CHAPTER 2 Verilog

OBJECTIVES :

The learning outcome is that student is able to:

- *explain the hierarchy and modelling structures of Verilog,*
- *identify the different types of net and variables,*
- *distinguish the usage of concurrent and procedural statements and*
- *design digital electronic systems using sub-circuits and functions.*

1 Introduction

In the 1980s rapid advances in integrated circuit technology lead to efforts to develop standard design practices for digital circuits. Verilog was produced as a part of that effort. The original version of Verilog was developed by Gateway Design Automation, which was later acquired by Cadence Design Systems. In 1990 Verilog was put into the public domain, and it has since become one of the most popular languages for describing digital circuits. In 1995 Verilog was adopted as an official IEEE Standard, called 1364-1995. An enhanced version of Verilog, called Verilog 2001, was adopted as IEEE Standard 1364-2001 in 2001. While this version introduced a number of new features, it also supports all of the features in the original Verilog standard.

Verilog was originally intended for simulation and verification of digital circuits. Sub-sequently, with the addition of synthesis capability, Verilog has also become popular for use in design entry in CAD systems. The CAD tools are used to synthesize the Verilog code into a hardware implementation of the described circuit.

Verilog allows the designer to describe a desired circuit in a number of ways. One possibility is to use Verilog constructs that describe the structure of the circuit in terms of circuit elements, such as logic gates. A larger circuit is defined by writing code that connects such elements together. This approach is referred to as the *structural* representation of logic circuits. Another possibility is to describe a circuit more abstractly, by using logic expressions and Verilog programming constructs that define the desired behaviour of the circuit, but not its actual structure in terms of gates. This is called the *behavioural* representation.

Verilog is a complex, sophisticated language. Learning all of its features is a daunting task. However, for use in synthesis only a subset of these features is important. In this module, it will cover only the necessary commands needed to write codes for designing simple digital logic systems.

2 Hierarchical Design in Verilog

Verilog allows hierarchical design as shown in Figure 1. The Verilog structures which build the hierarchy are: *modules* and *ports*.

A Verilog **model** is composed of **modules**. A module is the basic unit of the model, and it may be composed of **instances** of other modules. A module which is composed of other module instances is called a *parent module*, and the instances are called *child modules*. Ports are input and output that allows data to flow into and out of modules.

In Figure 1, there are four modules: *system*, *comp_1*, *comp_2*, and *sub_3*. System is the parent of comp_1 and comp_2, and comp_2 is the parent of sub_3. comp_1 and comp_2 are the children of system, and sub_3 is the child of comp_2.

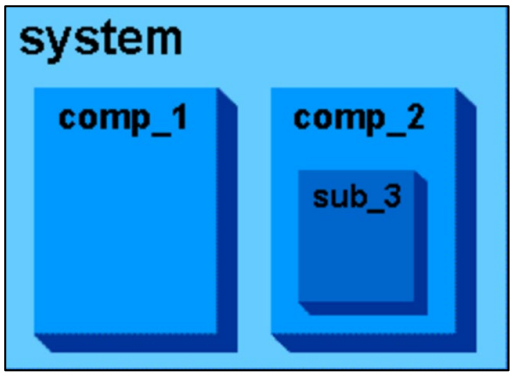

Figure 1. Hierarchical Design.

3 Verilog syntax and constructs

Just like any programming language, one needs to know the syntax of writing Verilog program. Verilog code is *case sensitive*. Documentation can be included in Verilog code by writing a comment. A short comment begins with the double slash, // and continues to the end of the line. A long comment can span multiple lines and is contained inside the delimiters /* and*/ . Examples of comments are:

/ / This is a short comment
/*This is a long Verilog comment that spans more than ……..one line */

White space characters, such as SPACE and TAB, and blank lines are ignored by the Verilog compiler. Multiple statements can be written on a single line. However, this is not recommended as it makes the code hard to read. ; (semicolon) is the end of a command.

Identifiers are the names of variables and other elements in Verilog code. The rules for specifying identifiers are simple: any letter or digit may be used, as well as the _ (underscore) and $ characters. There are two caveats: an identifier must begin with an alphabet or underscore only and an identifier should not be a Verilog keyword. In this notes, Verilog keyword will be in **bold**.

For special purposes Verilog allows a second form of identifier, called an *escaped identifier*. Such identifiers begin with the \ (backslash) character, which can then be followed by any printable ASCII characters except white spaces. Examples of escaped identifiers are \123, \sig-name, and \a+b. Escaped identifiers should not be used in normal Verilog code; they are intended for use in code produced automatically when other languages are translated into Verilog.

3.1 Verilog module and port

The general syntax for a module is

module module_name [(port_name{, portname })];
[**parameter** declarations]
[**input** declarations]
[**output** declarations]
[**inout** declarations]
[**wire** or **tri** declarations]
[**reg** or **integer** declarations]
[**assign** continuous assignments]
[**initial** block]
[**always** blocks]
[gate instantiations]
[module instantiations]
endmodule

In this notes, bold word are keyword, [anything in square bracket] are optional and { anything in braces } are one or more additional entries.

Figure 2 shows a behavioural code for a full adder with module named fulladd, three input ports Cin, x and y respectively and two output s and Cout. The rest of the command in Figure 2 will be covered later on.

3.2 Data types

Verilog has just a few basic data types that it deals with. These data types are built-in, and cannot be changed or added to, as in some other languages. The data types are *scalar*, *vector*, *integer*, *real* and *time*.

```
// Full adder
module fulladd (Cin, x, y, s, Cout);
   input Cin, x, y;
   output reg s, Cout;

   always @(Cin, x, y)
   begin
      case ( {Cin, x, y} )
         3'b000: {Cout, s} = 'b00;
         3'b001: {Cout, s} = 'b01;
         3'b010: {Cout, s} = 'b01;
         3'b011: {Cout, s} = 'b10;
         3'b100: {Cout, s} = 'b01;
         3'b101: {Cout, s} = 'b10;
         3'b110: {Cout, s} = 'b10;
         3'b111: {Cout, s} = 'b11;
      endcase
   end

endmodule
```

Figure 2. Behavioural code for Full adder.

3.2.1 Scalar data type

A scalar quantity is a single bit. The value of this bit can be one of:

0 - logic zero (false)
1 - logic one (true)
x - unknown value
z - high-impedance value

The z and x values can also be denoted by the capital letters Z and X. The value x can be used to denote a don't-care condition; the symbol ? can also be used for this purpose.

3.2.1 Vector data type

The Verilog word for a multi-bit quantity is a vector. If arithmetic is performed on a vector, the vector value is considered to be an unsigned integer. The value of a vector variable is specified by giving a constant of the form

[size]['radix]constant

where size is the number of bits in the constant, and radix is the number base. Supported radices are d(decimal - default), b(binary), h(hexadecimal) and o(octal). If size specifies

more bits than are needed to represent the given constant, then in most cases the constant is padded with zeros. The exceptions to this rule are when the first character of the constant is either x or z, in which case the padding is done using that value. Figure 3 shows some examples of constants in Verilog.

0	the number 0
10	the decimal number 10
'b10	the binary number $10 = (2)_{10}$
'h10	the hex number $10 = (16)_{10}$
4'b100	the binary number $0100 = (4)_{10}$
4'bx	an unknown 4-bit value xxxx
8'b1000_0011	_ can be inserted for readability
8'hfx	equivalent to 8'b1111_xxxx

Figure 3. Examples of constant in Verilog.

3.2.3 Integer data type

An integer is a signed, 2-s complement value. It is declared in an integer statement, and it is a register (see Section 3.4.2).

3.2.4 Real data type

Real values are contained in registers which are declared as real.

3.2.5 Time data type

Time values are 64-bit, unsigned integers. These values are contained in registers which have been declared using the time keyword in the declaration.

3.3 Parameters

A parameter associates an identifier name with a constant. Examples:

```
parameter n = 4;
parameter S0= 2'b00, Sl = 2'b01;
```

Then the identifier n can be used in place of the number 4, the name S0 can be substituted for the value 2'b00, and so on.

3.4 Signals

In Verilog, a signal in a circuit is represented as a *net* or a *variable* with a specific type. A net or variable declaration has the form

type [range] signal_name{, signal_name};

Without the range field the declared net or variable is scalar and represents a single-bit signal. The range is used to specify vectors that correspond to multibit signals.

3.4.1 Nets

A net represents a node in a circuit. To distinguish between different types of circuit nodes there exist several types of nets, called *wire*, *tri*, and a number of others that are not needed for synthesis, and are not covered in this module.

The wire type is employed to connect an output of one logic element in a circuit to an input of another logic element and can only be used to model combinational logic. The following are examples of scalar wire declarations.

wire x;
wire Cin, AddSub;

A vector wire represents multiple nodes, such as

wire [3:0] S;
wire [1:2] sum;

The square brackets are the syntax for specifying a vector's range. The range [R_a:R_b] can be either increasing or decreasing. In either case, R_a is the index of the most-significant (leftmost) bit in the vector, and R_b is the index of the least-significant (rightmost) bit. The indices R_a and R_b can be either positive or negative integers. The net S can be used as a four-bit quantity, or each bit can be referred to individually as S[3], S[2], S[l], and S[0]. If a value is assigned to S such as S = 4'b0011, the result is S[3] = 0, S[2] = 0, S[l] = 1, and S[0] = 1.

The tri type denotes circuit nodes that are connected in a tri-state fashion. These nets are treated in the same manner as the wire type, and they are used only to enhance the readability of code that includes tri-state gates. Examples of tri nets are

tri y;
tri [7:0] DataOut;

3.4.2 Variables

Nets provide a means for interconnecting logic elements, but they do not allow a circuit to be described in terms of its behaviour. For this purpose, Verilog provides variables. A *variable* can be assigned a value in one Verilog statement, and it retains this value until it is overwritten in a subsequent assignment statement. There are two types of variables, *reg* and *integer*.

In Verilog code, reg variables can be used to model either combinational or sequential parts of a circuit. The keyword reg can be used to store information ('state') like registers. Example:

reg [3:0] S;

Instead of using a separate statement to declare that the variable S is of reg type, we can include this declaration in the statement that specifies S to be an output, as follows:

output reg [3:0] S;

Integer variables are useful for describing the behaviour of a module, but they do not directly correspond to nodes in a circuit. Integers are usually used as loop control variables, constant and parameters. Example:

integer k;

3.5 Operators

Verilog has a large number of operators, as shown in Table 1. The first column gives the category, the second column indicates how each operator is used, and the third column specifies the number of bits produced in the result. In this table f is scalar, A, B and C are 3 bits vector and D is 9 bits vector.

Most of the operators in Table 1 are self-explanatory and similar to the C++ language that you learned in first year Structure Programming.

For the command, C=-A; If A=101 then C=010+1=011.
For the command, C=A&B; If A=101 and B=001 then C=101& 001= 001.
For the command, f= |A; If A=101 then f=1+0+1= 1.
For the command, B=A<<1; If A=101 then B=010.
For the command, B=A>>2; If A=101 then B=001.
For the command, D={A,B,C}; If A=101, B=001 and C=110 then D=101001110.

Table 2 shows the Verilog operators in descending order of precedence. Operators with equal precedence are shown grouped.

3.6 Concurrent statements

In any HDL, the concept of a concurrent statement means that the code may include a number of such statements, and each represents a part of the circuit. The word concurrent means the statements are considered in parallel and the ordering of statements in the code does not matter.

3.6.1 Continuous assignment

Continuous assignments is used in the description of a circuit's function. The general form of this statement is

assign net_assignment {, net_assignment};

Category	Examples	Bit Length
Bitwise	C=~A; // 1's complement	L(A)
	C=-A; // 2's complement	
	C=A&B; // AND	max (L(A), L(B))
	C=A|B; // OR	
	C=A^B; //XOR	
	C=A~^B; // XNOR	
	C=A^~B; // XNOR	
Logical (used in conjunction with relational and equality operators	f = !A // NOT A - f=1 if A=000 only	1 bit
	if A&&B // A AND B	
	if A||B // A OR B	
Reduction	f=&A; //AND A	1 bit
	f=~&A; //NAND A	
	f=|A; //OR A	
	f=~|A; //NOR A	
	f=^A; //XOR A	
	f=~^A; //XNOR A	
	f=^~A; //XNOR A	
Relational	if A==B // A equal B	1 bit
	if A!=B // A not equal B	
	if A>B // A greater B	
	if A<B // Aless than B	
	if A>=B // A greater than or equal B	
	if A<=B // A less than or equal B	
Arithmetic	C=A+B; // addition	max (L(A), L(B))
	C=A-B; // subtraction	
	C=A*B; // multiplication	
	C=A/B; // division	
	C=A%B; //modulus	
Shift	B=A<<n; // B equal A shift left by n bits	L(A)
	B=A>>n; // B equal A shift right by n bits	
Concatentation	D={A,B,C}; // combination of A,B,C	L(A)+ L(B)+L(C)
Conditional	C=(A<B)?A:B; // C=A if A<B else C=B	max (L(A), L(B))

Table 1. Verilog operators

The net_assignment can be any expression involving the operators listed in Table 1; Examples of continuous assignments are

assign Cout = (x & y) | (x & Cin) | (y & Cin);
assign s = x ^ y ^ z;

Multiple assignments can be specified in one assign statement, using commas to separate the assignments, as in

assign Cout = (x & y) | (x & Cin) | (y & Cin), s = x ^ y ^ z;

Verilog Operator	Name	Functional Group
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
-	unary (sign) minus	bitwise
{ }	concatenation	concatenation
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	logical equality	equality
!=	logical inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
^~ or ~^	bit-wise XNOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Table 2. Verilog operator precedence in descending order.

It is possible to combine a continuous assignment with a wire declaration, as in:

wire s = x ^ y, c = x & y;

3.7 Procedural statements

Whereas concurrent statements are executed in parallel, procedural statements are evaluated in the order in which they appear in the code sequentially. Verilog syntax requires that procedural statements be contained inside an always block.

3.7.1 Always block

An always block is a construct that contains one or more procedural statements. It has the form shown in Figure 4.

```
always @(sensitivity _list)
[begin]
    [procedural assignment statements]
    [if-else statements]
    [case statements]
    [while, repeat, and for loops]
    [task and function calls]
[end]
```

Figure 4. Always block

When multiple statements are included in an always block, the begin and end keywords are needed; otherwise, these keywords can be omitted.

The sensitivity_list is a list of signals that directly affect the output results generated by the always block. If the value of a signal in the sensitivity list changes, then the statements inside the always block are evaluated in the order presented. In Figure 2, if either Cin, x or y change the statements within the begin-end block will execute sequential. If there are no change in Cin, x or y then the statements within the begin-end block will not execute.

A Verilog module may include several always blocks, each representing a part of the circuit being modeled. While the statements inside each always block are evaluated in order, there is no meaningful order among the different always blocks. In this sense, each entire always block can be considered as a concurrent statement, because a Verilog compiler evaluates all always blocks concurrently.

3.7.2 Procedural Assignment Statements

Any signal assigned a value inside an always block has to be a variable of type reg or integer. A value is assigned to a variable with a procedural assignment statement. There are two kinds of assignments: *blocking* assignments, denoted by the = symbol, and *non-blocking* assignments, denoted by the <= symbol. The term blocking means that the assignment statement completes and updates its left-hand side before the subsequent statement is evaluated. This concept is best explained in the context of simulation. Consider the blocking assignments

```
S = X + Y;
p = S[0];
```

At simulation time, t_i the statements are evaluated, in order. The first statement sets S using the current values of X and Y, and then the second statement sets p according to this new value of S.

Consider the non-blocking assignments

```
S <= X + Y;
p <= S[0];
```

In this case, at simulation time t_i the statements are still evaluated in order, but they both use the values of variables that exist at the start of the simulation time t_i. The first statement determines a new value for S based on the current values of X and Y, but S is not actually changed to this value until all statements in the associated always block have been evaluated. Therefore, the value of p at time t_i is based on the value of S at time t_{i-1}. The difference between blocking and non-blocking assignments can be summarized as follows. For blocking assignments, the values of variables seen at time t_i by each statement are the new values set in t_i by any preceding statements in the always block. For non-blocking assignments, the values of variables seen at time t_i are the values set in time t_{i-1}. For combinational circuits, only blocking assignments should be used.

3.7.3 If-else Statements

Figure 5 shows the syntax for the if-else statement.

```
if (expression 1)
begin
    statement(s);
end
else if (expression2)
begin
    statement(s);
end
else
begin
    statement(s);
end
```

Figure 5. If-else syntax

If expression 1 is true, then the first statement is evaluated. When multiple statements are involved, they have to be included inside a begin-end block. The else if and else clauses are optional. Verilog syntax specifies that when else if or else are included, they are paired with the most recent unfinished if or else if.

Figure 6 shows a Verilog code for a 2-to-1 multiplexer.

```
module mux2tol (in0, inl, s, f);
   input in0, inl, s;
   output reg f;
   always @(in0, in1, s)
      if (s == 0)
         f = in0;
      else
         f = in1;
endmodule
```

Figure 6. 2-to-1 multiplexer using if-else statement

3.7.4 Case Statements

The general form of a case statement is shown in Figure 7.

```
case (expression)
   alternative 1: begin
                     statement;
                  end
   alternative 2: begin
                     statement;
                  end
   [default:     begin
                     statement;
                  end]
endcase
```

Figure 7. Syntax for case statement.

The bits in expression, called the controlling expression, are checked for a match with each alternative. The first successful match causes the associated statements to be evaluated. Each digit in each alternative is compared for an exact match of the four values 0, 1, x, and z. A special case is the default clause, which takes effect if no other alternative matches. When using Verilog for simulation, an alternative can be a general expression, but for synthesis these items are restricted to a single constant, such as 1'b0:, or a list of constants separated by commas, such as 1, 2, 3:.

Figure 2 shows a full adder describe using case statement.

3.7.5 Casez and Casex Statements

In the case statement, the values x or z in an alternative are checked for an exact match with the same values in the controlling expression. The casez statement adds more flexibility, by treating a z digit in an alternative as a don't-care condition. The casex statement treats both x and z as don't cares. The alternatives do not have to be mutually exclusive. If they are not, then the first matching item has priority. Figure 8 shows a priority encoder describe using casex statement.

```
module priority (w, y, z);
   input [3:0] w;
   output reg [1:0] y;
   output z;
   always @(w)
   begin
      z = 1;
      casex (w)
         4'blxxx: y = 3;
         4'b0lxx: y = 2;
         4'b00lx: y = l;
         4'b000l: y = 0;
         default: begin
                     z = 0;
                     y = 2'bx;
                  end
      endcase
   end
endmodule
```

Figure 8. A priority encoder describe using a case x statement.

Example 1

Write the truth table of priority encoder shown in Figure 8.

Solution:

3.7.6 Loop Statements

Verilog includes four types of loop statements: for, while, repeat, and forever. Synthesis tools typically support the for loop, which has the general form shown in Figure 9.

```
for (initial_index; terminal_index; increment)
begin
    statement;
end
```

Figure 9. Syntax for for-loop.

The initial_index is evaluated once, before the first loop iteration, and typically performs the initialization of the integer loop control variable, such as k = 0. In each loop iteration, the begin-end block is performed, and then the increment statement is evaluated. A typical increment statement is k = k + 1. Finally, the terminal_index condition is checked, and if it is True (1), then another loop iteration is done. For synthesis, the terminal_index condition has to compare the loop index to a constant value, such as k < 8.

Figure 10 shows a ripple-carry adder describe using for-loop statement.

```
module ripple (carryin, X, Y, S, carryout);
   parameter n = 4;
   input carryin;
   input [n-1:0] X, Y;
   output reg [n-1:0] S;
   output reg carryout;
   reg [n:0] C;
   always @( X, Y, carryin)
   begin
      C[0] = carryin;
      for (k = 0; k<= n-1; k=k+1)
      begin
         S[k] = X[k] ^ Y[k] ^ C[k];
         C[k+1] = (X[k] & Y[k]) | (C[k] & X[k]) | (C[k] & Y[k]);
      end
      carryout=C[n];
   end
endmodule
```

Figure 10. A ripple-carry describe using for-loop statement.

3.7.7 Gate instantiations

Verilog includes predefined modules that implement basic logic gates. These gates allow a circuit's structure to be described using gate instantiation statements of the form

gate_name [instance_name] (output_port, input_port{, input_port});

The gate_name specifies the desired type of gate, and the instance_name is any unique identifier. Each gate may have a different number of ports, with the output port listed first, followed by a variable number of input ports.

Figure 11 shows a full-adder describe using gate instantiation. The code defines four wire nets, z1 to z4, that connect the gates together, and each gate has a specified instance name.

The logic gates supported in Verilog are summarized in Table 1. The second column describes the function of each gate, and the rightmost column gives an example of instantiating the gate. Verilog allows gates with any number of inputs to be specified, but some CAD systems set practical limits. The notif and bufif gates represent tri-state buffers (drivers). The gate notif0 is an inverting tri-state buffer with active-low enable, and notif1 provides the same functionality with an active-high enable. The bufif0 and bufif1 gates are tri-state buffers that do not invert the output.

```
// Structural specification of a full-adder
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;
    wire zl,z2,z3,z4;
    and And1 (z l, x, y);
    and And2 (z2, x, Cin);
    and And3 (z3, y, Cin);
    or Or1 (Cout, zl, z2, z3);
    xor Xorl (z4, x, y);
    xor Xor2 (s, z4, Cin);
endmodule
```

Figure 11. A full-adder describe using gate instantiation

Name	Description	Usage
and	$f = (a \cdot b \cdot \cdots)$	**and** $(f, a, b, \ldots)$
nand	$f = \overline{(a \cdot b \cdot \cdots)}$	**nand** $(f, a, b, \ldots)$
or	$f = (a + b + \cdots)$	**or** $(f, a, b, \ldots)$
nor	$f = \overline{(a + b + \cdots)}$	**nor** $(f, a, b, \ldots)$
xor	$f = (a \oplus b \oplus \cdots)$	**xor** $(f, a, b, \ldots)$
xnor	$f = (a \odot b \odot \cdots)$	**xnor** $(f, a, b, \ldots)$
not	$f = \overline{a}$	**not** (f, a)
buf	$f = a$	**buf** (f, a)
notif0	$f = (!e \ ? \ \overline{a} : \ 'bz)$	**notif0** (f, a, e)
notif1	$f = (e \ ? \ \overline{a} : \ 'bz)$	**notif1** (f, a, e)
bufif0	$f = (!e \ ? \ a : \ 'bz)$	**bufif0** (f, a, e)
bufif1	$f = (e \ ? \ a : \ 'bz)$	**bufif1** (f, a, e)

Table 1. Verilog logic gates

Example 2

Draw the schematic diagram of the full adder shown in Figure 8.

Solution:

3.7.8 Sub-circuit

A Verilog module can be included as a sub-circuit in another module. For this to work, both modules must be defined in the same file or else the Verilog compiler must be told where each module is located (the mechanism for doing this varies from one compiler to the next). The general form of a module instantiation statement is similar to a gate instantiation statement as shown:

module_name [#(parameter overrides)] instance_name (.port_name ([expression]){, .port_name ([expression])});

The instance_name can be any legal Verilog identifier and the port connections specify how the module is connected to the rest of the circuit. The same module can be instantiated multiple times in a given design provided that each instance name is unique. The #(parameter overrides) can be used to set the values of parameters defined inside the module_name module. Each port_name is the name of a port in the sub-circuit, and each expression specifies a connection to that port. The syntax .port_name is provided so that the order of signals listed in the instantiation statement does not have to be the same as the order of the ports given in the module statement of the sub-circuit. In Verilog jargon, this is called *named port connections*. If the port connections are given in the same order as in the sub-circuit, then .port_name is not needed. This format is called *ordered port connections*.

```
module adder4(carryin,X,Y,S, carryout);
   input carryin;
   input [3:0] X, Y;
   output [3:0] S;
   output carryout;
   wire [3:1] C;
   fulladd stage0 (carryin, X[0], Y[0], S[0], C[l]);
   fulladd stagel (C[l], X[l], Y[l], S[l], C[2]);
   fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
   fulladd stage3 (.Cout(carryout), .s(S[3]), .y(Y[3]), .x(X[3]), .Cin(C[3]));
endmodule
```

Figure 12. 4-bit ripple carry adder using module instantiation

Figure 12 is a code for a four-bit ripple-carry adder built using four instances of the fulladd sub-circuit in Figure 2. The inputs to the adder are carryin and two four-bit numbers X and Y. The output is the four-bit sum, S, and carryout. A three-bit signal, C, represents the carries from stages 0, l, and 2. This signal is declared as a wire vector with three bits. The adder4 module instantiates four copies of the full-adder sub-circuit. In the first three instantiation statements, we use *ordered port connections* because the signals are listed in the same order as given in the declaration of the fulladd module in Figure 2. The last instantiation statement gives an example of *named port connections*. The port connections used in the instantiation statements specify how the fulladd instances are interconnected by nets to create the adder module.

3.8 Verilog for Combinational Circuits

It is best practice to write simple combinational circuits using continuous assign statements (as shown in Figure 13). For more complicated combinational circuits use the *blocking* assignments in the *always* construct.

```
module mux2to1(w0,w1,s,f);
    input w0,w1,s;
    output f;

    assign f=s?w1:w0;

endmodule
```

Figure 13. 2-to-1 multiplexer using assign statement

Example 3

Derive the truth table for the Verilog code in Figure 13 and hence write the same multiplexer using a *blocking always* construct.

Solution:

Example 4

Derive the truth table for the Verilog code in Figure 14.

```
module Ex4(W,Y,z);
   input [3:0] W;
   output reg [1:0] Y;
   output reg z;
   integer k;
   always @(W)
   begin
      Y=2'bx;
      z=0;
      for (k=0; k<4; k=k+1)
         if (W[k])
            begin
               Y=k;
               z=1;
            end
   end
endmodule
```

Figure 14. Verilog code for Example 4

Solution:

3.9 Verilog for Sequential Circuits

It is best practice to write sequential circuits using the *non-blocking* assignments in the *always* construct.

```
module D_latch(D,LE,Q);
   input D,LE;
   output reg Q;
   always @(D or LE)
   begin
      if (LE) Q<=D;
   end
endmodule
```

Figure 15. Verilog code for D-latch.

Example 5

Figure 16 shows the input stimulus for the D-latch in Figure 15. Draw the expected Q waveform of the D-latch. You may assume Q is initially low.

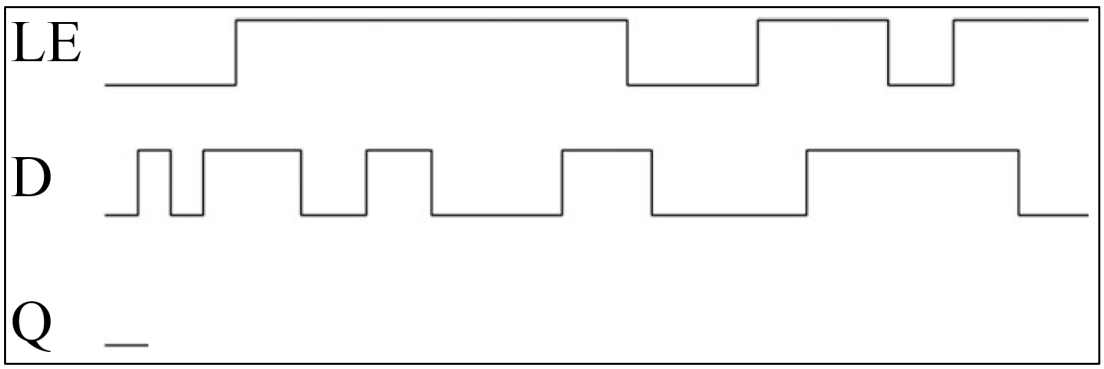

Figure 16.

Example 6

Identify the sequential device as described in the Verilog code in Figure 17. Figure 18 is the input stimulus for this device. Draw the output Q[3] to Q[0].

```
module Ex6(R,L,w,Clk,Q);
   input [3:0] R;
   input L,w,Clk;
   output reg [3:0] Q;
   integer k;
   always @(posedge Clk)
     if (L) Q <= R;
     else
       begin
          for (k=0; k<3; k=k+1)
               Q[k] <= Q[k+1];
          Q[3] <= w;
       end
endmodule
```

Figure 17. Verilog code for Example 6

Solution:

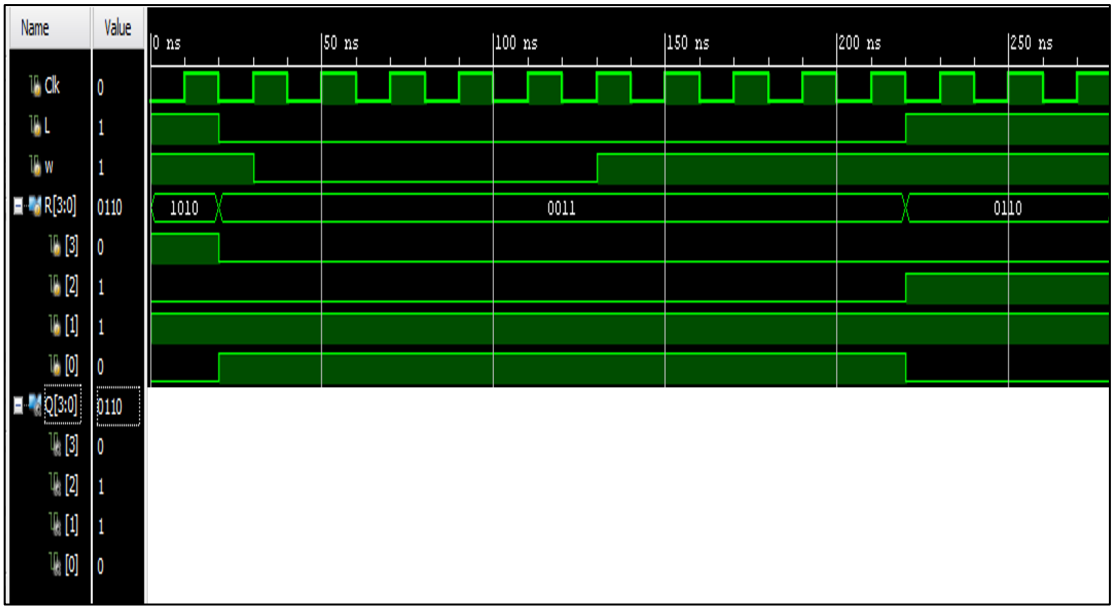

Figure 18. Input stimulus for Example 6

4 Simulation

Recalled from Chapter 3, section 3.3, after writing the Verilog code for the design, we can test out the design by doing a functional simulation. The input test vectors for the simulation can be entered by drawing the input waveforms or normally by writing a Verilog code called a *test-bench* as shown in Figure 19. This test-bench will instantiate the design code and the simulator will use the signals in the test-bench to simulate the design and display the output result in graphical or text form as shown in Figure 19.

Note that in Figure 19, the outputs of the design, *Chap1Counter* becomes the input to the tester, *Chap1CounterTester* and the input of the design becomes the output of the tester.

Figure 20 shows the test bench written for the waveforms in Figure 18.

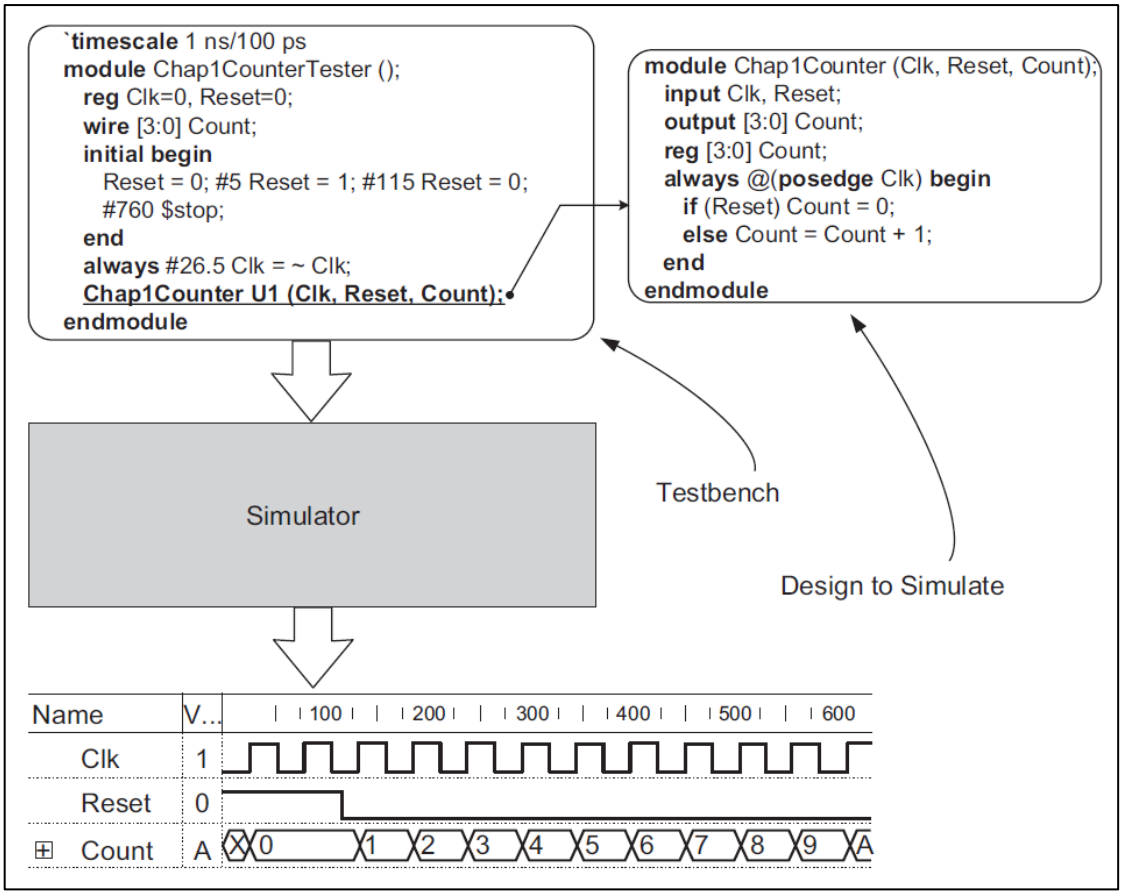

Figure 19. Verilog simulation with a Testbench

```
`timescale 1ns / 1ps // simulator time unit = 1 ns and precision = 1 ps
module Ex6_tb();    // module name Ex6_tb, input and output is optional here
   wire [3:0] Q;       // output in design become input in test-bench
   reg [3:0] R;        // input in design become output in test-bench
   reg L;
   reg w;
   reg Clk=0;          // Clk is initialized with logic '0'
   initial             // initial block is not synthesizeable, meant for simulation only
     begin
       L=1;            // Initially L= '1'
       #20 L=0;        // 20 ns later L= '0'
       #200 L=1;       // another 200 ns later L= '1'
     end
   initial
      begin
        w=1;           // Initially w = '1'
        #30 w=0;       // 30 ns later w= '0'
        #100 w=1;      // another 100 ns later w= '1'
      end
    initial
      begin
        R=4'b1010;          // Initially R = "1010"
        #20 R=4'b0011;    // 20 ns later R = "0011"
        #200 R=4'b0110;  // another 200 ns R = "0110"
       end
    always #10 Clk=~Clk;  // Clock pulse train of period = 10 + 10 ns = 20 ns
    Ex6 dut(R,L,w,Clk,Q);    //an instance of Ex6 is added and called dut
endmodule
```

Figure 20. Verilog testbench for Figure 18