DIGITAL SYSTEM DESIGN USING VERILOG

COURSE CODE: 19EC5DCDSV (3 CREDITS)

Faculty In-Charge: Dr. Dinesha P

drdinesh-ece@dayanandasagar.edu

- Verilog provides two operators for shifting the bits of an unsigned value. The operator <<performs a logical shift left, and the >> operator performs a logical shift right.
- For example, if the unsigned net or variable s has the value 00010011 (19),

S<<2 after shift

00010011

After 1 shift, 00100110

After 2 shift 01001100 (76)

S>>2 after shift

00010011

After 1 shift, 00001001

After 2 shift 00000100 (4)

- In Verilog, we can apply the and <<< and >>> operators to signed operands. The <<< operator, like the << operator, performs a logical shift left, but the >>> operator performs an arithmetic shift right.
- For example, if the signed net or variable S has the value 11110011, representing the value -13, the Verilog expression

S<<<2

11110011(-13)

After 1 shift, 11100110

After 2 shift 11001100 (-52)

s>>>2

11110011

After 1 shift, 11111001

After 2 shift 11111100 (-4)

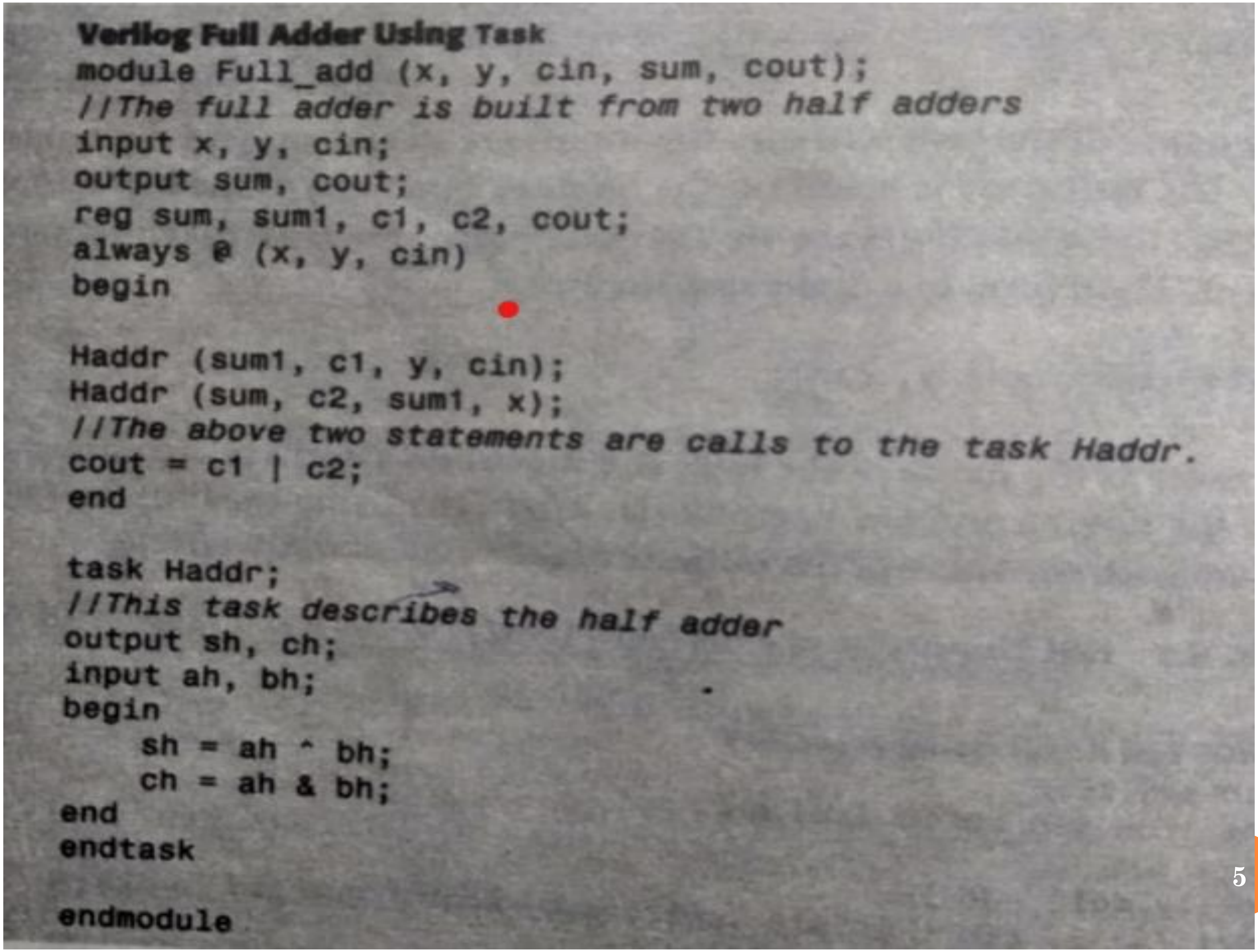
Additional Topics in Verilog:
Verilog Tasks:

- Tasks are Verilog subprograms. They can be implemented to execute specified routines repeatedly.
- Task has two parts declaration and body. In declaration, the name of task is specified and the inputs and outputs are listed.

```
Task addr;
Output c, d;
Input a, b;
```

- In body of the task cannot include always or initial. A task must be called within the behavioral statements always or initial.

```
begin
C= a^b;
……….
end
endtask
```



Verilog Full Adder Using Task

```
module Full_add (x, y, cin, sum, cout);
//The full adder is built from two half adders
input x, y, cin;
output sum, cout;
reg sum, sum1, c1, c2, cout;
always @ (x, y, cin)
begin

Haddr (sum1, c1, y, cin);
Haddr (sum, c2, sum1, x);
//The above two statements are calls to the task Haddr.
cout = c1 | c2;
end

task Haddr;
//This task describes the half adder
output sh, ch;
input ah, bh;
begin
     sh = ah ^ bh;
     ch = ah & bh;
end
endtask

endmodule
```

Example:

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
```

Verilog function:

- Functions are behavioural statements.
- 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.
- Functions have declaration statement and a body.
- In the declaration type and name of the output are specified, as well as the names and sizes of the inputs.
- A Verilog function is similar to a Verilog task. 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.

Example:
```
Function exp;
Input a, b;
```

Declares a function with a name exp. The function has two inputs, a and b, and one output exp.

The general form of a function declaration is

```
function function-name
input [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.

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.

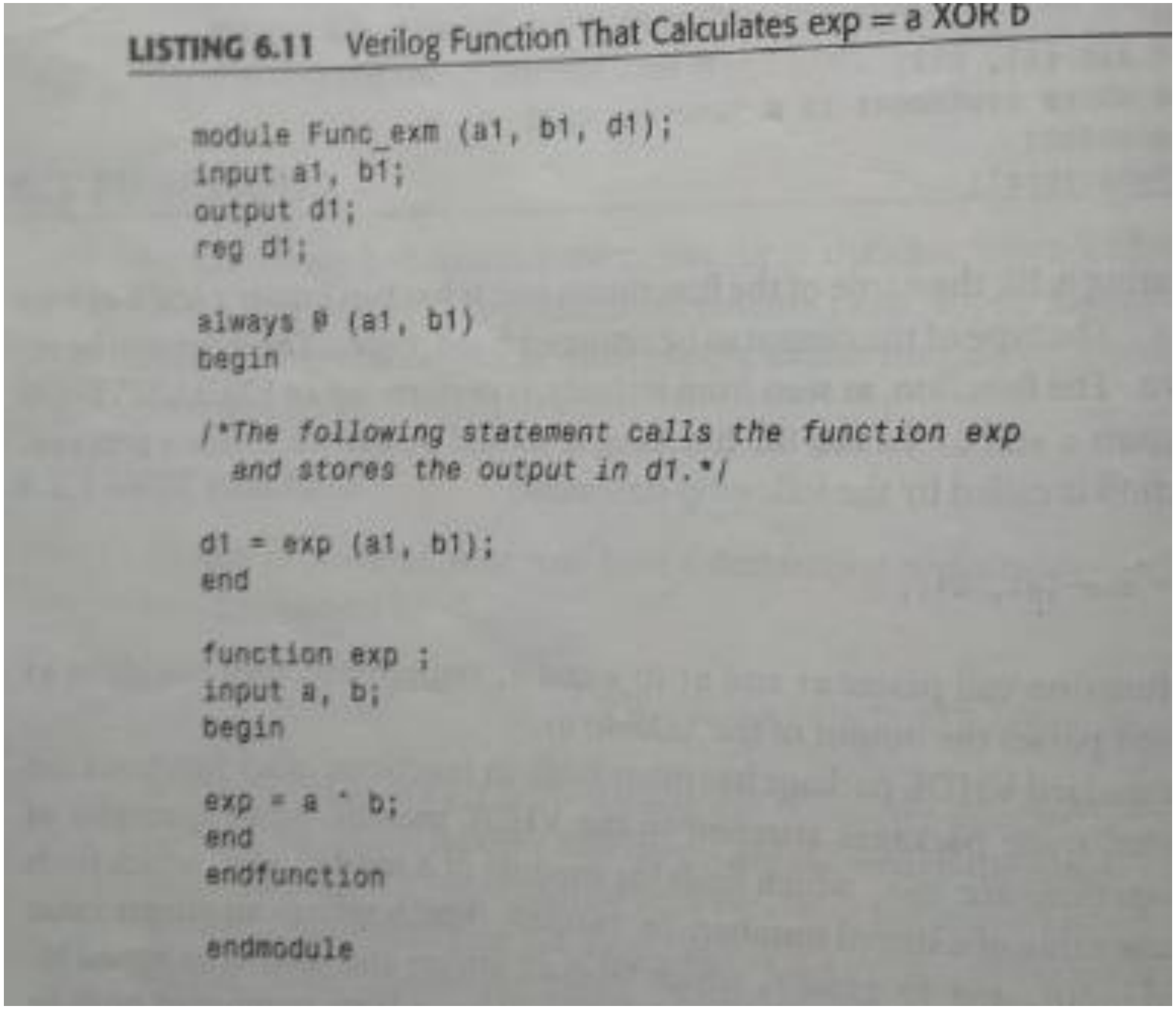
Parity Generation Using a Function:

```
// 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
```



LISTING 6.11 Verilog Function That Calculates exp = a XOR b

```
module Func_exm (a1, b1, d1);
input a1, b1;
output d1;
reg d1;

always @ (a1, b1)
begin

/*The following statement calls the function exp
  and stores the output in d1.*/

d1 = exp (a1, b1);
end

function exp ;
input a, b;
begin

exp = a ^ b;
end
endfunction

endmodule
```

Example: Add Function (illustrates a function using a for loop)

```
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 following example 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.

```
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
```

Comparisons of task and function:

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

Multivalued Logic and Signal Resolution:

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:

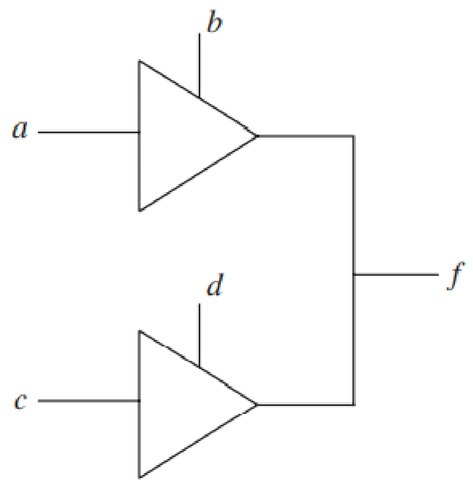
• X	Unknown
• 0	0
• 1	1
• Z	High impedance

The high impedance state is used for modeling tristate buffers and buses.

The 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.



The tri-state 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.

Verilog Code for Tri-stated Buffers:
(a) Tristate module with always statements

```
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
```

(b) Tristate module with assign 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
```

The operation of a tri-state 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*

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.

Built-in Primitives:

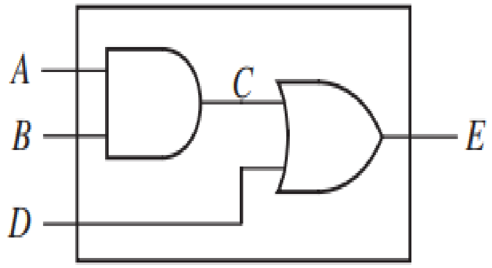
Verilog allows modeling at switch level and has several predefined primitives for that. It also has predefined gate-level primitives with drive strength among other things. There are 14 predefined logic gate primitives and 12 predefined switch primitives to provide the gate and switch-level modeling 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) (ii) There are primitives such as the bidirectional transfer gate that cannot be otherwise modeled using continuous assignments

The Verilog module of Figure shown is defined 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);



```
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 provides built-in primitives for tri-state 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 tri-state control input. An array of four inverting tri-state buffers can be created as shown

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);
```

Verilog Tristate Buffer Module Using Built-In Primitives:

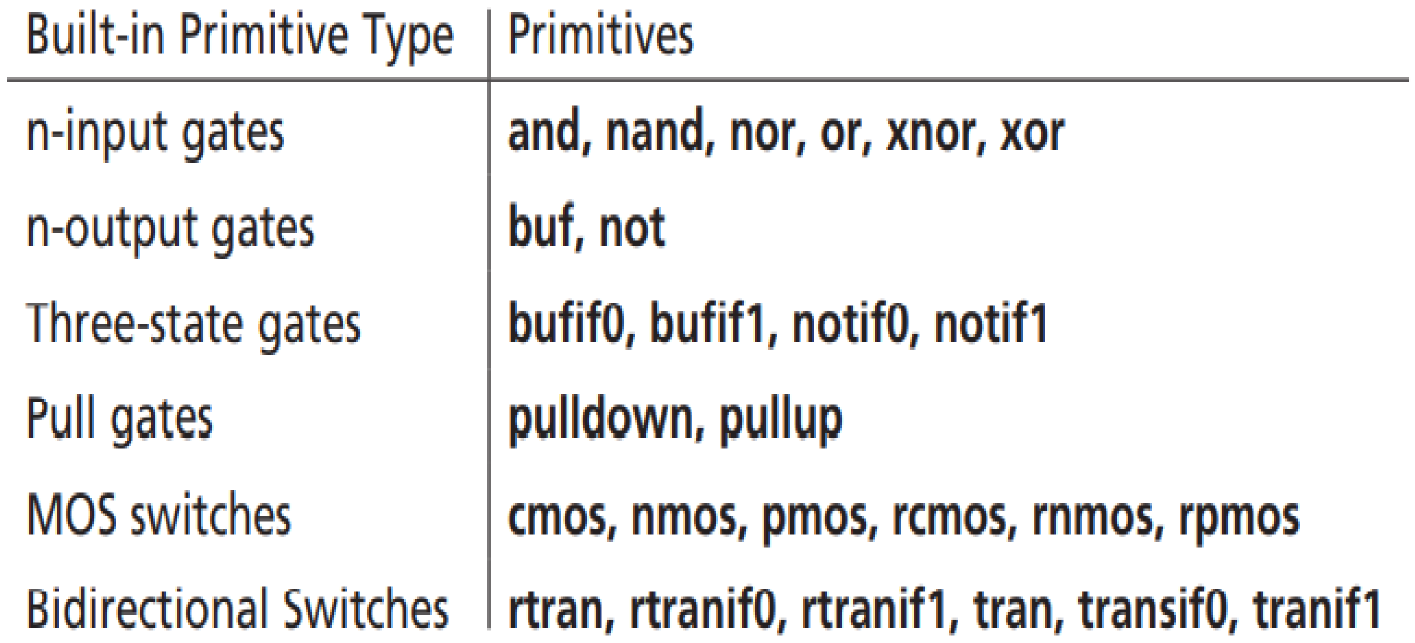
```
module tri_buffer (a,b,c,d,f);
   input a,b,c,d;
   output f;

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

endmodule
```

Table shows the lists of the built-in primitives in Verilog.

The first 12 are gate-level primitives, and the rest are switch-level primitives.



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**

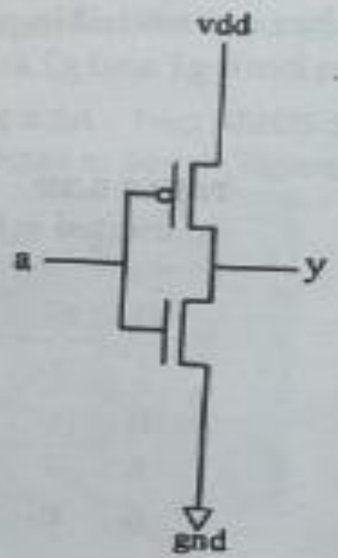
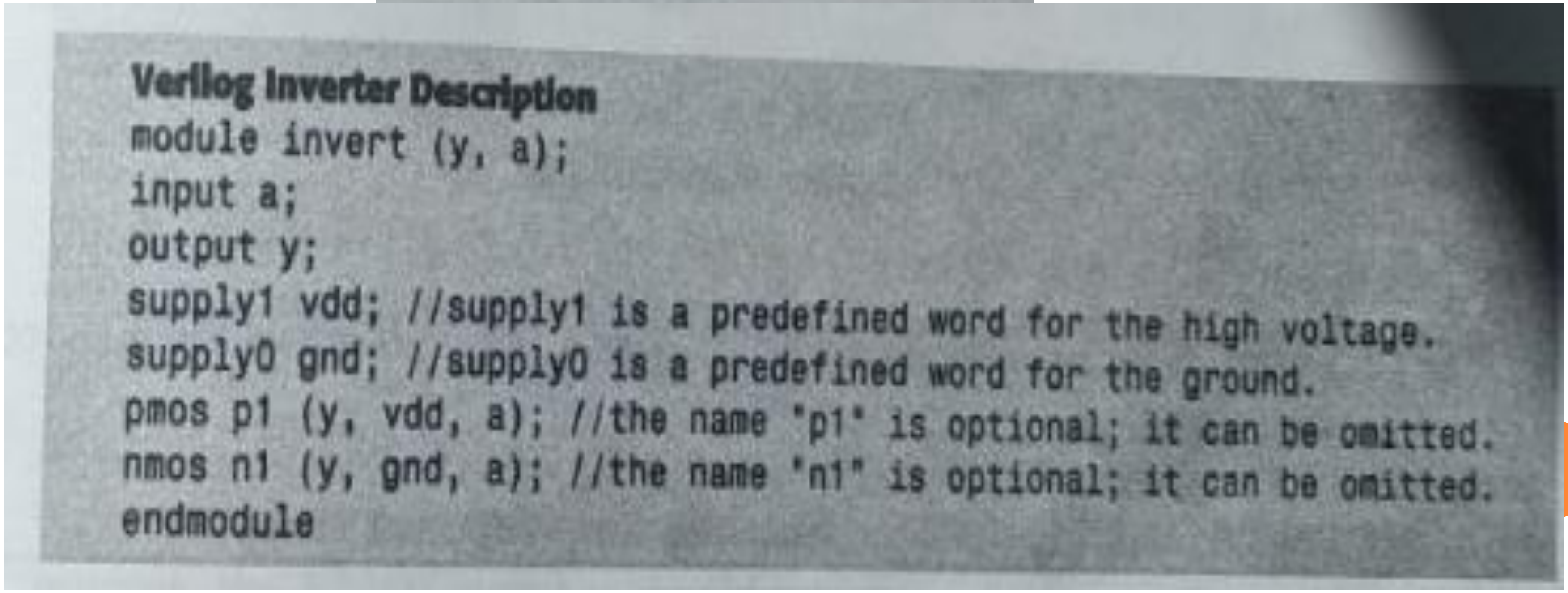


FIGURE 5.3 An inverter.



Verilog Inverter Description

```
module invert (y, a);
input a;
output y;
supply1 vdd; //supply1 is a predefined word for the high voltage.
supply0 gnd; //supply0 is a predefined word for the ground.
pmos p1 (y, vdd, a); //the name "p1" is optional; it can be omitted.
nmos n1 (y, gnd, a); //the name "n1" is optional; it can be omitted.
endmodule
```

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

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

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 delay512
 bufif0 #(10,12,11) b3 (out, in, ctrl); // rise, fall, and
                                        // turn-off delays
```

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

- 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.

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.

Each row of the table in the UDP is terminated by a semicolon.

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
```

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
```

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 Cin input, Co[0] to the Cout 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.

Alternatively, 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], Cout 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.

Generate Statements:

- We know that, 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 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 *Cin* is the same as C(0) and Cout 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.
- A generate statement itself is defined to be a concurrent statement, so nested generate statements are allowed.

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
```

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. Example of system tasks are **$display**, **$monitor**, and **$write**.
- These system functions are not for synthesis, however; they are intended for convenience during simulation.
- System 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.

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.

Simulation Control Tasks:

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
```

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
```

File I/O Functions:

The ability to handle files and text is very valuable while testing large Verilog designs.

Files are frequently used with test benches to provide a source of test data and to provide storage for test results.

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
```

To close an opened file, the $fclose function can be used.

```
$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.
$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.
$fgetc	reads a single character from the specified file and returns it. If the end-of-file is reached, $fgetc returns EOF.
$fscanf	parses formatted text from a file according to the format and writes the results to args.

$fprintf	writes a formatted string to a file.
$fread	reads binary data from the file specified by the file descriptor into a register or into a memory.
$fwrite	writes binary data from a register to the file specified by the file descriptor.
$readmemb **$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.

- The following Verilog code illustrates the read 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.

```
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
```

- The following Verilog code illustrate the read a file with commands, addresses, and data values. The code uses $fopen for accessing the file and $fscanf for parsing each line. Also, 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.

```
  `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
```

Compiler Directives:

Verilog has several compiler directives, All Verilog compiler directives are preceded by the (`) character. This character is called grave accent (ASCII 0x27). There is no semicolon (;) at the end of the line with the compiler directive

Example: `define, `include, `ifdef

`define: 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.

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.

`include:

This compiler directive is used to insert the contents of one source file into another file during compilation.

The `include compiler directive can be used to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries.

The syntax of the ‘include directive is as follows

‘include “filename”

Where the filename is the name of the file to be included.

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
```

`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.

The `ifdef compiler directive has the following form

```
  `ifdef macroname
 ifdef_group_of _lines
```