KYUNGPOOK NATIONAL UNIVERSITY

COMP311-1 Logic Circuit Design Chap. 7

Instructor: Taigon Song (송대건)

2019 Fall

“Output z” or “output reg z”?

```
output y;
assign y=a+b;
```

```
output reg y;
always@(*) 를 쓰는 경우 reg 사용
(if,else,while,for,case 등)
```

Levels of Design Abstraction

- Behavioral or algorithmic level
  - Highest level of abstraction. Almost as similar to C programming
- Dataflow level
  - How data flows between HW registers and how data is processed
- Gate level
  - AND, OR, INV…etc
- Switch level
  - Transistor level

7.1 Structured Procedures

7.1.1 'initial' Statement

- An 'initial' block
  - Starts at time 0
  - Executes exactly once during simulation
  - If multiple initial blocks → each block executes concurrently (동시) at time 0
  - 'begin' and 'end' to gather multiple statements
  - When one behavioral statement → no grouping necessary

Example 7-1 initial Statement

```
module stimulus;

reg x,y, a,b, m;

initial
    m = 1'b0; //single statement; does not need to be grouped

initial
begin
    #5 a = 1'b1; //multiple statements; need to be grouped
    #25 b = 1'b0;
end

initial
begin
    #10 x = 1'b0;
    #25 y = 1'b1;
end

initial


    #50 $finish;

endmodule
```

time	statement executed
0	m = 1'b0;
5	a = 1'b1;
10	x = 1'b0;
30	b = 1'b0;
35	y = 1'b1;
50	$finish;

7.1 Structured Procedures

7.1.1 'initial' Statement

- Some other possible declaration/initialization styles

Example 7-2 Initial Value Assignment

```
//The clock variable is defined first
reg clock;
//The value of clock is set to 0
initial clock = 0;

//Instead of the above method, clock variable
//can be initialized at the time of declaration
//This is allowed only for variables declared
//at module level.
reg clock = 0;
```

Example 7-3 Combined Port/Data Declaration and Variable Initialization

```
module adder (sum, co, a, b, ci);
output reg [7:0] sum = 0; //Initialize 8 bit output sum
output reg       co  = 0; //Initialize 1 bit output co
input      [7:0] a, b;
input            ci;

--
--
endmodule
```

Example 7-4 Combined ANSI C Port Declaration and Variable Initialization

```
module adder (output reg [7:0] sum = 0, //Initialize 8 bit output
              output reg       co  = 0, //Initialize 1 bit output co
              input      [7:0] a, b,
              input            ci
              );
--
--
endmodule
```

하드웨어는 초기화 안됨

7.1 Structured Procedures

7.1.2 'always' Statement

- An 'always' block
  - Starts at time 0
  - Executes the statements continuously in a looping fashion
  - Used to model a block of activity that is repeated continuously
    - An example: clock generator

Example 7-5 always Statement

```
module clock_gen (output reg clock);

//Initialize clock at time zero
initial
        clock = 1'b0;

//Toggle clock every half-cycle (time period = 20)
always
        #10 clock = ~clock;

initial
        #1000 $finish;

endmodule
```

7.2 Procedural Assignments

- Procedural assignments update values of reg, integer, real, or time variables
  - Value on a variable will remain unchanged until another procedural assignment updates the variable with a different value

```
module mux21(output y, input i0, i1, s);

  // #4 using assign statements (1/2)
  assign y = (i0 & ~s) | (i1 & s);
endmodule
```

assign 문 안의 변수 중 하나라도 바뀌면 그 즉시 y값이 바뀜

```
`timescale 1ns / 1ns
module mux21_tb();
  wire y;
  reg i0, i1, s;

  initial begin
    i0=0;
    i1=0;
    s=0;
  end

  always
    #1 i0 = ~i0;

  always
    #2 i1 = ~i1;

  always
    #4 s = ~s;

  mux21 umux21(y, i0, i1, s);

endmodule
```

명령을 하면 값이 바뀜

7.2 Procedural Assignments

7.2.1 Blocking Assignments

- Executed in the order they are specified in a sequential block
- '=' specify blocking assignments

Example 7-6 Blocking Statements 차례대로 실행

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;

//All behavioral statements must be inside an initial or always block
initial
begin
        x = 0; y = 1; z = 1; //Scalar assignments
        count = 0; //Assignment to integer variables
        reg_a = 16'b0; reg_b = reg_a; //initialize vectors

        #15 reg_a[2] = 1'b1; //Bit select assignment with delay
        #10 reg_b[15:13] = {x, y, z} //Assign result of concatenation
to
                                     // part select of a vector
        count = count + 1; //Assignment to an integer (increment)
end
```

- Note
  - For procedural assignments to registers, left-hand-side/right-hand-side bits will always match to the left-hand-side bits
    - E.g., 4bit/5bit, 5bit/4bit

7.2 Procedural Assignments

7.2.2 Non-blocking Assignments

- '<=‘ operator used to specify non-blocking assignments
  - Q) Which is operated first?

동시에 실행

Example 7-7 Nonblocking Assignments

```
reg x, y, z;
reg [15:0] reg_a, reg_b;
integer count;


//All behavioral statements must be inside an initial or always block
initial
begin
        x = 0; y = 1; z = 1; //Scalar assignments
        count = 0; //Assignment to integer variables
        reg_a = 16'b0; reg_b = reg_a; //Initialize vectors

        reg_a[2] <= #15 1'b1; //Bit select assignment with delay
       reg_b[15:13] <= #10 {x, y, z}; //Assign result of concatenation
                                      //to part select of a vector
        count <= count + 1; //Assignment to an integer (increment)
end
```

- Note
  - Typically, non-blocking assignments are executed last in the time step (after all blocking assignments in the time step are executed)
  - Not recommended to mix blocking/non-blocking assignments in the same always block

7.2 Procedural Assignments

7.2.2 Non-blocking Assignments

- Application of non-blocking assignments
  - Used as a method to model several concurrent data transfers that take place after a common event

```
always @(posedge clock)
begin
    reg1 <= #1 in1;
    reg2 <= @(negedge clock) in2 ^ in3;
    reg3 <= #1 reg1; //The old value of reg1
end
```

  - Event order
    - 1. Read operations on right-hand-side variable in1, in2, in3, and reg1
    - 2. Write operations scheduled
    - 3. Write operations executed at the scheduled time steps

7.2 Procedural Assignments

7.2.2 Non-blocking Assignments

- Compare the two operations

Example 7-8 Nonblocking Statements to Eliminate Race Conditions

```
//Illustration 1: Two concurrent always blocks with blocking
//statements
always @(posedge clock)
            a = b;

always @(posedge clock)
            b = a;
```

What would happen here?

```
//Illustration 2: Two concurrent always blocks with nonblocking
//statements
always @(posedge clock)
            a <= b;

always @(posedge clock)
            b <= a;
```

Example 7-9 Implementing Nonblocking Assignments using Blocking Assignments

```
//Emulate the behavior of nonblocking assignments by
//using temporary variables and blocking assignments
always @(posedge clock)
begin
   //Read operation
   //store values of right-hand-side expressions in temporary variables
    temp_a = a;
    temp_b = b;
    //Write operation
    //Assign values of temporary variables to left-hand-side variables
    a = temp_b;
    b = temp_a;
end
```

7.3 Timing Controls

7.3.1 Delay-based Timing Control – Regular delay control

- Three types of delay control for procedural assignments
  - Regular delay control, intra-assignment delay control, and zero delay control

Example 7-10 Regular Delay Control

```
//define parameters
parameter latency = 20;
parameter delta = 2;
//define register variables
reg x, y, z, p, q;

initial
begin

        x = 0; // no delay control
        #10 y = 1; // delay control with a number. Delay execution of
                                // y = 1 by 10 units

        #latency z = 0; // Delay control with identifier. Delay of 20
units
        #(latency + delta) p = 1; // Delay control with expression

        #y x = x + 1; // Delay control with identifier. Take value of
y.

        #(4:5:6) q = 0; // Minimum, typical and maximum delay values.
                        //Discussed in gate-level modeling chapter.
end
```

7.3 Timing Controls

7.3.1 Delay-based Timing Control – Intra-assignment delay control

- Assign a delay to the right of the assignment operator

Example 7-11 Intra-assignment Delays

```
//define register variables
reg x, y, z;

//intra assignment delays
initial
begin
        x = 0; z = 0;
        y = #5 x + z; //Take value of x and z at the time=0, evaluate
                      //x + z and then wait 5 time units to assign value
                     //to y.

end

//Equivalent method with temporary variables and regular delay control
initial
begin
        x = 0; z = 0;
        temp_xz = x + z;
        #5 y = temp_xz; //Take value of x + z at the current time and
                  //store it in a temporary variable. Even though x and
z
                  //might change between 0 and 5,
                 //the value assigned to y at time 5 is unaffected.
end
```

7.3 Timing Controls

7.3.1 Delay-based Timing Control – Zero delay control

- Order of execution of statements in different always-initial blocks is nondeterministic
  - Zero delay control ensures that a statement is executed last, after all other statements in that simulation time are executed
  - To eliminate race conditions
  - Order between multiple zero delay statements are nondeterministic

Example 7-12 Zero Delay Control

```
initial
begin
    x = 0;
    y = 0;
end

initial
begin
    #0 x = 1; //zero delay control
    #0 y = 1;
end
```

- Note,
  - Not recommended to use #0 in practice

7.3 Timing Controls

7.3.2 Event-based Timing Control

- An event is the change in the value on a register or a net. Events can be used to trigger execution of a statement or multiple statements
  - Regular event control, named event control, event OR control, and level-sensitive timing control
  - '@' for regular event control

input에 따라 output이 즉각 반응할때

Example 7-13 Regular Event Control

@(이벤트) 이벤트가 변할때마다 할 일

```
@(clock) q = d; //q = d is executed whenever signal clock changes value
@(posedge clock) q = d; //q = d is executed whenever signal clock does
                        //a positive transition ( 0 to 1,x or z,
                        // x to 1, z to 1 )
@(negedge clock) q = d; //q = d is executed whenever signal clock does
                        //a negative transition ( 1 to 0,x or z,
                        //x to 0, z to 0)
q = @(posedge clock) d; //d is evaluated immediately and assigned
                        //to q at the positive edge of clock
```

7.3 Timing Controls

7.3.2 Event-based Timing Control

- Named event control
  - User can declare an event then trigger and recognize the occurrence of the event

Example 7-14 Named Event Control

```
//This is an example of a data buffer storing data after the
//last packet of data has arrived.

event received_data; //Define an event called received_data

always @(posedge clock) //check at each positive clock edge
begin
        if(last_data_packet) //If this is the last data packet
                ->received_data; //trigger the event received_data
end

always @(received_data) //Await triggering of event received_data
                        //When event is triggered, store all four
                        //packets of received data in data buffer
                       //use concatenation operator { }
        data_buf = {data_pkt[0], data_pkt[1], data_pkt[2],
data_pkt[3]};
```

7.3 Timing Controls

7.3.2 Event-based Timing Control

- Event OR Control
  - A transition of any one of multiple signals/events can trigger the execution of a statement or a block of statements

Example 7-15 Event OR Control (Sensitivity List)

여러 이벤트들 중 하나라도 변하는걸 확인할때 : 'or' ' , ' '*' 등으로 사용가능

```
//A level-sensitive latch with asynchronous reset
always @( reset or clock or d)
                              //Wait for reset or clock or d to
change
begin
        if (reset)                 //if reset signal is high, set q to 0.
                q = 1'b0;
        else    if(clock)          //if clock is high, latch input
                q = d;
end
```

Example 7-16 Sensitivity List with Comma Operator

```
//A level-sensitive latch with asynchronous reset
always @( reset, clock, d)
                              //Wait for reset or clock or d to
change
begin
        if (reset)                 //if reset signal is high, set q to 0.
                q = 1'b0;
        else    if(clock)          //if clock is high, latch input
                q = d;
end


//A positive edge triggered D flipflop with asynchronous falling
//reset can be modeled as shown below
always @(posedge clk, negedge reset) //Note use of comma operator
if(!reset)
   q <=0;
else
   q <=d;
```

Example 7-17 Use of @* Operator

```
//Combination logic block using the or operator
//Cumbersome to write and it is easy to miss one input to the block
always @(a or b or c or d or e or f or g or h or p or m)

begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end

//Instead of the above method, use @(*) symbol
//Alternately, the @* symbol can be used
//All input variables are automatically included in the
//sensitivity list.
always @(*)
begin
out1 = a ? b+c : d+e;
out2 = f ? g+h : p+m;
end
```

7.3 Timing Controls

7.3.3 Level-Sensitive Timing Control

- Verilog allows level-sensitive timing control
  - '@' is for edge-sensitive control
  - Executes when a certain condition is 'true'
  - Keyword 'wait'

엣지가 아닌 레벨 기준으로 컨트롤할때 wait 사용

```
always
    wait (count_enable) #20 count = count + 1;
```

7.4 Conditional Statements

Keyword 'if'

if문

```
//Type 1 conditional statement. No else statement.
//Statement executes or does not execute.
if (<expression>) true_statement ;

//Type 2 conditional statement. One else statement
//Either true_statement or false_statement is evaluated
if (<expression>) true_statement ; else false_statement ;

//Type 3 conditional statement. Nested if-else-if.
//Choice of multiple statements. Only one is executed.
if (<expression1>) true_statement1 ;
else if (<expression2>) true_statement2 ;
else if (<expression3>) true_statement3 ;
else default_statement ;
```

Example 7-18 Conditional Statement Examples

```
//Type 1 statements
if(!lock) buffer = data;
if(enable) out = in;

//Type 2 statements
if (number_queued < MAX_Q_DEPTH)
begin
        data_queue = data;
        number_queued = number_queued + 1;
end
else
        $display("Queue Full. Try again");

//Type 3 statements
//Execute statements based on ALU control signal.
if (alu_control == 0)
        y = x + z;

else if(alu_control == 1)
        y = x - z;
else if(alu_control == 2)
        y = x * z;
else
        $display("Invalid ALU control signal");
```

7.5 Multiway Branching

7.5.1 'case' Keyword

case문
변수가 정상적인 값이 아닐 경우를 대비해
default 동작을 설정해준다.

- Default statement optional

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

Example 7-19 4-to-1 Multiplexer with Case Statement

```
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0}) //Switch based on concatenation of control signals
        2'd0 : out = i0;
        2'd1 : out = i1;
        2'd2 : out = i2;
        2'd3 : out = i3;
        default: $display("Invalid control signals");
endcase

endmodule
```

7.5 Multiway Branching

7.5.2 'casex', 'casez' Keywords

casez 에서 10xz는 1xxx로 가지만
casex 에서는 default로 간다.

- Two variations of the 'case' statement
  - casex: all x and z values as don't cares
  - casez: all z values as don't cares

Example 7-21 casex Use

```
reg [3:0] encoding;
integer state;

casex (encoding) //logic value x represents a don't care bit.
4'b1xxx : next_state = 3;
4'bx1xx : next_state = 2;
4'bxx1x : next_state = 1;
4'bxxx1 : next_state = 0;
default : next_state = 0;
endcase
```

- Q) What happens to 'encoding = 4'b10xz'?

7.6 Loops

7.6.1 'while' Loop

- Executes until the while expression is not true

Example 7-22 While Loop

```
//Illustration 1: Increment count from 0 to 127. Exit at count 128.
//Display the count variable.
integer count;

initial
begin
        count = 0;

        while (count < 128) //Execute loop till count is 127.
                            //exit at count 128
        begin
                $display("Count = %d", count);
               count = count + 1;
        end
end
```

```
//Illustration 2: Find the first bit with a value 1 in flag (vector
variable)
'define TRUE 1'b1';
'define FALSE 1'b0;
reg [15:0] flag;
integer i; //integer to keep count
reg continue;

initial
begin
  flag = 16'b 0010_0000_0000_0000;
  i = 0;
  continue = 'TRUE;


   while((i < 16) && continue ) //Multiple conditions using operators.
   begin
     if (flag[i])
     begin
       $display("Encountered a TRUE bit at element number %d", i);
       continue = 'FALSE;
     end
     i = i + 1;
   end
end
```

Q) When will the $display statement execute?

7.6 Loops

7.6.2 'for' Loop

- Three parts for the 'for' loop:
  - an initial condition
  - a check for the terminating condition
  - a procedural assignment to change value of the control variable

Example 7-23 For Loop

```
integer count;

initial
    for ( count=0; count < 128; count = count + 1)
          $display("Count = %d", count);
```

```
//Initialize array elements
'define MAX_STATES 32
integer state [0: 'MAX_STATES-1]; //Integer array state with elements
0:31
integer i;

initial
begin
   for(i = 0; i < 32; i = i + 2) //initialize all even locations with 0
      state[i] = 0;

   for(i = 1; i < 32; i = i + 2) //initialize all odd locations with 1
      state[i] = 1;
end
```

7.6 Loops

7.6.3 'repeat' Loop

repeat() 괄호안에는 숫자만 사용 가능, 지정된 변수도 가능

- Executes the loop for a fixed number of times
  - cannot be used to loop on a general logical expression
  - use constant (numbers) only
  - if variable, evaluated only when the loop starts

Example 7-24 Repeat Loop

```
//Illustration 1 : increment and display count from 0 to 127
integer count;

initial
begin
   count = 0;
   repeat(128)
   begin
        $display("Count = %d", count);
        count = count + 1;
   end
end
```

```
//Illustration 2 : Data buffer module example
//After it receives a data_start signal.
//Reads data for next 8 cycles.

module data_buffer(data_start, data, clock);

parameter cycles = 8;
input data_start;
input [15:0] data;
input clock;

reg [15:0] buffer [0:7];
integer i;

always @(posedge clock)
begin
  if(data_start) //data start signal is true
  begin
    i = 0;
    repeat(cycles) //Store data at the posedge of next 8 clock
                   //cycles
    begin
      @(posedge clock) buffer[i] = data; //waits till next
                                         // posedge to latch data
      i = i + 1;
    end
  end
end

endmodule
```

7.6 Loops

7.6.4 'forever' Loop

컨트롤 어려워서 잘안씀

- Executes forever until the $finish task is encountered
  - runs FOREVER!

Example 7-25 Forever Loop

```
//Example 1: Clock generation
//Use forever loop instead of always block
reg clock;

initial
begin
        clock = 1'b0;
        forever #10 clock = ~clock; //Clock with period of 20 units
end

//Example 2: Synchronize two register values at every positive edge of
//clock
reg clock;
reg x, y;

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

이것도 포에버문이다

7.7 Sequential and Parallel Blocks

7.7.1 Block Types

- Sequential blocks 순차적 진행
  - keywords ‘begin’ and 'end’
- Parallel blocks 동시에 진행
  - keywords ‘fork’ and ‘join’

Example 7-26 Sequential Blocks

```
//Illustration 1: Sequential block without delay
reg x, y;
reg [1:0] z, w;

initial
begin
        x = 1'b0;
        y = 1'b1;
        z = {x, y};
        w = {y, x};

end
```

```
//Illustration 2: Sequential blocks with delay.
reg x, y;
reg [1:0] z, w;

initial
begin
        x = 1'b0; //completes at simulation time 0
        #5 y = 1'b1; //completes at simulation time 5
        #10 z = {x, y}; //completes at simulation time 15
        #20 w = {y, x}; //completes at simulation time 35
end
```

```
//Parallel blocks with deliberate race condition
reg x, y;
reg [1:0] z, w;

initial
fork
        x = 1'b0;
        y = 1'b1;
        z = {x, y};
        w = {y, x};
join
```

Example 7-27 Parallel Blocks

```
//Example 1: Parallel blocks with delay.
reg x, y;
reg [1:0] z, w;

initial
fork
        x = 1'b0; //completes at simulation time 0
        #5 y = 1'b1; //completes at simulation time 5
        #10 z = {x, y}; //completes at simulation time 10
        #20 w = {y, x}; //completes at simulation time 20
join
```

7.7 Sequential and Parallel Blocks

7.7.2 Special Features of Blocks

- Nested Blocks

모듈안에 모듈은 안되지만 블록안에 블록은 가능

  - Blocks can be nested and sequential/parallel blocks can be mixed

Example 7-28 Nested Blocks

```
//Nested blocks
initial
begin
        x = 1'b0;

        fork
                #5 y = 1'b1;
                #10 z = {x, y};
        join
        #20 w = {y, x};
end
```

7.7 Sequential and Parallel Blocks

7.7.2 Special Features of Blocks

- Named Blocks 블록에 이름 지정 : 반복문 탈출하거나 할때 쓰임
  - Blocks can have names
    - names can be local variables
    - part of the design hierarchy (for debugging purposes)
    - can be disabled (i.e., 'break' in C/C++)

Example 7-29 Named Blocks

```
//Named blocks
module top;

initial
begin: block1 //sequential block named block1
integer i; //integer i is static and local to block1
           // can be accessed by hierarchical name, top.block1.i
...
...
end

initial
fork: block2 //parallel block named block2
reg i; // register i is static and local to block2
                // can be accessed by hierarchical name, top.block2.i
...
...
join
```

7.7 Sequential and Parallel Blocks

7.7.2 Special Features of Blocks

- Disabling named blocks
  - can disable any named block in the design

Example 7-30 Disabling Named Blocks

```
//Illustration: Find the first bit with a value 1 in flag (vector
//variable)
reg [15:0] flag;
integer i; //integer to keep count

initial
begin
  flag = 16'b 0010_0000_0000_0000;
  i = 0;
  begin: block1 //The main block inside while is named block1
  while(i < 16)
   begin
      if (flag[i])
      begin
          $display("Encountered a TRUE bit at element number %d", i);
          disable block1; //disable block1 because you found true bit.
      end
      i = i + 1;
   end
  end
end
```

7.8 Generate Blocks

시험안나옴

- Allow Verilog code to be generated dynamically at elaboration time
  - facilitates the creation of “parameterized models”
  - allow control over the declaration of variables, functions, and tasks
- Generated instantiations can be one or more of the following types
  - modules, user-defined primitives (UDPs), Verilog gate primitives, continuous assignments, initial and always blocks
- Generate statements permit the following Verilog data types to be declared within the generate scope:
  - net, reg, integer, real time, realtime, event
- Generated data types have unique identifier names and can be referenced hierarchically
- Parameter redefinition using ordered or named assignment or a defparam statement can be declared with the generate scope

7.8 Generate Blocks

- Task and function declarations are permitted within the generate scope but not within a generate loop
  - generated tasks and functions have unique identifier names and can be referenced hierarchically
- Some module declarations and module items are not permitted in a generate statement
  - parameters, local parameters, input, output, inout
  - specify blocks

- Three methods to create generate statements
  - generate loop
  - generate conditional
  - generate case

7.8 Generate Blocks

7.8.1 Generate Loop

- Permits one or more of the following to be instantiated multiple times using a ‘for’ loop:
  - Variable declarations
  - Modules
  - User defined primitives, gate primitives
  - Continuous assignments
  - Initial and always blocks

Example 7-31 Bit-wise Xor of Two N-bit Buses

```
// This module generates a bit-wise xor of two N-bit buses

module bitwise_xor (out, i0, i1);
// Parameter Declaration. This can be redefined
parameter N = 32; // 32-bit bus by default
// Port declarations
output [N-1:0] out;
input [N-1:0] i0, i1;

// Declare a temporary loop variable. This variable is used only
// in the evaluation of generate blocks. This variable does not
// exist during the simulation of a Verilog design
genvar j;

//Generate the bit-wise Xor with a single loop
generate for (j=0; j<N; j=j+1) begin: xor_loop
 xor g1 (out[j], i0[j], i1[j]);
end //end of the for loop inside the generate block
endgenerate //end of the generate block

// As an alternate style,
// the xor gates could be replaced by always blocks.
// reg [N-1:0] out;
//generate for (j=0; j<N; j=j+1) begin: bit
// always @(i0[j] or i1[j]) out[j] = i0[j] ^ i1[j];
//end
//endgenerate

endmodule
```

7.8 Generate Blocks

7.8.2 Generate Conditional

- Is like an if-else-if generate construct that permits the following Verilog constructs to be conditionally instantiated into another module based on an expression that is deterministic at the time the design is elaborated:
  - modules
  - user-defined primitives, gate primitives
  - continuous assignments
  - initial and always blocks

Example 7-33 Parametrized Multiplier using Generate Conditional

```
// This module implements a parametrized multiplier

module multiplier (product, a0, a1);
// Parameter Declaration. This can be redefined
parameter a0_width = 8; // 8-bit bus by default
parameter a1_width = 8; // 8-bit bus by default

// Local Parameter declaration.
// This parameter cannot be modified with defparam or
// with module instance # statement.
localparam product_width = a0_width + a1_width;

// Port declarations
output [product_width -1:0] product;
input [a0_width-1:0] a0;
input [a1_width-1:0] a1;


// Instantiate the type of multiplier conditionally.
// Depending on the value of the a0_width and a1_width
// parameters at the time of instantiation, the appropriate
// multiplier will be instantiated.
generate
 if (a0_width <8) || (a1_width < 8)
    cla_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
 else
    tree_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
endgenerate //end of the generate block

endmodule
```

7.8 Generate Blocks

7.8.3 Generate Case

- Permits the following Verilog constructs to be conditionally instantiated into another module based on a select-one-of-many case construct that is deterministic at the time the design is elaborated
  - modules
  - user-defined primitives, gate primitives
  - continuous assignments
  - initial and always blocks

Example 7-34 Generate Case Example

```
// This module generates an N-bit adder

module adder(co, sum, a0, a1, ci);
// Parameter Declaration. This can be redefined
parameter N = 4; // 4-bit bus by default

// Port declarations
output [N-1:0] sum;
output co;
input [N-1:0] a0, a1;
input ci;

// Instantiate the appropriate adder based on the width of the bus.
// This is based on parameter N that can be redefined at
// instantiation time.
generate
case (N)
  //Special cases for 1 and 2 bit adders
  1: adder_1bit adder1(c0, sum, a0, a1, ci); //1-bit implementation
  2: adder_2bit adder2(c0, sum, a0, a1, ci); //2-bit implementation
  // Default is N-bit carry look ahead adder
  default: adder_cla #(N) adder3(c0, sum, a0, a1, ci);
endcase
endgenerate //end of the generate block

endmodule
```