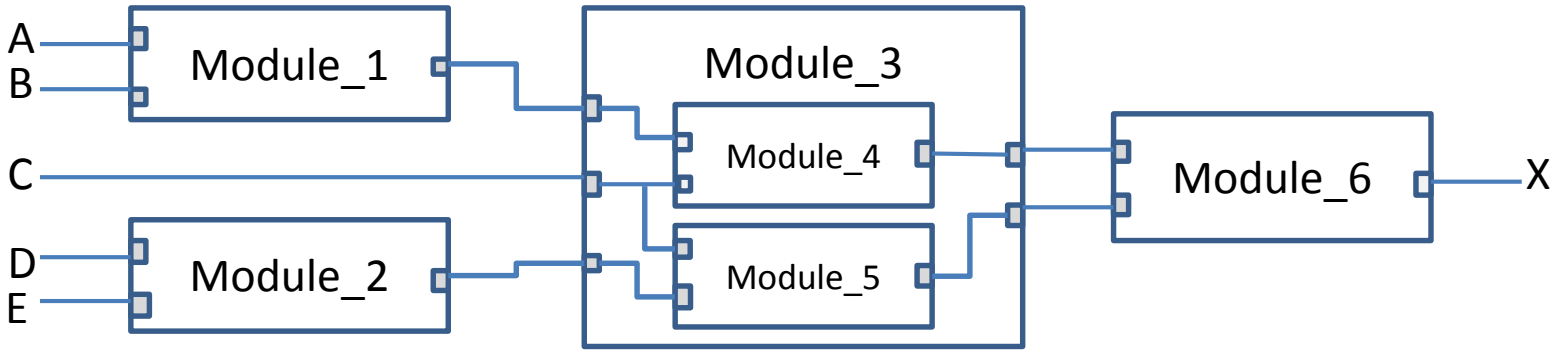
NANYANG TECHNOLOGICAL UNIVERSITY

CX1005
Digital Logic

Verilog Recap

Verilog Hardware Description Language

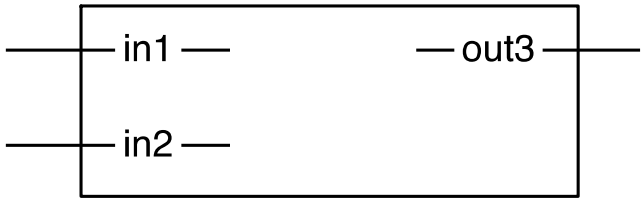
- Language with special constructs for describing hardware
- The most important rule when thinking about Verilog design, is to think *in hardware*
- In Verilog, designs are broken down into ***modules***
- Modules can contain code to describe hardware and also instances of other modules



Ports, Wires, Reg

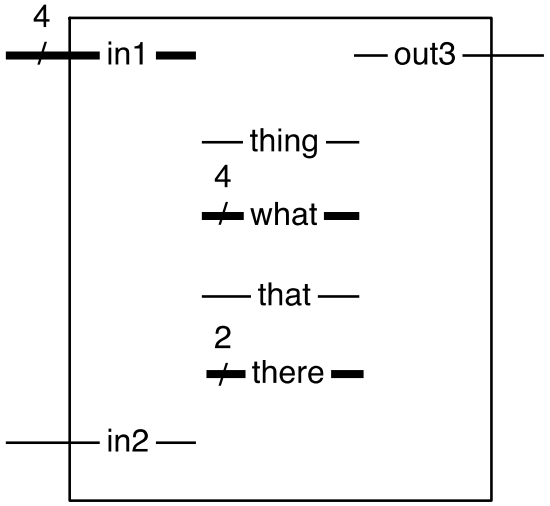
- Ports act exactly like wires:
  - Can "read" the value of an input or output port
  - But can only "assign"/"write" to an output port

```
module simple (input in1, in2,
               output out3);
endmodule
```



- Regs are exactly the same as wires, but can only be assigned to from within always blocks
  - Can be "read" anywhere

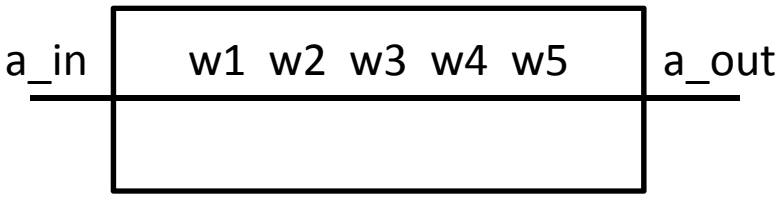
```
module simple (input [3:0] in1,
               input in2,
               output out3);
    wire thing;
    wire [3:0] what;
    reg that;
    reg [1:0] there;
endmodule
```



Ports, Wires, Regs

- Internal wires are just anchor points to which you can connect logic:
- What will this do?

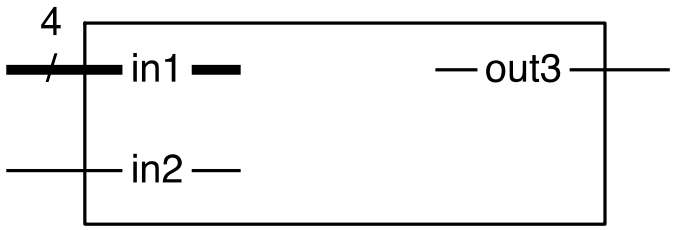
```
module nothing (input a_in,
                output a_out);

wire w1, w2, w3, w4, w5;

assign w1 = a_in;
assign w2 = w1;
assign w3 = w2;
assign w4 = w3;
assign w5 = w4;
assign a_out = w5;

endmodule
```



Ports, Wires, Regs

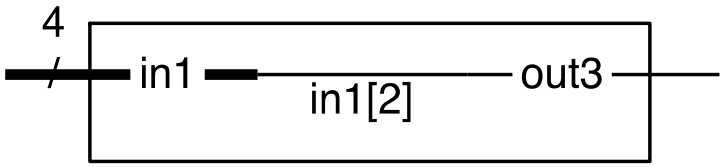
- Any port, wire, or reg can be declared multi-bit by preceding its name with an index range:

```
module simple (input [3:0] in1,
               input in2,
               output out3);
endmodule
```



- We can slice sections or single bits of any port, wire, or reg and use them as we do for single wires:

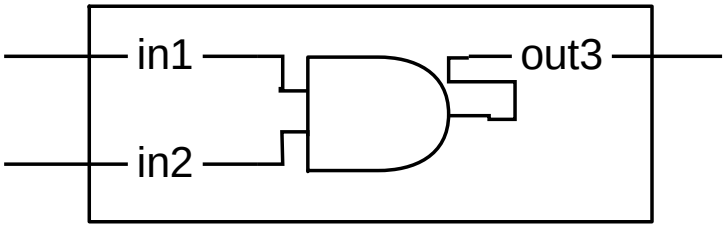
```
module simple (input [3:0] in1,
               output out3);
    assign out3 = in1[2];
endmodule
```



Instantiating Gate-Level Primitives

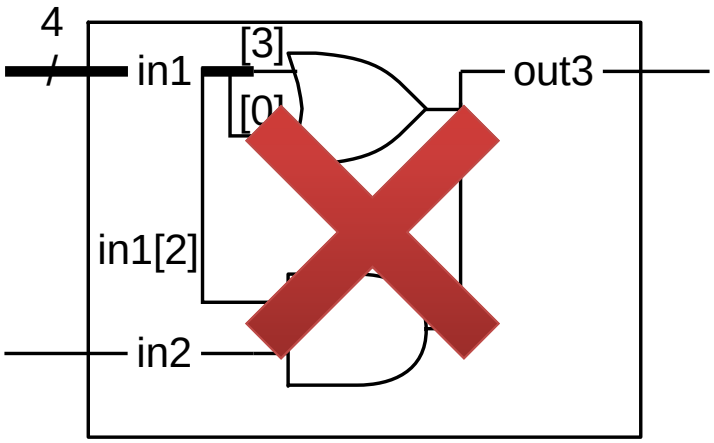
- We can connect ports/wires/reg by instantiating primitives or modules:

```
module simple (input in1, in2,
               output out3);

   and a1 (out3, in1, in2);

endmodule
```



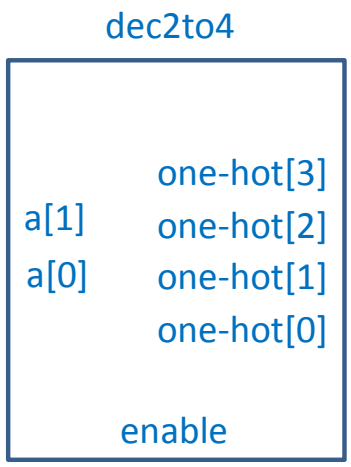
- You never want to assign, by any means, to a wire, port, or reg multiple times:

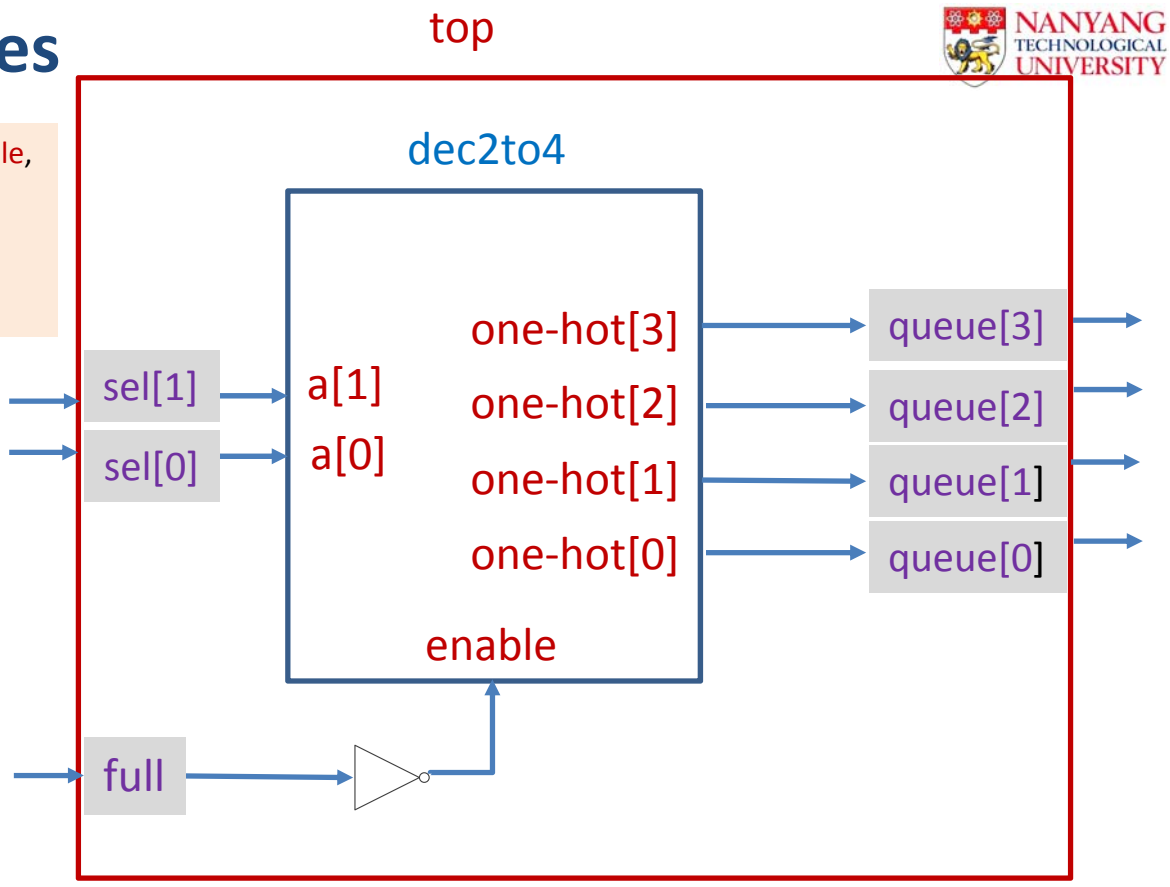
```
module simple (input [3:0] in1,
               input in2,
               output out3);
   and (out3, in1[2], in2);
   or  (out3, in1[3], in1[0]);
endmodule
```



Instantiating Modules

```
module dec2to4 (input [1:0] a,  input enable,
                output [3:0] one-hot);

...
endmodule
```



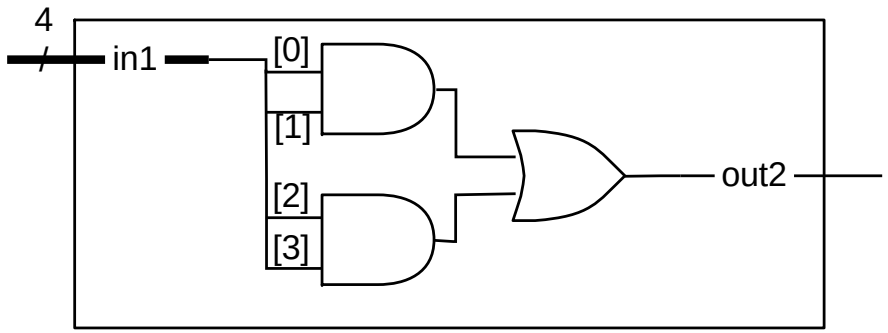



```
module top (input [1:0]  sel,
            input        full,
            output [3:0] queue);

    dec2to4 (.a(sel), .enable(~full), .one-hot(queue));

endmodule
```

Assign Statements

- These are the same:

```
module simple (input [3:0] in1,
               output out2);

  and (w1, in1[0], in1[1]);
  and (w2, in1[2], in1[3]);
  or  (out2, w1, w2);

endmodule
```

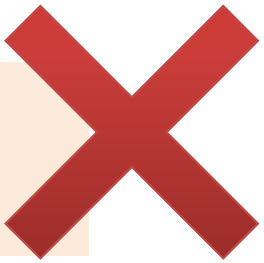



```
module simple (input [3:0] in1,
               output out2);

  assign out2 = ((in1[0] & in1[1]) |
                 (in1[2] & in1[3]));

endmodule
```

Assign Statements

- Assign statements let us assign the result of an expression to a signal
- Allow us to use many more operators
  - Bitwise or Logical
  - Comparisons and equality
  - Arithmetic
- This is always preferred to instantiating gates or simple low-level modules
- Remember, you cannot assign to regs!
  - Reg can only be assigned within an always block

```
reg temp;

assign temp = 1'b0;
```



~	(bitwise NOT)
&	(bitwise AND)
|	(bitwise OR)
^	(bitwise XOR)
~^ or ^~	(bitwise XNOR)

!	(logical NOT)
&&	(logical AND)
||	(logical OR)

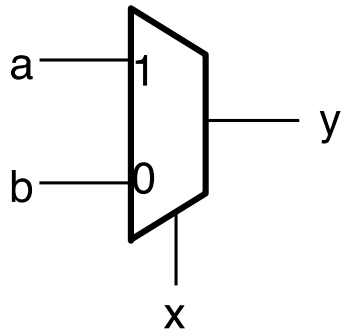
<	(less than)
<=	(less than or equal to)
>	(greater than)
>=	(greater than or equal to)
==	(equal to)
!=	(not equal to)

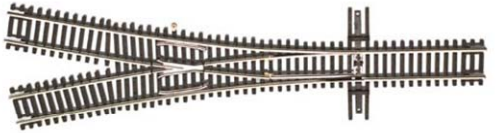
+	(addition)
-	(subtraction)
*	(multiplication)
/	(division)
%	(modulus)

Conditional Assignment

- One other trick with the assign statement is conditional assignment
- Creates a multiplexer, like an if/else statement:

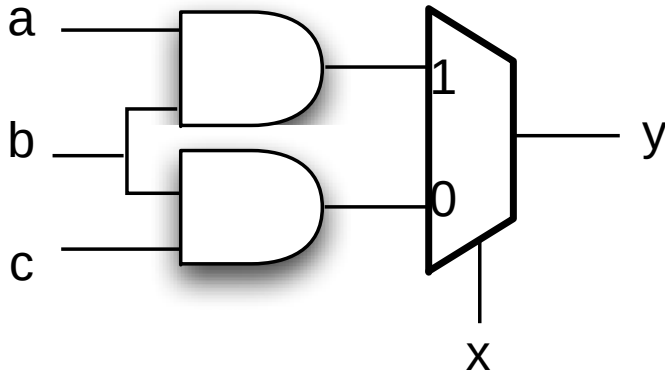
```
assign y = x ? a : b;
```





- Any expression in the body of the conditional assignment is also generated as hardware

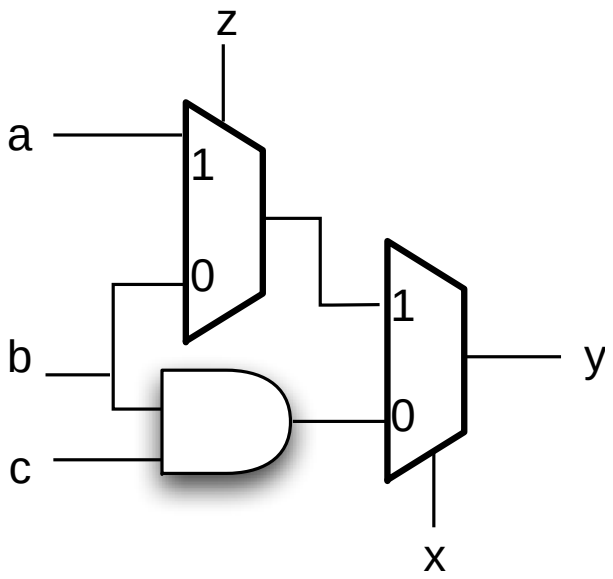
```
assign y = x ? a & b : b & c;
```



Conditional Assignment

- Those expressions can themselves be conditional assignments (Brackets make this easier to read)

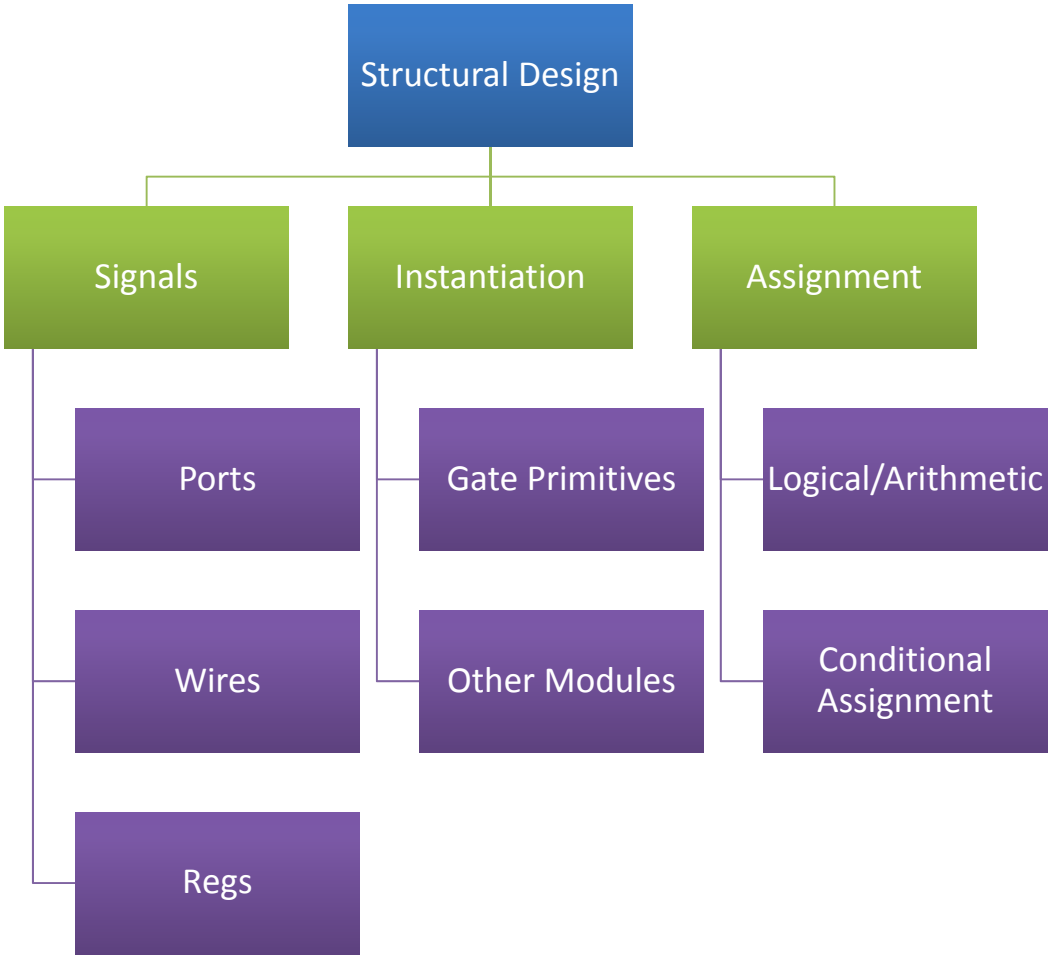
```
assign y = x ? (z ? a : b):
               b & c;
```



- Everything in an assign statement gets turned into a mini combinational circuit
- The signal being assigned to is the output of this mini circuit
- The tools take care of how best to design the circuit

- Conditional assignment simply adds a multiplexer to select which expression to finally route to the assigned signal

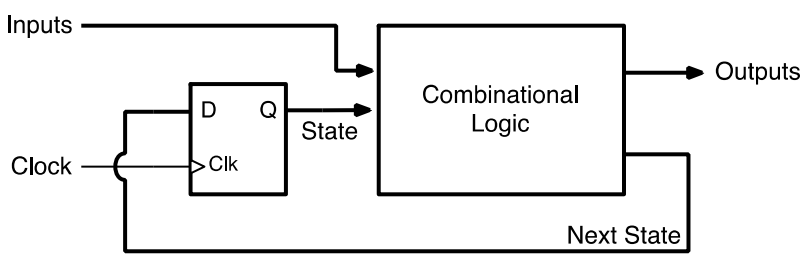
Overview of Structural Design



Behavioral Design

- In behavioral design, we raise the level of our description and rely on the synthesis tools to do more
- We use the always block to describe circuits behaviorally
- We can design both combinational and sequential circuits using always blocks
- Always block is effectively a super-mega assign statement
- There is no other way to build synchronous blocks

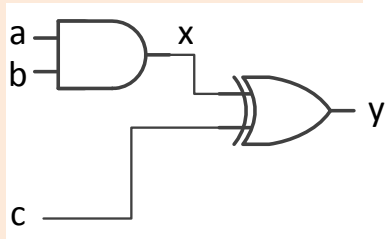
```
always @ *
begin
   a = … ;
   b = … ;
end

assign c = … ;

always @ (posedge clk)
begin
   y <= … ;
   x <= … ;
end

assign z = … ;
```



Combinational Always Blocks

- Has a list of signals in its sensitivity list, or better: *
- Can assign to one or more signals
- Must only assign to reg signals
- We use a blocking assignment (=)
- **Statement order** within the always block does **matter**
- Each assignment results in a combinational circuit
- If and case statements are very useful
- All signals to which it assigns must be assigned in all possible cases

```
module gate2 (input a, b, c,
              output reg y);
   reg x;

   always @*
   begin
     x = a & b;
     y = x ^ c;
   end

endmodule
```

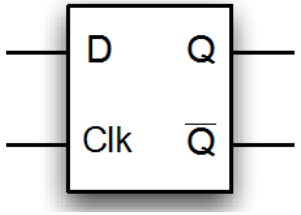



```
module mux4 (output reg q,
             input [3:0] d,
             input [1:0] sel);
   always @* begin
      case (sel)
         2'b00 : q = d[0];
         2'b01 : q = d[1];
         2'b10 : q = d[2];
         2'b11 : q = d[3];
      endcase
   end
endmodule
```

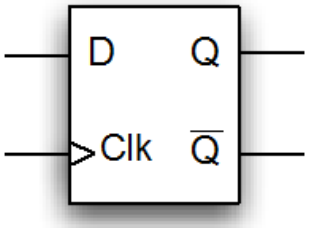
Sequential Circuits

- Sequential circuits are any circuits that have the concept of state: include latches, flip-flops, etc.
- Latches are level-sensitive, i.e., their outputs change when the control (enable) input is high
- A flip-flop/register is edge-sensitive, i.e. their outputs change at the rising/falling edge of the control (enable)
- Modern design is almost always synchronous (one shared clock)

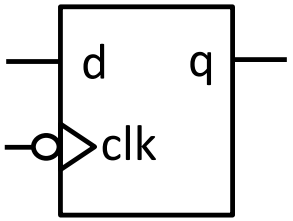
D Latch

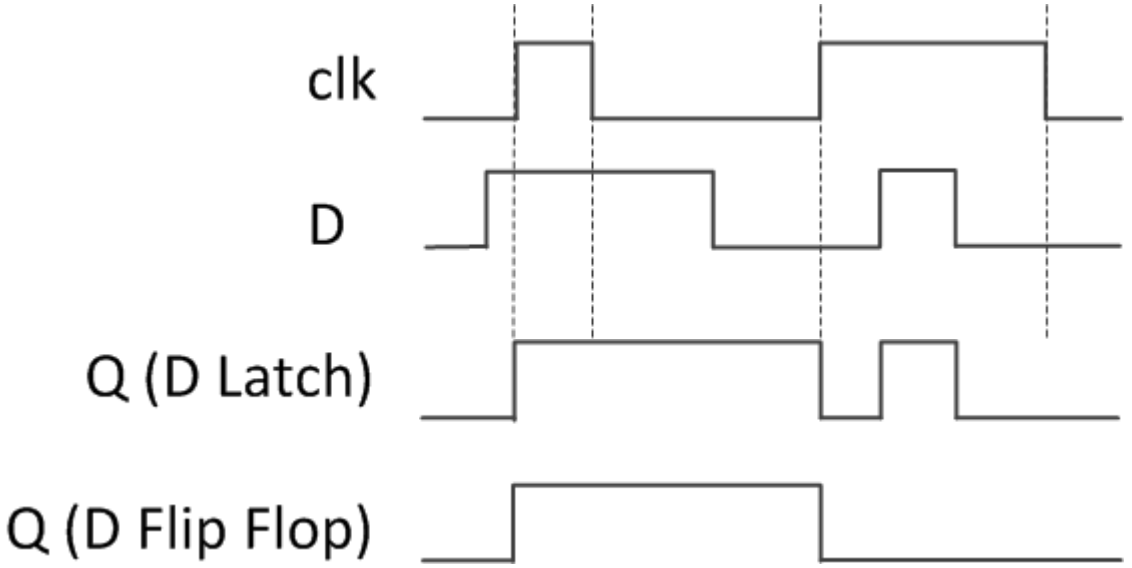


Positive edge triggered Flip-Flop



Negative edge triggered Flip-Flop

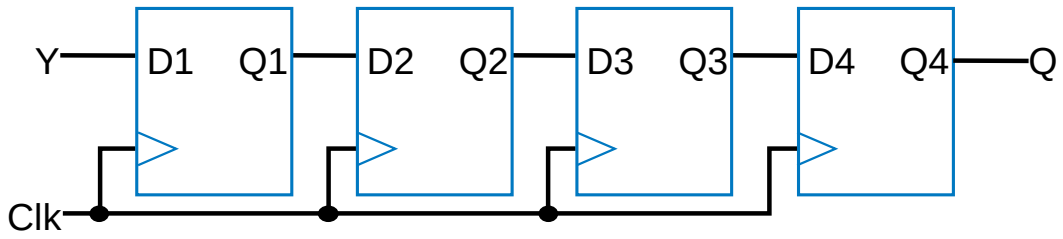




Synchronous Always Block

- Use ***posedge*** *clk* in its sensitivity list
- Can assign to one or more signals
- Must only assign to reg signals
- Each assigned signal becomes a register connected to any logic given in the expressions
- We use non-blocking assignment (<=)
- **Statement order does not matter** with non-blocking assignments
- We should ensure all registers respond to a reset

```
reg q1, q2, q3, q;

always@(posedge clk)
begin
  if (rst) begin
    q1 <= 1'b0;
    q2 <= 1'b0;
    q3 <= 1'b0;
    q  <= 1'b0;
  end
  else begin
    q1 <= y;
    q2 <= q1;
    q3 <= q2;
    q  <= q3;
  end
end
```

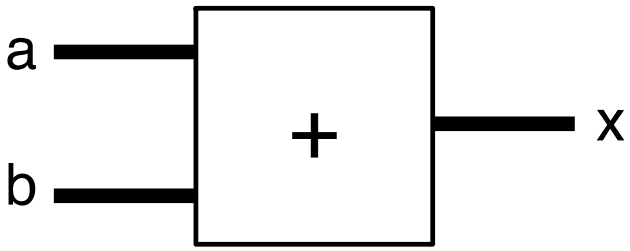


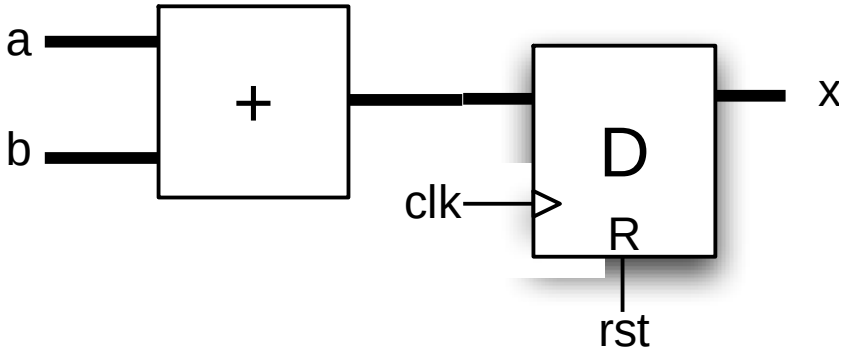
Combinational and Synchronous

- Identical statements produce the same circuit, but with a synchronous always, we get a register on the end:

```
always @ *
begin
    x = a + b;
end
```

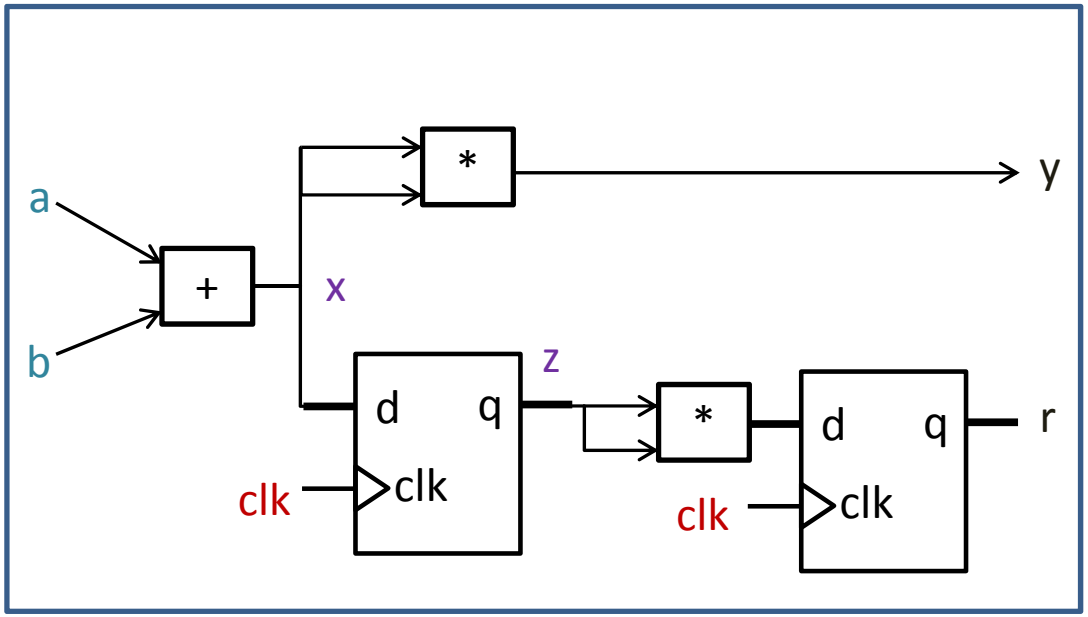
```
always @ (posedge clk)
begin
    if (rst)
        x <= 6'b000000;
    else
        x <= a + b;
end
```

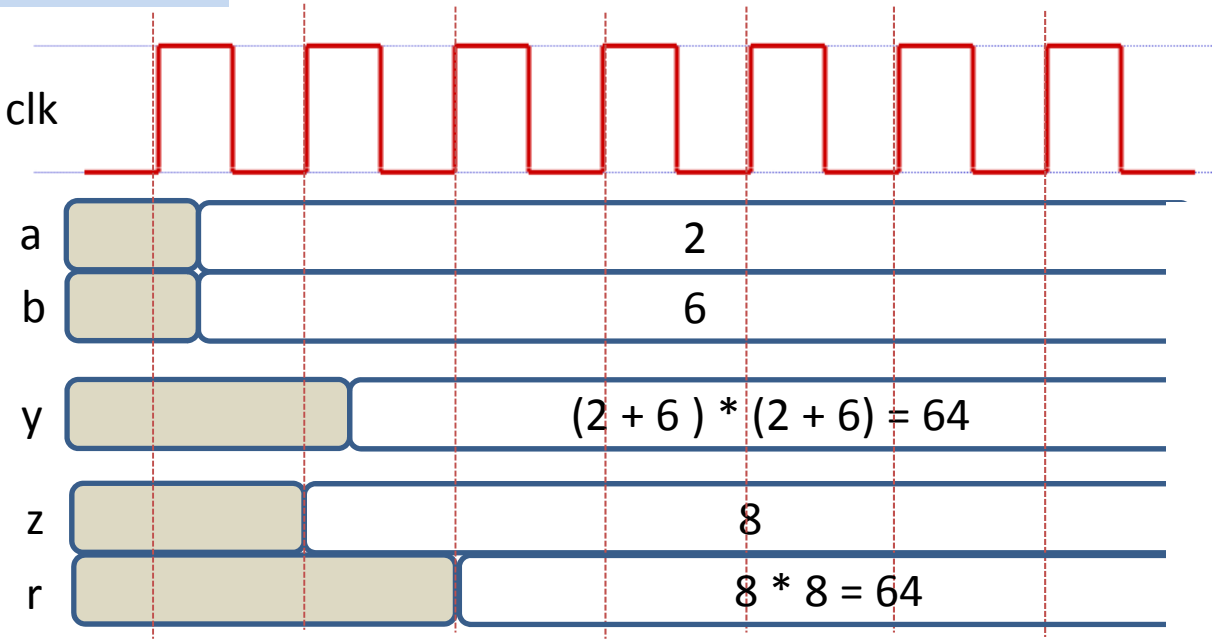




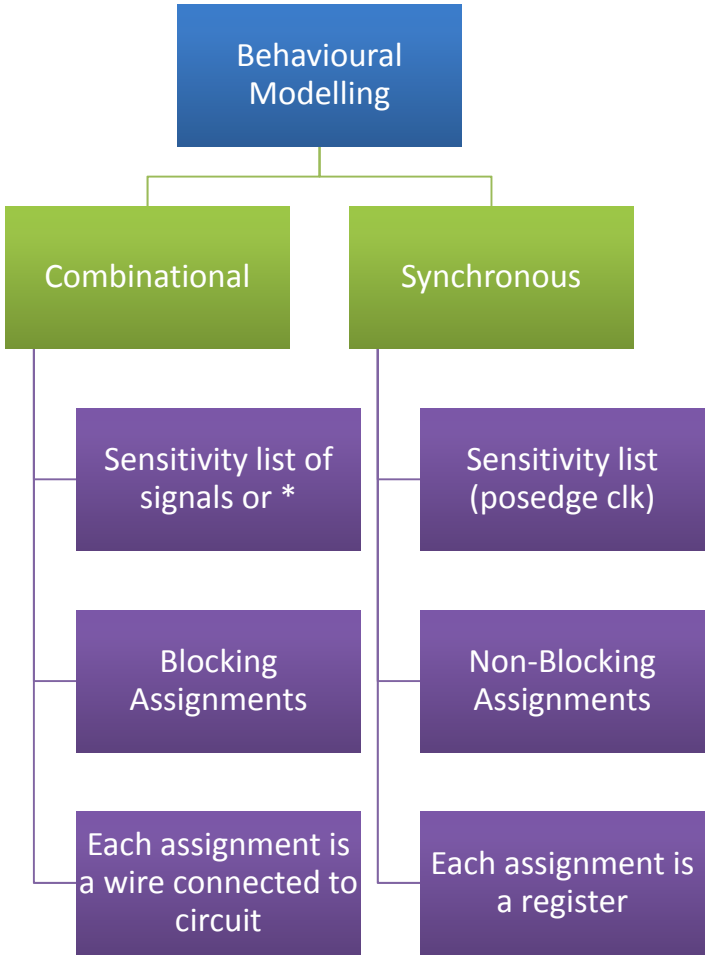
Combinational and Synchronous

```
always @ *
begin
    x = a + b;
    y = x * x;
end

always@(posedge clk)
begin
    z <= x;
    r <= z * z;
end
```



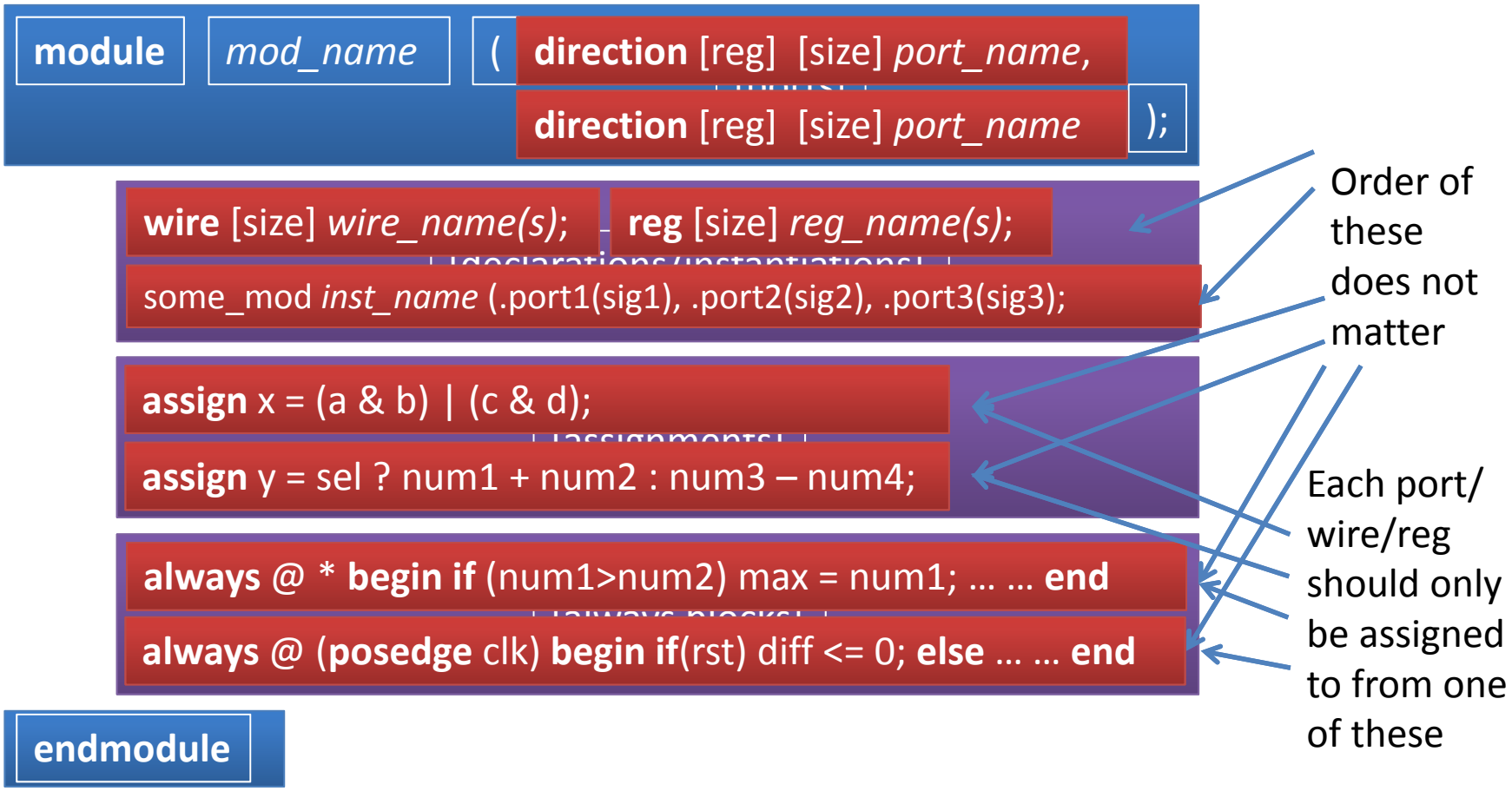


Behavioral Verilog



Code structure

- This general structure for a module should help:



Verilog Summary

NANYANG TECHNOLOGICAL UNIVERSITY

- Think in terms of hardware blocks, not instructions
- Think of block descriptions, not programs
- If you can't understand what the circuit will look like, likely it won't work
- There are many ways to achieve the same thing
  - Go for simplicity
  - Split things up when they get unwieldy
- We have not covered everything:
  - Simulation and verification
  - Non-synthesizable constructs