CSCI 2461 Project 3 Report

Danish Imtiaz, Seamus Malley, Tim Traversy

Concept

Starting Off

We began this project by studying the microarchitecture of the original LC-3, to completely understand the nature of each subset of the machine and begin thinking about what changes we would need to make to create our own computer. We found most of this information in Appendix C of the textbook.

Operations

Next we designed our opcodes and instructions. To choose between four registers we would need **2 bits** for each register. To allow for all three registers to be included in an 8 bit instruction, the opcode would have to be **2 bits** long as well. That is only enough for 4 instructions, so we used the same opcode for LD and ST and used the third bit to pick between the two. With this in mind, we created the following instructions.

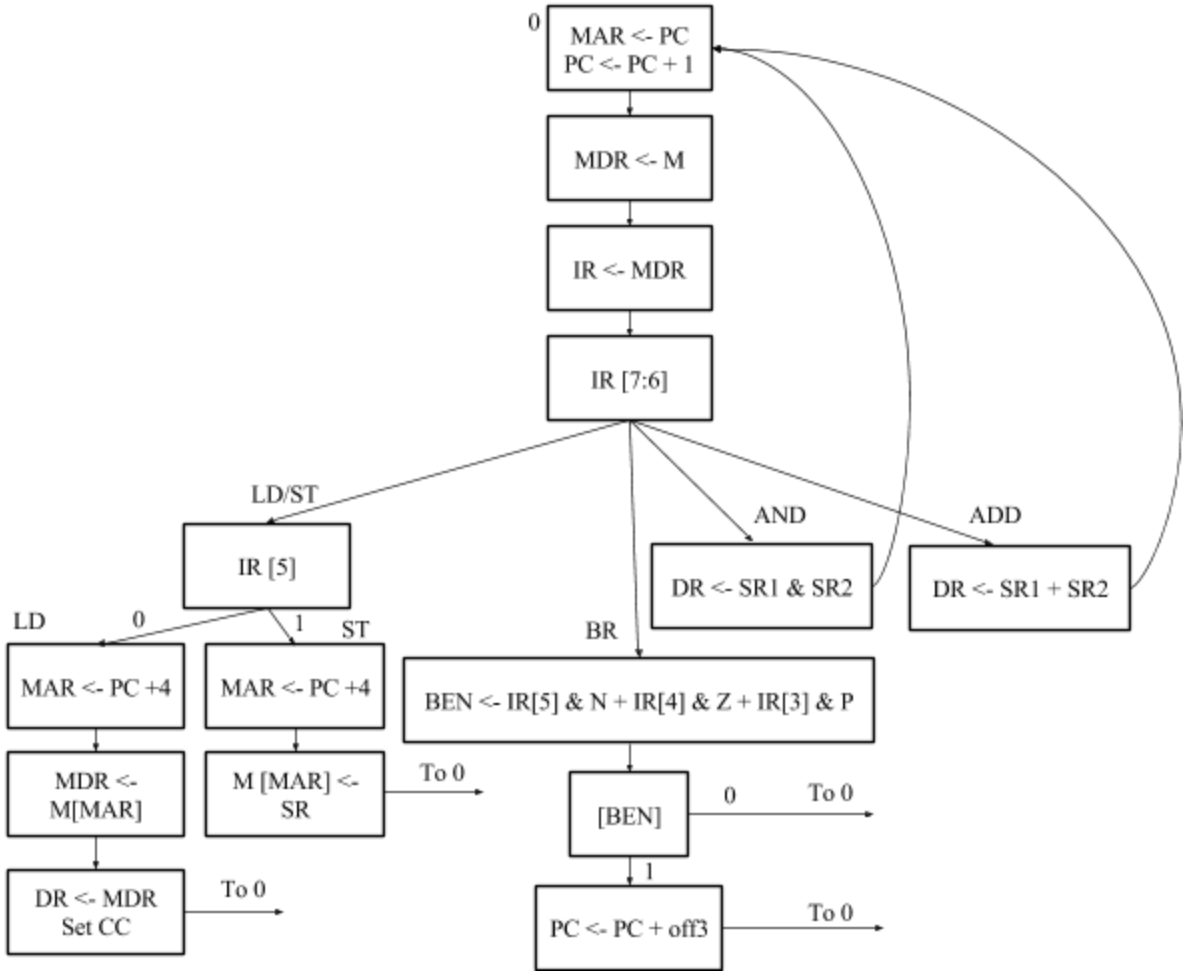
Operation	OP 1	OP 0	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
BR	0	0	n	z	p	offset3	offset3	offset3
LD	0	1	0	R0/R1	offset4	offset4	offset4	offset4
ST	0	1	1	R0/R1	offset4	offset4	offset4	offset4
ADD	1	0	DR [1]	DR [0]	SR1 [1]	SR1 [0]	SR2 [1]	SR2 [0]
AND	1	1	DR [1]	DR [0]	SR1 [1]	SR1 [0]	SR2 [1]	SR2 [0]

State Diagram

To step through our program we knew that a state diagram would have to be implemented. We used the one given in the textbook as a template, our own flowchart had a few differences:

1. All unneeded operations were deleted
2. The state where the source register is placed in the MDR was deleted. Instead we chose to take the bits directly off the global bus. *Note:* In our final microinstruction set this state is still present, because we decided to remove it late in the process, and couldn't rewrite all our instructions around it. Future references to the state diagram will reference 16 states, although we skip this one in the actual circuit.
3. We moved the branch check to a new state, because we only wanted to check for it when the Branch instruction was called.

This yielded the following diagram:



Microinstructions

The next step was the creation of our control unit, the most complex part of our program. Using our state diagram, we determined we would need 16 states. In the LC3, each microinstruction holds microsequencer control (MC) signals and data path control (MDC) signals. We needed seven MC signals:

1-4. J bits: Select which state to go to next
2. Branch: check to see if branch is enabled
3. LD/ST selector: select which state to go to from the LD/ST state
4. IRD: overwrite the next state with the opcode from the instructions

The amount of MDC signals we would need is unknown, so we filled them out after we designed all our gates and components. Here is a complete list of our microinstructions. *Note*: Because of CedarLogic limitations, we found the most efficient storage was 6 4x4 ROMs. ROM 1 holds the J bits, ROM 2 holds the other MC signals, plus a MDC signal that we realized we needed late in the process.

Microinstructions

ROM 1 and 2

Address	Operation	J[0]	J[1]	J[2]	J[3]	ST/LD	STBit	BEN	IRD
0	MAR<-PC	1	0	1	0	0	0	0	0
1	LD	0	1	0	1	0	0	0	0
2	BR2	0	0	0	0	0	0	0	0
3	BR	1	0	0	0	0	0	0	0
4	BR1	0	0	0	0	0	0	1	0
5	LD1	1	0	0	0	0	0	0	0
6	ST2	0	0	0	0	0	0	0	0
7	LD/ST	0	0	0	1	**1**	0	0	0
8	LD2	0	0	0	0	0	0	0	0
9	ST	1	1	1	0	0	0	0	0

A	MDR<-M	1	1	0	0	0	0	0	0
B	ADD	0	0	0	0	0	0	0	0
C	IR<-MDR	1	1	0	1	0	0	0	0
D	IR[7:6]	0	0	0	0	0	0	0	**1**
E	ST1	0	1	1	0	0	**1**	0	0
F	AND	0	0	0	0	0	0	0	0

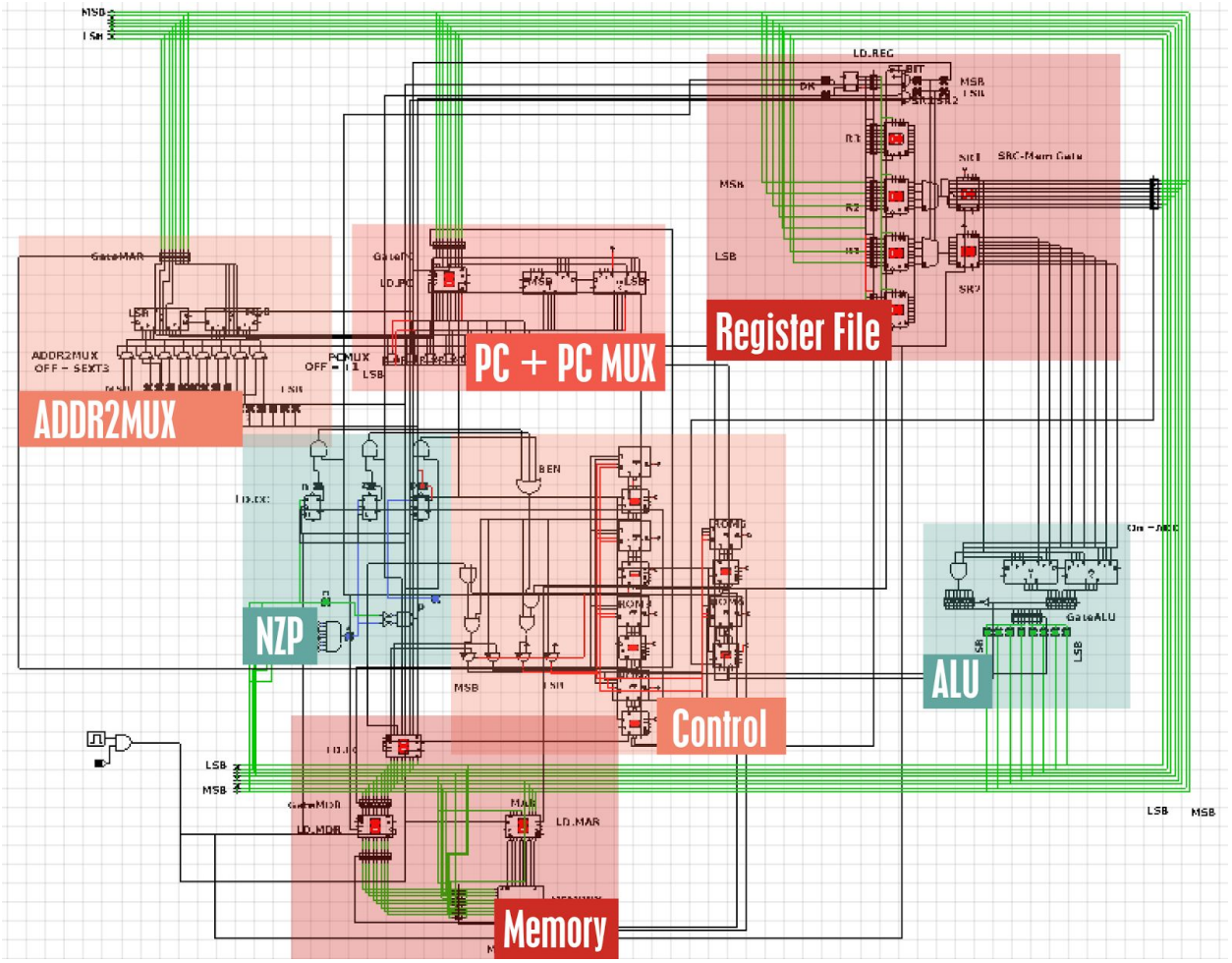
ROM 3 and 4

Address	Operation	Gate MAR	Gate PC	Gate ALU	GateMDR	LD. PC	LD. REG	LD. CC	LD. IR
0	MAR<-PC	0	**1**	0	0	**1**	0	0	0
1	LD	**1**	0	0	0	0	0	0	0
2	BR2	0	0	0	0	**1**	0	0	0
3	BR	0	0	0	0	0	0	0	0
4	BR1	0	0	0	0	0	0	0	0
5	LD1	0	0	0	0	0	0	0	0
6	ST2	0	0	0	0	0	0	0	0
7	LD/ST	0	0	0	0	0	0	0	0
8	LD2	0	0	0	**1**	0	**1**	**1**	0
9	ST	**1**	0	0	0	0	0	0	0
A	MDR<-M	0	0	0	0	0	0	0	0
B	ADD	0	0	**1**	0	0	**1**	**1**	0
C	IR<-MDR	0	0	0	**1**	0	0	0	**1**
D	IR[7:6]	0	0	0	0	0	0	0	0
E	ST1	0	0	0	0	0	0	0	0
F	AND	0	0	**1**	0	0	**1**	**1**	0

ROM 5 and 6

Address	Operation	LD. MAR	LD. MDR	MEM.EN /R	/W	ADDR2 MUX	ALUK	PCMUX	GateSR-Mem
0	MAR<-PC	**1**	0	0	0	0	0	0	0
1	LD	**1**	0	0	0	**1**	0	0	0
2	BR2	0	0	0	0	0	0	**1**	0
3	BR	0	0	0	0	0	0	0	0
4	BR1	0	0	0	0	0	0	0	0
5	LD1	0	**1**	**1**	0	0	0	0	0
6	ST2	0	0	0	**0**	0	0	0	0
7	LD/ST	0	0	0	0	0	0	0	0
8	LD2	0	0	0	0	0	0	0	0
9	ST	**1**	0	0	0	**1**	0	0	0
A	MDR<-M	0	**1**	**1**	0	0	0	0	0
B	ADD	0	0	0	0	0	**1**	0	0
C	IR<-MDR	0	0	0	0	0	0	0	0
D	IR[7:6]	0	0	0	0	0	0	0	0
E	ST1	0	**1**	0	1	0	0	0	**1**
F	AND	0	0	0	0	0	0	0	0

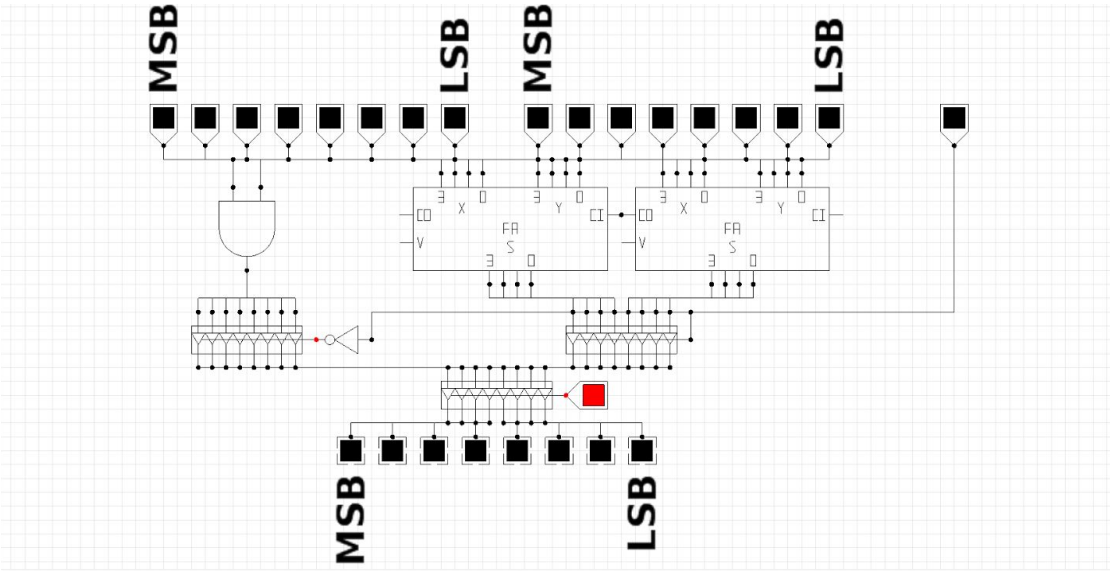
Building Circuit



In this section we will describe how we created each part of the machine. We first built the major components: Register File, ALU, Control Unit, Memory, NZP, PC and PCMUX, and ADDR2MUX. The specifics of each component is detailed in the following pages.

After we built these parts we worked as a group to determine how many control signals we would need and we built them into the ROM. We then connected each ROM output to the corresponding gate or selector.

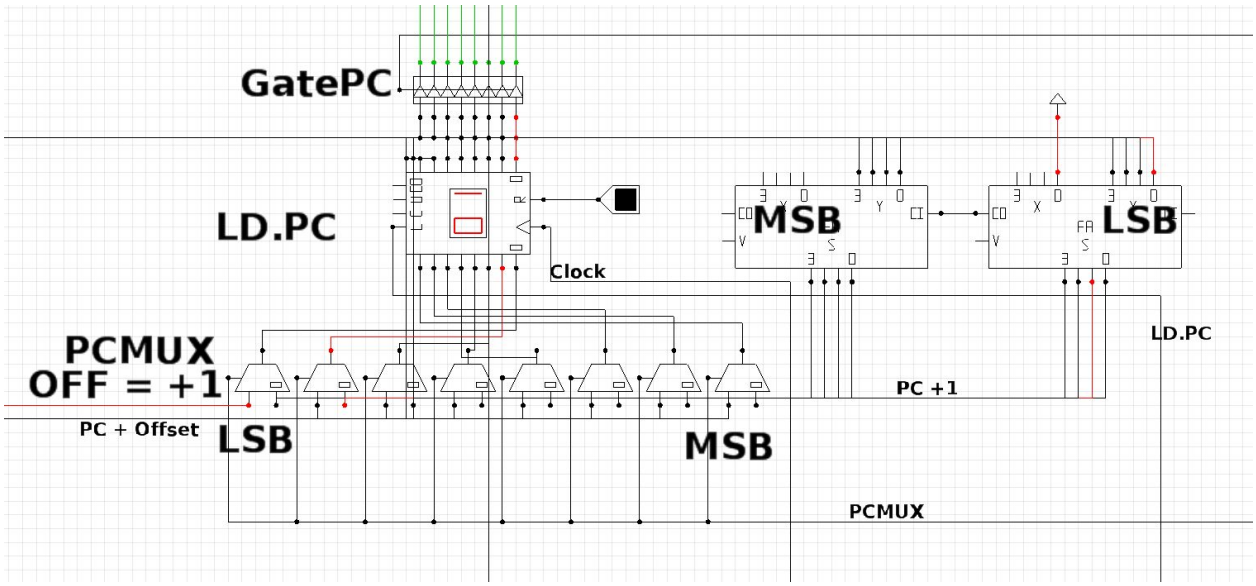
ALU



The ALU's purpose is to take two inputs, X and Y, from the registers, and output the value of either the AND or ADD operation to be stored into a third register, Z. The AND function is completed by implementing the Boolean function Z[i] = (X[i])(Y[i]) using two-input AND gates. The output of each AND gate is fed into an 8-bit bus. The ADD function is executed by feeding the two inputs into two four-bit full adders, and the output is sent to an 8-bit bus. Since the word length of our design is 8-bits, the carry-out of the higher magnitude adder is disregarded. Every time the ALU is called, both operations will be completed. This does not affect the efficiency of our design because both processes occur concurrently. The desired output is determined by selecting between the ADD bus and the AND bus, which is done with a self-built two bit decoder, that activates the AND bus on an input of 0 to the ALUK and activates the ADD bus on an input of 1 to the ALUK. Our design was based on the several comparators that we have looked at in class, but we replaced the compare function with an ADD function. Danish developed the conceptual design for this piece, and Seamus implemented it.

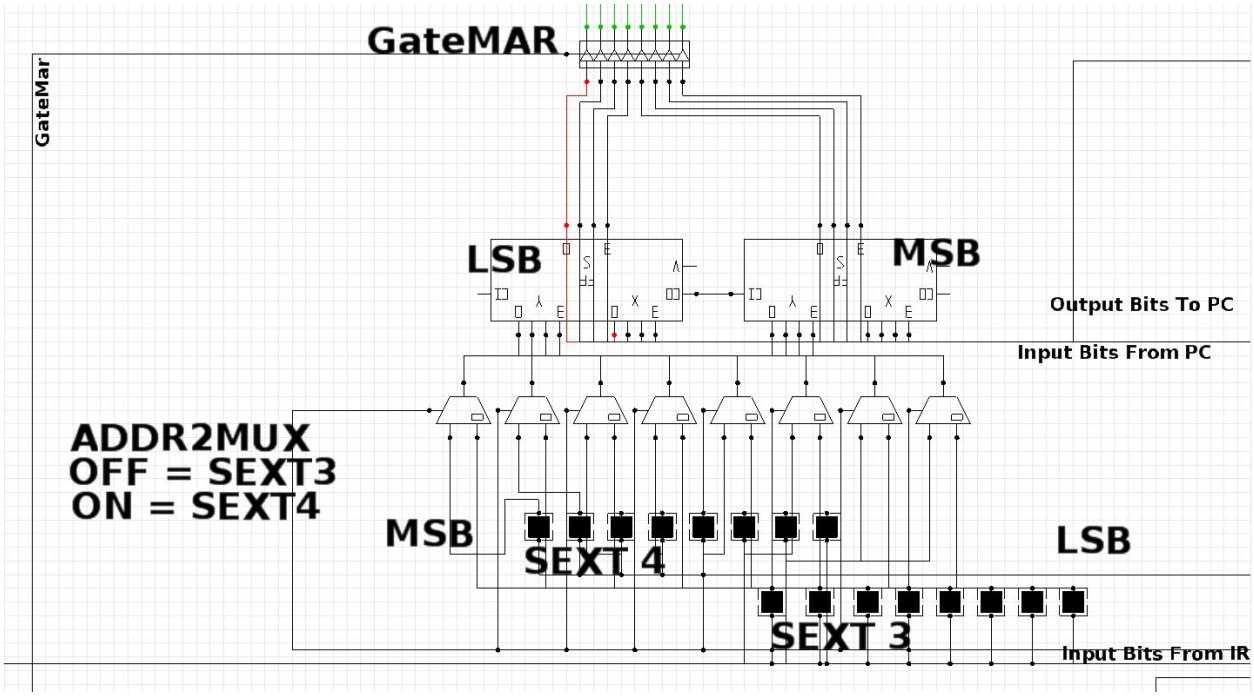
In the image above, the two 8-bit words are the inputs, and the lone control is the ALUK. The bottom word is the output.

PC



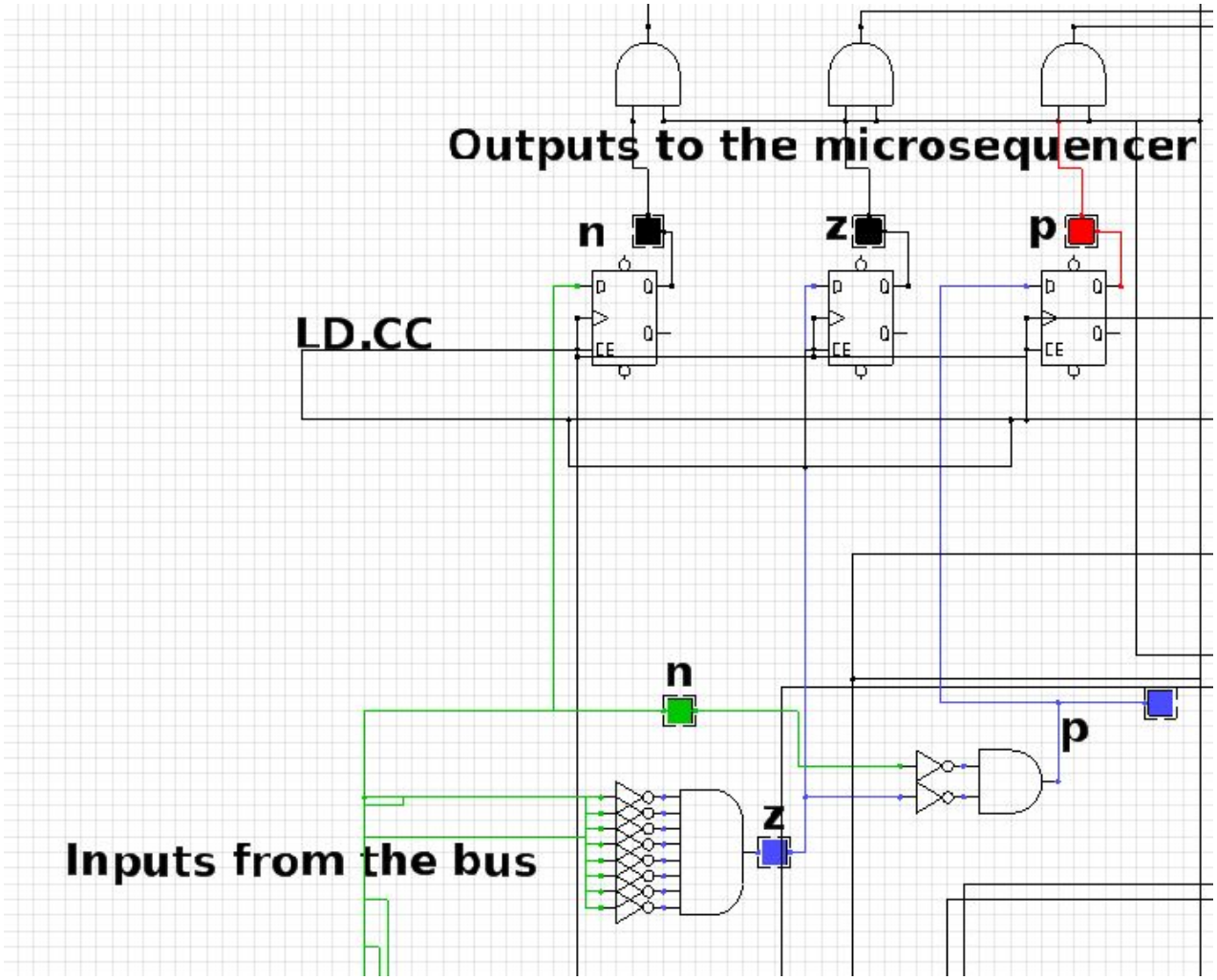
This is the PC and the PCMUX that decides what to load the PC with. It consists of the PC register, the PC incrementer (the two adders), multiplexer, and 8-bit tri state bus.The PC register holds the current PC. A multiplexer was chosen for the inputs to the PC because in the current ISA for this processor, there are only two ways the PC can be changed, 1. The PC and an offset coming from the instruction, or the 2. Incremented PC. Multiplexers are perfect for these kind of functions. The decision to choose between the two PC inputs comes from the PCMUX control signal bit where 0 asserts choosing the incremented PC (PC + 1 lines) and 1 asserts choosing the incremented PC + offset (PC + Offset lines). To prevent the PC from being incremented continuously or being loaded at the wrong time. The LD.PC control signal determines whether the PC can be loaded depending on the current state. The PC register is also connected to the clock, and the button connected to the R allows the PC to be manually reset. The two adders are used to increment PC. The GatePC control signal from the ROM decides whether the PC contents should be put on to the bus or not. Tim and Seamus implemented the initial part of the PC. Later Tim and Danish finish up the whole PC and added the PCMUX stuff.

PC Offset Calculator



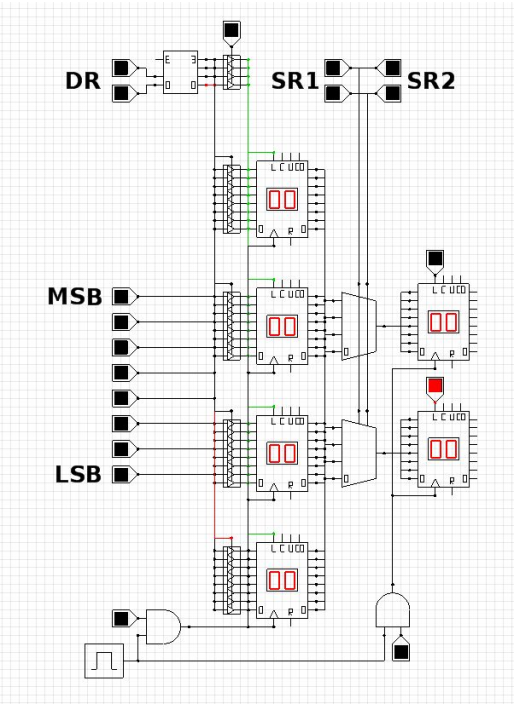
This component takes the offset bits from the instructions and adds them to the current incremented PC. It consists of two 4 bit adders, multiplexers, and a 8-bit tri state bus. Since the required ISA and instructions only had two types of PC offset, a 2 bit multiplexer was chosen to select which offset should be computed. It either selects the sign extended PC offset 3 or the sign extended PC offset 4 depending on the ADDR2MUX control signal. To make the sign extension, the MSB of each bit string to be sign extended was just hardwired to the remaining bits in the 8 bit string. The lights in the middle are there just to visually confirm the sign extension is working. The diagrams in Appendix C showed distinct SEXTs depending on the instruction, so we feel that that each type of offset has its own method for getting the sign extension. The inputs the the sign extension portion are just the corresponding bits from the IR. The inputs to the adder are the outputs from the multiplexer and the current PC. The outputs from the adder go to the GateMAR to be loaded into the bus, and they also go to the PCMUX if the offset needs to loaded into the PC. (For a branch instruction). The control signal called GateMAR decides whether the offset + the PC should go on the bus to be loaded into the memory address register. We chose to do the sign extension like this because we feel this is how the book did it. Tim and Danish implemented this component after finishing the PC.

NZP Logic (CC)



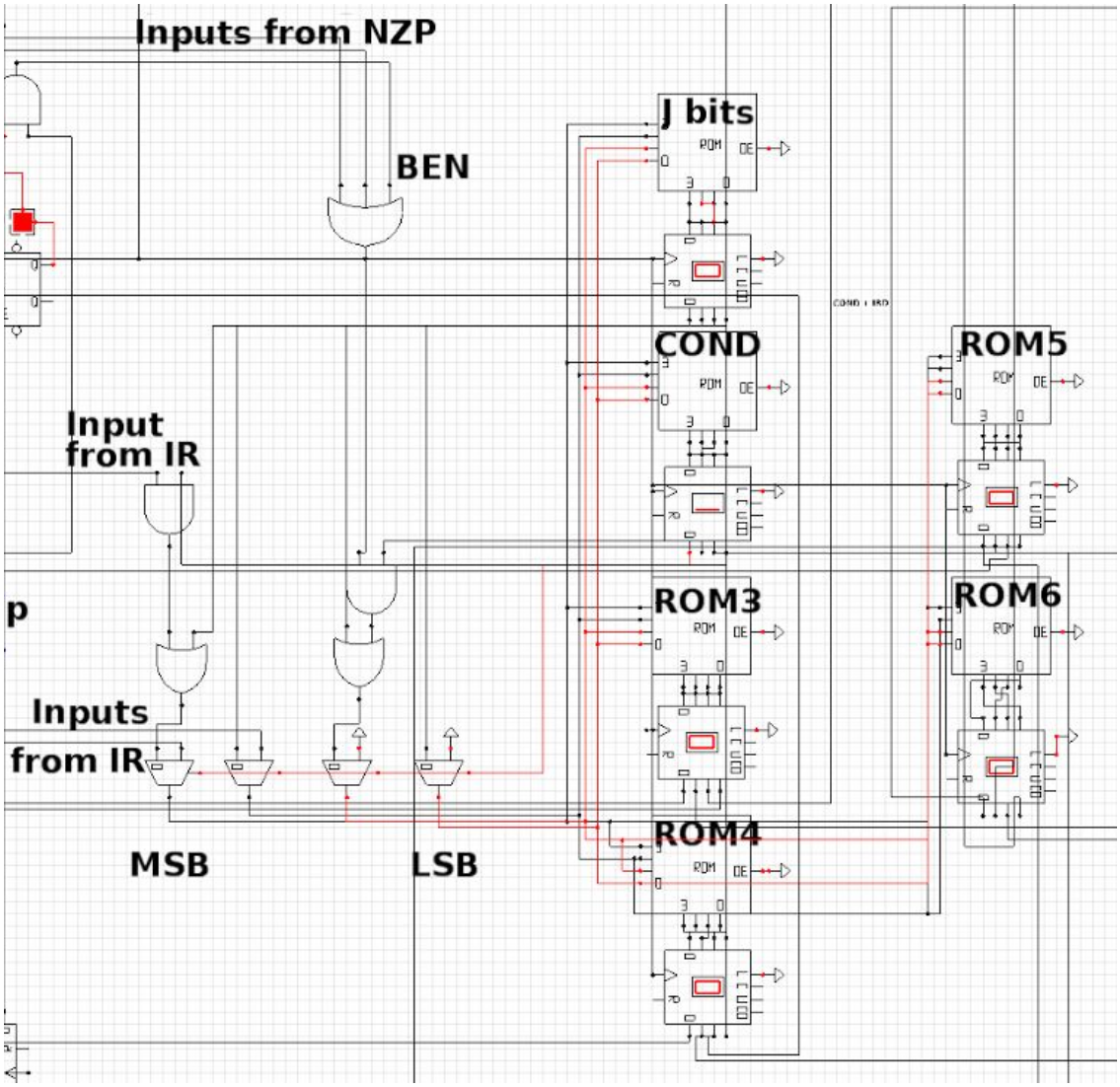
This is the logic that sets the NZP condition codes in the processor. This component was made by Seamus. The wires in the bottom left are the inputs from the bus. They go through a sequential logic circuit to the 3 D flip flops at the top. The flip flops were chosen because they could efficiently store 1 bit of information for each condition code. The circuit at the bottom checks whether the input from the bus is positive, zero, or negative. It does this by first checking to see if it is negative or zero. These are the easiest cases to check. If it is not one of these two, then the input must be positive, so that is what the last AND gate with the two NOT inputs is for. The outputs of this logic go to the corresponding flip flop. When the LD.CC control signal from teh ROM is asserted, the value is loaded into the flip flop. The outputs of each flip flop is ANDed with the corresponding bit in the instruction (from the IR). This will set branch enable (BEN) in the microsequencer. Seamus implemented the NZP logic.

Register File



The register file contains four registers, R0 - R3, in this image ordered from the bottom to the top. The register file takes an 8-bit word as input for the value to be stored, and outputs it to four buses, one for each register. The bus for the correct register to store the value is chosen by a 2-to-4 decoder, that takes input from the instructions stored in the RAM. The decoder is also connected to the load enable, so only the selected register will update its value. If the load enable inputs were not connected to the decoder output, the registers would be set to 0 any time another registers value was being loaded. The load enables can also only be set if the LD.REG bit of the microinstruction is set. SR1 and SR2 are temporary variables used for executing operations, but do not need to be stored permanently. They copy the info from a register selected from the instruction code, and then are output to the ALU. They do not necessarily need to be real registers, but we decided to implement the hardware to make the dataflow easier to follow. The register file was designed and implemented by Danish and Seamus.

Control Unit



Tim designed the microsequencer and implemented the ROMs based on the microinstructions we generated as a group.

The control unit holds the microinstruction of each state, and determines the next state. Each ROM register holds four bits at each address, and each are hooked up to the same four wires that select the address. In this way we can access all 24 bits of each microinstruction. The registers were placed after each ROM to prevent a bug that was caused because the ROMs output high impedance values instead of 0 when their input is low.

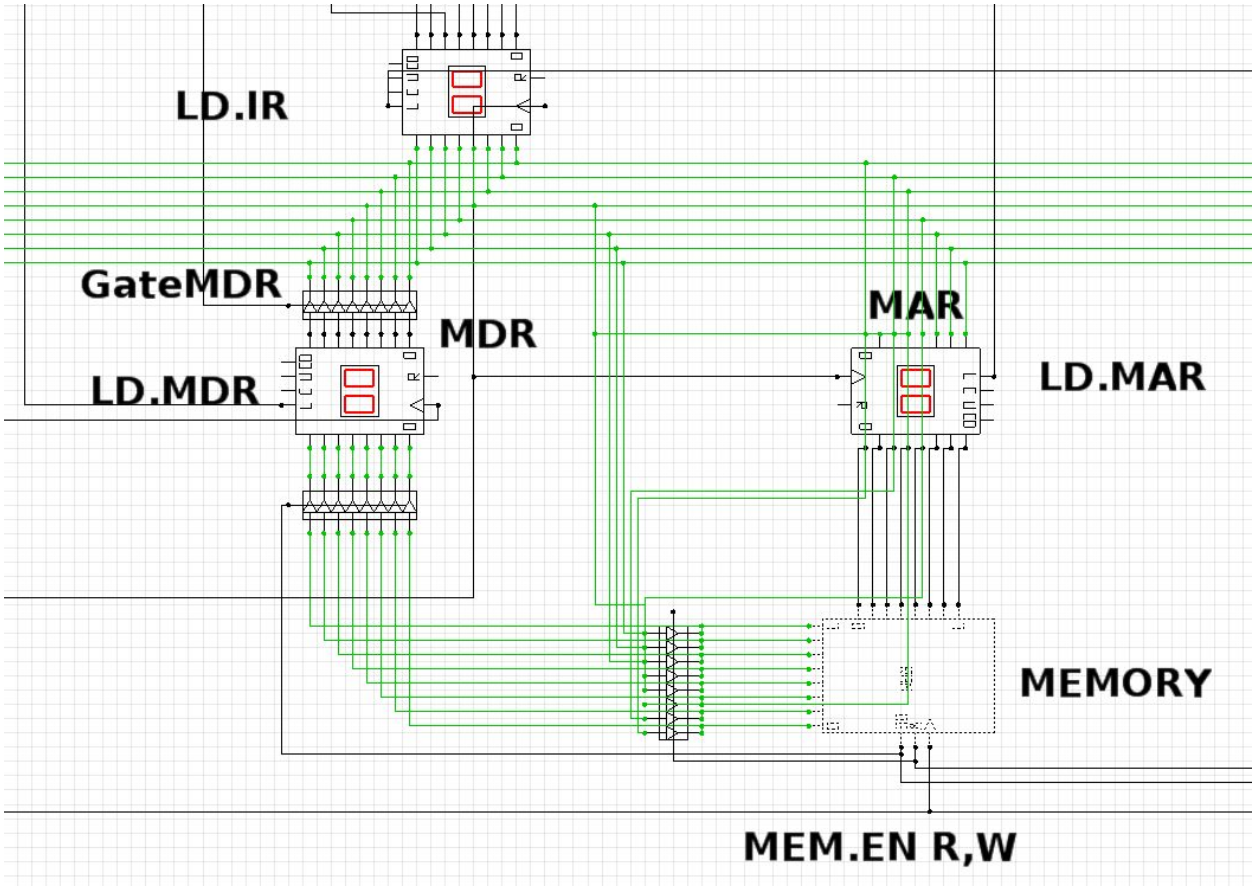
Our machines is based on the idea that one state flows to the next. Interrupting this flow are microsequencer codes, which tell the machine to check if different things have happened, like Branch being enabled (BEN) or the state of the LD/ST selector. This overwrites the bits that are sent back to the ROM, allowing us to choose from many possible next states.

The gates and multiplexers on the left make up the microsequencer. The multiplexer chooses between the instruction register code and the output from the RAM. When the IRD is asserted from ROM2, the inputs from the first two bits of the IR are read along with two ones, jumping to the state of that operation. Otherwise the states flow through normally.

The gates above the leftmost multiplexer check the LD/ST bit of the instruction, and to see if the condition code to check for LD/ST is set to 1. The output of that logic sends the control unit either to state LD (0001) or ST (1001). The gates above the third multiplexer check to see if the BEN is set and that the current state is checking for BEN. If this is true, that bit of the state is overwritten and the machine jumps to the correct state.

The rest of the outputs from ROMS 3-6 are data path control signals. They send bits to the various gates and selectors around the machine based on the instruction being completed. For example, during an ADD the ALU Gate, LD.REG, LD.CC, and ALUK are all set to 1. This allows all the proper bits to flow from the two sources into the ALU, onto the bus, and back into the correct register.

RAM and IR



Tim initially designed the RAM, but worked with Danish on the final product.

This is where the instructions are found and processed. The main purpose of this part of the machine is to read the data at each address, based off the PC. The PC value comes in off the global bus and is loaded into MAR (while LD.MAR enabled). In the next clock cycle, the memory is queried at that address and the value stored there is placed in the MDR (while MEM.EN R and LD.MDR are enabled). In the next cycle, the value in the MDR is placed in the IR where it can be accessed by the microprocessor and the rest of the machine.

These devices are also used for Loads and Stores. In a Load, the MAR tells the RAM which address to get data from. The RAM sends that data to the MDR, which sends it onto the global bus and into the source register. In a Store, a value from a register is fed to the RAM from the global bus, and the MAR tells the RAM where to put it.

The two tri-state busses control when bits are pushed into the RAM and when bits flow out. The bottom one is turned on when the RAM is writing, and the top one is turned on when the RAM is reading. The registers are controlled by signals from the ROM.

Testing

Each component was tested individually before it was added to the main file. Once all the components were connected. After extensive debugging, a test program was used to test all the required operations for the processor. This test program was put into the RAM. It first loads two numbers into two registers. Then it adds the two numbers and puts the result in a different register. After that it stores a value from one of the registers into a RAM location. After that it branches to the AND instruction. This test program. After the ADD instruction the value in R2 was 9 and 7 was stored in the location where 2 was originally. The branch worked. It skipped the two memory locations being used for operand and branched to the AND instruction. For further testing, some of the offsets were changed. For example, #-2 was used in the branch instruction to make sure that the SEXT was working and the branch could instructions could handle negative values.

To test the machine, simply load these hex values into the ***second*** RAM address (i.e. put them in 01. Address 00 is where the machine is when the file is opened). Then turn on the clock.

Instruction	**Hex (RAM value)**
LD R0 #4	44
LD R1 #4	54
ADD R2 R1 R0	A4
ST R1 #1	71
BRp #2	0A
2	02
7	07
AND R2 R1 R0	E4

Work Split

While we did have some components tasked out to individual group members (who are named in the description of each component above), most of the work was done with all group members present. The first few days consisted of whole team discussions about our approach and instruction design. Debugging and editing of each subunit was done by all group members at one computer.

In the end every component in the processor was looked at by each group member at some point in time.

Extra Credit

Adding the extra credit involves making the LD and ST instructions get the address from a register and not computing it by adding the offset in the instruction to the incremented PC. To implement this we could use the same states, except we would have to change some control signals. Instead of asserting GATEMAR and putting the incremented PC + some offset on the bus, we would have the first state in that instruction flow assert the SRCMEM control signal on the 8 bit tri state bus that transmits data from the source registers to the bus. This would put the address stored in the register containing address onto the bus. The correct register would be determined using the bits from the instruction in the IR. The LD.MAR would also be asserted to make sure the address from the bus is loaded in the MAR. After this the OutputEnable control signal for the RAM and the LD.MDR would be asserted to get the data from the RAM into the MDR. The last state would involve asserting LD.REG and GateMDR to get the data onto the bus and into the destination register. The correct destination register would be identified using the instruction bits in the IR.