CHAPTER 8

Additional Topics in Verilog

Up to this point, we have described the basic features of Verilog and how they can be used in the digital system design process. In this chapter, we describe additional features of Verilog that illustrate its power and flexibility. Verilog functions and tasks are presented. Several additional features such as user-defined primitives, generate statements, compiler directives, built-in primitives, file I/O, and others are presented. A simple memory model is presented to illustrate the use of tristate signals.

8.1 Verilog Functions

A key feature of VLSI circuits is the repeated use of similar structures. Verilog provides functions and tasks to easily express repeated invocation of the same functionality or the repeated use of structures. These functions are described in this section. A Verilog function is similar to a Verilog task, which will be described later. They have small differences: a function cannot drive more than one output, nor can a function contain delays.
The functions can be defined in the module that they will be used in. The functions can also be defined in separate files, and the compile directive **`include** will be used to include the function in the file (compiler directives are explained in Section 8.12). The functions should be executed in zero time delay, which means that the functions cannot include timing delay information. The functions can have any number of inputs but can return only a single output.

A function returns a value by assigning the value to the function name. The function's return value can be a single bit or multiple bits. The variables declared within the function are local variables, but the functions can also use global variables when no local variables are used. When local variables are used, basically output is assigned only at the end of function execution. The functions can call other functions but cannot call tasks. Functions are frequently used to do type conversions.

A function executes a sequential algorithm and returns a single value to the calling program. When the following function is called, it returns a value equal to the input rotated one position to the right:

```
function [7:0] rotate_right;
  input [7:0] reg_address;

  begin
    rotate_right = reg_address >> 1;
  end
endfunction
```

A function call can be used anywhere that an expression can be used. For example, if $A = 10010101$, the statement

```
B <= rotate_right(A);
```

would set B equal to 11001010 and leave A unchanged.

The general form of a function declaration is

```
function <range or type> function-name
  input [declarations]
<declarations>    // reg, parameter, integer, etc.

  begin
  sequential statements
  end

endfunction
```

The general form of a function call is

```
function_name(input-argument-list)
```

The number and type of parameters on the `input-argument-list` must match the input [declaration] in the function declaration. The parameters are treated as input values and cannot be changed during the execution of the function.

Example Write a Verilog function for generating an even parity bit for a 4-bit number. The input is a 4-bit number and the output is a code word that contains the data and the parity bit.

The answer is shown in Figure 8-1. The function name and the final output to be returned from the function must be the same.

FIGURE 8-1: Parity Generation Using a Function

```
// Function example code without a loop
// This function takes a 4-bit vector
// It returns a 5-bit code with even parity

function [4:0] parity;
  input [3:0] A;
  reg temp_parity;

  begin
    temp_parity = A[0] ^ A[1] ^ A[2] ^ A[3];
    parity = {A, temp_parity};
  end
endfunction
```

If parity circuits are used in several parts in a system, one could call the function each time it is desired. The function can be called as follows:

```
module function_test(Z);
  output reg [4:0] Z;
  reg [3:0] INP;
  initial
  begin
    INP = 4'b0101;
    Z = parity(INP);
  end

endmodule
```

Figure 8-2 illustrates a function using a **for** loop. In Figure 8-2, the loop index *i* will be initialized to 0 when the **for** loop is entered, and the sequential statements will be executed. Execution will be repeated for $i = 1$, $i = 2$, and $i = 3$; then the loop will terminate.

FIGURE 8-2: Add Function

```
// This function adds two 4-bit vectors and a carry.
// Illustrates function creation and use of loop.
// It returns a 5-bit sum.

function [4:0] add4;
  input [3:0] A;
  input [3:0] B;
  input       cin;

  reg   [4:0] sum;
  reg         cout;

  begin
    integer i;
    for (i=0; i<=3; i=i+1)
    begin
      cout = (A[i] & B[i]) | (A[i] & cin) | (B[i] & cin);
      sum[i] = A[i] ^ B[i] ^ cin;
      cin = cout;
  end

   sum[4] = cout;
   add4 = sum;
  end
endfunction
```

The function given in Figure 8-2 adds two 4-bit vectors plus a carry and returns a 5-bit vector as the sum. The function name is *add4*; the input arguments are *A*, *B*, and *carry*; and the add4 will be returned. The local variables C_{out} and *sum* are defined to hold intermediate values during the calculation. When the function is called, C_{in} will be initialized to the value of the carry. The **for** loop adds the bits of *A* and *B* serially in the same manner as a serial adder.

The first time through the loop, C_{out} and *sum[0]* are computed using *A[0]*, *B[0]*, and C_{in}. Then the C_{in} value is updated to the new C_{out} value, and execution of the loop is repeated. During the second time through the loop, C_{out} and *sum[1]* are computed using *A[1]*, *B[1]*, and the new C_{in}. After four times through the loop, all values of *sum[i]* have been computed and *sum* is returned. The function call is of the form

```
add4(A, B, carry)
```

A and *B* may be replaced with any expressions that evaluate to a return value with dimensions 3 `downto` 0, and *carry* may be replaced with any expression that evaluates to a bit. For example, the statement

```
Z <= add4(X, ~Y, 1);
```

calls the function *add4*. Parameters *A*, *B*, and *carry* are set equal to the values of *X*, ~*Y*, and 1, respectively. *X* and *Y* must be multiple bits dimensioned 3:0. The function computes

$$Sum = A + B + carry = X + {\sim}Y + 1$$

and returns this value. Since ~$Y + 1$ equals the 2's complement of *Y*, the computation is equivalent to subtracting by adding the 2's complement. If we ignore the carry stored in $Z[4]$, the result is $Z[3:0] = X - Y$.

Figure 8-3 illustrates the function to compute square of numbers. The function as well as the function call is illustrated. In the illustrated call to the function, the number is 4 bits wide.

FIGURE 8-3: A Function to Compute Squares and Its Call

```
module test_squares (CLK);

   input CLK;
   reg[3:0] FN;
   reg[7:0] answer;

   function [7:0] squares;
      input[3:0] Number;

      begin
        squares = Number * Number;
      end
   endfunction

   initial
   begin
```

```
      FN = 4'b0011;
   end

   always @(posedge CLK)
   begin
     answer = squares(FN);
   end
endmodule
```

A function executes in zero simulation time. Functions must not contain any timing control statements or delay events. Functions must have at least one input argument. Functions cannot have **output** or **inout** arguments.

Functions can be recursive. Recursive functions must be declared as **automatic**, which causes distinct memory to be allocated each time a function is called. Otherwise, if a function is called recursively, each call will overwrite the memory allocated in the previous call.

8.2 Verilog Tasks

Tasks facilitate decomposition of Verilog code into modules. Unlike functions, which return only a single value implicitly with the function name, tasks can return any number of values. The form of a task declaration is

```
task task_name
  input [declarations]
  output [declarations]

 <declarations>     // reg, parameter, integer, etc.

begin
  sequential statements
end task_name;
```

The `formal-parameter-list` specifies the inputs and outputs to the task and their types. A task call is a sequential or concurrent statement of the form

```
task_name(actual-parameter-list);
```

Unlike functions, a task can be executed in non-zero simulation time. Tasks may contain delay, event, or timing control statements. Tasks can have zero or more arguments of type input, output, or inout. Tasks do not return with a value, but they can pass multiple values through **output** and **inout** arguments.

As an example, we will write a task *Addvec*, which will add two *4*-bit data and a carry and return a *4*-bit sum and a carry. We will use a task call of the form

```
Addvec(A, B, Cin, Sum, Cout);
```

where *A*, *B*, and *Sum* are *4*-bit data and C_{in} and C_{out} are 1-bit data.

Figure 8-4 gives the task definition. *Add1*, *Add2*, and C_{in} are input parameters, and *sum* and C_{out} are output parameters. The addition algorithm is essentially the

FIGURE 8-4: Task for Adding Multiple Bits

```
// This task adds two 4-bit data and a carry and
// returns an n-bit sum and a carry. Add1 and Add2 are assumed
// to be of the same length and dimensioned 3 downto 0.

    task Addvec;
      input [3:0] Add1;
      input [3:0] Add2;
      input Cin;
      output [3:0] sum;
      output cout;

      reg C;

      begin
         C = Cin;
         integer i;
         for(i = 0; i <= 4; i = i + 1)
            begin
               sum[i] = Add1[i] ^ Add2[i] ^ C ;
               C = (Add1[i] & Add2[i]) | (Add1[i] & C) | (Add2[i] & C);
            end
         cout = C ;
      end
    endtask
```

same as the one used in the *add4* function. *C* must be a variable, since the new value of *C* is needed each time through the loop; hence it is declared as **reg**. After *4* times through the loop, all 4 bits of *sum* have been computed. It is desirable not to mix blocking and non-blocking statements in tasks.

The tasks can be defined in the module that the functions will be used in. The tasks can also be defined in separate files, and the compile directive '**include should be used (')** (described in Section 8.12) will be used to include the task in the file. The variables declared within the task are local to the task. When no local variables are used, global variables will be used for input and output. When only local variables are used within the task, the variables from the last execution within the task will be passed to the caller.

Functions:
At least one input arguments, but no output or inout arguments
Returns a single value by assigning the value to the function name
Can call other functions, but cannot call tasks
Cannot embed delays, wait statements or any time-controlled statement
Executes in zero time
Can be recursive
Cannot contain non-blocking assignment or procedural continuous assignments

Tasks:
Any number of input, output or inout arguments
Outputs need not use task name
Can call other functions or tasks
May contain time-controlled statements
Task can recursively call itself

8.3 Multivalued Logic and Signal Resolution

In previous chapters, we have mostly used two-valued bit logic in our Verilog code. In order to represent tristate buffers and buses, it is necessary to be able to represent a third value, *Z*, which represents the high-impedance state. It is also at times necessary to have a fourth value, *X*, to represent an unknown state. This unknown state may occur if the initial value of a signal is unknown, or if a signal is simultaneously driven to two conflicting values, such as 0 and 1. If the input to a gate is *Z*, the gate output may assume an unknown value, *X*.

8.3.1 A 4-Valued Logic System

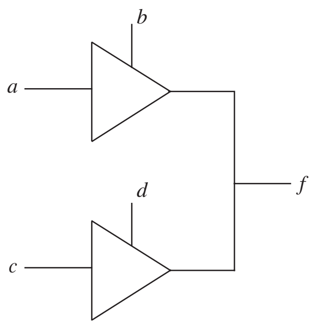
Signals in a 4-valued logic can assume the four values: *X*, 0, 1, and *Z*, where each of the symbols represent the following:

• *X*	Unknown
• 0	0
• 1	1
• *Z*	High impedance

The high impedance state is used for modeling tristate buffers and buses. This unknown state can be used if the initial value of a signal is unknown, or if a signal is simultaneously driven to two conflicting values, such as 0 and 1. Verilog uses the 4-valued logic system by default.

Let us model tristate buffers using the 4-valued logic. Figure 8-5 shows two tristate buffers with their outputs tied together, and Figure 8-6 shows the

FIGURE 8-5: Tristate Buffers with Active-High Output Enable



corresponding Verilog representation in two different ways. Data type X01Z, which can have the four values X, 0, 1, and Z, is assumed. The tristate buffers have an active-high output enable, so that when $b = 1$ and $d = 0$, $f = a$; when $b = 0$ and $d = 1$, $f = c$; and when $b = d = 0$, the f output assumes the high-Z state. If $b = d = 1$, an output conflict can occur. Tristate buffers are used for bus management whereby only one active-high output should be enabled to avoid the output conflict.

FIGURE 8-6: Verilog Code for Tristated Buffers

```
module t_buff_exmpl (a, b, c, d, f);

   input a;
   input b;
   input c;
   input d;
   output f;
   reg f;

   always @(a or b)
   begin : buff1
      if (b == 1'b1)
         f = a ;
      else
         f = 1'bz ; //"drive" the output high Z when not enabled
   end

   always @(c or d)
   begin : buff2
      if (d == 1'b1)
         f = c ;
      else
         f = 1'bz ; //"drive" the output high Z when not enabled
   end

endmodule
```

(a) Tristate module with always statements

```
module t_buff_exmpl2 (a, b, c, d, f);
  input a;
   input b;
   input c;
   input d;
   output f;

   assign f = b ? a: 1'bz ;
   assign f = d ? c: 1'bz ;

endmodule
```

(b) Tristate module with assign statements

The operation of a tristate bus with the 4-valued logic, is specified by the following table:

	X	0	1	*Z*
X	*X*	*X*	*X*	*X*
0	*X*	0	*X*	0
1	*X*	*X*	1	1
Z	*X*	0	1	*Z*

This table gives the resolved value of a signal for each pair of input values: *Z* resolved with any value returns that value, *X* resolved with any value returns *X*, and 0 resolved with 1 returns *X*. If individual wires `s(0), s(1), s(2)` were to change as indicated in the following table at the times indicated, signal R in the last column shows the result of the resolution

Time	s(0)	s(1)	s(2)	R
0	*Z*	*Z*	*Z*	*Z*
2	0	*Z*	*Z*	0
4	0	1	*Z*	*X*
6	*Z*	1	*Z*	1
8	*Z*	1	1	1
10	*Z*	1	0	*X*

AND and OR functions for the 4-valued logic may be defined using the following tables:

AND	*X*	0	1	*Z*
X	*X*	0	*X*	*X*
0	0	0	0	0
1	*X*	0	1	*X*
Z	*X*	0	*X*	*X*

OR	*X*	0	1	*Z*
X	*X*	*X*	1	*X*
0	*X*	0	1	*X*
1	1	1	1	1
Z	*X*	*X*	1	*X*

The first table corresponds to the way an AND gate with 4-valued inputs would work. If one of the AND gate inputs is 0, the output is always 0. If both inputs are 1, the output is 1. In all other cases, the output is unknown (*X*), since a high-Z gate input may act like either a 0 or a 1. For an OR gate, if one of the inputs is 1, the output is always 1. If both inputs are 0, the output is 0. In all other cases, the output is *X*.

8.4 Built-in Primitives

The focus of this book is on behavioral-level modeling, and hence we focused on Verilog constructs that allow that. However, Verilog allows modeling at switch-level details and has several predefined primitives for that. It also has predefined gate-level primitives with drive strength among other things. There are 14 predefined logic gate primitives and 12 predefined switch primitives to provide the gate- and switch-level modeling facility. Modeling at the gate and switch level has several advantages:

(i) Synthesis tools can easily map it to the desired circuitry since gates provide a very close one-to-one mapping between the intended circuit and the model.

(ii) There are primitives such as the bidirectional transfer gate that cannot be otherwise modeled using continuous assignments.

The Verilog module in Figure 2-7 is defined in Figure 8-7 using built-in primitives. The **and** and **or** are built-in primitives with one output and multiple inputs. The output terminal must be listed first followed by inputs as in

```
and (out, in_1, in_2, ..., in_n);
```

FIGURE 8-7: Verilog Gate Module Using Built-In Primitives

```
module two_gates (A, B, D, E);       // Figure 2-7 shows the same module using
   output E;                         // concurrent statements instead of
   input A, B, D;                    // built-in primitives

   wire C;

   or    (E,C,D);                    // output port first followed by inputs
   and   (C,A,B);                    // output port first

endmodule
```

Verilog also provides built-in primitives for tristate gates. The **bufif0** is a non-inverting buffer primitive with active-low control input while **bufif1** has active-high control input. The **notif0** and **notif1** are inverting buffers with active-low and active-high controls, respectively. These primitives can support multiple outputs. The outputs must be listed first followed by input and finally the tristate control input. An array of four inverting tristate buffers can be created as shown in Figure 8-8.

FIGURE 8-8: Verilog Array of Tristate Buffers Using Built-In Primitives

```
module tri_driver (in, out, tri_en);
   input [3:0] in;
   output [3:0] out;
   input tri_en;

   bufif0 buf_array[3:0] (out, in, tri_en); // array of three-state buffers

endmodule
```

The statement

```
bufif0 buf_array[3:0] (out, in, tri_en);   // instance name is
                                           // indexed here
```

in Figure 8-8 uses buf_array[3:0] as instance name. Notice that the instance name is indexed here. This statement could be replaced using

```
bufif0 buf_array3 (out[3], in[3], tri_en); // instance name not
                                           // indexed
bufif0 buf_array2 (out[2], in[2], tri_en);
bufif0 buf_array1 (out[1], in[1], tri_en);
bufif0 buf_array0 (out[0], in[0], tri_en);
```

where the instance names are not indexed. The instance names are not signal names.

The circuit in Figure 8-5, presented using the behavioral model in Figure 8-6, can be written using Verilog built-in primitives as in Figure 8-9.

FIGURE 8-9: Verilog Tristate Buffer Module Using Built-In Primitives

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

   bufif1 buf_one (f,a,b); //output first followed by inputs, control last.
   bufif1 buf_two (f,c,d);

endmodule
```

Table 8.1 lists the built-in primitives in Verilog. The first 12 are gate-level primitives, and the rest are switch-level primitives. Switch-level models are not discusseded in this book, but they are useful if low-level models of designs are to be built.

TABLE 8-1: Built-In Gate and Switch Primitives in Verilog

Built-in Primitive Type	Primitives
n-input gates	**and, nand, nor, or, xnor, xor**
n-output gates	**buf, not**
Three-state gates	**bufif0, bufif1, notif0, notif1**
Pull gates	**pulldown, pullup**
MOS switches	**cmos, nmos, pmos, rcmos, rnmos, rpmos**
Bidirectional Switches	**rtran, rtranif0, rtranif1, tran, transif0, tranif1**

The built-in primitives can have an optional delay specification. A delay specification can contain up to three delay values, depending on the gate type. For a three-delay specification,

(i) the first delay refers to the transition to the rise delay (i.e., transition to 1),

(ii) the second delay refers to the transition to the fall delay (i.e., transition to 0), and

(iii) the third delay refers to the transition to the high-impedance value (i.e., turn-off).

If only one delay is specified, it is rise delay. If there are only two delays specified, they are rise delay and fall delay. If turn-off delay must be specified, all three delays must be specified. The **pullup** and **pulldown** instance declarations must not include delay specifications. The following are examples of built-in primitives with one, two, and three delays:

```
and #(10) a1 (out, in1, in2);            // only one delay
                                         // (so rise delay=10)
and #(10,12) a2 (out, in1, in2);         // rise delay=10 and
                                         // fall delay=12
bufif0 #(10,12,11) b3 (out, in, ctrl);   // rise, fall, and
                                         // turn-off delays
```

8.5 User-Defined Primitives

The predefined gate primitives in Verilog can be augmented with new primitive elements called user-defined primitives (UDPs). UDPs define the functionality of the primitive in truth table or state table form. Once these primitives are specified by the user, instances of these new UDPs can be created in exactly the same manner as built-in gate primitives are instantiated. While the built-in primitives are only combinational, UDPs can be combinational or sequential.

A *combinational UDP* uses the value of its inputs to determine the next value of its output. A *sequential UDP* uses the value of its inputs and the current value of its output to determine the value of its output. Sequential UDPs provide a way to model sequential circuits such as flip-flops and latches. A sequential UDP can model both level-sensitive and edge-sensitive behavior.

A UDP can have multiple input ports but has exactly one output port. Bidirectional inout ports are not permitted on UDPs. All ports of a UDP must be scalar—that is, vector ports are not permitted. Each UDP port can be in one of three states: 0, 1, or X. The tristate or high-impedance state value Z is not supported. If Z values are passed to UDP inputs, they shall be treated the same as X values. In sequential UDPs, the output always has the same value as the internal state.

A UDP begins with the keyword **primitive**, followed by the name of the UDP. The functionality of the primitive is defined with a truth table or state table, starting with the keyword **table** and ending with the keyword **endtable**. The UDP definition then ends with the keyword **endprimitive**. The truth table for a UDP consists of a section of columns, one for each input followed by a colon and finally the output column. The multiplexer we defined in Figure 2-36 using continuous assign statements is redefined as a UDP in Figure 8-10.

FIGURE 8-10: User-Defined Primitive (UDP) for a 2-to-1 Multiplexer

```
primitive mux1 (F, A, I0, I1);
  output F;
  input A, I0, I1; //A is the select input

table
  // A   I0  I1      F
     0   1   0   :   1   ;
```

```
      0   1   1   :   1   ;
      0   1   x   :   1   ;
      0   0   0   :   0   ;
      0   0   1   :   0   ;
      0   0   x   :   0   ;
      1   0   1   :   1   ;
      1   1   1   :   1   ;
      1   x   1   :   1   ;
      1   0   0   :   0   ;
      1   1   0   :   0   ;
      1   x   0   :   0   ;
      x   0   0   :   0   ;
      x   1   1   :   1   ;
endtable

endprimitive
```

The first entry in the truth table in Figure 8-10 can be explained as follows: when `A` equals `0`, `I0` equals `1`, and `I1` equals `0`, then output `F` equals `1`. The input combination `0xx (A=0, I0=x, I1=x)` is not specified. If this combination occurs during simulation, the value of output port `F` will become `x`. Each row of the table in the UDP is terminated by a semicolon.

The multiplexer model can also be specified more concisely in a UDP by using ?. The ? means that the signal listed with ? can take the values of 0,1, or x. Using ? the multiplexer can be defined as in Figure 8-11.

FIGURE 8-11: User-Defined Primitive (UDP) for a 2-to-1 Multiplexer Using?

```
primitive mux2 (F, A, I0, I1);
  output F;
  input A, I0, I1;

table
  // A   I0  I1      F
      0   1   ?   :   1   ;    // ? can equal 0, 1, or x
      0   0   ?   :   0   ;
      1   ?   1   :   1   ;
      1   ?   0   :   0   ;
      x   0   0   :   0   ;
      x   1   1   :   1   ;
endtable

endprimitive
```

UDPs are instantiated just as are built-in primitives. For instance, the preceding multiplexer can be instantiated by using the statement

```
mux1 (outF, Sel, in1, in2)
```

where `outF, Sel, in1`, and `in2` are the names of the output, select, data input1, and data input2 signals.

As an example of a sequential UDP, we present the description of a D flip-flop in Figure 8-12. In a sequential UDP, the output must be defined as a **reg**. Edge-sensitive behavior can be represented in tabular form by listing the value before and after the edge. For instance, (01) means a positive (rising) edge and (10) means a negative (falling) edge. In the sequential UDP, there is an additional colon separating the inputs to the present state in the state table.

FIGURE 8-12: User-Defined Primitive (UDP) for a D Flip-Flop

```
primitive DFF (Q, CLK, D);
  output Q;
  input CLK, D;

  reg Q;

table
  // CLK,  D,    Q,  Q+
    (01)  0   : ? : 0  ; //rising edge with input 0
    (01)  1   : ? : 1  ; //rising edge with input 1
    (0?)  1   : 1 : 1  ; //Present state 1, either rising edge or steady clock
    (?0)  ?   : ? : -  ; //Falling edge or steady clock, no change in output
     ?  (??) : ? : -  ; //Steady clock, ignore inputs, no change in output
endtable

endprimitive
```

The (01) in the first line of the table indicates a rising edge clock. The '-' in the output column means that the output should not change for any of the circumstances covered by that line. For instance, the line

```
(?0) ? ? : - ;
```

means that if there is a falling edge or steady clock, whether the input and present state are 0,1, or x, the output must not change. This line actually represents 27 possibilities, because each ? represents three possibilities. If this line was omitted in the code, the simulator would yield an x output in these situations. It is important to make the truth table as unambiguous as possible by specifying all possible cases. The last line of code clarifies further that under steady clock (i.e., no clock edges), the flip-flop must ignore all inputs.

```
? (??) : ? : - ;
```

The rows in the primitive truth table do not need to be in order. Hence the row order

```
0 1 0 : 1 ;
0 0 0 : 0 ;
```

is acceptable. A simulator scans the truth table from top to bottom. If level-sensitive behavior such as asynchronous set and reset are in a table along with edge-sensitive behavior for data, the level-sensitive behavior should be listed before the edge-sensitive behavior.

8.6 SRAM Model

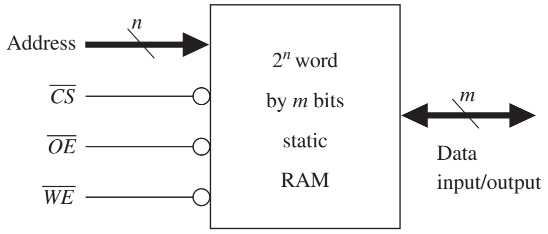
In this section, we develop a Verilog model to represent the operation of a static RAM (SRAM). RAM stands for random-access memory, which means that any word in the memory can be accessed in the same amount of time as any other word. Strictly speaking, ROM memories are also random-access, but historically, the term RAM is normally applied only to read-write memories. This model also illustrates the usefulness of the multivalued logic system. Multivalued logic is used to model tristate conditions on the memory data lines.

Figure 8-13 shows the block diagram of a static RAM with *n* address lines, *m* data lines, and three control lines. This memory can store 2^n words, each *m* bits wide. The data lines are bidirectional in order to reduce the required number of pins and the package size of the memory chip. When reading from the RAM, the data lines are outputs; when writing to the RAM, the data lines serve as inputs. The three control lines function as follows:

- $\overline{CS}$ When asserted low, chip select selects the memory chip so that memory read and write operations are possible.
- $\overline{OE}$ When asserted low, output enable enables the memory output onto an external bus.
- $\overline{WE}$ When asserted low, write enable allows data to be written to the RAM.

We say that a signal is asserted when it is in its active state. An active-low signal is asserted when it is low, and an active-high signal is asserted when it is high.

FIGURE 8-13: Block Diagram of Static RAM



The truth table for the RAM (Table 8-2) describes its basic operation. High-Z in the I/O column means that the output buffers have high-Z outputs and the data inputs are not used. In the read mode, the address lines are decoded to select m memory cells, and the data comes out on the I/O lines after the memory access time has elapsed. In the write mode, input data is routed to the latch inputs in the selected memory cells when $\overline{WE}$ is low, but writing to the latches in the memory cells is not completed until either $\overline{WE}$ goes high or the chip is deselected. The truth table does not take memory timing into account.

TABLE 8-2: Truth Table for Static RAM

$\overline{CS}$	$\overline{OE}$	$\overline{WE}$	Mode	I/O pins
H	X	X	not selected	high-Z
L	H	H	output disabled	high-Z
L	L	H	read	data out
L	X	L	write	data in

We now write a simple Verilog model for the memory that does not take timing considerations into account. In Figure 8-14, the RAM memory array is represented by an array of registers (*RAM1*). This memory has 256 words, each of which are 8 bits. The RAM process sets the I/O lines to high-Z if the chip is not selected. If *We_b* = 1, the RAM is in the read mode and *IO* is the data read from the memory array. If *We_b* = 0, the memory is in the write mode, and the data on the I/O lines is stored in *RAM1* on the rising edge of *We_b*. This is a RAM with asynchronous read and synchronous write.

FIGURE 8-14: Simple Memory Model

```
module RAM6116 (Cs_b, We_b, Oe_b, Address, IO);

   input Cs_b;
   input We_b;
   input Oe_b;
   input[7:0] Address;
   inout[7:0] IO;

   reg[7:0] RAM1[0:255];

   assign IO = (Cs_b == 1'b1 | We_b == 1'b0 | Oe_b == 1'b1) ?
         8'bZZZZZZZZ : RAM1[Address] ; // read from RAM

   always @(We_b, Cs_b)
   begin
       @(negedge We_b)  //falling edge of We_b
       if(Cs_b == 1'b0)
       begin
        RAM1[Address] <= IO ;  // write to RAM
       end
   end
endmodule
```

8.7 Model for SRAM Read/Write System

In order to further illustrate the use of multivalued logic, we present an example with a **bidirectional tristate bus**. We will design a memory read-write system that reads the content of 32 memory locations from a RAM, increments each data

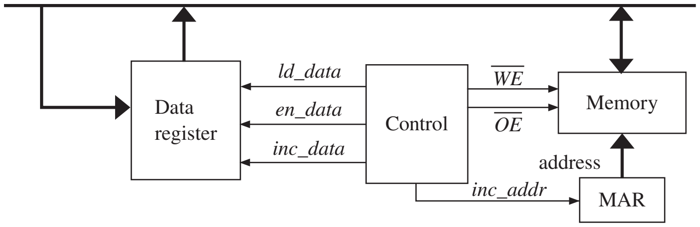


FIGURE 8-15: Block Diagram of RAM Read-Write System

value, and stores it back into the RAM. A block diagram of the system is shown in Figure 8-15. In order to hold the word that we read from memory, we use a **data register**. In order to hold the memory address that we are accessing, we use a memory address register **(MAR)**. The system reads a word from the RAM, loads it into the data register, increments the data register, stores the result back in the RAM, and then increments the memory address register. This process continues until the memory address equals 32.

The data bus is used as a **bidirectional bus**. During the read operation, the memory output appears on the bus, and the data register output to the data bus will be in a tristate condition. During the write operation, the data register output is on the data bus and the memory will use it as input data.
Control signals required to operate the system are as follows:

ld_data	load data register from Data Bus
en_data	enable data register output onto Data Bus
inc_data	increment Data Register
inc_addr	increment MAR
$\overline{WE}$	Write Enable for SRAM
$\overline{OE}$	Output Enable for SRAM

Figure 8-16 shows the SM chart for the system. The SM chart uses four states. In state S_0, the SRAM drives the memory data onto the bus and the memory data is loaded into the Data Register. The control signal $\overline{OE}$ and *ld_data* are true in this state. The Data Register is incremented in S_1. The *en_data* control signal is true in state S_2, and hence the Data Register drives the bus. Write enable $\overline{WE}$ is an active-low signal that is asserted low only in S_2; as a result, $\overline{WE}$ is high in the other states. The writing to the RAM is initiated in S_2 and completed on the rising edge of $\overline{WE}$, which occurs during the transition from S_2 to S_3. The memory address is incremented. The process continues until the address is 32. State S_3 checks this and produces a done signal when the address reaches 32.

FIGURE 8-16: SM Chart for RAM System

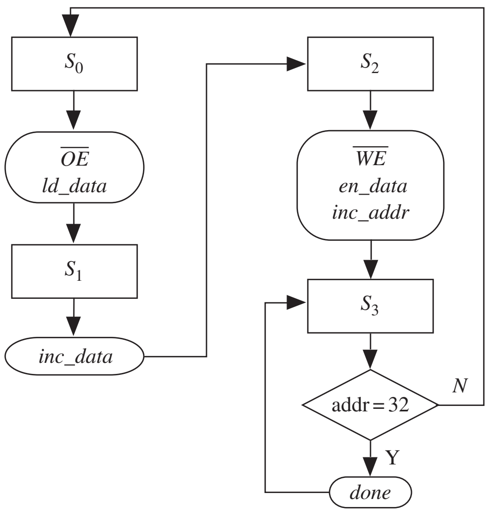


Figure 8-17 shows the Verilog code for the RAM system. The first always statement represents the SM chart, and the second always statement is used to update the registers on the rising edge of the clock. A short delay is added when the address is incremented to make sure the write to memory is completed before the address changes. A concurrent statement is used to simulate the tristate buffer, which enables the Data Register output to go onto the I/O lines.

FIGURE 8-17: Verilog Code for RAM System

```
// SRAM Read-Write System model

module RAM6116_system ();

   reg[1:0] state;
   reg[1:0] next_state;
   reg inc_adrs;
   reg inc_data;
   reg ld_data;
   reg en_data;
   reg Cs_b;
   reg clk;
   reg Oe_b;
   reg done;
   reg We_b;
   reg[7:0] Data;
   reg[7:0] Address;
   wire[7:0] IO;

   initial
   begin
      inc_adrs = 1'b0;
      inc_data = 1'b0;
      ld_data = 1'b0;
      en_data = 1'b0;
      clk = 1'b0;
      Cs_b = 1'b0;
      Oe_b = 1'b0;
      done = 1'b0;
      We_b = 1'b1; //initialize to read mode
      Address = 8'b00000000; //address register
   end

   RAM6116 RAM1 (Cs_b, We_b, Oe_b, Address, IO);

   always @(state or Address)
   begin : control
      ld_data = 1'b0 ;
      inc_data = 1'b0 ;
      inc_adrs = 1'b0 ;
      en_data = 1'b0 ;
      done = 1'b0 ;
      We_b = 1'b1 ;
      Cs_b = 1'b0 ;
      Oe_b = 1'b1 ;

      case (state)
         0 :
                   begin
                      Oe_b = 1'b0 ;
                      ld_data = 1'b1 ;
                      next_state = 1 ;
                   end
         1 :
                   begin
                      inc_data = 1'b1 ;
                      next_state = 2 ;
                   end
         2 :
                   begin
                      We_b = 1'b0 ;
                      en_data = 1'b1 ;
                      inc_adrs = 1'b1 ;
                      next_state = 3 ;
                   end
         3 :
                   begin
                      if (Address == 8'b00100000)
                      begin
                         done = 1'b1 ;
                         next_state = 3 ;
                      end
                      else
                      begin
                         next_state = 0 ;
                      end
                   end
      endcase
   end

   always @(posedge clk) //always block to update data register
   begin : register_update
      state <= next_state ;
      if (inc_data == 1'b1)
      begin
         Data <= Data + 1 ; //increment data in data register
      end
      if (ld_data == 1'b1)
      begin
         Data <= unsigned(IO) ; //load data register from bus
      end
      if (inc_adrs == 1'b1)
      begin
```

```
          Address <= #1 Address + 1 ; //delay added to allow completion of memory write
       end
   end

//Concurrent statements
   always #100 clk = ~clk ;
   assign IO = (en_data ==1'b1) ? Data : 8'bZZZZZZZZ ;

endmodule
```

This system can be modified to include all memory locations for testing the correctness of the entire SRAM. Memory systems are often tested by writing checkerboard patterns (alternate 0s and 1s) in all locations. For instance, one can write 01010101 (55 hexadecimal) into all odd addresses and 10101010 (hexadecimal AA) into all even addresses. Then the odd and even locations can be swapped. Developing Verilog code for such a system is left as an exercise problem.

8.8 Rise and Fall Delays of Gates

In Section 8.4, we described built-in primitives with delay specifications. In this section, we show how modules can be created with variable delays. *Parameters* are commonly used to specify constant values for a module in such a way that the parameter values may be specified when the module is instantiated. For example, the rise and fall times for a gate could be specified as parameters, and different numeric values for these parameters could be assigned for each instance of the gate. The example of Figure 8-18 describes a two-input NAND gate whose rise and fall delay times depend on the number of loads on the gate. In the NAND2 module declaration, *Trise*, *Tfall*, and *load* are parameters that specify default values for rise time, fall time, and the number of loads. An internal *nand_value* is computed whenever *a* or *b* changes. If *nand_value* has just changed to a 1, a rising output has occurred and the gate delay time is computed as

*Trise + 3ns * load*

where 3ns is the added delay for each load. Otherwise, a falling output has just occurred, and the gate delay is computed as

*Tfall + 2ns * load*

where 2ns is the added delay for each load.

FIGURE 8-18: Rise/Fall Time Modeling Using Parameter

```
module NAND2 (a, b, c);

   parameter Trise  = 3;
   parameter Tfall  = 2;
   parameter load   = 1;
   input a;
   input b;
   output c;

   reg c;

   wire nand_value;

   assign nand_value = ~(a & b) ;

   always @ (nand_value)
   begin
     if (nand_value == 1'b1)
       #(Trise + 3 * load) c = 1'b1;
     else
       #(Tfall + 3 * load) c = 1'b0;
   end

endmodule

module NAND2_test (in1, in2, in3, in4, out1, out2);
   input in1;
   input in2;
   input in3;
   input in4;
   output out1;
   output out2;

   NAND2 #(2, 1, 2) U1 (in1, in2, out1);
   NAND2 U2 (in3, in4, out2);
endmodule
```

The module *NAND2_test* tests the NAND2 component. The parameter declaration in the module specifies default values for *Trise*, *Tfall*, and *load*. When *U1* is instantiated, the parameter map specifies different values for *Trise*, *Tfall*, and *load*. When *U2* is instantiated, no parameter map is included, so the default values are used. Another way to instantiate *U1* is by using **defparam** as follows:

```
defparam U1.Trise = 2;
defparam U1.Tfall = 1;
defparam U1.load = 2;
NAND2 U1 (in1,in2, out1);
NAND2 U2 (in3, in4, out2);
```

8.9 Named Association

Up to this point, we have used *positional association* in the port maps and parameter maps that are part of an instantiation statement. For example, assume that the module declaration for a full adder is

```
module FullAdder (Cout, Sum, X, Y, Cin);

output Cout;
output Sum;
input X;
input Y;
input Cin;

.........

endmodule
```

The statement

```
FullAdder FA0 (Co[0], S[0], A[0], B[0], Ci[0]);
```

creates a full adder and connects *A[0]* to the *X* input of the adder, *B[0]* to the *Y* input, Ci[0] to the C_{in} input, Co[0] to the C_{out} output, and *S[0]* to the *Sum* output of the adder. The first signal in the port map is associated with the first signal in the module declaration, the second signal with the second signal, and so on.

As an alternative, we can use *named association*, in which each signal in the port map is explicitly associated with a signal in the port of the module declaration. For example, the statement

```
FullAdder FA0 (.Sum(S[0]), .Cout(Co[0]), .X(A[0]), .Y(B[0]),
.Cin(Ci[0]));
```

makes the same connections as the previous instantiation statement—that is, *Sum* connects to *S[0]*, C_{out} connects to *Co[0]*, *X* connects to *A[0]*, and so on. When named association is used, the order in which the connections are listed is not important, and any port signals not listed are left unconnected. Use of named association makes code easier to read, and it offers more flexibility in the order in which signals are listed.

When named association is used with a parameter map, any unassociated parameter assumes its default value. For example, if we replace the statement in Figure 8-18 labeled `U1` with the following:

```
NAND2 #(.load(3), .Trise(4)) U1 (in1, in2, out1);
```

Tfall would assume its default value of 2 ns.

8.10 Generate Statements

In Chapter 2, we instantiated four full adders and interconnected them to form a 4-bit adder. Specifying the port maps for each instance of the full adder would become very tedious if the adder had 8 or more bits. When an iterative array of identical operations or module instance is required, the **generate** statement provides an easy way of instantiating these components. The example of Figure 8-19 shows how a **generate** statement can be used to instantiate four 1-bit full adders to create a 4-bit adder. A 5-bit vector is used to represent the carries, with C_{in} the same

FIGURE 8-19: Adder4 Using Generate Statement

```
module Adder4 (A, B, Ci, S, Co);
   input[3:0] A; //inputs
   input[3:0] B;
   input Ci;
   output[3:0] S; //outputs
   output Co;

   wire[4:0] C;

   assign C[0] = Ci ;

   genvar i;
   generate
   for (i=0; i<4; i=i+1)
     begin: gen_loop
       FullAdder FA (A[i], B[i], C[i], C[i+1], S[i]);
     end
   endgenerate

   assign Co = C[4] ;

endmodule

module FullAdder (X, Y, Cin, Cout, Sum);
   input X; //inputs
   input Y;
   input Cin;
   output Cout; //outputs
   output Sum;

   assign #10 Sum = X ^ Y ^ Cin ;
   assign #10 Cout = (X & Y) | (X & Cin) | (Y & Cin) ;
endmodule
```

as $C(0)$ and C_{out} the same as $C(4)$. The **for** loop generates four copies of the full adder, each with the appropriate **port map** to specify the interconnections between the adders.

Another example where the **generate** statement would have been very useful is the array multiplier. The Verilog code for the array multiplier (Chapter 4) made repeated use of port map statements in order to instantiate each module instance. They could have been replaced with **generate** statements.

In the preceding example, we used a **generate** statement of the form

```
genvar gen_variable1,…;
generate
for (  for_loop_condition with gen_variables )
  concurrent statement(s)
endgenerate
```

At compile time, a set of `concurrent statement(s)` is generated for each value of the identifier in the given range. In Figure 8-19, one concurrent statement—a module instance instantiation statement—is used. The statement

```
FullAdder FA (A[i], B[i], C[i], C[i+1], S[i]);
```

inside the **generate** clause creates the effect of the following four statements:

```
FullAdder FA (A[0], B[0], C[0], C[1], S[0]);
FullAdder FA (A[1], B[1], C[1], C[2], S[1]);
FullAdder FA (A[2], B[2], C[2], C[3], S[2]);
FullAdder FA (A[3], B[3], C[3], C[4], S[3]);
```

A **generate** statement itself is defined to be a concurrent statement, so nested generate statements are allowed.

Conditional Generate

A **generate** statement with an **if** clause may be used to conditionally generate a set of concurrent statement(s). This type of **generate** statement has the form

```
generate
if condition
  concurrent statement(s)
endgenerate
```

In this case, the concurrent statements(s) are generated at compile time only if the condition is true. Conditional compilation can also be accomplished using the **`ifdef** compiler directive, which is explained in Section 8-12. Conditional compilation is useful for

(i) Selectively including behavioral, structural, or switch-level models as desired.

(ii) Selectively including different timing or structural information.

(iii) Selectively including different stimuli for different runs under different scenarios.

(iv) Selectively adapting the module functionality to similar but different needs from different customers.

Figure 8-20 illustrates the use of conditional compilation using a **generate** statement with an **if** clause. An N-bit left-shift register is created if *Lshift* is true, using the statement

```
generate
  if(Lshift)
    assign shifter = {Q[N - 1:1], Shiftin} ;
  else
    assign shifter = {Shiftin, Q[N:2]} ;
endgenerate
```

If *Lshift* is false, a right-shift register is generated using another conditional **generate** statement. The example also shows how parameters and **generate** statements can be used together. It illustrates the use of parameters to write a Verilog model with parameters so that the size and function can be changed when it is instantiated. It also shows another example of named association.

FIGURE 8-20: Shift Register Using Conditional Compilation

```
module shift_reg (D, Qout, CLK, Ld, Sh, Shiftin);
   parameter N = 4;
   parameter Lshift = 1;

   output[N:1] Qout;
   input[N:1] D;
   input CLK;
   input Ld;
   input Sh;
   input Shiftin;

   reg[N:1] Q;
   wire[N:1] shifter;

   assign Qout = Q ;

   generate
     if(Lshift)
       assign shifter = {Q[N - 1:1], Shiftin} ; //left shift register
     else
       assign shifter = {Shiftin, Q[N:2]} ; //right shift register
   endgenerate

   always @(posedge CLK)
   begin
     if (Ld == 1'b1)
       begin
         Q <= D ;
       end
     else if (Sh == 1'b1)
       begin
         Q <= shifter ;
       end
   end
endmodule
```

8.11 System Functions

In addition to tasks and functions that the user can create, Verilog has tasks and functions at the system level. System tasks and functions in Verilog start with the **$** sign. System tasks that we have used so far include **$display, $monitor,** and **$write**. The file I/O functions tasks discussed in Section 8.13 are also system tasks. These system functions are not for synthesis, however; they are intended for convenience during simulation.

Systems tasks mainly include display tasks for outputting text or data during simulation, file I/O tasks, and simulation control tasks such as **$finish** and **$stop.** We describe a few of the system tasks/functions in this section.

8.11.1 Display Tasks

Display tasks are very useful during simulation to check outputs. There are several of them, and there are variations for displaying data in binary, hex, or octal formats (e.g., **$displayb, $displayh, $displayo).** The major tasks are the following:

$display	Immediately outputs text or data with new line.
$write	Immediately outputs text/data without new line.
$strobe	Outputs text or data at the end of the current simulation step.
$monitor	Displays text or data for every event on signal.

8.11.2 File I/O Tasks

File I/O tasks are described in detail in Section 8.13.

8.11.3 Simulation Control Tasks

There are two system tasks to control the simulation: **$stop** and **$finish**. The **$stop** task is a system function to temporarily suspend simulation for interaction. **$finish** is a system task to come out of simulation. The **$finish** system task simply makes the simulator exit and pass control back to the host operating system. Both these tasks take an optional expression argument (0, 1, or 2) that determines what type of diagnostic message is printed.

8.11.4 Simulation Time Functions

It can be beneficial to access simulation time. The following system functions provide access to current simulation time:

$time	Returns an integer that is a 64-bit time, scaled to the timescale unit of the module that invoked it.
$stime	Returns an unsigned integer that is a 32-bit time, scaled to the time-scale unit of the module that invoked it. If the actual simulation time does not fit in 32 bits, the low-order 32 bits of the current simulation time are returned.
$realtime	Returns a real number time that, like **$time**, is scaled to the time unit of the module that invoked it.

The following statement can be used to print simulation time:

```
$monitor($time);
```

We can assign it to other variables as follows:

```
time simtime;     // time is one of the variable data types
simtime = $time;  // Assign current simulation time to variable
                  // simtime
```

8.11.5 Conversion Functions

There are also system functions to perform conversions between data types.

$signed() and $unsigned()

Two system functions are used to handle type casting on expressions: **$signed()** and **$unsigned()**. These functions evaluate the input expression and return a value with the same size and value of the input expression and the type defined by the function.

```
reg [3:0] regA;
reg signed [7:0] regB;

regA = $unsigned(-4);          // regA = 4'b1100
regB = $signed (8′b11111100);  // regB = -4
```

$realtobits and $bitstoreal

The system functions **$realtobits** and **$bitstoreal** are useful for type conversions. Although Verilog is not a strongly typed language, there are type restrictions on many operators. For instance, the concatenate, replicate, modulus, case equality, reduction, shift, and bit-wise operators cannot work with real-number operands. Real numbers can be converted to bits or vice versa using the afore-mentioned conversion functions. The system functions **$realtobits** and **$bitstoreal** can also be used for passing bit patterns across module ports, when they are represented as real numbers on the other side. For example, consider the following code:

```
module bus_driver (net_bus);
  output net_bus;

  real sig;
  wire [64:1] net_bus = $realtobits(sig);

endmodule
```

The variable `sig`, which is in real type is converted to bits using the statement

```
wire [64:1] net_bus = $realtobits(sig);
```

and passed to `net_bus`.

8.11.6 Probablistic Functions

Verilog has system functions to generate probabilistic distributions or random numbers**.** The **$random** function returns a random signed integer that is 32-bits.

8.12 Compiler Directives

Verilog has several compiler directives that add programming convenience to the development and maintenance of Verilog modules. All Verilog compiler directives are preceded by the (‘) character. This character is called *grave accent* (ASCII 0x60). It is different from the character (′), which is the *apostrophe* character

(ASCII 0x27). The scope of a compiler directive extends from the point where it is processed, across all files processed, to the point where another compiler directive supersedes it or the processing completes. There is no semicolon (;) at the end of the line with the compiler directive. This section describes a few compiler directives.

8.12.1 `define

The directive **`define** creates a macro for text substitution. This directive can be used both inside and outside module definitions. After a text macro is defined, it can be used in the source description by using the (') character, followed by the macro name. This directive can be used to define constants or expressions. For example,

```
`define wordsize 16
```

causes the string `wordsize` to be replaced by 16. Such replacements can increase the readability and modifiability of the code.

In the following example, a macro `max` is defined to represent an expression. In order to invoke the macro, one must write `` `max``. For example, the statement

```
`define max(a,b)((a) > (b) ? (a) : (b))
```

defines the macro. The larger of the two operands is returned by this macro. The statement

```
n = `max(p+q, r+s) ;
```

invokes the macro. This macro expands as

```
n = ((p+q) > (r+s)) ? (p+q) : (r+s) ;
```

Here, the larger of the two expressions out of (`p + q`) and (`r + s`) will be evaluated twice.

8.12.2 `include

This compiler directive is used to insert the contents of one source file into another file during compilation (i.e., for file inclusion). The result is as though the contents of the included source file appear in place of the **`include** compiler directive. The **`include** compiler directive can be used to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries. It helps modular code development and facilitates structured organization of the files in the design. It seriously contributes to the convenience in managing source code of a design.

The compiler directive is followed by the filename to be included. Only white space or a comment may appear on the same line as the **`include** compiler directive. A file included in the source using the **`include** compiler directive may contain other **`include** compiler directives. The number of nesting levels allowed might vary in different Verilog compilers. The following are valid examples of the **`include** compiler directive:

```
`include "lab3/sevenseg.v"
`include "myfile"
`include "myfile" // including myfile. A comment or only white
                  // space allowed
```

If file `sub.v` contains

```
reg g;
initial
   g = 0;
```

and file `main.v` contains

```
module main;
`include sub.v
always @(posedge clk)
   g <= ~g;
endmodule;
```

it is equivalent to having one file as follows:

```
module main;
reg g;
initial
   g = 0;
always @(posedge clk)
   g <= ~g;
endmodule;
```

8.12.3 `ifdef

Verilog contains several compiler directives for conditional compilation. These directives allow one to optionally include lines of a Verilog HDL source description during compilation. Conditional compilation can also be accomplished with the conditional **generate** statement as in Section 8.10; however, the **`ifdef** compiler directive is convenient due to the existence of similar directives in high-level languages such as C. Many designers are already familiar with similar directives from C.

The **`ifdef** compiler directive is followed by a `text_macro_name` in the code. At compile time, the compiler checks for the definition of the `text_macro_name`. If the `text_macro_name` is defined, then the lines following the **`ifdef** directive are included. Otherwise, the statements in the **`else** clause are included. The end of the construct is marked with the **`endif**. As mentioned in Section 8.10, conditional compilation is useful for

- **(i)** Selectively including behavioral, structural, or switch-level models as desired.
- **(ii)** Selectively including different timing or structural information.
- **(iii)** Selectively including different stimulus for different runs under different scenarios.
- **(iv)** Selectively adapting the module functionality to similar but different needs from different customers.

The example that follows shows a simple usage of an **`ifdef** directive for selecting between a behavioral design and a structural design. If the identifier `behavioral` is defined, a continuous net assignment will be compiled; otherwise, a built-in gate primitive **and** will be instantiated. In **and** `a1 (f,a,b), a1` is the instance name of the instantiated gate and **and** is the built-in gate primitive introduced in Section 8.4.

```
module selective_and (f, a, b);
  output f;
  input a, b;

  `ifdef behavioral
     wire f = a & b;
  `else
     and a1 (f,a,b);
  `endif

endmodule
```

8.13 File I/O Functions

The ability to handle files and text is very valuable while testing large Verilog designs. This section introduces file input and output in Verilog. Files are frequently used with test benches to provide a source of test data and to provide storage for test results. The format of the file I/O functions is based on the C standard I/O functions, such as fopen and fclose. A file can be opened for reading or writing using the **$fopen** function as shown in the following:

```
integer file_r, file_w;

file_r = $fopen("filename",r);    // Reading a file
file_w = $fopen("filename",w);    // Writing a file
```

The filename can be either a double-quoted string or a reg having filename information. $fopen can have multiple modes: – ‘r’ for reading, ‘w’ for writing, and ‘a’ for appending. If the file is successfully opened, it returns an integer number indicating the file number. If there is an error for opening the file, NULL value will be returned.

To close an opened file, the $fclose function can be used. If $fclose is used without any arguments, it closes all opened files. If an argument is specified, it will close only a file in which the descriptor is given.

```
$fclose(file_r);   // Closing only one file
$fclose();         // Closing all opened files
```

Verilog supports the following ways of handling a file:

$fopen/$fclose	open/close an existing file for reading or writing.
$feof	tests for end of file. If an end-of-file has been reached while reading a file, a non-zero value is returned; otherwise, a 0 is returned.

```
integer file;
reg eof;

eof = $feof(file);
```

$ferror returns the error status of a file. If an error has occurred while reading a file, $ferror returns a non-zero value; otherwise, it returns a 0.

```
integer file;
reg error;

error = $ferror(file);
```

$fgetc reads a single character from the specified file and returns it. If the end-of-file is reached, $fgetc returns EOF.

```
integer file, char;

char = $fgetc(file);
```

$fputc writes a single character to a file. It returns EOF if there is an error, 0 otherwise.

```
integer stream, r, char;

r = $fputc(stream, char);
```

$fscanf parses formatted text from a file according to the format and writes the results to args.

```
integer file, count, tmp_a;

count = $fscanf(file,"%d", tmp_a);
//   count = $fscanf(file, format, args);
```

$fprintf writes a formatted string to a file.

```
integer file, tmp_a, tmp_b, ret;

file = $fopen("test.log", w);
ret = $fprintf(file, "%d : %x", tmp_a, tmp_b);
```

$fread reads binary data from the file specified by the file descriptor into a register or into a memory.

```
integer rd, file;
reg rd_value;

rd = $fread(file, rd_value);
```

$fwrite writes binary data from a register to the file specified by the file descriptor.

```
integer file;
reg tmp_a, tmp_b;

tmp_a = 0;
tmp_b = 0;
file = $fopen("test.log");
$fwrite(file,"A=%d B=%d",tmp_a,tmp_b);
```

$readmemb
$readmemh To read data from a file and store it in memory, use the functions **$readmemb** and **$readmemh**. The **$readmemb** task reads binary data, and **$readmemh** reads hexadecimal data. Data has to be present in a text file.

```
$readmemb ("file", memory [,start_addr [,finish_addr]]) ;
$readmemh ("file", memory [,start_addr [,finish_addr]]) ;

$readmemb("file.bin", mem);
$readmemh("file.hex", mem);
```

Figure 8-21 gives a Verilog code that reads hexadecimal data from a file using **$readmemh**. Four 32-bit data are stored in a file "data.txt". **$readmemh** will read data and store it to the storage.

FIGURE 8-21: Verilog Code to Read Hexadecimal Data from a File Using $readmemh

```
module example_readmemh;
  reg [31:0] data [0:3];

  initial $readmemh("data.txt", data);

  integer i;

  initial begin
    $display("read hexa_data:");
    for (i=0; i < 4; i=i+1)
    $display("%d:%h",i,data[i]);
  end
endmodule
```

Figure 8-22 shows a Verilog code that read a file with commands, addresses, and data values. The code uses **$fopen** for accessing the file and **$fscanf** for parsing each line. In Figure 8-22, the **disable** statement is used to terminate the block named 'file_read'. The **disable** statement followed by a task name or block name will disable only tasks and named blocks. It cannot disable functions.

FIGURE 8-22: Verilog Code to Read and Parse a File Using $fopen

```
   `define NULL 0
   `define EOF 32'hffffffff

 module file_read;
 integer file, ret;
 reg [31:0] r_w, addr, data;

 initial
    begin : file_read

    file = $fopen("data", r);
    if (file == `NULL)
        disable file_read;

   while (!$feof(file))
        begin
        ret = $fscanf(file, "%s %h %h\n", r_w, addr, data);
        case (r_w)
```

```
         "rd":
              $display("READ mem[%h] => %h", addr, data);
          "wr":
              $display("WRITE mem[%h] <= %h", addr, data);
          default:
              $display("Unknown command '%s'", r_w);
        endcase
        end

    ret = $fclose(file);
    end
endmodule
```

8.14 Timing Checks

Verilog facilitates several timing checks—for example, to check whether setup and hold times of flip-flops are met. The timing check tasks start with a **$**, but they are not system tasks. Timing checks must not appear in procedural code. Timing checks must be within **specify** ….. **endspecify** blocks. Verilog provides several timing checks, but we will discuss only the few following examples:

$setup	This timing check displays a warning **if setup time is not met**. It needs a data event, a reference event, and a limit (the setup time) to be specified as in **$setup (data-event, reference_event, limit)** For example, **$setup (D, posedge CLK, 10)**
$hold	This timing check displays a warning **if hold time is not met**. It needs a reference event, a data event, and a limit (the hold time) to be specified as in **$hold (reference_event, data_event,limit)** For example, **$hold (posedge CLK, D, 2)**
$skew	This timing check displays a warning **if skew is above the limit**. It needs a reference event, a data event, and a limit to be specified as in **$skew (reference_event, data-event, limit)** For example, **$skew (posedge CLK1, posedge CLK2, 4)**
$width	This timing check displays a warning **if pulse width is shorter than limit**. It needs a data event, a reference event, and a limit, but the data event is not explicitly specified. The data event is derived from reference event as a reference event signal with opposite edge. The pulse width has to be greater than or equal to the limit in order to avoid a timing violation. The reference event must be an edge-triggered event. It is specified as in **$width (reference event, limit)** For example, **$width (posedge CLK, 20)**

Problems

8.1 Write a Verilog function to complete the following module, one that automatically counts the number of shifts. It needs to shift a word to the left until the most significant bit of the input word is equal to 1.

```
module word_shift(in_word, num_shifts);
  input [7:0] in_word;
  output [7:0] num_shifts;
  assign num_shifts = shift_count(in_word);
  //write the function below
  ….
  ….
  ….
endmodule
```

8.2 **(a)** Write a Verilog function that will create the 2's complement of an *N*-bit vector. Use a call of the form `comp2(bit_vec, N)`, where *bit_vec* is the vector and *N* is the length of the vector. Do the complement on a bit-by-bit basis using a loop. You may declare N as a global parameter in the calling module.
(b) Write a Verilog module that will call the function in part (a).

8.3 The following Verilog code defines a function that computes the factorial of a number recursively. The ***automatic*** keyword is used to allow the function to be called recursively. Fill in the missing code that follows.

```
module factorial_test;

  function automatic integer factorial;
  input [31:0] num;

  /*
         Insert your code to implement factorial
  */

  endfunction

  integer result;

  initial begin
  /*
         Insert your code to compute the factorial of 9
  */
  $display("factorial = %d", result);
  end
endmodule
```

8.4 **(a)** *A* and *B* are bit vectors that represent unsigned binary numbers. Write a Verilog function that returns TRUE (1) if $A > B$. The function call should be of the form `GT(A, B, N)`, where *N* is the length of the bit vectors. *Hint*: start comparing the most significant bits of *A* and *B* first and proceed from left to right. As soon as you find a pair of unequal bits, you can determine whether $A > B$. For example, if $A = 1011010$ and $B = 1010110$, you can determine that $A > B$ when you make the fourth comparison. You may declare N as a global parameter in the calling module.

(b) Write a Verilog module that will call the function in part (a).

8.5 What are the major differences between Verilog functions and Verilog tasks?

8.6 Write a Verilog module that could read numbers from a text file line by line and sort them by a user-defined task. The text file is named as "sort.txt", which contains 10 positive integers. The ten inputs should be stored in a global integer array that is declared as

integer ARRAY [9:0];

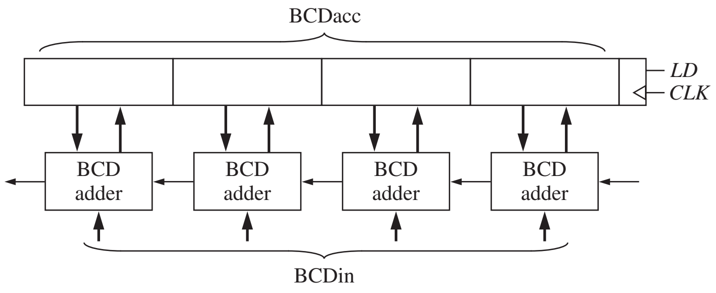
Write a task to sort this global array in ascending order. The sorted array can occupy the same global array. All the temporary space for sorting should be defined inside the task.

8.7 **(a)** Write a Verilog task that counts the number of 1s in an input bit vector that is up to N bits long ($N \leq 31$). The output should be 5 bits long. The task call should have the following form: (N, A,B) where A is the input and B is the output. You may declare N as a global parameter in the calling module.

(b) Write a Verilog module that will call the task in part (a).

8.8 Write a Verilog module that implements a 4-digit BCD adder with accumulator (see the block diagram that follows). If $LD = 1$, then the contents of $BCDacc$ are replaced with $BCDacc + BCDin$.

Write a task that adds two BCD digits and a carry and returns a BCD digit and a carry. Call this task concurrently four times in your code.



8.9 For the following Verilog code, list the values of B and C at each time a change occurs. Include all deltas and stop your listing when time > 8 ns. Assume that B is changed to 0110 at time 5 ns. Indicate the times at which task P1 is called.

```
module Q1(B,C);
  input[3:0] B;
  output[3:0] C;
  integer i;

  task P1;
  input[3:0] A;
  output reg[3:0] D;
  begin
      for(i = 1; i<= 3;i = i+1)
```

```
                D[i] <= A[i-1];
         D[0] <= A[3];
    end
    endtask

    always @(B)
    begin
          P1(B,C);
          #1;
          P1(B,C);
    end

    assign C = B;

  endmodule
```

8.10 Hamming codes are used for error detection and correction in communication and memory systems. Error detection and correction capability is incorporated in these codes by inserting extra bits into the data word. Addition of one parity bit can detect odd number of bit flips, but no error correction is possible with one parity bit. A (7,4) Hamming code has 4 bits of data but 7 bits in total, including the 3 parity bits. It can detect two errors and correct one error. This code can be constructed as follows: If we denote data bits as $d_4d_3d_2d_1$, the encoded code word would be $d_4d_3d_2p_4d_1p_2p_1$, where p_4, p_2, and p_1 are the added parity bits. These bits must satisfy the following conditions for even parity:

$$p_4 = d_2 \text{ XOR } d_3 \text{ XOR } d_4; \ldots\ldots\ldots\ldots(1)$$
$$p_2 = d_1 \text{ XOR } d_3 \text{ XOR } d_4; \ldots\ldots\ldots\ldots(2)$$
$$p_1 = d_1 \text{ XOR } d_2 \text{ XOR } d_4; \ldots\ldots\ldots\ldots(3)$$

When the 7 bits are received/decoded, an error syndrome $S_3S_2S_1$ is calculated as follows:

$$p_4 \text{ XOR } d_2 \text{ XOR } d_3 \text{ XOR } d_4 = S_3; \ldots\ldots\ldots\ldots(4)$$
$$p_2 \text{ XOR } d_1 \text{ XOR } d_3 \text{ XOR } d_4 = S_2; \ldots\ldots\ldots\ldots(5)$$
$$p_1 \text{ XOR } d_1 \text{ XOR } d_2 \text{ XOR } d_4 = S_1; \ldots\ldots\ldots\ldots(6)$$

The syndrome indicates which bit is wrong. For example, if the syndrome is 110, it indicates that bit 6 from right end (i.e., d3) has flipped. If $S_3S_2S_1$ is 000, there is no error.

(a) Is there any error in the code word 0110111? If yes, which bit? What was the original data? What must be the corrected code word?

(b) How will these 6 equations get modified for odd parity? Write the 6 equations for odd parity.

(c) Write a Verilog module for error detection without using tasks. The inputs to the module are the 7-bit encoded data word and the type of parity, and the output is the syndrome. The type of parity is encoded as 0 for odd parity and 1 for even parity.

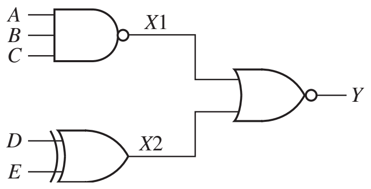
```
module  error_detector(data, PARITY, Syndrome)
{
}
```

(d) Modify the Verilog module in **(c)** to use Verilog tasks. Write required Verilog task(s) that will generate the error syndrome given the code word. The input to the task is data, and output is the syndrome. You have to call necessary task(s) for each type of parity.

```
task  error_p(data, Syndrome)
{

}
```

8.11 Write a Verilog model that uses only built-in primitives to implement the following circuit.



Rise delay of NAND gate is 15 ns.
Rise delay of XOR gate is 14 ns, fall delay of XOR gate is 16 ns.
Rise delay of NOR gate is 12 ns, and fall delay of NOR gate is 14 ns.

8.12 Create a user-defined primitive (UDP) for a J-K flip-flop with asynchronous clear and preset using "?" if needed

8.13 Write a Verilog user-defined primitive (UDP) for generating odd parity for 4-bit input data.

8.14 A Verilog module has inputs *A* and *B* and outputs *C* and *D*. *A* and *B* are initially high. Whenever *A* goes low, *C* will go high 5 ns later, and if *A* changes again, *C* will change 5 ns later. *D* will change if *B* does not change for 3 ns after *A* changes. *Note*: The timing checks should be done inside: specify …. Endspecify

(a) Write the Verilog module with an always block that determines the outputs *C* and *D*.
(b) Write another always block to check that *B* is stable 2 ns before and 1 ns after *A* goes high. The always block should also report an error if *B* goes low for a time interval less than 10 ns.

8.15 Write a Verilog module of an address decoder/address match detector. One input to the address decoder is an 8-bit address, addr. The second input is the 6-bit vector check. The address decoder will output *Sel* = 1 if the upper 6 bits of the 8-bit address match the 6-bit check vector. For example, if *addr* = 10001010 and *check* = 1000XX, then *Sel* = 1. Only the 6 leftmost bits of *addr* will be compared; the remaining bits are ignored. An *X* in the check vector is treated as a don't care.

8.16 Write a Verilog module for one flip-flop in a 74HC374 (octal D-type flip-flop with 3-state outputs. Given the D-type flip-flop setup time = 15 ns, hold time = 5 ns, pulse width time = 15 ns). Assume that all logic values are *x*, 0, 1, or *z*. Check setup, hold, and pulse-width specs using monitor statements. Unless the output is *z*, the output should be *x* if *CLK* or *OC* is *x*, or if an *x* has been stored in the flip-flop. *Note*: The timing checks should be done inside: specify …. Endspecify

8.17 Write a Verilog function to compare two 8-bit vectors to determine whether they are equal. Report an error if any bit of either vector is not 0, 1, or *z*. The function call should pass only the vectors. The function should return TRUE (1) if the vectors are equal, else FALSE (0). All bits including *z*s should match (i.e., *z* only matches *z*).

8.18 In the following code, all signals are 1-bit. Draw a logic diagram that corresponds to the code. Assume that a D flip-flop with CE is available.

```
assign F = (EA == 1) ? A: ((EB == 1)? B : Z);
always @(posedge CLK)
begin
 if(Ld == 1)
       A <= B;
 if(Cm == 1)
       A <= ~A;
end
```

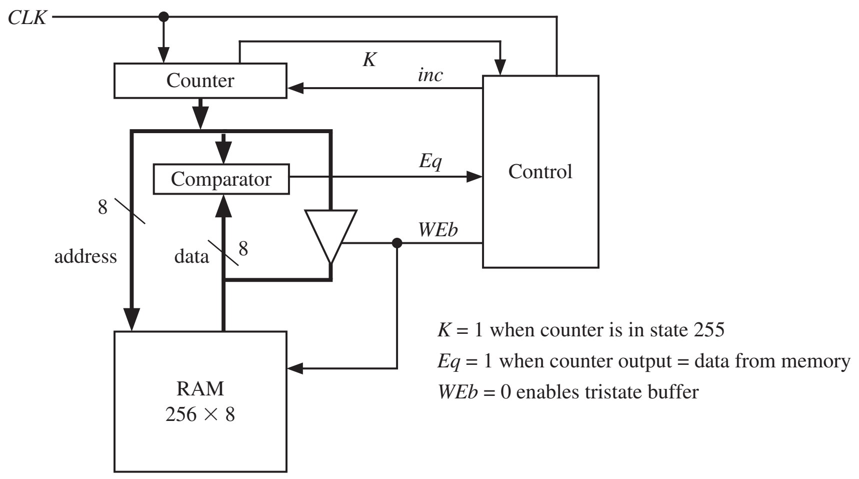
8.19 Design a memory-test system to test the first 256 bytes of a static RAM memory. The system consists of simple controller, an 8-bit counter, a comparator, and a memory as shown subsequently. The counter is connected to both the address and data (IO) bus so that 0 will be written to address 0, 1 to address 1, 2 to address 2, . . ., and 255 to address 255. Then the data will be read back from address 0, address 1, . . ., address 255 and compared with the address. If the data does not match, the controller goes to the fail state as soon as a mismatch is detected; otherwise, it goes to a pass state after all 256 locations have been matched. Assume that $OE\_b = 0$ and $CS\_b = 0$.

(a) Draw an SM chart or a state graph for the controller (five states). Assume that the clock period is long enough so that one word can be read every clock period.

(b) Write Verilog code for the memory-test system.

8.20 Design a memory-test system similar to that of Problem 8.18, except write a checkerboard pattern into memory (01010101 into address 0, 10101010 into address 1, etc.). Draw the block diagram and the SM chart.

8.21 Design a memory tester that verifies the correct operation of a 6116 static RAM (Figure 8-15). The tester should store a checkerboard pattern (alternating 0s and 1s


CLK
Counter
K
inc
Comparator
Eq
Control
8
address
data
8
WEb
RAM
256 × 8
K = 1 when counter is in state 255
Eq = 1 when counter output = data from memory
WEb = 0 enables tristate buffer

in the even addresses, and alternating 1s and 0s in the odd addresses) in all memory locations and then read it back. The tester should then repeat the test using the reverse pattern.

(a) Draw a block diagram of the memory tester. Show and explain all control signals.
(b) Draw an SM chart or state graph for the control unit. Use a simple RAM model and disregard timing.
(c) Write Verilog code for the tester and use a test bench to verify its operation.

8.22 A clocked T flip-flop has propagation delays from the rising edge of *CLK* to the changes in *Q* and *Q*' as follows: if *Q* (or *Q*') changes to 1, $t_{plh} = 8$ ns, and if *Q* (or *Q*') changes to 0, $t_{phl} = 10$ ns. The minimum clock pulse width is $t_{ck} = 15$ ns, the setup time for the *T* input is $t_{su} = 4$ ns, and the hold time is $t_h = 2$ ns. Write a Verilog model for the flip-flop that includes the propagation delay and that reports if any timing specification is violated. Write the model using parameters with default values. *Note*: The timing checks should be done inside: specify …. Endspecify

8.23 **(a)** Write a model for a D flip-flop with a direct clear input. Use the following timing parameters: t_{plh}(10 ns), t_{phl}(10 ns), t_{su}(5 ns), t_h(3 ns), and t_{cmin}(20). The minimum allowable clock period is t_{cmin}. Report appropriate errors if timing violations occur. *Note*: The timing checks should be done inside: specify …. Endspecify
(b) Write a test bench to test your model. Include tests for every error condition.

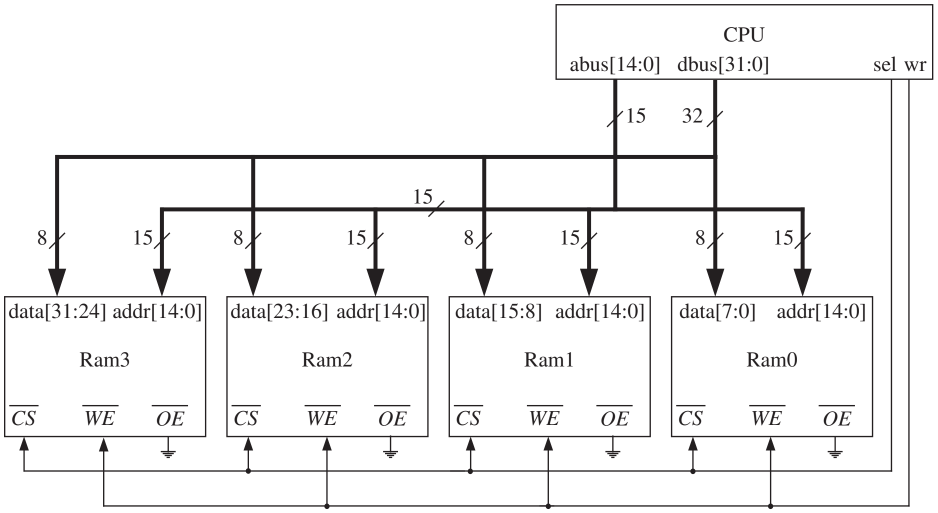
8.24 Write a Verilog model for an *N*-bit comparator using an iterative circuit. In the module, use the parameter *N* to define the length of the input bit vectors *A* and *B*. The comparator outputs should be $EQ = 1$ if $A = B$, and $GT = 1$ if $A > B$. Use a for loop to do the comparison on a bit-by-bit basis, starting with the high-order bits. Even though the comparison is done on a bit-by-bit basis, the final values of *EQ* and *GT* apply to *A* and *B* as a whole.

8.25 Four RAM memories are connected to CPU busses as shown here. Assume that the following RAM component is available.

```
module SRAM(cs-b, we-b, oe-b, address, data);
input cs-b,we-b,oe-b;
input[14:0] address;
inout[7:0] data;

endmodule
```

Write a Verilog code segment that will connect the four RAMs to the busses. Use a **generate** statement and a named association.



8.26 Write structural Verilog code for a module that is an *N*-bit serial-in, serial-out right-shift register. Inputs to the shift register are bit signals: *SI* (serial input), *Sh* (shift enable), and *CLK*. Your module should have a **generate** statement. Assume that a component for a D flip-flop with clock enable (*CE*) is available.

8.27 Write structural Verilog code for a module that has two inputs: an *N*-bit vector *A*, and a control signal *B* (1 bit). The module has an *N*-bit output vector, *C*. When $B = 1$, $C <= A$. When $B = 0$, *C* is all 0s. Use parameter to specify the value of *N* (default = 4). To implement the logic, use a **generate** statement that instantiates *N* 2-input AND gates.

8.28 The structural Verilog code that follows is a 2-input NOR gate with the rise/fall time defined as parameters.

```
module NOR2(a, b, c);
  parameter Trise = 3;
  parameter Tfall = 2;
```

```
    parameter load = 1;

    input a, b;
    output reg c;

    wire nor_value;

    assign nor_value = ~(a | b);

    always @(nor_value)
    begin
       if(nor_value == 1'b1)
          #(Trise + 3*load) c = 1'b1;
       else
          #(Tfall + 3*load) c = 1'b0;
    end
    endmodule

    module NOR2_TEST(in1, in2, in3, in4, out1, out2);
    input in1, in2, in3, in4;
    output out1, out2;

    NOR2 U1 (in1, in2, out1);

    /*
                     place for your new code
    */

  endmodule
```

(a) Instantiate NOR2 (U2) by using the parameter map method (Trise = 5, Tfall = 4, load = 3).
(b) Use the **defparam** to pre-define the timing values and instantiate NOR2 (U3), which has rise time = 4, fall time = 3, and load = 2.
(c) What are the rise time, fall time, and fan-out time of U1?
(d) What are the rise and fall delays of NOR2 (U2)?

8.29 Create a 4×4 array multiplier using **generate** statements. Use full adder, half adder, and AND gate components as in Chapter 4.

8.30 *B* is an integer array declared as integer B [4:0]. Write a Verilog code segment that will read five integers in a line of text from a file named “FILE2” and then write the five integers into array *B*.

8.31 Write a task that has an integer signal and a file name as parameters. Each line of the file contains a delay value and an integer. The task reads a line from the file, waits for the delay time, assigns the integer value to the signal, and then reads the next line. The task should return when end-of-file is reached.

8.32 Write a task that logs the history of values of a bit vector signal to a text file. Each time the signal changes, write the current time and signal value to the file. Verilog has a built-in funtion called `$time` that could display the current simulation time.

8.33 Indicate the final decimal value of the regA and regB after executing the following instructions.

```
reg[8:0] regA;
reg[8:0] regB;

regA <= $unsigned -9;
regB <= $signed 9'b100000000;
```

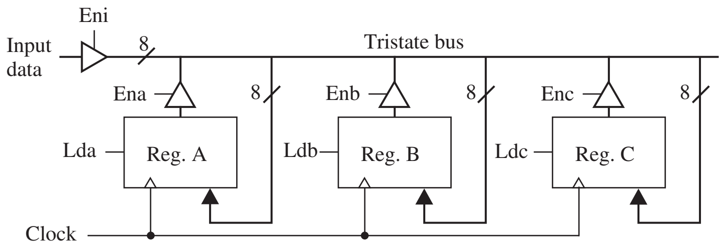
8.34 **(a)** Complete the following code by defining a macro *Sum*. Use parameters to initialize the numbers to 9 and 11.

```
always @(posedge clk)
begin
   $display("The sum of %d and %d is %d", A, B, Sum(A,B));
   $stop;
   $display("This sentence should be on the screen");
end
```

(b) What will the completed code print on the screen when it executes?

8.35 What is the difference between **$stop** and **$finish**?

8.36 Based on the tri-state bus circuit (Figure 1-56), fill out the table that follows by using 4-valued logic. The values stored in RegA, RegB, and RegC are 8'd5, 8'd10, and 8'd15, respectively. Assume Eni is equal to 0.



Time	Enc	Ena	Enb	Bus
0	Z	Z	Z	z
2	1	0	0	
4	0	1	0	
6	0	0	1	
8	1	1	1	
10	1	1	1	

CHAPTER 9

Design of a RISC Microprocessor

A microprocessor is an example of a complex digital system. In this chapter, we will describe a microprocessor from MIPS Technologies, the MIPS R2000, and implement a subset of the MIPS processor's instruction set architecture (ISA). The term instruction set architecture denotes the instructions that are visible to the assembly language programmer—the number of registers, the addressing modes, and the operations (opcodes) available in the particular processor. An introduction to the RISC philosophy is presented first followed by a description of the MIPS ISA. The arithmetic, memory access, and control transfer instructions of the MIPS are presented along with a design to implement a subset of the ISA. A synthesizable Verilog model for the MIPS subset is then presented. Use of a test bench for testing the design is illustrated.

9.1 The RISC Philosophy

Many early microprocessors, such as the Intel 8086 and Motorola 68000, incorporated a variety of powerful instructions and addressing modes. A natural consequence of this was the complexity of the design, especially the control unit complexity. These microprocessors included a microprogrammed control unit because it was difficult to design and debug a hardwired control unit for such complex digital systems. (See Chapter 5 for tradeoffs between microprogramming and hardwiring.)

The value of simplicity became clearer in the late 1970s and early 1980s. The result was the advent of **RISC** or the reduced instruction set computing philosophy. RISC processors are a type of microprocessor that uses a small and simple set of instructions rather than a variety of complex instructions and versatile addressing modes. The first RISC projects came from IBM, Stanford, and UC–Berkeley in the late 1970s and early 1980s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy, which has become known as RISC. In contrast, earlier processors such as the Intel 8086 and the Motorola 68000/68020 started to be called **CISC** or complex instruction set computing processors after the advent of the RISC philosophy. The first generation of RISC processors included MIPS R2000 from MIPS, SPARC from Sun Microsystems, and RS/6000 from IBM. The IBM RS/6000 has evolved into the POWERPC and POWER architecture.

MIPS: MIPS Technologies is a computer manufacturer that has designed and sold several RISC microprocessors starting with the MIPS R2000 processor in the 1980's. The term MIPS was commonly known to computer designers as a performance metric, the Millions of Instructions Per Second metric. The MIPS in the name of the MIPS Corporation however does not stand for that. The original acronym stood for Microprocessor without Interlocked Pipelined Stages. In a pipelined processor, there must exist a mechanism to enforce dependencies between instructions. So, if one instruction needs the result of the previous one, the second instruction should not proceed until the first instruction's result is ready. Enforcing of this type of dependency is usually done by hardware. The first MIPS processor, however, did not have hardware interlocks. It reflected the early RISC idealism that anything that can be done in software should be done in software. In early MIPS processors, pipeline interlocks were implemented by software by inserting the appropriate number of nop (no operation) instructions between the dependent instructions.

Certain design features have been characteristic of most RISC processors:

- **Uniform instruction length:** All instructions have the same length—32 bits. This is in sharp contrast to previous microprocessors that contained instructions as small as a byte and as large as 16 bytes.
- **Few instruction formats:** The RISC ISAs emphasized having as few instruction formats as possible and encoding the different fields in the instruction as uniformly as possible. This greatly simplifies instruction decoding.
- **Few addressing modes:** Most RISC processors support only one or two memory-addressing modes. Addressing modes offer different ways an instruction can indicate the memory address to be accessed. Examples are direct addressing, immediate addressing, base-plus-offset addressing, based ndexed addressing, indirect addressing, among others. Many RISC processors support only one addressing mode. Typically, this addressing mode specifies addresses with a register and an offset.
- **Large number of registers:** The RISC design philosophy generally incorporates a larger number of registers to prevent the loss of performance by frequently accessing memory. RISC processors are also often called **register-register architectures**. All arithmetic operations operate on register operands. CISC architectures typically contained 8 or 12 registers, whereas most RISC architectures contained 32 registers.
- **Load/store architecture:** RISC architectures are also called load/store architectures. The key idea is the absence of arithmetic instructions that directly operate on memory operands (i.e., arithmetic instructions that take one or more operands from memory). The only instructions that are allowed to access memory are load and store instructions. The load instructions bring the data to registers, and arithmetic operations operate on the data in the registers. These

architectures are also called register-register architectures because input and output operands for computation operations are in registers. A load/store architecture inherently means that it is also a register-register architecture.

- **No implied operands or side effects:** Most earlier ISAs contained implied operands, such as accumulators or implied results (**side effects**), such as flags (condition codes), to indicate such conditions as carry, overflow, negative, and the like. Implied operands and side effects can cause difficulties and challenges in pipelined and parallel implementations. A principle behind RISC architectures is to have minimal implied operands/operations and side effects.

The RISC philosophy has been to adhere to these features and embrace simplicity of design. The terms RISC and CISC are used very often as antonyms, but perhaps it is not clear how “reduced” is the opposite of “complex”. It is not even clear that RISC processors have a smaller instruction set than prior CISC processors. Some RISC ISAs have 100+ instructions whereas some CISC processors have only 80 instructions. However, these 80 CISC instructions could assume several addressing modes. A CISC processor, the Motorola 68020 supported up to 20 different addressing modes. Considering all the different forms an instruction could take, most RISC ISAs do contain fewer instructions than CISC ISAs. The key point in the RISC philosophy has been its emphasis on simplicity: having only simple basic operations, simplifying instruction formats, reducing the number of addressing modes, and eliminating complex operations. This computing paradigm could have been called simple instruction set computing (SISC), but SISC sounds like CISC.

CISC architectures are not without advantages. Instruction encoding is denser in CISC than in RISC. The fixed instruction width in RISC leads to using more bits than necessary for some instructions. In CISC ISAs, every instruction is just as wide as it needs to be. Hence, code size is smaller in the CISC case. If instruction memory size has to be kept small, as in embedded environments, CISC ISAs have an advantage.

Most modern microprocessors have RISC ISAs. Some examples are the MIPS R12000, Sun UltraSPARC, IBM PowerPC, HP PA-RISC, among others. The Pentium 4 or the x86 processors in general are examples of modern processors with a CISC ISA. (The term x86 is used to refer to the different processors that have used the ISA that originated with Intel 8086. This list includes Intel 8086, 80286, 80386, 80486; Pentium and AMD K5; K6; Opteron; and others.)

Whether RISC or CISC is better was a topic of intense debate in the 1980s and 1990s. It has now become understood that decoding and processing is easy with a RISC ISA; however, it also has been shown that hardware can translate complex CISC-style instructions into RISC-style instructions and process them. Pentium 4 and other current high-end x86 processors have a CISC ISA; however, they use hardware to convert each CISC instruction to one or more RISC-type instructions or microoperations (called **uops** or **R-ops**) that can be pipelined easily. In spite of all the arguments that have taken place, there is no disagreement about the ease of implementation of RISC ISAs.

The MIPS instruction set architecture is one of the earliest RISC ISAs and one of the simplest ones. It has only one memory-addressing mode. In contrast, another early RISC architecture, the SPARC, has two memory-addressing modes.