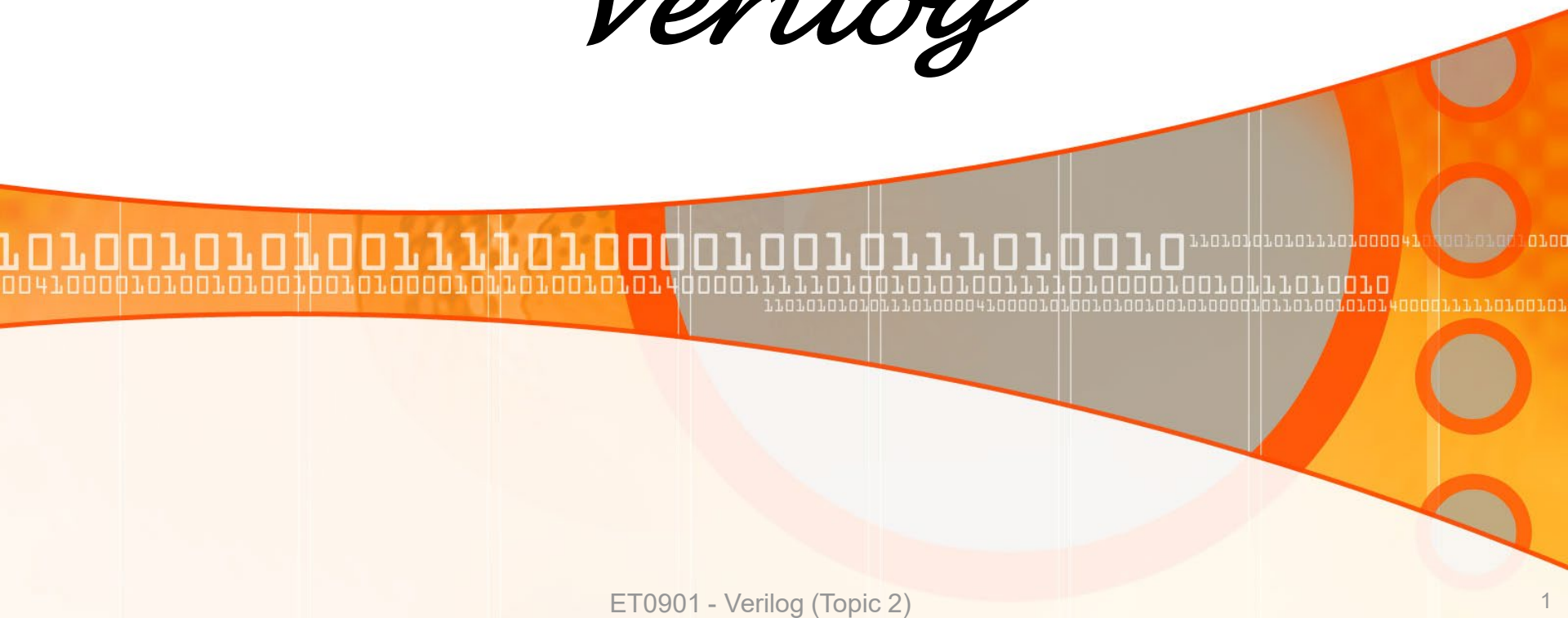
Chapter 2

Verilog



Objectives

At the end of the topic, the students should be able to:

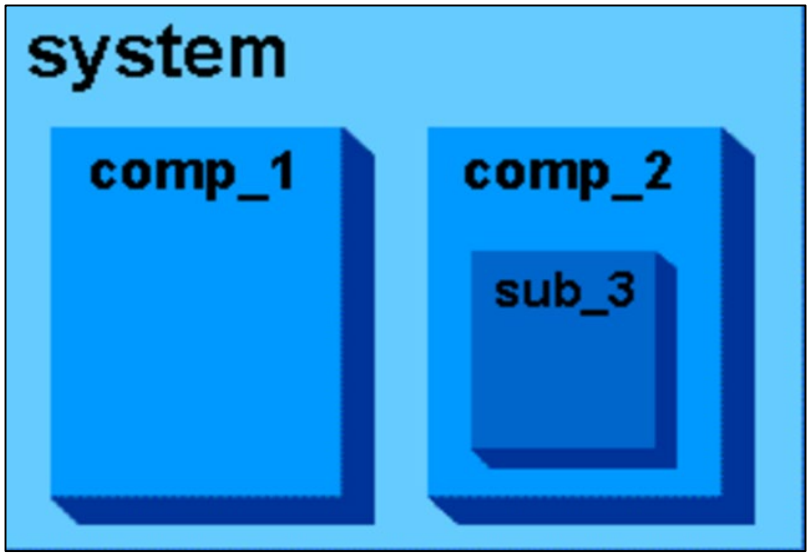
- explain the hierarchy and modelling structures of Verilog,
- identify the different types of net and variables,
- distinguish the usage of concurrent and procedural statements and
- design digital electronic systems using sub-circuits and functions.

Introduction

- **Verilog** was originally intended for simulation and verification of digital circuits.
- Later use in design entry in **CAD** systems.
- CAD tools are used to ***synthesize*** the Verilog code into a hardware implementation of the described circuit.
- 2 ways to describe a circuit in Verilog:
  - *Structural* – using circuit elements eg. logic gates.
  - Behavioural – using logic expressions & programming constructs

Hierarchical Design in Verilog

- *The Verilog structures which build the hierarchy are:* ***modules*** *and* ***ports****.*
- ***Module*** – basic unit of a Verilog model & may be composed of instances of other modules.
- ***Port*** – allows data to flow into and out of modules.



- *How many modules are there in this diagram?*
- *Which module(s) is/are parent/child?*

Verilog syntax and constructs

- Verilog code is ***case sensitive***.
- *//This is a short comment*
- */*This is a long Verilog comment that spans more than ……one line */*
- White space characters, eg. SPACE and TAB, and blank lines are ignored by the Verilog compiler.
- Multiple statements can be written on a single line but not recommended.
- ; is the end of a command.
- *Identifiers* are the names of variables and other elements in Verilog code.

Verilog syntax and constructs

- 2 simple rules for specifying identifiers
  - *must begin with an alphabet or underscore only*
  - *must not be a Verilog keyword*

General syntax for module and port

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

Behavioural code for a full adder

```
// 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
```

Data types

- Verilog has just a few basic data types, namely:
  - scalar,
  - vector,
  - integer,
  - real and
  - time.

Scalar Data types

- Single bit value of the following:

  0 - logic zero (false)

  1 - logic one (true)

  x/X - unknown value

  z/Z - high-impedance value

- x/? can be used to denote a don't-care

Vector Data types

- multi-bit quantity.
- 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

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

Integer Data types

- An integer is a signed, 2's complement value.
- It is declared in an integer statement, and it is a register.

Real Data types

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

Time Data types

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

Parameters

- A parameter associates an identifier name with a constant. Eg

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

Signals

- A signal in a circuit is represented as a *net* or a *variable* with a specific type.
- Syntax for declaring a net or variable is:

  type *[range] signal_name{, signal_name};*

- ***Type*** is any of the net or variable type such as **input**, **output** etc.
- Without the range field the declared net or variable is ***scalar*** and represents a single-bit signal.
- Range is used to specify ***vectors*** that correspond to multibit signals

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 few other types.
- ***wire*** 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. Eg.

```
wire x;                    wire [3:0] S;
wire Cin, AddSub;          wire [1:2] sum;
```

- The **tri** type denotes circuit nodes that are connected in a tri-state fashion. Eg

```
tri y;
tri [7:0] DataOut;
```

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.
- 2 two types of variables ➔ ***reg*** and ***integer***.
- **reg** variables can be used to model either combinational or sequential parts of a circuit. Eg.

  reg [3:0] S; **output reg** [3:0] S;

- Integers are usually used as loop control variables, constant and parameters. Eg

  integer k;

Operators

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	

Operators

Category	Examples	Bit Length
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))

Operators precedence

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

Concurrent statements

- Concurrent statements are considered in parallel and the ordering of statements in the code does not matter.
- Concurrent statement is executed when any one of the signals in the RHS of assignment changes.

Continuous assignment

- used in the description of a circuit's function.
- general form of this statement is:

  assign net_assignment {, net_assignment};

- net_assignment can be any expression involving the operators listed in the Operators table. Eg:

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

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

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

Procedural statements

- evaluated in the order in which they appear in the code sequentially
- must be contained inside an ***always*** block.

Always block

- a construct that contains one or more procedural statements

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

- begin and end keywords may be omitted if there is only 1 procedural statement.

Always block

- If the value of a signal in the sensitivity list changes, then the statements inside the ***always*** block are evaluated sequentially.
- A module may include several ***always*** blocks.
- Each ***always*** block can be considered as a concurrent statement.

```
// 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
```

Procedural Assignment Statements

- Any signal assigned a value inside an ***always*** block has to be a variable of type ***reg*** or ***integer***.
- 2 kinds of procedural assignments:
  - *blocking* assignments, denoted by the = symbol. The assignment statement completes and updates its left-hand side before the subsequent statement is evaluated.
  - *non-blocking* assignments, denoted by the <= symbol.

Procedural Assignment Statements

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

- Initially S=1, X=2, Y=3, p=4
- After simulation:

  S=2+3=5

  p = 5

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

- Initially S=1, X=2, Y=3, p=4
- After simulation:

  S=2+3=5

  p = 1

- For combinational circuits, only blocking assignments should be used.

If-else Statements

- ***begin*** and ***end*** keywords may be omitted if there is only 1 statement.
- ***else if*** and ***else*** clauses are optional.

```
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
```

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

Case Statements

```
// 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
```

native is compared
he four values 0, 1,
default clause,
other alternative

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

Casez and Casex Statements

- ***casex*** statement treats both x and z as don't cares.
- ***casez*** statement adds more flexibility, by treating a z digit in an alternative as a don't-care condition.

```
module priority (w, y, z);
  input [3:0] w;
  output reg [1:0] y;
  output reg 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
```

Example 1

Write the truth table of priority encoder shown.

Solution:

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

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

Loop Statements

- Four types of loop statements: ***for, while, repeat, and forever***.
- Synthesis tools typically support the ***for*** loop.

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

Loop Statements

```
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
```

Gate Instantiations

- Verilog has basic logic gates that 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});
```

Gate Instantiations

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)

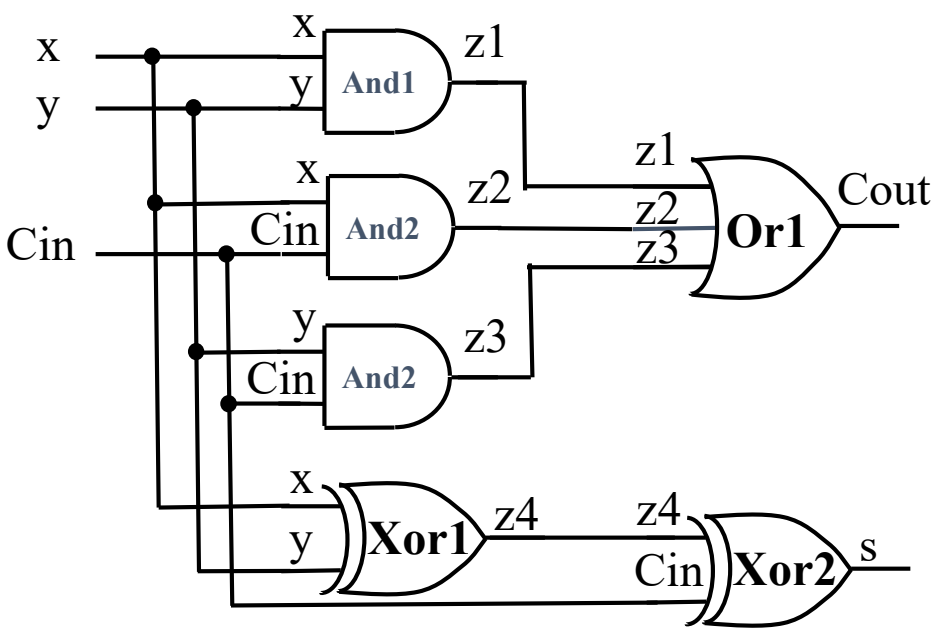
Gate Instantiations

```
// 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
```

Example 2

Draw the schematic diagram of the full adder as described here.

Solution:




```
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
```

Sub-circuit

- A Verilog module can be included as a sub-circuit in another module.
- Both modules must be defined in the same file or else the Verilog compiler must be told where each module is located.
- General form of a module instantiation statement is:

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

- #(parameter overrides) can be used to set the values of parameters defined inside the module_name module.

Sub-circuit

- .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 ➔ *named port connections*
- If the port connections are given in the same order as in the sub-circuit, then .port_name is not needed. ➔ *ordered port connections*.

Sub-circuit

```
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
```

Verilog for Combinational Ckts

- best practice to write simple combinational circuits using *continuous* assign statements.
- For more complicated combinational circuits use the *blocking* assignments in the *always* construct.

Example 3a

Derive the truth table for this Verilog code.

Solution:

s	w1	w0	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

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

  assign f=s?w1:w0;

endmodule
```

Example 3b

Modify the following Verilog code using blocking always construct.

Solution:

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

   always @ (w0,w1,s)
       f=s?w1:w0;

endmodule
```

Example 4

Derive the truth table for this Verilog code.

Solution:

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

```
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
```

Verilog for Sequential Ckts

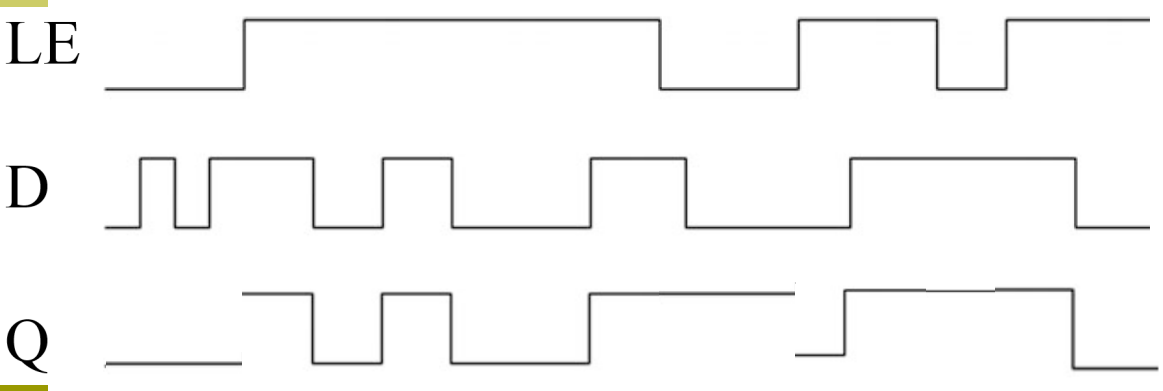
- best practice to write sequential circuits using the non-blocking assignments in the always construct.

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

Example 5

Draw the expected Q signal for this code.

Solution:




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

Example 6a

Identify the sequential device for this Verilog code.

Solution:

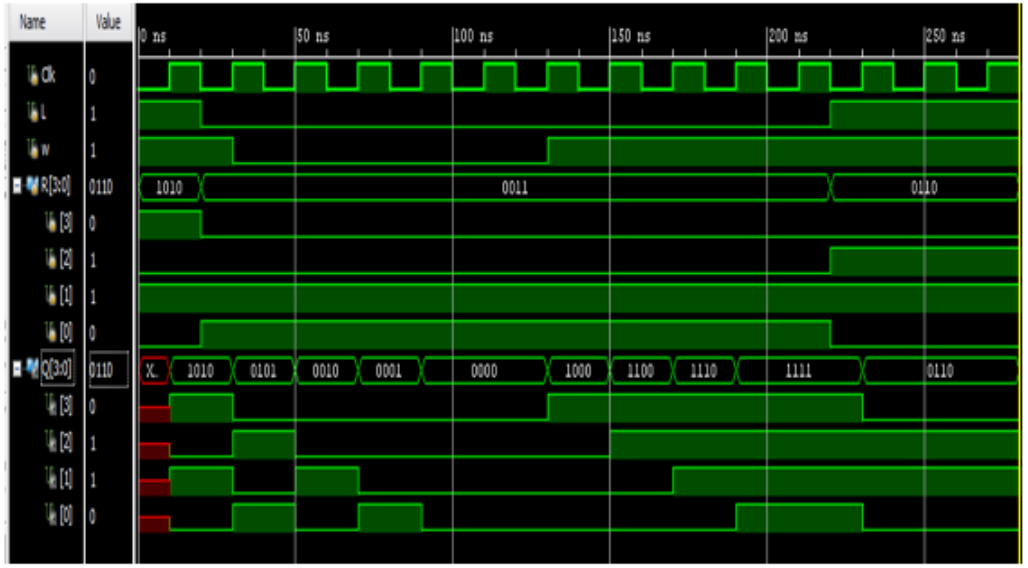
4-bit shift register

```
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
```

Example 6b

Draw the output Q[3] to Q[0] for this Verilog code.

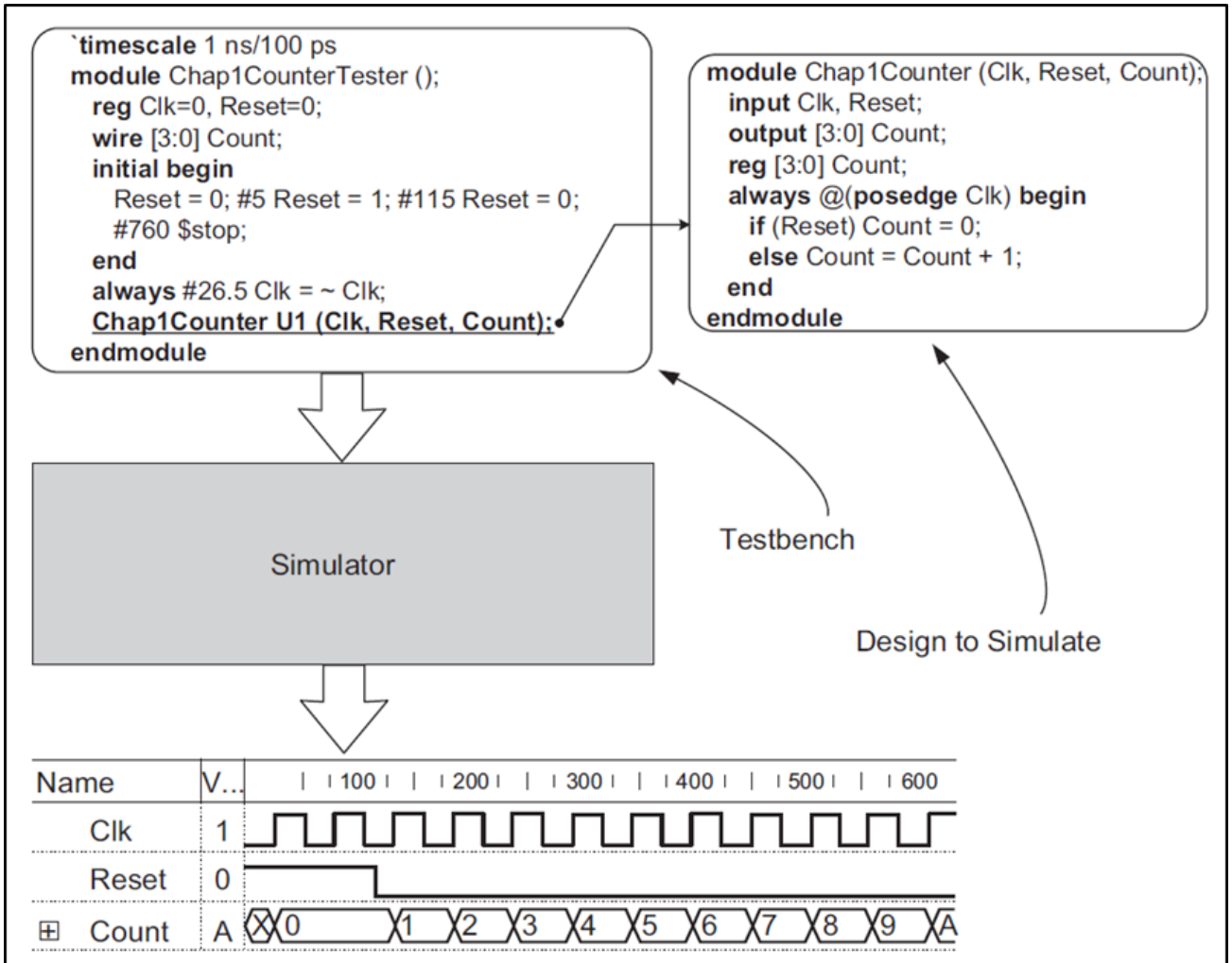
Solution:




```
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
```

Verilog test bench for Simulation

- Test vectors used for simulation in HDL is called *test bench*.



Test bench for Example 6

```
`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
```

Test bench for Example 6

```
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
```