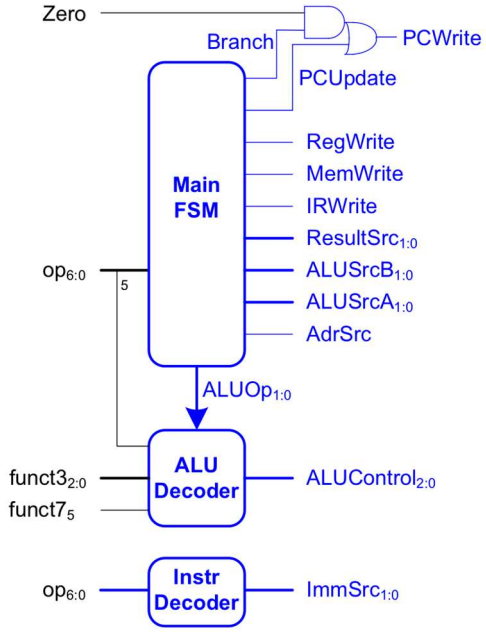
Multicycle Controller

Problem Statement

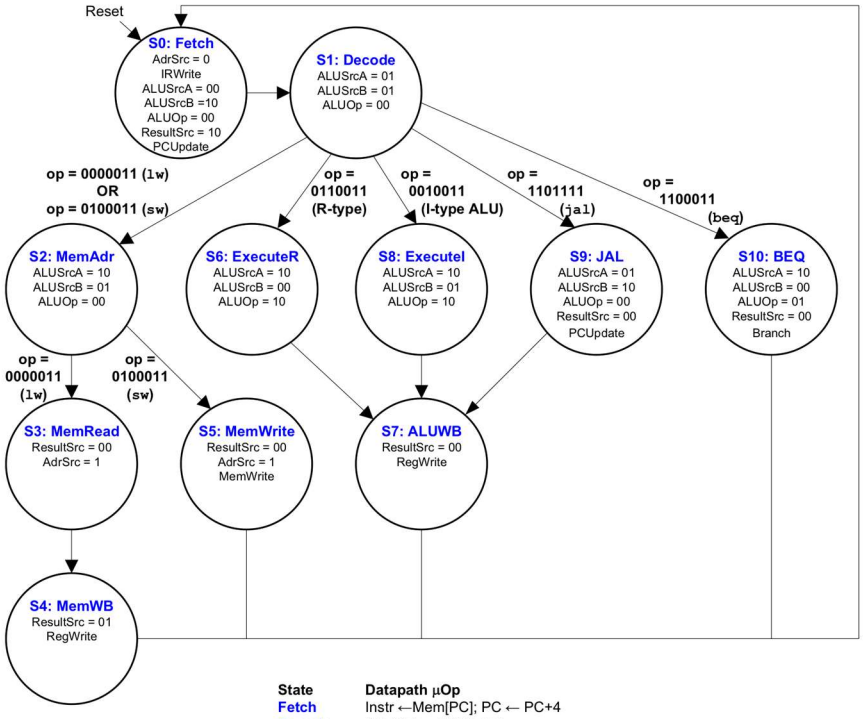
Design a **RISC-V multicycle controller** in SystemVerilog. At this point, the RISC-V processor will be a black box for you. It is described by some block diagrams, state transition diagrams, and truth tables. Your goal is to write clean SystemVerilog code that implements the specification and passes the testbench. If you continue to E85B, you will learn how the processor works and combine your controller with a data path to build a complete microprocessor.

Start with the controller.sv file below, which contains the top-level interface and the testbench. Write a **hierarchical description** of the controller, following the hierarchy shown in the top-level block diagram below. The system has **12 bits of inputs**: op[6:0], funct3[2:0], funct7[5], and Zero. It also implicitly receives a clock and reset for the Main FSM. It produces 16 bits of outputs.



The **Arithmetic/Logic Unit (ALU) Decoder**, is described by the following truth table. It has 7 bits of input and 3 bits of output. It looks at the intended ALU operation and some bits of the instruction (funct3[2:0], op[5], and funct7[5]) to determine what control signals to send to the ALU, but what they mean is not important to the implementation in this lab. You do not need to consider any input patterns beyond those on the table.

The Main FSM is a **Moore machine** described by the state transition diagram below. The FSM takes op[6:0] as an input. It produces a bunch of outputs. Outputs that are named but not given a value are implied to be asserted, i.e., 1. Any output not listed should be given a value of 0. The summary at the bottom explains how the FSM will relate to processing in the datapath, but is not important to implementing.



State	Datapath μOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PCTarget
MemAdr	ALUOut ← rs1 + imm
MemRead	Data ← Mem[ALUOut]
MemWB	rd ← Data
MemWrite	Mem[ALUOut] ← rd
ExecuteR	ALUOut ← rs1 op rs2
ExecuteI	ALUOut ← rs1 op imm
ALUWB	rd ← ALUOut
BEQ	ALUResult = rs1-rs2; if Zero, PC = ALUOut
JAL	PC = ALUOut; ALUOut = PC+4

ALUOp	funct3	{op_5, $funct7_5$}	ALUControl	Instruction
00	x	x	000 (add)	lw, sw
01	x	x	001 (subtract)	beq
10	000	00, 01, 10	000 (add)	add
	000	11	001 (subtract)	sub
	010	x	101 (set less than)	slt
	110	x	011 (or)	or
	111	x	010 (and)	and

The **Instruction Decoder** takes the op[6:0] and products ImmSrc[1:0] according to the following truth table. Again, you do not need to consider any input patterns beyond those on the table. Note that ImmSrc for R-type instructions is technically a **don't care**, but is listed as 00 to get consistent answers from the testbench.

Simulate your system with the testbench provided. Unless you are extraordinarily unfortunate, you will have made some mistakes in your SystemVerilog and the system will not pass the simulation on the first try. Systematically debug your system by identifying the first time an output is bad, developing an understanding of the expected behaviour so you can trace the bad output back in the **waveform viewer**. Once you have identified a block with good inputs and bad outputs, you have isolated the bug. Repeat until you have fixed all of your bugs. Don't just stare at the system hoping to find the error with your superior intellect, or make random changes in the hopes of getting lucky. The biggest learning objective of this lab is to become confident that you can **systematically trace** and **fix bugs** in a complex system.

It is important to get this lab working. It's the **capstone** of ENGR85A. If you take ENGR85B, you'll need Lab 5 as a piece of the full multicycle processor that you will build.