

Verilog HDL

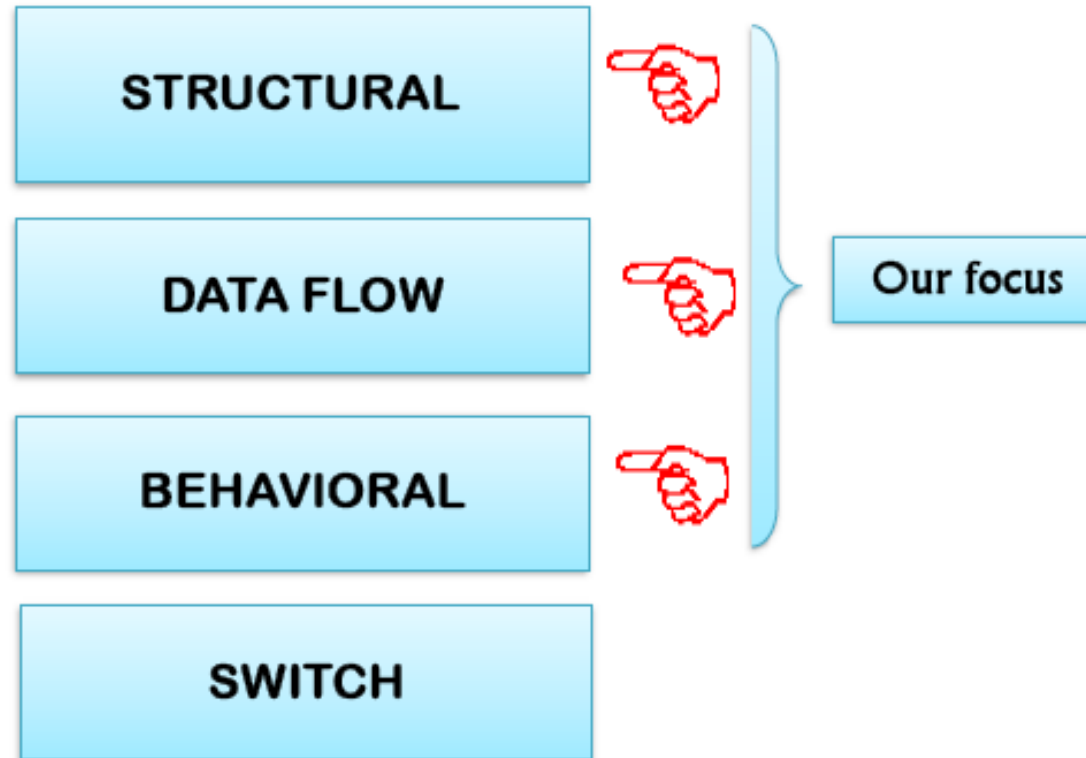
The Verilog Language

- Originally a modeling language for a very efficient event-driven digital logic simulator
 - Later pushed into use as a specification language for logic synthesis
 - Now, one of the two most commonly-used languages in digital hardware design (VHDL is the other)
 - Virtually every chip (FPGA, ASIC, etc.) is designed in part using one of these two languages
 - Combines structural and behavioral modeling styles
-

Concurrency

- Verilog or any HDL has to have the power to model concurrency which is natural to a piece of hardware.
 - There may be pieces of two hardware which are even independent of each other.
 - Verilog gives the following constructs for concurrency:
 - always
 - assign
 - module instantiation
 - non-blocking assignments inside a sequential block
-

VERILOG MODELING



STRUCTURE OF VERILOG HDL

module **module_name** (Port List)

Declaration of regs and other variables

- Dataflow statement (**assign**)
- **always** or **initial** blocks.
- Instantiation of lower level modules
- Tasks and functions

endmodule

LOGICAL OPERATORS

□ $\&\&$ \rightarrow logical AND

□ \parallel \rightarrow logical OR

□ $!$ \rightarrow logical NOT

□ Operands evaluated to ONE bit value: 0 , 1 or x

□ Result is ONE bit value: 0 , 1 or x

$A = 1;$

$B = 0;$

$C = x;$

$A \&\& B \rightarrow 1 \&\& 0 \rightarrow 0$

$A \parallel !B \rightarrow 1 \parallel 1 \rightarrow 1$

$C \parallel B \rightarrow x \parallel 0 \rightarrow x$

RELATIONAL OPERATORS

□ $>$ → greater than

□ $<$ → less than

□ $>=$ → greater or equal than

□ $<=$ → less or equal than

□ Result is one bit value: 0, 1 or x

$1 > 0 \rightarrow 1$

$'b1x1 <= 0 \rightarrow x$

$10 < z \rightarrow x$

EQUALITY OPERATORS

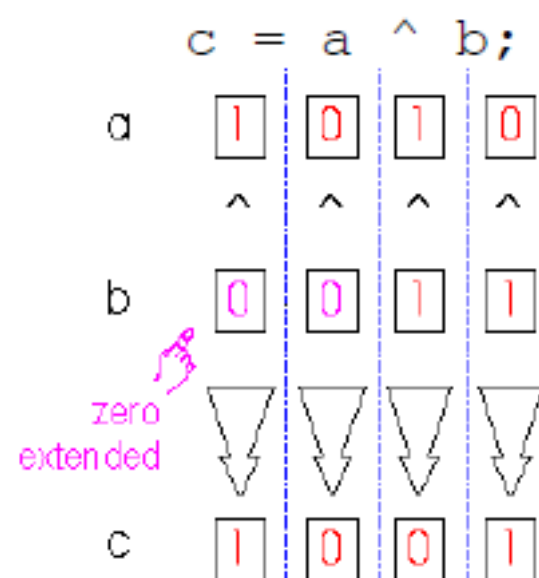
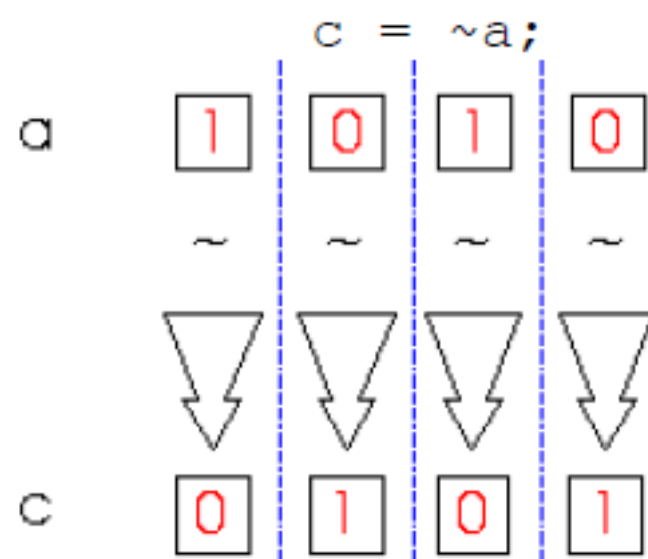
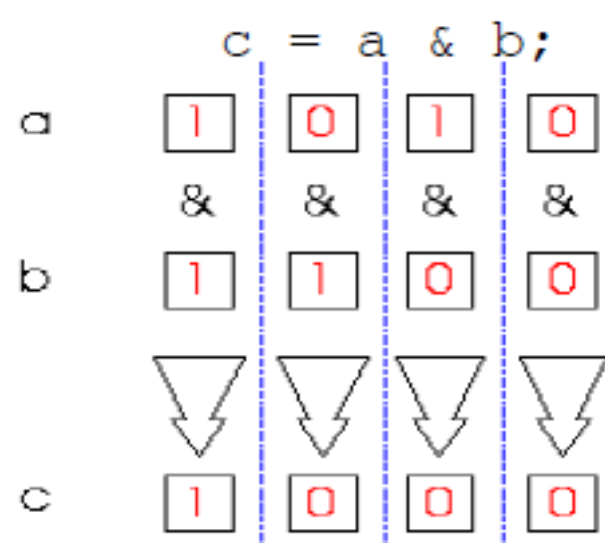
<input type="checkbox"/> ==	→ logical equality	} Return 0, 1 or x
<input type="checkbox"/> !=	→ logical inequality	
<input type="checkbox"/> ===	→ case equality	} Return 0 or 1
<input type="checkbox"/> !==	→ case inequality	

```
// A = 4, B = 3
// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx

A == B // Results in logical 0
X != Y // Results in logical 1
X == Z // Results in x
Z === M // Results in logical 1 (all bits match, including x and z)
Z === N // Results in logical 0 (least significant bit does not match)
M !== N // Results in logical 1
```


BITWISE OPERATORS

- $\&$ → bitwise AND
- $|$ → bitwise OR
- \sim → bitwise NOT
- \wedge → bitwise XOR
- $\sim\wedge$ or $\wedge\sim$ → bitwise XNOR
- Operation on bit by bit basis



REDUCTION OPERATORS

□ $\&$ \rightarrow AND

□ $|$ \rightarrow OR

□ \wedge \rightarrow XOR

□ $\sim\&$ \rightarrow NAND

□ $\sim|$ \rightarrow NOR

□ $\sim\wedge$ or $\wedge\sim$ \rightarrow XNOR

□ One multi-bit operand \rightarrow One single-bit result

`a = 4'b1001;`

`c = la; // c = 1101011 = 1`

SHIFT OPERATORS

❑ `>>` → shift right

❑ `<<` → shift left

❑ Result is same size as first operand, **always zero filled**

```
a = 4'b1010; d = a >> 2;    // d = 0010
```

```
c = a << 1;    // c = 0100
```

❑ `>>>` → arithmetic shift right

❑ `<<<` → arithmetic shift left

❑ Vacant position filled by either LSB (`>>>`) or MSB (`<<<`) value

```
a = 4'b0101;  d = a >>> 1;    // d = 1010
```

```
c = a <<< 1;    // c = 1010
```

CONCATENATION OPERATORS

❑ {op1, op2, ..} → concatenates op1, op2, .. to single number

❑ Operands must be sized !!

```
reg a;
```

```
reg [2:0] b, c;
```

```
..
```

```
a = 1'b 1;
```

```
b = 3'b 010;
```

```
c = 3'b 101;
```

```
catx = {a, b, c};
```

```
// catx = 1_010_101
```

```
caty = {b, 2'b11, a};
```

```
// caty = 010_11_1
```

```
catz = {b, 1};
```

```
// WRONG !!
```

❑ Replication ..

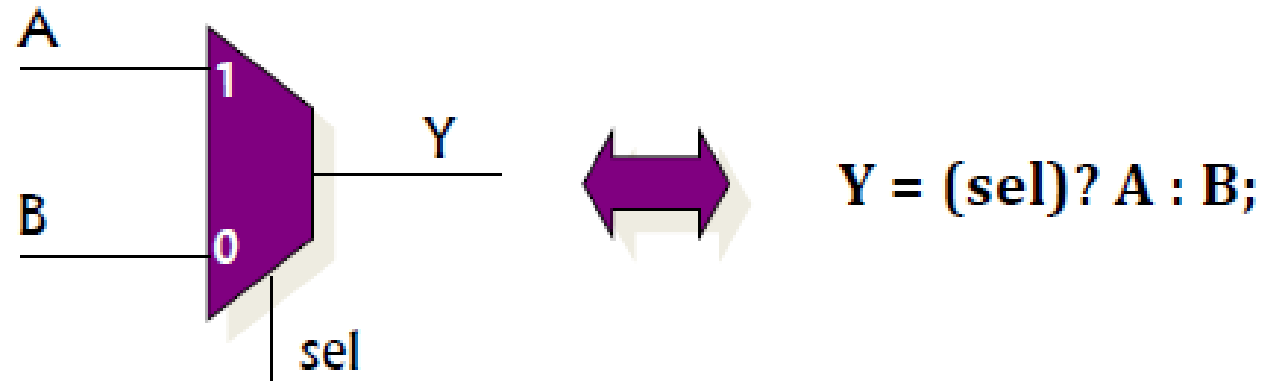
```
catr = {4{a}, b, 2{c}};
```

```
// catr = 1111_010_101101
```

CONDITIONAL OPERATORS

❑ `cond_expr ? true_expr : false_expr`


❑ Like a 2-to-1 mux ..



❑ For 4-to-1 mux ..

```
assign out = sel[1] ? (sel[0] ? in3 : in2) : (sel[0] ? in1 : in0);
```

OPERATORS PRECEDENCE

<code>+ - ! ~ unary</code>	highest precedence
<code>* / %</code>	
<code>+ - (binary)</code>	
<code>< < > ></code>	
<code>< <= == > ></code>	
<code>== != === !==</code>	
<code>& ~ &</code>	
<code>^ ^~ ~^</code>	
<code> ~ </code>	
<code>& &</code>	
<code> </code>	
<code>?: conditional</code>	lowest precedence

Use parentheses to
enforce your priority

GATE PRIMITIVES

- ❑ All logic circuits can be designed by using basic gates. Verilog supports basic logic gates as predefined *primitives*. There are three classes of basic gates.
- ❑ Multiple-input gates: and, or, nand, nor, xor, xnor
- ❑ Multiple-output gates: buffer, not
- ❑ Tristate gates: bufif0, bufif1, notif0, notif1
- ❑ *These primitives are instantiated like modules* except that they are predefined in Verilog and do not need a module definition.
- ❑ Logic gates can be used in design using gate instantiation. A simple format of a gate instantiation is,

```
Gate_type [instance_name] (term1, term2,.....termn);
```

Basic syntax for multiple-input gates is:

Multiple_input_Gate_type [instance_name] (output, input1, input2,.....inputn);

AND GATE



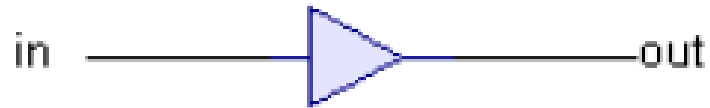
```
// Module Name:  Andgate  
module Andgate(i1, i2, out);  
    input i1;  
    input i2;  
    output out;  
    and (out,i1,i2);  
endmodule
```

OR GATE



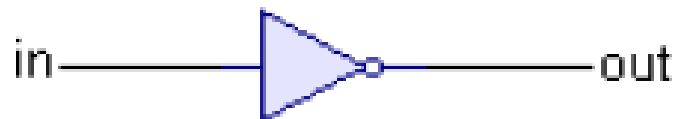
```
// Module Name:  Orgate  
module Orgate(i1, i2, out);  
    input i1;  
    input i2;  
    output out;  
    or(out,i1,i2);  
endmodule
```


BUFFER GATE



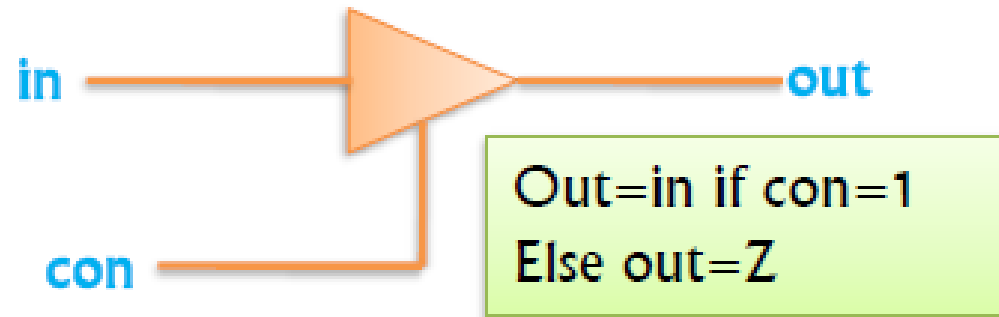
```
// Module Name:  Buffer
module Buffer(in, out);
    input in;
    output out;
    buf(out,in);
endmodule
```

NOT GATE



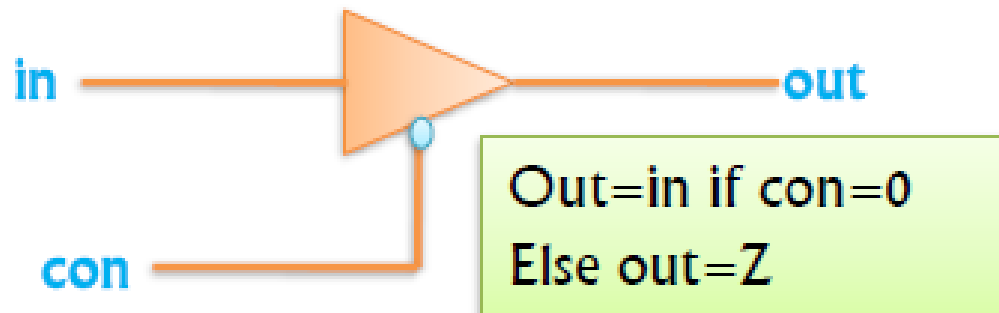
```
// Module Name:  Notgate
module Notgate(in, out);
    input in;
    output out;
    not(out,in);
endmodule
```

BUFIF1 GATE



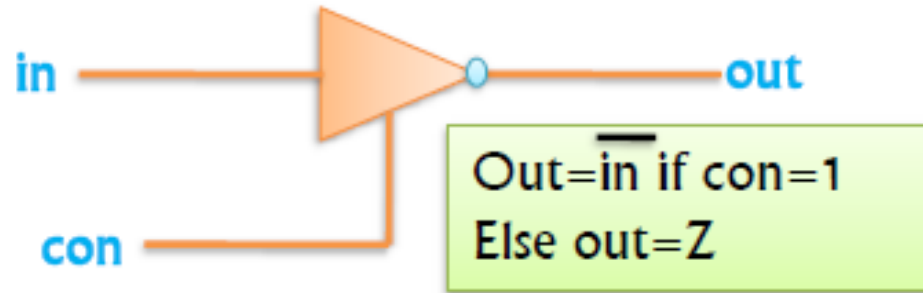
```
// Module Name:  Bufif1  
module Bufif1(in, out, con);  
    input in,con;  
    output out;  
    bufif1(out,in,con);  
endmodule
```

BUFIF0 GATE



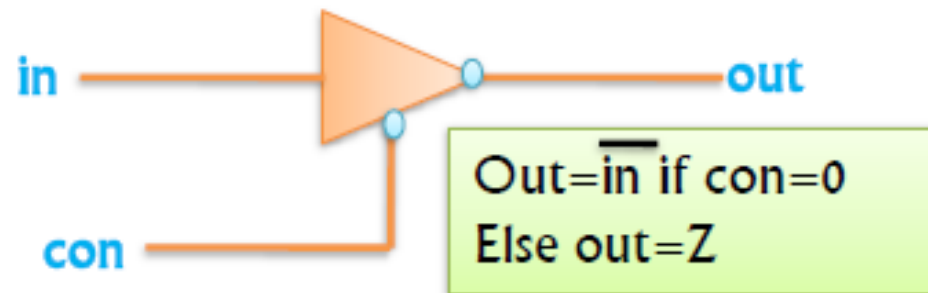
```
// Module Name:  Bufif0  
module Bufif0(in, out, con);  
    input in,con;  
    output out;  
    bufif0(out,in,con);  
endmodule
```

NOTIF1 GATE



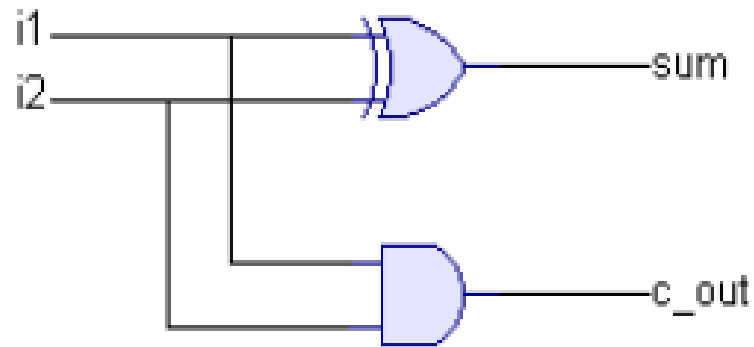
```
// Module Name:  notif1  
module notif1(in, out, con);  
    input in,con;  
    output out;  
    notif1(out,in,con);  
endmodule
```

NOTIF0 GATE



```
// Module Name:  notif0  
module notif0(in, out, con);  
    input in,con;  
    output out;  
    notif0(out,in,con);  
endmodule
```

HALF ADDER



```
// Module Name:   HalfAdder
module HalfAdder(sum, c_out, i1, i2);
    output sum;
    output c_out;
    input i1;
    input i2;
    xor(sum,i1,i2);
    and(c_out,i1,i2);
endmodule
```

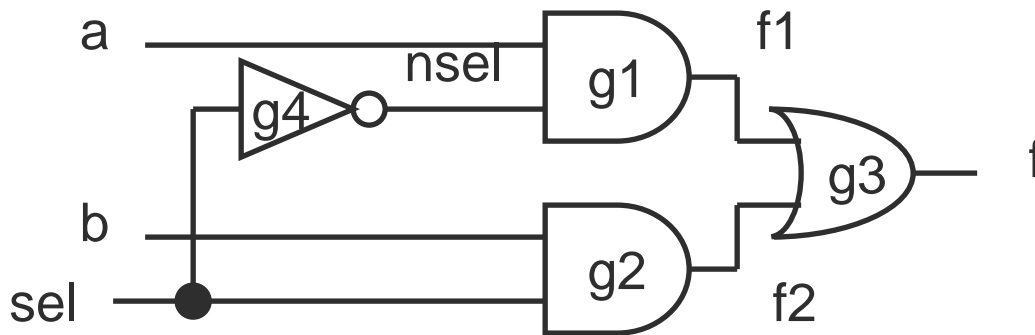
Multiplexer Built From Primitives

```
module mux(f, a, b, sel);  
  output f;  
  input a, b, sel;  
  
  and g1(f1, a, nsel),  
    g2(f2, b, sel);  
  or g3(f, f1, f2);  
  not g4(nsel, sel);  
  
endmodule
```

Verilog programs built from modules

Each module has an interface

Module may contain structure: instances of primitives and other modules



Identifiers in Verilog

- Any Sequence of letter, digits, dollar sign, underscore.
 - First character must be a letter or underscore.
 - It cannot be a dollar sign.
 - Cannot use characters such as hyphen, brackets, or # in verilog names
-

Verilog Logic Values

- Predefined logic value system or value set
: '0', '1', 'x' and 'z';
 - 'x' means uninitialized or unknown logic value
 - 'z' means high impedance value.
-

Verilog Data Types

- Nets: wire, supply1, supply0
 - wire:
 - i) Analogous to a wire in an ASIC.
 - ii) Cannot store or hold a value.
 - Integer: used for the index variables of say for loops. No hardware implication.
-

The reg Data Type

- Register Data Type: Comparable to a variable in a programming language.
 - Default initial value: 'x'
 - ```
module reg_ex1;
 reg Q; wire D;
 always @(posedge clk) Q=D;
```
  - A reg is not always equivalent to a hardware register, flipflop or latch.
  - ```
module reg_ex2; // purely combinational  
    reg c;  
    always @(a or b) c=a|b;  
endmodule
```
-

Wire: helps to connect

- Consider a set of tristate drivers connected to a common bus.
 - The output of the wire depends on *all* the outputs and not on the last one.
 - To model connectivity, any value driven by a device must be driven continuously onto that wire, in parallel with the other driving values.
-

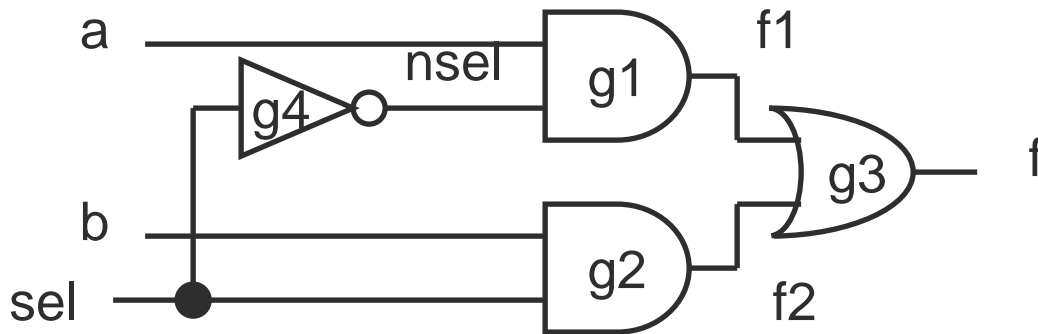
Multiplexer Built From Primitives

```
module mux(f, a, b, sel);  
  output f;  
  input a, b, sel;
```

Identifiers not
explicitly defined
default to wires

```
    and    g1(f1, a, nsel),  
          g2(f2, b, sel);  
    or     g3(f, f1, f2);  
    notg4(nsel, sel);
```

```
endmodule
```



Syntax- Always and Initial block

- In Verilog, the **always block** is one of the procedural blocks.
 - Statements inside an always block are executed sequentially.
 - An always block always executes, unlike initial blocks that execute only once at the beginning of the simulation. The always block should have a sensitive list or a delay associated with it
 - The sensitive list is the one that tells the always block when to execute the block of code.
 - The ***always*** block indicates a free-running process, but the ***initial*** block indicates a process executes exactly once. Both constructs begin execution at simulator time 0, and both execute until the end of the block
-

Syntax- Always and Initial block

- Initial blocks can be used in either synthesizable or non-synthesizable blocks. They are commonly used in test benches.
 - Initial blocks cause particular instructions to be performed at the beginning of the simulation before any other instructions operate. Initial blocks only operate once.
-

```
1 // Template #1: Use for combinational logic, all inputs mentioned in
2 // sensitivity list ensures that it infers a combo block
3 always @ (all_inputs) begin
4     // Combinational logic
5 end
6
7 // Template #2: Use of a if condition without else can cause a latch
8 // because the previous value has to be held since new value is not
9 // defined by a missing else clause
10 always @ (all_inputs) begin
11     if (enable) begin
12         // latch value assignments
13     end
14 end
15
16 // Template #3: Use clock in sensitivity list for sequential elements
17 always @ (posedge clk) begin
18     // behavior to do at posedge clock
19 end
20
21 // Template #4: Use clock and async reset in sensitivity list
22 always @ (posedge clk or negedge resetn) begin
23     if (! resetn) begin
24         // behavior to do during reset
25     end else begin
26         // behavior when not in reset
27     end
28 end
```

■ Procedural assignments

- must be placed inside initial or always blocks.
- update values of variable data types (reg, integer, real, or time.)

variable_lvalue = [timing_control] expression

[timing_control] variable_lvalue = expression

- variable_lvalue can be:
 - a reg
 - integer,
 - real,
 - time, or
 - a memory element,
 - a bit select, a part select, a concatenation of any of the above.

Procedural Assignments

- The bit widths of both left-hand and right-hand sides need not be the same.
 - The right-hand side is truncated if it has more bits.
 - by keeping the least significant bits
 - The right-hand side is filled with zeros in the most significant bits when it has fewer bits.
 - Two types of procedural assignments:
 - **blocking**: using the operator “=”
 - **nonblocking**: using the operator “<=“
-

Blocking Assignments

- Blocking assignments
 - are executed in the order they are specified.
 - use the “=” operator.

```
// an example illustrating blocking assignments
module blocking;
reg x, y, z;
// blocking assignments
initial begin
    x = #5 1'b0; // x will be assigned 0 at time 5
    y = #3 1'b1; // y will be assigned 1 at time 8
    z = #6 1'b0; // z will be assigned 0 at time 14
end
endmodule
```

Nonblocking Assignments

■ Nonblocking assignments

- are executed without blocking the other statements.
- use the `<=` operator.
- are used to model several concurrent data transfers.

```
// an example illustrating nonblocking assignments
module nonblocking;
reg x, y, z;
// nonblocking assignments
initial begin
    x <= #5 1'b0; // x will be assigned 0 at time 5
    y <= #3 1'b1; // y will be assigned 1 at time 3
    z <= #6 1'b0; // z will be assigned 0 at time 6
end
endmodule
```

■ Coding style: In the always block

- Use nonblocking operators (\leq) when it is a piece of sequential logic;
 - Otherwise, the result of RTL behavioral may be inconsistent with that of gate-level.
 - Use blocking operators ($=$) when it is a piece of combinational logic.
-

Blocking and Nonblocking Assignments

- Blocking procedural assignments must be executed before the procedural flow can pass to the subsequent statement.
 - A Non-blocking procedural assignment is scheduled to occur without blocking the procedural flow to subsequent statements.
-

Nonblocking Statements are odd!

```
a = 1;  
b = a;  
c = b;
```

Blocking assignment:

```
a = b = c = 1
```

```
a <= 1;  
b <= a;  
c <= b;
```

Nonblocking assignment:

```
a = 1  
b = old value of a  
c = old value of b
```

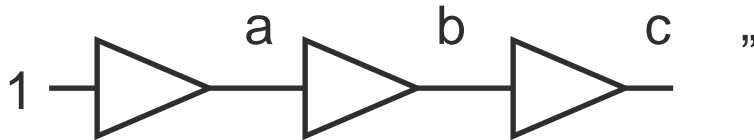
Nonblocking Looks Like Latches

- RHS of nonblocking taken from latches
- RHS of blocking taken from wires

`a = 1;`

`b = a;`

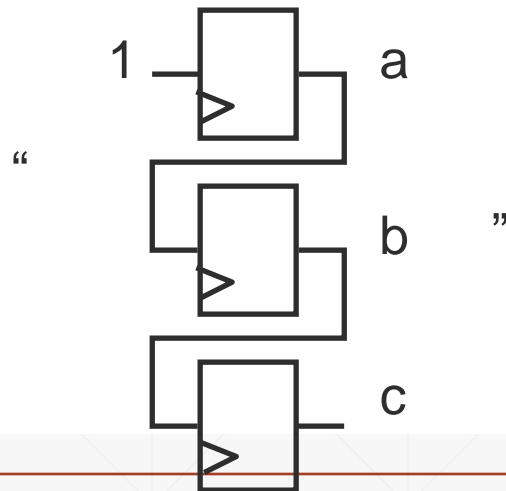
`c = b;`



`a <= 1;`

`b <= a;`

`c <= b;`



Procedural Continuous Assignment

- **Assign and deassign**

```
reg q;
```

```
initial begin
```

```
    assign q = 0;
```

```
    #10 deassign q;
```

```
end
```

- **Force release:**

```
reg o, a, b;
```

```
initial begin
```

```
    force o = a & b;
```

```
    .....
```

```
    release o;
```

```
end
```

Always and Initial block syntax

- Always block syntax:

always @ (event)

[statement]

ex:1

always @ (event) **begin**

[multiple statements]

end

ex:2

always @ (posedge clk) **begin**

[statements]

end

- Initial block syntax:

initial

[single statement]

initial begin

[multiple statements]

end

Initial Blocks in a module

This is an **initial** block
which starts at time 0ns

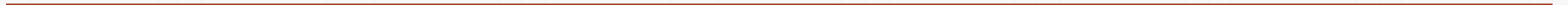
Here, **a** will get value **2'b10** at
time 0ns

```
module behave;  
  reg [1:0] a, b;
```

initial

a = 2'b10;

endmodule



Initial Blocks in a module

This will advance time by 10ns.

```
module behave;  
    reg [1:0] a, b;  
  
    initial begin  
        a = 2'b10;  
        #10 b = 2'b00;  
    end  
endmodule
```

- **a** will get value **2'b10** at **0ns**,
- Time will advance to **10ns**,
- Then **b** will be assigned **2'b00**

a = 2'b10; ←
#10 b = 2'b00;

Initial Blocks in a module

There can be multiple
initial blocks

```
module behave;
```

```
reg [1:0] a, b;
```

```
initial begin
```

```
  a = 2'b10;
```

```
  #20 b = 2'b11;
```

```
end
```

```
initial begin
```

```
  #10 a = 2'b11;
```

```
  #40 b = 2'b10;
```

```
end
```

```
initial
```

```
  #60 $finish;
```

```
endmodule
```

#<delay> advances time by
<delay>units

Each **initial** block starts
at time 0ns as three
separate threads.

b is assigned 40ns after
a is assigned.

- There are no limits to the number of initial blocks that can be defined inside a module. The code shown below has three initial blocks, all of which are started at the same time and run in parallel.
- However, depending on the statements and the delays within each initial block, the time taken to finish the block may vary.

Example of using always block in combinational circuit

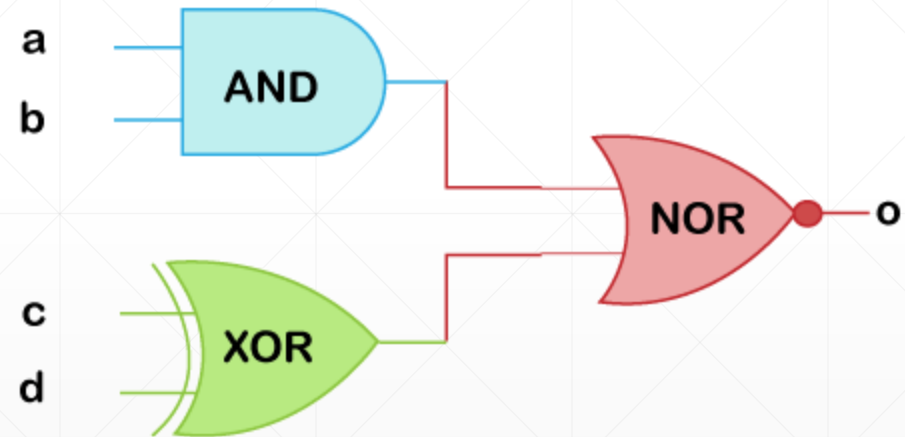
```
module combo (input a, input b,  
input c, input d, output reg o);
```

```
  always @ (a or b or c or d) begin
```

```
    o <= ~((a & b) | (c^d));
```

```
  end
```

```
endmodule
```



Selection Constructs

- Selection structures
 - make a selection according to the given condition.
- Two types
 - if...else statement
 - case statement
- if...else statement syntax
 - if statement only
 - if and one else statement
 - nested if-else-if statement
- The else part is always associated to the closest previous if that lacks an else.

if-else and case statements syntax

- The conditional statement (or if-else statement) is used to make a decision as to whether a statement is executed or not.

- Template 1:

- if (<expression>) <statement_or_null>

- Template 2:

if (<expression>) <statement_or_null>

else <statement_or_null>

<statement_or_null>

- Template 3:

if (<expression>)

<statement>

else if (<expression>)

<statement>

else if (<expression>)

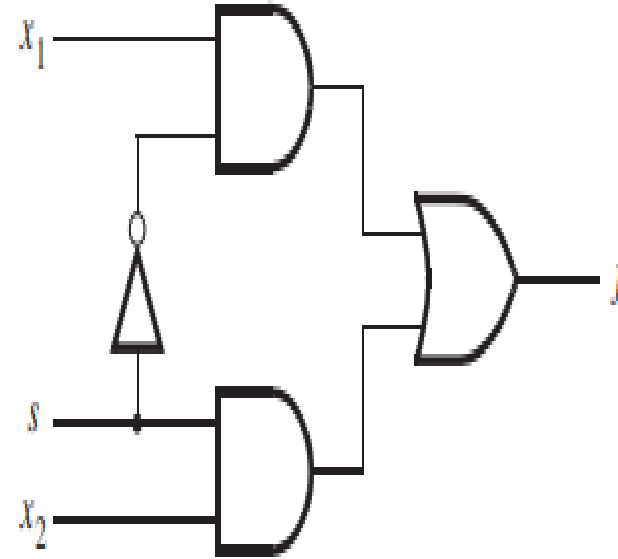
<statement>

else

<statement>

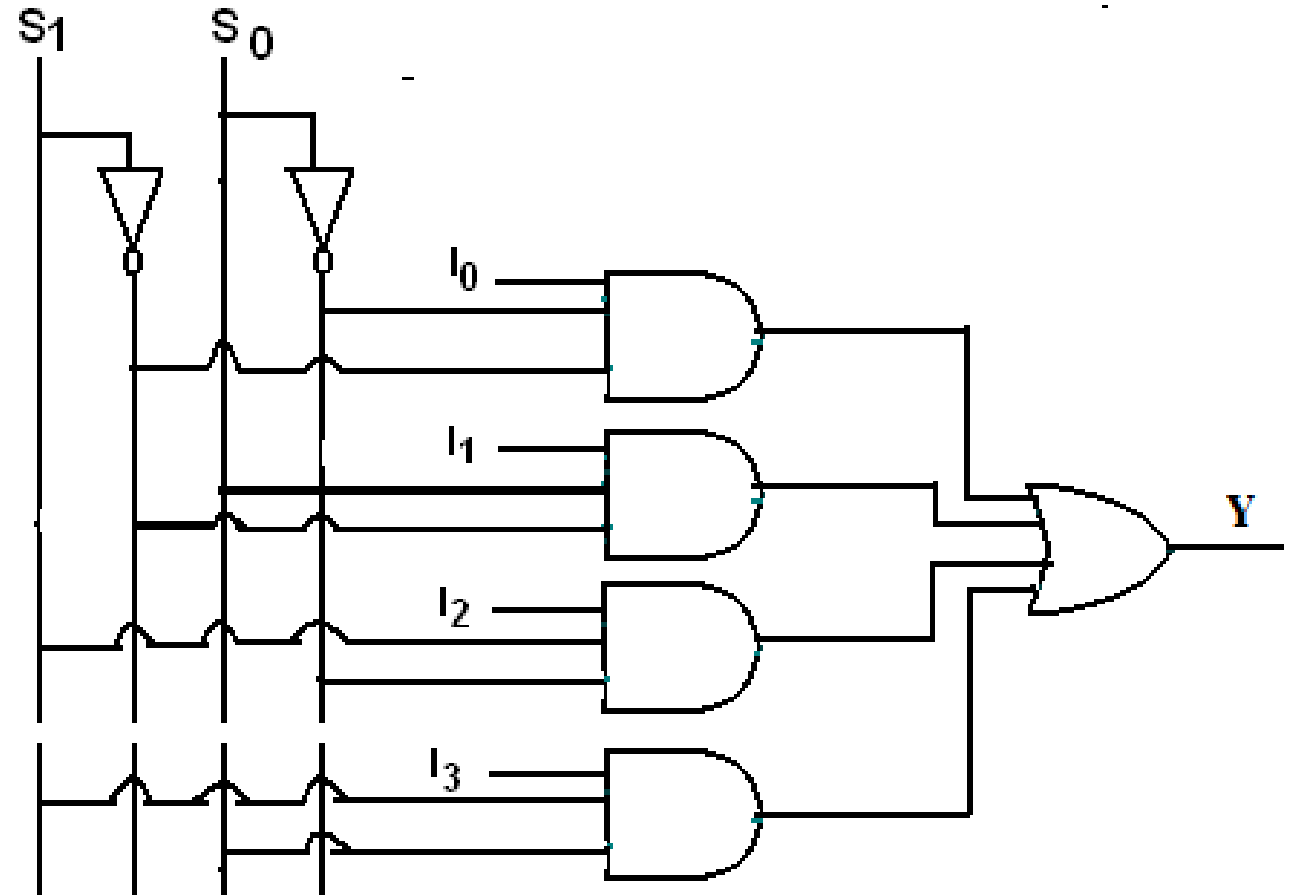
2:1 MUX- behavior modeling

```
module mux_21 (  
    input x1,x2,s,  
    output f );  
//behavior modeling statements  
always @ (x1 or x2 or s)  
begin  
    if(s==0)  
        f=x1;  
    else  
        f=x2;  
    end  
endmodule
```



Input	S1	S0	Y
I ₀	0	0	I ₀
I ₁	0	1	I ₁
I ₂	1	0	I ₂
I ₃	1	1	I ₃

$$Y = S_1 S_0 I_3 + S_1 \bar{S}_0 I_2 + \bar{S}_1 S_0 I_1 + \bar{S}_1 \bar{S}_0 I_0$$



4 to 1 Multiplexer and its truth table

Data-flow modeling

```
module m41 ( input a,  
input b,  
input c,  
input d,  
input s0, s1,  
output out);
```

```
    assign out = s1 ? (s0 ? d : c) : (s0 ? b : a);
```

```
endmodule
```

if s1 is high, the (s0 ? d : c) block will be executed, else (s0 ? b : a) will be executed. Further, if s0 is high, d OR b will get transferred to the out variable, depending on the s1 select line, else c OR a will be the output.



Case statement

- Syntax:

```
case (case_expression)
  case_item1: procedural_expression;
  case_item2: begin
    procedural_statements;
  end
  ....
  default: expression;
endcase
```

```
module m41 ( a, b, c, d, s0, s1, out);
```

```
input wire a, b, c, d;
```

```
input wire s0, s1;
```

```
output reg out;
```

```
always @ (a or b or c or d or s0, s1)
```

```
begin
```

```
  case (s0 | s1)
```

```
    2'b00 : out <= a;
```

```
    2'b01 : out <= b;
```

```
    2'b10 : out <= c;
```

```
    2'b11 : out <= d; endcase
```

```
  end
```

```
endmodule
```

If –else statement

```
module m41 ( din ,sel ,dout  
);
```

```
output dout ;  
reg dout ;
```

```
input [3:0] din ;  
wire [3:0] din ;  
input [1:0] sel ;  
wire [1:0] sel ;
```

```
always @ (din or sel) begin  
    if (sel==0)  
        dout = din[3];  
    else if (sel==1)  
        dout = din[2];  
    else if (sel==2)  
        dout = din[1];  
    else  
        dout = din[0];  
end  
  
endmodule
```

Testbench

```
module top;
wire out;
reg a;
reg b;
reg c;
reg d;
reg s0, s1;
m41 name(.out(out), .a(a), .b(b), .c(c), .d(d), .s0(s0), .s1(s1));
initial
begin
a=1'b0; b=1'b0; c=1'b0; d=1'b0;
s0=1'b0; s1=1'b0;
#500 $finish;
end
always #40 a=~a;
always #20 b=~b;
always #10 c=~c;
always #5 d=~d;
always #80 s0=~s0;
always #160 s1=~s1;
always@(a or b or c or d or s0 or s1)
$monitor("At time = %t, Output = %d", $time, out);
endmodule;
```

Loop Constructs

- Loop constructs control the execution of a statement zero, one, or more times.
 - Loop constructs
 - can appear only inside an initial or always block.
 - may contain delay expressions.
 - Four types
 - **while loop** executes a statement until an expression becomes false.
 - **for loop** repeatedly executes a statement.
 - **repeat loop** executes a statement a fixed number of times.
 - **forever loop** continuously executes a statement.
-

while loop

■ A while loop

- executes until the condition is false.
- shall not be executed at all if the condition_expr starts out false.

```
while (condition_expr) statement;
```

```
while (count < 12) count <= count + 1;  
while (count <= 100 && flag) begin  
    // put statements wanted to be carried out here.  
end
```

// an example illustrating how to count the zeros in a byte.

```
module zero_count_while (data, out);  
    input  [7:0] data;  
    output reg [3:0] out;    //output declared as register  
    integer i;  
    always @(data) begin  
        out = 0; i = 0;  
        while (i <= 7) begin    // simple condition  
            if (data[i] == 0) out = out + 1; // may be replaced with out = out + ~data[i].  
            i = i + 1; end  
        end  
    endmodule
```

for Loop

- A for loop is used to perform a counting loop. It
 - behaves like the `for` statement in C programming language.

```
for (init_expr; condition_expr; update_expr) statement;
```

- is equivalent to

```
init_expr;  
while (condition_expr) begin  
    statement;  
    update_expr;  
end
```

for Loop

```
// an example illustrating how to count the zeros in a byte.  
module zero_count_for (data, out);  
input  [7:0] data;  
output reg [3:0] out; // output declared as register  
integer i;  
always @(data) begin  
    out = 0;  
    for (i = 0; i <= 7; i = i + 1) // simple condition  
        if (data[i] == 0)  
            out = out + 1; // may be replaced with out = out + ~data[i].  
end  
endmodule
```


repeat Loop

- A repeat loop performs a loop a fixed number of times.

```
repeat (counter_expr) statement;
```

- counter_expr can be a constant, a variable or a signal value.
 - counter_expr is evaluated only once before starting the execution of statement (loop).
-

repeat Loop

— Examples:

```
i = 0;
repeat (32) begin
    state[i] = 0;    // initialize to zeros
    i = i + 1;      // next item
end
repeat (cycles) begin                // cycles must be evaluated to a number
    @(posedge clock) buffer[i] <= data; // before entering the loop.
    i <= i + 1;    // next item
end
```

forever Loop

- A forever loop continuously performs a loop until the `$finish` task is encountered. It
 - is equivalent to a while loop with an always true expression such as `while (1)`.

```
forever statement;
```
 - can be exited by the use of `disable` statement.
-

forever Loop

- The forever statement example

```
initial begin
    clock <= 0;
    forever begin
        #10 clock <= 1;
        #5  clock <= 0;
    end
end
```

- The forever statement is usually used with timing control statements.

```
reg clock, x, y;

initial
    forever @(posedge clock) x <= y;
```

Timing Controls

- Timing controls specify the simulation time at which procedural statements will be executed.
 - In Verilog HDL, if there are no timing control statements, the simulation time will not advance.
 - Timing Controls
 - Delay timing control
 - Regular delay control
 - Intra assignment delay control
 - Event timing control
 - Edge-triggered event control
 - Named event control
 - Event or control
 - Level-sensitive event control
-

Regular Delay Control

- Regular delay control
 - A non-zero delay is specified to the left of a procedural assignment.
 - It defers the execution of the entire statement.

```
reg x, y;  
integer count;  
// The "<=" operators in the following statements can be replaced with "="  
// without affecting the results.  
#25      y <= ~x;           // execute at time 25  
#15 count <= count + 1;    // execute at time 40
```

Intra-Assignment Delay Control

- Intra-assignment delay control
 - A non-zero delay is specified to the right of the assignment operator.
 - It defers the assignment to the left-hand-side variable.

```
y = #25 ~x;           // evaluate at time 0 but assign to y at time 25  
count = #15 count + 1; // evaluate at time 0 but assign to count at time 40
```

```
y <= #25 ~x;           // evaluate at time 0 but assign to y at time 25  
count <= #15 count + 1; // evaluate at time 0 but assign to count at time 15
```

Event Timing Control

- Event Timing Control
 - An event is the change in the value on a variable or a net.
 - The execution of a procedural statement can be synchronized with an event.
 - Two types of event control
 - Edge-triggered event control
 - Named event control
 - Event **or** control
 - Level-sensitive event control
-

Edge-Triggered Event Control

■ Edge-triggered event control

- The symbol `@` is used to specify such event control.
 - `@(posedge clock)`: at the positive edge
 - `@(negedge clock)`: at the negative edge

```
always @(posedge clock) begin
    reg1 <= #25 in_1;                // intra-assignment delay control
    reg2 <= @(negedge clock) in_2 ^ in_3; // edge-triggered event control
    reg3 <= in_1;                    // no delay control
end
```

Named Event Control

- A named event
 - is declared with the keyword `event`.
 - does not hold any data.
 - is triggered by the symbol `->`.
 - is recognized by the symbol `@`.

```
event received_data; // declare an event received_data
// trigger event received_data
always @(posedge clock) if (last_byte) -> received_data;
always @(received_data) begin ..... end // execute event-dependent operations
```

Event or Control

■ Event **or** control

- uses the keyword **or** to specify multiple triggers.
- can be replaced by the “,”.
- can use $@^*$ or $@(*)$ to mean a change on **any** signal.

```
always @(posedge clock or negative reset_n) // event or control
begin
    if (!reset_n) q <= 1'b0; // asynchronous reset.
    else          q <= d;
end
```

Level-Sensitive Event Control

- Level-sensitive event control
 - uses the keyword `wait`.

```
always  
  wait (count_enable) count = count - 1 ;
```

```
always  
  wait (count_enable) #10 count = count - 1 ;
```

Multiplexer Built With Always

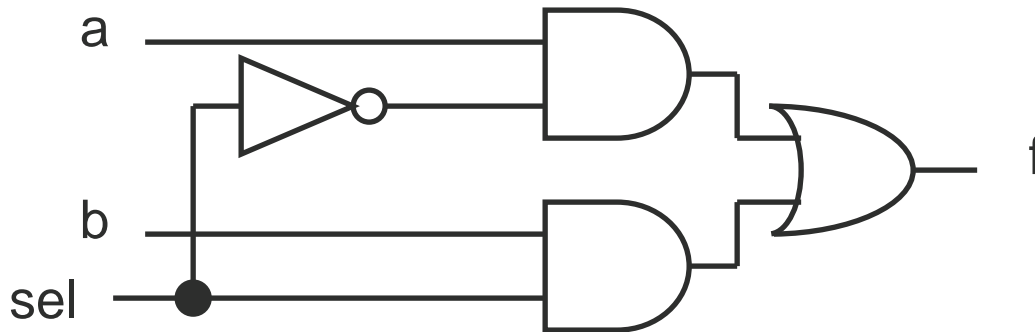
```
module mux(f, a, b, sel);  
output f;  
input a, b, sel;  
reg f;
```

```
  always @(a or b or sel)  
    if (sel) f = b;  
    else f = a;
```

```
endmodule
```

Modules may contain one or more *always* blocks

Sensitivity list contains signals whose change triggers the execution of the block



Multiplexer Built With Always

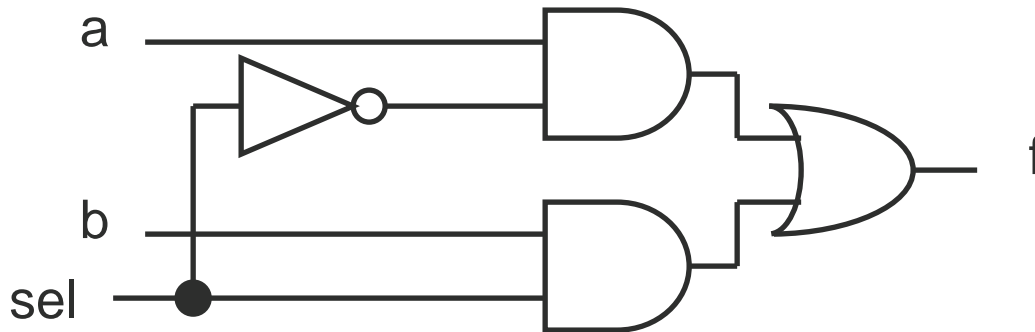
```
module mux(f, a, b, sel);  
output f;  
input a, b, sel;  
reg f;
```

A *reg* behaves like memory:
holds its value until
imperatively assigned
otherwise

```
always @(a or b or sel)  
if (sel) f = b;  
else f = a;
```

Body of an *always*
block contains
traditional imperative
code

```
endmodule
```

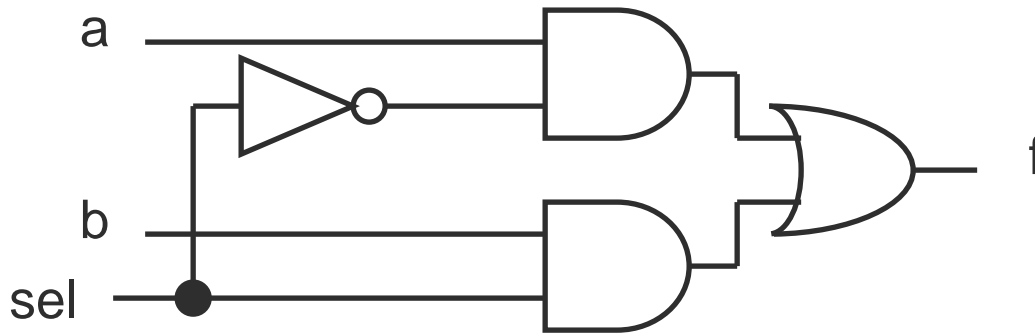


Mux with Continuous Assignment

```
module mux(f, a, b, sel);  
  output f;  
  input a, b, sel;  
  
  assign f = sel ? b : a;  
  
endmodule
```

LHS is always set to the
value on the RHS

Any change on the right
causes re-evaluation



System Tasks

Displaying information:

\$display(p1, p2, p3 ,....., pn);

- main system task for displaying values of variables or strings or expressions.

Monitoring information:

\$monitor(p1 ,p2,p3 ,....., pn);

- provides a mechanism to monitor a signal when its value changes.
-

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name (no argument required)
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format (e.g., 3e10)
%f or %F	Display real number in decimal format (e.g., 2.13)
%g or %G	Display real number in scientific or decimal, whichever is shorter

System Tasks

Stopping and finishing in a simulation

- The task **\$stop** is provided to stop during a simulation.
 - *Usage:* **\$stop;**
 - The **\$finish** task terminates the simulation.
 - *Usage:* **\$finish;**
-

```
// Stop at time 100 in the simulation and examine the results
// Finish the simulation at time.
initial // to be explained later. time = 0
begin
clock = 0;
reset = 1;
#100 $stop; // This will suspend the simulation at time = 100
#900 $finish; // This will terminate the simulation at time = 1000
end
```

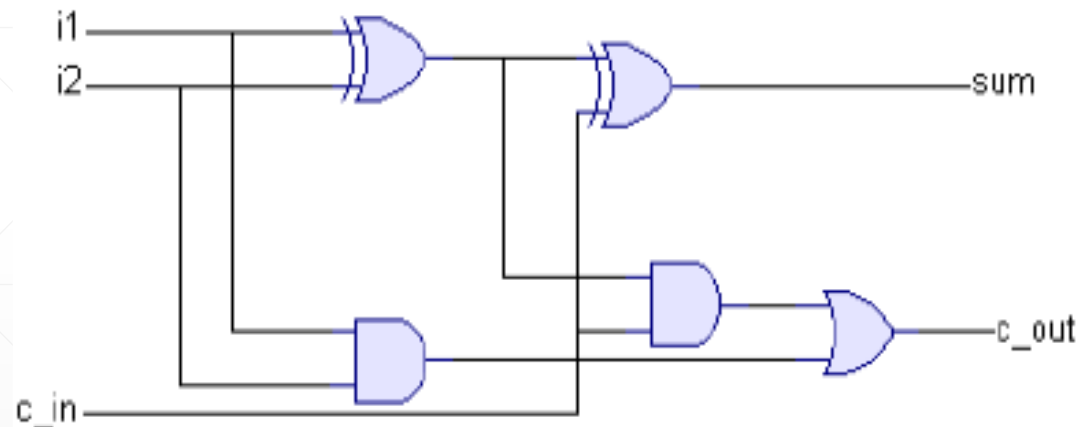
Time Scale for Simulations

❖ Time scale compiler directive

``timescale time_unit / time_precision`

- The `time_precision` must not exceed the `time_unit`.
 - For instance, with a timescale 1 ns/1 ps, the delay specification `#15` corresponds to 15 ns.
 - It uses the same time unit in both behavioral and gate-level modeling.
 - For FPGA designs, it is suggested to use `ns` as the time unit.
-

FULL ADDER USING MODULE INSTANTIATION



```
// Module Name: FullAdder
module FullAdder(sum, c_out, i1, i2, c_in);
    output sum;
    output c_out;
    input i1;
    input i2;
    input c_in;
    wire s1, c1, c2;
    HalfAdder HA1(s1, c1, i1, i2);
    HalfAdder HA2(sum, c2, s1, c_in);
    or(c_out, c1, c2);
endmodule
```

4-BIT PARALLEL ADDER USING MODULE INSTANTIATION

```
module adder_4bit (S, COUT,A,B);
```

```
    output [3:0]S ;
```

```
    output COUT ;
```

```
    input [3:0] A ;
```

```
    input [3:0] B ;
```

```
    wire [2:0]C;
```

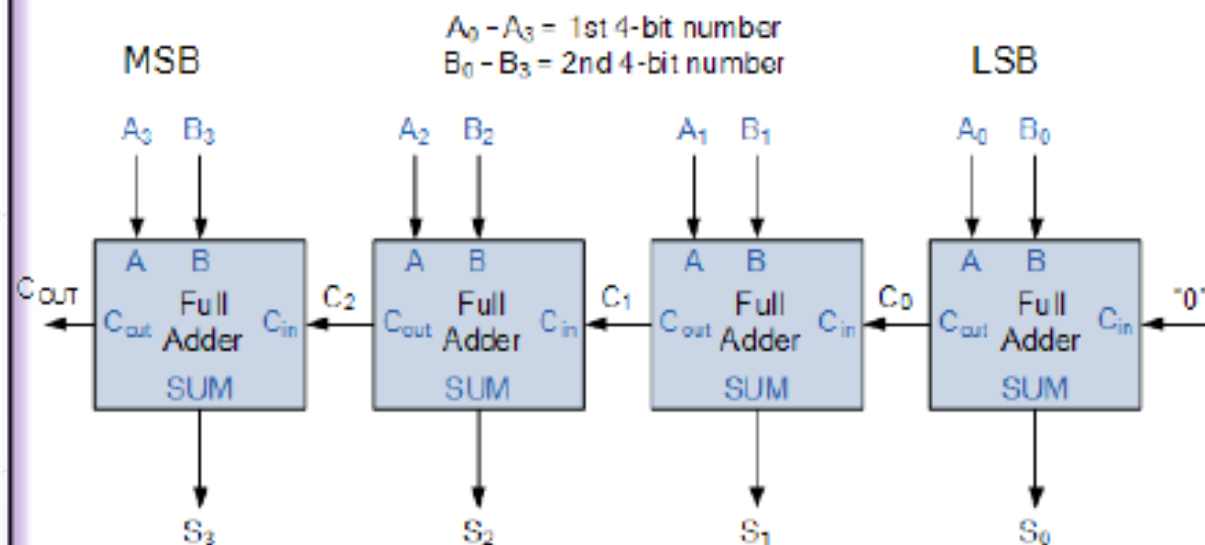
```
    FullAdder FA0(S[0],C[0],A[0],B[0],1'b0);
```

```
    FullAdder FA1(S[1],C[1],A[1],B[1],C[0]);
```

```
    FullAdder FA2(S[2],C[2],A[2],B[2],C[1]);
```

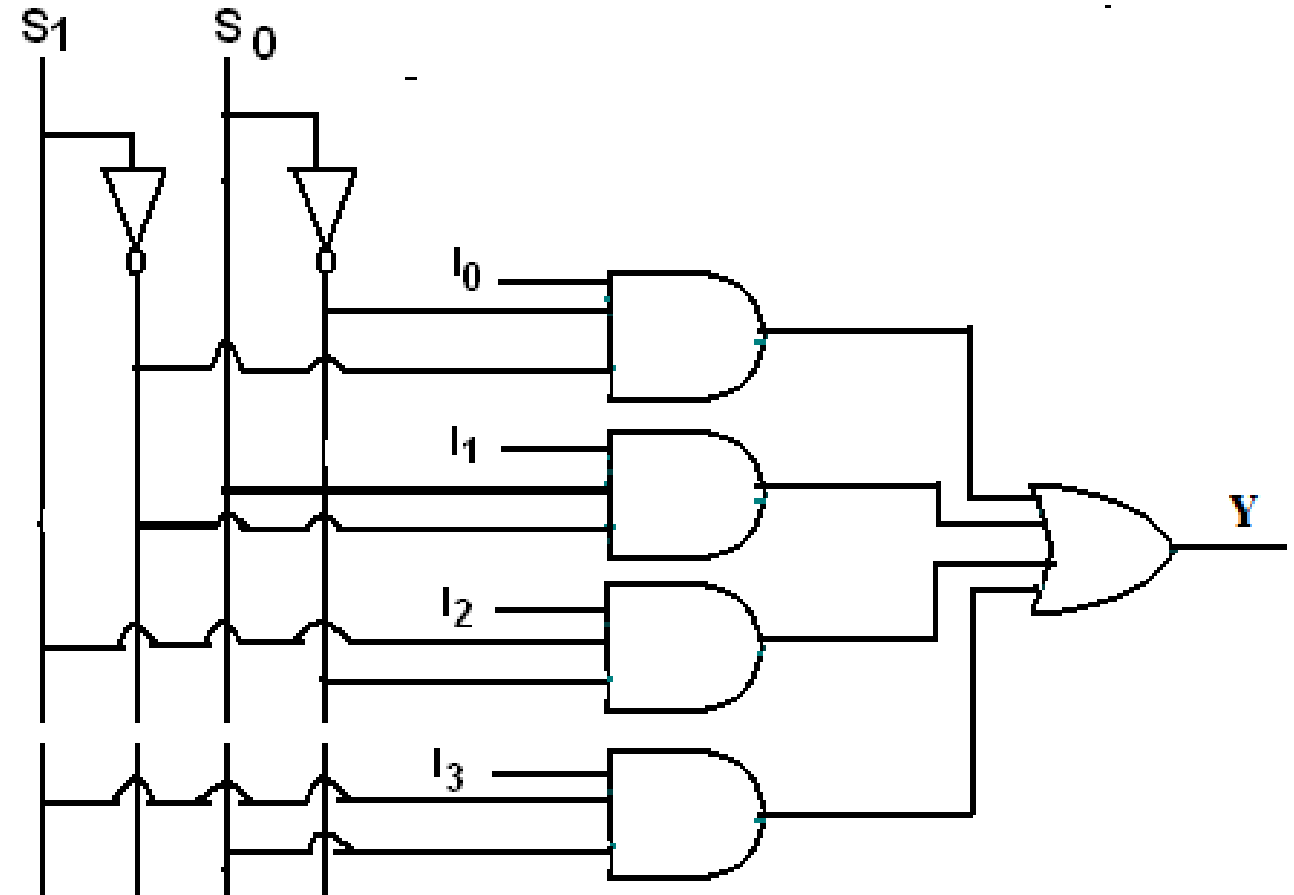
```
    FullAdder FA3(S[3],COUT,A[3],B[3],C[2]);
```

```
endmodule
```



Input	S1	S0	Y
I ₀	0	0	I ₀
I ₁	0	1	I ₁
I ₂	1	0	I ₂
I ₃	1	1	I ₃

$$Y = S_1 S_0 I_3 + S_1 \bar{S}_0 I_2 + \bar{S}_1 S_0 I_1 + \bar{S}_1 \bar{S}_0 I_0$$



4 to 1 Multiplexer and its truth table

Numbers

- Format of integer constants:

Width' radix value;

- Verilog keeps track of the sign if it is assigned to an integer or assigned to a parameter.
 - Once verilog loses sign the designer has to be careful.
-

Hierarchy

- Module interface provides the means to interconnect two verilog modules.
 - Note that a reg cannot be an input/ inout port.
 - A module may instantiate other modules.
-

Instantiating a Module

- Instances of

```
module mymod(y, a, b);
```

- Lets **instantiate** the module,

```
mymod mm1(y1, a1, b1);           // Connect-by-position
```

```
mymod mm2(.a(a2), .b(b2), .y(c2)); // Connect-by-name
```

Sequential Blocks

- Sequential block is a group of statements between a begin and an end.
 - A sequential block, in an always statement executes repeatedly.
 - Inside an initial statement, it operates only once.
-

Procedures

- A Procedure is an always or initial statement or a function.
 - Procedural statements within a sequential block executes concurrently with other procedures.
-

Parameterized Design

```
▪ module vector_and(z, a, b);  
    parameter cardinality = 5;  
    input [cardinality-1:0] a, b;  
    output [cardinality-1:0] z;  
    wire [cardinality-1:0] z = a & b;  
endmodule
```

We override these parameters when we instantiate the module as:

```
module Four_and_gates(  
    OutBus, InBusA, InBusB);  
    input [3:0] InBusA, InBusB;  
    output[3:0] OutBus;  
    Vector_And #(4) My_And(OutBus, InBusA,  
InBusB);  
endmodule
```

Functions

- Function Declaration and Invocation
 - Declaration syntax:

```
function <range_or_type> <func_name>;  
    <input_declaration(s)>  
    <variable_declaration(s)>  
    begin // if more than one statement needed  
        <statements>  
    end      // if begin used  
endfunction
```

Functions – Example

Controllable Shifter

```
module shifter;  
  `define LEFT_SHIFT    1'b0  
  `define RIGHT_SHIFT   1'b1  
  reg [31:0] addr, left_addr, right_addr;  
  reg control;  
  
  initial  
  begin  
      ...  
  
  end  
  always @(addr)begin  
    left_addr =shift(addr, `LEFT_SHIFT);  
    right_addr =shift(addr, `RIGHT_SHIFT);  
  end
```

```
function [31:0]shift;  
  input [31:0] address;  
  input control;  
  begin  
    shift = (control==`LEFT_SHIFT) ?(address<<1)  
      : (address>>1);  
  end  
endfunction  
  
endmodule
```

Function Examples Controllable Shifter

```
module shifter;
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr,
    right_addr;
reg control;
```

```
initial
begin
```

```
    ...
```

```
end
```

```
always @(addr)begin
    left_addr  =shift(addr,
        `LEFT_SHIFT);
    right_addr
        =shift(addr,`RIGHT_SHIFT);
```

```
end
```

```
function [31:0]shift;
input [31:0] address;
input control;
begin
    shift = (control==`LEFT_SHIFT)
        ?(address<<1) : (address>>1);
```

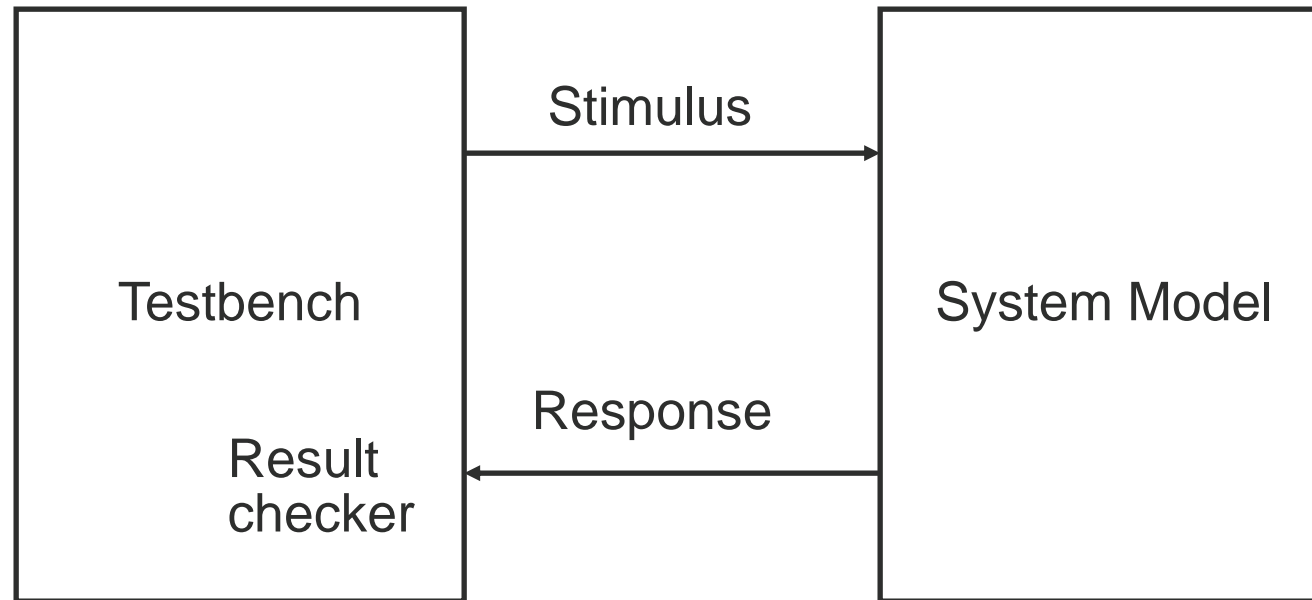
```
end
```

```
endfunction
```

```
endmodule
```


How Are Simulators Used?

- Testbench generates stimulus and checks response
- Coupled to model of the system
- Pair is run simultaneously



Conclusion : Write codes which can be translated into hardware !

The following cannot be translated into hardware(non - synthesizable):

- Initial blocks
 - Used to set up initial state or describe finite testbench stimuli
 - Don't have obvious hardware component
 - Delays
 - May be in the Verilog source, but are simply ignored
 - In short, write codes with a hardware in your mind. In other words do not depend too much upon the tool to decide upon the resultant hardware.
 - Finally, remember that you are a better designer than the tool.
-