# Verilog Coding Guidelines

Digital Circuit Lab

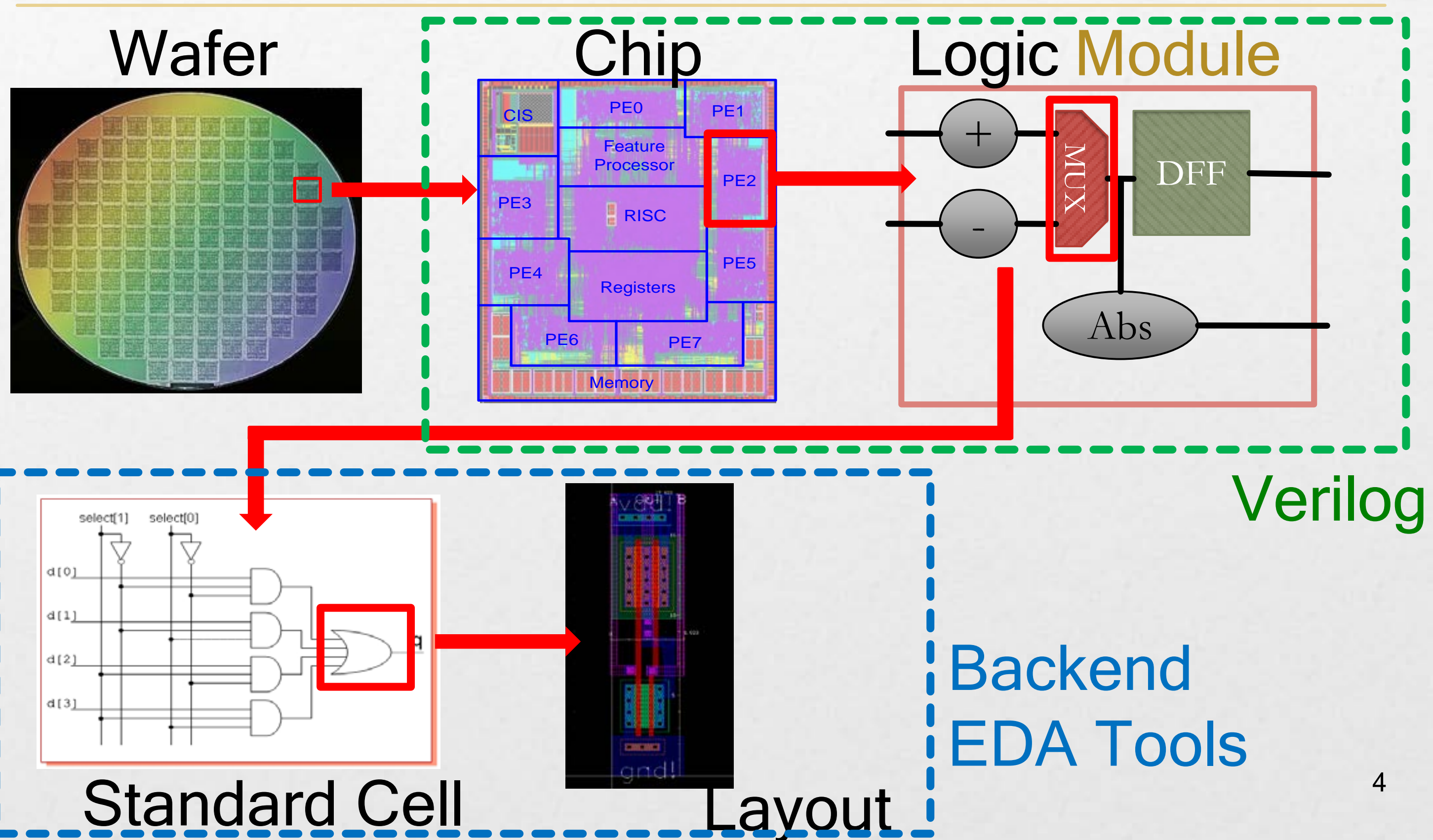
TA: Po-Chen Wu

# Outline

- Introduction to Verilog HDL
- Verilog Syntax
- Combinational and Sequential Logics
- Module Hierarchy
- Write Your Design
- Finite State Machine

# Introduction to Verilog HDL

# What is Verilog Doing…

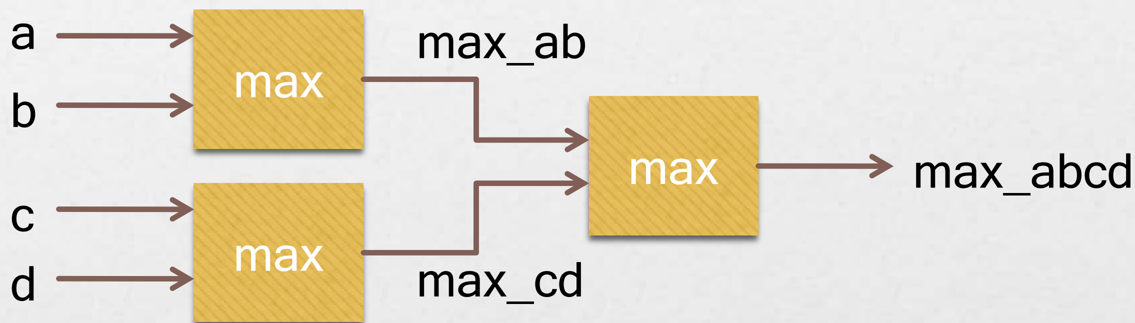


# Verilog HDL

- Verilog HDL
  - Hardware Description Language
  - Programming language
  - Describes a hardware design
- Other hardware description language
  - VHDL

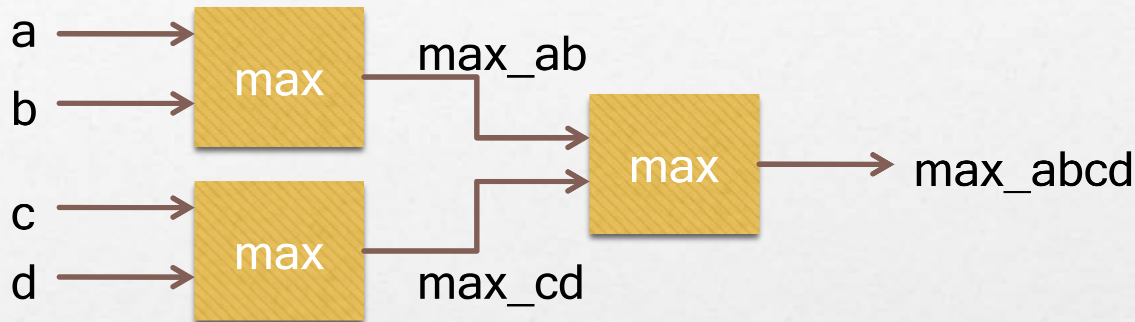
# Represent a Circuit (1/2)

- In software
  - max_abcd = max( max(a,b), max(c,d) );




- In verilog ??

# Represent a Circuit (2/2)




```
wire [3:0] a, b, c, d;
reg [3:0] max_ab; max_cd;
reg [3:0] max_abcd;
```

data declaration

```
always@(*) begin
  max_ab = (a > b)? a: b;
  max_cd = (c > d)? c: d;
  max_abcd = (max_ab > max_cd)? max_ab: max_cd;
end
```

logic behavior

# Verilog Syntax

# Blocking and Nonblocking Statements (1/2)

- Blocking Statements "="
  - A blocking statement must be executed before the execution of the statements that follow it in a sequential block.

- Nonblocking Statements "<="
  - Nonblocking statements allow you to schedule assignments without blocking the procedural flow.

# Blocking and Nonblocking Statements (2/2)

```
module block_nonblock();
  reg a, b, c, d, e, f;

  // Blocking assignments
  initial begin
    a = #10 1'b1; // The simulator assigns 1 to a at time 10
    b = #20 1'b0; // The simulator assigns 0 to b at time 30
    c = #40 1'b1; // The simulator assigns 1 to c at time 70
  end

  // Nonblocking assignments
  initial begin
    d <= #10 1'b1; // The simulator assigns 1 to d at time 10
    e <= #20 1'b0; // The simulator assigns 0 to e at time 20
    f <= #40 1'b1; // The simulator assigns 1 to f at time 40
  end
endmodule
```

# Data Types (1/3)

- wire
  - Used as inputs and outputs within an actual module declaration.
  - Must be driven by something, and cannot store a value without being driven.
  - Cannot be used as the left-hand side of an = or <= sign in an always@ block.
  - The only legal type on the left-hand side of an assign statement.
  - Only be used to model combinational logic.

# Data Types (2/3)

- reg
  - Can be connected to the input port (but not output port) of a module instantiation.
  - Can be used as outputs (but not input) within an actual module declaration.
  - The only legal type on the left-hand side of an always@ (or initial) block = or <= sign.
  - Can be used to create registers when used in conjunction with always@(posedge Clock) blocks.
  - Can be used to create both combinational and sequential logic.

# Data Types (3/3)

data declaration

```
wire [15:0] longdata; // 16-bit wire
wire shortvalue; // 1-bit wire
reg [3:0] areg; //4-bit reg
```

wire

```
assign longdata = areg + 88;
```

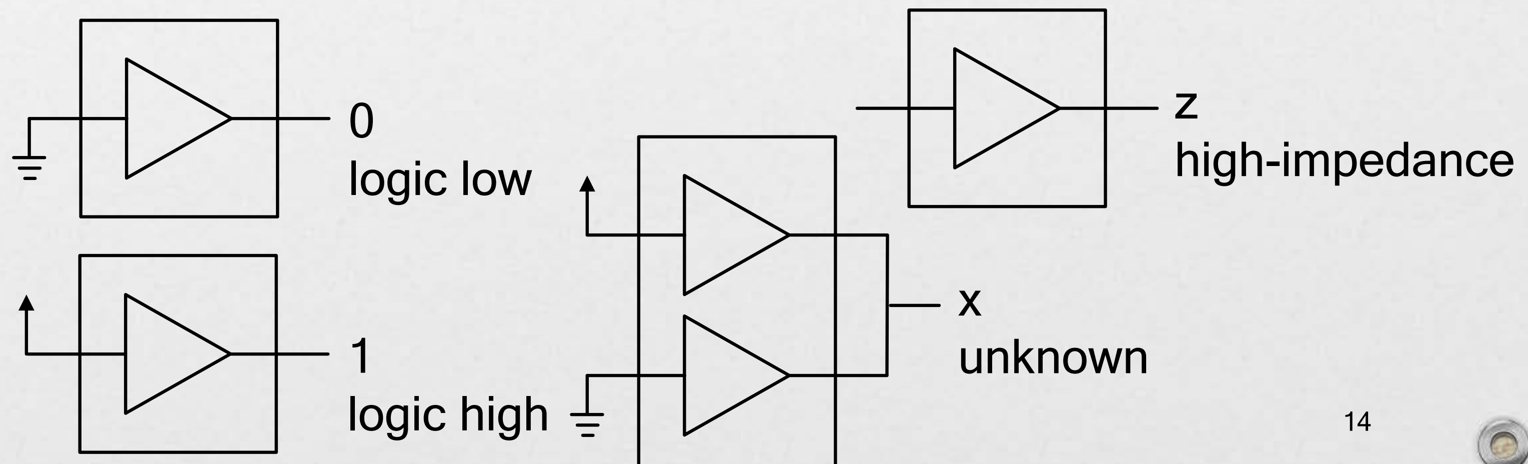
reg

```
always@(*) begin
  if(shortvalue == 1'b1)
    areg = shortvalue + 3;
  else
    areg = shortvalue + 7;
end
```

logic behavior

# Value Set

- 0 - logic zero, or false condition
- 1 - logic one, or true condition
- z - high-impedance state
- x - unknown logic value - unrealistic value in design



# Numbers

- binary('b), decimal('d), hexadecimal('h), octal('o)
- Format
  - <number>: reg data = 127;
  - '<base><number>: reg data = 'd127;
  - <width>'<base><number> → complete format

```
             659     //  decimal
           'o7460    //  octal
          4'b1001    //  4-bit binary
reg data = 3'b01x    //  3-bit with unknown
          16'hz      //  16-bit high impedance
          -8'd6      //  two's complement of 6
           8'd-6     //  illegal syntax
           4af       //  illegal (requires 'h)
```

# Case-Sensitive Control

- Suspends subsequent statement execution until any of the specified list of events occurs
  - Support event driven simulation
- Sensitivity list
  - always@(… or … or … or…)
  - always@(a or b or c )
  - always@(*)
    - verilog 2001, automatic list generation

# Operators (1/6)

- Arithmetic operators
  - +, -, *, /, %
- Bitwise operators
  - Perform the operation one bit of a operand and its equivalent bit on the other operand to calculate one bit for the result
  - ~, &, |, ^, ~^

| operator | operation |
| --- | --- |
| + | arithmetic addition |
| - | arithmetic substraction |
| * | arithmetic multiplication |
| / | arithmetic division |
| % | arithmetic modulus |

Arithmetic operators

| operator | operation |
| --- | --- |
| ~ | bit-wise NOT |
| & | bit-wise AND |
| \| | bit-wise OR |
| ^ | bit-wise XOR |
| ~^ | bit-wise XNOR |

Bitwise operators

# Operators (2/6)

- Unary reduction operators
  - Perform the operation on each bit of the operand and get a one-bit result
  - &, ~&, |, ~|, ^, ~^

| operator | operation |
|---|---|
| & | unary reduction AND |
| ~& | unary reduction NAND |
| \| | unary reduction OR |
| ~\| | unary reduction NOR |
| ^ | unary reduction XOR |
| ~^ | unary reduction XNOR |

Unary reduction operators

&4'b1110→0
&4'b1111→1

Unary reduction AND

|4'b0000→0
|4'b0001→1

Unary reduction OR

^4'b1111→0
^4'b1110→1

Unary reduction XOR

# Operators (3/6)

- Logical operators operate with logic values
  - Equality operators
    - Logical equality
    - ==, !=
    - Case equality
    - ===, !==
- Logical negation
  - !
- Logical
  - &&, ||

example

```
!4'b0100 → 0
!4'b0000 → 1
!4'b00z0 → x
```

**Operators (cont.)**

- logical operators

| operator | operation |
|---|---|
| ! | logical NOT |
| && | logical AND |
| \|\| | logical OR |
| == | logical equality |
| != | logical inequality |
| === | logical identity |
| !== | the inverse of === |

– logical operator operate with logic values. ( non-zero is true, and zero value is false).

if( sel == 4'h03) ……. else ……..

Logical operators

# Operators (4/6)

## Operators (cont.)

- Example

opa = 0010    opb = 1100    opc = 0000

unary reduction

& opa = 0

0 & 0 & 1 & 0 = 0

logical operation

opa && opc = 0

opa = 0010 → true
opc = 0000 → false
true && false = false

bit-wise operation

opa & opb = 0000

```
  0000
& 1100
------
  0000
```

bit-wise operation

~ opa = 1101

logical operation

opa && opb = 1

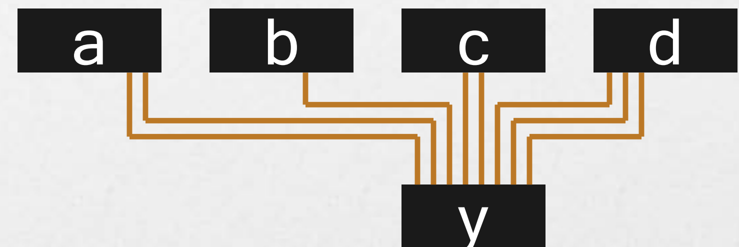
opa = 0010 → true
opb = 1100 → true
true && true = true

logical operation

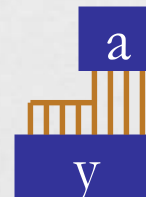
! opa = 0

# Operators (5/6)

- Concatenation operator
  - {}
  - Join bits from two or more expressions together
  - Very convenient
  - Multiple layers
  - {{},{}…}
- Replication operator
  - {n{}}




y = {a[1:0], b[2], c[4,3], d[7:5]}




y = {{4{a[3]}},a}

# Operators (6/6)

- Shift operators
  - <<, >>
- Relational operators
  - <, <=, >=, >

```
if(a <= b) d = 0;
else d = 1;
```

- Conditional operator
  - ? :

```
d = (a<=b) ? 0 : 1
```

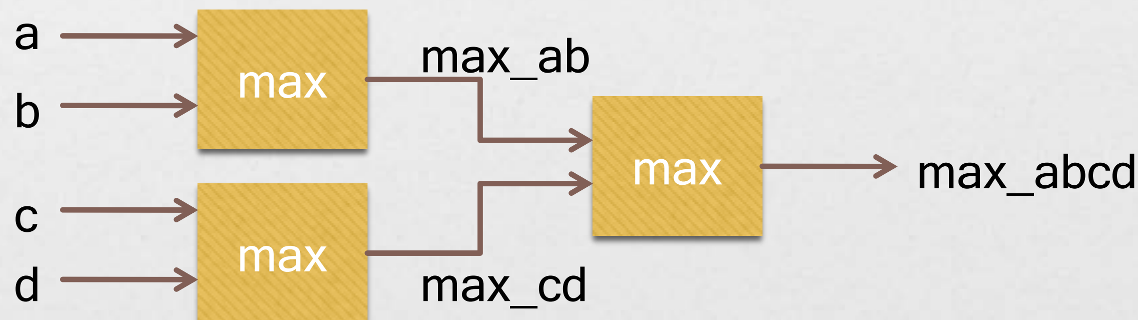
➢ Operator precedence

<table>
<tr><th colspan="2">Operator Precedence Rules</th></tr>
<tr><td>+ - ! ~ (unary)<br>* / %<br>+ - (binary)<br><< >><br>< <= > >=<br>== != === !==<br>&<br>^ ^~<br>|<br>&&<br>||<br>?: (ternary operator)</td><td>highest precedence<br>↓<br>lowest precedence</td></tr>
</table>

# Combinational and Sequential Logics

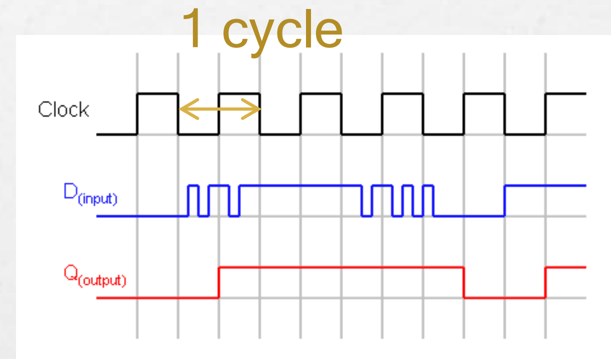
# Two Types of Logics (1/2)

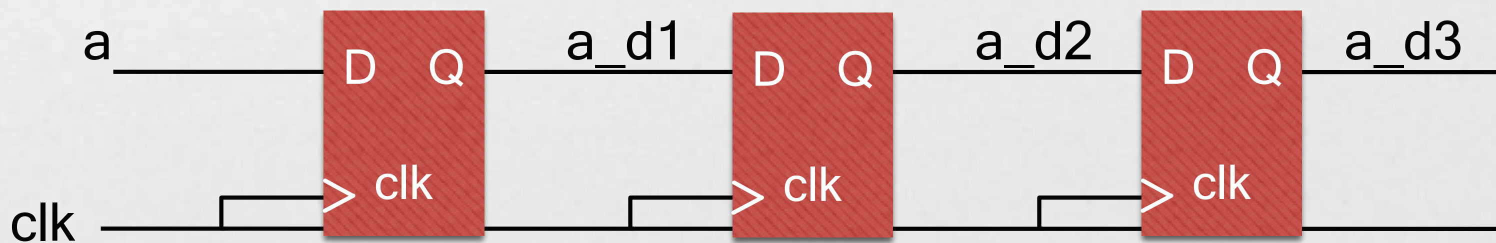
- Combinational Logics
  - data-in → data-out
  - instant response
  - fixed arrangement



# Two Types of Logics (2/2)

- Sequential Logics
  - always and only update at clock edges
  - posedge / negedge
  - memory



# Case-Sensitive Control (1/2)

- register in sequential changes only at clock edges

```
always@(posedge clk) begin
    ......
end
```

```
always@(negedge clk) begin
    ......
end
```

- with reset signal

```
always @(posedge clk or negedge rst_n) begin
    ......
end
```

# Case-Sensitive Control (2/2)

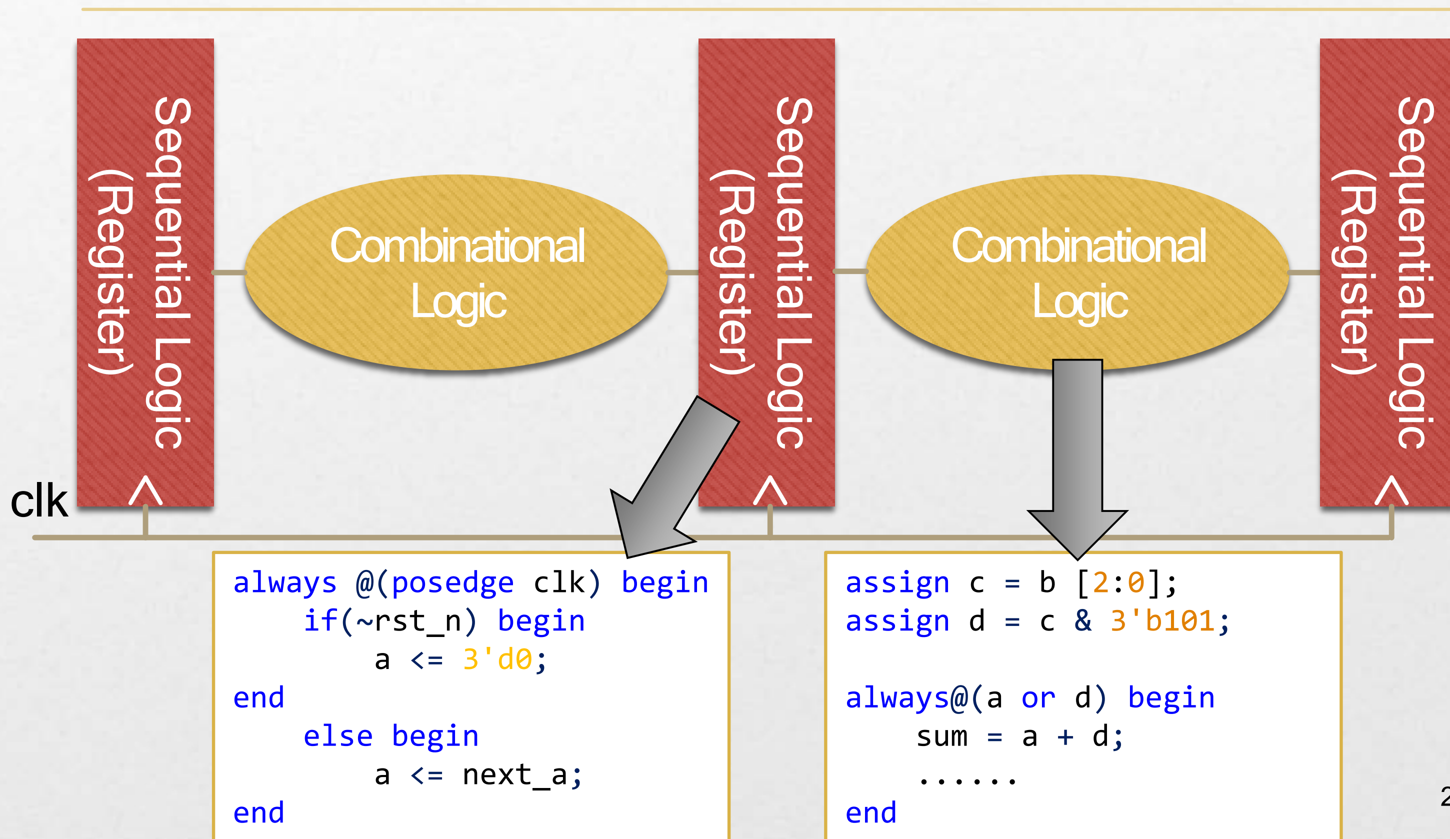
- Sequential Logics
  - always and only update at clock edges
  - posedge / negedge

```
reg [3:0] a, b; // declaration

always @(posedge clk or negedge rst_n) begin
    if(rst_n ==1'b0) begin //registers
        a <= 4'd0;
        b <= 4'd0;
    end
    else begin
        a <= next_a;
        b <= next_b;
end
```

# The Use of Sequential Circuits?

- Temporal storage (memory)
- Split long computation lines
  - timing issue
  - divide circuits into independent stages
    - work at the same time !!
- Combinational logics handle the computation
- Sequential logics store inter-stage temporal data

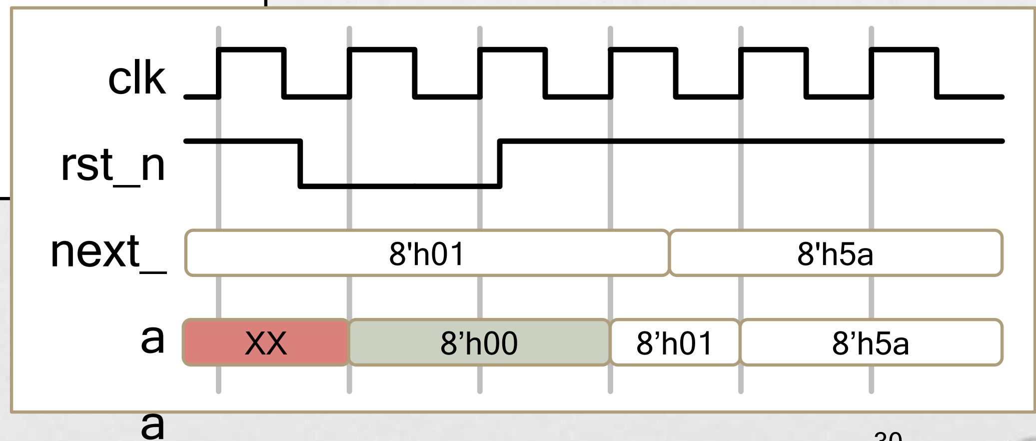
# Sequential and Combinational Logics



# Sequential Circuit (1/3)

- Synchronous reset

```
always @(posedge clk) begin
    if(~rst_n) begin
        a <= 8'd0;
    end
    else begin
        a <= next_a;
end
```

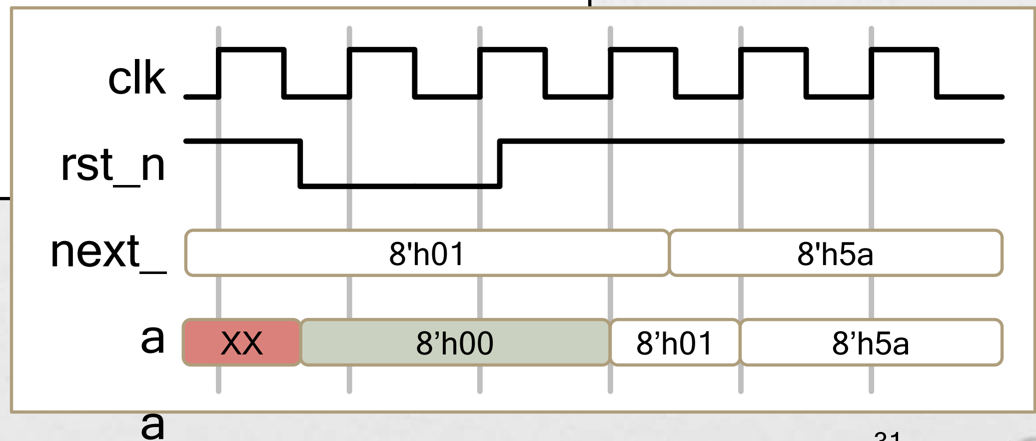



a

# Sequential Circuit (2/3)

- Asynchronous reset

```
always @(posedge clk or negedge rst_n)  begin
    if(~rst_n) begin
        a <= 8'd0;
    end
    else begin
        a <= next_a;
end
```




a

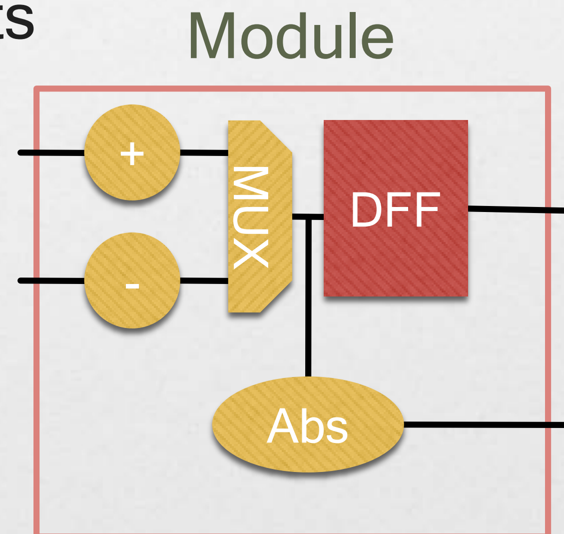
# Sequential Circuit (3/3)

| | | |
|---|---|---|
| Synchronous reset | Pros | • Easy to synthesize, just another synchronous input to the design |
| | Cons | • Require a free-running clock, especially at power-up, for reset to occur |
| Asynchronous reset<br><br>Prefer | Pros | • Does not require a free-running clock<br>• Uses separate input on FF, so it does not affect FF data timing |
| | Cons | • Harder to implement, usually a tree of buffers is inserted at P&R<br>• Makes static timing analysis and cycle-based simulation more difficult, and can make the automatic insertion of test structures more difficult |

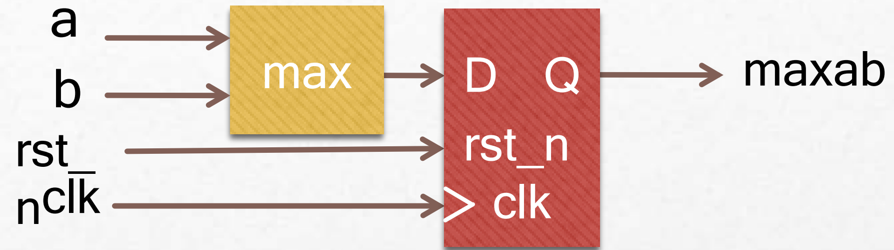
# Module Hierarchy

# What is a Module?

- Group circuits into meaningful building blocks
- Combine highly-related circuits
- Leave simple i/o interface
- Easier to reuse / maintain



# A Module




```
module maxab(clk,rst_n,a,b,maxab);
input clk;
input rst_n;
input [3:0] a;
input [3:0] b;
output [3:0] maxab;

reg [3:0] maxab;
wire [3:0] next_maxab;

assign next_maxab = (a>b)? a: b;
```

module port definition

wire, reg declaration

combinational logics

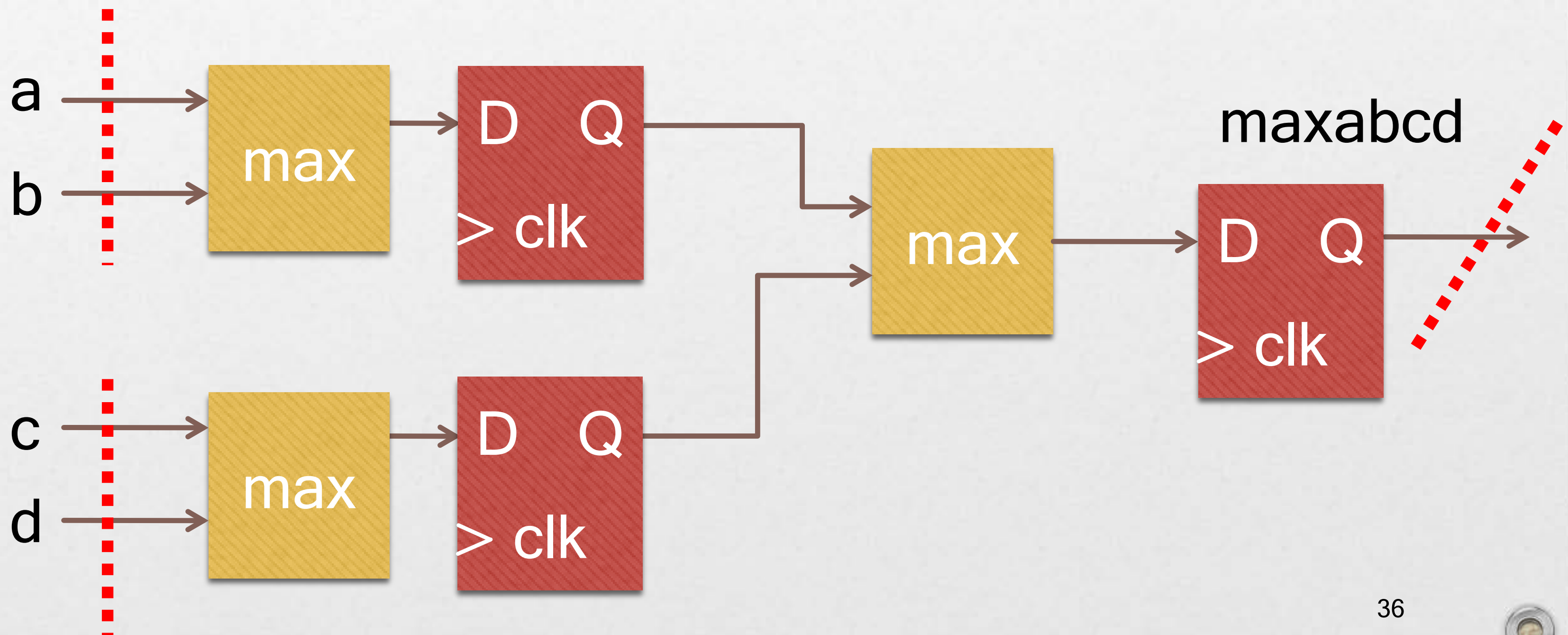
```
always@(posedge clk or negedge
rst_n) begin
    if (rst_n == 1'b0) begin
        maxab <= 4'd0;
    end
    else begin
        maxab <= next_maxab;
    end
end

endmodule
```

sequential logics

ending your module!

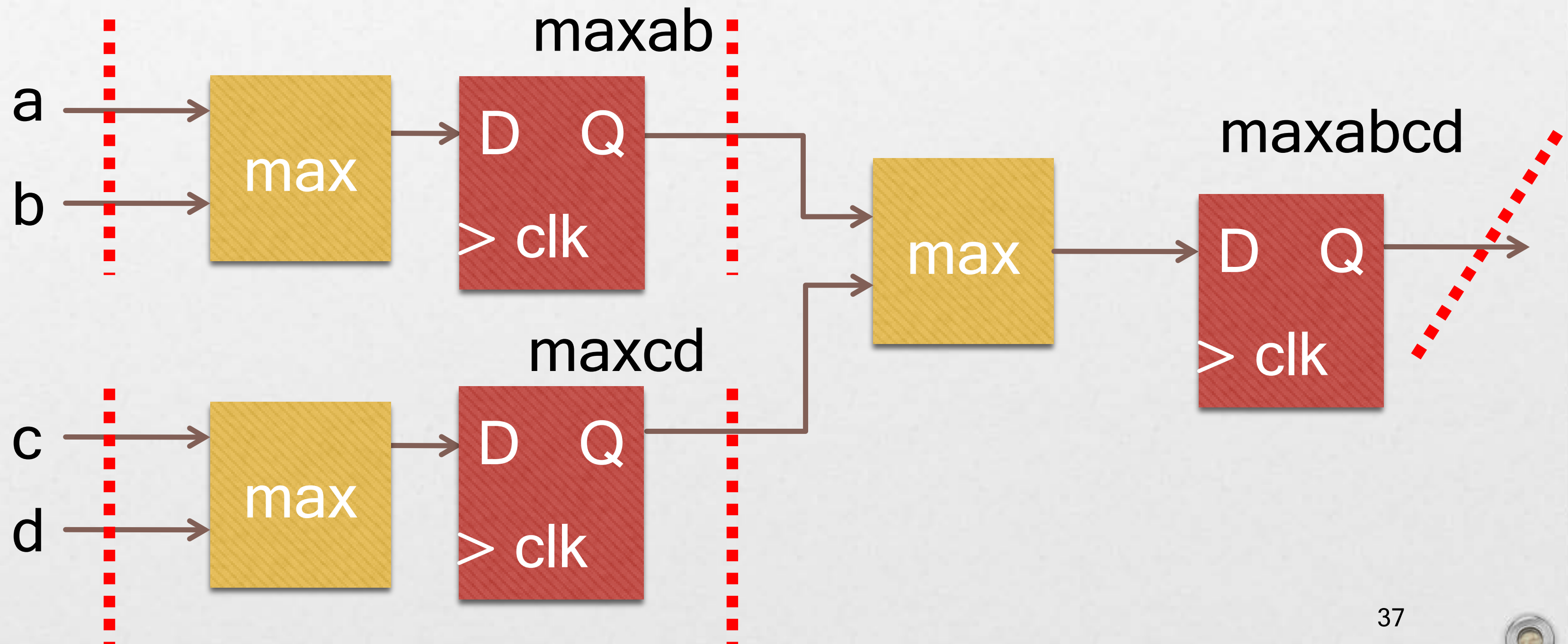
# Connection Between Modules (1/2)

- Where you "cut" your design.



# Connection Between Modules
(2/2)

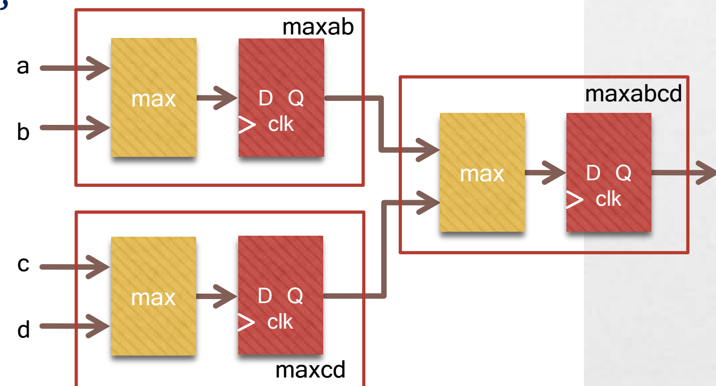
- Where you "cut" your design.



# Module Instantiation

- Build a module by smaller modules

```
module maxabcd(clk,rst_n,a,b,c,d,maxabcd);
input clk;
input rst_n;
input [3:0] a, b, c, d;
output [3:0] maxabcd;

wire [3:0] maxab, maxcd;

maxab m1(.clk(clk), .rst_n(rst_n), .a(a), .b(b), .maxab(maxab));
maxab m2(.clk(clk), .rst_n(rst_n), .a(c), .b(d), .maxab(maxcd));
maxab m3(.clk(clk), .rst_n(rst_n), .a(maxab), .b(maxcd), .maxab(maxabcd));

endmodule
```



# Write Your Design

# Use Parameters (1/2)

- Build a module by smaller modules

```
`define INST_WakeUp 0
`define INST_GoToSleep 1
`define BUS_WIDTH 64
input [`BUS_WIDTH-1:0] databus;
case (instruction)
    `INST_WakeUp:
    …
    `INST_GoToSleep:
    …
endcase
```

# Use Parameters (2/2)

- Use parameter for reusable modules

```
parameter [4:0] FA_BitSize = 8;
reg [FA_BitSize-1:0] = datain;
```

- Use localparam for inner-module variables

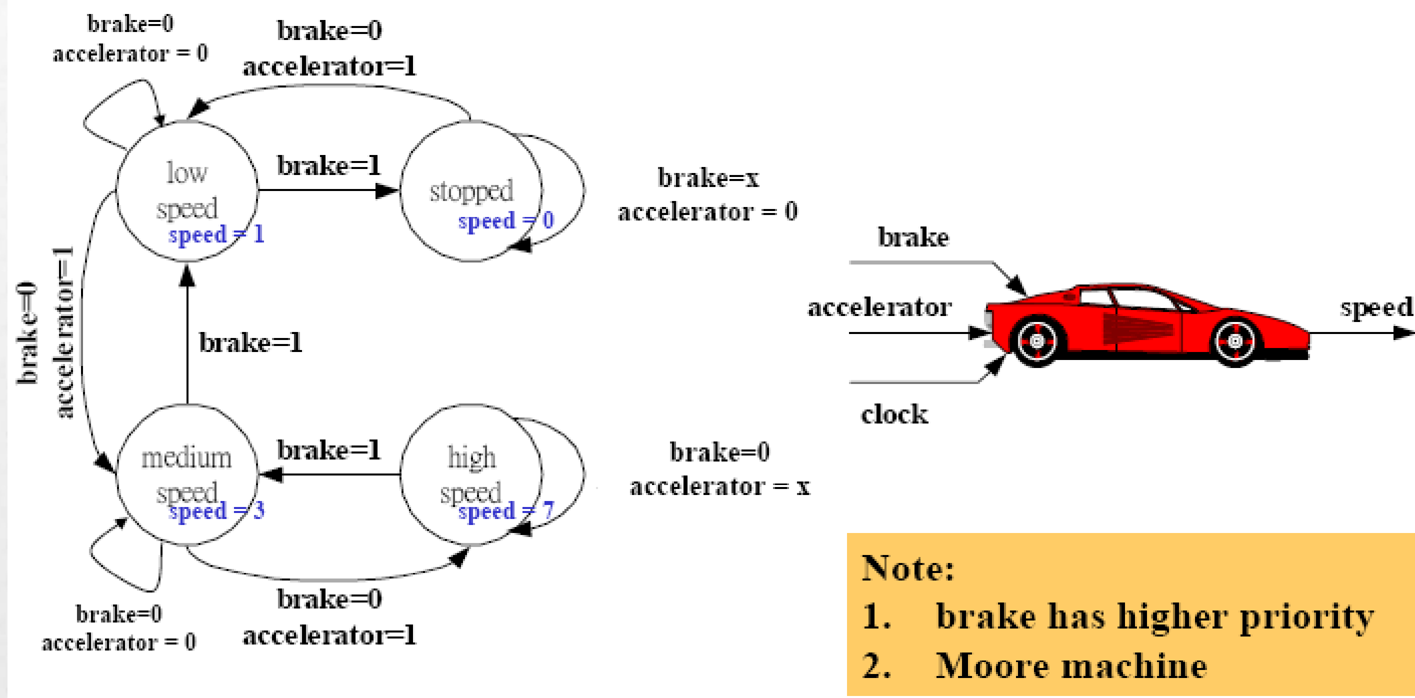
```
localparam FSM_StateSize = 5;
localparam [FSM_StateSize-1:0] FSM_Idle = 5'd0;
```

# Finite State Machine

# Finite State Machine (1/2)

- Synchronous (i.e. clocked) finite state machines (FSMs) have widespread application in digital systems
  - Controllers in computational units and processors.
- Synchronous FSMs are characterized by
  - A finite number of states
  - Clock-driven state transitions.

# Finite State Machine (2/2)



# Elements of FSM (1/2)

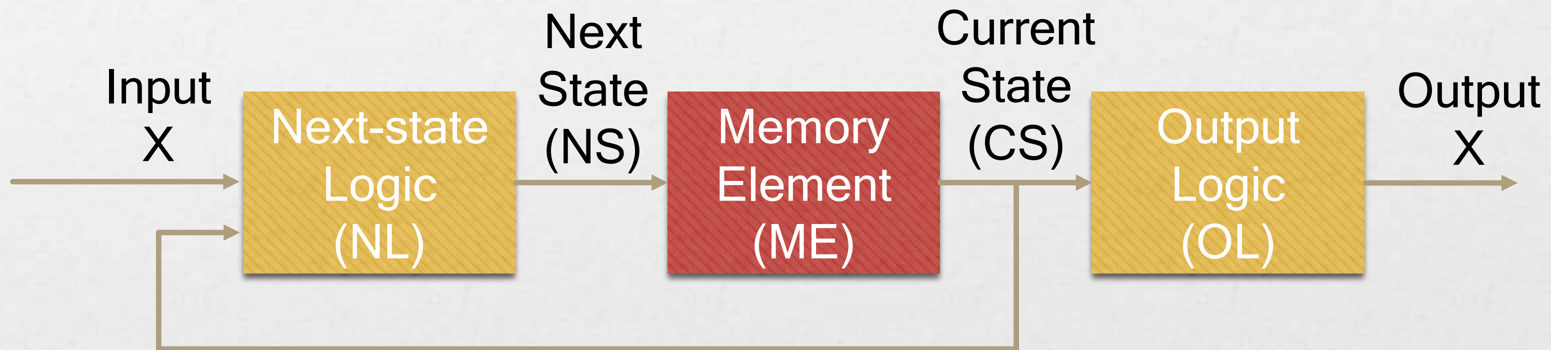
- Memory elements (ME)
  - Memorize current state (CS)
  - Usually consist of FF or latch
  - N-bit FF have $2^N$ possible states
- Next-state logic (NL)
  - Combinational logic
  - Produce next state
  - Based on current state (CS) and input (X)

# Elements of FSM (2/2)

- Output logic (OL)
  - Combinational logic
  - Produce outputs (Z)
  - Based on current state (Moore)
  - Based on current state and input (Mealy)

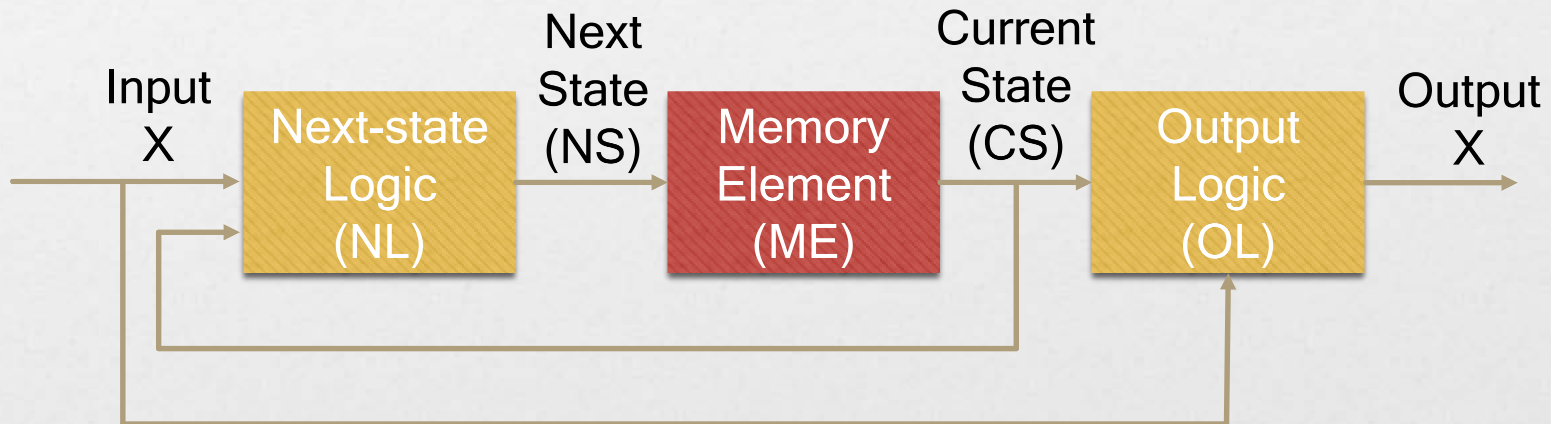
# Moore Machine

- Output is function of CS only
  - Not function of inputs



# Mealy Machine

- Output is function of both
  - Input and current state



# Modeling FSM in Verilog

- Sequential circuits
  - Memory elements of current state (CS)
- Combinational circuits
  - Next-state logic (NL)
  - Output logic (OL)

# The End.

Any question?

# Reference


1. "Verilog_Coding_Guideline_Basic" by members of DSP/IC Design Lab
2. http://inst.eecs.berkeley.edu/~cs150/Documents/Nets.pdf by Chris Fletcher, UC Berkeley
3. CIC training course： "Verilog_9807.pdf"
4. http://www.asic-world.com/