

---


# Lecture 5: HDL IV

ECE 228

Mostafizur Rahman  
rahmanmo@umkc.edu

# Operator Precedence

---

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

Use parentheses to  
enforce your  
priority

# Verilog Operators

Verilog Operator	Name	Functional Group
()	bit-select or part-select	
()	parenthesis	
! ~ &   ~& ~  ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	Logical Bit-wise Reduction Reduction Reduction Reduction Reduction Reduction
+ -	unary (sign) plus unary (sign) minus	Arithmetic Arithmetic
{}	concatenation	Concatenation
{{}}	replication	Replication
* / %	multiply divide modulus	Arithmetic Arithmetic Arithmetic
+ -	binary plus binary minus	Arithmetic Arithmetic
<< >>	shift left shift right	Shift Shift

Verilog Operator	Name	Functional Group
> >= < <=	greater than greater than or equal to less than less than or equal to	Relational Relational Relational Relational
== !=	logical equality logical inequality	Equality Equality
=== !==	case equality case inequality	Equality Equality
&	bit-wise AND	Bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
	bit-wise OR	Bit-wise
&&	logical AND	Logical
	logical OR	Logical
?:	conditional	Conditional

# Open Questions

---

- Rotate is not available as a function, how to implement?
- Work in group:
  - How to implement  $w[i] := w[i-16] + s0$
  - $i$  indicates position in an array,  $W$  and  $S0$  are variables; draw block diagram first and then write HDL codes
- Will your code work for real circuits?

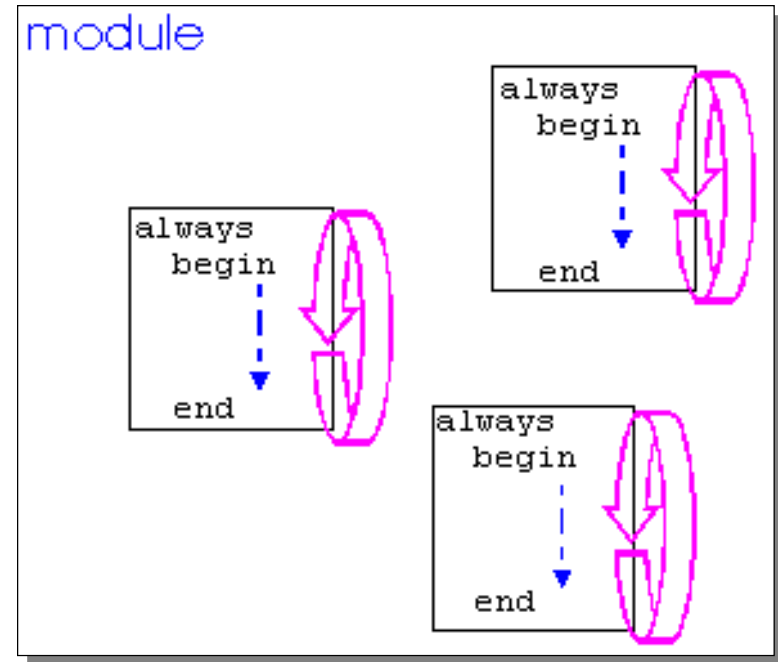
# Behavioral Model - Procedures (i)

---

- The procedural block defines
  - A region of code containing sequential statements
  - The statements execute in the order they are written
- Procedures: sections of code that we know they execute sequentially
- Procedural statements: statements inside a procedure (they execute sequentially)
- Two types of procedural blocks in Verilog
  - The “always” block
    - A continuous loop that never terminates
  - The “initial” block
    - Executed once at the beginning of simulation (used in Test-benches for the purpose of verification)

## Behavioral Model - Procedures (ii)

```
always @ (event_expression)
begin
statement;
.....
.....
statement;
end
```



- Modules can contain any number of procedures
- Procedures execute in parallel (in respect to each other) and ..

# An initial Block Example

---

- Executes exactly once during simulation
- Starts at simulation time 0

```
reg x, y, z;  
initial begin      // complex statement  
    x = 1`b0; y = 1`b1; z = 1`b0;  
#10    x = 1`b1; y = 1`b1; z = 1`b1;  
end  
initial x = 1`b0; // single statement
```

# An always Block Example

---

- Starts at simulation time 0
- Executes continuously during simulation

```
reg clock;  
initial clock = 1`b0;  
  
always #5 clock = ~clock;
```



# Event control statements

- An event occurs when a net or register changes its value. The event can be further specified as a rising edge (by **posedge**) or falling edge (by **negedge**) of a signal.

- `@`  
always @(signal1 or signal2 or ..) begin  
    ..  
end

execution triggers every  
time any signal changes

always @(posedge clk) begin  
    ..  
end

execution triggers every  
time clk changes  
from 0 to 1

always @(negedge clk) begin  
    ..  
end

execution triggers every  
time clk changes  
from 1 to 0

# Event Control

---

- Event Control
  - Edge Triggered Event Control
  - Level Triggered Event Control
- Edge Triggered Event Control
  - @ (posedge CLK) //Positive Edge of CLK
  - Curr\_State = Next\_state;

@ negedge	@ posedge
1 → x	0 → x
1 → z	0 → z
1 → 0	0 → 1
x → 0	x → 1
z → 0	z → 1

- Level Triggered Event Control
  - @ (A or B) //change in values of A or B
  - Out = A & B;

# Examples

---

- half adder implementation

```
module half_adder(S, C, A, B);  
  output S, C;  
  input A, B;  
  
  reg S,C;  
  wire A, B;  
  
  always @(A or B) begin  
    S = A ^ B;  
    C = A && B;  
  end  
  
endmodule
```

- Behavioral edge-triggered DFF implem

```
module dff(Q, D, Clk);  
  output Q;  
  input D, Clk;  
  
  reg Q;  
  wire D, Clk;  
  
  always @(posedge Clk)  
    Q = D;  
  
endmodule
```

Why not clock for adder and D for flipflop inside always?

# Procedural Statements: if

---

```
if (expr1)
    true_stmt1;

else if (expr2)
    true_stmt2;

..

else
    def_stmt;
```

E.g. 4-to-1 mux:

```
module mux4_1(out, in, sel);
    output out;
    input [3:0] in;
    input [1:0] sel;

    reg out;
    wire [3:0] in;
    wire [1:0] sel;

    always @(in or sel)
        if (sel == 0)
            out = in[0];
        else if (sel == 1)
            out = in[1];
        else if (sel == 2)
            out = in[2];
        else
            out = in[3];
endmodule
```

## Activity

---

- Write behavioral code for 4:1 multiplexer, with 4 inputs (A[3:0]) and selects (S[1:0]). Starting value for select should be 11

# Example of Flip-flop

---

```
module Flip_flop ( q, data_in, clk, rst );  
  input          data_in, clk, rst;  
  output         q;  
  reg  q;
```

```
  always @ ( posedge clk )
```

← *Declaration of synchronous behavior*

```
  begin
```

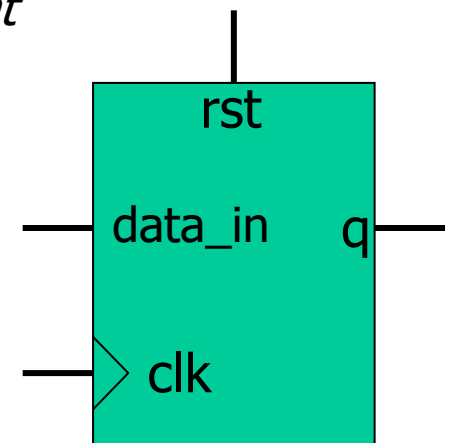
```
    if ( rst == 1) q = 0;
```

```
    else q = data_in;
```

```
  end
```

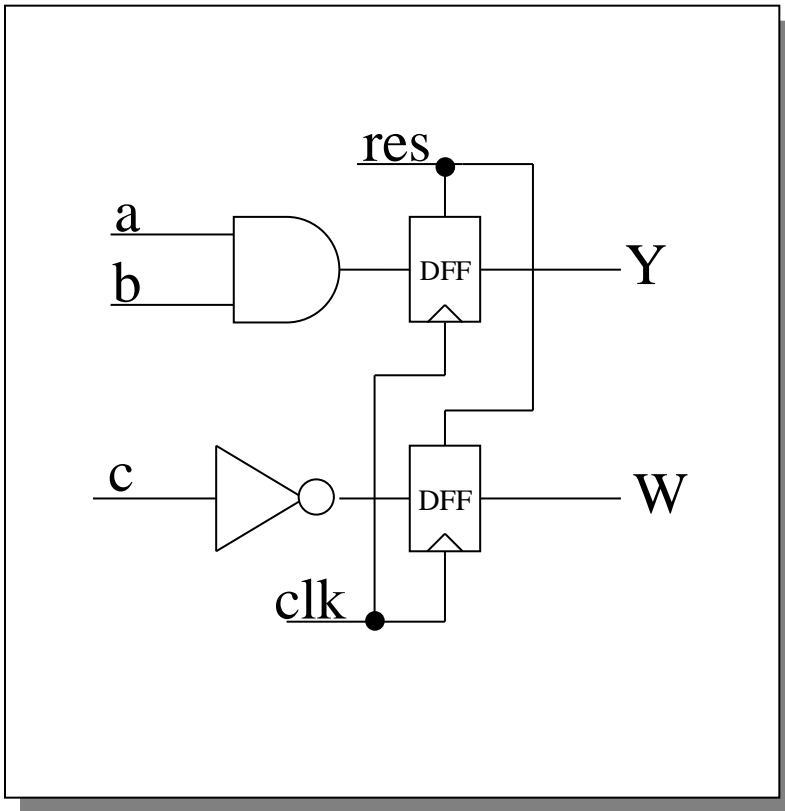
```
endmodule
```

← *Procedural statement*



How to implement t and jk flipflops?

# Example



```
always @(res or posedge clk) begin
    if (res) begin
        Y = 0;
        W = 0;
    end
    else begin
        Y = a & b;
        W = ~c;
    end
end
```

What is the corresponding circuit implementation?

# Conditional Statements - if ... else

---

- **if Statement**

- Format:

  - if** (condition)

  - procedural\_statement

  - else if** (condition)

  - procedural\_statement

  - else**

  - procedural\_statement

- Example:

  - if** (Clk)

  - Q = 0;

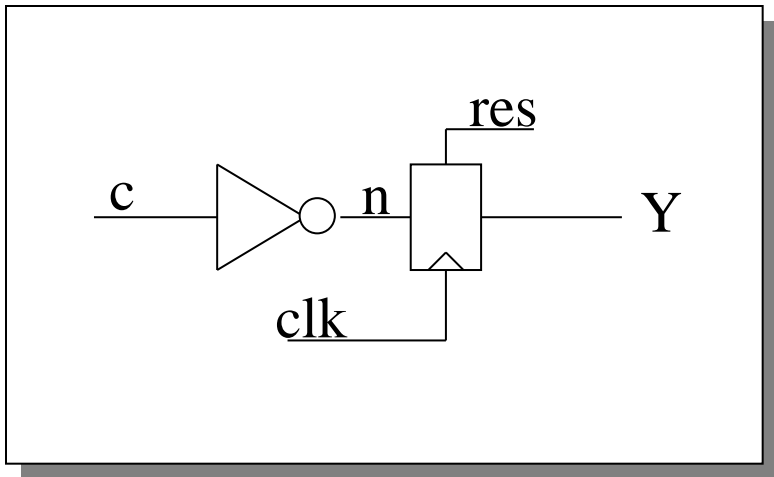
  - else**

  - Q = D;



# Mixed Model

Code that contains various both structure and behavioral styles



```
module simple(Y, c, clk, res);  
output Y;  
input c, clk, res;
```

```
reg Y;  
wire c, clk, res;  
wire n;
```

```
not(n, c); // gate-level
```

```
always @(res or posedge clk)  
    if (res)  
        Y = 0;  
    else  
        Y = n;  
endmodule
```

# Procedural Assignment with always

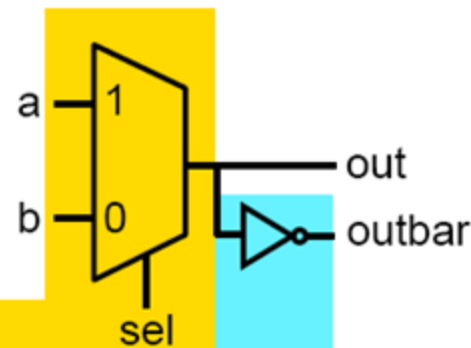
- Procedural and continuous assignments can co-exist within a module
- Procedural assignments update the *reg* value. The value remains unchanged till another procedural assignment updates the variable.

```
module mux_2_to_1(a, b, out,  
                  outbar, sel);  
  input a, b, sel;  
  output out, outbar;  
  reg out;
```

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

```
  assign outbar = ~out;
```

```
endmodule
```



*procedural  
description*

*continuous  
description*

# The Case Statement

---

- Case statements:

```
case (<expression>)
  <value1>: <statement>
  <value2>: <statement>
  .....
default: <statement>
endcase
```

- case* and *if* may be used interchangeably to implement conditional execution within *always* blocks
- case* is easier to read than a long string of *if* then *else* statements

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

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

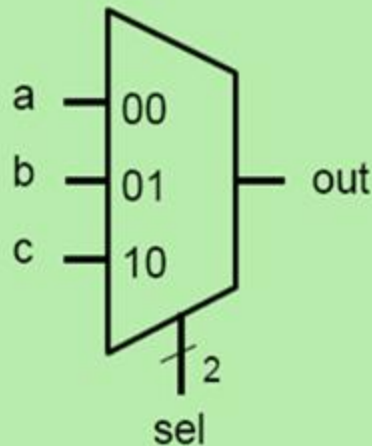
  assign outbar = ~out;
endmodule
```

```
module mux_2_to_1(a, b, out,
                  outbar, sel);
  input a, b, sel;
  output out, outbar;
  reg out;

  always @ (a or b or sel)
  begin
    case (sel)
      1'b1: out = a;
      1'b0: out = b;
    endcase
  end
endmodule
```

# Danger of Incomplete Specification

## Goal:



### **3-to-1 MUX**

('11' input is a don't-care)

## Proposed Verilog Code:

```
module maybe_mux_3to1(a, b, c,
                      sel, out);

    input [1:0] sel;
    input a,b,c;
    output out;
    reg out;

    always @(a or b or c or sel)
    begin
        case (sel)
            2'b00: out = a;
            2'b01: out = b;
            2'b10: out = c;
        endcase
    end
endmodule
```

*Is this a 3-to-1 multiplexer?*