

Lecture 10

Verilog Language 3

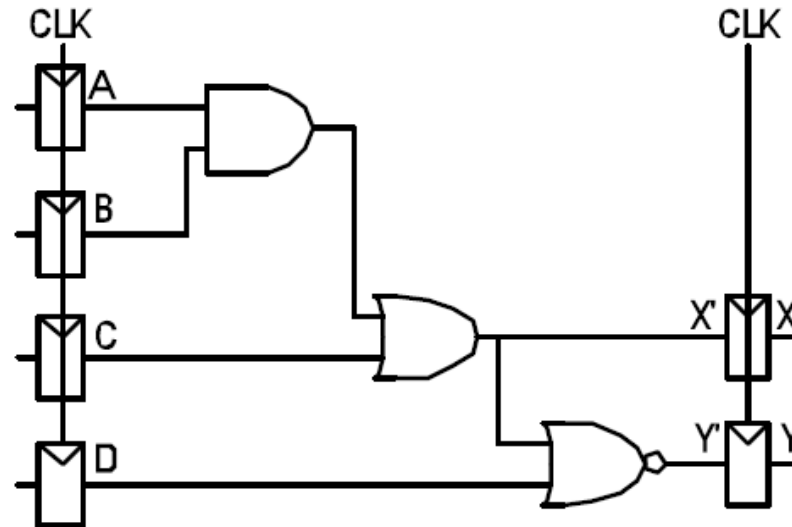


Outline

- Verilog Language (***)
 - Sequential Circuits
 - Textbook 5.14, 10.3



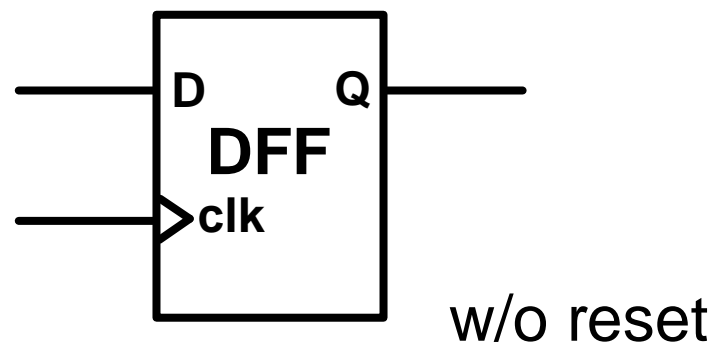
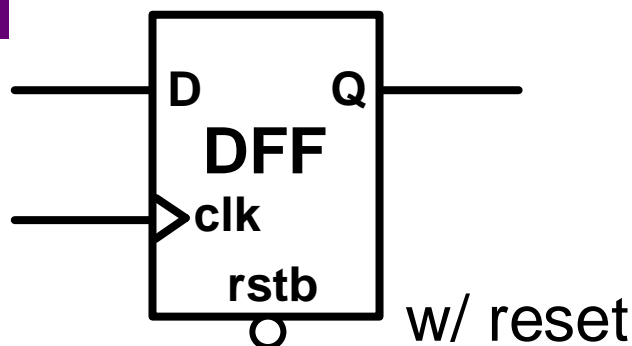
How to synthesize a sequential circuit in Verilog



- For pipelined operation, we need to generate flip-flops between combinational logic



Common Flip-flop



- Normally called "DFF"
- Often also has an asynchronous reset
 - Reset initial value to 0
 - Reset is normally active low
 - Reset independently of clock
 - Can have set or synchronous reset as well



Always Statement for Sequential Circuit

- One of the most important keyword
- Three main places to use
 - Define an iterative sequential event
 - Used in testbench
 - *Define sequential circuits, e.g. flip-flops*
 - Define combination circuits
 - Only use for complex cases
 - Be careful to not create a latch



Always for Sequential Logic

- Sequential logic only updates values upon clock transition
- Define clock in sensitivity list of always
 - For example: `always @(posedge clk)`
 - The flip-flop will update only at positive edge of clock
- Good practice: use non-blocking statement
 - Use “`<=`” rather than “`=`”

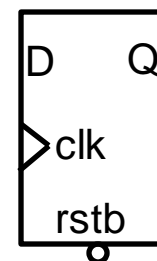


Always for Sequential Logic

Below is the most commonly used asynchronous reset flip-flop

```
module my_pipeline(D, clk, rstb, Q);  
  input D, clk, rstb;  
  output Q;  
  reg Q;  
  always @(posedge clk or negedge rstb)  
    if (!rstb) begin  
      Q <= 0;  
    end  
    else begin  
      Q <= D;  
    end  
endmodule
```

Sensitivity list: Rising edge
of clk or rstb=0



When rstb=0, Q=0



Always for Sequential Logic

- For sequential logic, under "always" you can use most operators:
 - Basic operators, e.g. "+", "*", "&", etc.;
 - if, else;
 - Case;
- You cannot use:
 - Concurrent command:
 - assign
 - instantiation

```
always @(negedge rstb or posedge clk)
    if (!rstb) begin
        Q <= 0;
    end
    else begin
        Q <= A & B;
    end
```

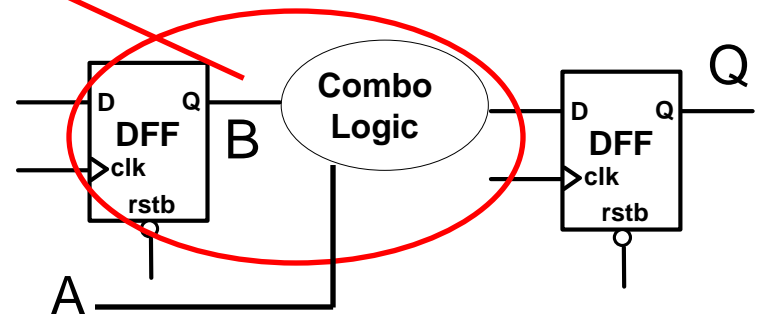
You can build your logic statement here, e.g. if, case, operators.



Example

```
module my_pipeline(A, clk, rstb, Q);  
  input A, clk, rstb;  
  output [1:0] Q;  
  reg B; //Operation of B is not shown in this example  
  reg [1:0] Q;  
  always @(posedge clk or negedge rstb)  
    if (!rstb) begin  
      Q <= 0;  
    end  
    else begin  
      Q <= A + B;  
    end  
endmodule
```

- This is an adder in a pipeline
- A, B can be input or other registers



You can use combinational logic, conditional statement, and registers here



Example

- Use “if...else” for conditional logic

```
module my_counter(clk, rstb, enable, up_en, down_en);  
  input clk, rstb, enable, up_en, down_en;  
  reg [3:0] counter;  
  always @ (posedge clk or negedge rstb)  
    if (rstb == 1'b0) begin  
      counter <= 4'b0000;  
    end  
    else if (enable == 1'b1 && up_en == 1'b1) begin  
      counter <= counter + 1'b1;  
    end  
    else if (enable == 1'b1 && down_en == 1'b1) begin  
      counter <= counter - 1'b1;  
    end  
    else begin  
      counter <= counter;  
    end  
endmodule
```

Can use “if” inside
procedure statement,
e.g. always

This else line can be
omitted since counter
is a “reg” type



Blocking and Nonblocking Assignment

- Under always, you can use:
 - Nonblocking assignment: “<=”
 - Signals are updated in parallel, no order
 - Blocking assignment: “=”
 - Signals are updated in order
- Practically, if you use always to create flip-flop, you should just use nonblocking assignment “<=”
- This is where you need to think about hardware
 - Don't follow C-language (sequential execution)



Use Nonblocking Assignment

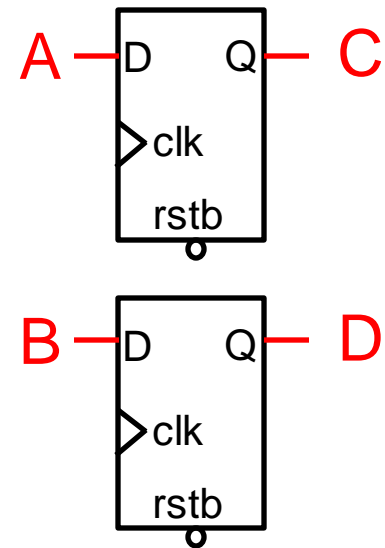
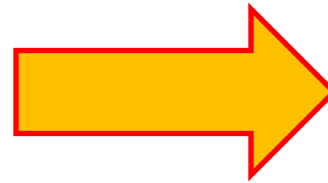
- Practically, if you use always to create flip-flop, you should just use nonblocking assignment “<=”

always @(posedge clk) begin

C <= A;

D <= B;

end



C, D are updated at the same time (rising edge of clock); Order does not matter;



Non-blocking Assignment

always @(posedge clk) begin

All get
new data
at rising
clock
edge

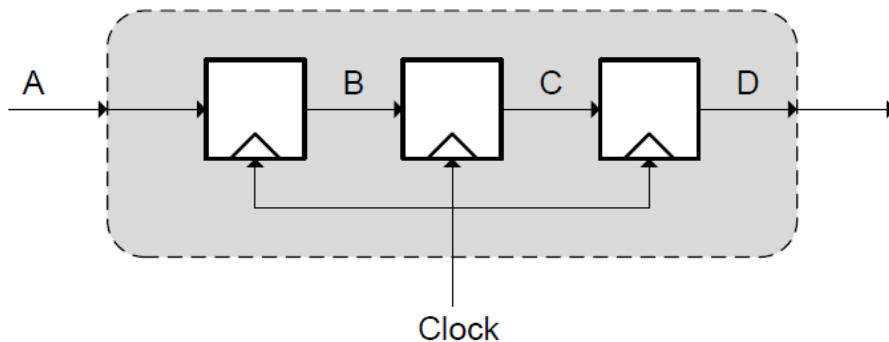
B <= A;
C <= B;
D <= C;

end

These three can be written
in any order

You should think of this:
At clock rising edge, all
“previous data inputs” are
passed into flops

This is a common pipeline
structure.





Blocking Assignment

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

```
    B = A;
```

```
    C = B;
```

```
    D = C;
```

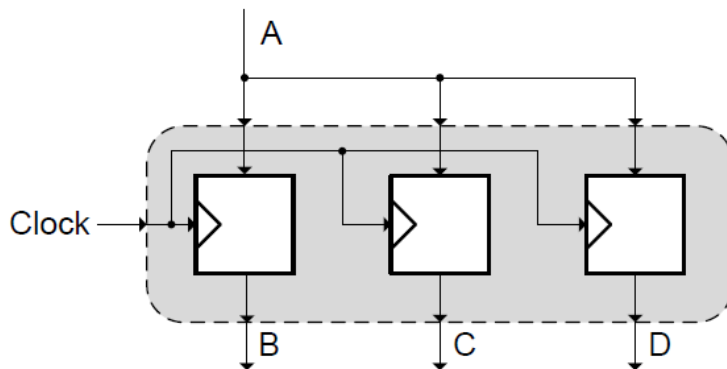
```
end
```

Values are updated sequentially

So everyone gets the same value A;

You rarely need this circuit;

So DO NOT use “blocking” inside an always for sequential circuits





Understanding Non-blocking Operation

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

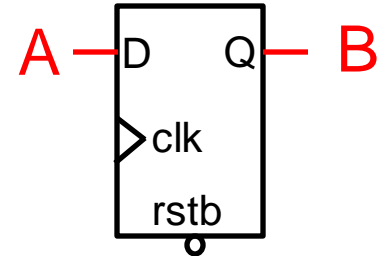
```
  B <= A;
```

```
  C <= B;
```

```
  D <= C;
```

```
end
```

Think right-hand-side
of data as different
data from left-hand-
side of data



Intuitively it is doing the following



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

```
  B(n) <= A(n-1);
```

```
  C(n) <= B(n-1);
```

```
  D(n) <= C(n-1);
```

```
end
```

This is not the right
syntax, but help you
understand

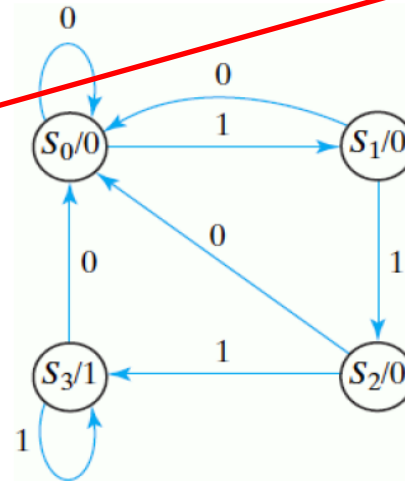
clock cycle n clock cycle n-1



Always for Finite State Machine

- Use "case" to create multiple choices
 - No need to cover all the state (no longer combinational)
 - Will talk more later in finite state machine

```
module seq3_detect_moore(x,clk, y);  
  // Moore machine for a three-1s sequence detection  
  input x, clk;  
  output y;  
  reg [1:0] state;  
  parameter S0=2'b00, S1=2'b01, S2=2'b10,  
    S3=2'b11;  
  // Define the sequential block  
  always @(posedge clk)  
    case (state)  
      S0: if (x) state <= S1;  
          else state <= S0;  
      S1: if (x) state <= S2;  
          else state <= S0;  
      S2: if (x) state <= S3;  
          else state <= S0;  
      S3: if (x) state <= S3;  
          else state <= S0;  
    endcase  
  // Define output during S3  
  assign y = (state == S3);  
endmodule
```



Should use
rstb to
initialize the
state first
(missing in
this example)



Putting Together

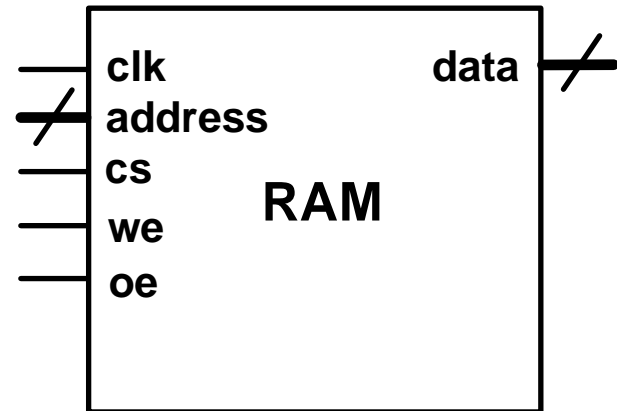
- Use "always @(posedge clk)"
- Good to have reset to initialize the value
"if (rstb == 1'b0) begin"
- Always use non-blocking statement "<="
- Use "reg" for signal assigned by "<="
- Cannot mix with concurrent statement
 - No "assign" and instantiation within always
 - Do not duplicate assignment



Another Example

```
module ram_sp_sr_sw (clk, address, data, cs, we, oe);
    parameter DATA_WIDTH = 8 ;
    parameter ADDR_WIDTH = 8 ;
    parameter RAM_DEPTH = 1 << ADDR_WIDTH;
    input      clk      ;
    input [ADDR_WIDTH-1:0] address  ;
    input      cs       ;
    input      we       ;
    input      oe       ;
    inout [DATA_WIDTH-1:0] data    ;
    reg [DATA_WIDTH-1:0] data_out ;
    reg [DATA_WIDTH-1:0] mem [0:RAM_DEPTH-1];
    reg          oe_r;
    assign data = (cs && oe_r && !we) ? data_out : 8'bz;
    always @ (posedge clk)
        begin : MEM_WRITE
            if ( cs && we ) begin
                mem[address] <= data;
            end
        end
    always @ (posedge clk)
        begin : MEM_READ
            if (cs && !we && oe) begin
                data_out <= mem[address];
                oe_r <= 1;
            end else begin
                oe_r <= 0;
            end
        end
    end
endmodule
```

(from ASIC-world.com)



- Register File or Random Access Memory (RAM)
- “cs”: chip select, equal to 1 to operate
- “we”: write enable, equal to 1 to write
- “oe”: output enable, equal to 1 to output