CO002
Computer Hardware Experiment

Lecture 08: Finite State Machine

Liang Yanyan

澳門科技大學
Macau of University of Science and Technology

Finite State Machine (FSM)

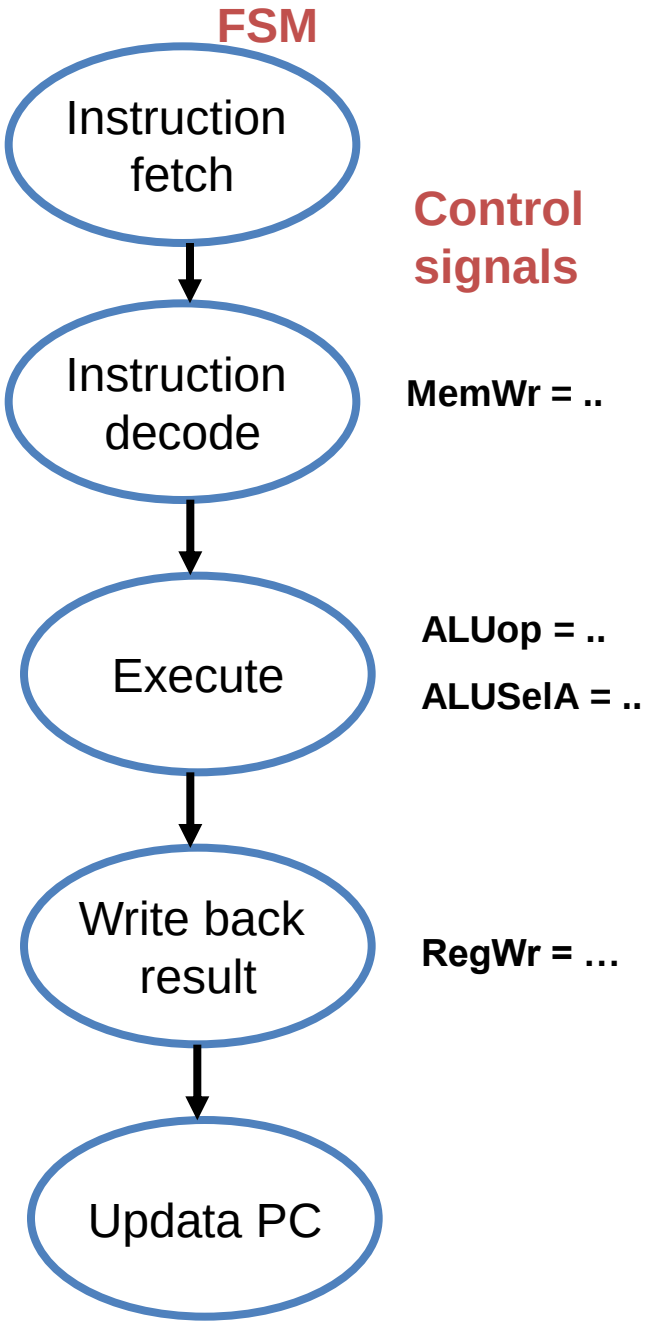
- A finite state machine has
  - A set of inputs
  - A set of outputs
  - A set of states
  - A function that maps states and inputs to outputs
  - A function that maps states and inputs to states -> state transition function
  - A description of the initial state
- FSM is very useful in designing certain types of systems, e.g. controller.

Control FSM of R-type

- R-type
  - Instruction fetch
  - Instruction decode, load register
  - Execute
  - Write back result to register
- Can we design a finite state machine representing these states? And output different control signal at different states to control the operation of datapath?

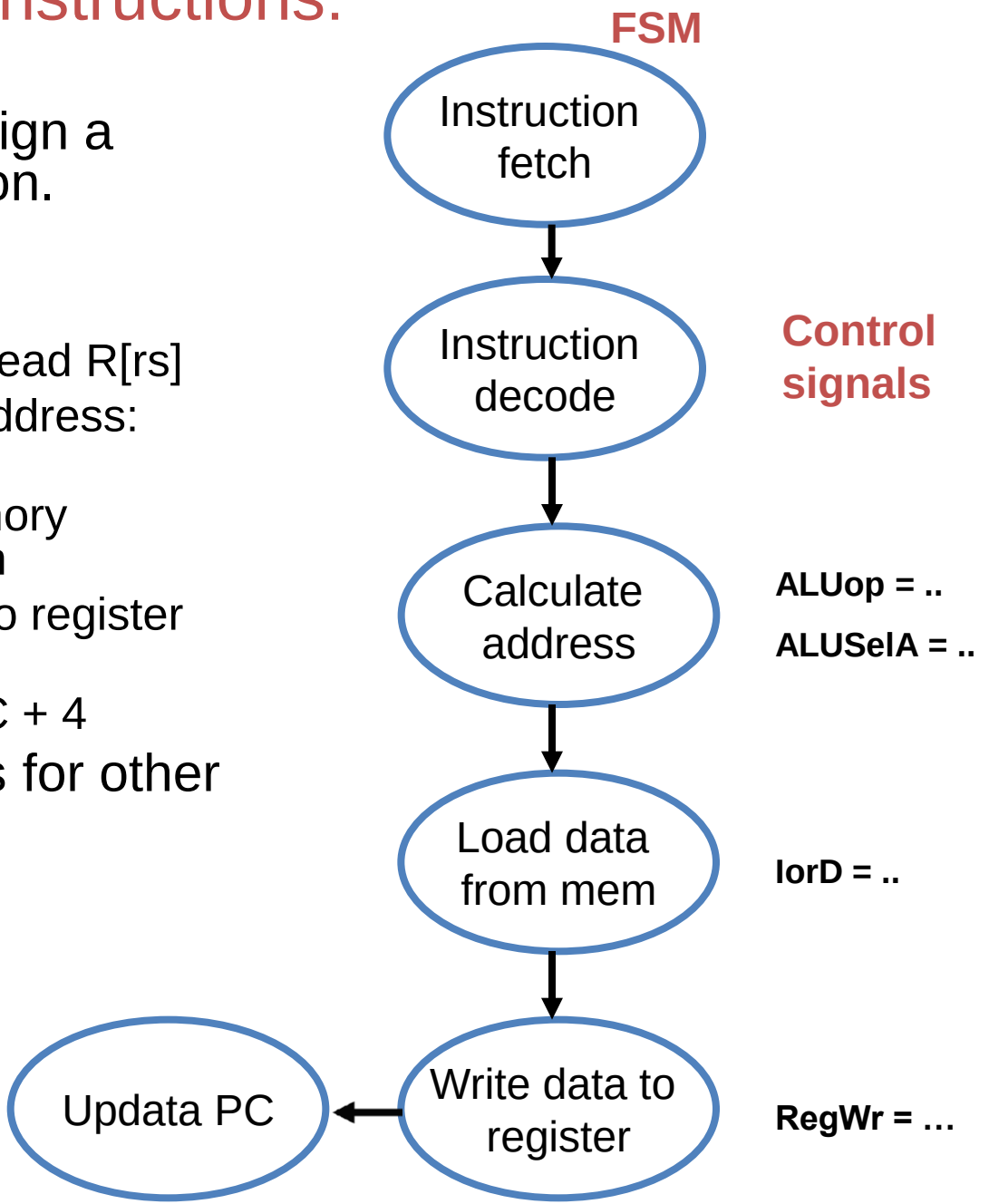
Steps to execute instructions: R-type

- R-type, e.g. add $t0, $t1, $t2
  - Instruction fetch
  - Instruction decode, read R[rs] and R[rt]
  - Execute instruction: R[rs] op R[rt]
  - Write result back to register R[rd]
  - Update PC: PC = PC + 4
- Can we design a finite state machine to represent these states? And generate different control signals at different states to control the operation of the datapath?
- E.g. design a FSM with 5 states: IF, ID, EX, WB, PC. Each state outputs appropriate control signals to control the datapath to finish adequate operation in a certain state.



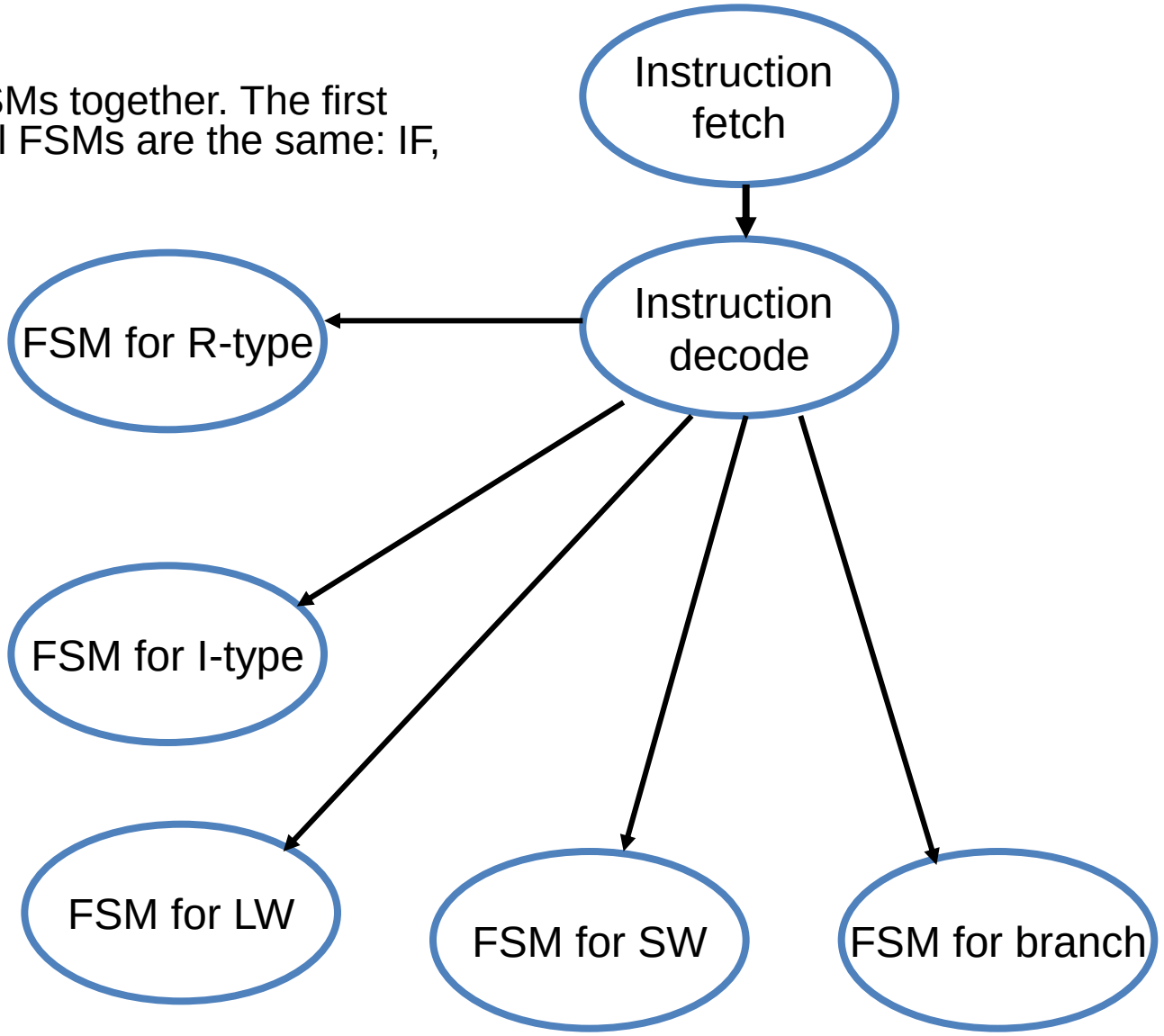
Steps involved in instructions: LW

- Similarly, we can design a FSM for LW instruction.
- Load word
  - Instruction fetch
  - Instruction decode: read R[rs]
  - Calculate memory address: R[rs] + Imm
  - Load data from memory address: R[rs] + Imm
  - Write memory data to register R[rt]
  - Update PC: PC = PC + 4
- We can design FSMs for other instructions.



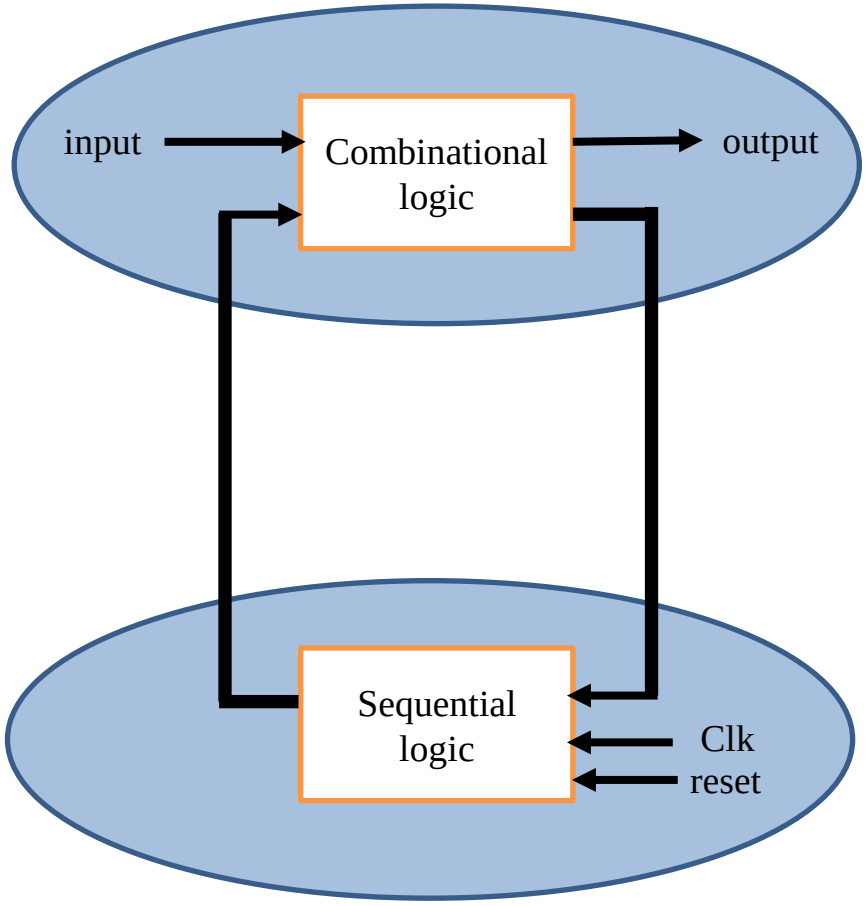
Control FSM for the multi-cycle processor

- Combine all FSMs together. The first two states of all FSMs are the same: IF, ID.



Design of FSM

- Combinational logic: implement state transition function
  - Can be implemented using “process” statement.
- Sequential logic: change state every clock cycle.
- Output depends on input and pr_state -> Mealy
- Output depends on pr_state only -> Moore
- pr_state: present state
- nx_state: next state



Design the sequential part

```
PROCESS
Begin
        IF (reset = '1') THEN
                pr_state <= initial_state;
        ELSIF (Clk'EVENT AND Clk='1') THEN
                pr_state <= nx_state;
        END IF;
END PROCESS;
```

Design the combinational part

```
PROCESS (input, pr_state)
Begin
        CASE pr_state IS
                WHEN state0 =>
                        IF (input = “01”) THEN
                                output <= ‘111’;
                                nx_state <= state1;
                        ELSIF …
                        END IF;
                WHEN state1 =>
                        IF (input = “10”) THEN
                                output <= ‘001’;
                                nx_state <= state2;
                        ELSIF …
                        END IF;
                …
                END CASE;
                statenum <= pr_state;

END PROCESS;
```

← Mealy

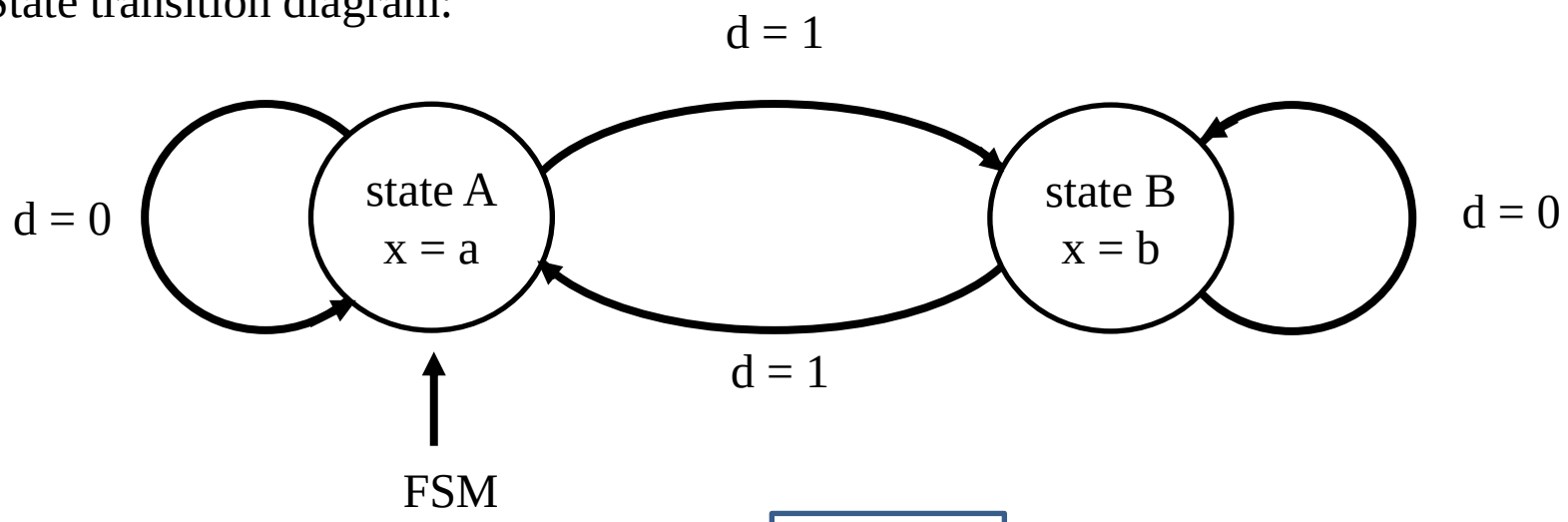
Design the combinational part

```
PROCESS (input, pr_state)
Begin
        CASE pr_state IS
                WHEN state0 =>
                        output <= '111';
                        IF (input = "01") THEN
                                nx_state <= state1;
                        ELSIF …
                        END IF;
                WHEN state1 =>
                        output <= '001';
                        IF (input = "10") THEN
                                nx_state <= state2;
                        ELSIF …
                        END IF;
                …
                END CASE;
END PROCESS;
```

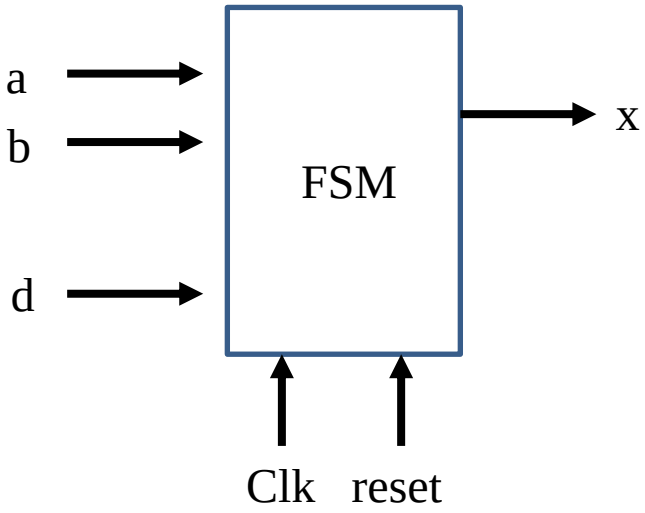
Moore

Example

State transition diagram:



The FSM



Entity

```
ENTITY simple_fsm IS
PORT (
        a, b, d, Clk, reset: IN std_logic;
        x: OUT std_logic);
END simple_fsm;
```

Architecture

```
ARCHITECTURE simple_fsm_ar OF simple_fsm IS

signal pr_state: std_logic_vector(1 downto 0);
signal nx_state: std_logic_vector(1 downto 0);

BEGIN
        combinational part
        sequential part

END simple_fsm_ar;
```

Sequential part

```
PROCESS (reset, Clk)
BEGIN
        IF (reset = ‘1’) THEN
                pr_state <= “00”;
        ELSIF (Clk’EVENT AND Clk=‘1’) THEN
                pr_state <= nx_state;
        END IF;
END PROCESS;
```

Combinational part

```
PROCESS (a, b, d, pr_state)
BEGIN
        CASE pr_state IS
                WHEN “00” =>
                        x <= a;
                        IF (d = ‘1’) THEN
                                nx_state <= “01”;
                        ELSE
                                nx_state <= “00”;
                        END IF;
                WHEN “01” =>
                        x <= b;
                        IF (d = ‘1’) THEN
                                nx_state <= “00”;
                        ELSE
                                nx_state <= “01”;
                        END IF;
                WHEN others =>
                        NULL;        --No action here
                END CASE;
END PROCESS;
```