EECS 151/251A: Homework № 3

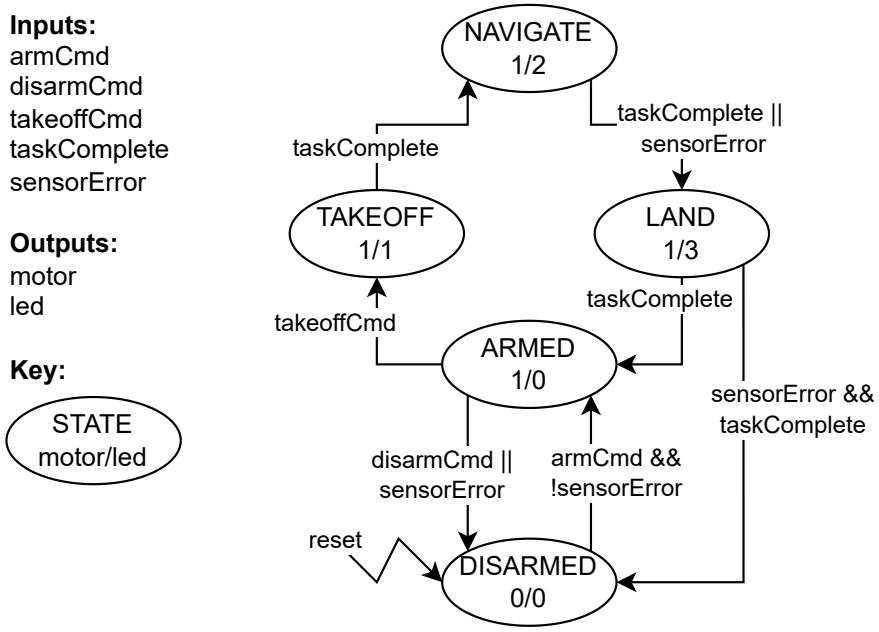
Due Friday, February 18th

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.

```
`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
```

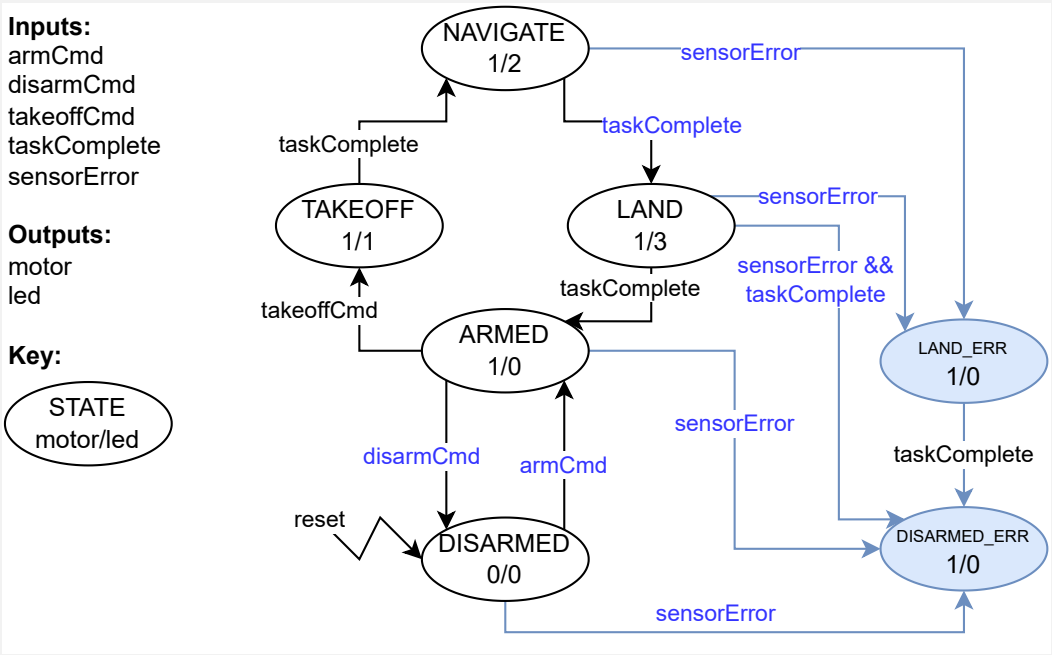
Part c) A system designer wants to immediately set the LED to 5 whenever a `sensorError` occurs, without waiting for a clock edge. Is this possible without changing if this is a Mealy/Moore Machine? If not, what type of FSM would the system become?

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:



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, `addi20, rd, imm20`, defined as `rd = rd + 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 `lui`.

part f) Branch if integers are within a threshold, `brth, rs1, rs2, rs3, imm`, defined as `pc = abs(rs1 - rs2) < rs3 ?  pc + imm :  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

funct7 (31:25)	rs2 (24:20)	rs1 (19:15)	funct3 (14:12)	rd (11:7)	opcode (6:0)
0x20	1	2	0x0	4	0x33

Part b) `xori x2, x3, 15`

Solution: Binary encoding: 0x00f1c113

Instruction format: I-Type

Imm[11:0] (31:20)	rs1 (19:15)	funct3 (14:12)	rd (11:7)	opcode (6:0)
15	3	0x4	2	0x13

Part c) `lb x3, 8(x5)`

Solution: Binary encoding: 0x00828183

Instruction format: I-Type

Imm[11:0] (31:20)	rs1 (19:15)	funct3 (14:12)	rd (11:7)	opcode (6:0)
8	5	0x0	3	0x03

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

Imm[11:0] (31:20)	rs1 (19:15)	funct3 (14:12)	rd (11:7)	opcode (6:0)
2047	11	0x0	10	0x67

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)

```
li x0, 30
li x1, 10
addi x2, x0, 20
sub x2, x2, x1
```

```
x0 = ________
x1 = ________
x2 = ________
```

Solution:

```
li x0, 30           // x0 = 0
li x1, 10           // x1 = 10
addi x2, x0, 20     // x2 = 20
sub x2, x2, x1      // x2 = 10
```

```
x0 = 0
x1 = 10
x2 = 10
```

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 = 0xbee (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 = 0xbeeeeeef
```

---

x1 = 0xbeef
x2 = 0x64
x3 = 0xbeeeeeef

Part c)

```
    li x1, 1
    slli x1, x1, 31
    li x2 1
    li x3 -1
    li x4 0
f1: sra x1 x1 x2
    addi x4 x4 1
    blt x1 x3 f1
```

```
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
                        // ...
                        // x1 = 0xfffc000
                        // ...
                        // x1 = 0xffffffff
    addi x4 x4 1        // x4 = 1
                        // ...
                        // x4 = 17
                        // ...
                        // x4 = 31
    blt x1 x3 f1        // branch until x1 = -1
```

```
x1 = 0xffffffff
x2 = 1
x3 = 0xffffffff
x4 = 31
```