Lecture 6: HDL V

ECE 228

Mostafizur Rahman
rahmanmo@umkc.edu

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

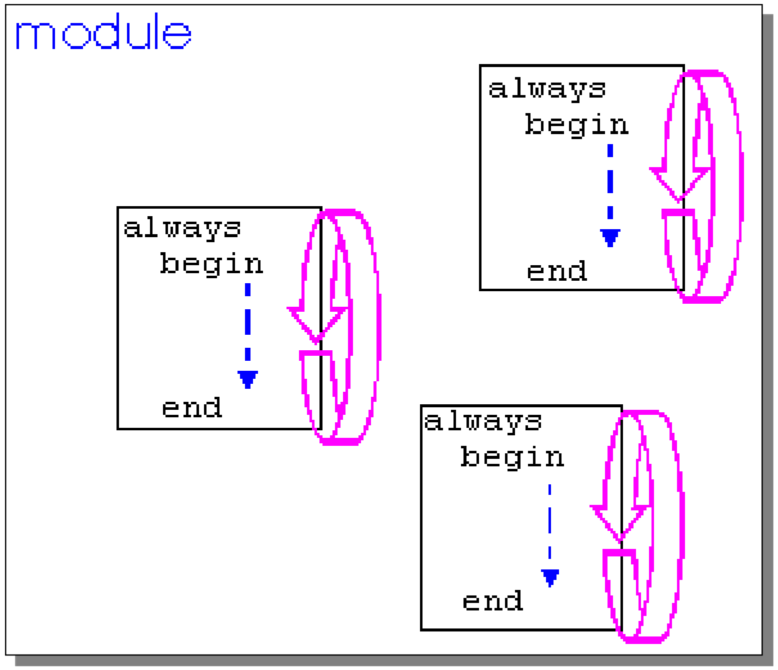
- Procedural statements: statements inside a procedure (they execute sequentially)

- 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 ..

Assignment

- Continuous assignment
  - Values are assigned to ***net variables*** due to some input variable changes
  - "assign …=… "
- Procedural assignment
  - Procedural assignment is used to assign value to variables. A variable cannot be a net type signal
  - Values are assigned to ***register variables*** when certain statement is executed in a behavioral description
  - Procedural assignment, "="
  - Variable assignments must be in behavioral blocks.
- Difference between continuous assignment and procedural assignment
  - Continuous assignment changes the value of the target net whenever the right-hand-side operands change value.
  - Procedural assignment changes the target register only when the assignment is executed according to the sequence of operations

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 it 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
$1 \to x$	$0 \to x$
$1 \to z$	$0 \to z$
$1 \to 0$	$0 \to 1$
$x \to 0$	$x \to 1$
$z \to 0$	$z \to 1$

- Level Triggered Event Control

  @ (A or B) //change in values of A or B

  Out = A & B;

Activity- Draw Timing Diagrams and Correct

- 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 D)
   Q = D;

endmodule
```

Notice outputs?

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

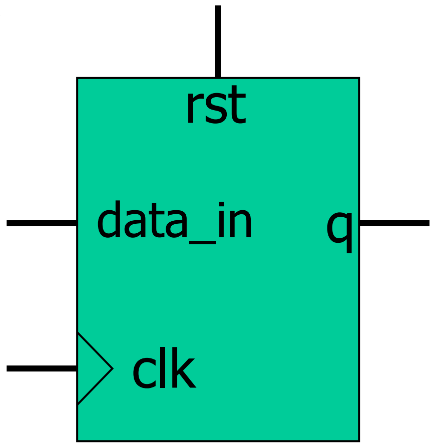
- Write behavioral code for 4:1 multiplexer, with 4 inputs (A, B, C) each 1 bit and selects (S0, S1). Starting value for select should be 11

Example of Flip-flop

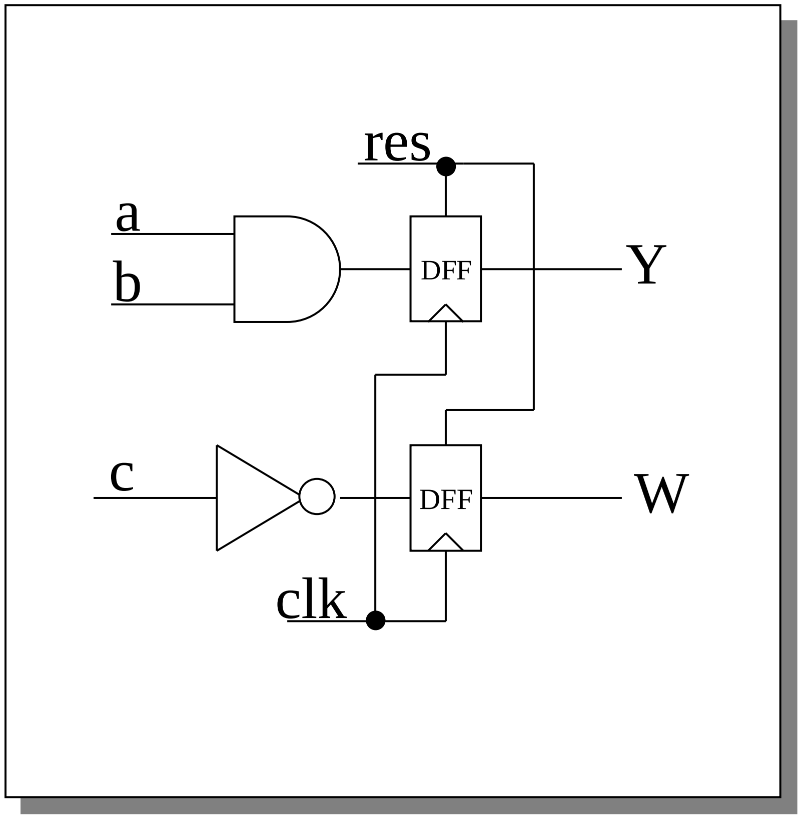
```
module Flip_flop ( q, data_in, clk, rst );
    input          data_in, clk, rst;
    output         q;
    reg   q;

    always @ ( posedge clk )
      begin
          if ( rst == 1) q = 0;
          else q = data_in;
      end
endmodule
```

Declaration of synchronous behavior

Procedural statement



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:

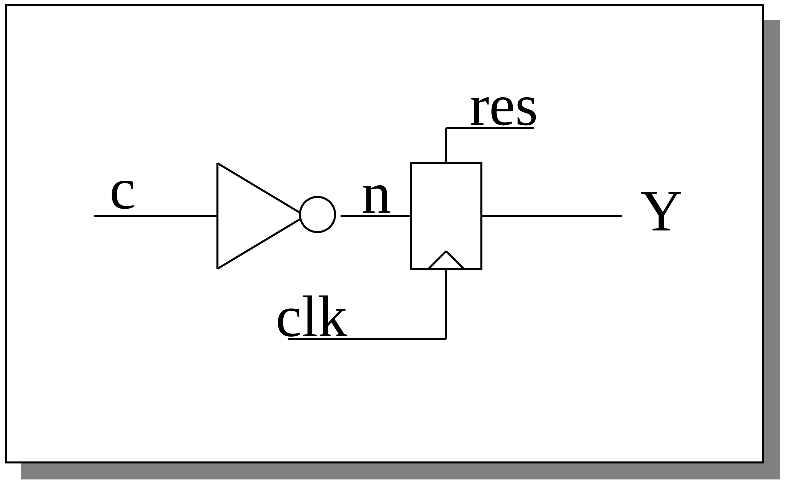
```
if (condition)
   procedural_statement
else if (condition)
   procedural_statement
else
   procedural_statement
```

- 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
```

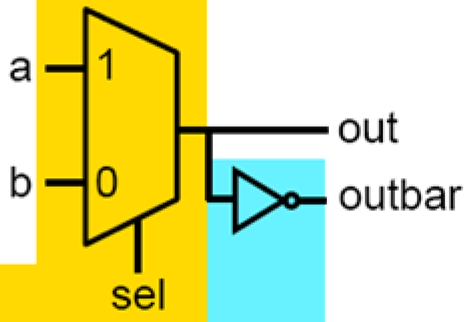
Any other examples?

Procedural Assignment with always

- Procedural and continues 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

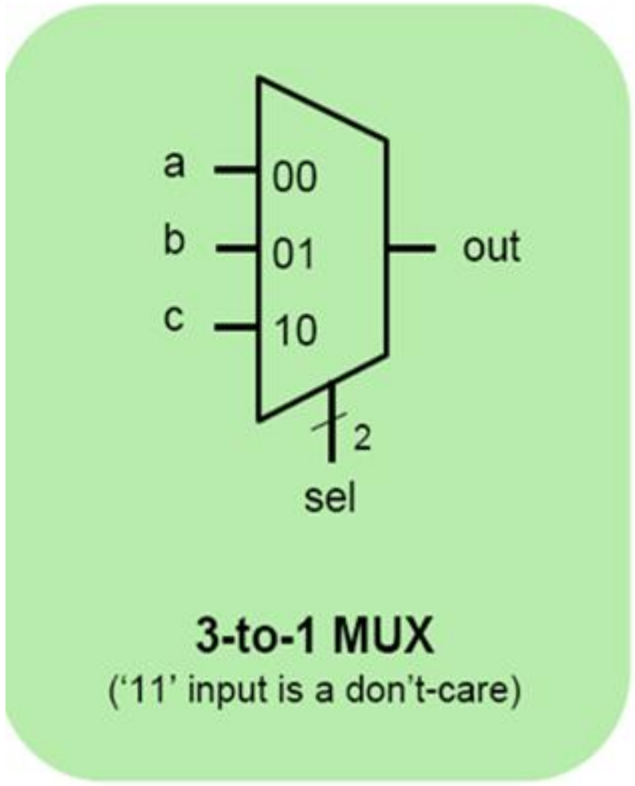
- ***case*** and ***if*** may used interchangeably to implement conditional execution within ***always*** blocks
- case is easier to read than a long string of ***if*** then ***else*** statements

```
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;
```

```
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
```

Danger of Incomplete Specification

Goal:



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?

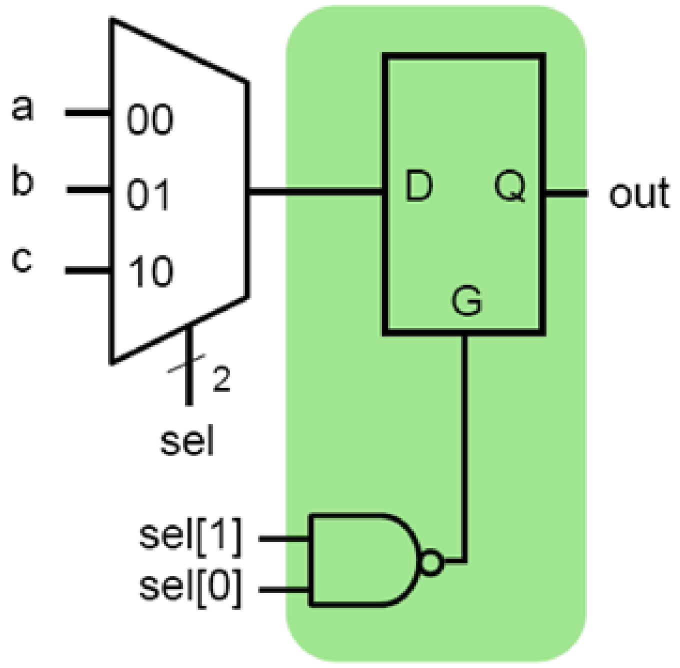
Incomplete Specifications

- Incomplete Specification Infers Latches

```
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
```

if `out` is not assigned during any pass through the always block, then the previous value must be retained!

Synthesized Result:



- Latch memory “latches” old data when G=0
- In practice, we almost *never* intend this

Avoiding Incomplete Specification

- Precede all the conditional statements with a default assignment for all signals assigned within them

```
always @(a or b or c or sel)
  begin
    out = 1'bx;
    case (sel)
      2'b00: out = a;
      2'b01: out = b;
      2'b10: out = c;
    endcase
  end
endmodule
```

- Or fully specify all the branches of conditional statements and assign all signals from all branches
  - For each ***if***, include ***else***
  - For each ***case***, include ***default***

```
always @(a or b or c or sel)
 begin
 case (sel)
     2'b00: out = a;
     2'b01: out = b;
     2'b10: out = c;
     default: out = 1'bx;
   endcase
 end
endmodule
```

Behavioral : 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( a or b or c or d or sel )
  begin
    if ( sel == 0 )
      out = a;
    else if ( sel == 1 )
      out = b
    else if ( sel == 2 )
      out = c
    else if ( sel == 3 )
      out = d
  end

endmodule
```

An always block is a behavioral block which contains a list of expressions which are (usually) evaluated sequentially

The code in an always block can be very abstract (similar to C code) – here we implement a mux with an if/else statement

Behavioral : 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( a or b or c or d or sel )
  begin
    if ( sel == 0 )
      out = a;
    else if ( sel == 1 )
      out = b
    else if ( sel == 2 )
      out = c
    else if ( sel == 3 )
      out = d
  end

endmodule
```

An always block can include a sensitivity list – if any of these signals change then the always block is executed

Behavioral : 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( a, b, c, d, sel )
  begin
    if ( sel == 0 )
      out = a;
    else if ( sel == 1 )
      out = b
    else if ( sel == 2 )
      out = c
    else if ( sel == 3 )
      out = d
  end

endmodule
```

In Verilog-2001 we can use a comma instead of the `or`

Behavioral : 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( a, b, c, ~~d~~, sel )
  begin
    if ( sel == 0 )
      out = a;
    else if ( sel == 1 )
      out = b
    else if ( sel == 2 )
      out = c
    else if ( sel == 3 )
      out = d
  end

endmodule
```

What happens if we accidentally leave off a signal on the sensitivity list?

The always block will not execute if just `d` changes – so if `sel == 3` and d changes then out will not be updated

Behavioral : 4-input Multiplexer

```
module mux4( input  a, b, c, d
             input [1:0] sel,
             output out );

  reg out;

  always @( * )
  begin
    if ( sel == 2’b00 )
      out = a;
    else if ( sel == 2’b01 )
      out = b
    else if ( sel == 2’b10 )
      out = c
    else if ( sel == 2’b11 )
      out = d
  end

endmodule
```

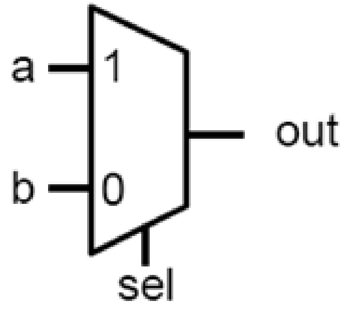
In Verilog-2001 we can use the @(*) construct which creates a sensitivity list for all signals read in the always block

Sequential *always* Block

- Edge triggered circuits are described using a sequential ***always*** block

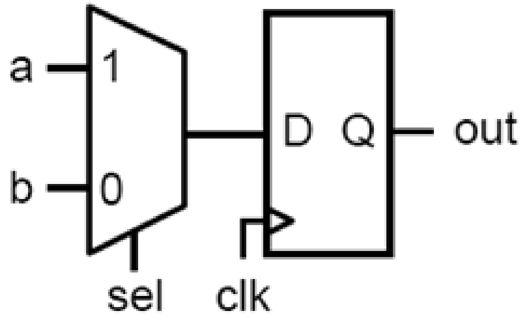
Combinational

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

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

endmodule
```



Sequential

```
module sequential(a, b, sel,
                     clk, out);
  input a, b;
  input sel, clk;
  output out;
  reg out;

  always @ (posedge clk)
  begin
    if (sel) out <= a;
    else out <= b;
  end

endmodule
```



Blocking & Non-blocking Assignments

- Sequential statements within procedural blocks (“always” and “initial”) can use two types of assignments:
  - Blocking assignment: A blocking statement must be executed before the execution of the statements that follow it in a sequential block.
    - Uses the ‘=‘ operator
    - Strict order of execution; block the assignments that follows
    - Recommended style for modeling combinational circuit
  - Non-blocking assignment: Allows to schedule assignments without blocking the procedural flow. Can make several register assignments within the same time step irrespective of the statements order or dependence upon each other.
    - Uses the ‘<=‘ operator
    - Recommended style for modeling sequential circuit

Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within `always` blocks, with subtly different behaviors.
- *Blocking assignment:* evaluation and assignment are immediate

```
always @ (a or b or c)
begin
  x = a | b;                 1. Evaluate a | b, assign result to x
  y = a ^ b ^ c;             2. Evaluate a^b^c, assign result to y
  z = b & ~c;                3. Evaluate b&(~c), assign result to z
end
```

- *Nonblocking assignment:* all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)

```
always @ (a or b or c)
begin
  x <= a | b;                1. Evaluate a | b but defer assignment of x
  y <= a ^ b ^ c;            2. Evaluate a^b^c  but defer assignment of y
  z <= b & ~c;               3. Evaluate b&(~c) but defer assignment of z
end                          4. Assign x, y, and z with their new values
```

- Sometimes, as above, both produce the same result. Sometimes, not!

Blocking and Non-blocking Assignment

```
initial
  begin
    a = 1;
    b = 0;
    a = b;    // a = 0;
    b = a;    // b = 0;
  end
initial
  begin
    a = 1;
    b = 0;
    a <= b;    // a = 0;
    b <= a;    // b = 1;
  end
```

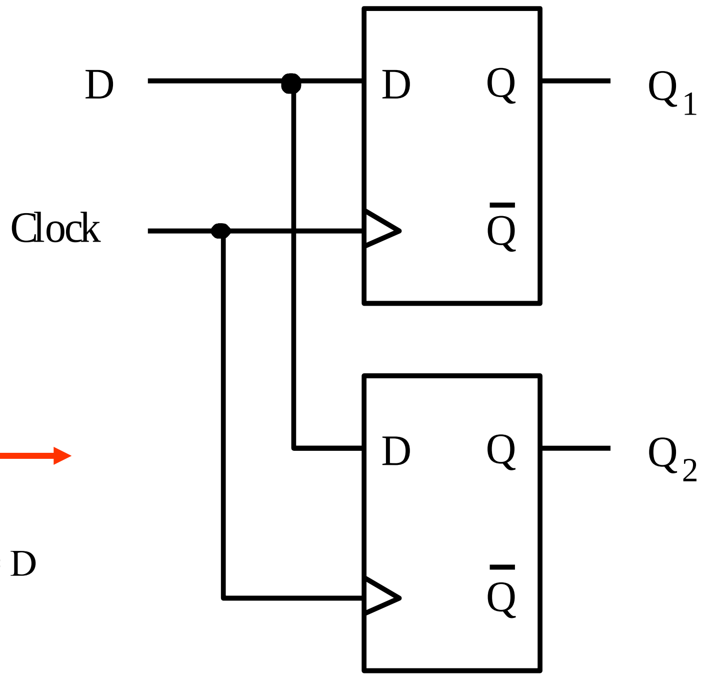
- Blocking assignment “=“
  - Statement order matters
  - A statement has to be executed before next statement
- Non-blocking assignment “<=“
  - Concurrent assignment
  - If there are multiple non-blocking assignments to same variable in same behavior, latter overwrites previous

Blocking Assignment

– The order of statements inside an always block can affect the synthesized design
  - Verilog evaluates the assignments in an always block in the order in which they are written
  - This can cause unexpected designs

```
module example(Q1,Q2,D,Clock);
    input D, Clock;
    output Q1, Q2;
    reg Q1, Q2;

    always @(posedge Clock) begin
        Q1 = D;
        Q2 = Q1;
    end
endmodule
```

treated as Q2 = D since Q1 = D

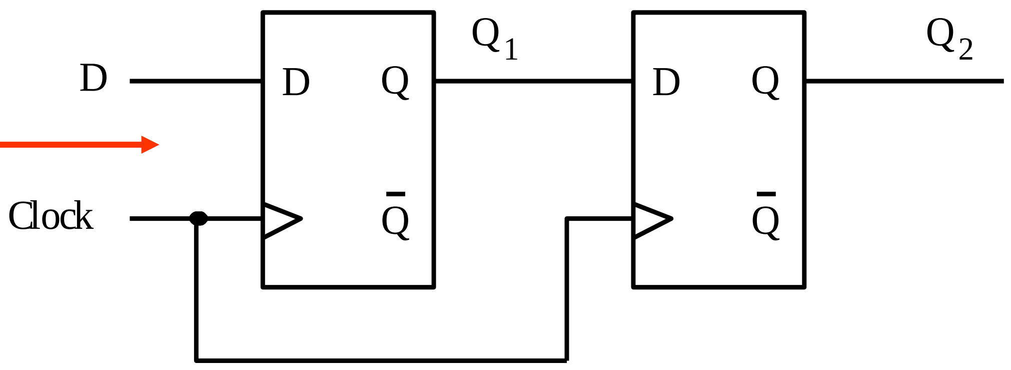


Non-blocking Assignment

- What if you wanted a “cascaded” design for the previous example?
  - The assignment to Q2 should be from the “previous” Q1
  - Need both assignments to occur in parallel on the same clock edge
    - The LHS of a non-blocking assignment is updated after all RHS values for all assignments have been evaluated

```
module example(Q1,Q2,D,Clock);
    input D, Clock;
    output Q1, Q2;
    reg Q1, Q2;

    always @(posedge Clock) begin
        Q1 <= D;
        Q2 <= Q1;
    end
endmodule
```

non-blocking assignment


D
Clock
D
Q
Q̄
Q_1
D
Q
Q̄
Q_2