EECS 151/251A
SP2022 Discussion 3

GSI: Yikuan Chen, Dima Nikiforov

Slides modified from Alisha Menon's, Zhenghan Lin's and Daniel Grubb's slides
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).

https://www.macworld.com/article/3275567/iphone-ipad/apple-7nm-a12-chip-iphone.html
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

\[ Q = 1 \] means inserted a Quarter.
Moore

\[ N = \sigma, D = 1, \delta = 1, R = 0 \]
Mealy

No S25 because dispense (output = 1) happens as soon as money is enough → then go back to S0
module vending_machine(
    input clk, rst,
    input Q, D, N, // Quarter(25¢), Dime(10¢), Nickle(1¢)
    output dispense
);

// 1. Define state registers

// 2. Define state names as local params (optional but recommended)

// 3. An always@(posedge clk) block to handle state assignment

// 4. An 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
);

    // enumerate all states
    reg [2:0] NS, CS; // next state, current state
    localparam S0 = 3’d0, S5 = 3’d1, S10 = 3’d2, S15 = 3’d3, S20 = 3’d4, S25 = 3’d5;

    // always @(posedge clk) begin
    if (rst) CS <= S0; // state always transitions!! (transitioning to itself is also a valid transition)
    else CS <= NS; // in each cycle, we assign current state to be next state
    end

    //}
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

assign dispense = (CS == S25);
endmodule
# Mealy vs Moore

<table>
<thead>
<tr>
<th></th>
<th>Mealy</th>
<th>Moore</th>
</tr>
</thead>
<tbody>
<tr>
<td># of states</td>
<td>fewer</td>
<td>more</td>
</tr>
<tr>
<td>Output depend on</td>
<td>input / state</td>
<td>state</td>
</tr>
<tr>
<td>Next state depend on</td>
<td>input / state</td>
<td>input / state</td>
</tr>
<tr>
<td>Is output synchronous?</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Output latency</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>
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**

- **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`, `srli`

- 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

---

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

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

**S-type assembly:**
```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

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