condition select = 7 and I0 = 1, branch
      // if condition select = 8 and load /increment= 1, branch
            assign ld_inc =
      ( regCMDB [22:19] == 0 )?1'b0:  // if cmdb= 0 ld_inc = 0
      ( regCMDB [22:19] == 1 )?Z:     // if cmdb= 1 ld_inc = Z
      ( regCMDB [22:19] == 2 )?C:     // if cmdb= 2 ld_inc = C
      ( regCMDB [22:19] == 3 )?I3:    // if cmdb= 3 ld_inc = I3
      ( regCMDB [22:19] == 4 )?XC2:   // if cmdb= 4 ld_inc = XC2
      ( regCMDB [22:19] == 5 )?XC1:   // if cmdb= 5 ld_inc =XC1
      ( regCMDB [22:19] == 6 )?XC0:   // if cmdb= 6 ld_inc = XC0
      ( regCMDB [22:19] == 7 )?I0:    // if cmdb= 7 ld_inc = I0
      ( regCMDB [22:19] == 8 )?1'b1:  // if cmdb= 8 ld_inc = 1
       1'bx;                             // else       ld_inc = x
       if (ld_inc)
            regMPC = regCMDB [18:13]; // load branch address
       else
            regMPC = regMPC + 1;      // increment MPC by 1
      end
endmodule
```

//Register 8 bit module

```
// General Purpose Register (GPR)
module reg_8bit (b, a, sel, clk);
input [7:0] a;
input [2:0] sel;
input clk;
output [7:0] b;
reg [7:0] b;
      always @ (sel)
            begin
                  b <=   (sel==0)?b:     // b = b if sel = 0
                         (sel==1)?0 :    // b= 0 if sel = 1
                         (sel==2)?b+1 :  // b= b+1 if sel = 2
                         (sel==4)?a:     // b= a if sel = 4
                         8'bx;           // else b=xxxxxxxx
            end
endmodule
```

//ALU module

```
// ALU with zero and carry flags
module alu_8bit ( f, z_flag, c_flag, a, b, sel);
input [2:0] sel;
input [7:0] a, b;
output [7:0] f;
output z_flag, c_flag;
reg z_flag, c_flag;
   initial
      begin
        z_flag = 1'b0;  // initialize zero and carry flag to zero
            c_flag = 1'b0;   //
        end
```

```
        assign f =(sel==0)?0 :      // f=0 if sel=0
                  (sel==1)?b:       // f=b if sel=1
                  (sel==2)?a+b:     // f=a+b if sel=2
                  (sel==3)?a-b:     // f=a-b if sel=3
                  (sel==4)?a+1 : // f=a+1 if sel=4
                  (sel==5)?a-1 ://f=a-1 if sel=5
                  (sel==6)?a&b://f=a&b if sel=6
                  (sel==7)?~a://f=~a if sel=7
                  8'bx;       // else f=xxxxxxxx
//Carry and Zero Flag registers
  always @ ( f )
      begin
            if (f==0)     // if alu output = 0, zero flag = 1
                  assign z_flag =1;
         else if ( f != 0 & ( sel != 3'bxxx ))  // if f not zero
                                                // and
                                                //  sel not xxx
               assign z_flag = 0;               //  zero flag = 0

            end

     always@ ( f )

      begin
        if(sel==4 | sel==2)
                  carry = (a[7]+b[7])*f[7]+a[7]*b[7];
        if ( carry )      // if alu outputs carry, carry flag = 1
        assign c_flag = 1;
        else if ( !carry & ( sel != 3'bxxx )) // if not carry and
       assign c_flag = 0;  // sel not xxx, carry = 0
    end
endmodule
```

//Processor module (Figures 7.53 and 7.56)

```
// Processor

module processor (I3, XC0, XC1, XC2, XC3, I0, z_flag, c_flag, clock,
c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12);
input clock;
input c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12;
output I3, XC0, XC1, XC2, XC3, I0, z_flag, c_flag;
wire [7:0] IR_out;
wire [7:0] F_out, BUFF_out, RAM_dataout, RAM_addr, MAR_in, PC_out;
reg [7:0] regA_out;
reg I0, I3, XC0, XC1, XC2, XC3;

//module mux_8bit(z, sel, mux_in0, mux_in1);
```

```
mux_8bit Mux1(MAR_in, c3, PC_out, BUFF_out);

//module alu_8bit(f, z_flag, c_flag, a, b, sel);
alu_8bit ALU1(F_out, z_flag, c_flag, regA_out, BUFF_out, {c10, c11,
c12});

//module reg_8bit(b, a, sel, clk);
//reg_8bit regA(regA_in, F_out, {c9, 1'b0, 1'b0}, clock);
reg_8bit regIR(IR_out, RAM_dataout, {c8, 1'b0, 1'b0}, clock);
reg_8bit regMAR(RAM_addr, MAR_in, {c4, 1'b0, 1'b0}, clock);
reg_8bit regPC(PC_out, RAM_dataout, {c2, c1, c0}, clock);
reg_8bit regBUFF(BUFF_out, RAM_dataout, {c7, 1'b0, 1'b0}, clock);

//module ram(dataout, memeaddr, datain, rw, en);
ram RAM1(RAM_dataout, RAM_addr, regA_out, c5, c6);
initial
begin
       XC0 <= 0;     //initialize control signals to zero
       XC1 <= 0;
       XC2 <= 0;
       XC3 <= 0;
       I0  <= 0;
       I3  <= 0;
end

always@(clock)
begin

I3 <= IR_out[3];     // instruction decoder
I0 <= IR_out[0];     // I3= irout[3]  ,  I0 = irout[0]

case ( {IR_out[2], IR_out[1]} )

2'd0:begin XC0 =1; XC1 =0; XC2 = 0; end //if irout[2:1]=0,XC0=1,
                                        //others zero
2'd1:begin XC1 =1; XC0 =0; XC2 = 0; end // if irout[2:1]=1,XC1=1,
                                       //others zero
2'd2:begin XC2 =1; XC0 =0; XC1 =0; end // if irout[2:1]=2,XC2=1,
                                      //others zero
2'd3:begin XC3 =1; XC0 =0; XC1=0; XC2= 0; end//if irout[2:1]=3,
                                      //XC3=1, others 0
 default:
begin XC0 =1'bx; XC1 = 1'bx; XC2 = 1'bx; XC3 =1'bx; end // else
                                        //everything x
endcase
end
```

```
      always @ (posedge clock)
            begin

            regA_out <=   (c9==0)?regA_out:  // if c9=0 , regA_
out= regA_out
                          (c9==1)?F_out:     // if c9 =1, regA_out
= F_out
                          8'bx;               // else regA_out=
xxxxxxxx
            end
endmodule
//Mux 8 bit module
module mux_8bit (z, sel, mux_in0, mux_in1);

input        sel;
input  [7:0] mux_in0, mux_in1;
output [7:0] z;

//     The output is defined as register
reg    [7:0] z;


// The output changes whenever any of the inputs changes
always @(sel or mux_in0 or mux_in1)
            // Check the control signal
            case (sel)
            1'b0:
                  z = mux_in0;   // if sel= 0 , z = in0
            1'b1:
                  z = mux_in1;  // if sel=1, z = in 1
            endcase
endmodule

//256 x 8 Ram
module ram ( dataout, memaddr, datain, rw, en );
//--------------Input Ports-----------------------
input [7:0] memaddr;
input [7:0] datain;
input rw, en;
output [7:0] dataout;
//--------------Internal variables----------------
reg [7:0] dataout ;
reg [7:0] mem [0:255];
//--------------Code Starts Here------------------
initial
mem[0] = 8'b00001000;      // LDA mem <addr>
mem[1] = 100;       // <addr> = 100, A<-5
mem[2] = 8'b00001010;      // ADD A <- A + MEM<addr>
mem[3] = 100;       // <addr> = 100, A<-10
mem[100] = 8'b00000101;    // init data = 5
always @ (memaddr or datain or rw)
begin : MEM_WRITE
   if ( !en && !rw )
```

```
        mem[memaddr] = datain;
end
always @ (memaddr or rw or en)
begin : MEM_READ
    if (!en && rw )
         dataout = mem[memaddr];
end
endmodule
```

//CPU module has only two inputs (system clock and system reset)

```
module cpu ( clock, reset );
input clock, reset;
wire xc2, xc1, xc0, i3, i0, z, c;
wire [12:0] cfn;
processor p1(.clock(clock), .XC2(xc2), .XC1(xc1), .XC0(xc0),
.I3(i3),
.I0(i0), .z_flag(z), .c_flag(c), .c0(cfn[12]), .c1(cfn[11]),
.c2(cfn[10]),
.c3(cfn[9]), .c4(cfn[8]), .c5(cfn[7]), .c6(cfn[6]), .c7(cfn[5]),
.c8(cfn[4]), .c9(cfn[3]), .c10(cfn[2]), .c11(cfn[1]), .c12(cfn[0])
);
memcntrl memc(.clk(clock), .reset(reset), .XC2(xc2), .XC1(xc1),
.XC0(xc0), .I3(i3), .I0(i0), .Z(z), .C(c), .C_fn(cfn));
endmodule
```

//Test Bench for CPU module

```
module test_cpu;
reg clock, rst;
cpu dut (clock, rst);
initial // Clock generator
  begin                    // generating clock with period of 2ns
    clock = 0;
    #1001 forever
    #1000 clock = !clock;
  end
initial      // Test stimulus
  begin
    rst = 1;             // reset goes high for 3.5 ns then goes
low
    #3500 rst = 0;
  end
 endmodule
```

Timing Diagram

All eleven instructions are tested successfully by simulating a sample program. Timing diagrams are generated accordingly. The following simple program inside the 256 x 8 RAM is simulated for testing the proper operation of two (LDA,ADD) of the eleven instructions. The timing diagram of Figure I.1 is generated. Note that PC is the program counter for the sample program in the RAM, and MPC is the microprogram counter for the symbolic program in the ROM (Figure 7.57) inside the memory control module.

Program for testing LDA and ADD:

```
mem[0] = LDA                    // A<- MEM <addr>
mem[1] = 100;                  // <addr> = 100, A<-5
mem[2] = ADD                  // A <- A + MEM<addr>
mem[3] = 100;                // <addr> = 100,A<-10
mem[100] = 8'b00000101;     // init data = 5
```

LDA (PC=0) instruction with reference address 100, goes through the subroutines in the symbolic program (Figure 7.57): FETCH (MPC=1 at t=2ns), branching to MEMREF(MPC=14 at t=8ns), then to LDSTO(MPC=23 at t=10ns), all the way through LOAD (MPC = 27 at t=18ns), and back to FETCH. At t=23ns, register A holds 05H, showing that it has loaded the contents of RAM memory address 100 (See figure J.1). Next, ADD (PC=2) operation is performed using reference address 100. At this point, ADD goes through the following subroutines in the symbolic program: FETCH (MPC=1 at t=24ns), branching to MEMREF(MPC=14 at t=30ns), then to ADDSUB(MPC=32 at t=34ns), all the way through ADD(MPC=37 at t=44ns), then back to FETCH (See figure J.1). At t=46ns, register A and BUFFER hold the contents of memory address 100. They are now the inputs to the ALU. The ALU will add these two values and its output will then go to register A, as commanded by the ADD<addr> instruction. At t=47ns, one can see that the contents of register A have changed to 0AH (10_{10}) (See figure I.1).

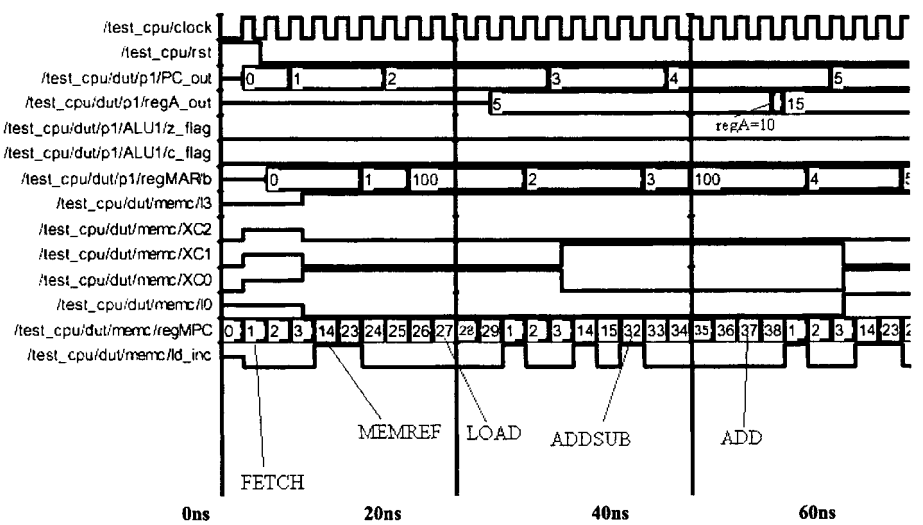


Figure I.1 Verilog Timing Diagram (Top diagram-CPU clock, Next-Reset, Next-PC, Next-reg A, Next-Zflag, Next-Cflag, Next-regMAR, Next-I3, Next-XC2, Next-XC1, Next-XC0, Next-I0, Next-mpc, Next-ld_inc)

QUESTIONS AND PROBLEMS

I.1 Write a Verilog description for each of the following:

(a) a 2-to-4 decoder using dataflow modeling , generating a low output when selected by a high enable.

(b) a 3-to-8 decoder using modeling description of your choice, generating a high output when selected by a high enable.

(c) the 4 -to-16 decoder of Problem 4.15 using modeling description of your

choice.

(d) a 4-to-1 multiplexer using conditional operator.

(e) a BCD to seven-segment converter for a common cathode display using behavioral modeling.

(f) the 2-bit unsigned comparator of Section 4.5.2.

I.2 Write a Verilog description for:

(a) the transparent latch of Section 5.2.3.

(b) the gated D flip-flop of Figure 5.5a.

(c) a D flip-flop with a synchronous reset input and a positive edge triggered clock. Use synchronous reset such that if reset ==0, the flip-flop is cleared to 0; on the other hand, if reset==1, the output of the flip-flop is unchanged until the procedural statements are evaluated at the positive edge of the clock.

(d) the T flip-flop (using D-ff and XOR gate) of Problem 5.13(b).

(e) the state machine of Problem 5.19.

(f) a 4-bit binary ripple counter. Note that in a binary ripple counter, the clock inputs of high order flip-flops are not triggered by the common clock, but by the transition outputs of the low order flip-flops. The 4-bit binary ripple counter contains four T flip-flops (obtained from D-ffs), with the output of each ff connected to the clock input of the next higher-order ff. The clock input is connected to the least significant T-ff. The 4-bit ripple counter can be designed using four T flip-flops (tff0 through tff3). Each T-ff can be obtained from a D-ff by connecting its output q to the input of an inverter, and then connecting the inverter output to the D input; the T-ff has one input (T input is the same as the clock input). This T-ff toggles every clock. The 4-bit ripple counter can be obtained by connecting the clock to the tff0 clock input, q0 of tff0 to clock input of tff1, q1 output of tff1 to clock input of tff2, and q2 output of tff2 to the clock input of tff3. Use negative edge-triggered D-ffs. Each D-ff will have a reset input to clear the ff.

(g) a 4-bit serial shift (right) register with a positive edge triggered reset and a positive edge triggered clock. The 4-bit serial shift register can be obtained by connecting four D-ff's to a common clock and a common reset. The four D-ff's are cleared to 0 at the positive edge triggered clock and positive edge triggered reset. Assume, v as the serial input bit connected to the D input of the leftmost D-ff with z as its output; z is connected to the D input of the next right D-ff with y as its output; y is connected to the D input of the next right D-ff with x as its output; finally, x is connected to the D input of the rightmost D-ff with w as its output.

(h) a 4-bit register with a reset input, a parallel load input and a positive edge-triggered clock. The 4-bit register is cleared to 0 at the positive edge of the reset. On the other hand, if the load input is high, 4-bit data is transferred to the register at the positive edge of the clock. Use behavioral modeling.

(i) the counters of Problems 5.24(a) through 5.24(c).

(j) the general purpose register of Problem 5.25.

I.3 Write a Verilog description for the Status register of Example 6.1 using structural modeling.

I.4 Write a Verilog description for the four-bit by four-bit unsigned multiplier (repeated addition) using:
(a) Hardwired control (Section 7.3.5). (b) Microprogramming (Section 7.3.5).