

EECS 151/251A

SP2022 Discussion 3

GSI: Yikuan Chen, Dima Nikiforov

Agenda

- Administrative
- Finite State Machines (FSM)
- RISC V ISA

Administrative

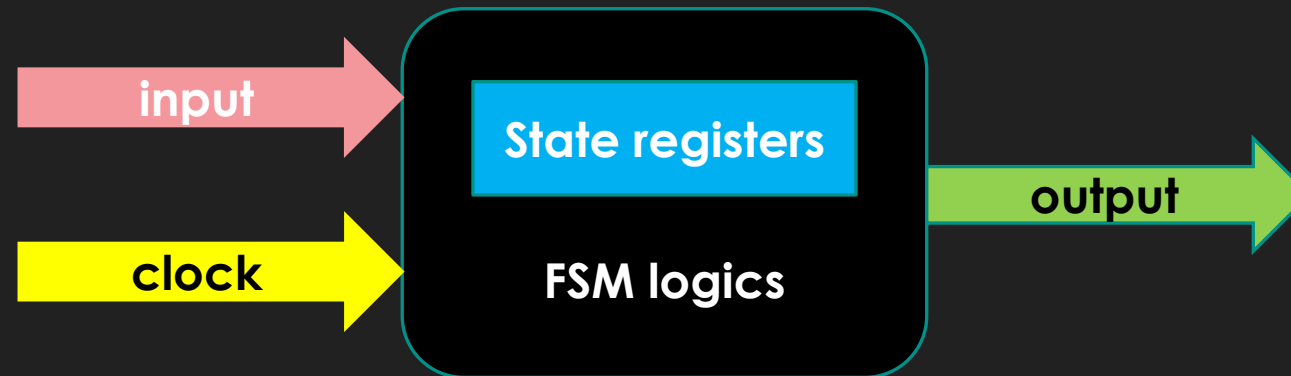
- Lab 1, 2, 3 checkoff has **one more week** of extension now
 - If your lab is on Wed/Thur, lab 1,2,3 are due next week.
 - If your lab is on Monday, lab 1,2,3 are due on Monday the week after next
- Piazza discussions – We now have megathread for each lab
 - Use this thread to benefit others as well
 - ...without disclosing your code 😊



Finite State Machine

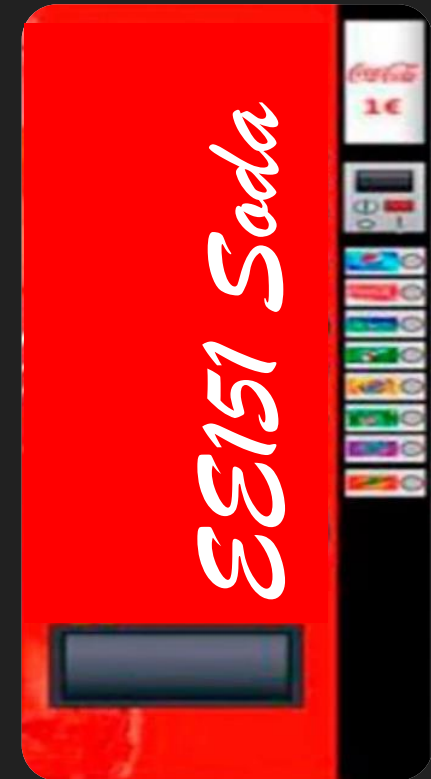
Finite State Machine

- **State** is nothing but a **stored value** of a signal, usually internal, but you could choose to make it visible to the outside.
- **State register** is the physical circuit element that stores the **state value**.
- **FSM** is a type of sequential (a.k.a. **clocked**) logic circuit whose **output** signal values depend on **state** (and/or **input** as well).



Finite State Machine in Real Life

- E.g. Vending machine
 - Dispenses a soda if it receives **at least 25 cents**
 - Doesn't return change or rollover to next purchase
 - Customer can **insert three different coins**:
 - Quarter – 25¢ – Q
 - Dime – 10¢ – D
 - Nickel – 5¢ – N



Vending Machine

Verilog Realization

Vending Machine Skeleton

```
module vending_machine(  
    input clk, rst,  
    input Q, D, N, // Quarter(25¢), Dime(10¢), Nickle(1¢)  
    output dispense  
);  
//①define state registers  
  
//②define state names as local params(optional but recommended)  
  
//③an always@(posedge clk) block to handle state assignment  
  
//④an always@(*) block to handle output for each state and state transition logic  
(both of them may also depend on input)
```

Verilog Realization

```
module vending_machine(  
    input clk, rst,  
    input Q, D, N,  
    output dispense  
);  
    //①  
    reg [2:0] NS, CS; //next_state, current_state  
    //② enumerate all states  
    localparam S0 = 3'd0, S5 = 3'd1, S10 = 3'd2, S15 = 3'd3, S20 = 3'd4, S25 = 3'd5;  
  
    //③  
    always @(posedge clk) begin  
        if (rst) CS <= S0;  
        else CS <= NS; //in each cycle , we assign current state to be next state  
    end
```


④ Mealy vs Moore

```
reg dispense;
always @(*) begin
    NS = CS;
    dispense = 1'b0;
    case (CS)
        S0: begin
            if (Q == 1'b1) begin
                NS = S0;
                dispense = 1'b1;
            end
            if (D == 1'b1) NS = S10;
            if (N == 1'b1) NS = S5;
            end
        ...
        S15: begin
            if (Q == 1'b1) begin
                NS = S0;
                dispense = 1'b1;
            end
            if (D == 1'b1) begin
                NS = S0;
                dispense = 1'b1;
            end
            if (N == 1'b1) NS = S10;
            end
        ...
        default: NS = S0;
    endcase
end
```

```
wire dispense;
always @(*) begin
    NS = CS;
    case (CS)
        S0: begin
            if (Q == 1'b1) NS = S25;
            if (D == 1'b1) NS = S10;
            if (N == 1'b1) NS = S5;
        end
        S5: begin
            if (Q == 1'b1) NS = S25;
            if (D == 1'b1) NS = S15;
            if (N == 1'b1) NS = S10;
        end
        ...
        S25: begin
            if (Q == 1'b1) NS = S25;
            if (D == 1'b1) NS = S10;
            if (N == 1'b1) NS = S5;
        end
        default:
            NS = S0;
    endcase
end

assign dispense = (CS == S25);
endmodule
```

Mealy vs Moore

	Mealy	Moore
# of states		
Output depend on		
Next state depend on		
Is output synchronous?		
Output latency		



RISC-V



- Reduced Instruction Set Computer

- We will use it to make our Final Project

- It is developed at Berkeley! (so is RISC 1,2,3,4)

- Pronounced as risk-five // nothing really risky here...

Terminologies

- **ISA** - instruction set architecture – (informal definition) an abstract model that specifies what a CPU could do.
- **ALU** - arithmetic logic unit
- **Immediate** - a constant (e.g. **ADDI** **rd** **rs1** **imm** means “**add** the value of **imm** with **rs1** and **store it in register rd**”)
- **PC** - program counter //not personal computer
- **Byte** - an 8 bit value
- **Word** - 32 (64, 16) bit value // depending on the ISA
- **Half-word** - 16 bit value

RISC-V ISA

31	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]				rs1		funct3		rd		opcode		I-type
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]								rd		opcode		U-type
imm[20 10:1 11]]				imm[19:12]				rd		opcode		J-type

- Load/store architecture: **operate on registers**, not directly on memory
- **Fixed length** (32bit) instructions, and Locations of register fields (in these 32 bits) **common** among instructions

RV32I is enough to run any C program

- Register loading, arithmetic, logic, memory load/store, branches, jumps
- Additional pseudo-instructions
- Reserved opcodes for extensions
- An incredibly useful doc for RISC V reference (models, syntax...):

<https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf>



Instruction Basics

- Arithmetic (R-, I-)
 - ALU modes
 - Different versions of same instruction:
 - slt, sltu, slti
 - srl, sra, slli
 - Load/store
 - Load: I-type, Store: S-type
 - Byte-addressing, little endian
 - Load/store granularity:
 - sw, sh, sb
 - lw, lh, lhu, lb, lbu
- * Sign extension

31	25	24	20	19	15	14	12	11	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]				rs1		funct3		rd		opcode		I-type
imm[11:5]		rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]		rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]								rd		opcode		U-type
imm[20 10:1 11]]				imm[19:12]				rd		opcode		J-type

R-type assembly:
`<inst> rd, rs1, rs2`

I-type assembly:
`<inst> rd, rs1, imm`
`<inst> rd, imm(rs1)`

S-type assembly:
`<inst> rd, imm(rs2)`

Instruction Basics - 2

Conditional Branches

- B-type instructions are formatted like S-type (think: what's different?)
- Branch comparison types - bltu, blt, beq

Jumps

- jal (J-type), jalr (I-type)
- 21b offset relative to PC vs 12b offset relative to an arbitrary address stored in rs1
- Both write PC+4 to rd unless rd == x0

Upper Immediate

- U-type used to get upper 20 bits of an immediate into rd
- auipc (add upper immediate to pc) also adds immediate to current PC

31	25 24	20 19	15 14	12 11	7 6	0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12 10:5]		rs2	rs1	funct3	imm[4:1 11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20 10:1 11]]		imm[19:12]			rd	opcode	J-type

B-type assembly:
 <inst> rs1, imm(rs2)

J-type assembly:
 jal rd, label(21bit offset
 relative to pc) //stores
 return address in a register

U-type assembly:
 <inst> rd, imm