Verilog HDL tutorial in Arabic

by Ahmed Hany

Verilog Introduction

Hardware Description Language

- Software Programming Language
  - Executed into a piece of instructions to CPU then on the computer
  - Sequential in nature and executed a piece of code sequentially
  - Example : C,C++,python

- Hardware Description Language
  - HDL is often used to model a digital circuit which synthesized to hardware
  - HDL is concurrent in nature and executed a piece of code in parallel
  - Example : VHDL, Verilog

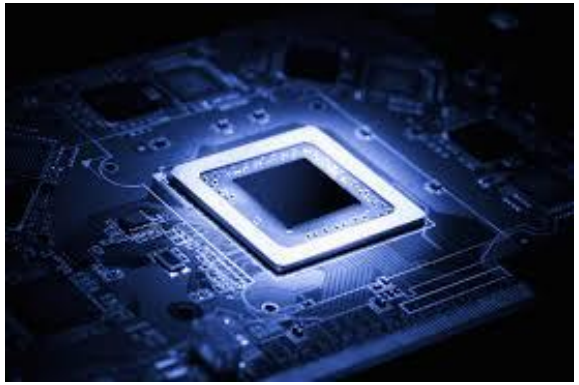
VHDL VS VERILOG

- VHDL
  - More verbose than Verilog but it is difficult to use
  - More natural to use at time
  - Most FPGA designs done in VHDL

FPGA

- Verilog
  - More compact than Verilog but it is easy to use
  - Similar in syntax to C programming
  - Most ASIC designs done in verilog

ASIC



VERILOG HISTORY

- Introduced in 1984 by gateway design automatic
- In 1989 cadence purchased gateway and open the standardization
- In 1995 Verilog became IEEE standard 1364
- In 2001 second IEEE standard of Verilog
- In 2005 new IEEE standard of Verilog
- In 2009 Verilog and systemverilog standards merged in standard 1800

Content of tutorial

- Verilog module
- Verilog coding style
- Verilog data types
- Verilog data operators
- Verilog conditions
- Verilog loops
- Verilog simulation
- Some examples

Tools

- Xilinx ISE
  - Previous tutorial : FPGA Design in arabic (attached in description)
- EDA playground
  - a free web application that allows users to edit, simulate, share, synthesize, and view waves for hardware description language (HDL) code.

Module Introduction

- Module definition

  module name and description
- Module interface

  port and parameter declaration
- Module body

  module functionality

```
# Definition
module name
# interface
( ) ;
# body
.
.
endmodule
```

Module Definition

- Module name
  - Case sensitive
  - Reversed words in lowercase
  - Starts with alphabetics or “_”
- Whitespace is used for readability
- Semicolon is the state terminator
- //… : single line comment
- /* .. */ : multi-line comment

```
/*
//new module
*/
module name
.
.
.
endmodule
```

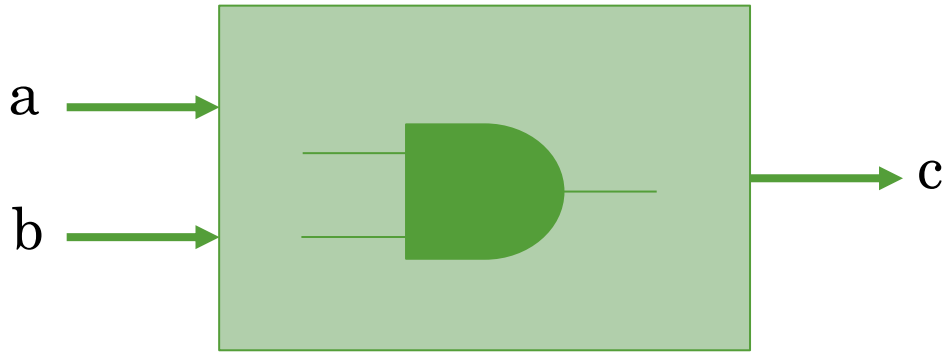
Module interface

- Port types : Input , Output and Inout
- data types : wire , reg and others
  - Input and inout ports are of type wire
  - Output port can be wire or reg
- Signal width
- Signal name


```
/*
//new module
*/
module name
(
input   wire [1:0]  a,
input   wire [1:0]  b,
output  reg  [1:0]  c
);
.
endmodule
```

Module Body

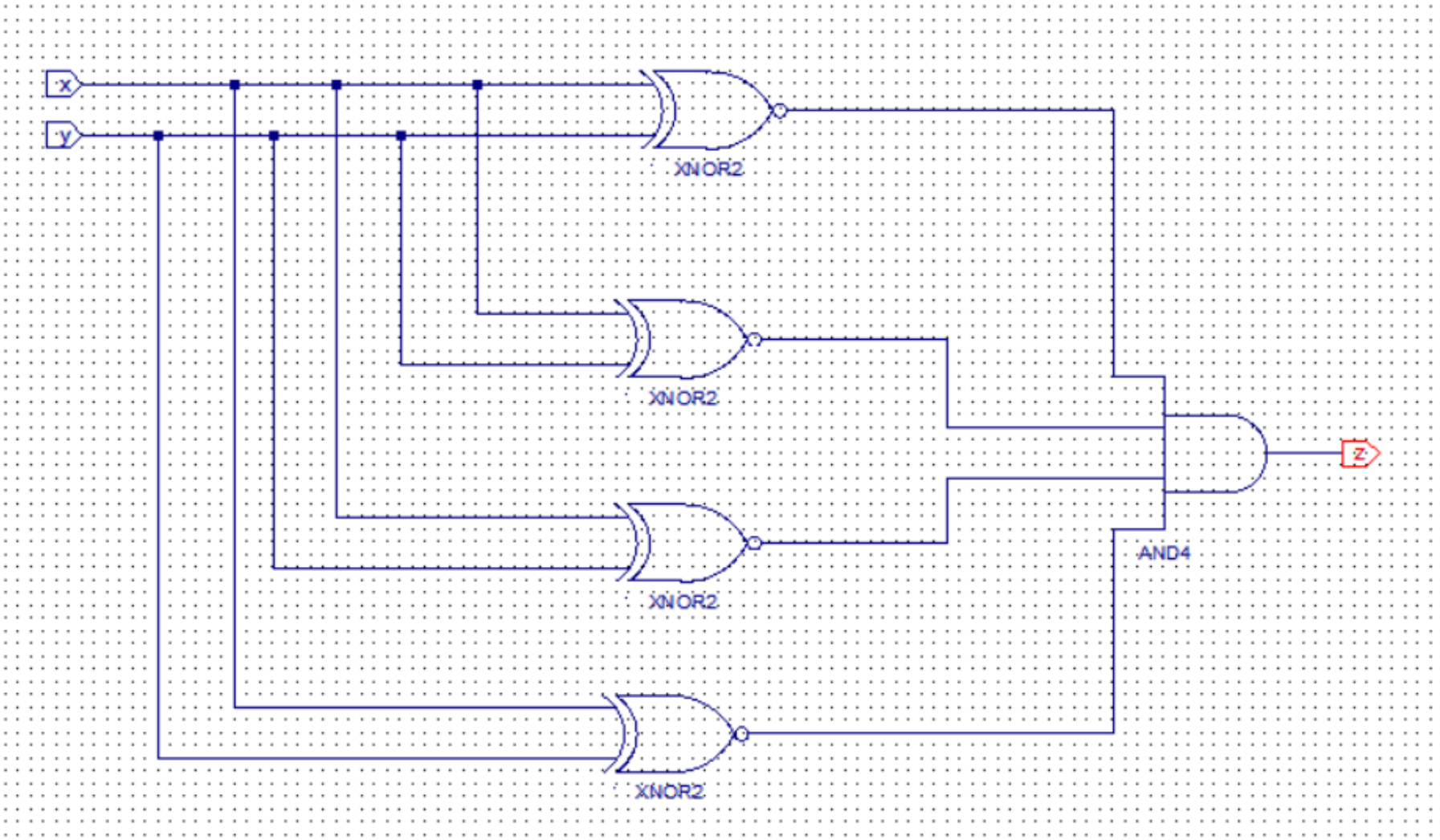
- Module function
- RTL coding style
  - Behavioral
  - Structural
  - Data flow




```
/*
//new module
*/
 module name
(
input wire [1:0] a,
input wire [1:0] b,
output reg [1:0]c
);
 assign c=a & b;
 endmodule
```

Coding Styles

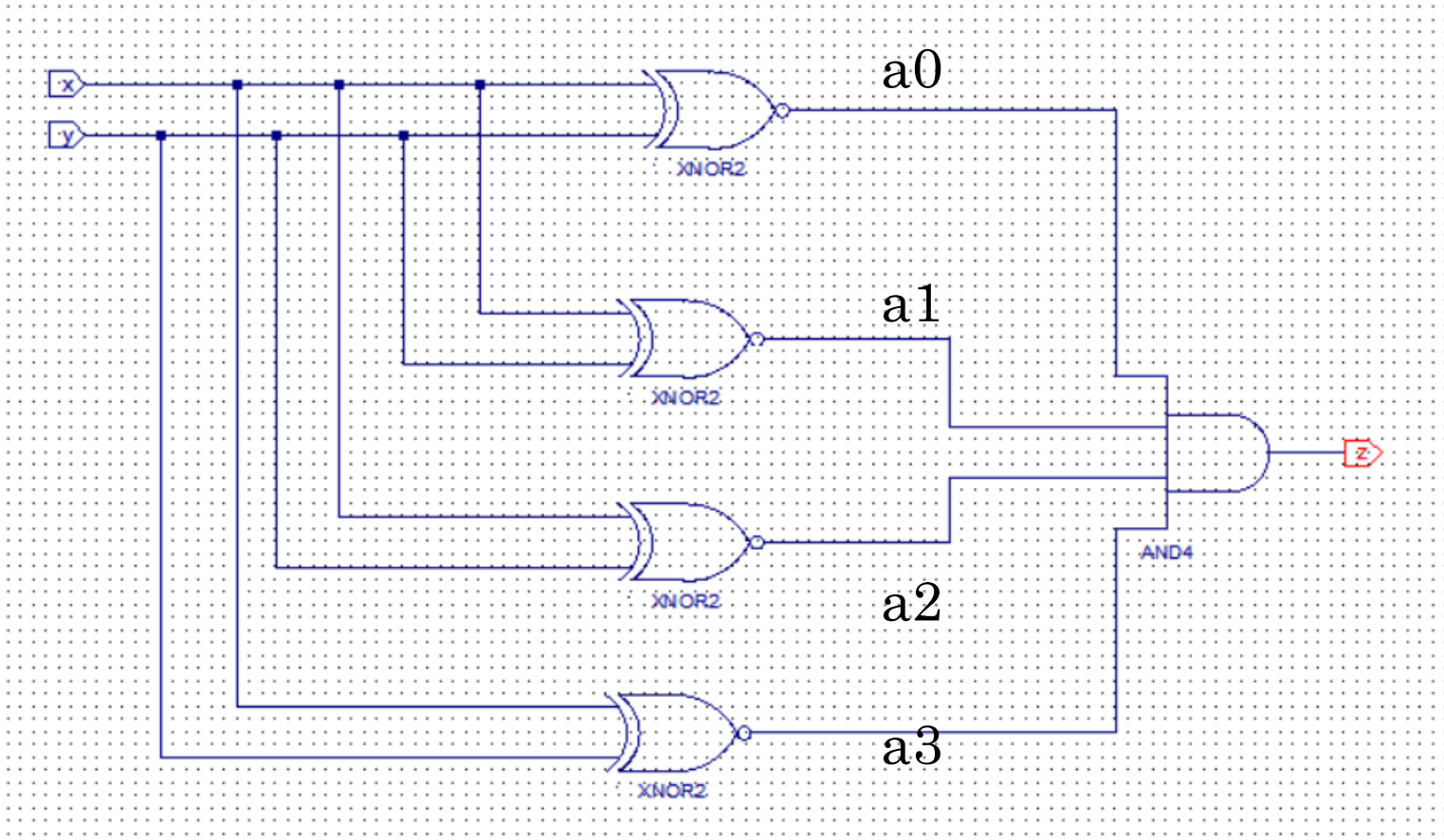
in1	in2	xor	xnor
0	0	0	1
0	1	1	0
1	0	1	0
1	1	0	1



Coding Styles

- Structural modeling (gate-level)
  - Use predefined or user-defined primitive gates
- Logic gates
  - and (output,input,..)
  - or (output,input,..)
  - xor (output,input,..)
  - nand (output,input,..)
  - nor (output,input,..)
  - xnor (output,input,..)
- Buffer and inverter gates
  - buf (output,input)
  - not (output,input)
- Tristate logic gates
  - bufif0 (output,input,enable)
  - notif0 (output,input,enable)
  - bufif1 (output,input,enable)
  - notif1 (output,input,enable)

Coding Styles




```
module comparator
(
 input wire [3:0] x,y,
 output wire z
);
 wire a0,a1,a2,a3;
xnor (a0,x[0],y[0] );
xnor (a1,x[1],y[1] );
xnor (a2,x[2],y[2] );
xnor (a3,x[3],y[3] );
and (z,a0,a1,a2,a3 );
endmodule
```

Coding Styles

- Dataflow modeling
  - Verilog allows a circuit to be designed in terms of the data flow between registers and how a design processes data
  - Describing the module in terms of logical equation
  - Using assignment statements (assign)

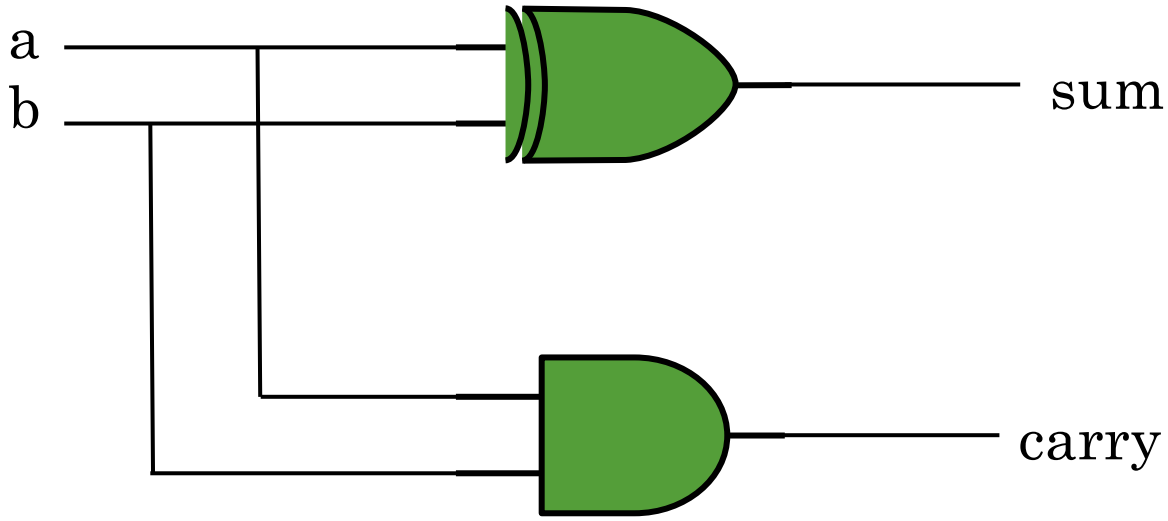
```
module comparator
(
 input wire [3:0] x,y,
 output wire z
);
 assign z = &((x ~^y ));
endmodule
```

Coding Styles

- behavioral modeling
  - The topmost abstraction layer in Verilog
  - Does not show the structural details
  - Used in sequential logic
  - Using procedural statements (always)

```
module comparator
(
 input wire [3:0] x,y,
 output reg z
);
always @ (x or y)
begin
z=0;
if (x ==y)
z=1;
end
endmodule
```

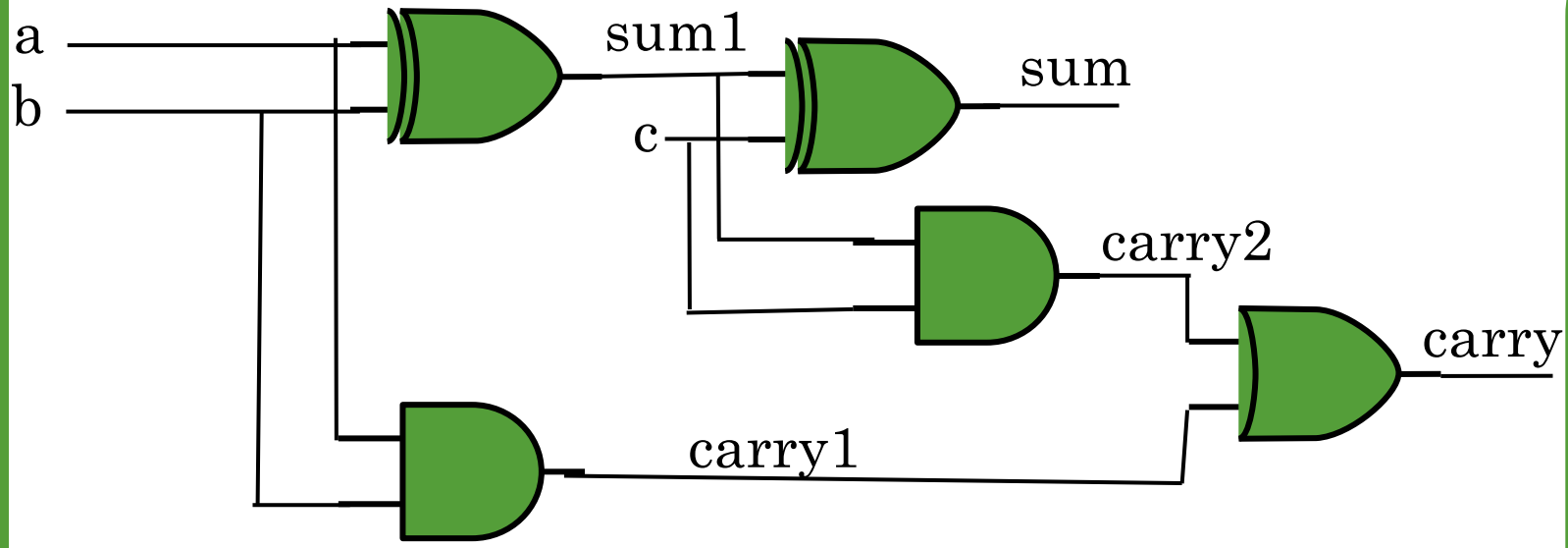
Half adder



in1	in2	Sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```
module halfadder
(
 input a,b,
 output sum,carry
);
assign sum=a^b;
assign  carry= a&b;
endmodule
```

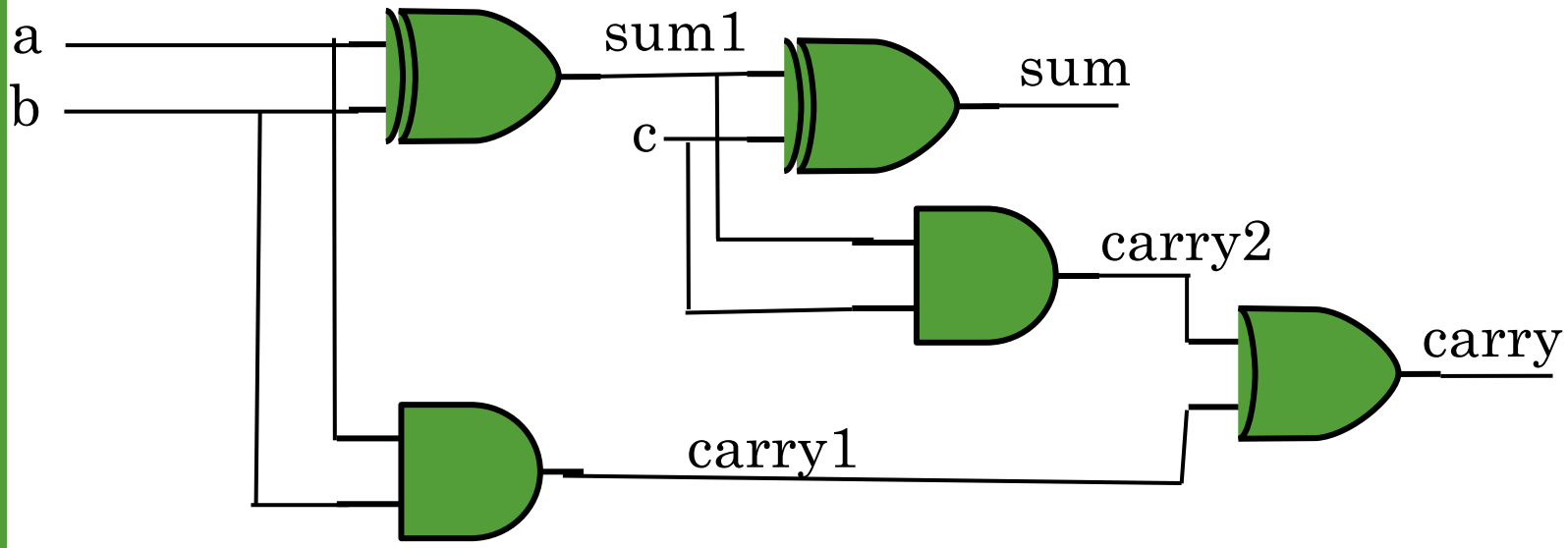
Full adder



in1	in2	in3	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

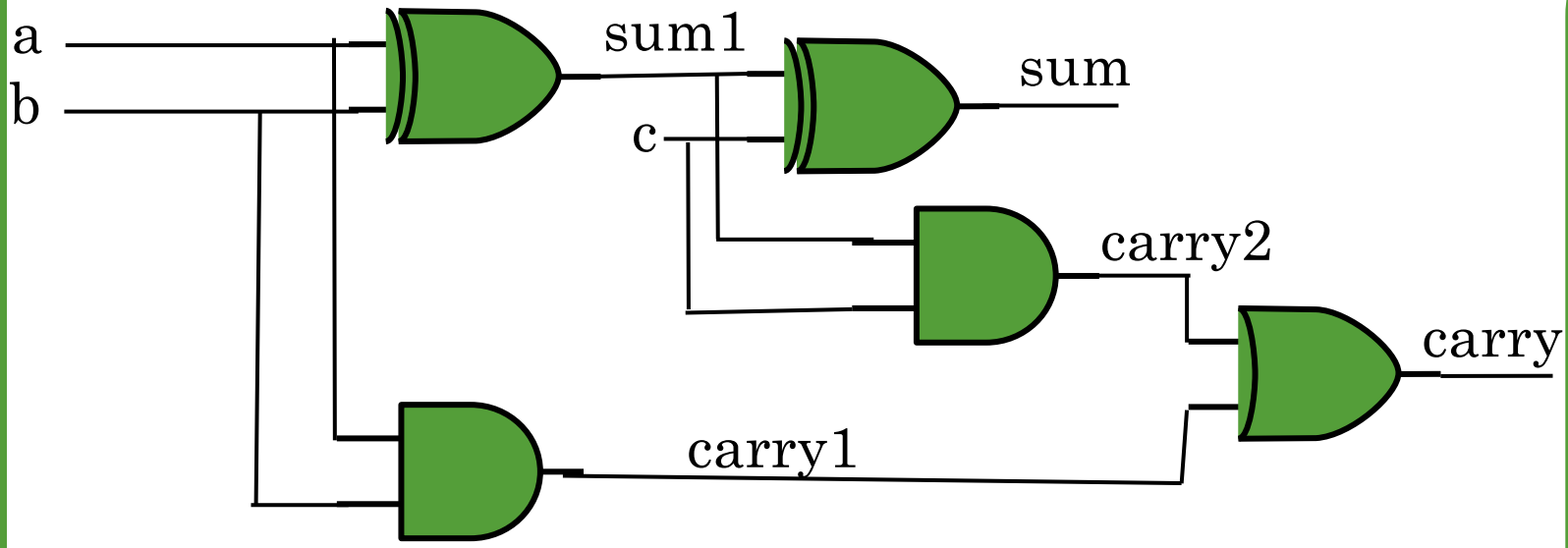
```
module fulladder
(
input  a, b, c ,
output sum, carry
    );
wire sum1,carry1,carry2;
xor  (sum1 ,a ,b);
 and (carry1,a ,b );
 xor (sum ,sum1 ,c );
and(carry2,sum1,c)
or  (carry ,carry1,carry2);
endmodule
```

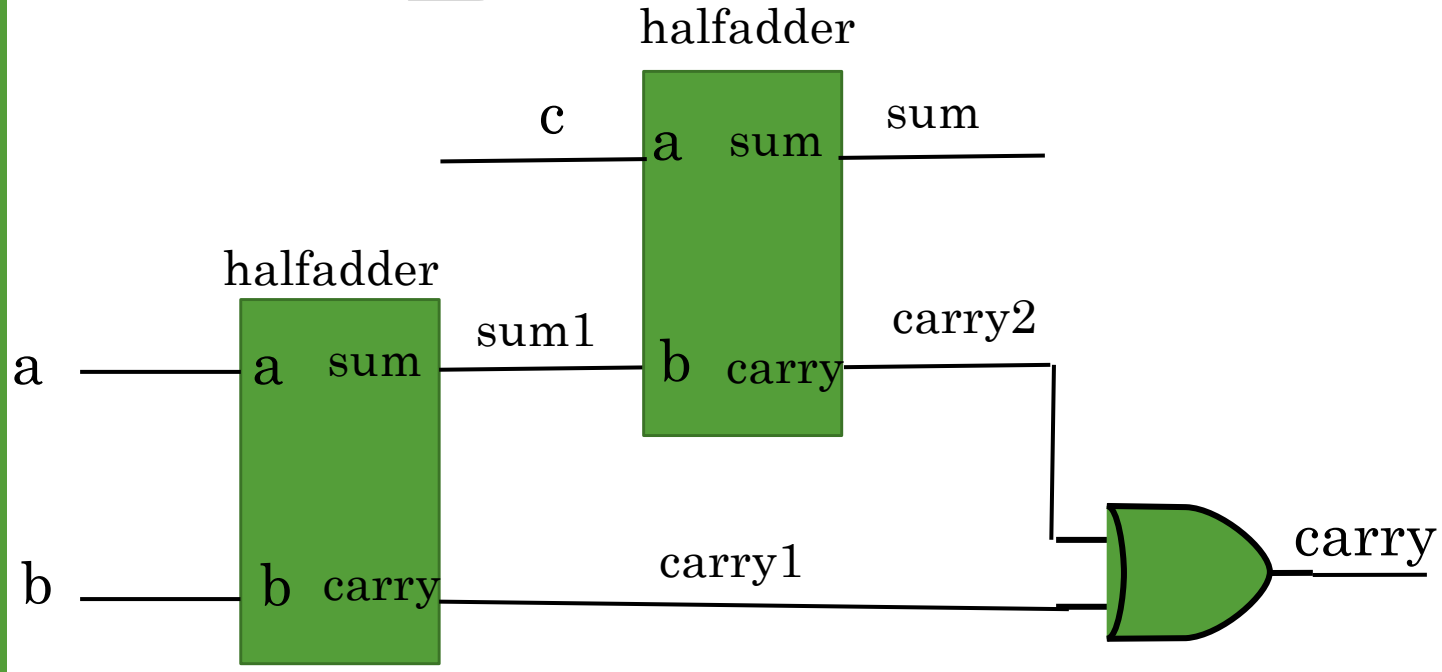
Full adder




```
module fulladd (
 input  a,
 input  b,
input c,
 output carry,
  output  sum
);
   assign {carry, sum} = a
+ b + c;
endmodule
```

Full adder






```
module fulladder(

input a,b,c,

 output sum,carry

);

wire s1,c1,c2;

halfadder ha1(

.a(a),.b(b),.sum(s1)  ,.carry(c1) );

halfadder ha2(

.a(c),.b(s1),.sum(sum)  ,.carry(c2)
);

assign carry =c2|c1;

endmodule
```

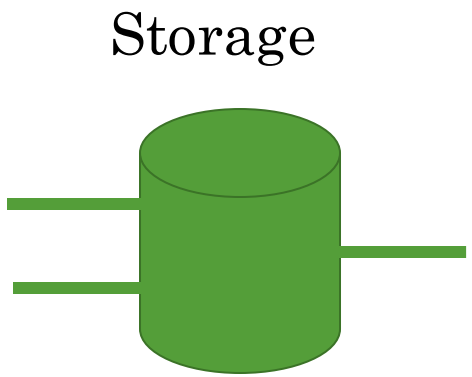
Data values

- 0
  - Zero
  - Logic zero
  - Logic false
- 1
  - One
  - Logic one
  - Logic true
- x
  - Unknown value
  - uninitialized value
- z
  - High impedance value
  - Floating value

Data Types

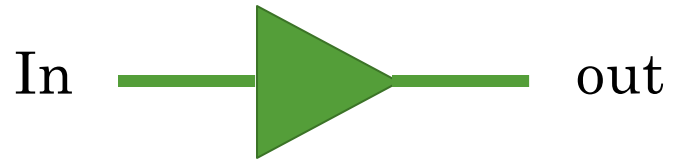
- Net data type
  - Represents physical interconnect between hardware elements
  - Must be driven continuously

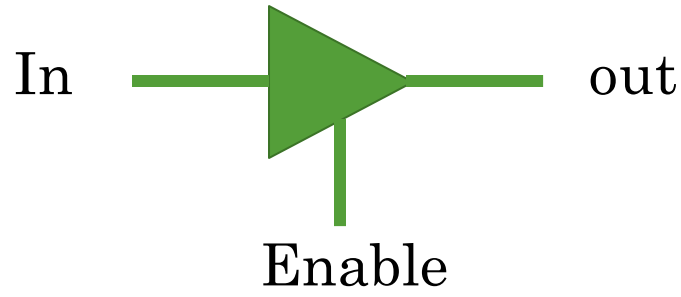
- Variable data type
  - Represents element to store data



Net Data Types

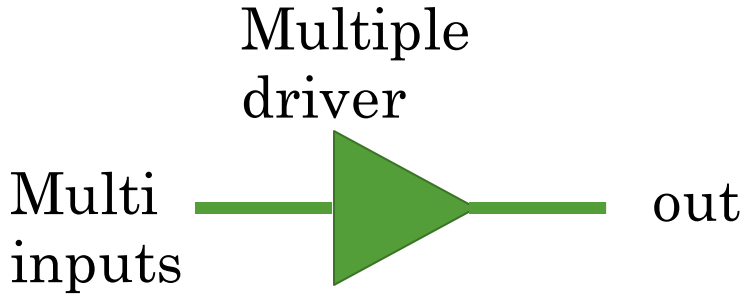
- Wire
  - Driven by a single continuous assignment
  - A default data type is wire scalar
- Wire Tri
  - If enable =1 -> out = in
- Wor trior
  - When any driver is 1 -> out =1





Wire tri	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

wor	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z



Net Data Types

- Wand triand
  - When any driver is 0 -> out =0
- Tri0
  - If enable =0 -> out = in
- Tri1
  - If enable =1 -> out = in

Wand	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

Tri0	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Tri1	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Variable Data Types

- Reg
  - Unsigned variable of any bit size(scalar/vector)
  - Only reg type variable can be assigned to in procedural statements (always or initial or ..)
  - Only net type variable can be assigned to in statements (assign)
  - Reg does not mean register. It can be modeled as a storage cell
  - Use reg sign for signed implementation
- Integer : signed 32-bit variable
- Real , time and realtime : no synthesis support

```
module comparator
(
 input wire  [3:0] x,y,
 output reg z
);
always @ (x or y)
begin
z=0;
If (x ==y)
z=1;
end
endmodule
```

Variable Data Types

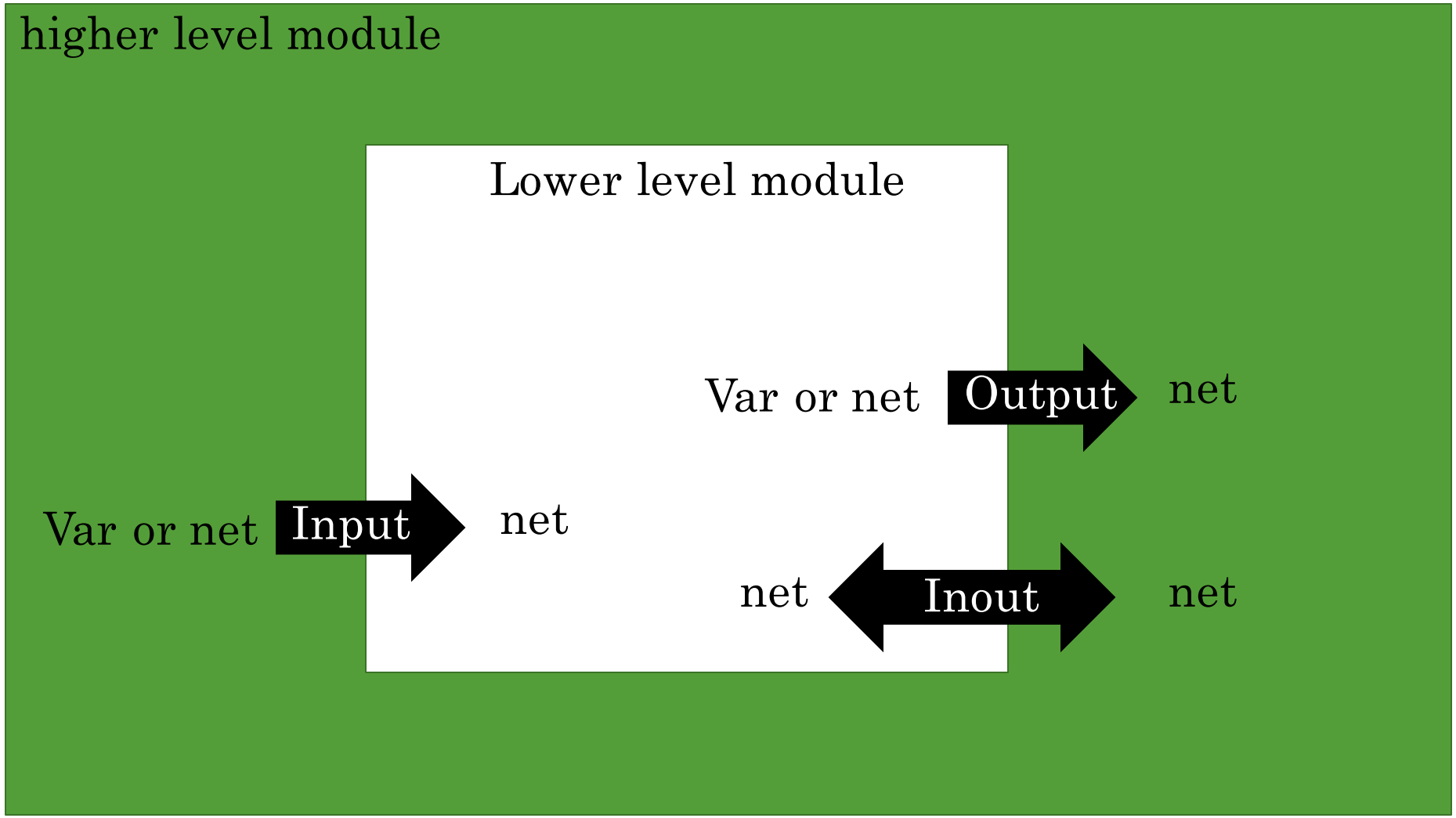
- Parameter
  - Giving name to constant value
  - Must resolve to constant at compile time
- localparam
  - Same as parameter but cannot be overwritten

Variable Data Types

```
module getsum(x,y,z,sum);
        parameter height = 3;
        parameter width = 4;
        localparam length = 5;
        input [width-1:0] x;
        input [height-1:0] y;
        input [length-1:0] z;
        output [width-1:0] sum;
      assign sum = x + y + z;
endmodule
```

```
module getsub
#(      parameter height = 3,
        parameter width = 4
) (x,y,z,sub);
input [width-1:0] x;
        input [height-1:0] y;
        input [width-1:0] z;
        output [width-1:0] sub;
      assign sub = z-y-x;
endmodule
```

Variable Data Types



Numbers Representation

- Numbers are sized and unsized
- General format : <sign> <size>‘ <base> <num>
- Sized numbers
  - 4’b1010 = 4-bit wide binary number
- Unsized numbers
  - 325 = 32-bit wide decimal number by default
  - Base is decimal by default
  - Size is 32-bit by default

Numbers Representation

- Base formats
  - Decimal : 16'd250 = 16-bit wide Decimal number 0000000011111010
  - Hexadecimal : 8'Hed = 8-bit wide Hexadecimal number 11101101
  - Binary : 'B101 = 32-bit wide Binary number 00000000000000000000000000000101
  - Octal : 8'o22 = 8-bit wide Octal number 00010010
  - Signed : 16'Shab = 16-bit wide Hexadecimal number 0000000010101011
- When size is smaller than value, then leftmost bits of value are neglegtable
- When size is larger than value, then leftmost bits are filled with :
  - '0' : if bit in value is '0' or '1'
  - 'x' : if leftmost bit in value is 'x'
  - 'z' : if leftmost bit in value is 'z'

Numbers Representation

- Examples
  - 15'oz20 = 15-bit wide Octal number — zzzzzzzzz010000
  - 32'Hab_37_56_df = 32-bit wide Hexadecimal number — 10101011001101110101011011011111
  - 12'h12x = 12-bit wide Hexadecimal number — 00010010xxxx
  - 'H18zx = 32-bit wide Hexadecimal number — 00000000000000000011000zzzzxxxx
  - 8'd256 = 8-bit wide Decimal number — 00000000
  - 'ox = 32-bit wide Octal number — xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

Numbers Representation

```
module number_representation;

reg [31:0]  a;

initial begin

  a = 15'oz20;

  $display ("Current Value of a = %b", a);

  a = 32'hab_37_56_df;

  $display ("Current Value of a = %b", a);

  a = 12'h12x;

  $display ("Current Value of a = %b", a);

  a = 'h18zx;

  $display ("Current Value of a = %b", a);

   a = 8'd256;

  $display ("Current Value of a = %b", a);

   a = 'ox;

  $display ("Current Value of a = %b", a);

  $finish;

end

endmodule
```

Logical Values

- Any non-zero value is true
- Anything else is false (include ‘x’ and ‘z’)
- Examples
  - 2’b0 : false
  - 3’b111 : true
  - 4’bx : false
  - 4’b 110x : true
  - 3’b 101 : true
  - 5’b 000x : false

bit-wise operators

- '~' : not each bit
- '&' : and each bit
- '|' : or each bit
- '^' : xor each bit
- Examples
  - ~4'b10xz : 4'b01xx
  - 3'b10x & 3'b111 : 3'b10x
  - 3'b10x & 3'b110 : 3'b100
  - 3'b10x | 3'b110 : 3'b11x
  - 3'b10x | 3'b111 : 3'b111
  - 3'b10x ^ 3'b110 : 3'b01x

- ~x = x
- 0&x = 0
- 1&x = x&x = x
- 1|x = 1
- 0|x = x|x = x
- 0^x = 1^x = x^x = x
- 0^~x = 1^~x = x^~x = x

relational operators

- '>' : greater than
- '>=' : greater than or equal
- '<' : less than
- '<=' : less than or equal
- '==' : equal
- '!=' : not equal
- '===' : identical
  - 'x' and 'z' must be identical
- '!==' : not identical

relational operators

- Examples
  - 4'b1101 >= 4'b1001 true
  - 4'b1001 == 4'b1001 true
  - 5'b1001x == 5'b1001x false
  - 5'b1001x != 5'b1001x false
  - 5'b1001x === 5'b1001x true
  - 5'b1001x !== 5'b1001x false

Logical operators

- '!' : logical not
- '&& ' : logical and
- '| |' : logical or
- The result is
  - 0 : if the relation is false
  - 1 : if the relation is true
  - x : if any of the operands has 'x'
- Examples
  - 1'b0 && 1'b1 false(0)
  - 1'b0 || 1'b1 true(1)
  - ! 1'b1 false(0)
  - ! 1'bx x

Arithmetic operators

- ‘+’ : add
- ‘- ‘ : subtract
- ‘*’ : multiply
- ‘/’ : divide
- ‘**’ : exponent
- The result is ‘x’ : if any of the operands has ‘x’
- Examples (a=2,b=4)
    - a+b=6
    - a-b=-2
    - b/a=2
    - a**4=16

Reduction operators

- '&' : and all bits
- '| ' : or all bits
- '^' : xor all bits
- Examples (x=4'b0101 , y=4'b01xz , z=4'b000z)

&x=1'b0	&y=1'b0	&z=1'b0
~&x= 1'b1	~&y= 1'b1	~&z= 1'b1
|x= 1'b1	|y= 1'b1	|z= 1'bx
~|x= 1'b0	~|y= 1'b0	~|z= 1'bx
^x= 1'b0	^y= 1'bx	^z= 1'bx
~^x= 1'b1	~^y= 1'bx	~^z= 1'bx

- ~x = x
- 0&x = 0
- 1&x = x&x = x
- 1|x = 1
- 0|x = x|x = x
- 0^x = 1^x = x^x = x
- 0^~x = 1^~x = x^~x = x

Data operators

```
module operators ;
 initial begin
 //Reduction operators
$display("&4'b0101= %b", &4'b0101);
   // relationship operators
$display (" 5'b1001x ===  5'b1001x= %b", (5'b1001x === 5'b1001x));
$display (" 1'bz  <=  10 = %b", (1'bz  <= 10));
 // Bit Wise operators
 $display (" ~4'b10xz= %b", (~4'b10xz));
// Logical operators
$display ("1'b1 || 1'b0 = %b", (1'b1 || 1'b0));
  // arithmetic operators
$display ("2**4 = %d", 2**4); end
endmodule
```

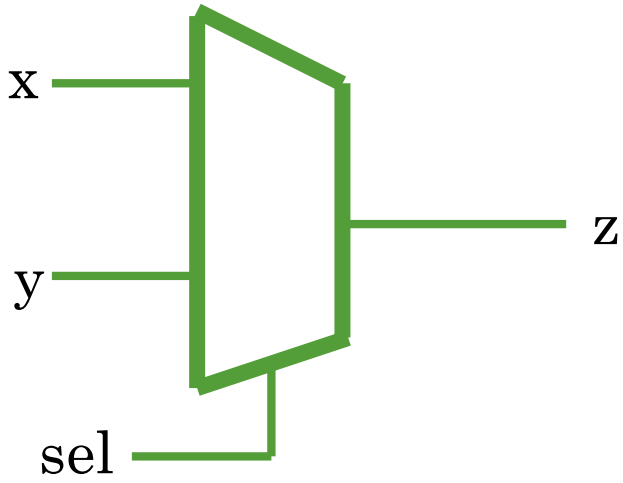
Shift operators

- ‘ <<’ : logical shift left
- ‘ >>’ : logical shift right
- ‘<<<’ : arithmetic shift left (keep sign)
- ‘>>>’ : arithmetic shift right (keep sign)
- Vacated position filled with zeros except at arithmetic shift right filled with sign bit (MSB)
- Examples (a=3’b100)
    - a << 2 = 3’b000
    - a <<< 2 = 3’b000
    - a >> 2 = 3’b001
    - a >>> 2 = 3’b111

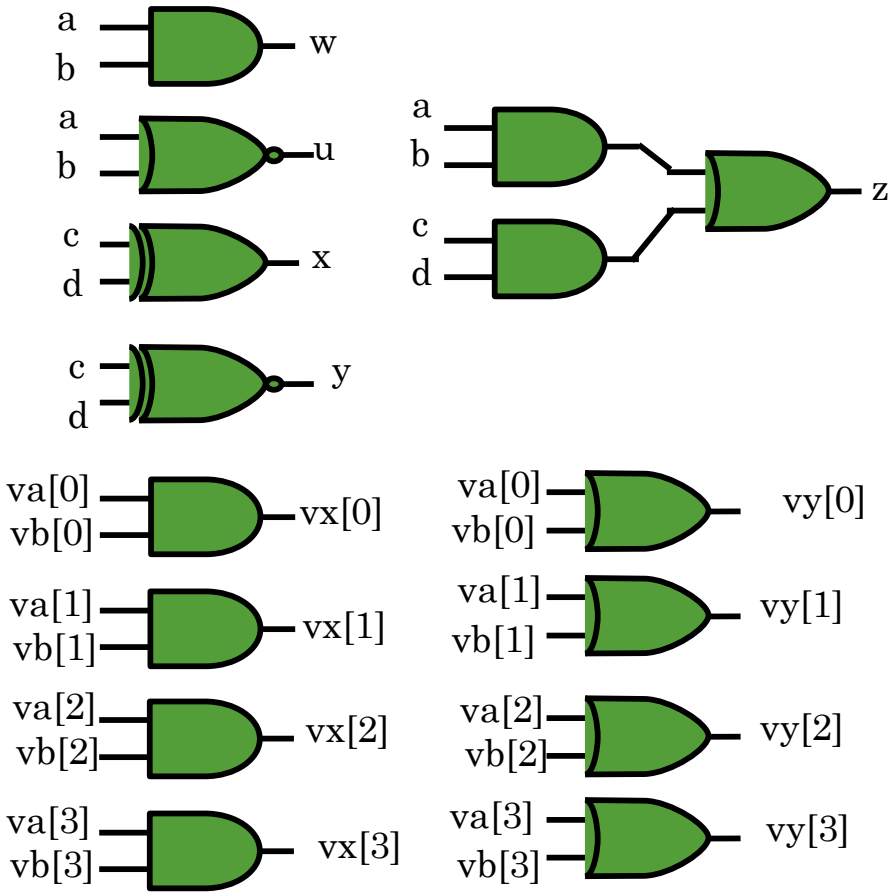
Miscellaneous operators

- ‘ ?’ : conditional operator
  - Condition ? True value : false value
  - z= (sel =1) ? x : y
- ‘{ }’ : concatenation operator
  - combination of two or more expressions
  - assignment z={ x[2:0] , y[1:0] }
- ‘{ { } } : replication operator

  {4{2’b10}} = 8’b10101010



Gates assign




```
module gates(
    input a, b, c, d,
    input [3:0] va, vb,
 output  w, u, x, y, z,
output [3:0] vx, vy
);
    assign w=a &b;
 assign u=~(a |b);
 assign x=c ^ d;
 assign y=c~ ^ d;
 assign z=(a &b) |
(c&d);
assign vx=va &vb;
assign vy=va |vb;
endmodule
```

Continuous assignment statements

- Model the behavior of logic using some expressions
  - Left hand side must be a net data type
  - right hand side can be a net or variable
  - Delay values can be assigned to model gate delay

```
assign #5 out = in1 & in2
```

Procedural assignment blocks

- Initial block
  - A program that runs only once. Used for simulation

```
initial
clk=1'b0;
```

- Always block
  - A program that stuck in an infinite loop

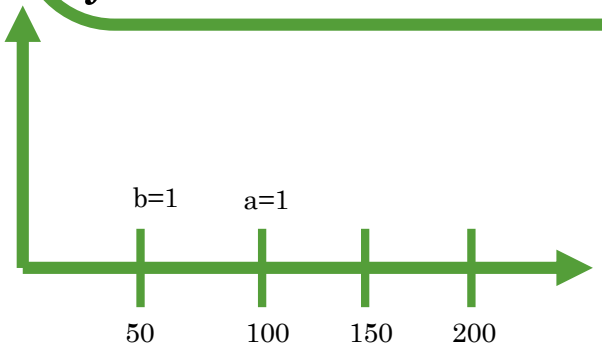
```
always
#50 clk=~clk;
```

- processes run in parallel and start simulation time 0
- statements inside a process execute sequentially

Procedural assignment blocks

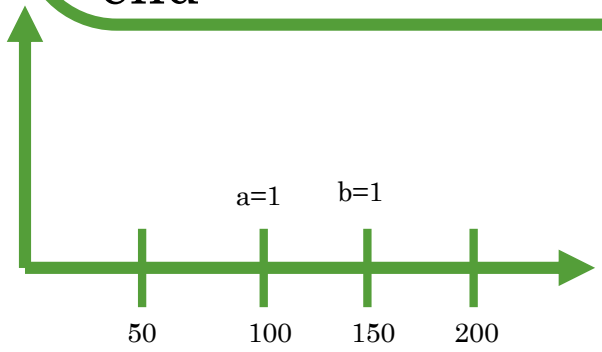
- Begin/end
  - Used to group statements, so that they execute in sequence
- Fork/join
  - Groups statements into a parallel block, so that they are executed concurrently.

```
always @(posedge clk)
fork
#100 a=1;
#50 b=1;
join
```



- the @ statement is used to wait for one or more events

```
always @(posedge clk)
begin
#100 a=1;
#50 b=1;
end
```



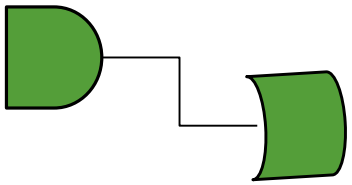
Procedural assignment blocks

- Begin/end vs fork/join

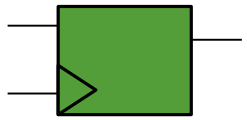
```
initial
   being
    #5;                          5
      fork
       #9;                       14
          being
           #9;                   14
           #9;                   23
               fork
                #4;              27
                #6;              29
               join
          end
     #16                         21
     join
   end
```

Procedural assignment

- Blocking procedural assignment (=)
  - RHS is executed and assignment is completed before the next statement is executed
- Non-blocking procedural assignment (<=)
  - RHS is executed and assignment takes place at the end of the current time step(not clock cycle)
- Use blocking procedural assignment in combinational always block



- Use non-blocking procedural assignment in sequential always block



```
// assume x=5
x=8;
y=x;
//x=8,y=8
```

```
// assume x=5
x<=8;
y<=x;
//x=8,y=5
```

Testbench

- `timescale time unit / time precision
  - `timescale 10ns/1ns
- Reg and wire declarations
  - Inputs in testbench are reg type
  - Outputs in testbench are wire type
- Instantiate unit under test
- Initial and always block

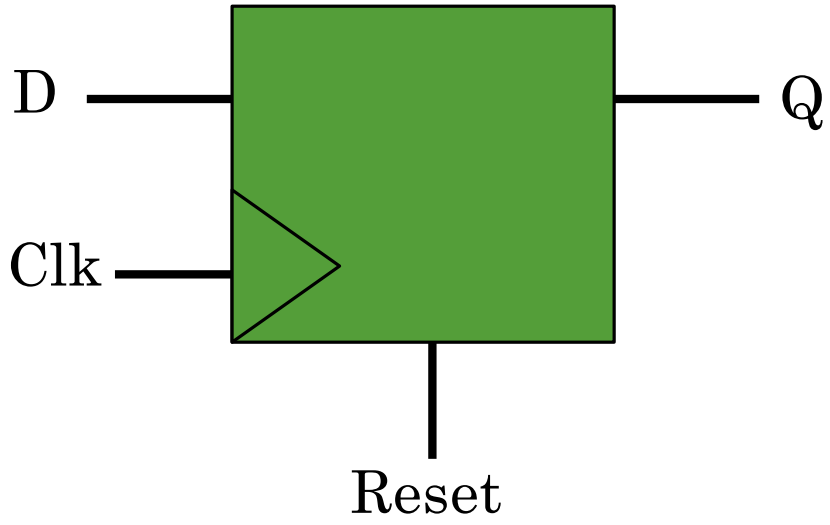
```
#1;      10ns
#0.5;    5ns
#0.02;   1ns
```

```
reg Input_sig;
wire Output_sig;
```

```
code uut(
.Input(Input_sig),
.Output(Output_sig)
);
```

D Flip flop

- The basic element memory
- If reset =1 -> Q=0
- If positive edge clk -> Q=D




```
module DFF
(
 input wire d,clk,rst,
 output reg q
);
always @ (posedge clk)
begin
if (rst == 1'b1)
q<=0;
else
q<=d;
end
endmodule
```

DFF test bench

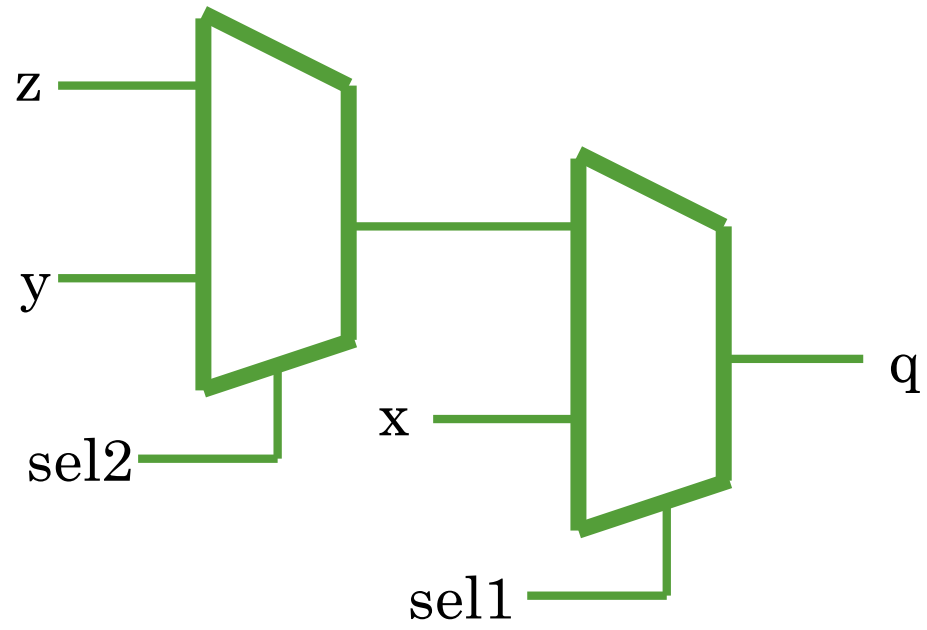
```
`timescale 1ns / 1ps

module dff_tb;

// Inputs

reg d;

reg clk;

reg rst;

// Outputs

wire q;

// Instantiate the Unit Under Test (UUT)

DFF uut (

.d(d),

.clk(clk),

.rst(rst),

.q(q)

);

// Initialize Inputs

initial begin

d = 0;

clk = 0;

rst = 0;

end

always

#50 clk=~clk;

always

#50 rst = 0;

always

#100 d=~d;

endmodule
```

Behavioral statements

- must be used inside a procedural block
- behavioral statements
  - if-else statement
    - conditions are evaluated in order from top to bottom
  - Case statement
    - conditions evaluated at once
  - loop statements
    - used for repetitive operations

If-else statement

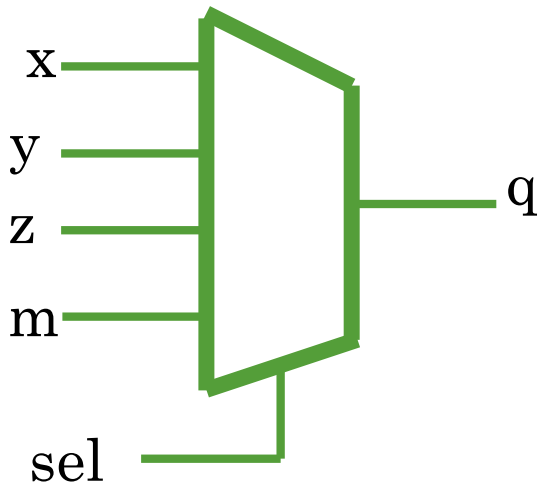
- a sequential statement that conditionally executes other sequential statements, depending upon the value of some condition.




```
always
begin
if (sel1)
q=x;
else if (sel2)
q=y;
else
q=z;
end
```

Case statement

A statement which conditionally executes at most one branch, depending on the value of the case expression.




```
always
begin
case (sel)
2'b00: q=x;
2'b01: q=y;
2'b10: q=z;
default: q=m;
endcase
end
```

Case statement

- Casez
  - treats both z and ? in the case conditions as don't cares
- Casex
  - treats z,x and ? in the case conditions as don't cares

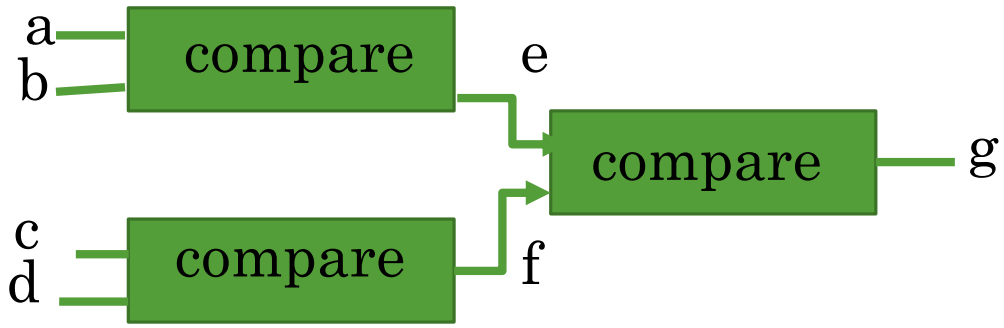
```
casez (sel)
5'b00001: q=a;
5'b0001?: q=b;
5'b001??: q=c;
5'b01???,
5'b1????: q=d;
default: q=e;
endcase
```

5'b00101	c
5'b00z01	a
5'b00?01	a
5'b00x01	e

```
casex (sel)
5'b00001: q=a;
5'b0001?: q=b;
5'b001??: q=c;
5'b01???,
5'b1????: q=d;
default: q=e;
endcase
```

5'b00101	c
5'b00z01	a
5'b00?01	a
5'b00x01	a

comparators




```
module compare(
    input a, b, c, d,
output reg  g
);
reg e,f;
    always @(a or b or c or d)
begin
if(a>b)
e=a;
 else
e=b;
if(c>d)
f=c;
else
f=d;
if(e>f)
g=e;
else
g=f;
end
endmodule
```

Compare test bench

```
`timescale 1ns / 1ps
module compare_tb;
// Inputs
reg a;
reg b;
reg c;
reg d;
// Outputs
wire g;
// Instantiate the Unit Under Test (UUT)
compare uut (
.a(a),
.b(b),
.c(c),
.d(d),
.g(g)
);
initial begin
a = 0;
b = 0;
c = 0;
d = 0;
end
always begin
#50 a=~a;
end
always begin
#100 b=~b;
end
always begin
#200 c=~c;
end
always begin
#400 d=~d;
end
endmodule
```

Loop

- While loop
  - A loop statement that repeats a statement or block of statements as long as a controlling expression remains true
- For loop
  - General purpose loop statement. Allows one or more statements to be executed iteratively

```
initial
begin
for(cnt=0;cnt<10;cnt=cnt+1) begin
q[cnt]=d[cnt];
end
end
```

```
initial
begin
cnt=0;
while( cnt<10) begin
cnt=cnt+1;
q[cnt]=d[cnt];
end
end
```

Loop

- wait statement
  - If the condition is already true then execution carries on immediately.
  - wait(rst=1'b1) cnt=0;
- forever statement
  - a block of code that will run continuously
  - forever begin a=~a; end
- repeat statement
  - block of code some defined number of times
  - repeat(3) begin a=~a; end

And scalar

```
module and_scalar (
    input x,
    input [3:0] d,
 output  reg [3:0] q
);
    reg [3:0] tmp;
integer i;
always @ (d or x)
begin
for(i=0;i<4;i=i+1)
begin
 tmp[i]=d[i] &x;
end
q= tmp;
end
endmodule
```

And testbench

```
`timescale 1ns / 1ps
module and_tb;
// Inputs
reg x;
reg [3:0] d;
// Outputs
wire [3:0] q;
// Instantiate the Unit Under Test (UUT)
and_scalar uut (
.x(x),
.d(d),
.q(q)
);
initial begin
d = 0;
forever begin #50 d=4'b1010; end
end
initial begin
x = 0;
repeat(8)  begin  #50 x=~x; end
end
endmodule
```