Computer Organization

Sequential Logic & RTL

Pyeongsu Park, Eunjin Baek

High Performance Computer System (HPCS) Lab

March 20, 2018

Objective

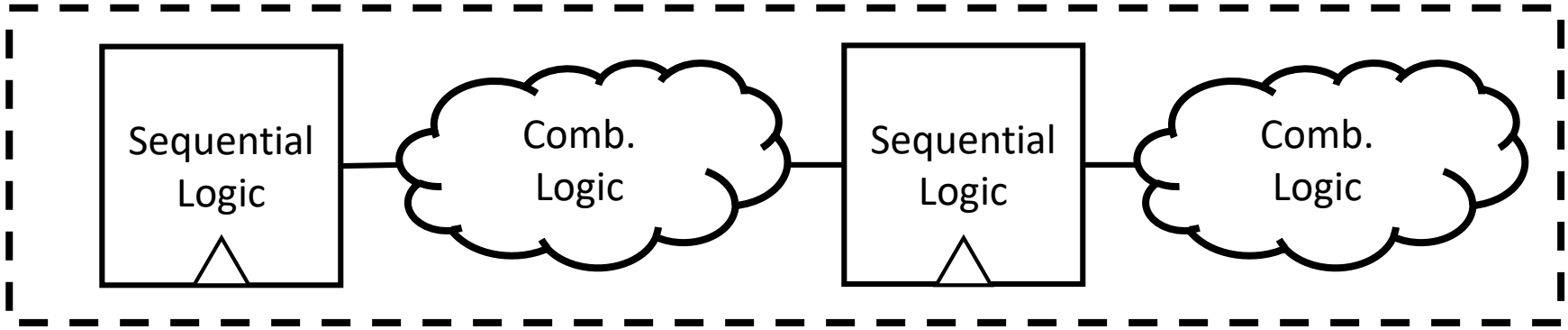
- Understand the **Register Transfer Level (RTL)** concept of HDL
- Understand **how to make a sequential logic** in Verilog

- If you are familiar with FSM and how to implement it in Verilog, then you can skip to page #29.

RTL: a way to represent a circuit

Digital Circuit

- Combinational logic + Sequential logic



Combinational Logic (CL)

- The last assignment, ALU, was a very common example of combinational circuits.

- Outputs are **a function of inputs** ONLY.
  - Outputs = f (inputs)

Sequential Logic (SL)

- Outputs of a logic are **a function of inputs & states**.
  - Outputs = f (inputs, state)

- SLs have **memory elements** to save their states.
  - Remember? SR-latches, JK-FF, D-FF, RAM, ROM

- SLs can contain CLs, but opposite does not hold.

Synchronous Circuit

- A circuit that is **synchronized** to a **clock signal**.
  - Consist of **combinational logics** and **memory elements**.
  - The changes in the value of memory elements are **synchronized to a clock** (positive or negative edge).
- Easy to debug and our focus on the projects.

- Opposite: asynchronous circuit
  - Consists of combinational logics and memory elements, but both are **not synchronized to a clock**.
  - Instead, memory elements change their values according to **external events** (e.g., previous calculation is done).

Verilog Representation – Gate Level

- Represent a logic as **wires** and **logic gates** (AND, NOT …).

- (+) More realistic code than RTL
- (-) Hard to program

Verilog Representation – Register Transfer Level (RTL)

- Represent a logic as **“registers”** and **“data manipulations”** between the registers.
- Programmers can concentrate on the data flow between registers, not real gates.
- Compilers convert/optimize RTL into gate level.

- (+) Can focus functional aspects, not gates
- (+) Easier and faster development than gate level
- (-) Not all the codes can be made into a real HW

Gate Level vs. RTL

```
module FullAdder_GL (A,B,Cin,Cout,S);
   input A, B, Cin;
   output Cout, S;
   wire w1, w2, w3, w4;

   xor (w1, A, B);
   and (w2, A, B);
   and (w3, w1, Cin);
   xor (S, w1, Cin)
   or (Cout, w3, w2);
endmodule
```

VS

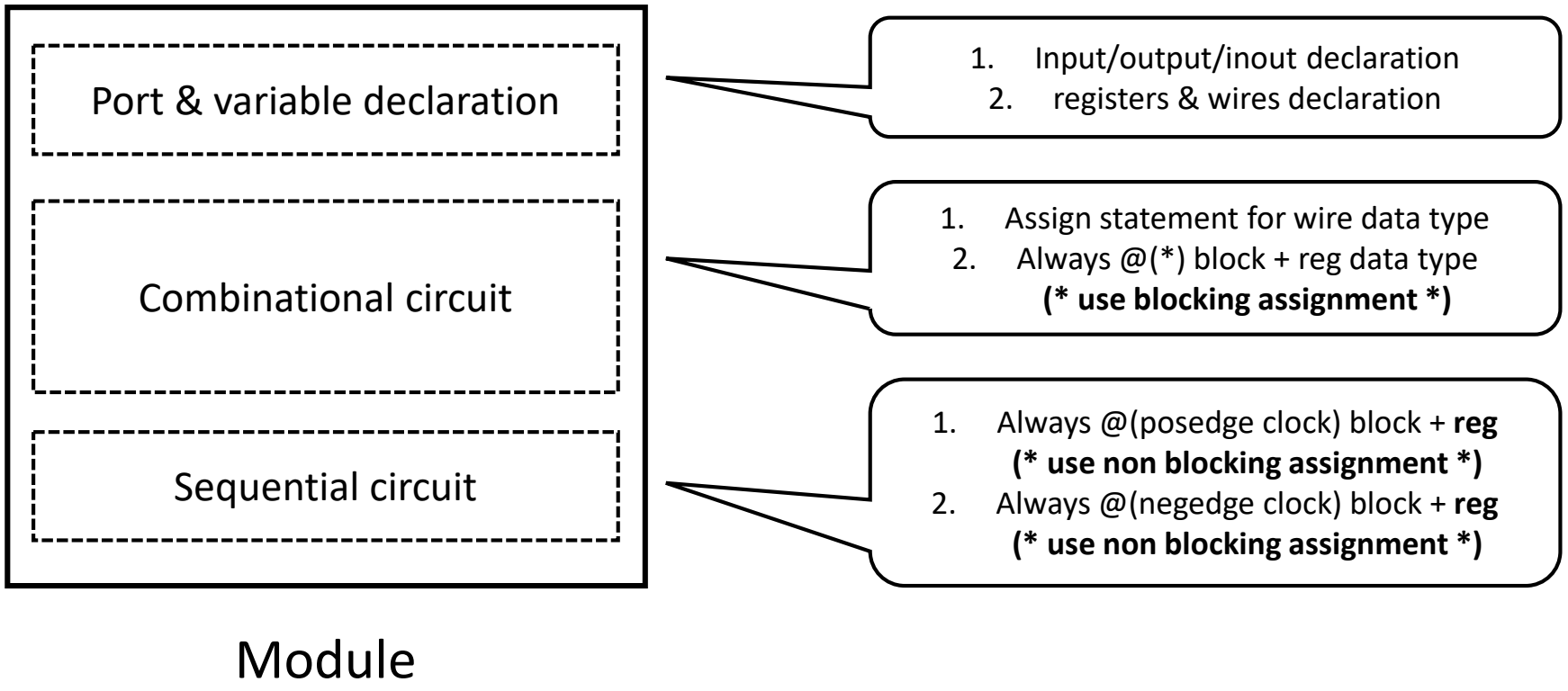
```
module FullAdder_RTL (A,B,S,Cin,Cout);
   input A, B, Cin;
   output S, Cout;

   assign S = A ^ B & Cin;
   assign Cout = (A & B) | (Cin & (A ^ B));
endmodule
```

Gate Level

RTL

General RTL Structure



RTL Programming Guide

- How to make an RTL code from the given block diagram?

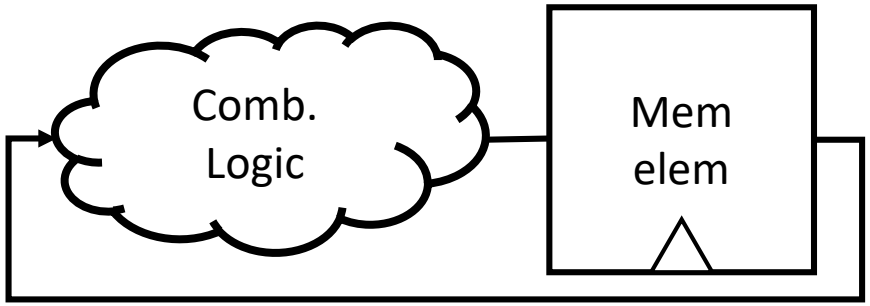
- (1) Understand inputs, outputs, and the functionality of it.
- (2) Determine what to be saved in the registers.
- (3) Think which values should be synchronized to the clock.
- (4) Think about how the data move between the registers.
- (5) Consider what operations should be done between the data movement.

(* step-by-step example will follow after the next few slides. *)

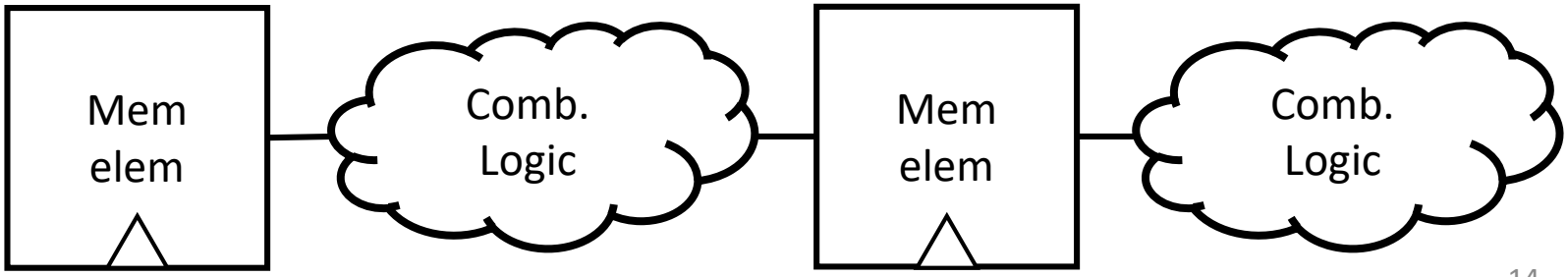
How to write a sequential logic in Verilog

Two Possible SLs in Verilog

- Finite State Machine (FSM)
  - Should know how to implement FSM using Verilog

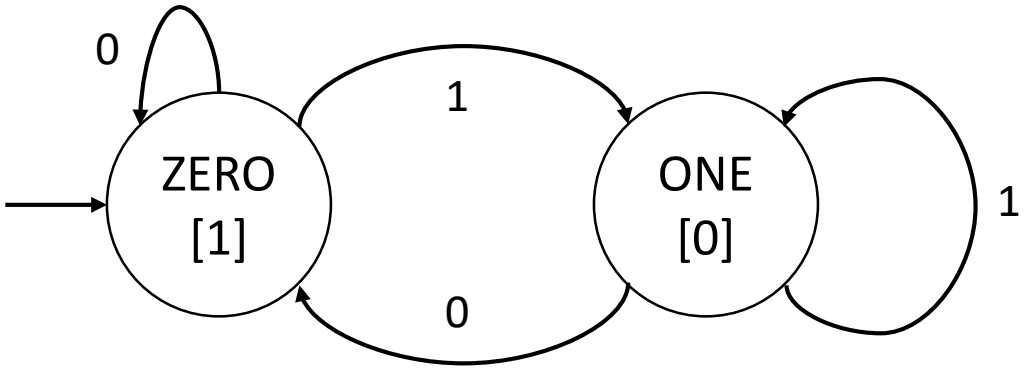


- Pipeline
  - Straight forward: connect each logic using mem elem.



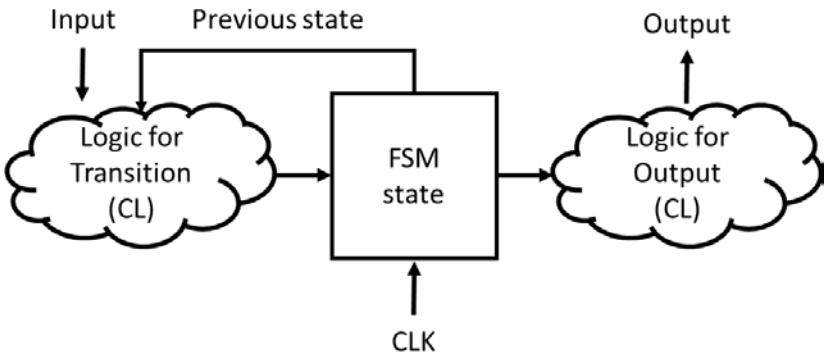
Finite State Machine (FSM)

- Represent a logic as a finite set of states (or nodes), outputs, and transition functions.
  - Transition: movement between nodes for given inputs
- In below example,
  - State = ZERO, ONE
  - Transition function: ZERO = t (ZERO, 0), ZERO = t (ONE, 0), ONE = t (ZERO, 1), ONE = t (ONE, 1)
  - Output: 1 when in ZERO, 0 otherwise.

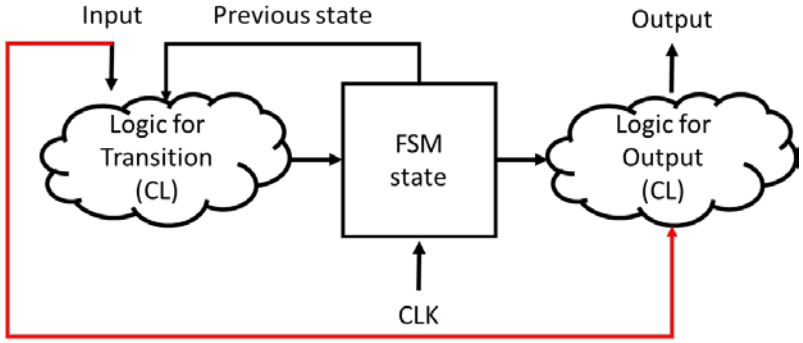


Two Ways to Represent FSM

- We can classify an FSM based on the relationship between outputs and inputs.
- The state is **synchronized to CLK**.
  - The state is changed when clock is positive edge or negative edge.
  - Assume the state is saved in D-flip-flops.
- Two types of representation of an FSM: **Moore & Mealy**
  - You can implement the same logic in both ways.



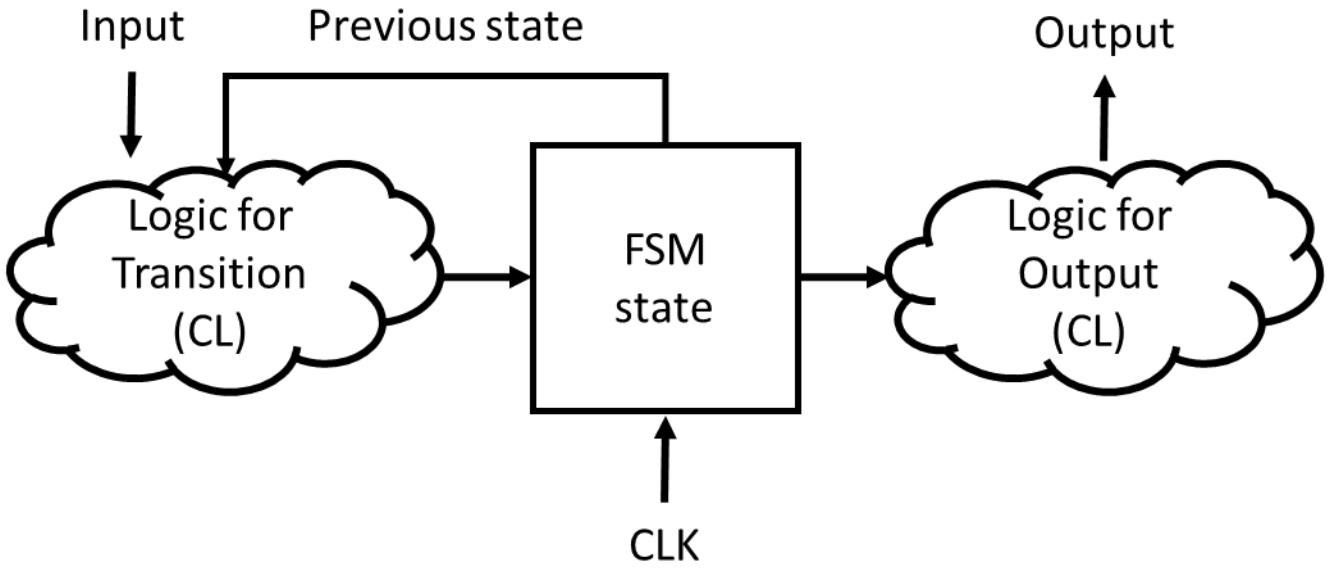
Moore Machine



Mealy Machine

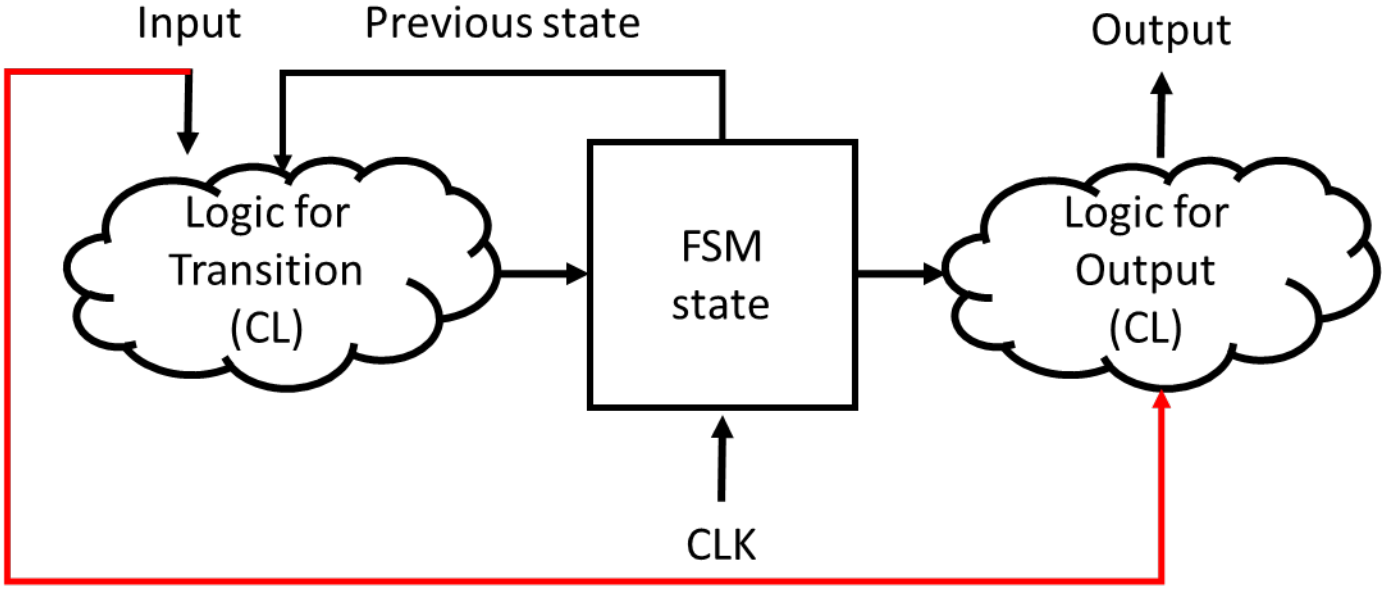
Moore Machine

- Output: function of only the current state
- **Synchronous** machine: the outputs are changed only when the state is changed.



Mealy Machine

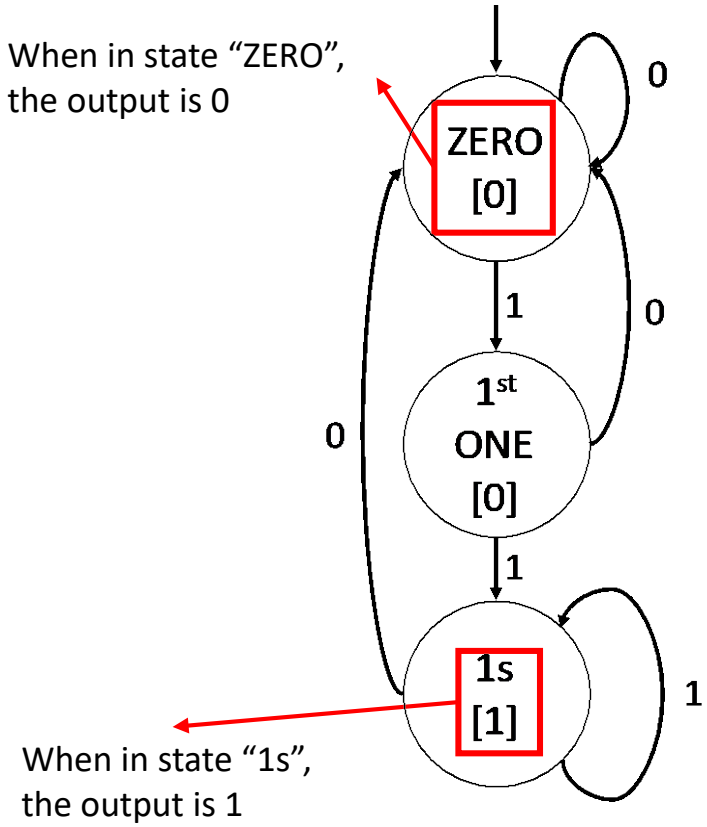
- Output: function of the current state and **inputs**
- **Asynchronous** machine: the outputs are changed as soon as the inputs are changed.



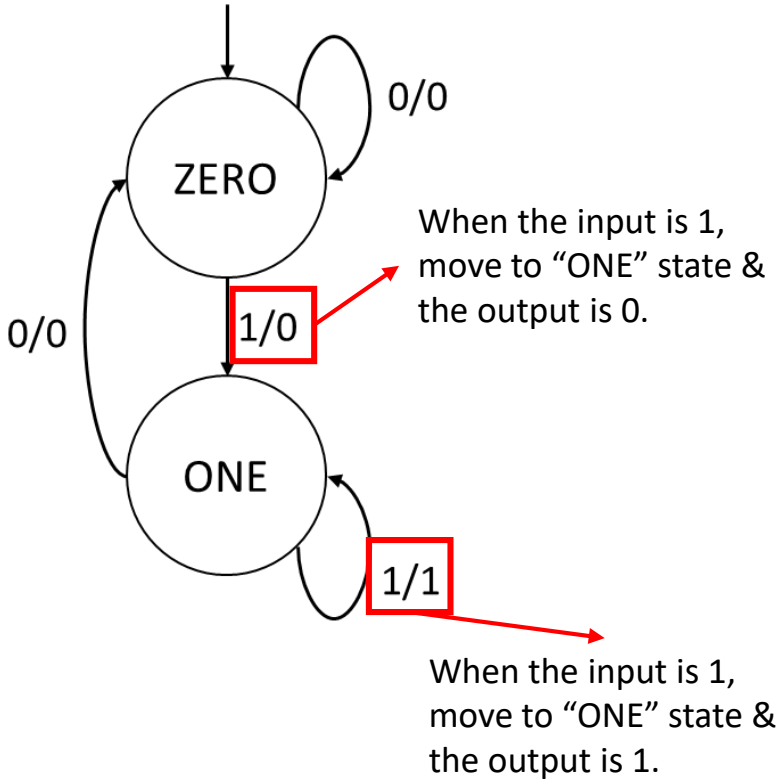
Example: Reduce 1s

- Input: binary stream
- Output: Convert first 1 to 0 when a stream of 1s is detected.
- Example:
  - 000011100 -> 000001100
  - 010101010 -> 000000000
  - 011001100 -> 001000100

Moore vs. Mealy



VS



Moore
(3 states)

Mealy
(2 states)

Moore vs. Mealy
State declaration & update

```
Module Moore (input CLK, input in,
output reg out)

   parameter zero=0, firstOne=1, ones=2;

  reg [1:0] state;
  reg [1:0] nextState;

  initial begin
       state <= zero;
  end

  always @(posedge CLK)
      begin
         state <= nextState;
      end
```

2 bits vs 1 bit

2 D-FFs

vs

```
Module Mealy (input CLK, input in, output
reg out)

   parameter zero=0, one=1;

  reg state;
  reg nextState;

  initial begin
       state <= zero;
  end

  always @(posedge CLK)
      begin
         state <= nextState;
      end
```

1 D-FF

Moore

Mealy

Moore vs. Mealy
Output & state logic

```
always @(in or state)
    begin
      case (state)
        zero:
        begin
          if (n == 1) nextState <= firstOne;
          else nextState <= zero;
         end
        firstOne:
        begin
          if (n == 1) nextState <= ones;
          else nextState <= zero;
         end
        ones:
        begin
          if (n == 1) nextState <= ones;
          else nextState <= zero;
         end
        default: nextState <= zero;
      endcase
    end
```

State logic

```
always @(state)
    begin
      case (state)
            zero: out <= 0;
            firstOne: out <= 0;
            ones: out <= 1;
            default: out <= 0;
      endcase
    end
```

Output logic

VS

Output logic

```
always @(in or state)
    begin
      case (state)
        zero:
        begin
          out <= 0;
          if (n == 1) nextState <= one;
          else nextState <= zero;
         end
        one:
        begin
            if (n == 1) begin
              out <= 1;
              nextState <= one;
            end else begin
              out <= 0;
              nextState <= zero;
            end
        end
      endcase
    end
```

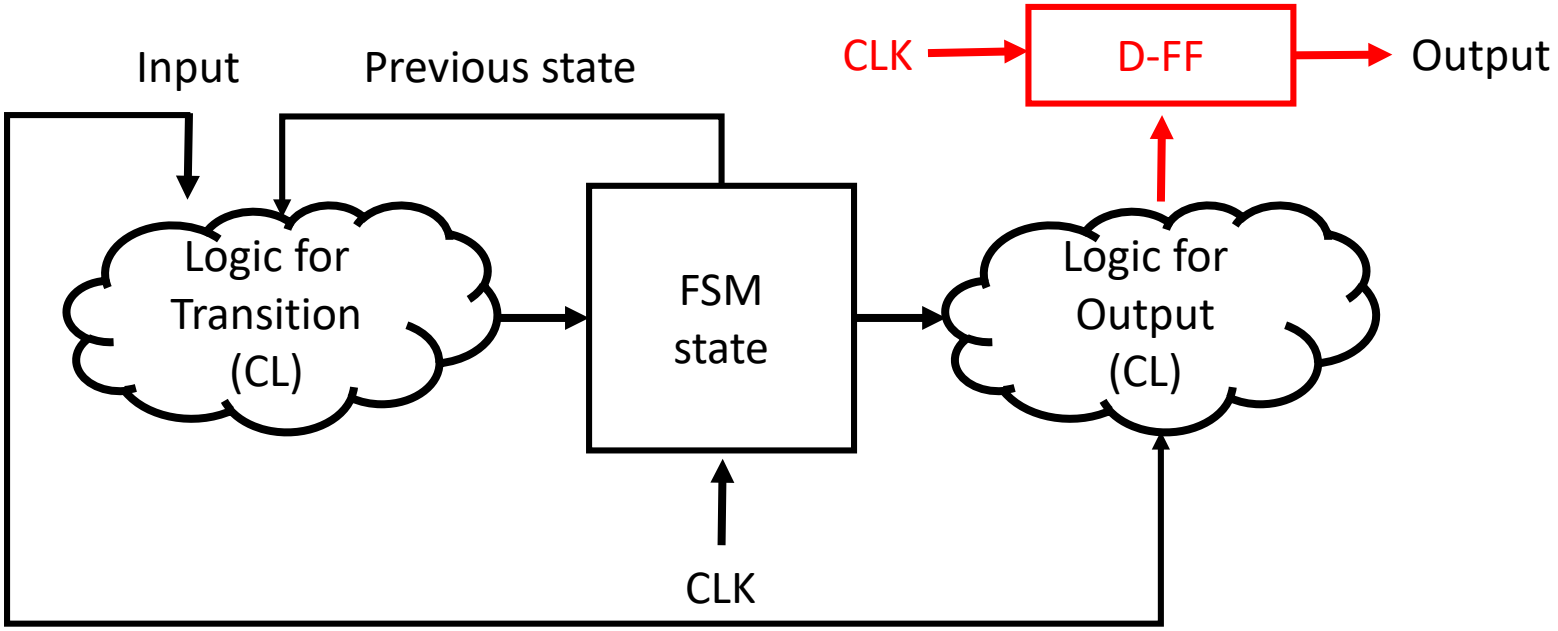
State & output logic

Moore

Mealy

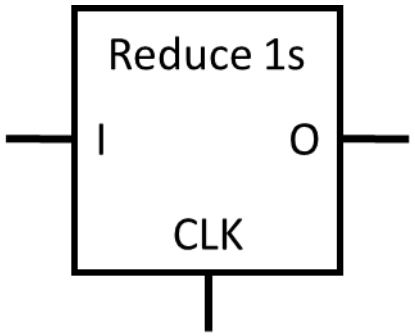
Synchronous Mealy Machine

- If you want to get the synchronized output in a Mealy Machine, buffer the output logic's value using D-FF.

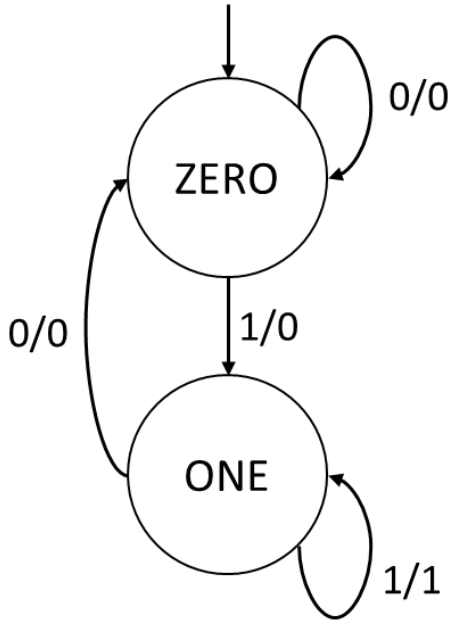


Step by Step Programming (1)

- Understand I/Os and the logic's functionality.



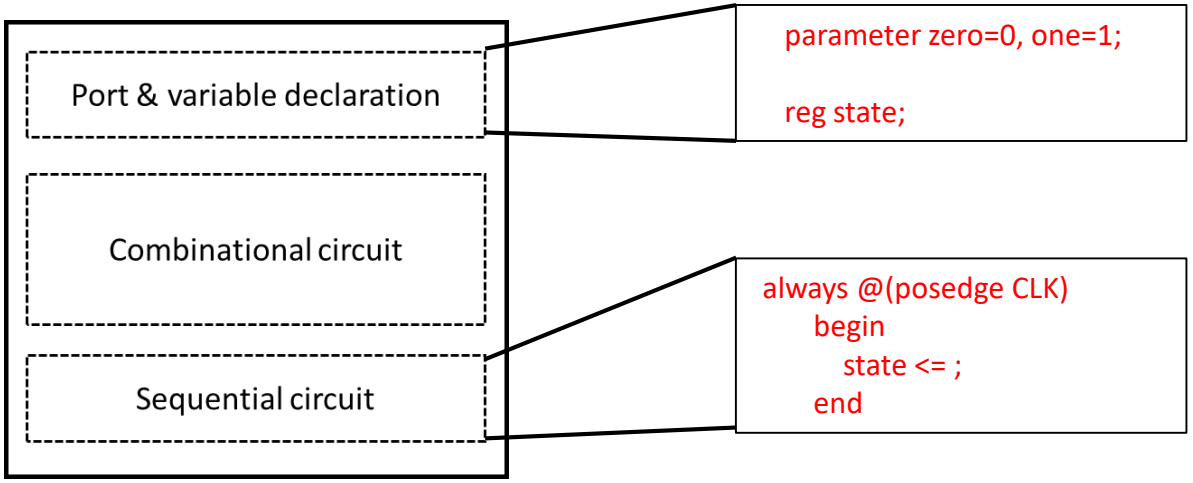
I/O



Functionality

Step by Step Programming (2-3)

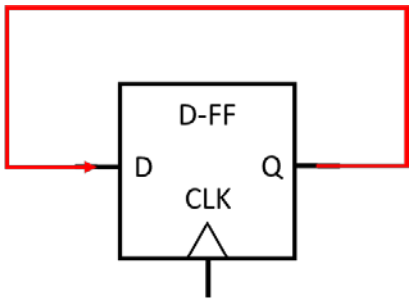
- (2) Determine what to be saved in the logic.
  - FSM: 2 states => 1 bit registers
- (3) What to be synchronized to the clock?
  - FSM: The state update should be synchronized to the positive edge of the clock.



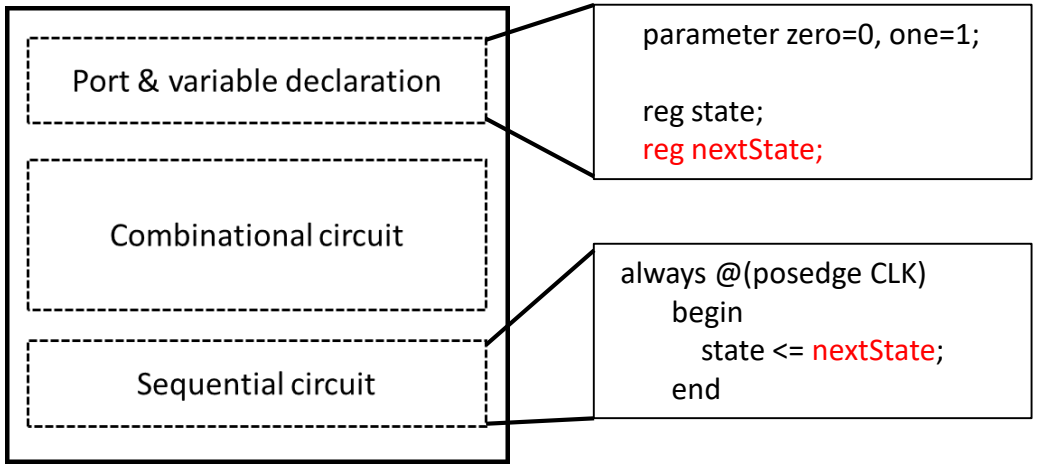
Result after (2-3)

Step by Step Programming (4)

- Think about how the data move between registers; that is, think about ***data flow*** .



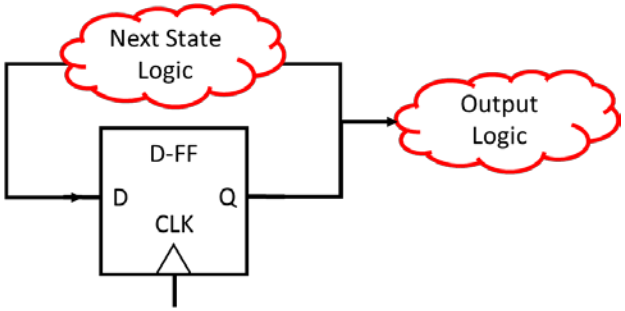
D-FF's data moves from D-FF to D-FF (i.e., same FF).



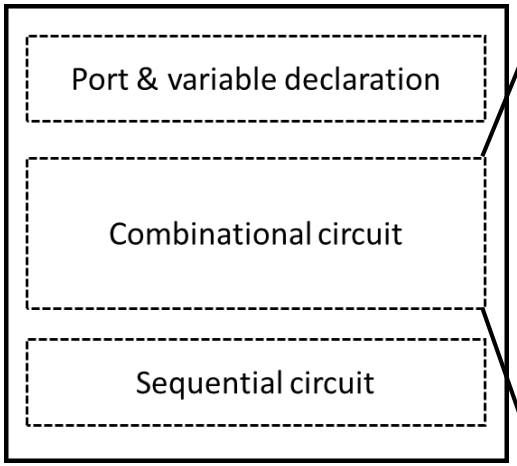
Result after (4)

Step by Step Programming (5)

- Which operations should be done during a data movement?
  - Next state and output should be computed.



Determine specific operations for each logic



Result after (5)

```
always @(in or state)
    begin
      case (state)
        zero:
        begin
          out <= 0;
          if (n == 1) nextState <= one;
          else nextState <= zero;
        end
        one:
        begin
                if (n == 1) begin
                  out <= 1;
                nextState <= one;
                end else begin
                  out <= 0;
                nextState <= zero;
                end
        end
      endcase
    end
```

Assignment #2:
010 Detector & Register File

Assign 2-1: 010 Detector

- Goal: Implement a synchronous 010 detector
  - Input: 1 bit data, 1 bit clock signal
  - Output: 1 if 010 is detected, 0 otherwise
  - Data is given to the detector 1 bit per cycle
  - Example sequence:
    - Input: 00010101001010000011111
    - Output: 00001010100101000000000

- How to start?
  - First, draw an FSM (Moore or Mealy).
  - Then, implement the FSM following the steps in the previous slides.

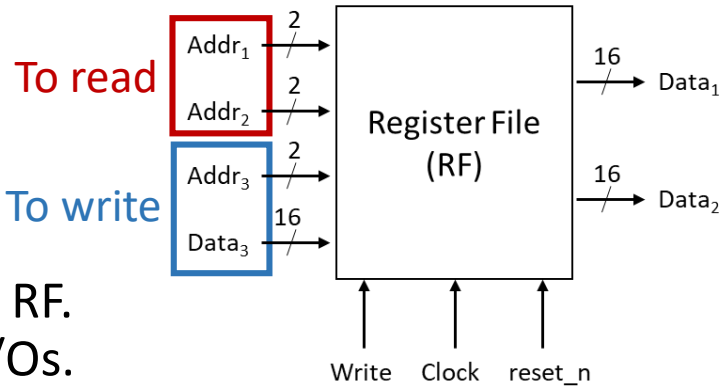
Assign 2-1: Register File (RF)

- Registers are the fundamental components of CPUs, and they have data to be computed or the state of the CPU.
  - E.g., $v0, $a1, $t4, $s3, $k0, and $ra in MIPS
- A register file is a **collection of registers**.

* You will use RF in the assign 2-2.

Assign 2-2: Register File

- Goal: Implement a **16-bit 2-read/1-write register file**
  - Input: write signal (1 bit), clock signal (1 bit), reset_n (1bit), three addresses (each 2 bits), one data to write (16 bits),
  - Output: two read data (each 16 bits)
  - Functionality:
    - When reset_n is low (i.e., negative logic), set all the register values to 0
    - $Data_x$ is for $Addr_x$
    - Write $Data_3$ to the register specified $Addr_3$ only when "Write" signal is High.
    - Read data to $Data_x$ from the register specified by $Addr_x$ regardless of write and clock (x = 1 or 2)
  - Q: how many registers in this RF?



- How to start?
  - Understand the I/Os and operations of RF.
  - Make a module for RF and specify its I/Os.

Note 1)

- Do NOT consider Verilog programming (or other HDL) as a variation of C programming.
  - The purpose of HDL is "**describing a circuit**".
  - The result of the HDL programming is the circuit itself, so think about how your codes would be when it becomes a real hardware (e.g., FF, latch, gates)
  - (c.f., SW programming is for procedures.)

- "reg" type
  - **"reg" does not mean either "register"** in CPU or flip-flops. It can be equivalent to even "wire" based on its usage.
  - When a "reg" is synchronized to a clock, it becomes FFs, or equivalent to FFs.

Note 2)

- Module
  - Make your module **as small as possible**.
  - Connect each small module to make a bigger module (i.e., bottom-up fashion).
  - Verify each module separately.

- Inter-module
  - Be careful on **endians** and **overflows**
  - wire [15:0] a != wire [0:15] a

```
module A()
   wire b_out;
   B submodule(b_out); // you connected wrongly
endmodule

module B(output wire [2:0] out);
   assign out = 3b'11;
endmodule
```

- Inside module
  - Separate "always block" as much as possible based on the functionality.

- Thanks