

Computer Organization

Sequential Logic & RTL

Wonsik Lee

Ilkwon Byun, Dongmoon Min, Heetaek Jeong, Seungho Lee

High Performance Computer System (HPCS) Lab

March 25, 2019

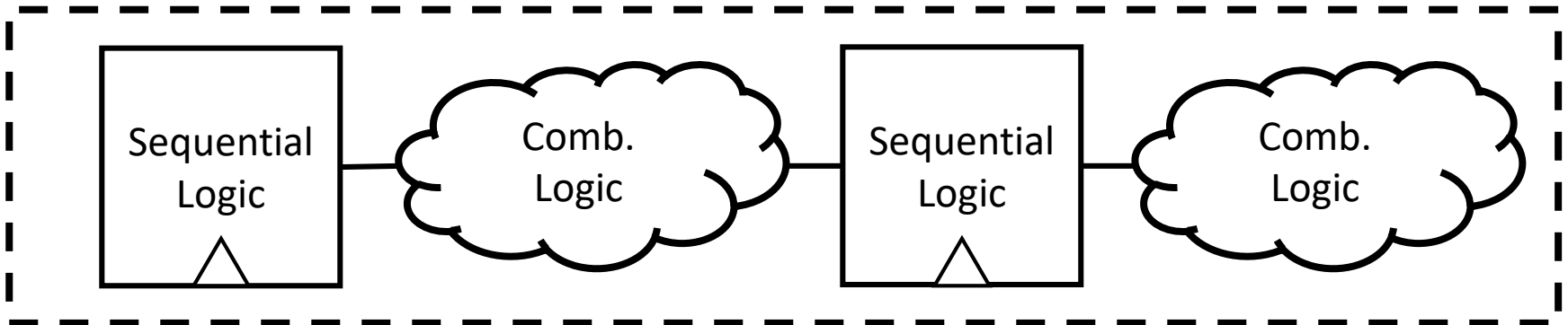
Objective

- 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**.
 - $\text{Outputs} = f(\text{inputs})$

Sequential Logic (SL)

- Outputs of a logic are **a function of inputs & states**.
 - $\text{Outputs} = f(\text{inputs}, \text{state})$
- 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, w2, w3);  
endmodule
```

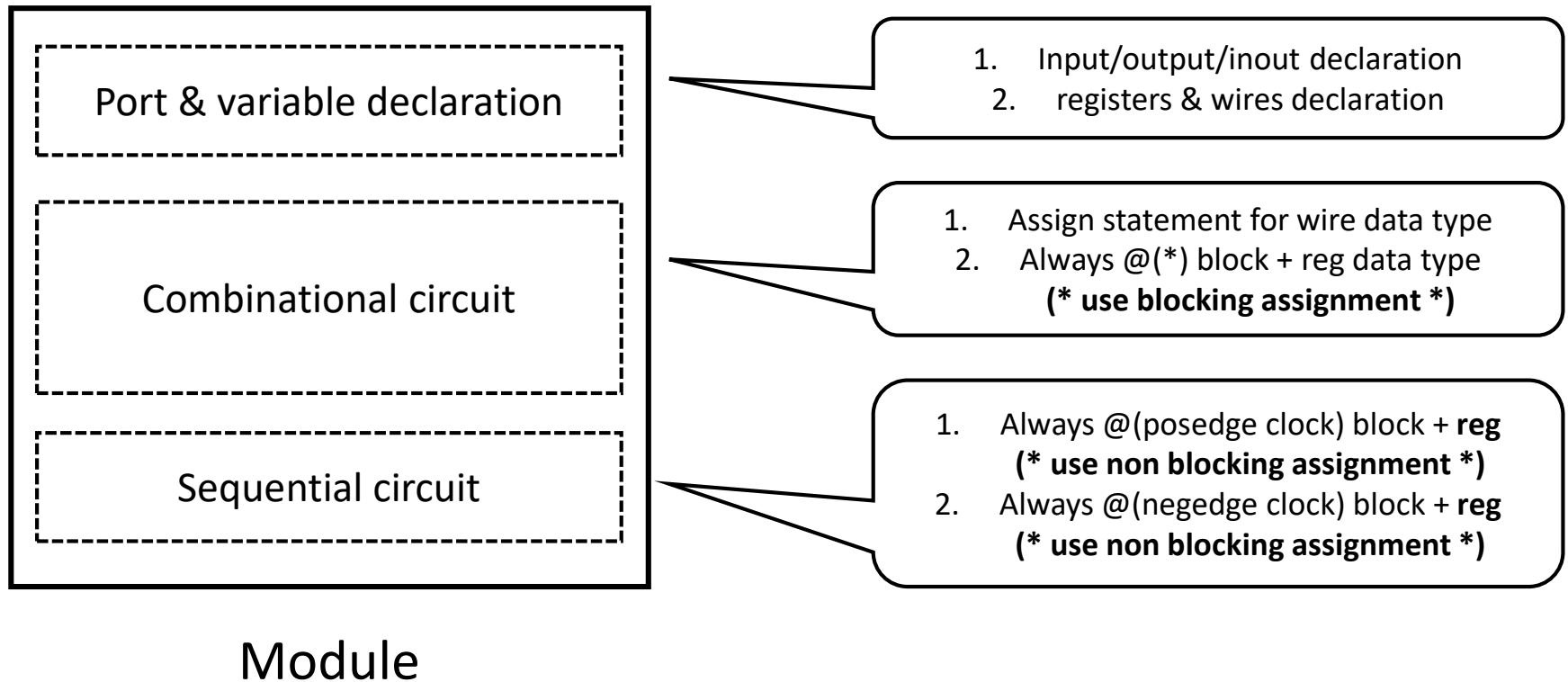
Gate Level

VS

```
module FullAdder_RTL (A,B,S,C);  
  input A, B;  
  output S, C;  
  
  assign S = A ^ B & Cin;  
  assign Cout = (A & B) | (Cin & (A ^ B));  
endmodule
```

RTL

General RTL Structure



RTL Programming Guide

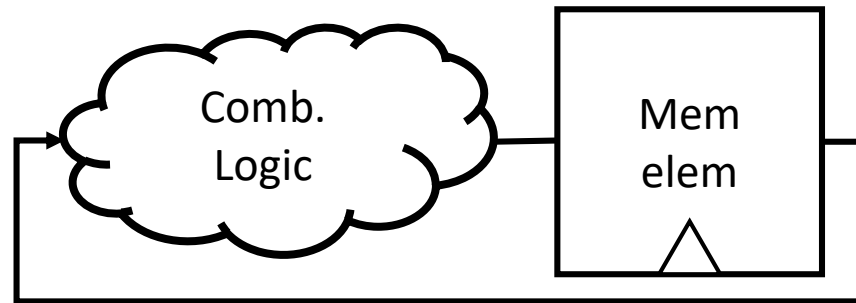
- How to make an RTL code from the given block diagram?
- (1) Understand inputs, outputs, and the functionality of it.
- (2) Determine what to be saved in the registers.
- (3) Think which values should be synchronized to the clock.
- (4) Think about how the data move between the registers.
- (5) Consider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

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

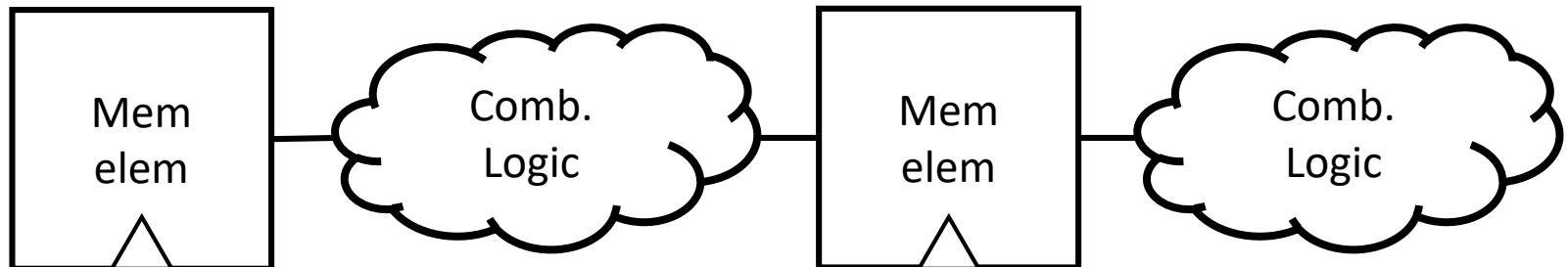
How to write a sequential logic in Verilog

Two Possible SLs in Verilog

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

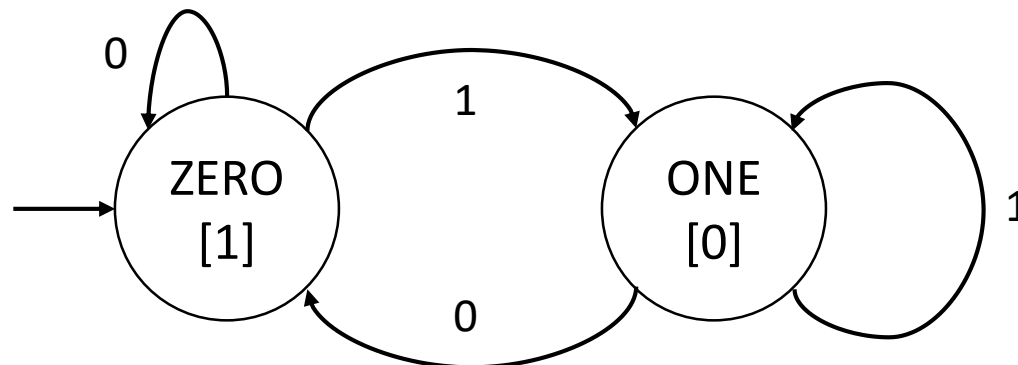


- Pipeline
 - Straight forward: connect each logic using mem elem.



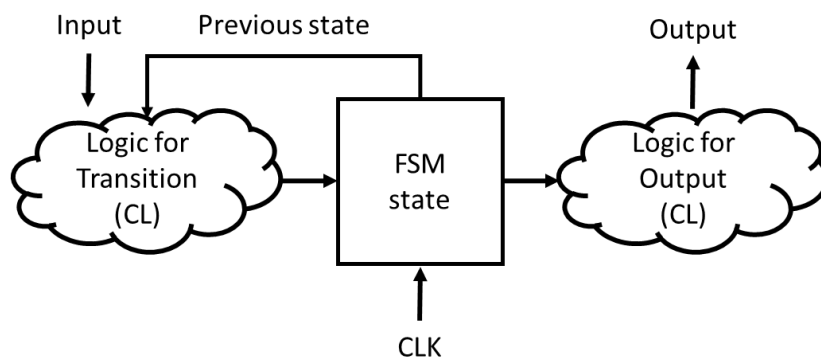
Finite State Machine (FSM)

- 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.

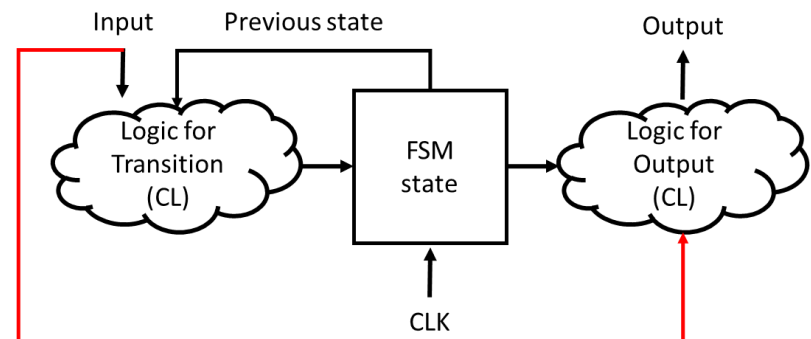


Two Ways to Represent FSM

- 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.



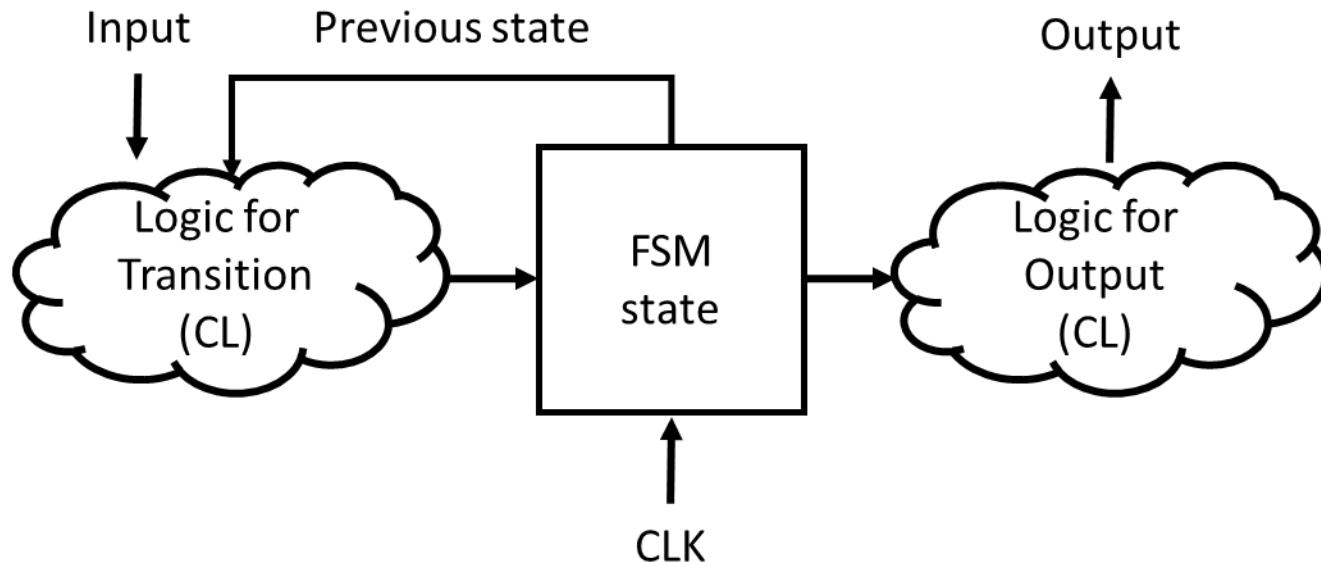
Moore Machine



Mealy Machine

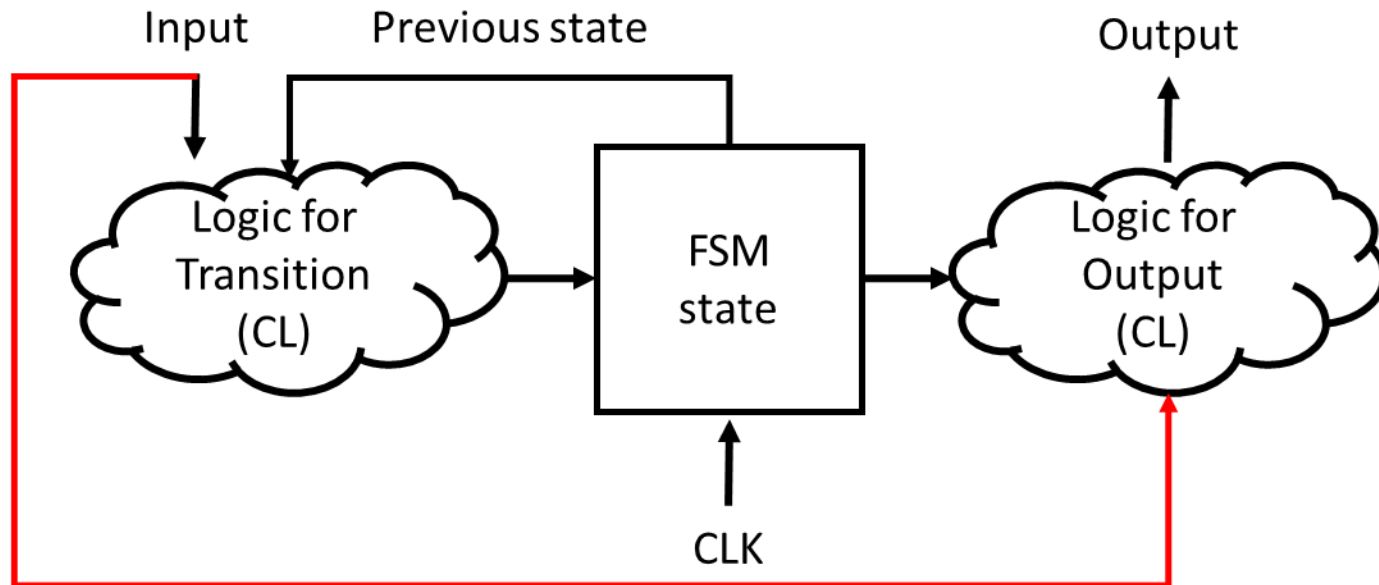
Moore Machine

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

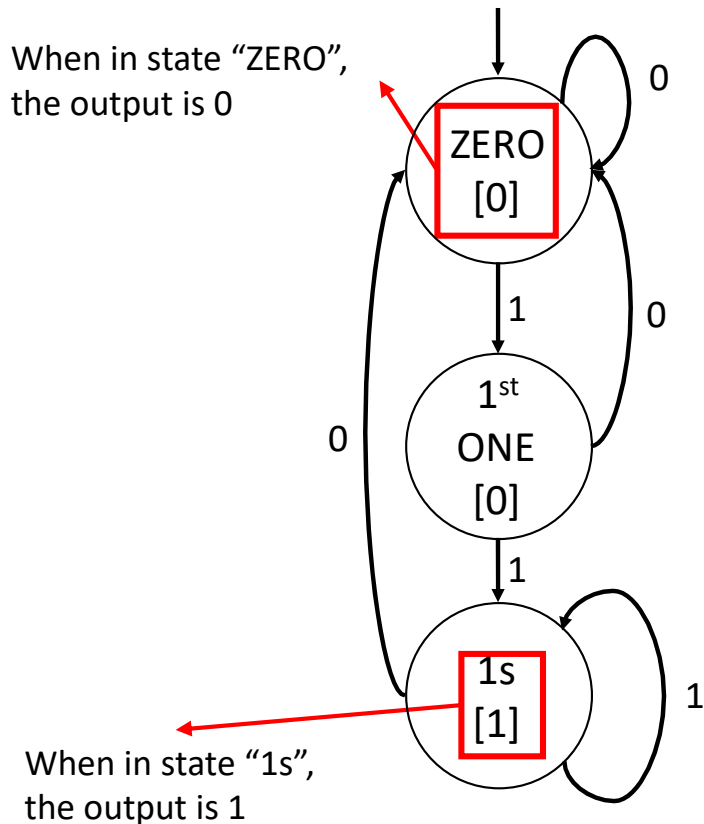


Mealy Machine

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

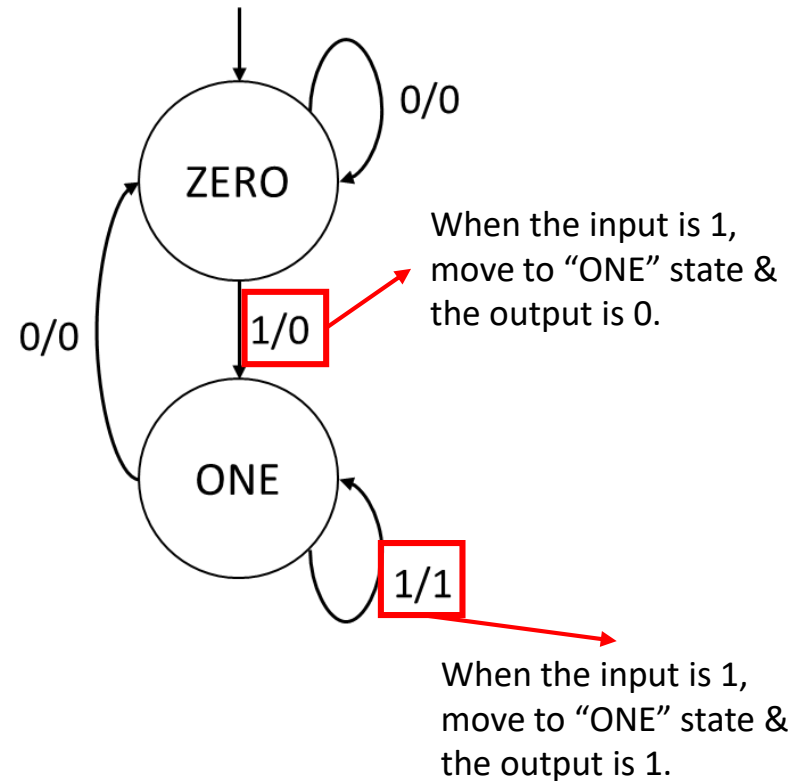


Moore vs. Mealy



Moore
(3 states)

VS



Mealy
(2 states)

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: 0001010100101000011111
 - Output: 0000101010010100000000
- How to start?
 - First, draw an FSM (Moore or Mealy).
 - Then, implement the FSM following the steps in the previous slides.

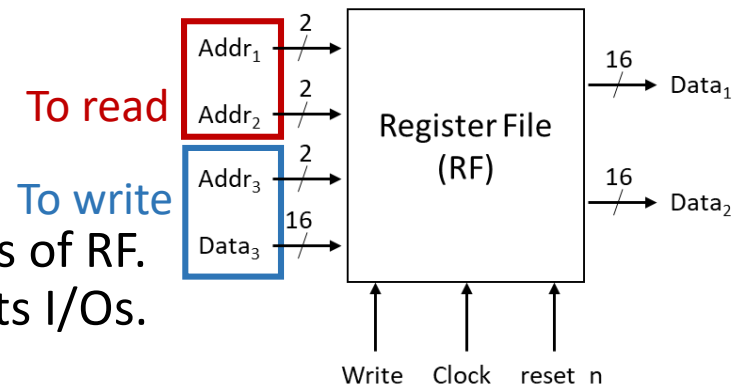
Assign 2-1: Register File (RF)

- 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 Data₃ to the register specified Addr₃ 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)

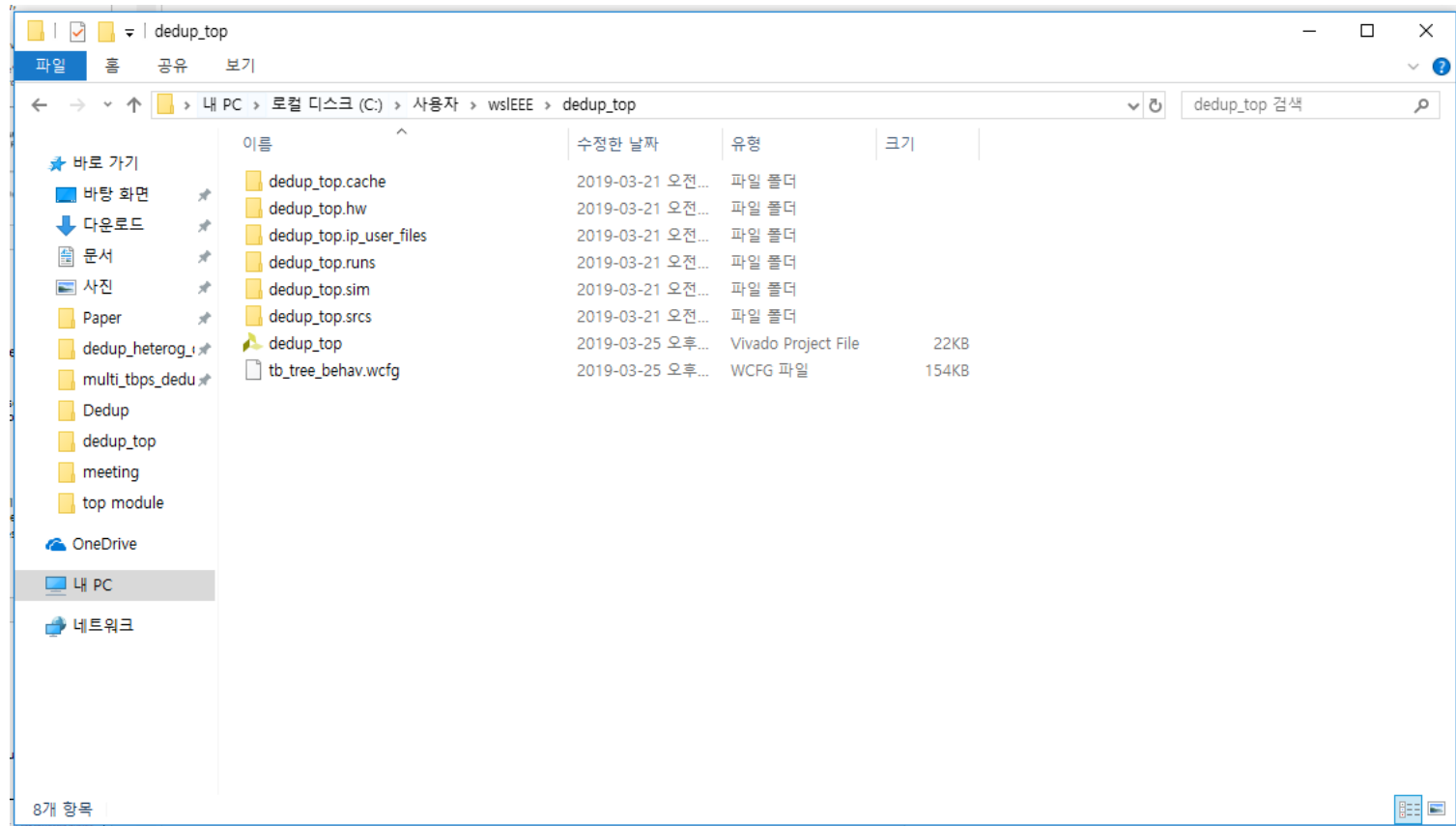
- Do NOT consider Verilog programming (or other HDL) as a variation of C programming.
 - The purpose of HDL is “**describing a circuit**”.
 - The result of the HDL programming is the circuit itself, so think about how your codes would be when it becomes a real hardware (e.g., FF, latch, gates)
 - (c.f., 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
- Inside module
 - Separate “always block” as much as possible based on the functionality.

```
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
```

Note 3)



Note 4)

