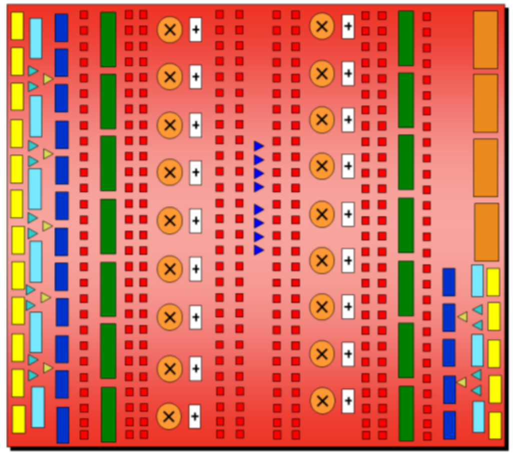
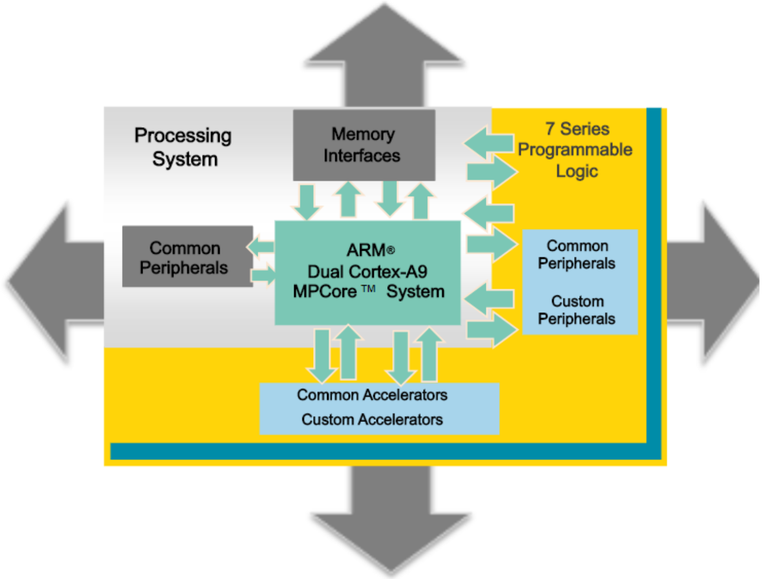
DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING

A2A Algorithms to Architecture

ECE 270: Embedded Logic Design





Behavioral Modeling for Combinational Circuits

- While writing Verilog code for synthesis, we need to be aware of how the various language constructs are mapped to hardware.
- **Common errors**:

1. Variable assigned in multiple always blocks
2. Incomplete sensitivity list
3. Incomplete branch and incomplete output assignments

Behavioral Modeling: Guidelines

- Variable assigned in multiple always blocks

```
reg y;
reg a, b, clear;
. . .
always @*
    if (clear) y = 1'b0;

always @*
    y = a & b;
```

```
always @*
    if (clear)
        y = 1'b0;
    else
        y = a & b;
```

- LHS code is not synthesizable since output y is driven by two blocks (i.e. two different circuits).
- No physical circuit exhibits such behaviour.

Behavioral Modeling: Guidelines

- Incomplete sensitivity list

```
module and_gate (out, in1, in2) ;
   input   in1, in2 ;
   output reg out ;

   always @(in1)    // should be (in1 or in2)
      begin
         out = in1 & in2 ;
      end
endmodule
```

- Leaving out an input trigger usually results in a sequential circuit
- **Use always@*** for combinational circuits

Behavioral Modeling: Guidelines

- Incomplete branch and incomplete output assignment

```
always @*
   if (a > b)        // eq not assigned in this branch
      gt = 1'b1;
   else if (a == b) // gt not assigned in this branch
      eq = 1'b1;
                     // final else branch is omitted
```

```
always @*
begin
   gt = 1'b0;   // default value for gt
   eq = 1'b0;   // default value for eq
   if (a > b)
      gt = 1'b1;
   else if (a == b)
      eq = 1'b1;
end
```

```
always @*
   if (a > b)
      begin
         gt = 1'b1;
         eq = 1'b0;
      end
   else if (a == b)
      begin
         gt = 1'b0;
         eq = 1'b1;
      end
   else     // i.e., a < b
      begin
         gt = 1'b0;
         eq = 1'b0;
      end
```

Behavioral Modeling: Guidelines

- Incomplete branch and incomplete output assignment

```
reg [1:0] s
. . .
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
endcase
```

```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   default: y = 1'b1;   // y gets 1 for 2'b01
endcase
```

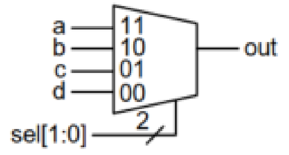
```
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
   default: y = 1'bx; // y gets x for 2'b01
endcase
```

```
y = 1'b0;    // can also use y = 1'bx for don't-care
case (s)
   2'b00: y = 1'b1;
   2'b10: y = 1'b0;
   2'b11: y = 1'b1;
endcase
```

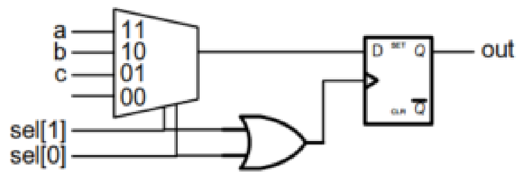
Case Statement: Full and Parallel

- Full Case Statement: all possible outcomes are accounted
- Parallel Case Statement: all stated alternatives are mutually exclusive

```
module full_par (sel, a, b, c, d, out);
input [1:0] sel;
input        a, b, c, d;
output       out; reg out;
always @ (sel or a or b or c or d)
   case (sel)
   2'b11:    out <= a;
   2'b10:    out <= b;
   2'b01:    out <= c;
   default: out <= d; // 2'b00
   endcase
endmodule
```



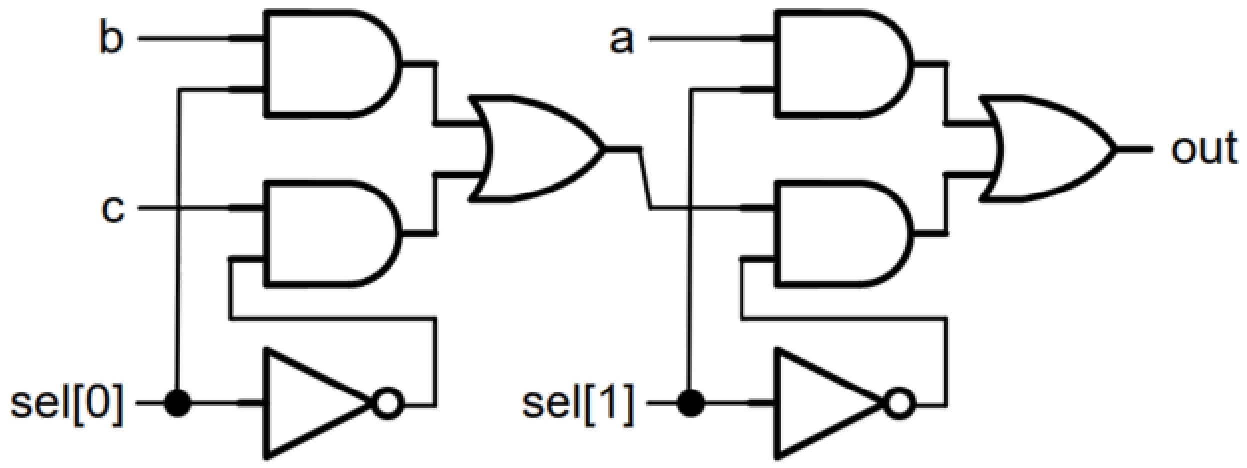
```
module par_not_full (sel, a, b, c, out);
input [1:0] sel;
input        a, b, c;
output       out; reg out;
always @ (sel or a or b or c)
   case (sel)
   2'b11:    out <= a;
   2'b10:    out <= b;
   2'b01:    out <= c;
   endcase
endmodule
```



Case Statement: Full and Parallel

- Full Case Statement: all possible outcomes are accounted
- Parallel Case Statement: all stated alternatives are mutually exclusive

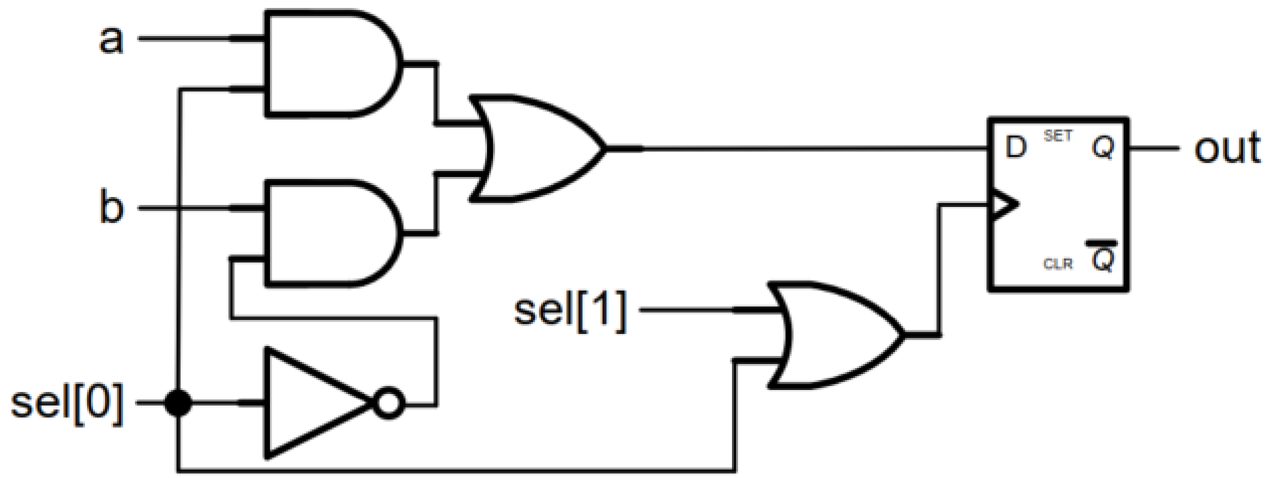
```
module full_not_par (sel, a, b, c, out);
input [1:0] sel;
input       a, b, c;
output      out; reg out;
always @ (sel or a or b or c)
   case (sel)
   2'b1?:   out <= a; // 2'b10, 2'b11
   2'b?1:   out <= b; // 2'b01, 2'b11
   default: out <= c; // 2'b00
   endcase
endmodule
// If the case is 2'b11 occurs, the first outcome gets higher priority because it is closer to the output.
```



Case Statement: Full and Parallel

- Full Case Statement: all possible outcomes are accounted
- Parallel Case Statement: all stated alternatives are mutually exclusive

```
module not_full_not_par (sel, a, b, out);
input [1:0] sel;
input       a, b;
output      out; reg out;
always @ (sel or a or b)
   case (sel)
   2'b1?:   out <= a; // 2'b10, 2'b11
   2'b?1:   out <= b; // 2'b01, 2'b11
   endcase
endmodule
```



Behavioral Modeling for Combinational Circuits

- If an always block executes and a variable is ***NOT*** assigned:
  - ➢ Variable keeps its old value
  - ➢ *NOT* combinational logic -> latch is inserted (implied memory)
  - ➢ This is usually *NOT* what you want

- Any variable assigned in an always block SHOULD be assigned for any (and every!) execution of the block
- Poorly coded always block leads to unnecessarily complex implementation or can not be synthesized at all.

Behavioral Modeling: Summary

- Assign a variable ONLY in a single always block
- **Use @*** to include all the desired identifiers automatically in the sensitivity list
- Make sure that **all branches** of the *if and case statements* are included
- Make sure that the outputs are assigned in all branches
- One way to satisfy previous two guidelines is to assign default values for outputs in the beginning of the always block
- **Use blocking assignments for combinational circuits**
- Think hardware, not C or Python or MATLAB code

Verilog (Three Concepts)

- Difference between Register and Wire (Next two lectures)
- Efficient Behavioral modelling
- **Difference between blocking and non-blocking assignments**

Blocking Vs Non-Blocking Assignments

Behavioral Modeling

- ***Blocking* assignment** statements are executed in the order they are specified in a sequential block
- A blocking assignment will NOT block execution of statements that follows in a parallel block
- The = operator is used to specify blocking assignments
- ***Non-blocking* assignments** allow scheduling of assignments without locking execution of the statements that follow in a sequential block
- A <= operator is used to specify non-blocking assignments

Behavioral Modeling

```
reg_a = 16’b0 ;
```

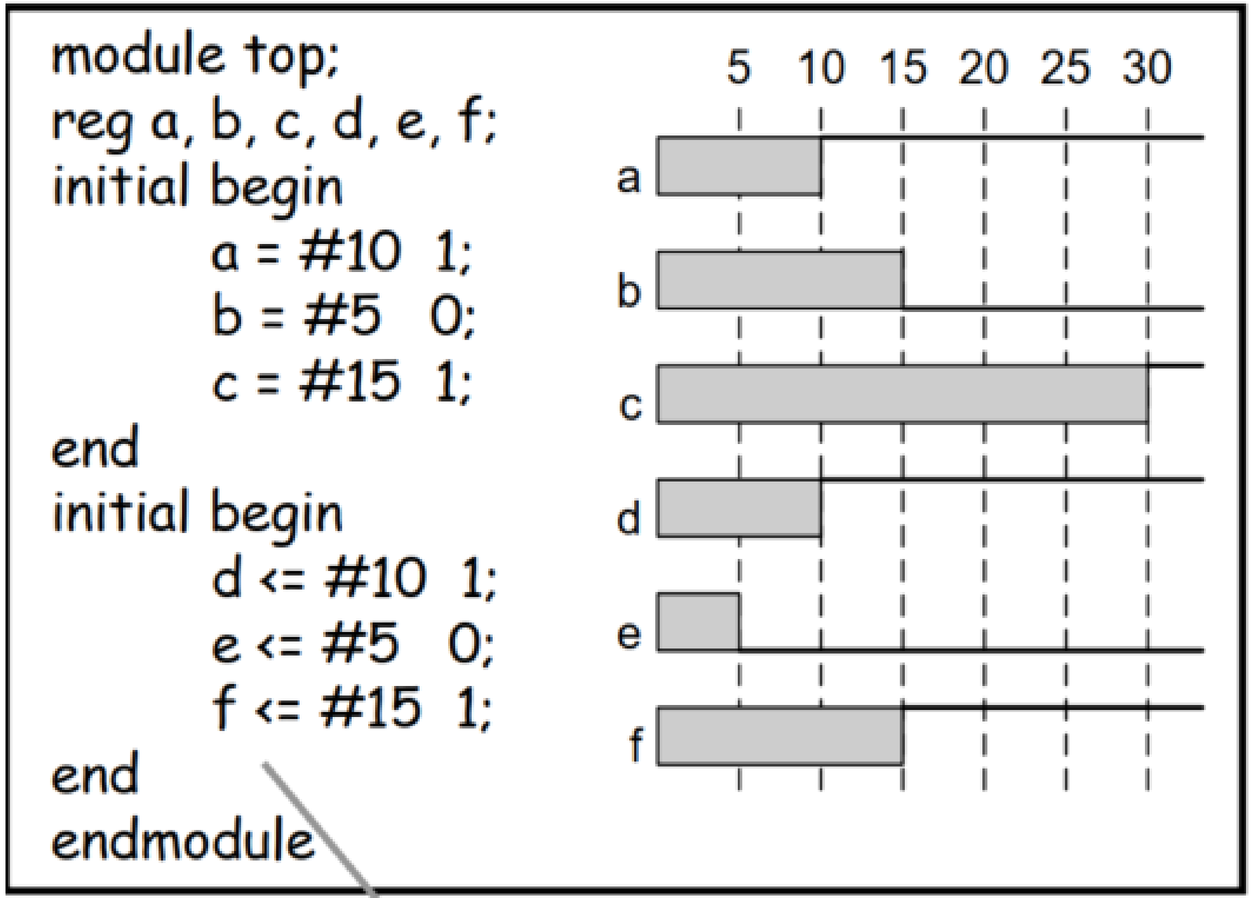
```
reg [15:0] reg_a, reg_b ;
reg x, y, z ;
integer count ;
Initial
    begin
        x = 0 ; y = 0; z = 0 ; count = 0 ;
        reg_b = reg_a ;
        #15 reg_a[2] = 1'b1 ;
        #10 reg_b[15:13] = {x, y, z} ;
        count = count + 1 ;
    end
```

Behavioral Modeling

```
reg_a = 16'b0 ;
```

```
reg [15:0] reg_a, reg_b ;
reg x, y, z ;
integer count ;
Initial
  begin
    x <= 0 ; y <= 0; z <= 0 ; count <= 0 ;
    reg_b <= reg_a ;
    #15 reg_a[2] <= 1'b1 ;
    #10 reg_b[15:13] <= {x, y, z} ;
    count <= count + 1 ;
  end
```

Behavioral Modeling



```
module top;
reg a, b, c, d, e, f;
initial begin
        a = #10  1;
        b = #5   0;
        c = #15  1;
end
initial begin
        d <= #10  1;
        e <= #5   0;
        f <= #15  1;
end
endmodule
```

Note that what are different with blocking and non-blocking assignments.

Behavioral Modeling

```
always @(posedge clock)

  begin reg1 <= #1 in1 ;

    reg2 <= @(negedge clock) in2 ^ in3 ;
    reg3 <= #1 reg1 ;
  end
```

Behavioral Modeling: Swapping

Race Conditions: // Two concurrent always blocks with blocking

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

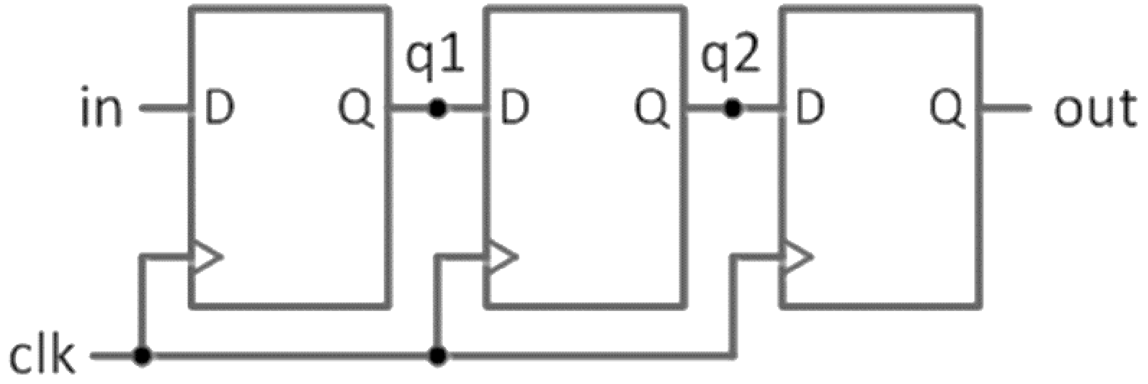
The values of both registers will not be swapped */

Race Conditions: /* Two concurrent always blocks with non- blocking

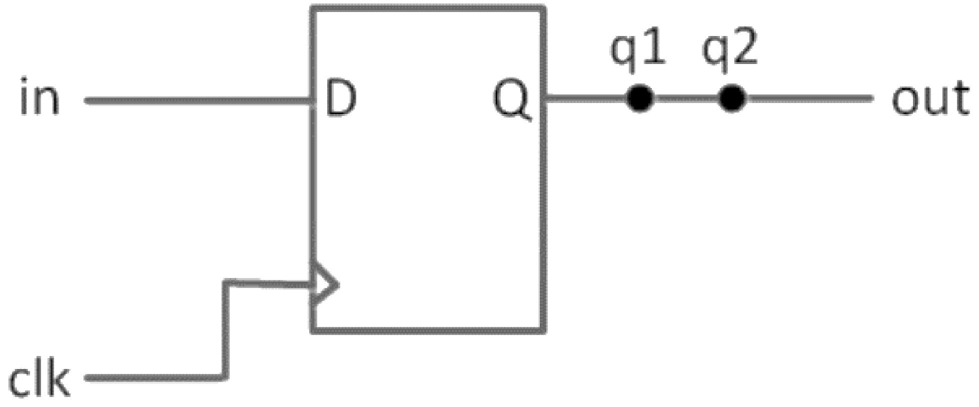
```
always @(posedge clock)
    a <= b ;
always @(posedge clock)
    b <= a ;
```

Behavioral Modeling

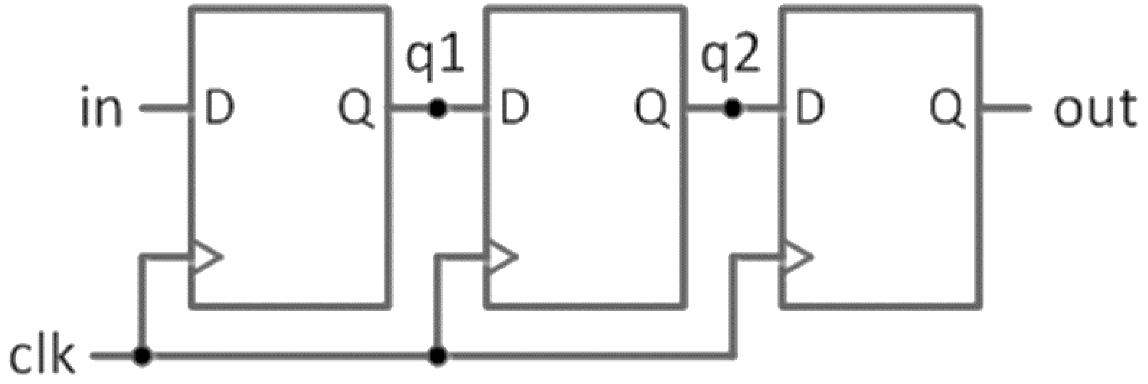
```
module blocking (in,clk,out) ;
   input in, clk ;
   output reg out ;
   reg q1, q2 ;
   always @(posedge clk)
      begin
         q1 = in ;
         q2 = q1 ;
         out = q2 ;
      end
endmodule
```



"at each rising clock edge, q1 = in,
after that, q2 = q1 = in
after that, out = q2 = q1 = in
Therefore, out = in"



Behavioral Modeling

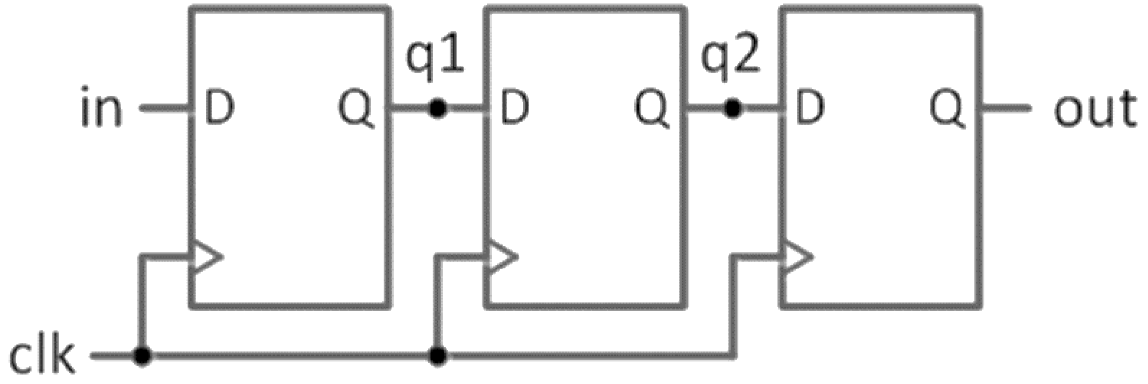


```
module nonblocking (in,clk,out) ;
   input in, clk ;
   output reg out ;
   reg q1, q2;
   always @(posedge clk)
      begin
         q1 <= in ;
         q2 <= q1 ;
         out <= q2 ;
      end
endmodule
```

"at each rising clock edge, q1, q2 and out simultaneously receive the old values of in, q1 and q2. Therefore, out = q2"

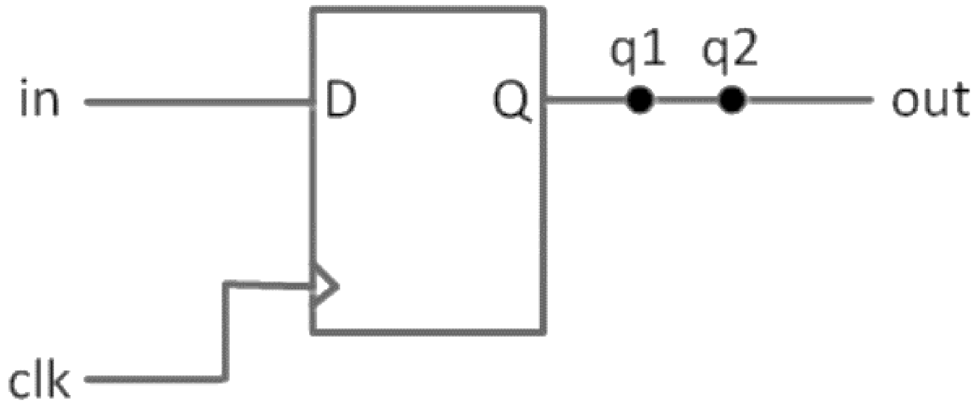
- Blocking assignments do not reflect the intrinsic behaviour of multi-stage sequential logic
- Use non-blocking assignments for sequential always blocks

Behavioral Modeling

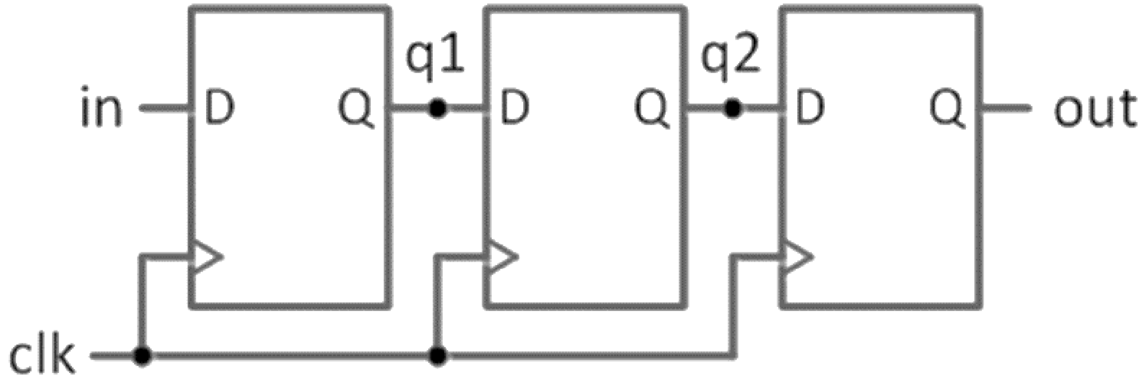


```
module blocking (in,clk,out) ;
   input in, clk ;
   output reg out ;
   reg q1, q2 ;
   always @(posedge clk)
      begin
         q1 = in ;
         q2 = q1 ;
         out = q2 ;
      end
endmodule
```

"at each rising clock edge, q1 = in,
after that, q2 = q1 = in
after that, out = q2 = q1 = in
Therefore, out = in"



Behavioral Modeling

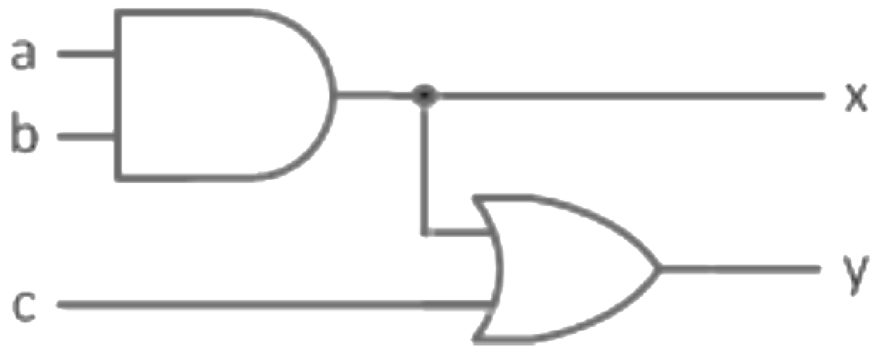


```
module nonblocking (in,clk,out) ;
   input in, clk ;
   output reg out ;
   reg q1, q2;
   always @(posedge clk)
      begin
         q1 <= in ;
         q2 <= q1 ;
         out <= q2 ;
      end
endmodule
```

"at each rising clock edge, q1, q2 and out simultaneously receive the old values of in, q1 and q2. Therefore, out = q2"

- Blocking assignments do not reflect the intrinsic behaviour of multi-stage sequential logic
- Use non-blocking assignments for sequential always blocks

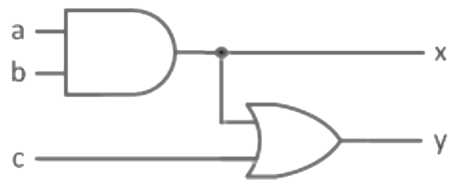
Behavioral Modeling



```
module blocking (a, b, c, x, y) ;
    input a, b, c ;
    output reg x, y ;
    always @ (a or b or c)
      begin
         x = a & b ;
         y = x | c ;
      end
endmodule
```

```
module nonblocking (a, b, c, x, y) ;
    input a, b, c ;
    output reg x, y ;
    always @ (a or b or c)
      begin
         x <= a & b ;
         y <= x | c ;
      end
endmodule
```

Behavioral Modeling



```
module blocking (a, b, c, x, y) ;
   input a, b, c ;
   output reg x, y ;
   always @ (a or b or c)
     begin
        x = a & b ;
        y = x | c ;
     end
endmodule
```

```
module nonblocking (a, b, c, x, y) ;
   input a, b, c ;
   output reg x, y ;
   always @ (a or b or c)
     begin
        x <= a & b ;
        y <= x | c ;
     end
endmodule
```

- Given initial conditions:

  a=1, b=1, c=0, x=1, y=1.
- a changes to 0. always block triggered.
- Blocking behaviour of simulator: 1st calculates x = a & b = 0. Then calculates y = x | c = 0
- Non-Blocking behavior of simulator: Concurrently calculates x(new) = a & b = 0. y = x(old) | c = 1
- Non-blocking assignment *do not* reflect the intrinsic behavior of multi-stage combinational logic
- While non-blocking assignments can be hacked to simulate correctly (expand sensitivity list), its not elegant
- **Guideline**: Use blocking assignments for combinational always blocks

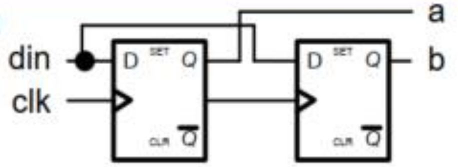
Behavioral Modeling

- When modelling sequential logic, use non-blocking assignments.
- When modelling combinational logic with an always block, use blocking assignments.
- When modelling both sequential and combinational logic within the same always block, use non-blocking assignments.
- Do not mix blocking and non-blocking assignments in the same always block.

Behavioral Modeling (Self Study)

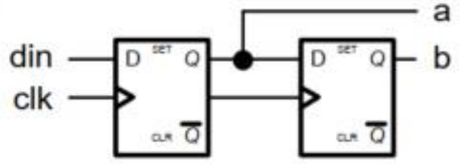
Blocking assignments (1/2)

```
always @ (posedge clk)  begin
   a = din;
   b = a;
end
```



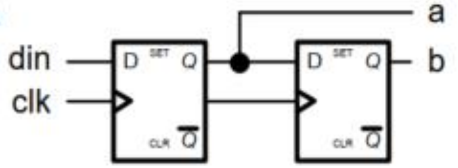
Non-blocking assignments (1/2)

```
always @ (posedge clk)  begin
   a <= din;
   b <= a;
end
```



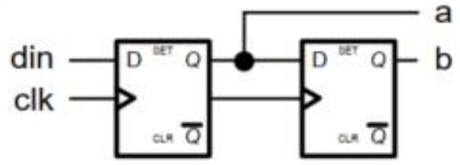
Blocking assignments (2/2)

```
always @ (posedge clk)  begin
   b = a;
   a = din;
end
```



Non-blocking assignments (2/2)

```
always @ (posedge clk)  begin
   b <= a;
   a <= din;
end
```


a
b
din
clk
D
Q
SET
CLR
Q̄