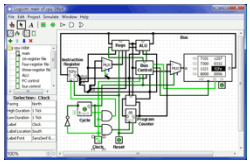
Lab 2: Processor Design in LogiSim

This lab will be done in the digital logic simulation program Logisim.

Download the software for your machine from moodle after satisfying the Java requirement. As mentioned in the website, "*Logisim is an educational tool for designing and simulating digital logic circuits. With its simple toolbar interface and simulation of circuits as you build them, it is simple enough to facilitate learning the most basic concepts related to logic circuits. With the capacity to build larger circuits from smaller subcircuits, and to draw bundles of wires with a single mouse drag, Logisim can be used (and is used) to design and simulate entire CPUs for educational purposes*."



As discussed in class, we will do this incrementally. We will first create an ALU, then a Register file and finally the single cycle Processor that we have discussed in class.

Part A: [ungraded] Familiarize yourself with Logisim

Complete all the steps listed in the Beginner's Tutorial. Implement a 4 bit Ripple Carry Adder. Simulate the design for all the 4 x 4 = 16 possible input conditions and submit the log output – show at least some cases where your outputs do not match the output as discussed in class. Specifically learn how to log outputs of the simulation as also the use of the Test Vector feature.

[The next parts will be expanded with more details during as we go along]

> You are only allowed to use Logisim's built-in components from the following libraries for all parts of this project:

Part B: Arithmetic Logic Unit (ALU) [7 points] Updated

This task is to create an ALU that supports all the operations needed by the instructions in the ISA. Please note that we treat overflow as RISC-V does with unsigned instructions, meaning that we ignore overflow.

You can use Logisim built-in components: Wires, Gates, Arithmetic, Memory

The ALU has 3 inputs.

Input Name	Bit Width	Description
A	32	Data to use for Input A in the ALU operation
B	32	Data to use for Input B in the ALU operation
ALUSel	4	Selects which operation the ALU should perform (see the list of operations with corresponding switch values below)

And one output

Output Name	Bit Width	Description
Result	32	Result of the ALU operation
OverFlow	1	Overflow die to ADD only
Zero	1	Zero if ALU operation results in 0

Implement the following functions:

ALUSel Value	Instruction
0	add: Result = A + B
1	sll: Result = A << B
2	slt: Result = (A < B (signed)) ? 1 : 0
3	Unused
4	xor: Result = A ^ B
5	srl: Result = (unsigned) A >> B
6	or: Result = A | B
7	and: Result = A & B
8	mul: Result = (signed) (A * B)[31:0]
9	mulh: Result = (signed) (A * B)[63:32]
10	Unused
11	mulhu: Result = (A * B)[63:32]
12	sub: Result = A - B
13	sra: Result = (signed) A >> B
14	Unused
15	bsel: Result = B

Use the names of your input and output of the adder unit exactly as specified in the Tables above viz A, B and ALUSel for inputs and Result for output. The name of file should be alu.circ.

Submission:

Please submit alu.circ. Submit a pdf document which has screen shots to show that the circuit is working.

Part C : Register File (RegFile) [6 points] [Updated]

Although the CPU that we discussed in class has 32 registers, you will only need implement 9 of them (specified below) to save you some repetitive work. This means your rs1, rs2, and rd signals will still be 5-bit, but we will only test you on the specified registers. Your RegFile should be able to write to or read from these registers specified in a given

RISC-V instruction without affecting any other registers. There is one notable exception: your RegFile should NOT write to x0, even if an instruction tries. Remember that the zero register should ALWAYS have the value 0x0. You should NOT gate the clock at any point in your RegFile: the clock signal should ALWAYS connect directly to the clock input of the registers without passing through ANY combinational logic.

The registers and their corresponding numbers are listed below.

Register #	Register Name
x0	zero
x1	ra
x2	sp
x5	t0
x6	t1
x7	t2
x8	s0
x9	s1
x10	a0

The register file circuit should have six inputs:

Input Name	Bit Width	Description
Clock	1	Input providing the clock. This signal can be sent into subcircuits or attached directly to the clock inputs of memory units in Logisim, but should not otherwise be gated (i.e., do not invert it, do not "and" it with anything, etc.).
RegWEn	1	Determines whether data is written to the register file on the next rising edge of the clock.
Read Register 1 (rs1)	5	Determines which register's value is sent to the Read Data 1 output, see below.
Read Register 2 (rs2)	5	Determines which register's value is sent to the Read Data 2 output, see below.
Write Register (rd)	5	Determines which register to set to the value of Write Data on the next rising edge of the clock, assuming that RegWEn is a 1.
Write Data	32	Determines what data to write to the register identified by the Write Register input on the next rising edge of the clock, assuming that RegWEn is 1.

The register file should also have the following outputs:

Output Name	Bit Width	Description
Read Data 1	32	Driven with the value of the register identified by the Read Register 1 input.
Read Data 2	32	Driven with the value of the register identified by the Read Register 2 input.

Please keep in mind registers have an “enable” input available, as well as a clock input.

Submission:

Please submit register.circ. Submit a pdf document which has screen shots to show that the circuit is working.

Part D: Single Cycle CPU [7 points]

In this part, you will be using logisim-evolution to implement a 32-bit single-cycle processor based on mini RISC-V ISA that we have defined.

Part E: Pipelined CPU [10 points]

In this you will be using logisim-evolution to implement a 32-bit multi [2?]-cycle processor based on our mini RISC-V ISA.