超大型積體電路設計實驗

授課老師：范志鵬
助教：張耿嘉 / 方志軒

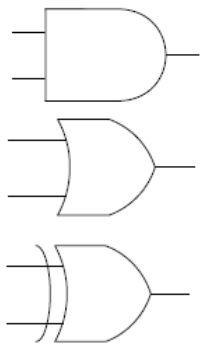
Outline

- Gate-Level Modeling
- Behavioral Modeling
  - Timing Control
  - Event-Based Timing Control
- Procedural Assignments
  - Blocking and Non blocking
  - Combinational and sequential circuit
- Conditional Statements
- Looping Statements

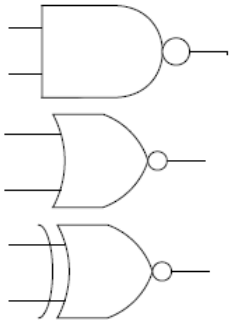
Gate-Level Modeling

- Primitive logic gate

- and
- or
- xor



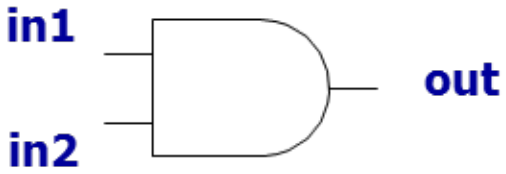
- nand
- nor
- xnor



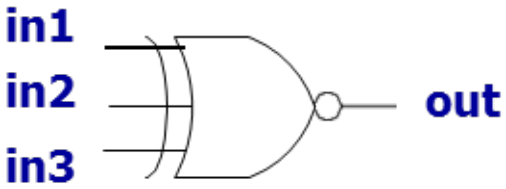
- The gates have one scalar output and multiple scalar inputs.
- The 1st terminal in the list of gate terminals is an output and the other terminals are inputs.

Gate-Level Modeling

- and(out, in1, in2) ;

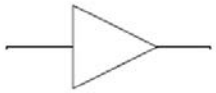


- xor(out, in1, in2, in3) ; ← can use with multiple inputs

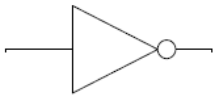


Gate-Level Modeling

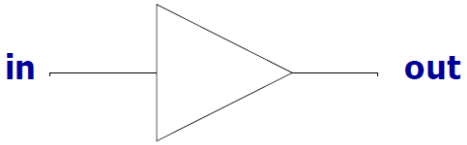
- buf/not gates
  - buf



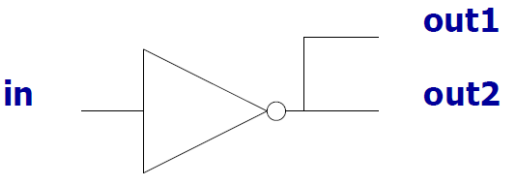
  - not



- buf(out, in) ;

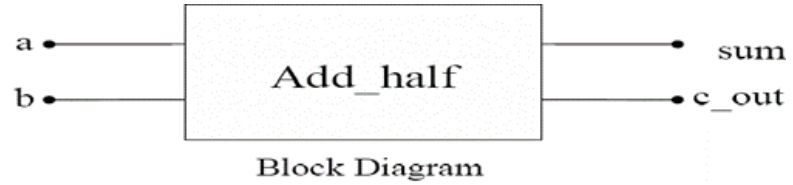


- not(out1, out2 ,in) ; ← can use with multiple outputs

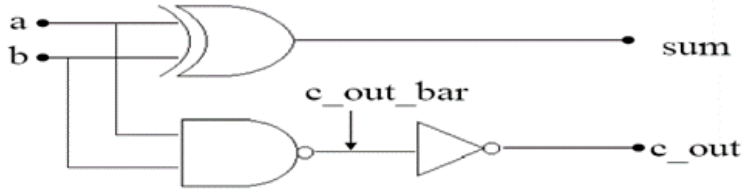


Gate-Level Modeling

- Ex: half adder



Block Diagram



Schematic

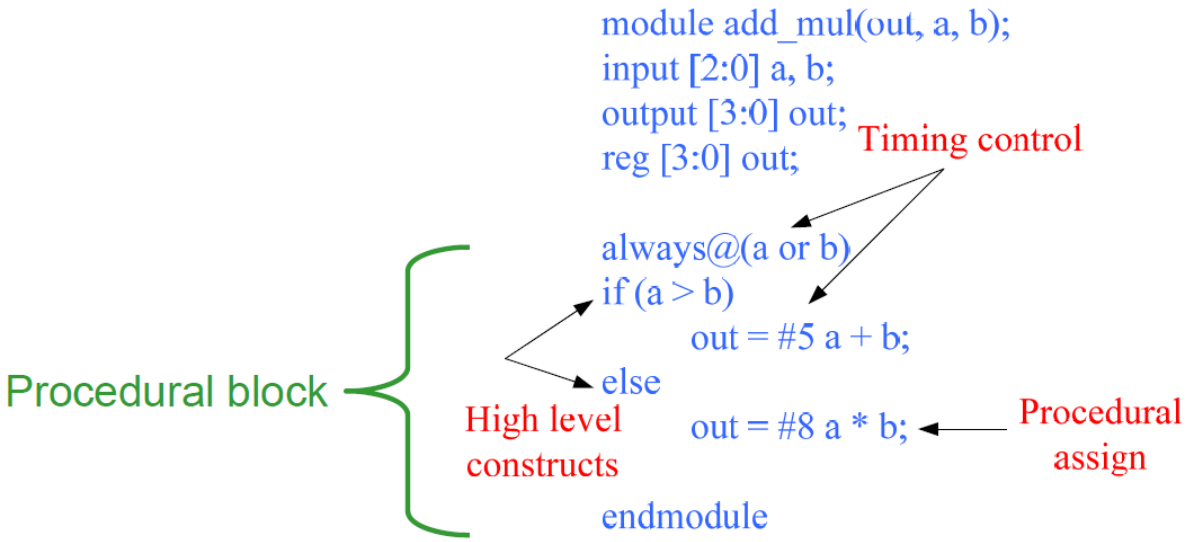
```
module ADDR_HALF(A, B, SUM, C_OUT);
input   A, B;
output  SUM, C_OUT;
wire    c_out_bar;
xor (SUM, A, B);
and (C_OUT, A, B);
endmodule
```

Behavioral Modeling

- ✓ Behavioral modeling enables you to describe the system at a high level of abstraction.
  - ➢ At this level of abstraction, implementation is not as important as the overall functionality of the system.
- ✓ High-level programming language constructs are available in Verilog for behavioral modeling.
  - ➢ These include ***wait***, ***while***, ***if else***, ***case***, and ***forever***.
- ✓ Behavioral modeling in Verilog is described by specifying a set of concurrently active procedural blocks that together describe the operation of the system.

Behavioral Modeling

- ✓ Procedural blocks have the following components:
  - ➢ Procedural assignment statements to describe the data flow within the block
  - ➢ High-level constructs (loops, conditional statements) to describe the functional operation of the block
  - ➢ Timing controls to control the execution of the block and the statements in the block



Procedural Timing Control

You can specify procedural timing inside of procedural blocks, using three types of timing controls:

1. Simple delays, or pound delays: ***#(delay)***
   - Delays execution for a specific number of time steps.
     - assign #3 a = ~b;

2. Level-sensitive timing control: **wait(<expr>)**
   - Delays execution until <expr> evaluates TRUE (non-zero). If <expr> is already TRUE, the statement executes immediately.
     - wait(a == b) c = a;

Procedural Timing Control

3. Edge-sensitive timing controls: **@(<signal>)**
   - Delays execution until an edge occurs on signal. You can specify the active edge of signal using ***posedge*** or ***negedge***. You can specify several signal arguments using the ***or*** keyword.
     - always@(posedge clk) a <= b;
     - always@(a or b) c = a + b;

Procedural Timing Control

- ✓ Use simple delays (#delays) to delay stimulus in a test bench, or to approximate real-world delays in behavioral models.

```
module muxtwo (out, a, b, sl);
input a,b,sl;
output out;
reg out;
always @(sl or a or b)
  if (!sl)
    #10 out = a; // The delay from a to out is 10 time units
  else
    #12 out = b; // The delay from b to out is 12 time units
endmodule
```

- ✓ You can use module parameters to parameterize simple delays.

```
module clock_gen (clk);  output clk;
reg clk;
parameter cycle = 20;  initial clk = 0;
always
   #(cycle /2) clk = ~clk;
endmodule
```

Procedural Timing Control

- ✓ Use the @ timing control for combinational and sequential models at the RTL and behavioral levels.
- ✓ You can qualify signal sensitivity with the ***negedge*** and ***posedge*** keywords, and you can wait for changes on multiple signals by using the **or** keyword.
  - ➢ The **or** event control modifier has nothing to do with the **bitwise-OR** operator "|" or the **logical-OR** operator "||".

In Verilog2001, can use 『,』 or 『or』

Pay Attention ! Sensitivity List !!
Triggers the action in the body
*In Verilog2001, can use 『 * 』*

Combinational block

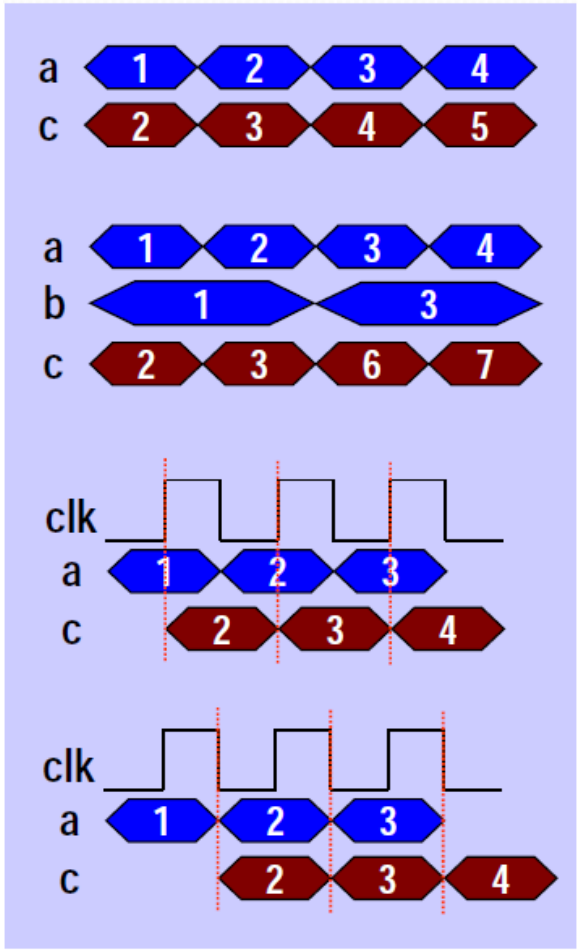
Sequential block

```
module reg_adder (out, a, b, clk);
input clk;
input [2:0]a,b;
output [3:0]out;
reg [3:0] out;
reg [3:0] sum;
always @(a or b) // When any change occurs on a or b
        #5 sum = a + b;
always @(negedge clk) // at every negative edge of clk
        out = sum;
endmodule
```

Event-Based Timing Control Example

- **Combinational circuit**
  - @(a): act if signal 'a' changes.
    Ex. always @(a) c <= a + 1;
  - @(a or b): act if signal 'a' or 'b' changes.
    Ex. always @(a or b) c <= a+ b;
  - The sensitivity list must include all inputs, you can use * mean all input.
    Ex. always @(*)

- **Register**
  - @(posedge clk): act at the rising edge of clk signal.
    - Ex. always @(posedge clk) c <= a + 1 ;
  - @(negedge clk): act at the falling edge of clk signal.
    - Ex. always @(negedge clk) c <= a + 1;



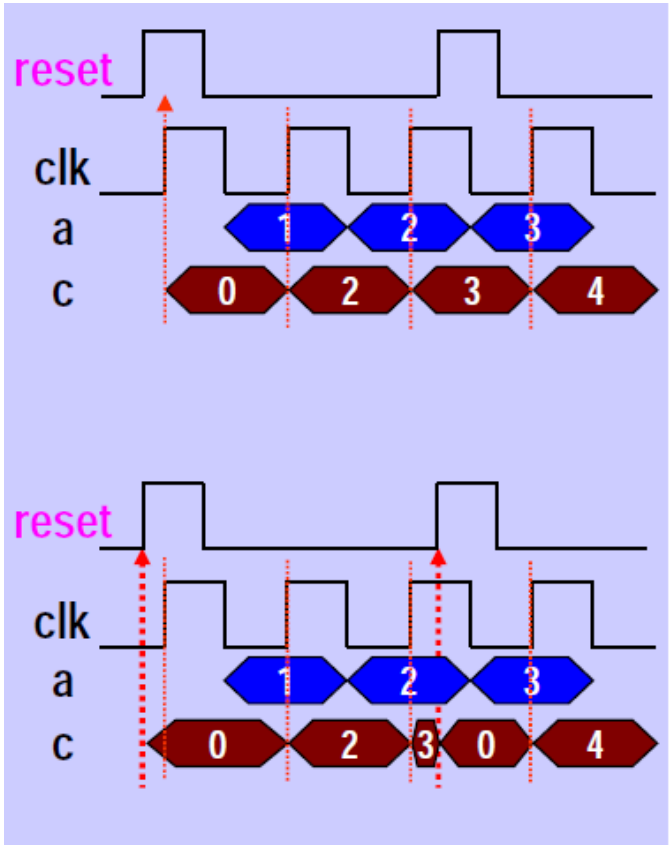
Event-Based Timing Control Example

- ✓ **Register with synchronous reset**
  - @(posedge clk): for synchronous reset

```
Ex.
 always @(posedge clk) begin
    if(reset) c <= 0;
  else c <= a+1;
end
```

- ✓ **Register with asynchronous reset**
  - @(posedge clk or posedge reset): for asynchron reset

```
Ex.
 always @(posedge clk or posedge reset) begin
    if(reset) c <= 0;
    else c <= a+1;
 end
```

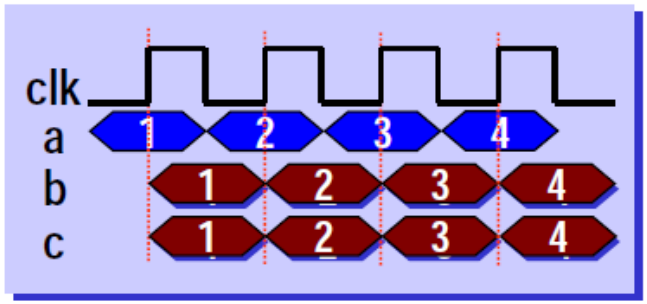


Procedural Assignments

- ✓ **The Verilog HDLcontains two types of procedural assignment**
  - **Blocking procedural assignment**
  - **Nonblocking procedural assignment**

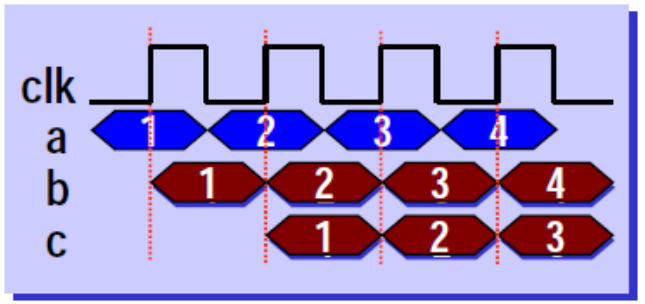
Blocking :

```
always @(posedge clk)
   begin
            b = a;
            c = b;
   end
```



Non-blocking :

```
always @(posedge clk)
   begin
            b <= a;
            c <= b;
   end
```



Procedural Assignments

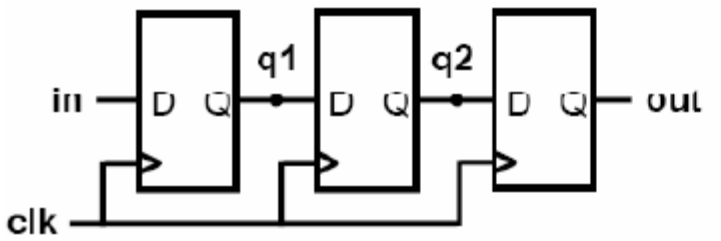
- ✓ Non-blocking assignment

Non-blocking

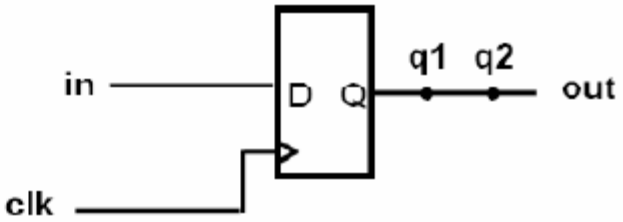
```
always @( posedge  clk)
begin
      q1 <=  in;
      q2 <=  q1;
      out <= q2;
end
```

Blocking

```
always @( posedge  clk)
begin
      q1  =  in;
      q2  =  q1;
      out  = q2;
end
```



Shift register behavior



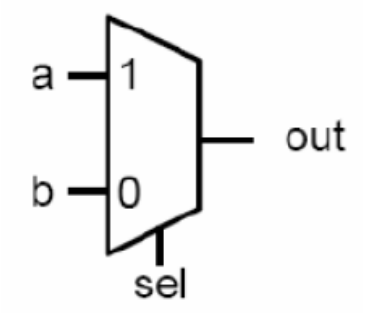
Single register behavior

Procedural Assignments

✓ Comparison

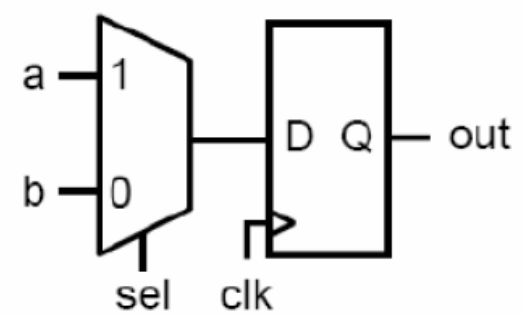
Blocking

```
module  Combinational(out,a,b,sel);
input         sel,a,b;
output        out;
reg           out;

always @(a  or  b  or  sel )
begin
      if( sel )       out  = a;
      else            out  = b;
end
endmodule
```



Non-blocking

```
module  Sequential(out,a,b,sel,clk);
input         sel,a,b,clk;
output        out;
reg           out;

always @(posedge  clk )
begin
      if( sel )       out  <= a;
      else            out  <= b;
end
endmodule
```



Event-Based Timing Control

- ✓ Event control can't be synthesized !!
- ✓ You can use changes on nets and register as event to trigger the execution of a statement.
- ✓ Syntax
  - ➢ @< event_expression > < statement_or _null >
    - Statement will wait for the data of <event_expression> changed, then execute statement.

- ✓ Example :
  - ➢ @(ee) rega = regb;
    // controlled by and value changes in the register ee;
  - ➢ @(posedge clk) rega = regb;
    // controlled by posedge on clk

```
initial  begin
         clk =  0;
         rst = 0;
         data_in = 0;
         #( cycle/2 ) rst = 1;
         @(  in_en ) data_in = 8'hff;
         @(  posedge clk ) data_in = 8'hff;
         for (i=0; i<=10; i=i+1) begin
                        @( posedge clk )
                        data_in = pattern[i];
         end
         ……
end
```

Event-Based Timing Control

- ✓ Use ***wait*** for level-sensitive timing control in behavioral code.
- ✓ The following behavioral model of an adder with a latched output illustrates edge-sensitive timing with the ***or*** keyword as well as level- sensitive timing with the ***wait*** statement.
- ✓ Note that wait is not synthesizable.

```
module latch_adder (out, a, b,enable);
input enable;
input [2:0]a,b;
output [3:0]out;
reg [3:0]out;
always @(a or b)  begin
        wait (!enable) // if enable is low, perform addition
        out = a + b;
end
endmodule
```

➢ When still waiting, changes of a or b would be ignored.

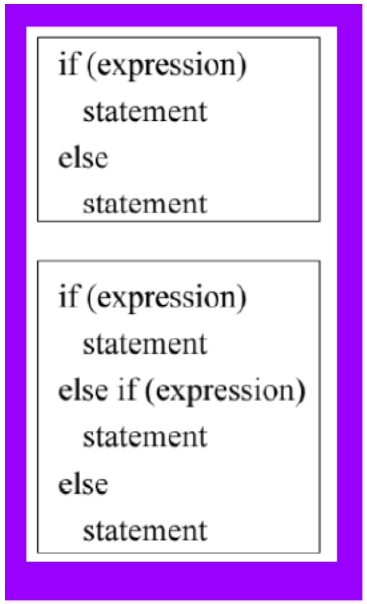
Behavioral Control Statements

- ✓ **Conditional Statements**
  - ➢ **if**
  - ➢ **ifelse**
  - ➢ **case**
- ✓ **Looping Statements**
  - ➢ **forever loop**
  - ➢ **repeat loop**
  - ➢ **while loop**
  - ➢ **for loop**

Behavioral Control Statements

❖ if and ifelse Statements

- ✓ In nested ***if*** sequences, ***else*** is associated with the closest previous ***if*** (*to avoid synthesis tool to produce latch devices*).
- ✓ If condition true (1) , the true_statement is executed. If **false (0) or ambiguous (x)**, the false_statement is executed
- ✓ To ensure proper readability and proper association, use ***begin...end*** block statements.




```
always@( posedge clock ) begin
if (index > 0) // Beginning of outer if
        if (rega > regb) // Beginning of the 1st inner if
                result = rega;
        else
                result = 0; // End of the 1st inner if
else
        if (index == 0)  begin
                $display("Note : Index is zero");
                result = regb;
        end
        else        $display("Note : Index is negative");
end
```

Behavioral Control Statements

- ✓ Conditional Statements: The conditional statement is decide whether toexecute a statement.
  - ➢ **if…,else if…,else…**: The most commonly used conditional statements. The statement occurs if the expressions controlling the if statement evaluates to be true.

```
module MUX2_1(out,a,b,sel);
input       a,b,sel;
output      out;
reg         out;

//Procedural assignment
always @(a or b or sel)
begin
   if (sel==0)   out  = a;
   else          out  = b;
end
endmodule
```

Anything assigned in an "*always*" block must also be declared as *reg* type

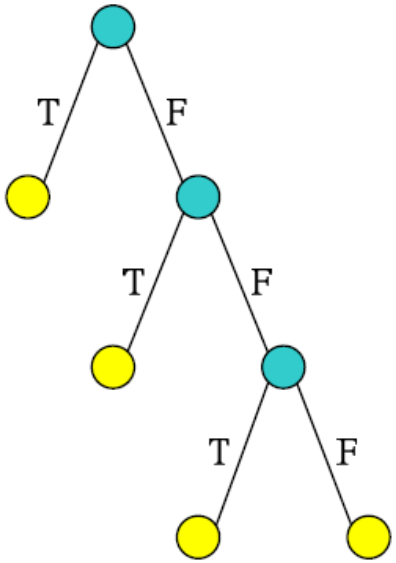
The "*always*" block runs once whenever a signal in the **sensitivity list** changes value

Behavioral Control Statements

❖ if else if statements

- ✓ The expressions are evaluated in order; if any expression is true, the statement associated with it is executed, and this terminates the whole chain. Each statement is either a single statement or a block statements.
- ✓ The last else part of the if-else-if construct handles the default case where none of the other conditions was satisfied.

```
always
  if (index < stage1)
      result = a + b;
  else if (index < stage2)
      result = a - b;
  else
      result = a;
```

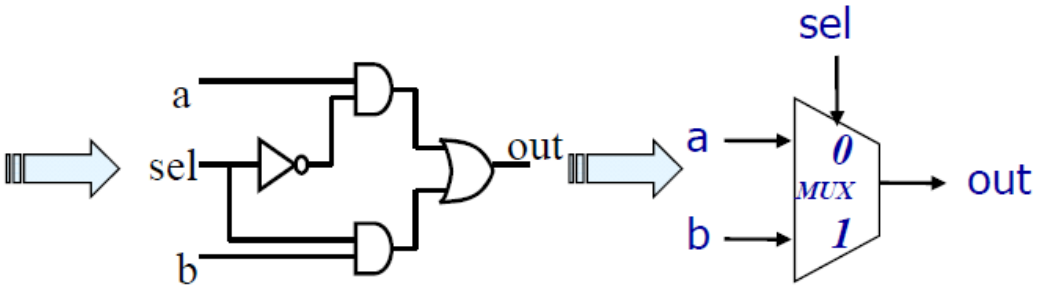


Behavioral Control Statements

✓Conditional Statements

- Syntax:
- assign <out_name> = (<expression>) ? true_statement : false_statement

```
module MUX2_1(out,a,b,sel);
input       a,b,sel;
output      out;
wire        out;

//Continuous assignment
assign  out = (sel==0)?a:b;

endmodule
```



Behavioral Control Statements

- **case** : (casex, casez) Be used for switching multiple selections.

- Syntax:

```
case(expression)
        alternative1 : statement1;
        alternative2 : statement2;
        alternative3 : statement3;
        ...
        default : default statement;
endcase
```

```
module MUX2_1(out,a,b,sel);
input       a,b,sel;
output      out;
reg         out;
//Procedural assignment
always @(a or b or sel) begin
  case(sel)
     1'b0:  out = a;
     1'b1:  out = b;
  endcase
end
endmodule
```

Behavioral Control Statements

- ✓ **Conditional Statements**
  - ➢ **if**
  - ➢ **ifelse**
  - ➢ **case**
- ✓ **Looping Statements**
  - ➢ **forever loop**
  - ➢ **repeat loop**
  - ➢ **while loop**
  - ➢ **for loop**

Behavioral Control Statements

- Four types
  - forever (can't synthesize)
    - Continuously executes a statement until to meet $finish or disable.
  - repeat
    - Executes a statement a fixed number of times.
  - while
    - Executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all.
  - for
    - Controls execution of its associated statements by a three step process
    - Note : The above is used to run simulation !!

Behavioral Control Statements

- for-loop syntax
  - for (initial_assignment; condition; step_assignment) begin
    statement;
    end
  - Can use to initialize a memory
    EX.

```
integer  i;
always@(posedge clk or posedge reset)
begin
  if(reset)
  begin
     for(i=0; i<1024; i=i+1)
     begin
        memory[i]=0;
     end
  end
  else
   .
   .
   .
```

Behavioral Control Statements

- ✓ Example
  - forever @(posedge clk) rega =~rega;
    - Need timing control to avoid dead-lock

      EX. forever rega = ~ rega // dead-lock !!
  - repeat(size) begin // if size = 5 then do loop five times
    statement
    end
  - While(temp) begin // do loop until temp=0 (false)
    statement
    temp >> 1;
    end

Assignments

- The assignment is always active
  - Whenever any change on the RHS of the assignment occurs, it is evaluated and assigned to the LHS.

```
wire [ 3:0 ] a;
assign a = b + c;        // continuous assignment
```

- Net declaration assignment
  - An equivalent way of writing net assignment statement.
  - Can be declared once for a specific net.

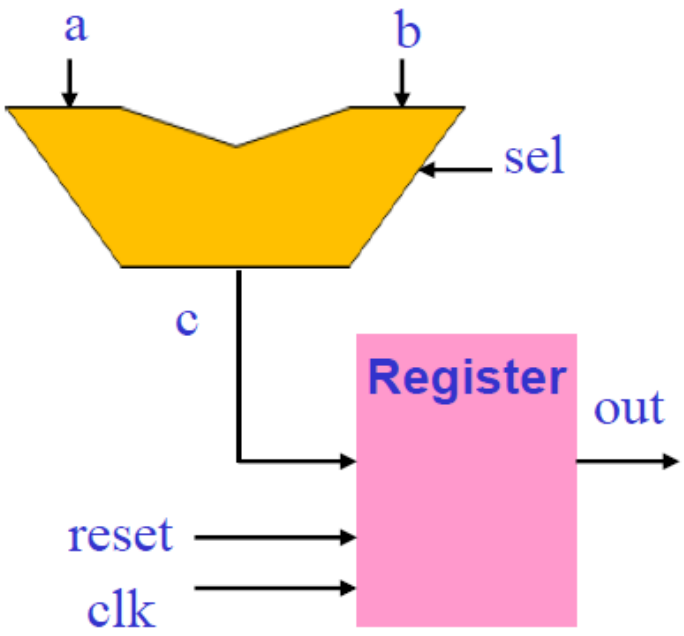
```
wire [ 3:0 ] a = b + c;
```

- In the implicit continuous assignment statement, It's not allowed which required a concatenation on the LHS.

```
wire [ 7:0 ] {co, sum} = a + b + ci;
```

Error!!

Behavioral Modeling Example




```
module MUX2_1(out,a,b,sel,clk,reset);
input       sel,clk,reset;
input       [7:0]       a,b;
output      [7:0]       out;
wire        [7:0]       c;
reg         [7:0]       out;

//Continuous assignment
assign  c = (sel==0)?a:b;

//Procedural assignment
always @(posedge clk or posedge reset)
begin
            if(reset==1) out <=  0;
            else out <=  c;
end
endmodule
```

About reset

- D-flip-flop with synchronous set and reset example:

```
module dff(q, d, clk, set, rst);
    input d, clk, set, rst;
    output q;
    reg q;

    always @(posedge clk)
        if (rst)
            q <= 1'b0;
        else if (set)
            q <= 1'b1;
        else
            q <= d;
endmodule
```

This gives priority to reset over set and set over d.

Common Mistakes

- Data has to be described in one always block

```
always@(posedge clk)
        out <= out + 1;
always@(posedge clk)
        out <= a;
```

Wrong!!

- Data has to be described by either blocking assignment or non-blocking assignment.

```
always@(posedge clk or posedge reset)
        if(reset) out = 0;
        else      out <= out + in;
```

Wrong!!

Thank you for your attention!