Topic A

Introduction to Verilog HDL

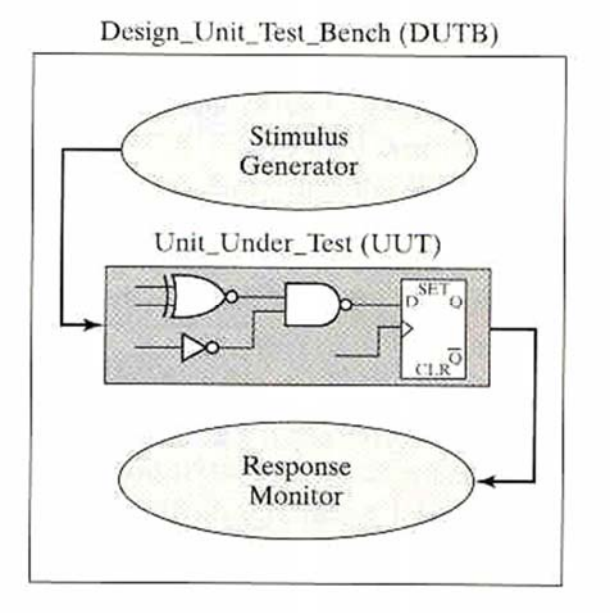
陳春僥

May 20, 2021

HDL

- HDL—
  - Hardware Description Language
- Two Popular HDL
  - Verilog HDL
    - Like C, C++
  - VHDL
    - Very High Speed Integrated Circuit Hardware Description Language
    - Like PASCAL

Organization of a Testbench



General Form of a Module

module module_name [(port_name{, port_name})];

[**input** declarations]
[**output** declarations]
[**inout** declarations]

[**wire** or **tri** declarations]
[**reg** or **integer** declarations]

[**parameter** declarations]

[**function** or **task** declarations]
[**gate** instantiations]
[**module** instantiations]
[**assign** continuous assignment]
[**initial** block]
[**always** block]

endmodule

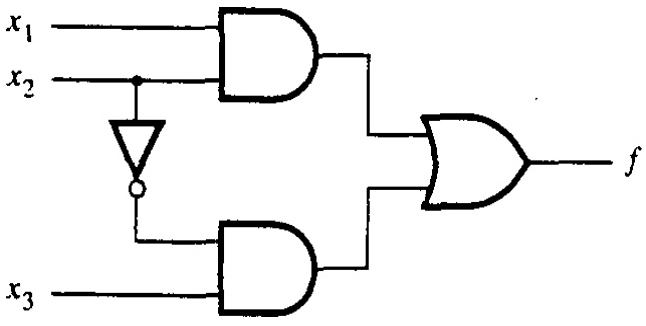
Testbench Template

```
module t_DUTB_name ();          // Substitute the name of the UUT
   reg ...;                     // Declaration of register variables
                                // for primary inputs of the UUT
   wire ...;                    // Declaration of primary outputs of the UUT
   parameter                    // Provide a value

   UUT_name M1_instance_name (UUT ports go here);

initial                         // Develop one or more behaviors for pattern
                                // generation and/or error detection
   begin
                                // Behavioral statements generating waveforms
                                // to the input ports, and comments documenting
                                // the test.
   end

 initial $monitor ();  // Specification of signals to be monitored and
                                // displayed as text
 initial #time_out $finish;     // Stopwatch to assure termination of simulation

endmodule
```


A Simple Logic Function

- Structural Specification

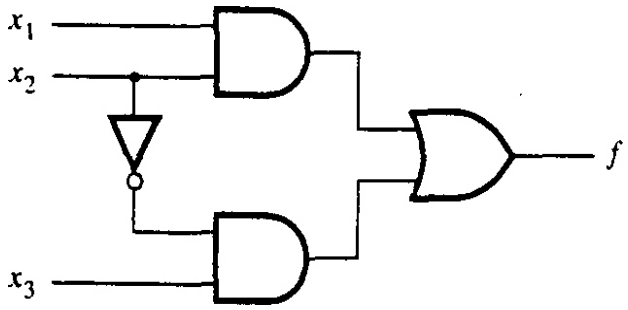
```
module example1 (f, x1, x2, x3);

   input x1, x2, x3;
   output f;

   and (g, x1, x2);
   not (k, x2);
   and (h, k, x3);
   or (f, g, h);

endmodule
```



A Simple Logic Function

- Dataflow Specification—
  - Continuous assignment

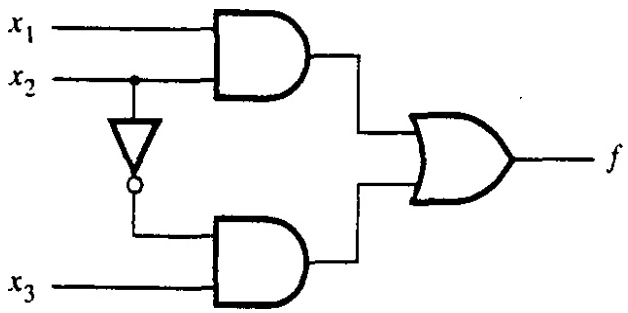
```
module example3 (f, x1, x2, x3);

    input x1, x2, x3;
    output f;

    assign f = (x1 & x2) | (~x2 & x3);

endmodule
```



A Simple Logic Function

- Behavioral Specification—
  - Procedural assignment

```
// Behavioral specification
module example5 (f, x1, x2, x3);

    input x1, x2, x3;
    output f;
    reg f;

    always @ (x1 or x2 or x3)
        if (x2 == 1)
            f = x1;
        else
            f = x3;

endmodule
```



A Simple Logic Function

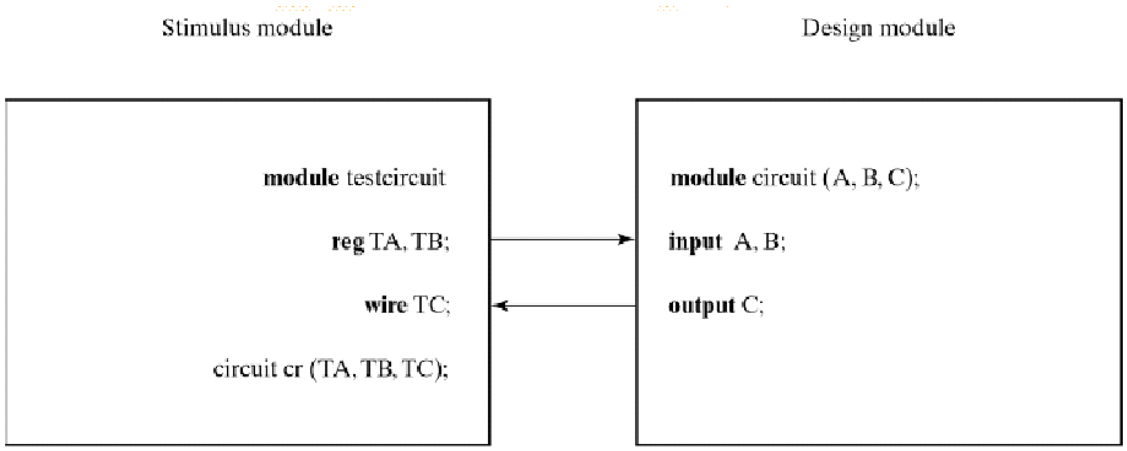
- Testbench

```
`timescale 1ns/1ns                  // t_unit/t_precision
module t_example1;
   reg x1, x2, x3;
   example1 test (f, x1, x2, x3); // Enter fixture code here
   initial
   begin
      x1 = 1'b0;
      x2 = 1'b0;
      x3 = 1'b0;
      #200
      x1 = 1'b0;
      x2 = 1'b0;
      x3 = 1'b1;
      #200
      x1 = 1'b0;
      x2 = 1'b1;
      x3 = 1'b0;
      #200
      $finish;
   end
endmodule // t_example1
```

```
`timescale 1ns/1ns
module t_example1;
   reg x1, x2, x3;
   example1 test (f, x1, x2, x3); // Enter fixture code here
   initial #600 $stop;
   initial
   begin
      x1 = 1'b0;
      x2 = 1'b0;
      x3 = 1'b0;
      #200
      x1 = 1'b0;
      x2 = 1'b0;
      x3 = 1'b1;
      #200
      x1 = 1'b0;
      x2 = 1'b1;
      x3 = 1'b0;
   end
endmodule // t_example1
```


Preparing a Testbench

- Stimulus and design module interaction



Preparing a Testbench

- System tasks
  - $display
    - Display one-time value of variables or strings with end-of-line return
  - $write
    - Same as $ display but without going to next line
  - $monitor
    - Display variables whenever a value changes during simulation run
  - $time
    - Displays simulation time
  - $finish
    - Terminates the simulation

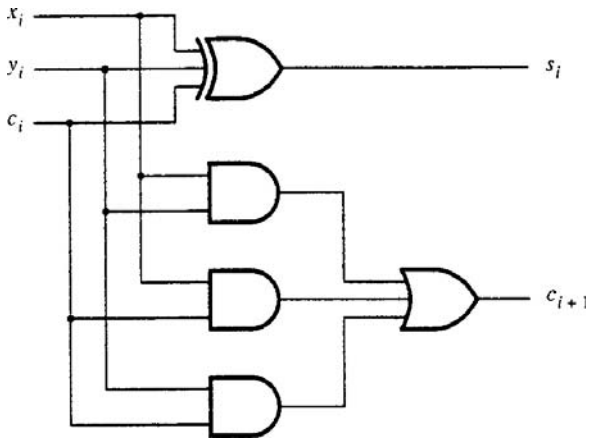
initial vs. always

- initial—
  - Executed once
  - Good for generating input signals to simulate a design
- always—
  - Executed when the conditions are met

Full Adder

- Structural Specification (version 1)

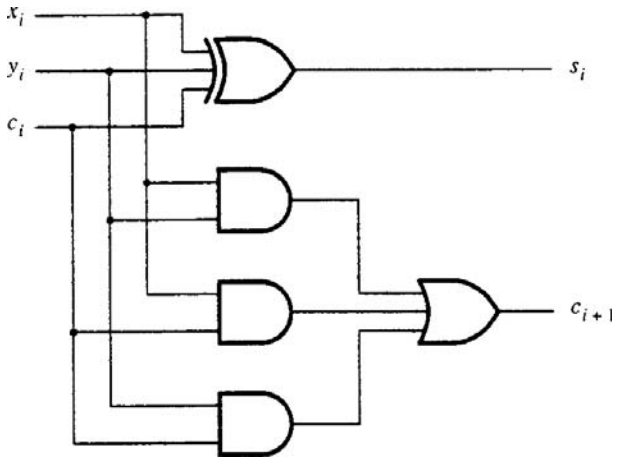
```
module fulladd (cout, s,  cin, x, y);

   input cin, x, y;
   output cout, s;

   xor (s, cin, x, y);
   and (z1, x, y);
   and (z2, x, cin);
   and (z3, y, Cin);
   or (cout, z1, z2, z3);

endmodule
```



Full Adder

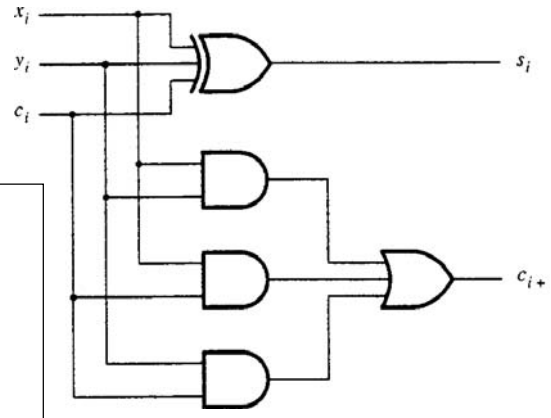
- Structural Specification (version 2)

```
module fulladd(cout, s, cin, x, y);

   input cin, x, y;
   output cout, s;

   xor (s, cin, x, y);
   and (z1, x, y),
       (z2, x, cin),
       (z3, x, cin);
   or (cout, z1, z2, z3);

endmodule
```



Full Adder

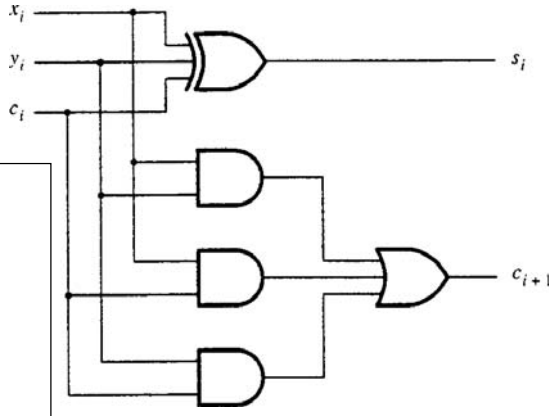
- Dataflow Specification—
  - Continuous assignment (version 1)




```
module fulladd (cout, s, cin, x, y);

    input cin, x, y;
    output cout, s;

    assign s = x ^ y ^ Cin;
    assign cout = (x & y) | (x & cin) | (y & cin);

endmodule
```


Full Adder

- Dataflow Specification—
  - Continuous assignment (version 2)

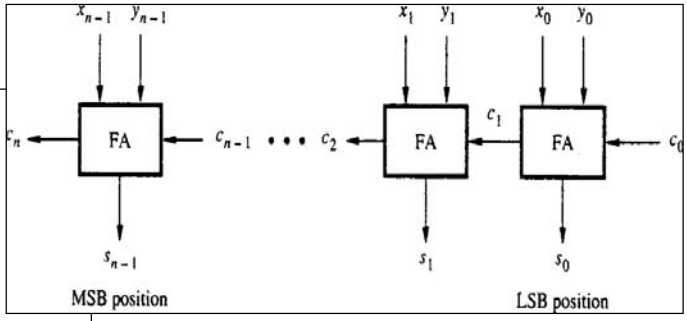



```
module fulladd (cout, s, cin, x, y);

    input cin, x, y;
    output cout, s;

    assign s = x ^ y ^ cin,
           cout = (x & y) | (x & cin) | (y & cin);

endmodule
```

4-Bit Adder

- Vectored signals

  input [3:0] W;
  wire [3:1] C;



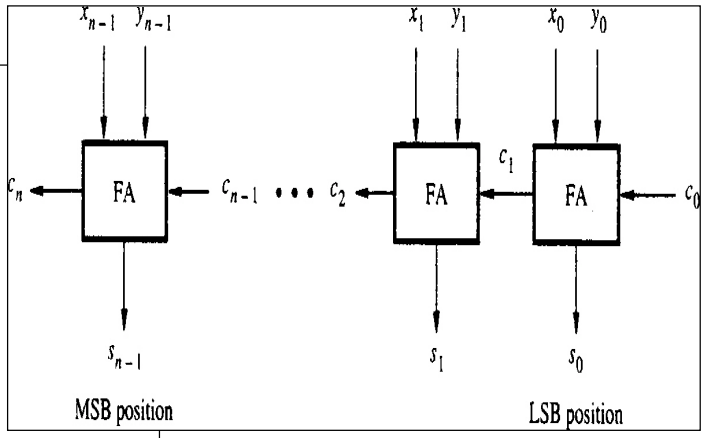

```
module adder4 (carryout, s, carryin, x, y);

   input carryin;
   input [3:0] x, y;
   output [3:0] s;
   output carryout;
   wire [3:1] c;

   fulladd stage0 (c[1], s[0], carryin, x[0], y[0]);
   fulladd stage1 (c[2], s[1], c[1], x[1], y[1]);
   fulladd stage2 (c[3], s[2], c[2], x[2], y[2]);
   fulladd stage3 (carryout, s[3], c[3], x[3], y[3]);

endmodule
```

ordered port connection

n-Bit Adder

- Parameter

  parameter n = 32;




```
module addern (carryout, s,  carryin, x, y);
   parameter N = 32;
   input carryin;
   input [N – 1 : 0] x, y;
   output [N – 1 : 0] s;
   output carryout;
   reg [N – 1 : 0] s;
   reg carryout;
   reg [n : 0] c;
   integer k;

   always @(x or y or carryin)
   begin
      c[0] = carryin;
      for (k = 0; k < n; k = k+1)
      begin
         s[k] = x[k] ^ y[k] ^ c[k];
         c[k + 1] = (x[k] & y[k]) | (x[k] & c[k]) | (y[k] & c[k]);
      end
      carryout = c[n];
   end
endmodule
```

Nets and Variables in Verilog

- Nets
  - wire— default type

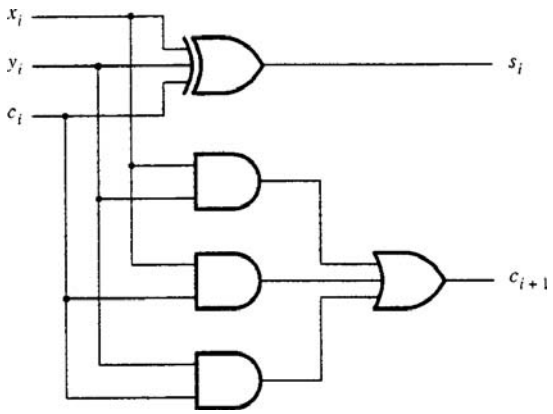
    wire c2, c1, c0;

    wire [2:0] c;
- Variables
  - reg
  - integer

Full Adder

- Behavioral Specification—
  - Procedural assignment

```
module fulladd (cout, s, cin, x, y);

    input cin, x, y;
    output s, cout;

    reg s, cout;

    always @(x or y or cin)
        {cout, s} = x + y + cin;

endmodule
```



n-Bit Adder

- Behavioral Specification—
  - Procedural assignment

```
module addern (s, carryin, x, y);

   parameter N = 32;
   input carryin;
   input [N–1 : 0] x, y;
   output [N–1 : 0] s;

   reg [N–1 : 0] s;

   always @(x or y or carryin)
      s = x + y + carryin;

endmodule
```


Representation of Numbers in Verilog Code

- Format—

  <size_in-bits>'<radix_identifier><significant_digits>

- Examples—
  - Fixed-sized
    - 12'b100010101001
    - 12'b1000_1010_1001
    - 12'b1000 1010 1001
    - 12'o4251
    - 12'h8A9
    - 12'd2217
  - Unspecified-sized
    - 'b100010110
    - 'b1_0001_0110
    - 'b1 0001 0110
    - 'o426
    - 'h116
    - 278

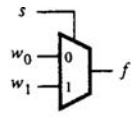
The Conditional Operator

- Syntax—

  Conditional_expression ? true_expression : false_expression

- Examples—

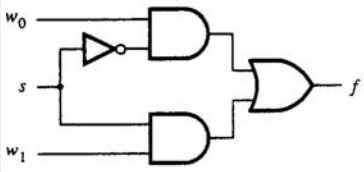
```
module mux2to1 (f, w0, w1, s);

   input w0, w1, s;
   output f;

   assign f = s ? w1 : w0;

endmodule
```



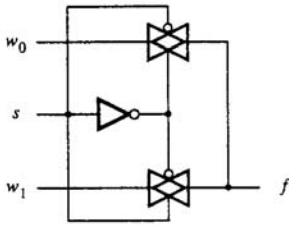
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table



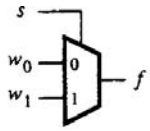
(c) Sum-of-products circuit



(d) Circuit with transmission gates

2-to-1 Multiplexer

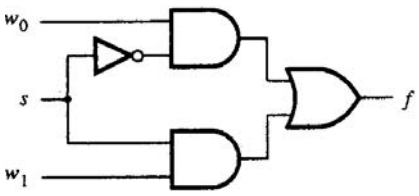
```
module mux2to1 (f, w0, w1, s);

   input w0, w1, s;
   output f;
   reg f;

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

endmodule
```



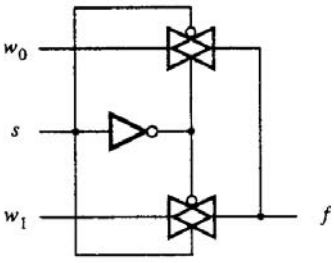
(a) Graphical symbol

s	f
0	w_0
1	w_1

(b) Truth table

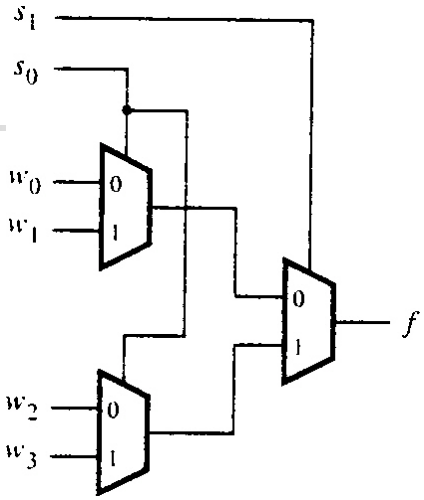


(c) Sum-of-products circuit



(d) Circuit with transmission gates

4-to-1 Multiplexer



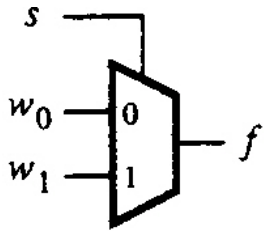

```
module mux4to1 (f, w0, w1, w2, w3, s);
   input w0, w1, w2, w3;
   input [1 : 0] s;
   output f;

   assign f = s[1] ? (s[0] ? w3 : w2) : (s[0] ? w1 : w0);

endmodule
```


The if-else Statement

- Syntax—

```
if (conditional_expression)
    statement;
else
    statement;
```

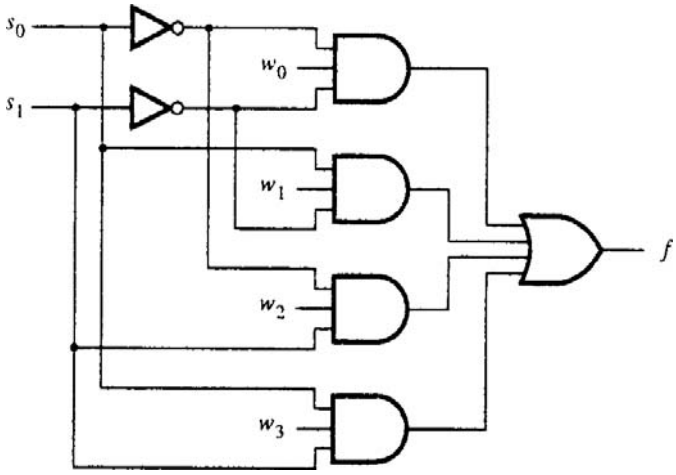
- Examples—




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

   input w0, w1, s;
   output f;
   reg f;

   always @(w0 or w1 or s)
      if (s == 0)
         f = w0;
      else
         f = w1;

endmodule
```

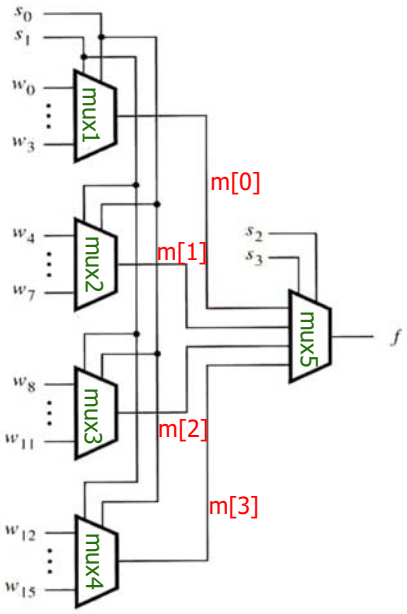
4-to-1 Multiplexer

```
module mux4to1 (f, w0, w1, w2, w3, s);

   input w0, w1, w2, w3;
   input [1:0] s;
   output f;
   reg f;

   always @(w0 or w1 or w2 or w3 or s)
      if (S  == 2'b00)
         f = w0;
      else if (s == 2'b01)
         f = w1;
       else if (s == 2'b10)
         f = w2;
      else if (s == 2'b11)
         f = w3;

endmodule
```



16-to-1 Multiplexer

```
module mux16to1 (f, w, s16);

   input [0 : 15] w;
   input [3 : 0] s16;
   output f;

   wire [0 : 3] m;

   mux4to1  mux1 (m[0], w[0:3], s16[1:0]);
   mux4to1  mux2 (m[1], w[4:7], s16[1:0]);
   mux4to1  mux3 (m[2], w[8:11], s16[1:0]);
   mux4to1  mux4 (m[3], w[12:15], s16[1:0]);
   mux4to1  mux5 (f, m[0:3], s16[3:2]);

endmodule
```



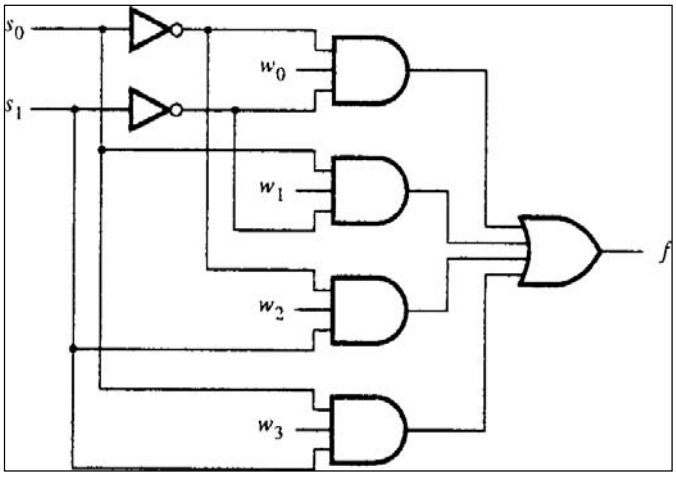
The case Statement

- Syntax—

```
case (expression)
    alternative1: statement;
    alternative2: statement;

    alternativej: statement;
    [default: statement;]
endcase
```


4-to-1 Multiplexer

```
module mux4to1 (f, w, s);

   input [0 : 3] w;
   input [1 : 0] s;
   output f;
   reg f;

   always @(w or s)
      case (s)
          0: f = w[0];
          1: f = w[1];
          2: f = w[2];
          3: f = w[3];
       endcase

endmodule
```



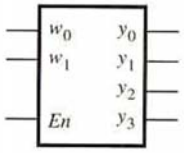
2-to-4 Decoder

- Example

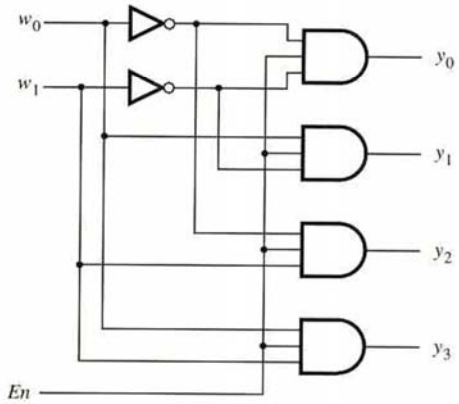
```
module dec2to4 (y, w, en);

   input [1 : 0] w;
   input en;
   output [0 : 3] y;

   reg [0 : 3] y;

   always @(w or en)
      case ({en, w})
         3'b100: y = 4'b1000;
         3'b101: y = 4'b0100;
         3'b110: y = 4'b0010;
         3'b111: y = 4'b0001;
        default: y = 4'b0000;
      endcase

endmodule
```

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table

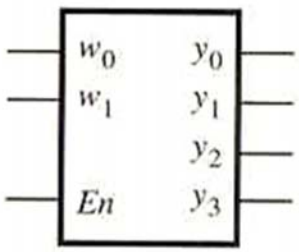


(b) Graphical symbol



(c) Logic circuit

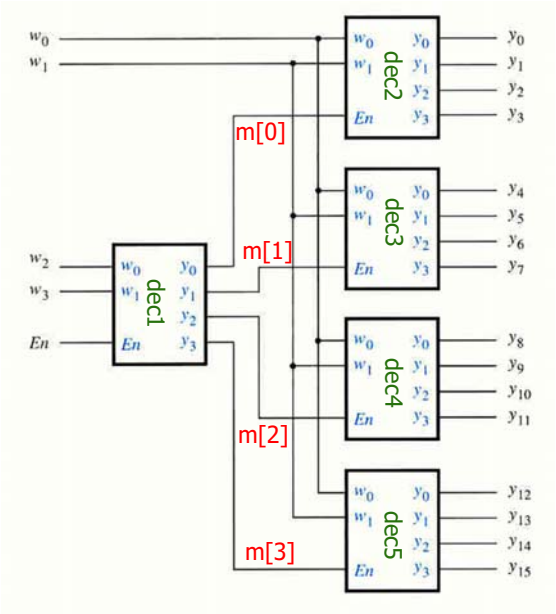
2-to-4 Decoder




```
module dec2to4 (y, w, en);

   input [1 : 0] w;
   input en;
   output [0 : 3] y;
   reg [0 : 3] y;

   always @(w or en)
   begin
      if (en == 0)
          y == 4'b0000;
      else
         case (w)
             0: y = 4'b1000;
             1: y = 4'b0100;
             2: y = 4'b0010;
             3: y = 4'b0001;
         endcase
   end

endmodule
```

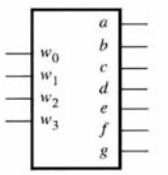
4-to-16 Decoder

```
module dec4to16 (y, w, en);

   input [3 : 0] w;
   input en;
   output [0 : 15] y;

   wire [0 : 3] m;

   dec2to4 dec1 (m[0:3], w[3:2], en);
   dec2to4 dec2 (y[0:3], w[1:0], m[0]);
   dec2to4 dec3 (y[4:7], w[1:0], m[1]);
   dec2to4 dec4 (y[8:11], w[1:0], m[2]);
   dec2to4 dec5 (y[12:15], w[1:0], m[3]);

endmodule
```

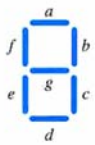


BCD-to-7-Segment Display Code Converter

```
module seg7 (leds, bcd);

   input [3:0] bcd;
   output [1:7] leds;
   reg [1:7] leds;

   always @(bcd)
      case (bcd)       //abcdefg
         0: leds = 7'b1111110;
         1: leds = 7'b0110000;
         2: leds = 7'b1101101;
         3: leds = 7'b1111001;
         4: leds = 7'b0110011;
         5: leds = 7'b1011011;
         6: leds = 7'b1011111;
         7: leds = 7'b1110000;
         8: leds = 7'b1111111;
         9: leds = 7'b1111011;
         default: leds = 7'bx;
      endcase

endmodule
```



(a) Code converter



(b) 7-segment display

w_3	w_2	w_1	w_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	1
0	1	0	0	0	1	1	0	0	1	1
0	1	0	1	1	0	1	1	0	1	1
0	1	1	0	1	0	1	1	1	1	1
0	1	1	1	1	1	1	0	0	0	0
1	0	0	0	1	1	1	1	1	1	1
1	0	0	1	1	1	1	1	0	1	1

(c) Truth table

74381 ALU

Operation	Inputs s_2 s_1 s_0	Outputs F
Clear	0 0 0	0 0 0 0
B−A	0 0 1	$B - A$
A−B	0 1 0	$A - B$
ADD	0 1 1	$A + B$
XOR	1 0 0	A XOR B
OR	1 0 1	A OR B
AND	1 1 0	A AND B
Preset	1 1 1	1 1 1 1

```
//74381 ALU
module alu (f, s, a, b);

   input [2:0] s;
   input [3:0] a, b;
   output [3:0] f;

   reg [3:0] f;

   always @(s or a or b)
      case (s)
         0: f = 4'b0000;
         1: f = b – a;
         2: f = a – b;
         3: f = a + b;
         4: f = a ^ b;
         5: f = a | b;
         6: f = a & b;
         7: f = 4'b1111;
         endcase

endmodule
```


casez and casex

- casez
  - treats all z values in the case alternatives and the controlling expression as don't cares
- casex
  - treats all z and x values as don't cares

4-to-2 Priority Encoder

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

   always @(w)
   begin
      z = 1;
      casex (w)
         4'b1xxx: y = 3;
         4'b01xx: y = 2;
         4'b001x: y = 1;
         4'b0001: y = 0;
         default:
         begin
               z = 0;
               y = 2'bx;
         end
      endcase
   end

endmodule
```

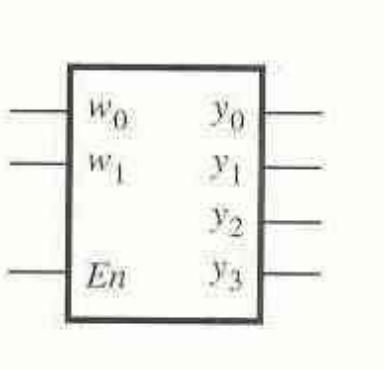
w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	d	d	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

The For Loop

- Syntax—

  for (initial_index; terminal_index; increment) statement;

- Examples—




```
module dec2to4 (y, w, en);

   input [1 : 0] w;
   input en;
   output [0 : 3] y;
   reg [0 : 3] y;
   integer k;

   always @(w or en)
      for (k= 0; k <= 3; k = k+1)
         if ((w == k) && (en == 1))
            y[k] = 1;
         else
            y[k] = 0;

endmodule
```

Priority Encoder

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

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

Verilog Operations (1/5)

Operator Type	Operator Symbols	Operation Perform
Bitwise	~	1 's complement
	&	Bitwise AND
	|	Bitwise OR
	^	Bitwise XOR
	~^ or ^~	Bitwise XNOR
Logical	!	NOT
	&&	AND
	||	OR

Verilog Operations (2/5)

Operator Type	Operator Symbols	Operation Perform
Reduction	&	Reduction AND
	~&	Reduction NAND
	|	Reduction OR
	~|	Reduction NOR
	^	Reduction XOR
	~^ or ^~	Reduction XNOR
Arithmetic	+	Addition
	-	Subtraction
	-	2's complement
	*	Multiplication
	/	Division

Verilog Operations (3/5)

Operator Type	Operator Symbols	Operation Perform
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
Equality	==	Logical equality
	!=	Logical inequality
Shift	>>	Right shift
	<<	Left shift
Concatenation	{,}	Concatenation
Replication	{{}}	Replication
Conditional	?:	Conditional

Verilog Operations (4/5)

- Truth tables for bitwise operators

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

~^	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Verilog Operations (5/5)

Operator Type	Operator Symbols	Precedence
Complement	! ~ -	Highest precedence
Arithmetic	* / + -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== !=	
Reduction	& ~& ^ ~^ | ~|	
Logical	&& ||	
Conditional	?:	Lowest precedence

Tasks and Functions

- Task

```
module mux16to1 (f, w, s16);
   input [0 : 15] w;
   input [3 : 0] s16;
   output f;
   reg f;

   always @(w or S16)
      case (s16[3:2])
         0: mux4to1 (w[0:3], s16[1:0], f);
         1: mux4to1 (w[4:7], s16[1:0], f);
         2: mux4to1 (w[8:11], s16[1:0], f);
         3: mux4to1 (w[12:15], s16[1:0], f);
      endcase
```

```
// Task that specifies a 4-to-1 multiplexer
   task mux4to1;
      input [0:3] x;
      input [1:0] s4;
      output g;
      reg g;

      case (s4)
         0: g = x[0];
         1: g = x[1];
         2: g = x[2];
         3: g = x[3];
      endcase
   endtask

endmodule
```


Tasks and Functions

- Function

```
module mux16to1 (f, w, s16);
   input [0 : 15] w;
   input [3 : 0] s16;
   output f;
   reg f;

   // Function that specifies a 4-to-1 multiplexer
   function mux4to1;
      input [0 : 3] x;
      input [1 : 0] s4;

      case (s4)
         0: g = x[0];
         1: g = x[1];
         2: g = x[2];
         3: g = x[3];
      endcase
   endfunction
```

```
   always @(w or s16)
      case (s16[3:2])
         0: mux4to1 (w[0:3], s16[1:0]);
         1: mux4to1 (w[4:7], s16[1:0]);
         2: mux4to1 (w[8:11], s16[1:0]);
         3: mux4to1 (w[12:15], s16[1:0]);
      endcase

endmodule
```

Gated D Latch

```
module d_latch (q, d, clk);

    input d, clk;
    output q;
    reg q;

    always @(q or clk)
        if (clk)
            q = d;

endmodule
```


D Flip-Flop

```
module flipflop (q, d, clk);
    input d, clk;
    output q;
    reg q;

    always @(posedge clk)
        q = d;

endmodule
```

Blocking (Procedural) Assignment vs. Non-Blocking (Concurrent) Assignment

Blocking assignment

```
always @(x)
begin
   count = 0;
   for (k = 0; k < n; k = k + 1)
      count = count + x[k];
end
```

```
count = 0 + x[0];
count = x[0] + x[1];
count = x[1] + x[2];
```

Non-blocking assignment

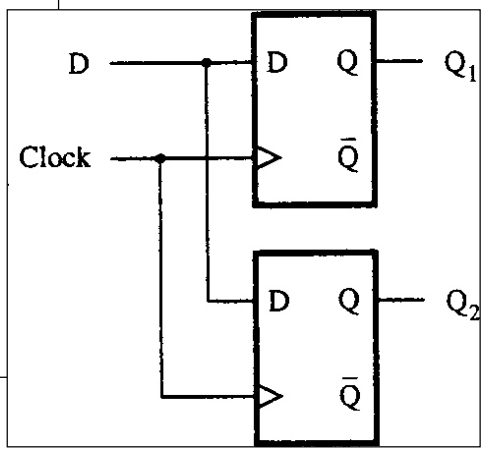
```
always @(x)
begin
   count = 0;
   for (k = 0; k < n; k = k + 1)
      count <= count + x[k];
end
```

```
count = 0 + x[0];
count = 0 + x[1];
count = 0 + x[2];
```


Two Parallel Flip-Flops

- Blocking (procedural) assignment

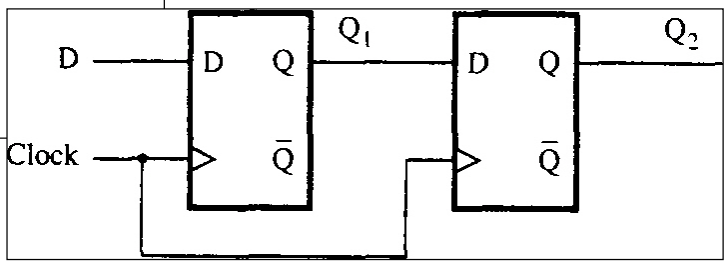
```
module example7_3 (q1, q2, d, clk);

   input d, clk;
   output q1, q2;
   reg q1, q2;

   always @(posedge clk)
   begin
      q1 = d;
      q2 = q1;
   end

endmodule
```



Two Cascaded Flip-Flops

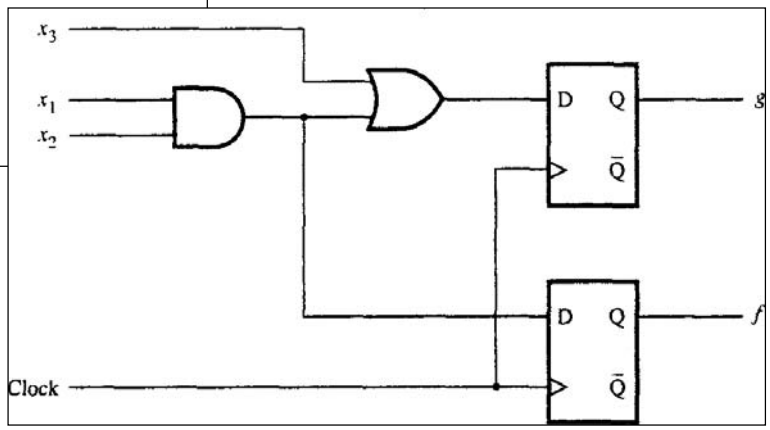
- Non-blocking (concurrent) assignment

```
module example(q1, q2, d, clk);
   input d, clk;
   output q1, q2;
   reg q1, q2;

   always @(posedge clk)
   begin
      q1 <= d;
      q2 <= q1;
   end

endmodule
```



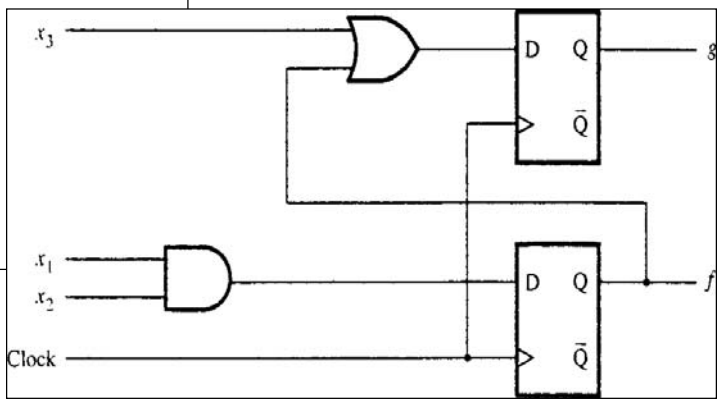
Example for Blocking (Procedural) Assignment

```
module example(f, g, x1, x2, x3, clk);
   input x1, x2, x3, clk;
   output f, g;
   reg f, g;

   always @(posedge clk)
   begin
      f = x1 & x2;
      g = f | x3;
   end

endmodule
```



Example for Non-Blocking (Concurrent) Assignment

```
module example (f, g, x1, x2, x3, clk);

   input x1, x2, x3, clk;
   output f, g;

   reg f, g;

  always @(posedge clk)
   begin
      f <= x1 & x2;
      g <= f | x3;
   end

endmodule
```



n-Bit Register

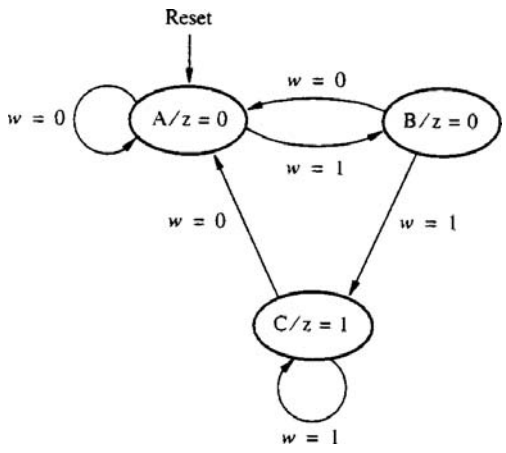
Synchronous Reset

```
module regn (q, d, clk, rst_n);

   parameter N = 16;
   input clk, rst_n;
   input [N – 1 : 0] d;
   output [N – 1 : 0] q;
   reg [N – 1 : 0] q;

   always @(posedge clk)
      if(!rst_n)
         q <= 0;
      else
         q <= d;

endmodule
```

Asynchronous Reset

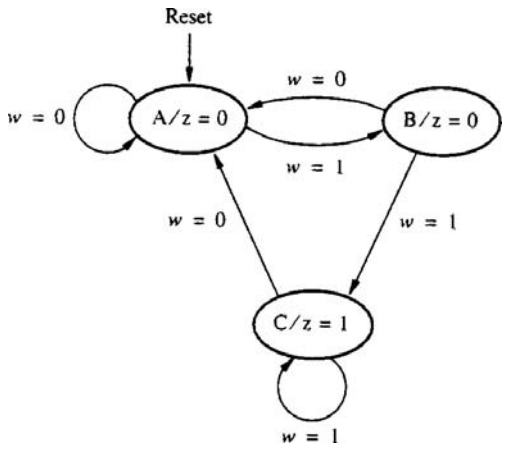
```
module regn (q, d, clk, rst_n);

   parameter N = 16;
   input clk, rst_n;
   input [N – 1 : 0] d;
   output [N – 1 : 0] q;
   reg [N – 1 : 0] q;

   always @(negedge rst_n or posedge clk)
      if(!rst_n)
         q <= 0;
      else
         q <= d;

endmodule
```

Moore-Type Finite State Machine



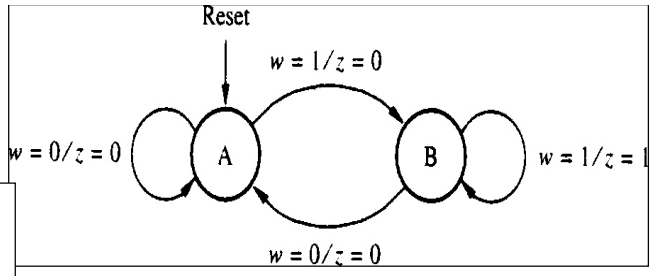

```
module moore (z, clk, w, rst_n);
   input clk, w, rst_n;
   output z;
   reg [1:0] y;
   parameter [1:0] a = 2'b00, b = 2'b01, c = 2'b10;

   always @(w or y)
   begin
      case (y)
         a:   if (w == 0) y = a;
              else        y = b;
         b:   if (w == 0) y = a;
              else        y = c;
         c:   if (w == 0) y = a;
              else        y = c;
         default:         y = 2'bxx;
      endcase
   end
   always @(posedge clk or negedge rst_n)
   begin
      if (rst_n == 0)
         y <= a;
      else
         y <= y;
   end
   assign z = (y == c)
endmodule
```


Moore-Type Finite State Machine




```
module moore (z, clk, w, rst_n);
   input clk, w, rst_n;
   output z;
   reg [1:0] state, next_state;
   parameter [1 : 0] a = 2'b00, b = 2'b01, c = 2'b10;

 always @(posedge clk or negedge rst_n)
      if (rst_n == 0)
         state <= a;
      else
         state <= next_state;

   always @(posedge clk or negedge rst_n)
   begin
      if (rst_n == 0)
         next_state <= a;
      else
         case (y)
            a:   if (w == 0) next_state <= a;
                 else        next_state <= b;
            b:   if (w == 0) next_state <= a;
                 else        next_state <= c;
            c:   if (w == 0) next_state <= a;
                 else        next_state <= c;
            default:         next_state <= 2'bxx;
         endcase
   end

   assign z = (y == c);
endmodule
```

Mealy-Type FSM

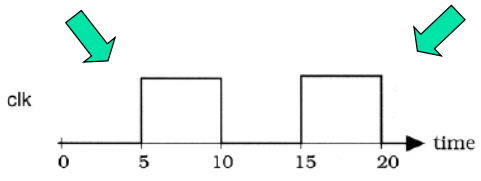



```
module mealy (z, clk, w, rst_n);

  input clk, w, rst_n;
  output z;
  reg [1:0] state, next_state;
  parameter a = 0, b = 1;

 always @(posedge clk or negedge rst_n)
      if (rst_n == 0)
         state <= a;
      else
         state <= next_state;

always @(w or state)
   begin
      case (state)
         a:   if (w == 0)
              begin
                  next_state = a; z = 0;
              end
              else
              begin
                 next_state = b; z = 0;
              end
         b:   if (w == 0)
              begin
                 y = a;
                 z = 0;
              end
              else
              begin
                 y = b;
                 z = 1;
              end
      endcase

endmodule
```


How to Generate Clock Signal in Testbenches?

```
initial
   begin
      clk = 1'b0;
      #5
      clk = 1'b1;
      #5
      clk = 1'b0;
      #5
      clk = 1'b1;
      $finish;
   end
```

```
initial
   begin
      clk = 1'b0;
      #20
      $finish;
   end
always
      #5 clk = ~clk;
```



Formal and Actual Names for Port Association by Name

```
module fulladd (cout, s, cin, x, y);
   input cin, x, y;
   output cout, s;

   assign s = x ^ y ^ cin;
   assign cout = (x & y) | (x & cin) | (y & cin);
endmodule
```

formal_name

```
module adder4 (carryout, s, carryin, x, y);
   input carryin;
   input [3:0] x, y;
   output [3:0] s;
   output carryout;
   wire [3:1] c;

   fulladd stage0 (c[1], s[0], carryin, x[0], y[0]);
   fulladd stage1 (c[2], s[1], c[1], x[1], y[1]);
   fulladd stage2 (c[3], s[2], c[2], x[2], y[2]);
   fulladd stage3 (carryout, s[3], c[3], x[3], y[3]);
endmodule
```

actual_name

.formal name (actual name)

```
(.cout (c[1]), .s (c[0]), .cin (carryin),
.x (x[0]), .y (y[0]))
```


always Blocks

```
always @(sensitivity_list)
   [begin]
      [procedural assignment statements]
      [if-else statements]
      [case statements]
      [for loops]
      [task and function calls]
   [end]
```

References / Sources

- Charles H. Roth, Jr., Larry L Kinney, *Fundamentals of Logic Design*, 7th Ed., Cengage Learning , 2014. (ISBN-13: 978-0-13-246557-1)
- Samir Palnitkar, *Verilog HDL*, 2nd. Ed., Prentice Hall, 2003. (ISBN-13: 978-0-13-259970-2)
- Michael D. Ciletti, *Advanced Digital Design with the Verilog HDL*, 2nd Ed., Prentice Hall, 2010. (ISBN-13: 978-0-13-246557-1)