Training Course
on

FPGA based Digital Design using Verilog HDL

By

NAUMAN MIR
(HDL Designer)

**** Organized by*** **Skill Development Council,**
(Ministry of Labour, Manpower and overseas Pakistani)
Govt. of Pakistan.

Logical Operators

- && → logical AND
- || → logical OR
- ! → logical NOT
- Operands evaluated to ONE bit value: *0*, *1* or *x*
- Result is ONE bit value: *0*, *1* or *x*

```
A = 6;        A && B → 1 && 0 → 0
B = 0;  ⇒     A || !B → 1 || 1 → 1
C = x;        C || B → x || 0 → x
```

but C&&B=0

Bitwise Operators

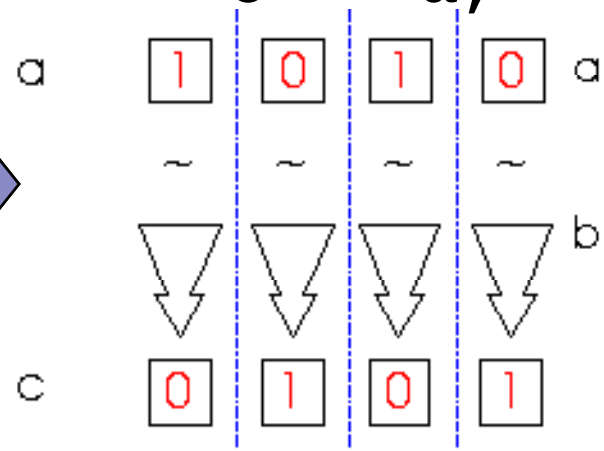
- & → bitwise AND
- | → bitwise OR
- ~ → bitwise NOT
- ^ → bitwise XOR
- ~^ or ^~ → bitwise XNOR

- Operation on bit by bit basis

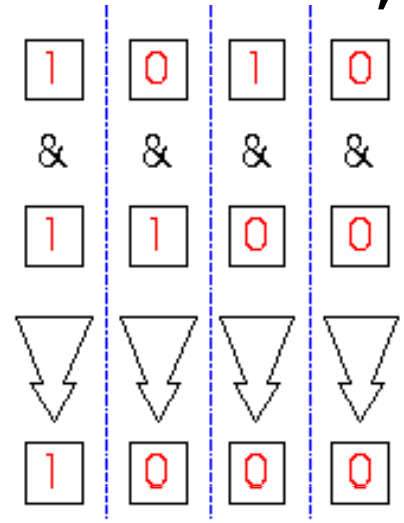
Bitwise Operators (contd)

- a = 4'b1010;
  b = 4'b1100;

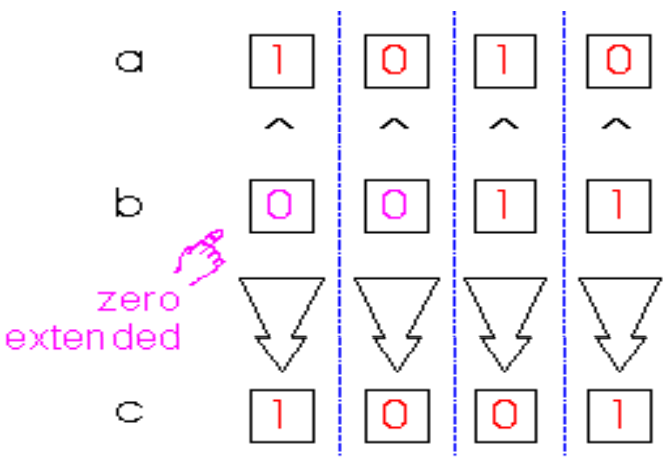
c = ~a;



c = a & b;



c = a ^ b;



- a = 4'b1010;
  b = 2'b11;

Reduction Operators

1. & → AND
2. ^ → XOR
3. ~| → NOR
4. | → OR
5. ~& → NAND
6. ~^ or ^~ → XNOR

- One multi-bit operand → One single-bit result

```
a = 4'b1001;
..
c = |a ;      // c = 1|0|0|1 = 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
```

Concatenation Operator

- {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 !!
```

Concatenation Operator (contd)

- **Replication ..**

```
catr = {4{a}, b, 2{c}};
                    // catr = 1111_010_101101
```

Example: Multiplication of two 5-bits numbers.

A → Multiplicand	11100
B → Multiplier	10010
	---
ppo = A & {5{B[0]}}	00000
pp1 = A & {5{B[1]}}	11100X
pp2 = A & {5{B[2]}}	00000XX
pp3 = A & {5{B[3]}}	00000XXX
pp4 = A & {5{B[4]}}	11100XXXX

	111111000

Replication: Verilog Code of Multiplier

```
module mul_5x5(A, B, mul_out);    // Multiplier Unsigned X Unsigned Number
input [4:0]    A, B ;
output [9:0] mul_out ;

wire [4:0] pp0, pp1, pp2, pp3, pp4 ;
wire [9:0] mul_out ;

assign ppo = A & {5{B[0]}} ;
assign pp1 = A & {5{B[1]}} ;
assign pp2 = A & {5{B[2]}} ;
assign pp3 = A & {5{B[3]}} ;
assign pp4 = A & {5{B[4]}} ;

assign mul_out = pp0 + {pp1, 1'b0} + {pp2, 2'b0} + {pp3, 3'b0} +
                 {pp4, 4'b0} ;
endmodule
```

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          → 1
'b1x1 <= 0     → x
10 < z         → x
```

Equality Operators

- `==` → logical equality
- `!=` → logical inequality

} Return *0*, *1* or *x*

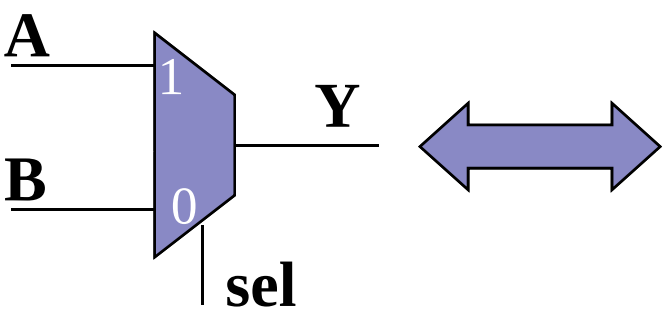
- `===` → case equality
- `!==` → case inequality

} Return *0* or *1*

- `4'b 1z0x == 4'b 1z0x` → *x*
- `4'b 1z0x != 4'b 1z0x` → *x*
- `4'b 1z0x === 4'b 1z0x` → *1*
- `4'b 1z0x !== 4'b 1z0x` → *0*

Conditional Operator

- cond_expr ? true_expr : false_expr
- Like a 2-to-1 mux ..




```
Y = (sel)? A : B;
```

Arithmetic Operators

- `+, -, *, /, %`
- If any operand is *x* the result is *x*
- Negative registers:
  - regs can be assigned negative but are treated as unsigned

```
reg [15:0] regA;
regA = -16'd12;
     //stored as 2^16-12 = 65524
```

→ `regA/3` evaluates to 21841

Operator Precedence

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

Use parentheses to enforce your priority

Behavioral Model - Procedures

- Procedures = sections of code that we know they execute sequentially
- Procedural statements = statements inside a procedure (they execute sequentially)
- e.g. another 2-to-1 mux implem:

Execution Flow

```
begin
if (sel == 0)
  Y = B;
else
  Y = A;
 end
```

Procedural assignments:
Y must be reg !!

Behavioral Model - Procedures (contd)

- Modules can contain any number of procedures
- Procedures execute in parallel (in respect to each other) and ..
- In behavioral modeling, everything comes in a procedural block

Behavioral Model - Procedures (contd)

- **Procedural block can be expressed in two types of blocks:**
  - **initial** → **they execute only once**
  - **always** → **they execute forever (until simulation finishes)**

"initial" Blocks

- This block starts with initial keyword
- This is not used in RTL
- This is non synthesizable
- All initial blocks execute concurrently in order independent
- They execute only once
- This block is used only in Test Bench

"initial" Blocks (contd)

- Start execution at sim time zero and finish when their last statement executes

```
module nothing;
initial
  $display("I'm first");
initial begin
  #50;
  $display("Really?");
  end
endmodule
```

Will be displayed at sim time 0

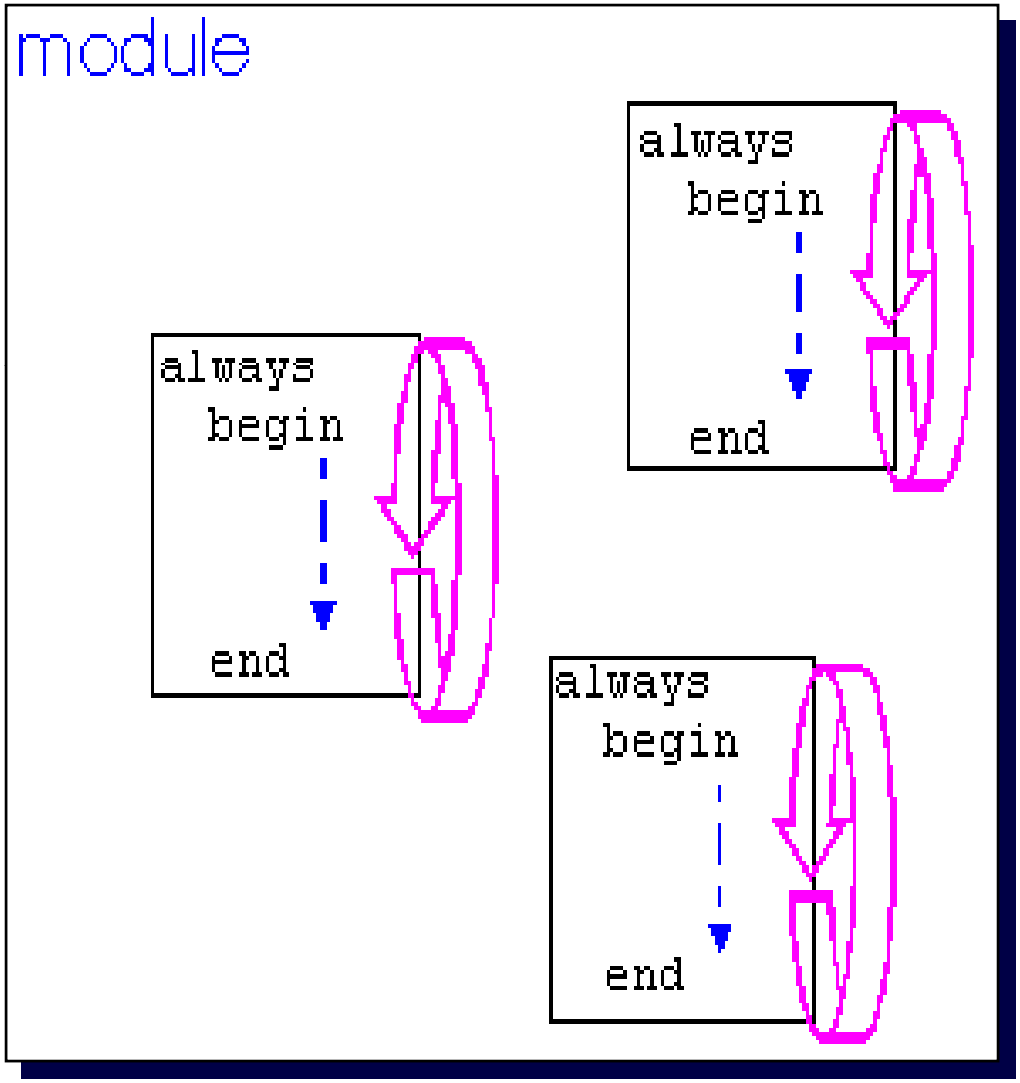
Will be displayed at sim time 50

"always" Blocks

- This block starts with always keyword
- This block is more like H/W
- *Always Blocks* execute forever until simulation finishes
- The always block can be viewed as continuously repeated activity in a digital circuit starting from power on

“always” Blocks (contd)

- Start execution at sim time zero and continue until sim finishes



Procedural assignments

- **Blocking assignment =**
  - Regular assignment inside procedural block
  - Assignment takes place immediately
  - LHS must be a register

```
always
begin
    X = Y
    Y = X
end
```

→ **X = Y and Y = Y**

Procedural assignments (contd)

- **Nonblocking assignment <=**
  - Compute right hand side
  - Assignment takes place at the end of block
  - LHS must be a register

```
always
begin
    X <= Y
    Y <= X
end
```

→ X = Y and Y = X

Procedural assignments (contd)

- Nonblocking statements are used whenever you want to make several register assignments within the same time step without regard to order or dependence upon each other
- They are executed in two steps:
  - the simulator evaluates the RHS
  - the assignment occurs at the end of the time step

Example: Blocking / Non-blocking

```
// Using Blocking Statement
module logic_1(clk, din, reg_b);
input   din, clk ;
output reg_b ;

reg reg_a, reg_b ;

always @(posedge clk)
begin
   reg_a = din ;
   reg_b = reg_a ;
end
endmodule
```

```
// Using Non-Blocking Statement
module logic_2(clk, din, reg_b);
input   din, clk ;
output reg_b ;

reg reg_a, reg_b ;

always @(posedge clk)
begin
   reg_a <= din ;
   reg_b <= reg_a ;
end
endmodule
```