EECS 151/251A: Homework № 3

Due Friday, February 18th

Problem 1: FSM

You have been tasked with designing a custom hardware FSM for managing the state of an autonomous drone. The desired state transition diagram depicted below.

The system inputs are `armCmd`, `disarmCmd`, and `takeoffCmd`, which are commands provided by the autonomous controller, `taskComplete`, which signals when the current maneuver completes, and `sensorError`, which indicates that sensor fault has occurred and the FSM must override the controller’s commands and return the drone to a safe state.

The system outputs are `motor`, routing power to the drone’s motors for flight, and `led`, a status LED onboard for debugging.
Part a) Is this FSM a Mealy or Moore Machine?

**Solution:** This is a Moore machine.

Part b) Complete the Verilog module below to implement the specified FSM. In addition to the system inputs, the module also takes in a system clock and reset signal.

```verilog
`define DISARMED 3'd0
`define ARMED 3'd1
`define TAKEOFF 3'd2
`define NAVIGATE 3'd3
`define LAND 3'd4

module droneFSM(
    input reset, clk,
    // TODO
);
// Internal state variables
reg [2:0] state;
reg [2:0] nextState;
// Combinational assignments for output logic
assign motor = // TODO
assign led = // TODO
// Combinational block for next-state logic
always @(*) begin
    case (state)
        `DISARMED: // TODO
        `ARMED:   // TODO
        `TAKEOFF: // TODO
        `NAVIGATE: // TODO
        `LAND:    // TODO
            default: nextState = state;
        endcase
    end

    // Sequential block for state transitions
    always @(posedge clk) begin
        if (reset) begin
            // TODO
        end else begin
            // TODO
        end
    end
endmodule
```
Solution:

```
`define DISARMED 3'd0
`define ARMED 3'd1
`define TAKEOFF 3'd2
`define NAVIGATE 3'd3
`define LAND 3'd4

module droneFSM(
    input reset, clk,
    input armCmd, disarmCmd, takeoffCmd,
    output motor,
    output [1:0] led,
);
// Internal state variables
reg [2:0] state;
reg [2:0] nextState;

// Combinational assignments for output logic
assign motor = state != `DISARMED
assign led = state == `TAKEOFF ? 2'd1 : state == `NAVIGATE ? 2'd2 :
                    state == `LAND ? 2'd3 : 2'd0
// Combinational block for next-state logic
always @(*) begin
    case (state)
        `DISARMED:
            nextState = armCmd && !sensorError ? `ARMED : `DISARMED;
        `ARMED:
            nextState = takeoffCmd ? `TAKEOFF : `ARMED;
        `TAKEOFF:
            nextState = taskComplete ? `NAVIGATE : `TAKEOFF;
        `NAVIGATE:
            nextState = taskComplete || sensorError ?
                        `LAND : `NAVIGATE;
        `LAND:
            nextState = taskComplete ? `ARMED :
                (sensorError && taskComplete ? `DISARMED : `LAND);
        default: nextState = state;
    endcase
end

// Sequential block for state transitions
always @(posedge clk) begin
    if (reset) begin
        state <= `DISARMED;
    end else begin
        state <= nextState;
    end
end
endmodule
```
Part c) A system designer wants to immediately set the LED to 5 whenever a `sensorError` occurs, without waiting for a clock edge. Is this possible without changing if this is a Mealy/Moore Machine? If not, what type of FSM would the system become?

Solution: No, it is not possible to change this without changing the type of FSM. The FSM would become a Mealy Machine, as the output (`led`) would be combinationally tied to an input (`sensorError`).

Part d) Another system designer wants to add some safety features to the drone. If any `sensorError` occurs, even if it is de-asserted later, the drone must land and be unable to be armed until the system is reset.

The designer suggests adding `LAND_ERR` and `DISARMED_ERR` states to track if a sensor error occurs at any point. Draw an updated FSM that implements these changes.

Solution:

```
<table>
<thead>
<tr>
<th>Inputs:</th>
<th>Outputs:</th>
<th>Key:</th>
</tr>
</thead>
<tbody>
<tr>
<td>armCmd</td>
<td>motor</td>
<td>STATE</td>
</tr>
<tr>
<td>disarmCmd</td>
<td>led</td>
<td>motor/led</td>
</tr>
<tr>
<td>takeoffCmd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>taskComplete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sensorError</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>STATE</th>
<th>motor/led</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAVIGATE 1/2</td>
<td>takeoffCmd</td>
</tr>
<tr>
<td></td>
<td>sensorError</td>
</tr>
<tr>
<td></td>
<td>taskComplete</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>TAKEOFF 1/1</td>
<td>takeoffCmd</td>
</tr>
<tr>
<td></td>
<td>sensorError</td>
</tr>
<tr>
<td></td>
<td>taskComplete</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ARMED 1/0</td>
<td>disarmCmd</td>
</tr>
<tr>
<td></td>
<td>sensorError</td>
</tr>
<tr>
<td></td>
<td>taskComplete</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DISARMED 0/0</td>
<td>armCmd</td>
</tr>
<tr>
<td></td>
<td>sensorError</td>
</tr>
<tr>
<td></td>
<td>taskComplete</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LAND 1/3</td>
<td>reset</td>
</tr>
<tr>
<td></td>
<td>sensorError</td>
</tr>
<tr>
<td></td>
<td>taskComplete</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LAND_ERR 1/0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>DISARMED_ERR 1/0</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Problem 2: RISC-V Instructions

Consider the following potential new 32-bit RISC-V instructions. Consider whether or not they are feasible to implement, and if so, which of the 32-bit instruction formats could be used?

If one instruction is not feasible, is it possible to implement the instruction as a sequence of existing instructions? If so, list the sequence.

Note: We recommend referring to the RISC-V specification found here, as well as the RISC-V green card.

Part a) An integer power function, `pow rd, rs1, rs2`, defined as `rd = rs1 ** rs2`

Solution: Yes, this is feasible. This could be formatted as an R-Type instruction.

Part b) An integer square root, `isqrt rd, rs2`, defined as `rd = sqrt(rs1)`

Solution: Yes, this is feasible. This could be formatted as an R-Type or I-Type instruction. In this case either the immediate or the `rs2` fields could be ignored.

Part c) A three integer addition, `add3, rd, rs1, rs2, rs3`, defined as `rd = rs1 + rs2 + rs3`

Solution: No, this not feasible. There are three source register fields, which is not compatible with any 32 bit RISC-V instruction format. However, this could be done in two existing instructions:

```
add rd, rs1, rs2
add rd, rd, rs3
```

Part d) A three integer accumulating addition, `add3, rd, rs1, rs2`, defined as `rd = rd + rs1 + rs2`

Solution: Yes, this is feasible. All though there are three integers being added, this could still be encoded in an R-Type instruction as there are two source fields and one destination field.
Part e) Add a 20-bit immediate, \texttt{addi20, rd, imm20}, defined as \( rd = rd + \text{imm20} \)

Solution: Yes, this is feasible. The immediate could be encoded in either a U-Type or J-Type instruction, though a U-Type would be more reasonable due to the similarities to instructions like \texttt{lui}.

Part f) Branch if integers are within a threshold, \texttt{brth, rs1, rs2, rs3, imm}, defined as
\[
\text{pc} = \text{abs}(rs1 - rs2) < rs3 ? \text{pc} + \text{imm} : \text{pc} + 4
\]

Solution: No, this is not feasible. There are three distinct register addresses and one immediate, which does not fit into any RISC-V instruction format. Furthermore, this cannot be implemented using existing instructions without changing the program state, as the specified instruction has no destination register, but intermediate instructions would need to write to at least one register.
Problem 3: Hand Assembly

Manually construct the binary instruction for the following assembly instructions. Submit all of the following for each instruction:

- The 32-bit binary number for the instruction
- The core instruction format it belongs to
- Delineate the 32 bits into the subfields of the instruction format and label each field with the opcode/registers/immediate/offset etc. specified by the instruction.

Note: we highly encourage you to do this by hand from the ISA spec, but it is possible to assemble them using RISC-V GCC or venus.

Part a) sub x4, x2, x1

**Solution:** Binary encoding: 0x40110233

Instruction format: R-Type

<table>
<thead>
<tr>
<th>funct7 (31:25)</th>
<th>rs2 (24:20)</th>
<th>rs1 (19:15)</th>
<th>funct3 (14:12)</th>
<th>rd (11:7)</th>
<th>opcode (6:0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>1</td>
<td>2</td>
<td>0x0</td>
<td>4</td>
<td>0x33</td>
</tr>
</tbody>
</table>

Part b) xori x2, x3, 15

**Solution:** Binary encoding: 0x00f1c113

Instruction format: I-Type

<table>
<thead>
<tr>
<th>Imm[11:0] (31:20)</th>
<th>rs1 (19:15)</th>
<th>funct3 (14:12)</th>
<th>rd (11:7)</th>
<th>opcode (6:0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3</td>
<td>0x4</td>
<td>2</td>
<td>0x13</td>
</tr>
</tbody>
</table>

Part c) lb x3, 8(x5)

**Solution:** Binary encoding: 0x00828183

Instruction format: I-Type

<table>
<thead>
<tr>
<th>Imm[11:0] (31:20)</th>
<th>rs1 (19:15)</th>
<th>funct3 (14:12)</th>
<th>rd (11:7)</th>
<th>opcode (6:0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>5</td>
<td>0x0</td>
<td>3</td>
<td>0x03</td>
</tr>
</tbody>
</table>
Part d)  CORRECTION: jalr x10, x11, 2047

ORIGINAL: jalr x10, x11, 2048

Solution (Original): Invalid instruction.

Solution (Corrected): Binary encoding: 0x7FF58567

Instruction format: I-Type

<table>
<thead>
<tr>
<th>Imm[11:0] (31:20)</th>
<th>rs1 (19:15)</th>
<th>funct3 (14:12)</th>
<th>rd (11:7)</th>
<th>opcode (6:0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2047</td>
<td>11</td>
<td>0x0</td>
<td>10</td>
<td>0x67</td>
</tr>
</tbody>
</table>
Problem 4: Assembly Execution

Write down the values of the specified registers after the following programs have run. Show your work by annotating the what happens/changes after each instruction. Note that some instructions are pseudo-instructions, such as li for load immediate. Refer to Table 25.2 in the RISC-V spec for a list of pseudo-instructions and their base implementations.

Part a)

\[
\begin{align*}
\text{li } x0, & \quad 30 \\
\text{li } x1, & \quad 10 \\
\text{addi } x2, & \quad x0, \quad 20 \\
\text{sub } x2, & \quad x2, \quad x1 \\
\end{align*}
\]

\[
\begin{align*}
x0 &= \text{_______} \\
x1 &= \text{_______} \\
x2 &= \text{_______} \\
\end{align*}
\]

Solution:

\[
\begin{align*}
\text{li } x0, & \quad 30 \quad \quad /\!\!/ x0 = 0 \\
\text{li } x1, & \quad 10 \quad /\!\!/ x1 = 10 \\
\text{addi } x2, & \quad x0, \quad 20 \quad /\!\!/ x2 = 20 \\
\text{sub } x2, & \quad x2, \quad x1 \quad \quad /\!\!/ x2 = 10 \\
\end{align*}
\]

\[
\begin{align*}
x0 &= 0 \\
x1 &= 10 \\
x2 &= 10 \\
\end{align*}
\]
Part b)

li x1, 0xbeef
li x2, 0x64
sb x1, 0(x2)
sra x1, x1, x2
sb x1, 1(x2)
sb x1, 2(x2)
sra x1, x1, x2
sb x1, 3(x2)
lw x3, 0(x2)

x1 = ________  
x2 = ________

Solution:

li x1, 0xbeef  // x1 = 0xbeef
li x2, 0x64   // x2 = 0x64
sb x1, 0(x2)  // MEM[0x64] = 0xef
sra x1, x1, x2 // x1 = 0xbeee (shamt is truncated)
sb x1, 1(x2)  // MEM[0x65] = 0xee
sb x1, 2(x2)  // MEM[0x66] = 0xee
sra x1, x1, x2 // x1 = 0xbe
sb x1, 3(x2)  // MEM[0x67] = 0xbe
lw x3, 0(x2)  // x3 = 0xbeeeeeeef

x1 = 0xbeef
x2 = 0x64
x3 = 0xbeeeeeeef
Part c)

```
li x1, 1  \ // x1 = 1
slli x1, x1, 31 \ // x1 = 0x80000000
li x2 1 \ // x2 = 1
li x3 -1 \ // x3 = 0xffffffff
li x4 0 \ // x4 = 0
f1: sra x1 x1 x2 \ // x1 = 0xC0000000
    addi x4 x4 1 \ // x4 = 1
    blt x1 x3 f1 \ // branch until x1 = -1
```

\[
x1 = \_\_\_
\]
\[
x2 = \_\_\_
\]
\[
x3 = \_\_\_
\]
\[
x4 = \_\_\_
\]

Solution:

```
li x1, 1  \ // x1 = 1
slli x1, x1, 31 \ // x1 = 0x80000000
li x2 1 \ // x2 = 1
li x3 -1 \ // x3 = 0xffffffff
li x4 0 \ // x4 = 0
f1: sra x1 x1 x2 \ // x1 = 0xC0000000
    addi x4 x4 1 \ // x4 = 1
    blt x1 x3 f1 \ // branch until x1 = -1
```

\[
x1 = 0xffffffff
\]
\[
x2 = 1
\]
\[
x3 = 0xffffffff
\]
\[
x4 = 31
\]