# Digital System Design Lab

## Lab 12
## Finite State Machines

Student ID: D1166506

Name: 周嘉禾

Date: 2023/12/13

## 1. Objectives

- To learn how to build finite state machine in verilog

## 2. Theorem

**Finite-state machines, including Moore and Mealy machines, are models used in digital systems to represent sequential logic. These machines are composed of a finite number of states and transitions between these states based on inputs.**

**(1) Moore Machine:**

**A Moore machine is a type of finite-state machine where the outputs are determined solely by the current state. Each state directly corresponds to a specific output value. Transitions between states occur based on inputs, and once in a state, the output remains constant until a transition to another state is triggered by an input signal. Moore machines are valuable in various applications requiring clear and predictable state-dependent outputs, such as digital circuit design, control systems, and protocol implementations.**

**(2) Mealy Machine:**

**Contrasting with Moore machines, Mealy machines are another form of finite-state machines where outputs not only depend on the current state but also on the inputs. The output values are determined by both the present state and the input signals, resulting in potentially different output values for the same state based on different input signals. Mealy machines are commonly used in applications requiring outputs that are influenced by both the current state and external inputs, offering flexibility in modeling sequential behaviors in digital systems.**

# 3. Experimental Results

## (1) Step 1

### a. *Think*

i. If reset, reset state to S0

ii. If clock, change state to next state

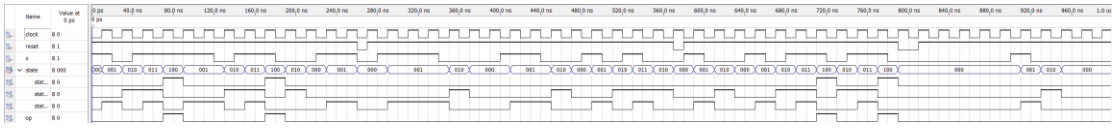
iii. Next state will alter according to x

iv. Light op if state is S4

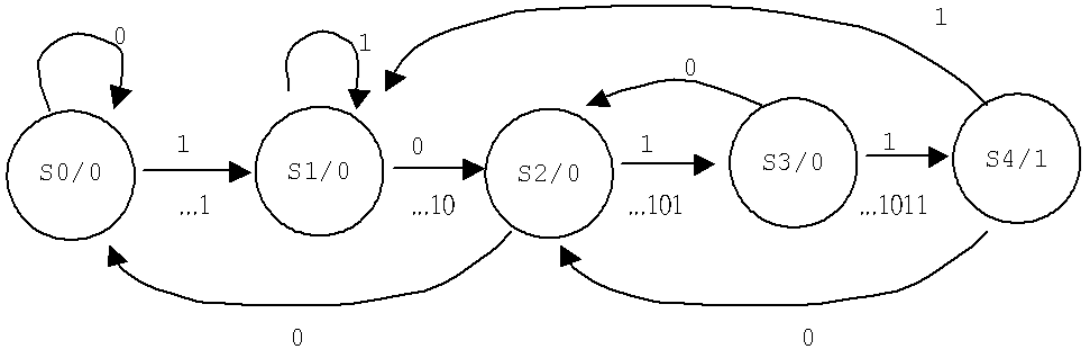
### b. *Code*

```
module step_1(x, clock, reset, state, op);
	input x, clock, reset;
	output reg[2:0] state;
	output reg op;
	parameter S0=3'b000, S1=3'b001, S2=3'b010, S3=3'b011,
S4=3'b100;

	reg [2:0] next_state;

	// State Register
	always @(posedge clock or negedge reset) begin
		if (!reset)
			state = S0;
		else
			state = next_state;
	end

	// Next State
	always @(x or state) begin
		case (state)
			S0: next_state = (x==1) ? S1 : S0;
			S1: next_state = (x==1) ? S1 : S2;
			S2: next_state = (x==1) ? S3 : S0;
			S3: next_state = (x==1) ? S4 : S2;
			S4: next_state = (x==1) ? S1 : S2;
		endcase
	end

	// Output
	always @(state) begin
		case (state)
			S4: op = 1'b1;
			default: op = 1'b0;
		endcase
	end
endmodule
```

### c. *Simulation*



*d. State Diagram*




**(2) Step 2**

*a. Think*

i. If reset, reset state to S0

ii. If clock, change state to next state

iii. Next state will alter according to w

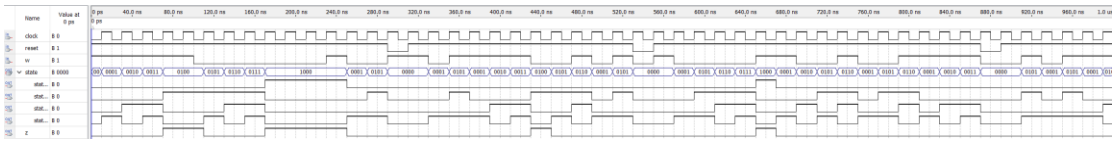
iv. Light op if state is S4 or S8

*b. Code*

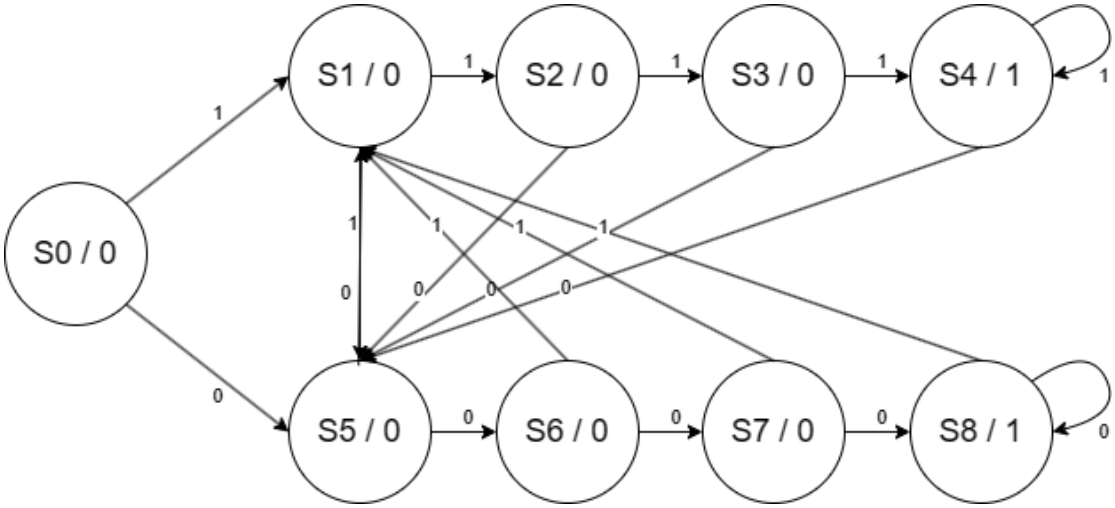
```
module step_2(w, clock, reset, state, z);
	input w, clock, reset;
	output reg[3:0] state;
	output reg z;
	parameter S0=4'b0000, S1=4'b0001, S2=4'b0010, S3=4'b0011,
                          S4=4'b0100, S5=4'b0101, S6=4'b0110,
S7=4'b0111, S8=4'b1000;

	reg [3:0] next_state;

	// State Register
	always @(posedge clock or negedge reset) begin
		if (!reset)
			state = S0;
		else
			state = next_state;
	end

	// Next State
	always @(w or state) begin
		case (state)
			S0: next_state = (w==1) ? S1 : S5;
			S1: next_state = (w==1) ? S2 : S5;
			S2: next_state = (w==1) ? S3 : S5;
			S3: next_state = (w==1) ? S4 : S5;
			S4: next_state = (w==1) ? S4 : S5;
			S5: next_state = (w==0) ? S6 : S1;
			S6: next_state = (w==0) ? S7 : S1;
			S7: next_state = (w==0) ? S8 : S1;
			S8: next_state = (w==0) ? S8 : S1;
		endcase
	end
```

```
        // Output
        always @(state) begin
                case (state)
                        S4, S8: z = 1'b1;
                        default: z = 1'b0;
                endcase
        end
endmodule
```

c. *Simulation*



d. *State Diagram*




## 4. Comments

None

## 5. Problems & Solutions

None

## 6. Feedback

None