ECEN 240: Final Project Description

Introduction

The ECEN 240 final project is to design a 4-bit microprocessor and write a simple program that will run on the processor. A microprocessor is a Central Processing Unit (CPU) that is implemented on a single chip. In the old days, a CPU could fill an entire room! Today, almost all CPUs are implemented on a single chip, making the name "CPU" practically synonymous with "microprocessor."

Most of the design specifications for this microprocessor will be given in this document, but you must fill in some of the "blanks" and make it work. Please construct your own circuit. Do not submit the work of someone else!

Design Description

Modern microprocessors are very complex, but their general functionality may be better understood by implementing and programming a simple processor, first. For this project, you will build a 4-bit CPU from the building blocks we have studied throughout the semester. Also, as part of the project, you will write a simple program using machine code to illustrate how the CPU blocks work in concert with each other. The CPU blocks for this project are:

- A binary counter (program counter) to "point" to the current instruction in the program.
- A Read Only Memory (ROM) to contain the program.
- A second ROM to implement the logic necessary to decode the instructions and control the blocks of the CPU.
- A Multiplexer (MUX) to select the input path to the register file.
- A 3-port register file (1 input port and 2 output ports).
- An Arithmetic Logic Unit (ALU).
- An ASCII display.

The block diagram of this 4-bit microprocessor is shown in Figure 1. Though conceptually simple, this CPU architecture illustrates how the hardware of a microprocessor is controlled by a program to implement sequential instructions. Each instruction is "fetched" from memory and decoded into control signals which are then used to control the data paths, registers, ALU, and display.

Though you are not required to make improvements to the architecture, you will likely envision hardware improvements that could enhance the CPU's performance. For example, this CPU uses a 4-bit architecture rendering it capable of dealing with signed numbers in the range of -8 to +7. Increasing the number of bits to 32, increases the range of numbers in excess of +/- 2 billion. This change (though conceptually minor) enables significant computational advantages. Other enhancements might include an increase in the number of ALU functions, the addition of RAM, and the capability of "jumping" to specific addresses in program memory.

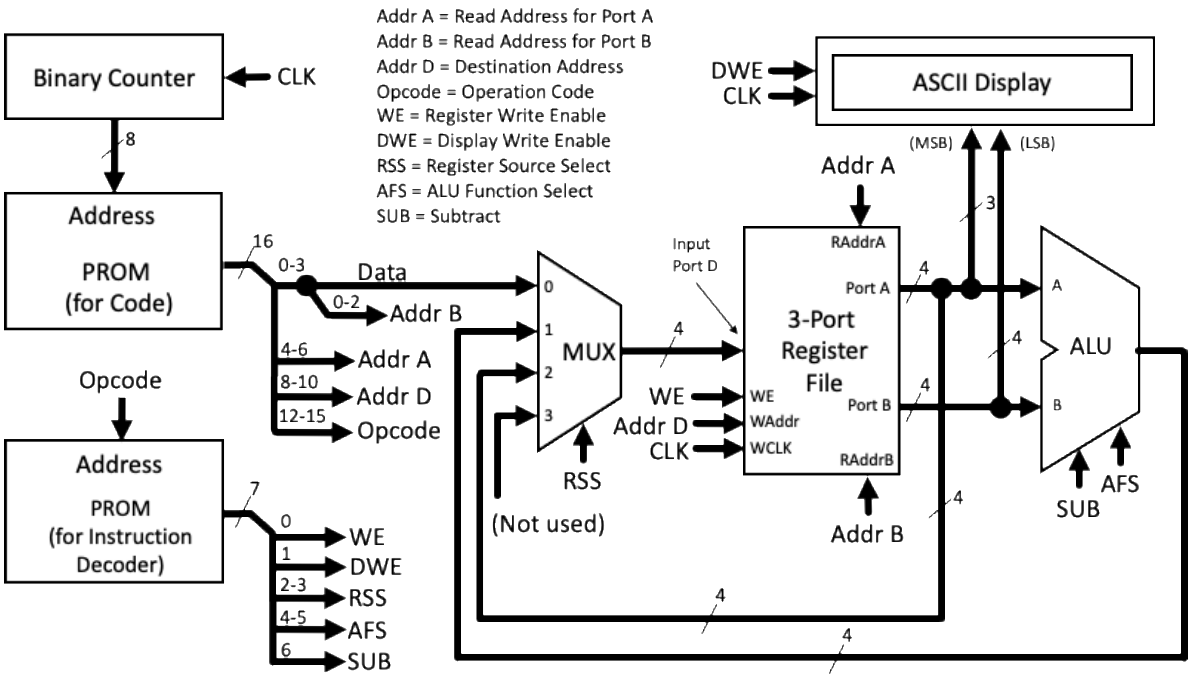

Figure 1 – A 4-bit CPU Block Diagram

Microprocessor Instruction Set

Large microprocessors have dozens (or even hundreds) of instructions (operations) in their instruction set. The number of bits required for each instruction will vary from processor to processor. This microprocessor architecture has nine instructions which may be represented in assembly code format or in 16-bit machine code format (see Table 1).

Opcode	Mnemonic	Instruction Operation	Assembly Code Format	Machine Code Format (16-bits)
0000	NOP	No Operation	NOP	0000 0000 0000 0000
0001	LD	Rd <= Data	LD Rd, Data	0001 0ddd 0000 DDDD
0010	MOV	Rd <= Ra	MOV Rd, Ra	0010 0ddd 0aaa 0000
0011	DISP	Display {Ra, Rb} on ASCII display	DISP Ra, Rb	0011 0000 0aaa 0bbb
0100	XOR	Rd <= Ra XOR Rb	XOR Rd, Ra, Rb	0100 0ddd 0aaa 0bbb
0101	AND	Rd <= Ra AND Rb	AND Rd, Ra, Rb	0101 0ddd 0aaa 0bbb
0110	OR	Rd <= Ra OR Rb	OR Rd, Ra, Rb	0110 0ddd 0aaa 0bbb
0111	ADD	Rd <= Ra + Rb	ADD Rd, Ra, Rb	0111 0ddd 0aaa 0bbb
1111	SUB	Rd <= Ra - Rb	SUB Rd, Ra, Rb	1111 0ddd 0aaa 0bbb
		Ra = Contents of register specified by Addr A Rb = Contents of register specified by Addr B Rd = Register specified by Addr D Data = Direct data from the program PROM		a = Port A Address bit b = Port B Address bit d = Port D Address bit D = data bit
When writing assembly code, Ra, Rb, and Rd are replaced by one of the actual register names, r0, r1, r2, … r7				

Table 1 – CPU Instruction Set Description

Low-level Computer Languages

The 16-bit machine instruction codes for this CPU are listed in the last column of Table 1. The first four bits of each instruction designate the operation to be performed (opcode). The remaining 12 bits are the register addresses and data words required by the instruction. The CPU instructions are entered into the program PROM in proper sequence to perform the required tasks, constituting a program.

Programs written in high-level languages (such as C++) are compiled and assembled into low-level machine code programs so they may be executed on a specific CPU. Machine code programs may be written without the use of a high-level language, but machine code is hard to read and understand. Assembly language is the next step up from machine code. It is more human friendly than the 1's and 0's of machine code. Each assembly language instruction usually translates to one machine code word. Below is an example of a simple program with both assembly code and machine code for our simple processor (refer to Table 1 and Figure 1 for a better understanding):

Assembly Code	Machine Code	Description of Operation
LD r2, 4	0001 0010 0000 0100_2 (1204_{16})	-Load a 4 into register 2 of the reg file.
LD r3, 2	0001 0011 0000 0010_2 (1302_{16})	-Load a 2 into register 3 of the reg file.
LD r4, 1	0001 0100 0000 0001_2 (1401_{16})	-Load a 1 into register 4 of the reg file.
ADD r5, r2, r3	0111 0101 0010 0011_2 (7523_{16})	-Add the contents of registers 2 and 3 and store the result in register 5. At the end of the operation, register 5 will contain a 6.
ADD r6, r5, r4	0111 0110 0101 0100_2 (7654_{16})	-Add the contents of registers 5 and 4 and store the result in register 6. At the end of the operation, register 6 will contain a 7.
LD r1, 3	0001 0001 0000 0011_2 (1103_{16})	-Load a 3 into register 1 of the reg file (this will be used to convert the math results to ASCII so that it may be displayed).
DISP r1, r6	0011 0000 0001 0110_2 (3016_{16})	-Display the contents of registers 1 and 6. At the end of the operation, a 37_{16} will be sent to the display, and a "7" will be displayed (refer to an on-line ASCII table for doing display writes).

The above code adds 4+2+1 and displays the result. There is more than one way to write the code to accomplish this task, just as there is more than one way to write a program in a high-level language.

To verify functionality of this CPU project, you are asked to write a "program" that accomplishes the tasks described in a later section of this document. In your Logisim CPU, the machine code program will be stored in the program PROM and will be manipulated and viewed in hexadecimal format. Though not required in this project, you might ask yourself, "what instructions would I find useful to add to this CPU to make it more capable?"

CPU Functional Blocks

This CPU requires only seven functional blocks. Below is a description of each block and the corresponding design requirements:

Program Counter

The program counter will be implemented with an 8-bit binary counter from the "Memory" menu of Logisim. The input to the counter is the system clock. The counter output connects to the address lines of the program PROM. The counter will count: 0000 0000 -> 0000 0001 -> ... -> 1111 1111 -> 0000 0000 -> ... As the counter counts, the address to the program PROM will be incremented, and a new machine code instruction will be executed. In class, you have learned how to design your own counters. Though not required in this project, you might ask yourself the question, "How would I design an 8-bit counter?" K-maps are not practical above 6-bits.

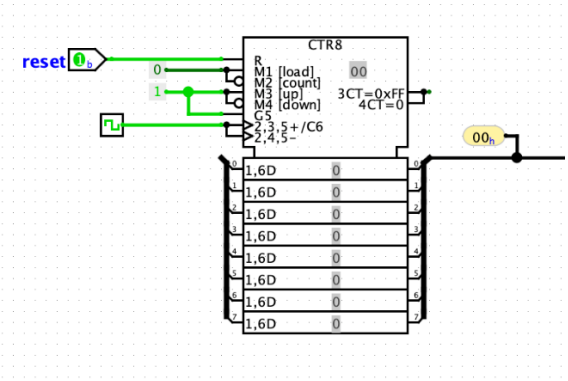

Figure 2 - Example Counter Placement and Configuration

Program PROM

The program PROM consists of a Logisim ROM from the "Memory" menu with 8 address bits and 16 data bits. It holds the program in the form of machine code instructions. The input address to the program ROM is the 8-bit binary count from the binary counter. Each word or location in the ROM holds one 16-bit instruction, constituting one line of machine code. Each line of machine code is segmented or "mapped" into parts. It contains an operation to perform, and may contain addresses for register access or data to be transfered to a register (see in Table 2).

Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0
Instruction (opcode)	Register Write Address (Bit 11 not used)	Register Read Address A (Bit 7 not used)	Register Read Address B (bit 3 not used), or Data

Table 2 – Program PROM Bit Map

Each time the program counter increments to the next count, the PROM outputs the new contents of the 16-bit word, and is processed by the CPU circuitry. This basic processor only allows sequential execution of a machine code program. More complex processors would allow the program counter to jump to a line of code outside of the normal counting sequence. Though not required in this project, you might ask yourself, "what would it take to allow jumps to various lines in the program PROM?"

Instruction Decoder PROM

The instruction decoder will receive a 4-bit instruction (opcode) from the program PROM (the 4 most significant bits), and configure the control signals of the functional blocks so that the instruction may be executed. The control signals consist of the enable signals and select lines listed in Table 3. To design the Instruction decoder, use Figure 1 to help you identify how each of the 7 functional blocks will be configured in order to execute each instruction. Enter these required signal states into the truth table.

This decoder could potentially be implemented by solving K-maps of the outputs of Table 3. Instead, you are asked to implement this logic with a Logisim ROM from the "Memory" menu. The ROM will have 4 address bits, and 7 data bits.
The required control bits are shown as outputs from the Instruction Decoder Prom in Figure 1. They are:

- Bit 0 = Register file write enable (WE).
- Bit 1 = The ASCII display write enable (DWE).
- Bits 3, 2 = The Register file Source Selector bits (RSS).
- Bits 4, 5 = ALU Function Selector bits (AFS).
- Bit 6 = Alters the addition function to subtraction (SUB).

Instruction Operation Code (opcode)		Bit 6 ALU SUB	Bits 5, 4 ALU Function	Bits 3, 2 Register Source MUX	Bit 1 Display WE	Bit 0 Register WE	Combined Hex Equivalent
0000 [NOP]		0	00	00	0	0	
0001 [LD]							
0010 [MOV]							
0011 [DISP]							
0100 [XOR]							
0101 [AND]							
0110 [OR]							
0111 [ADD]							
1111 [SUB]							

Table 3 – Instruction Decoder Truth Table

The left hand side of the truth table (the opcode) will be the address of each memory element of the instruction decoder PROM. The contents of the right hand side of the truth table will be enterred as data into the instruction decoder using hexadecimal numbers. Example: Address "0_{16}" of the PROM will contain "00_{16}" corresponding to the "No Operation" instruction.

Register Input Data MUX

This data MUX will select the input to the register file. It will have 4 inputs with 4 data bits per input. The 2 select lines of this MUX originate from the instruction decoder (RSS in Figure 1). The output consists of 4 bits that serve as inputs to the register file.

You may use a single multiplexer from the "Plexers" menu of Logisim to implement this MUX (instead of building your own from four 1-bit multiplexers).

The "Register Input Data MUX" selects one of the following inputs:

Input 0 - Four bits from the program PROM
Input 1 – Four bits from the ALU
Input 2 – Four bits from the output of the register file.
Input 3 – Not used

3-port Register File

The 3-port register file of this CPU is based on the 2-port design of a previous lab. The 2-port version had one input port and one output port, and an address shared by both the input and the output ports. This 3-port version will have one input port and two output ports. Each of these ports will have its own address. The register file will consist of eight, 4-bit registers, with each 4-bit register built from a sub-circuit of the 1-bit loadable register discussed in class (the MUX/Flip Flop combination).

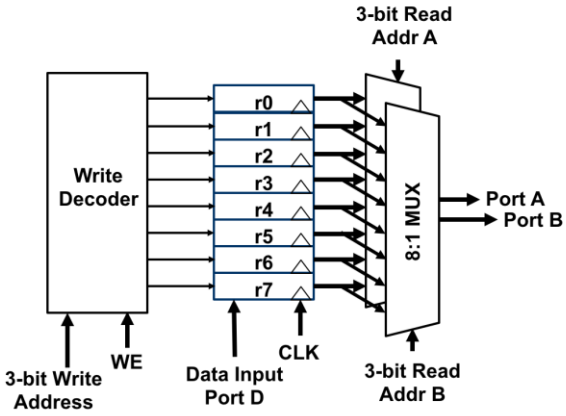

Figure 3 – Triple-ported Register File

To implement the 3-port version, there will need to be two multiplexers at the output instead of one (see Figure 3). The two output ports are called "Port A" and "Port B". Each output port is 4-bits wide. The data input port will receive 4-bit data from the "Register Input Data MUX." The other inputs are the clock (CLK), a 1-bit write enable (WE), a 3-bit write address, and two 3-bit read addresses. The output ports connect to the ALU inputs and to the ASCII display input.

ASCII Display

This project will have an ASCII display. Use the "TTY" component from the Input/Output menu of Logisim. The data input to the ASCII display is 7-bits wide, and will be driven by the combined 8 data bits from the register file (the MSB is not needed). The other inputs to the ASCII display will be a control bit from the instruction decoder (the write enable) and the clock. The output of the display will be a visual representation of the ASCII symbols stored in the selected registers of the register file (refer to an ASCII table).

ALU

The ALU will function like the ALU designed in the earlier lab except you will add the subtraction function. The data inputs are the 8 data bits from ports A and B of the register file (port A connects to the A input, port B connects to the B input). The control inputs to the ALU will consist of the 2 select bits required to select one of the four original ALU functions and a "Sub" signal to convert the adder into a subtracter (refer to Figure 9.4 of the textbook). The 4-bit output of the ALU will connect to one input of the "Register Input Data MUX" described earlier and shown in Figure 1. The ALU function selector bits should produce the following behavior:

Operation		Sub	S1	S0
A ^ B (XOR)		0	0	0
A & B (AND)		0	0	1
A | B (OR)		0	1	0
A + B (ADD)		0	1	1
A – B (Subtract)		1	1	1

Table 4 – ALU Operation Codes

You may use the MUX you designed originally, or you may simply use a 4-bit, 4:1 MUX from the "Plexer" menu of Logisim.

The adder of the ALU must be constructed from a subcircuit of the 1-bit full adder discussed in class. The 1-bit full adder sub-circuit will then used to build the 4-bit adder subcircuit.

Note: Even though the original 4-function ALU did not have a built in subtraction operation it is still possible to write a program that will make use of the original ALU functions to perform the "A – B" operation! The process of converting "B" to its negative form requires several passes through the ALU. One pass through the ALU will invert all of the bits of "B" by XORing it will "1111". A second pass is required to add 1, and a third pass to add "A" and "-B". We will refer to this multi-pass solultion as "software subtraction". Your test program will perform subtraction using this technique before using you newly designed hardware subtraction circuit.

Software subtraction is not very efficient. It requires many passes through the ALU. This illustrates how CPUs with reduced instruction sets are still able to perform complex operations. The hardware subtraction operation can take place with only one pass through the ALU.

Design Procedure

The following is a recommended procedure for designing your Logisim microprocessor:

1. Place the program counter from the “Memory” menu into your main circuit. Set the size of the counter to 8-bits.
2. Add a clock port from the “Wiring” menu (label it “CLK”), and add a regular port to clear the counter (label it CLR). Connect these two ports to the proper pins of the program counter, and try it out. Note: The “Clock” port from the wiring menu has special characteristics. It will automatically toggle if you select “Ticks Enabled” from the “Simulate” tab. You can adjust the frequency to the right speed for viewing with the “Tick Frequency” option under the same “Simulate” tab.
3. Add the program PROM (the ROM component from the “Memory” folder), and connect its address input to the output of the counter. The ROM should be set to 8 address bits and 16 data bits.
4. Use “splitters” from the “Wiring” folder to create the signals, Data, Addr B, Addr A, Addr D, and Opcode, as shown in the block diagram in Figure 1. You will need to become familiar with the Logisim splitter settings (lower left window) in order to do this correctly. When done, it should look something like this:

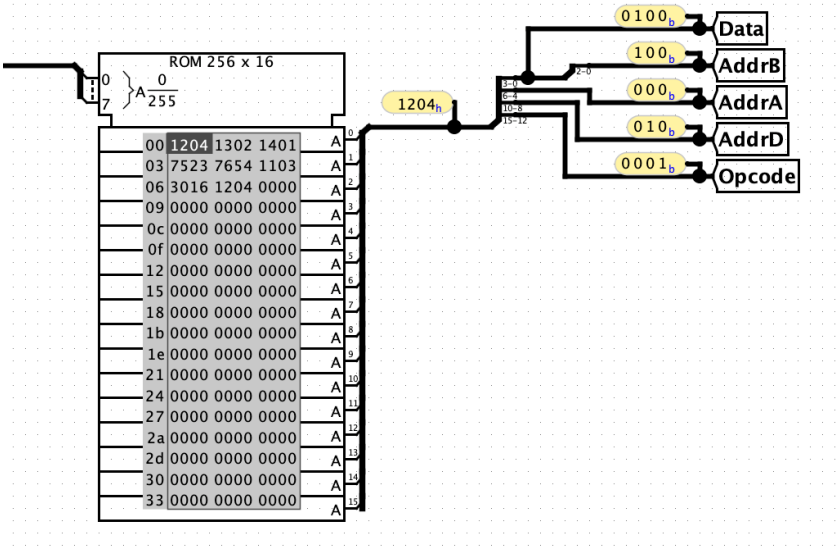

Figure 4 – Program PROM and Splitter Connections

For de-bug purposes, you can place the “Probe” component from the “Wiring” menu on each signal bus to see their contents. The probe tool can be set to display using the most appropriate number system (hexadecimal, binary, etc).

5. Test your combined counter/program PROM by filling the prom with a simple test program. The program shown in the above ROM diagram is the simple program of the example shown earlier in this document (4 + 2 + 1). Once enterred into the PROM, you can save this program (PROM contents) to a text file for later use. To do this, right-click on the ROM and select “Save Image”. It is recommended that you save the text to a file with the “.txt” extension. Once the image is saved to a text file, you may open it and edit it with a text editor. You can load it into the ROM with right-click “Load Image”. From this step forward, you can test each block with a simple test program.

6. Build the instruction decoder using a 16x8 ROM (16 locations that each hold 8 bits). Connect the address input to the appropriate signal from the program PROM. Split the 8 outputs into the required control signals with splitters (similar to the method used for the program PROM outputs in Figure 4).
7. Add the register input data MUX and connect input 0 to the "Data" output of the program PROM. This mux is a 4-input (2 select line) 4-bit multiplexer from the "Plexers" folder.
8. Construct the 3-port 8 word x 4-bit register file subcircuit:
    a. Look at (or copy) the register lab done previously
    b. Add the second multiplexer for an additional read port
    c. Make sure the addresses are separate so there are three, 3-bit addresses (an write address and 2 read addresses).
9. Place the register file subcircuit into the main CPU and connect the data input to the MUX output.
    a. Place a "Probe" from "Wiring" folder on each output port for testing.
    b. Load a simple test program into the ROM that verifies that the register file behaves as expected under program control.
10. Modify the ALU block subcircuit by adding a "SUB" signal.
    a. Look at (or copy) the previous ALU lab work.
    b. Add the "SUB" pin and the necessary hardware to turn the adder into a subtractor when the "SUB" pin is asserted.
    c. Make sure the circuitry for the two selector inputs S1 and S0, correspond with:
        i. XOR = 00
        ii. AND = 01
        iii. OR = 10
        iv. ADD = 11
11. Place the ALU subcircuit into the main CPU circuit and
    a. Connect the A and B inputs to the corresponding register file outputs.
    b. Connect the operation selection pins to the proper instruction decoder output.
    c. Place a "Probe" from the "Wiring" menu on the output of the ALU. Connect this ALU output signal to input "1" of the data selection MUX.
    d. Try to envision a way the ALU might implement either a software subtraction or a hardware subtraction (as described in the ALU section of this document).
12. Add the ASCII display (Logisim "TTY" component) to your main CPU.
    a. Connect the data input to the outputs of the register file. You will need to use splitters to create 7 bits from the two output ports of the register file.
    b. Connect the same clock used by the program counter and register file to the display.
    c. Connect the display write enable signal from the instruction decoder to the enable input of the display.
    d. Test with ASCII codes loaded into the register file with the instrution PROM.
13. Connect any remaing signals using the block diagram and the description to guide you.
12. Enter a simple test program into the program PROM and verify basic functionality.
13. Write a program that implements all of the tests listed in the "Writing Your Test Program" section below. You will want to add one test at a time, but all of the tests should be part of a single program. Remember,

you can actually edit the program with a text editer, and load the program into the program the program PROM. You may wish to use the programming template at the end of this document.

Writing Your Test Program

To test and pass off your microprocessor, you are asked to write a program in Machine code. This machine code will have specific requirements which are designed to exercise various components of the microprocessor. Program the PROM with code to do the following tasks:

1. Print your first and last name to the ASCII display followed by the "#" sign and a favorite 2-digit number (your birthday, lucky number, etc.). For example: "John Smith #21".
2. Print the equation "4+2=", and then print the result of using your ALU to do an actual 4+2.
3. Print the equation "4*2=", and then print the result of calculations done, using the ALU, to calculate 4*2. This is implementing a multiplication in software. For example: software would create a loop which adds "4" to itself as many times as the multiplicand indicates (2 in this case).
4. Print the equation "4-2=", and then print the result of doing the 4-2 subtract operation using just the instructions with opcodes 0000_2 through 0111_2 as defined in Table 1 (don't use the subtraction command). This "software subtraction" is to be done by placing the value of +2 in a register and then using the ALU functions XOR and ADD to convert the +2 to -2 (using 2's complement), which may then be added to the 4 to obtain the result of the subtraction, 4-2.
5. A second time, print the equation "4-2=" and use hardware subtraction command to calculate the value of 4-2. Print the result of the calculation.

Key Requirements Summary

- A Logisim ROM must be used for the instruction decoder.
- The only parts that may be used out of the "Memory" folder of Logisim for the final project are the D flip-flop, counter and ROM.
- None of the parts in the Logisim "Arithmetic" folder are to be used.
- The adder portion of the ALU must use the one bit full adder discussed in class and this must be in its own sub-circuit. The one bit full adder sub-circuit is then used to build the four bit adder.
- It is required that you use four, 1-bit loadable registers (as given in class), to build the 4-bit registers in the project. Consider putting this 2-bit loadable register in its own subcircuit for readability and neatness.
- It is not required that you use four 4:1 MUXes to design the ALU MUX.
- The software subtract operation is to put a +2 in a register and use ALU functions to create the -2 that is to be added to the value 4 to get the result of 4-2.
- The hardware subtract instruction is to use an opcode of 1111_2.

Hints and Helps

Don't wait to start this project! You will be inundated with assignments from all your classes and time will be

scarce, especially with everything else as “life happens.”

Making the parts that you construct into sub-circuits, makes the final project look cleaner and easier to debug.

Use the “top-down design, bottoms-up implementation” methodology to build and test your 4-bit processor. In other words, test each section separately and then integrate them together in Logisim. A few hardware notes:

- The part to use for the counter is just called “Counter.” It is found in the Memory folder. It should use 8 data bits, trigger on the rising edge and wrap around on overflow. You will probably need to play around with this part some to understand how to use it. Remember that holding the mouse over an input or output pin will give a “tool tip.”
- You may use a regular input pin to drive the clock on the counter, or you may use a “Clock” that is found in the Wiring folder. You control the “Clock” by using “Tick” related items under the “Simulate” menu.
- A ROM, under the Memory folder, with an address bus of 8 bits and a data bit width of 16 bits is used for the program PROM.
- The address bit width is 4 bits and the data bit width is 7 bits (6 bits without the hardware subtract instruction, but just use 7 all the time) for the ROM that is to be used for the instruction decoder.
- Do not use the preset, enable or clear inputs on the flip flop that is part of the one bit loadable register.
- There are several ways to edit Logisim ROMs. One way is to use the poke tool (the hand with the index finger extended). Select the poke tool, select the ROM, then select the cell you want to edit and type a hexadecimal value. The other way to edit the contents of the ROM is to select the ROM with the edit tool and then in the “Selection: ROM” window, click the “(click to edit)” item in the “Contents” row of the window. This pops up a “Logisim: Hex Editor” window. Select the cell you want to edit in the hex editor and enter a hexadecimal value. You can also right-click on the ROM and select “Save Image” to save a text file. You may then edit the saved file with a text editor and load your edited progam back into the ROM.
- The ASCII display in Logisim is called “TTY.” It is found in the Input/Output folder. You might set it to 8 rows and 32 columns. It only uses 7 bit ASCII. Use the lower 7 bits of the 8 bits coming from the registers as input to the TTY. You may want to hook this up in a small test circuit to understand how to operate it.

Programming Template

Assembly Code	Machine Code (Hex)	Description of Operation	Register Contents After Instruction Execution							
			r7	r6	r5	r4	r3	r2	r1	r0