CSE 141L Final Report

Daniel Pak, A15968786; Quoc-Zuy Do, A15819109

Academic Integrity

Your work will not be graded unless the signatures of all members of the group are present beneath the honor code.

To uphold academic integrity, students shall:

- Complete and submit academic work that is their own and that is an honest and fair representation of their knowledge and abilities at the time of submission.
- Know and follow the standards of CSE 141L and UCSD.

Please sign (type) your name(s) below the following statement:

I pledge to be fair to my classmates and instructors by completing all of my academic work with integrity. This means that I will respect the standards set by the instructor and institution, be responsible for the consequences of my choices, honestly represent my knowledge and abilities, and be a community member that others can trust to do the right thing even when no one is watching. I will always put learning before grades, and integrity before performance. I pledge to excel with integrity.

Daniel Pak
Quoc-Zuy Do

0. Team

Daniel Pak, Quoc-Zuy Do

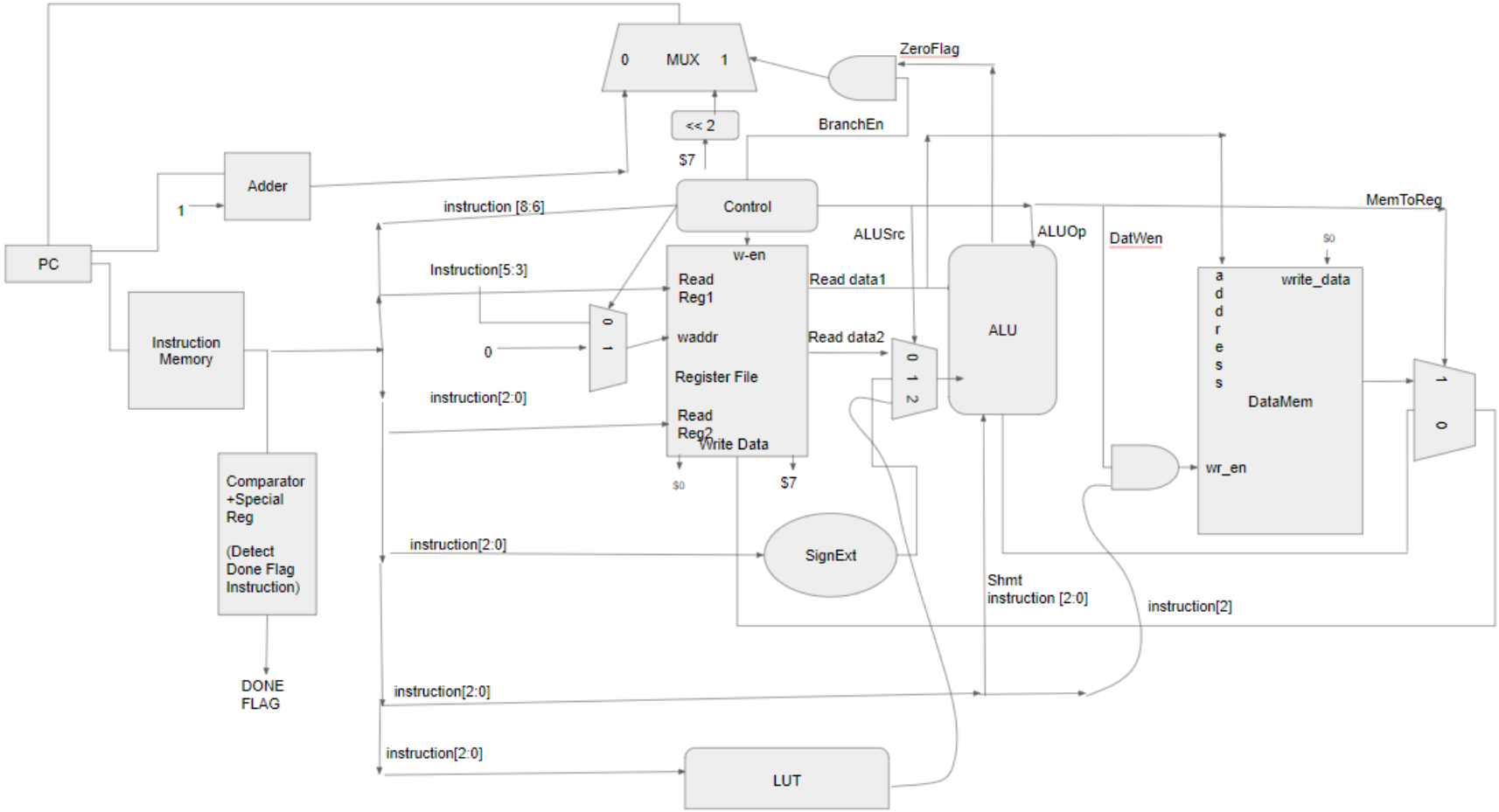
1. Introduction

Name: 2UCIS: 2 Unit Course Instruction Set

The overall philosophy of our ISA design is to be specialized towards forward error correction and pattern recognition of bit strings stored in memory while maintaining a compact instruction set. We specifically wanted to tailor our design so that extracting bits could be done as efficiently as possible as all three programs that run on our MIPs-like architecture require heavy usage of bit extracting. With our current design extracting the MSB/LSB can be done in a single operation. In addition, our design will have a LUT that will be pre programmed to extract a certain set of bit patterns which we will use to calculate parities. Our design philosophy also focuses on manipulating data one word at a time within the registers so data from memory must be loaded to the registers.

Our machine can be classified as a load/store machine with one of the registers acting as a pseudo-accumulator. We reserve one address to load/store into so it serves as an accumulator. The rest of the operations the ISA supports are instructions that perform operations on registers. Since data must be

2. Architectural Overview



3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	3 bit opcode, 3 bit source/destination operand register, 3 bit source operand register. (The first operand register will also be the place where we store the result)	OR, XOR, BEQ
L	3 bit opcode, 3 bit source/destination operand register, 3 bit LUT input vector.	LAND
S	3 bit opcode, 3 bit register operand, 1 bit sel (select), 2 bit shmt. For MA instructions. Sel = 0 is a load, Sel = 1 is a store and shmt = 00. For BS instructions. Sel = 0 is a left shift, Sel = 1 is a right shift, and shmt = shift amount.	MA (Memory Access), BS (Bit Shift) *MA can do either a load or a store depending on the value of the sel bit. *BS can do either left shift or right shift depending on the value of sel bit.
I	3 bit opcode, 3 bit operant destination register, 3 bit immediate (The operant register is both a source and the destination where we store results)	ADDI

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
XOR = bitwise xor	R	3 bit opcode (000) 3 bit operand/destination register (X) 3 bit operand register (X)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` `xor R0 R1` ⇔ `000_000_001` # after xor instruction, R0 now holds `0b1000_0001`	Implied destination register is always first operand.
RXOR = reduction xor	R	3 bit opcode (001) 3 bit destination register (X) 3 bit operand register (X)	Assume R1 has `0b1001_0000` `rxor R0 R1` ⇔ `001_000_001` # after xor instruction, R0 now holds `0b0000_0000`	Implied destination register is always first operand
OR = bitwise or	R	3 bit opcode (010) 3 bit operand/destination register (X) 3 bit operand register (X)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` `or R0 R1` ⇔ `010_000_001` # after or instruction, R0 now holds `0b1001_0001`	Implied destination register is always first operand.
BEQ	R	3 bit opcode (011) 3 bit operand/destination register (X) 3 bit operand register (X)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b0001_0001` `beq R0 R1` ⇔ `011_000_001` # after beq instruction, if operand registers are equal then we jump to the specified mem address. Else we proceed to the next instruction	If operand1 == operand 2 then we will jump to the address stored in R7. It is implied that the address we jump to is always going to be preloaded into this register before we make the jump.

LAND LUT - bitwise AND.	L	3 bit opcode (100)
		3 bit destination register
		3 bit LUT input vector
			# Assume R0 has `0b1111_1111`
			
			`land R1 0 ⇔ 100_001_000`
			
			# after land instruction, R1 will contain `8'b1000_1110`
			
			`R1 = R0 AND OUTPUT OF LUT`
			
			`***Special Case***`
			`In order to terminate the program/ raise done flag the user can input`
			
			`-> land R0 7`
			
			`The only LUT patterns we use in our ISA are from 0 - 4. Since LUT pattern 7 is unused, we make this machine code encoding raise the done flag.`
			
			`-> 100_000_111`
				The purpose of this operation is to mask bits (extract them) and place them in another register. We are going to be masking bits that we read from memory so the source is implied to always be R0.
				
				The operand within the LAND instruction specifies where the place the result.
				
				The LUT takes in a 3-bit input vector which maps to 8 different output values. These values will then get bitwise ANDed with the value in R0 and saved. This allows us to have a preset number of bit patterns that we can extract in a single instruction.
				
				LUT Mappings are as Such
				(000) -> `8'b1000_1110`
				(001) -> `8'b0110_1101`
				(010) -> `8'b0101_1011`
				(011) -> `8'b0000_0101`
				(100) -> `8'b0001_0111`
				(101) -> `8'b0000_0001`
				
				(111) -> SPECIAL CASE (see left column)

MA = (lw/sw)	S	3 bit opcode (101)
		3 bit operand reg
		1 bit selector
		2 bit shamt
			#Assume R1 has `0b0000_0001`
			
			`ma R1 l ⇔ 101_001_0_00`
			`ma R1 s ⇔ 101_001_1_00`
			
			# after memAcc with l, we load data from address 0000_0001 into register 0
			
			#after memAcc with s, we store data from register 0 to datamem at address 0000_0001
				Pseudo-accumulator-like design. Register 0 is reserved for loads/stores from memory. When we specify memAcc R1, l that means we go to the address in datamem and load the data there to register 0. Likewise, using s for the selector will store the value in register 0 to data memory.
				
				Selector bit = 0 for loads and 1 for stores
				
				Shmt = 00 for this instruction.

BS = (LS/RS)	S	3 bit opcode (110)
		3 bit operand reg
		1 bit selector
		2 bit shamt
			#Assume R1 has `0b0000_1000`
			
			`bs R1 l 3` ⇔ `110_001_0_11`
			`bs R1 r 3` ⇔ `110_001_1_11`
			
			# after bs instruction with l, R1 contains `0b0100_0000`
			
			#after bs instruction with r, R1 contains `0b0000_0001`
				The operand register specified in the instruction is the one that gets shifted. L and R are used as selectors for determining whether the shift is to the left or to the right. Shmt is the shift amount. It is limited to 2 bits so the maximum we can shift is 3 bits.
				
				Selector bit = 0 for left and 1 for right

addi	I	3 bit opcode (111) 3 bit operand/destination register (X) 3 bit signed immediate (X)	# Assume R0 has `0b0001_0001` `addi R0 3` ⇔ `111_000_011` # after addi instruction, R0 now holds `0b0001_0101`	The operand register is also the destination. This instruction will add a 3-bit wide signed immediate (value between -4 and 3) to the designated register and save the result to that same register.

Internal Operands

- R0 is reserved for loads/stores from memory. In general the word of data you are working on gets placed here and then stored when you are done manipulating it.
- R1 and R2 are prioritized for storing the read/write addresses respectively, but can be used for general purpose.
- R3-R6 are general purpose registers
- R7 gets loaded with an address prior to a BEQ instruction so it knows where to jump to. Otherwise it can be used as a general purpose register.
- **WE SUPPORT A TOTAL OF 8 REGISTERS**

Control Flow (branches)

- The only branch supported is BEQ (branch on equal). This instruction will compare to operand registers and if they are equal it will set the PC = the address within R7 which by our ISA design must be preloaded with the address to jump to. Target addresses will have to be pre-calculated based on the number of instructions we want to jump back/forward. BEQ will mainly be used to implement a loop and therefore after we have written out our assembly program we will know how many instructions forward we jump beforehand. Since the actual branching instruction works similar to jump register in the sense that the value within the register is loaded directly into memory, there is no maximum branch distance. You can branch anywhere in instruction memory as long as it fits in 8 bits. We cannot accommodate large jumps, so assembly must be written with a restricted jump range in mind.

Addressing Modes

- We support register-indirect addressing. Addresses should be preloaded into R1 and R2 as specified above in our internal operands section. These addresses can then be incremented/decremented using addi with a signed immediate. All three programs allow us to work mosty sequentially in terms of memory addresses so as long as the addresses are being updated properly as we iterate through the calculations there will be no separate address calculations needed.
    - Example: For program 1 we want to begin reading from mem[0] and writing to mem[30]. As we step through the program addi can be used to increment the address stored in the registers so we can address mem[1] and mem[31] as necessary. If we want to go back and read from an earlier value we can also use addi with a negative immediate (because we support signed immediates) and update the address in R0 to be 0b0000_0000 and then we can read from mem[0] again.
        - Want to load LSW, load mem[0]
        - Want to load MSW, addi R0, 1, then load mem[1]
        - Want to load LSW again, addi R0, -1 then load mem[0]
        - Want to store value, store mem[30]

4. Programmer's Model

4.1 How should a programmer think about how your machine operates? Provide a description of the general strategy a programmer should use to write programs with your machine. For example, one could say that the programmer should prioritize loading in the necessary values from memory into as many registers as possible, then perform calculations. Another approach could be loading and writing to memory in between every calculation step. Word limit: 200 words.

The approach the programmer should use is to load one word at a time and then do calculations before storing a word back to memory. In essence you want to be loading and writing to memory in between every calculation step. However, multiple loads may be needed before we can construct the word that we want to write back to memory. The programmer should be aware of the conventions that must be followed such as R0 being reserved for load/store and that R7 must be loaded with the address we want to jump to prior to a BEQ instruction. The programmer should also know that the instructions are heavily limited in size so immediates have only a small range (-4 to 3) and you can only shift a maximum of 3 bits in either direction at a time. Larger bit shifts will require multiple instructions. Lastly, the programmer should also be aware of the fact that the R-type instructions are “destructive” in that the first operand will also be the destination so the first operand will get overwritten with the result of the operation. Since you can only jump between 0 and 255, code should be written to jump to 255 to begin conditional statements, then jump back to before 255 to execute.

4.2 Can we copy the instructions/operation from MIPS or ARM ISA? If no, explain why not? How did you overcome this or how do you deal with this in your current design? Word limit: 100 words.

No, we cannot copy the instructions directly from MIPS/ARM because these instructions are too large and we are limited to 9-bit instructions. We had to reduce the max number of instructions we want to support to 8, giving us a 3 bit opcode. We will also only support 8 registers (3 bits to address) which also means that the max number of operands is 2. This also means that our R-type instructions must be destructive as there are not enough bits in the instruction to specify 2 operands and a destination register in the same instruction.

4.3 Will your ALU be used for non-arithmetic instructions (e.g., MIPS or ARM-like memory address pointer calculations, PC relative branch computations, etc.)? If so, how does that complicate your design?"

We do not use the ALU for MIPS/ARM-like memory address pointer calculations. We do have an instruction that is not strictly arithmetic, which is the LAND instruction. This instruction takes the value in a register and bitwise ANDs with a value in from a LUT instead of another register. This would complicate our design because we now need to create a LUT module as well as route parts of the instructions in order to ‘address’ into the LUT and get the 8-bit pattern out of the LUT and into the ALU for usage as an operand. In addition, our shifting operations are all logical shifts and not arithmetic shifts however that does not complicate the ALU design greatly as we are just filling in the shift-in bits with 0 instead of the signed MSB.

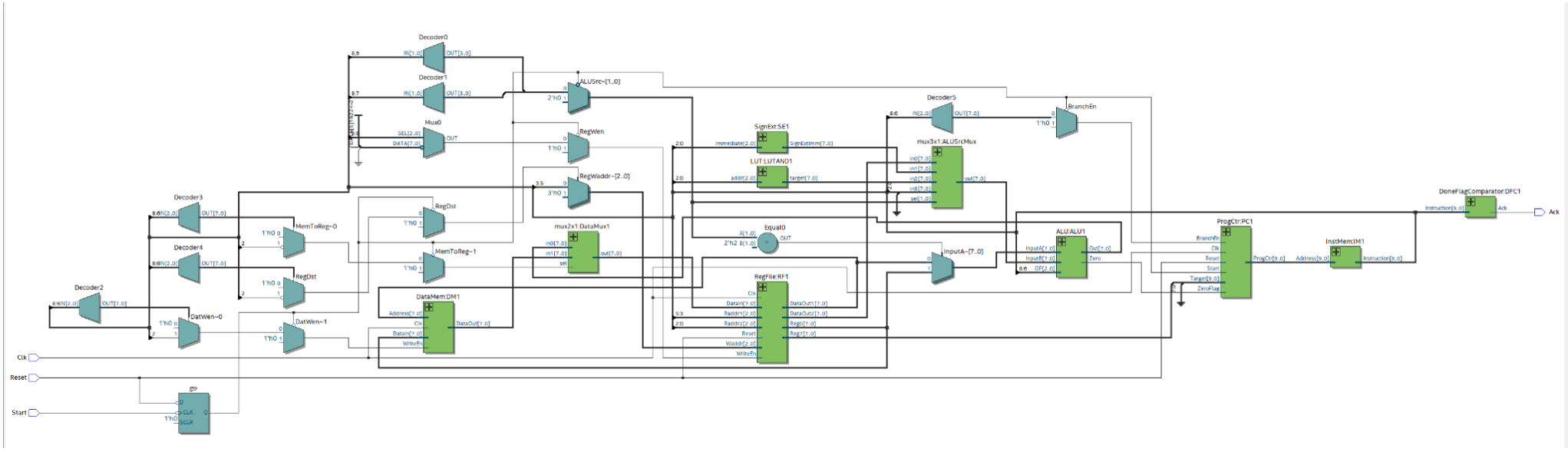
5. Individual Component Specification

a. Top Level

Module File Name: TopLevel.sv

Functionality Description: This contains all the connections between modules. Has one output port for Ack, the done flag, from the DUT.

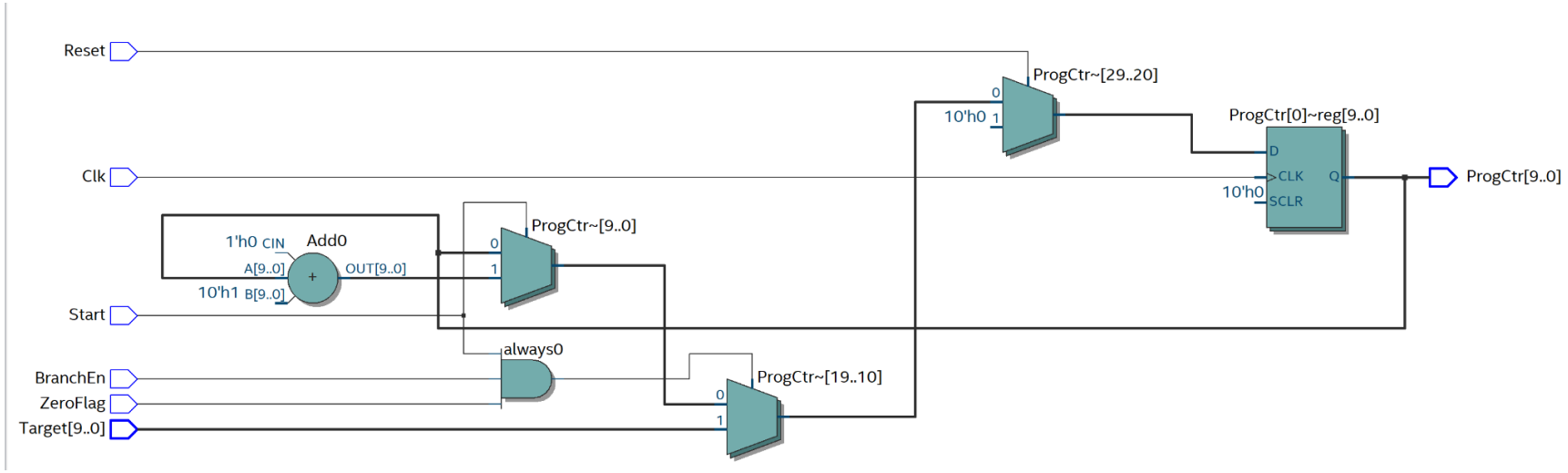
Schematic:



b. Program Counter

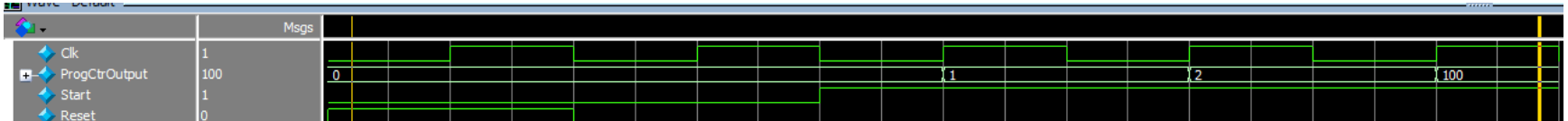
Module File Name: ProgCtr.sv
Module Test Bench File Name: ProgCtr_tb.sv
Functionality Description: This module is a readable and writable memory module/register that manages the address of the instruction to be executed next. It has one input port where it reads in either the address it just outputted on the last instruction by 1 or a new target address to jump to. The PC will only change on the positive edge of the clock. Whichever is inputted is handled by the mux. It will output a 10 bit wide address to instruction memory

Schematic:



Testbench Description: This testbench initializes all the inputs to the PC to 0 except for reset which starts high. On the first posedge clock, since reset is high the testbench expects that the program counter output will be 0. An assert statement is used to make sure this is indeed the case. Then we go on to test normal increment behavior. A couple of cycles get passed, and we check to see that after every posedge of the clock whether the PC is incrementing by 1 each time. This test passes as well. Then we go on to test the jump/branch behavior. IN order to branch our PC needs both BranchEn from the controller and the ZeroFlag from the ALU to be high, these values are set HIGH and on the next posedge clock we test to see if a target of 'd100 is loaded. This indeed does work and the PC is able to accommodate branch behavior.

Timing Diagram:

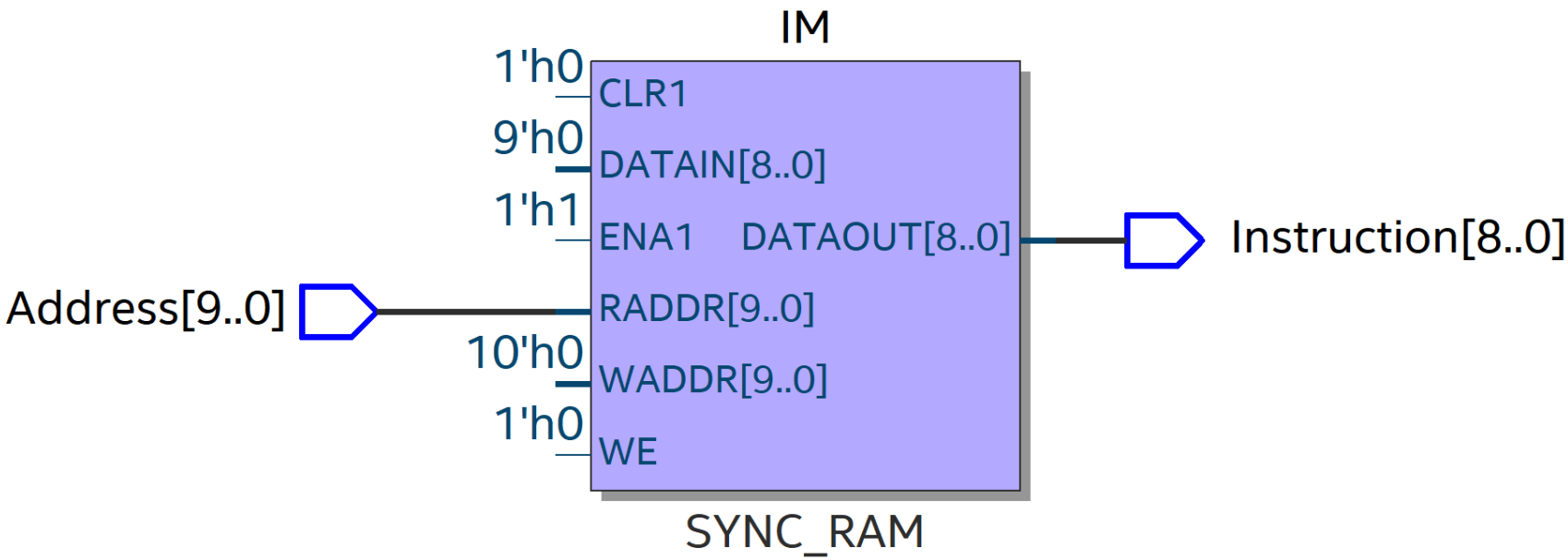


c. Instruction Memory

Module File Name: InstMem.sv

Functionality Description: This module is a read only memory module that serves as the instruction memory for our processor design. It is a combinational module that is not synchronized with the clock. It has one input port which is the Address we want to read from and it will output the corresponding output. The instruction memory is 9 bits wide to support our 9 bit machine code, and 2^10 elements deep. Thus the address will have to be 10-bits wide and the output is a 9-bit instruction. The instruction memory gets initialized in the initial begin block by reading the data from the file and loading into the IM 'core'. This is the only time instruction memory can be written to and modified.

Schematic:

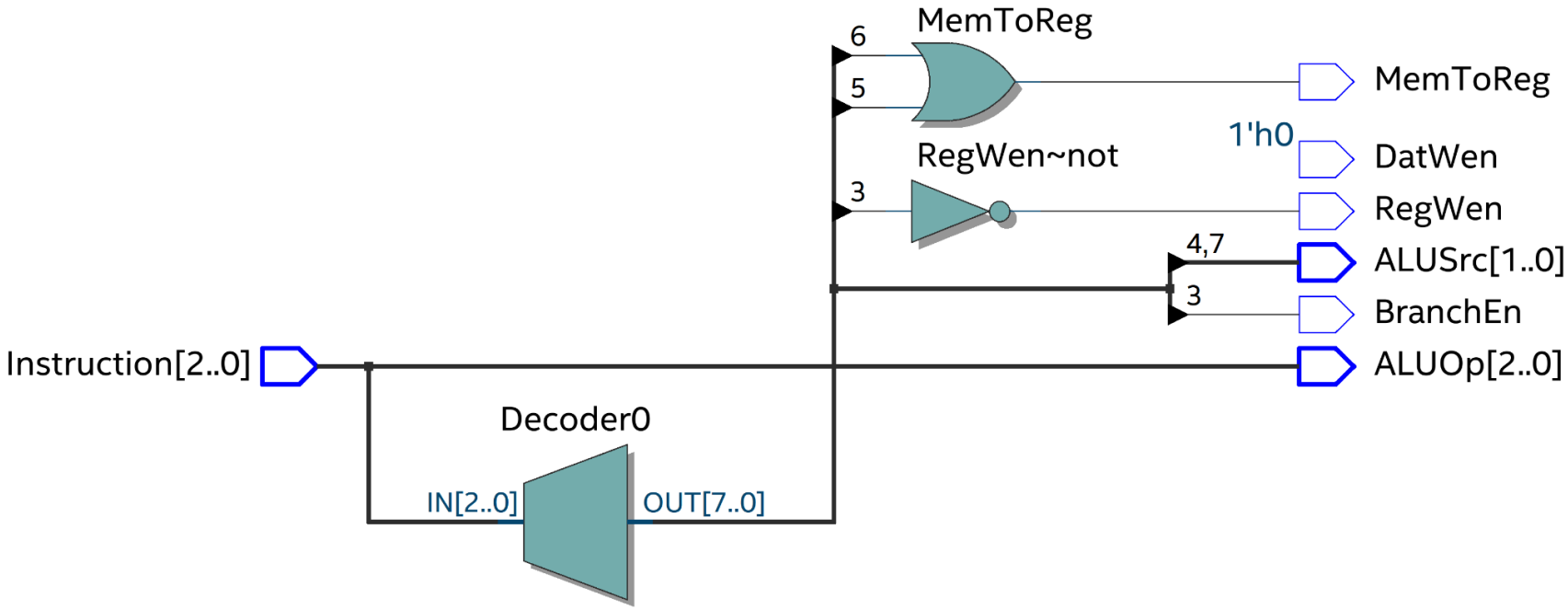


d. Control Decoder

Module File Name: Decoder.sv

Functionality Description: This module is our control decoder module. It has one input which is the machine code instruction and has outputs: DataWrEn, MemWrEn, ALU, jump, and ack (DONE flag).

Schematic:

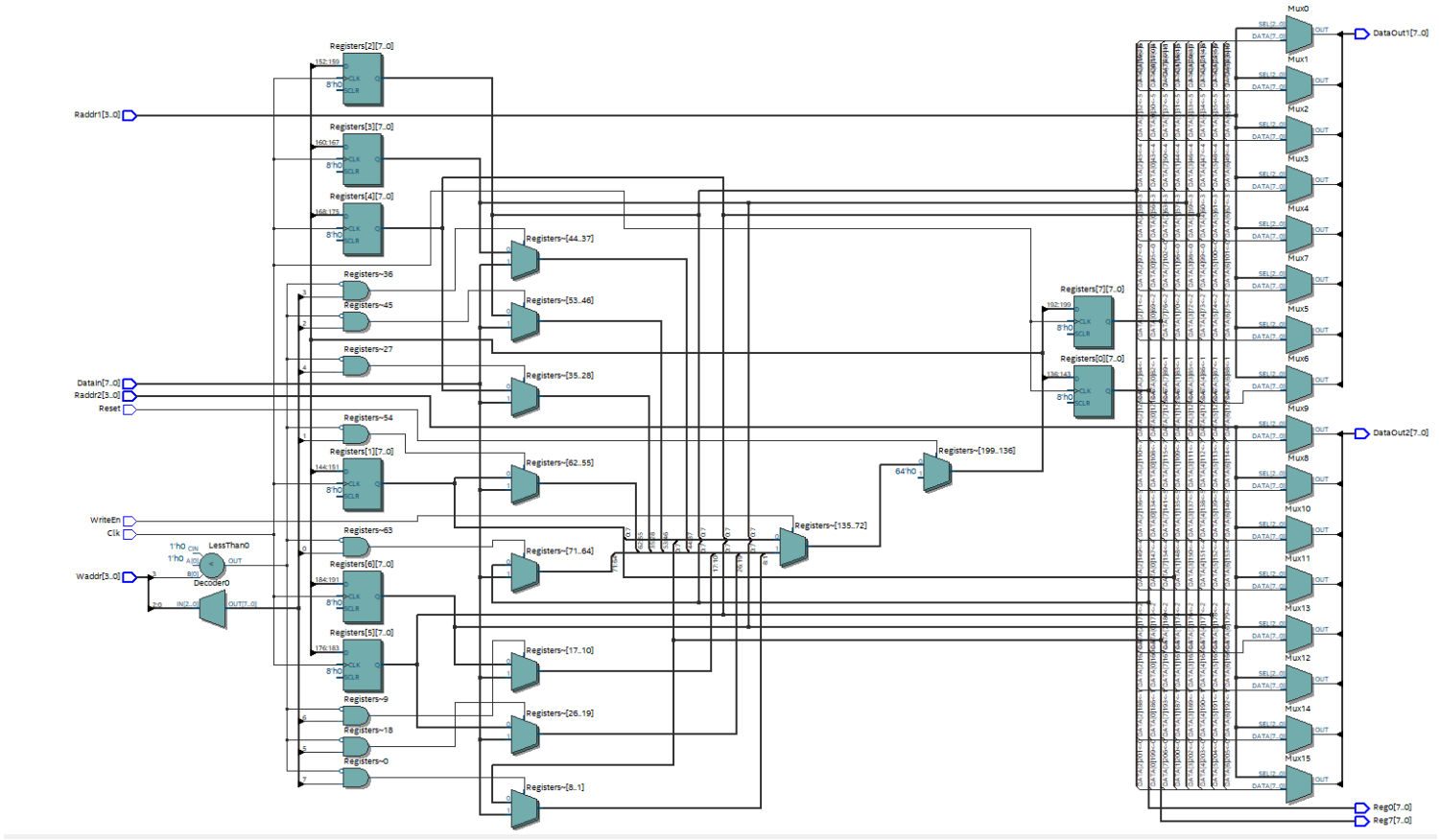


e. Register File

Module File Name: RegFile

Functionality Description: This module is our regfile. It is 8-bits wide and 8 registers deep. There are 2 read ports and a single sequential write port. We have 2 read addresses (each 3 bits wide to address 8 registers), the dataOut will be set to the corresponding read address. In addition to these normal read ports, we have additional output ports that always output the contents of R0 and R7 which are used for store word and beq respectively. For our write logic, it is sequential and we can only write on the posedge of the clock. There is a single data in port and a write address. On the posedge of the clock, the corresponding register that is addressed by the write address gets set to the value of the data in. There is also a reset signal that is also synchronous. On the posedge clock, if the reset signal is high then all the registers will be cleared (set to 0).

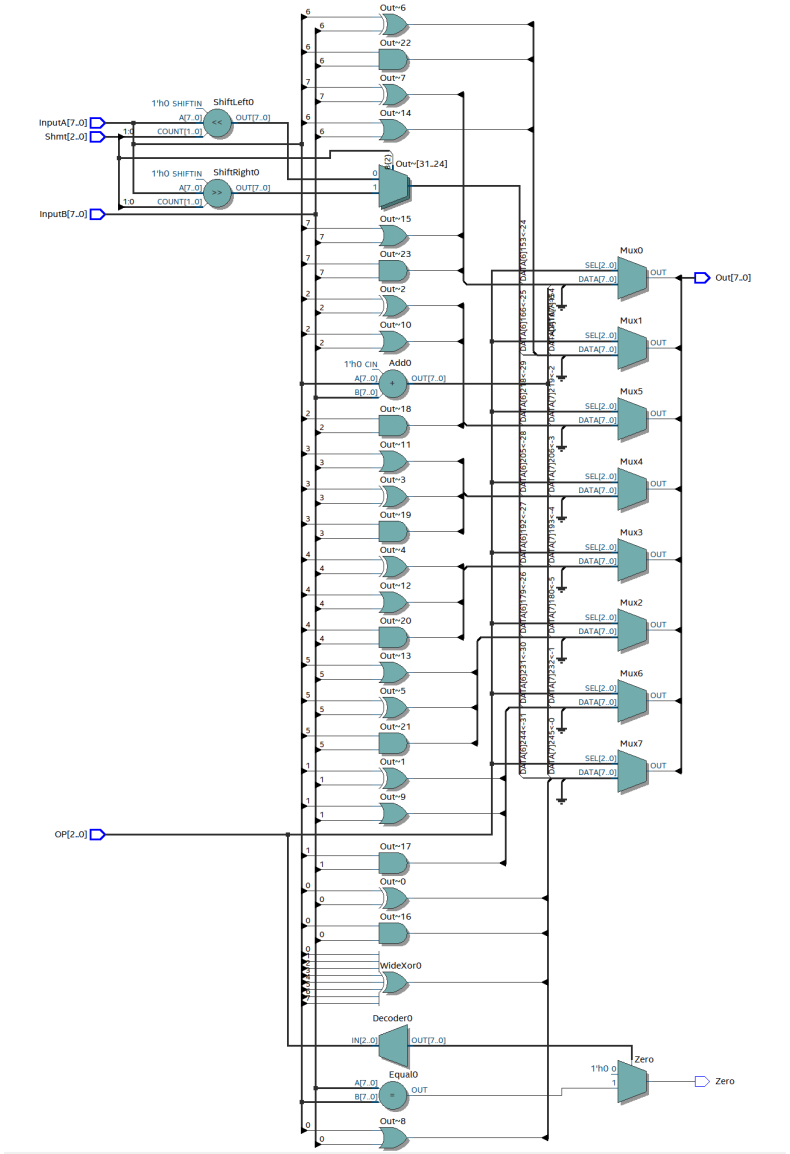
Schematic:

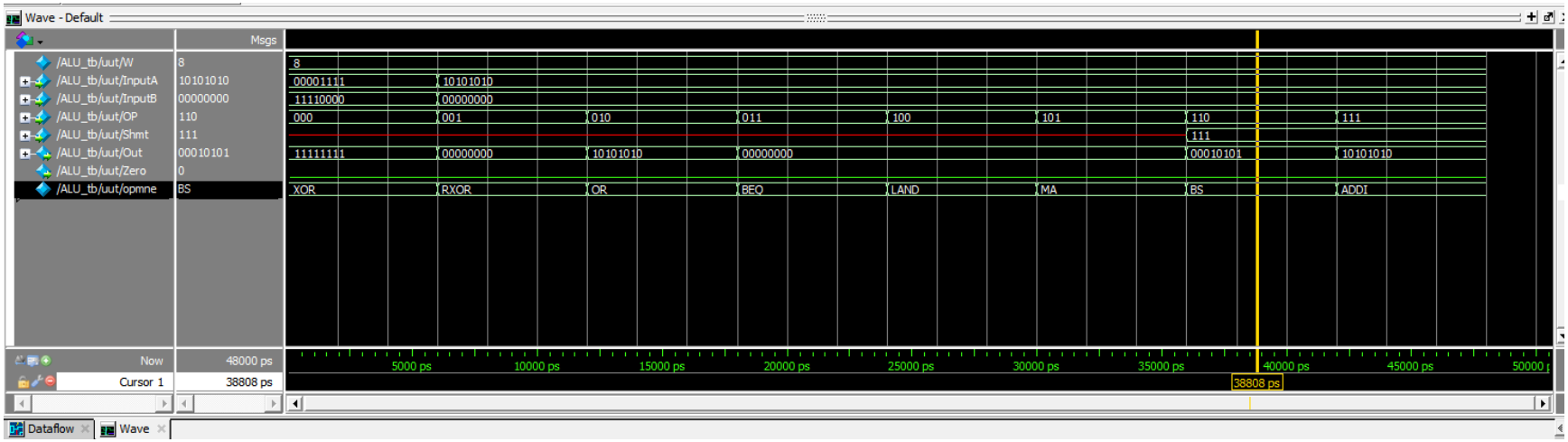


f. ALU

Module File Name: ALU.sv
Module Testbench File Name: ALU_tb.sv

Functionality Description: This module is our ALU, it is 8-bits wide and





Testbench Description: This testbench runs through each of the different operations that the ALU performs for each instruction based on the value of opmne which is a 3 bit logic taken from the opcode of our instructions. It also takes in the corresponding control signals which affect certain aspects of the instruction. It first runs an XOR on inputs A and B, then storing the result in out. For RXOR, the ALU performs a reduction XOR on input A, ignoring B, and stores it in out.

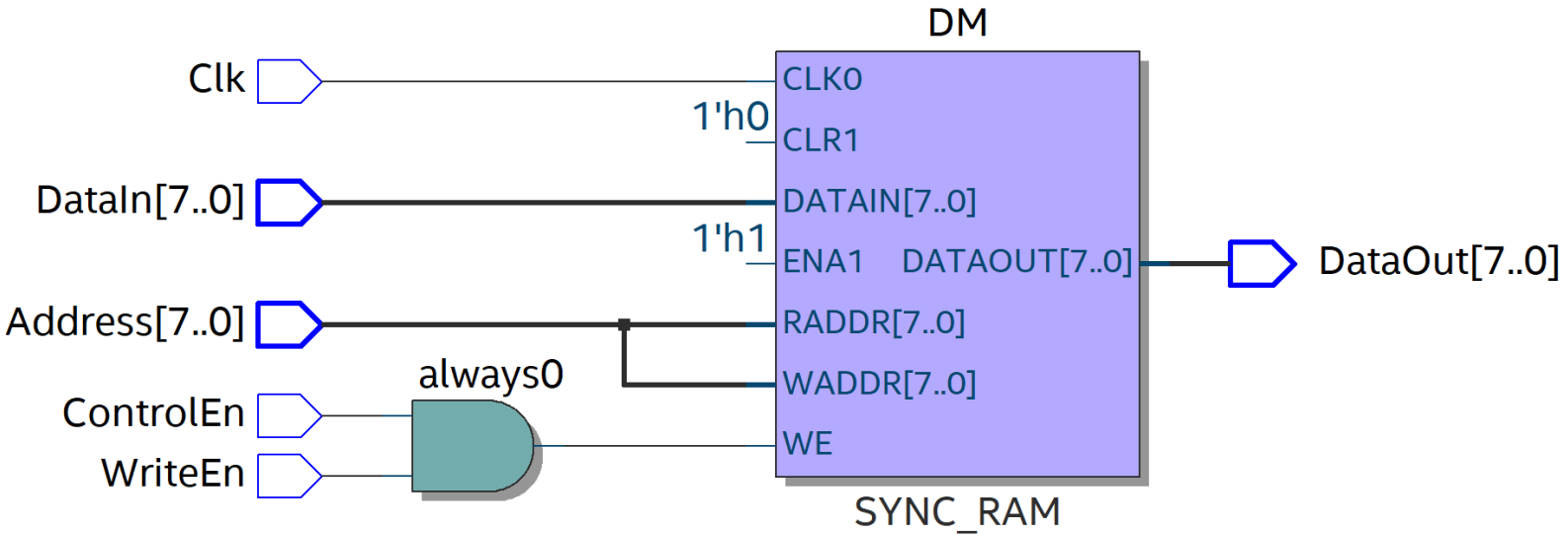
ALU Operations: Every single one of our instructions rely on ALU operations except for MA. For example, bitwise XOR, Addi, BS, OR, and RXOR are all instructions that are also ALU operations that we compute. BEQ is an instruction that relies on the ALU to perform an equality check on the two input operands. The LAND instruction requires an additional LUT in order to function but the idea is the same, the LAND will provide an 8-bit input to the ALU and this get's bitwise AND'ed with the other operand that is fed into the ALU. The memory access (MA) instructions do not rely on the ALU as we are using register indirect addressing and don't need to calculate base + displacement in the ALU. All the ALU operations tested passed.

g. Data Memory

Module File Name: DataMem.sv

Functionality Description: This data memory is a sequential readable and writable memory. As per the architecture guidelines the datamemory is limited in size to 2^8 entries, which means that the address is 8 bits long. We also have a DataIn port that will be used to store data mem. There is no read enable, you can read at any time from this memory. There are technically 2 control signal inputs, the first control signal ControlEn comes from the Control Decoder, it is set HIGH when the instruction is a MA (memory access) type instruction. However, you also need the WriteEn bit from the selector bit in the instruction. This is instruction[2]. You must AND these two control signals together and can only write to datamem only if these two signals are HIGH and on the positive edge of the clock. As a quick refresher, our MA instruction can do both loads and stores it just requires an additional selector bit to determine which one you do, this writeEn is that additional selector bit.

Schematic:

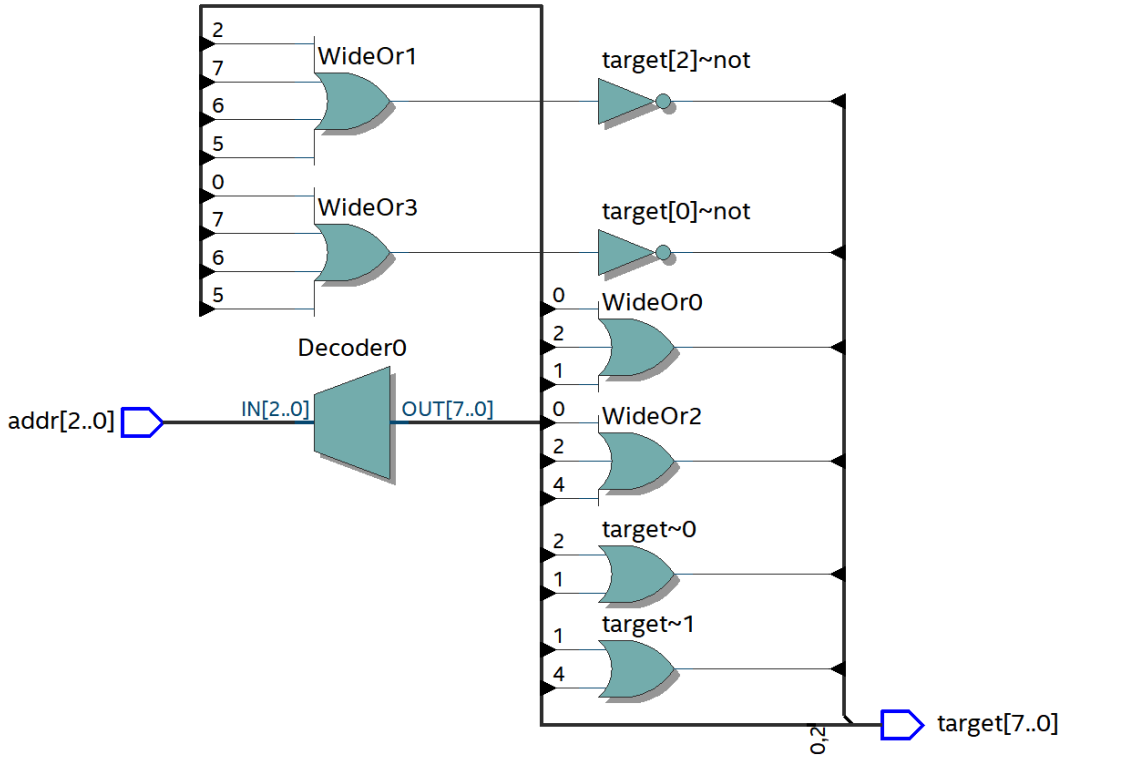


h. Lookup Tables

Module File Name(s): LUT

Functionality Description: This module is a look up table intended to be used in bitwise AND operation with the value in R0 and saved. This allows us to have a preset number of bit patterns that we can extract in a single instruction.

Schematic:



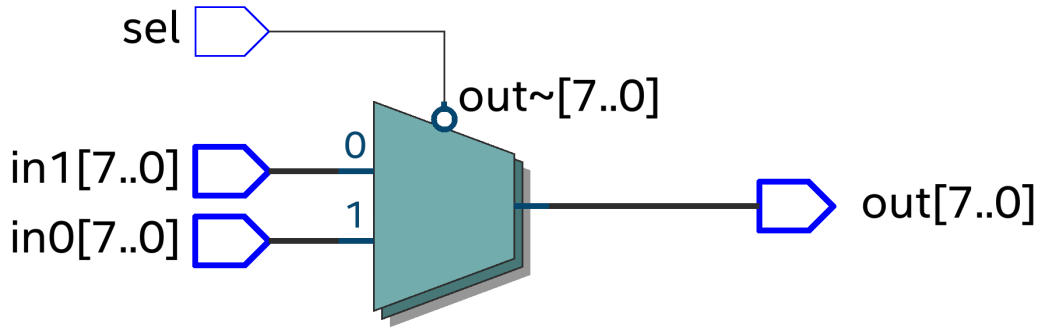
i. Muxes

Module File Name(s): mux2x1, mux3x1

Functionality Description: These two modules work as multiplexers. Our design will require both 2 to 1 muxes and a 3 to 1 mux. The 2x1 mux will take 2 different 8-bit vectors as inputs, and a 1-bit selector. If the selector = 0, then then the output will be set to the input labeled in0. If selector = 1 then the output will be set equal to the input labeled in1.

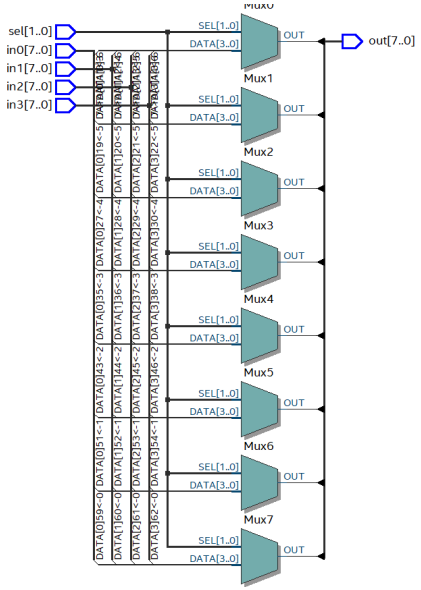
For the 4x1 mux, it will have 4 (8-bit vector) inputs and a single 2-bit selector that chooses which of the inputs gets passed through the output.

Schematic 2x1:



,

Schematic 4x1:

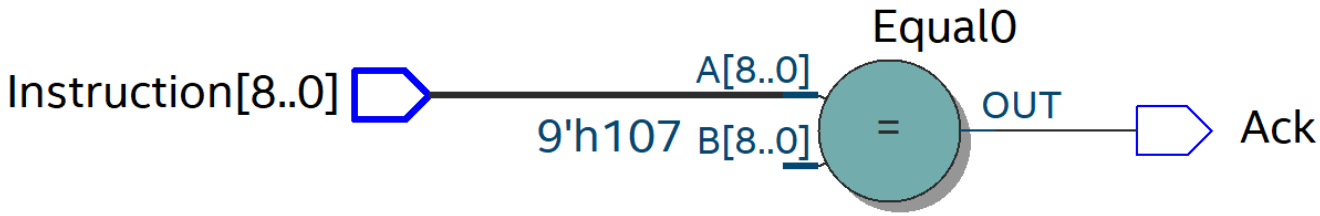


j. Other modules

Module File Name: DoneFlagComparator.

Functionality Description: This module compares the instruction that was just fetched with the code 9'b100000111. If it is equal then output the done flag. This instruction is reserved just for the done flag and cannot be used during the normal operation of the program.

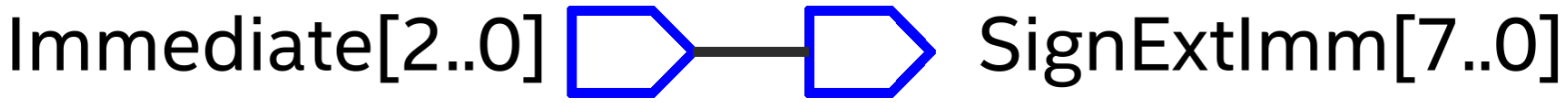
Schematic:



Module FIle Name: SignExt

Functionality Description: We support an addi instruction with a signed immediate value. THis allows us to do both adding and subtracting operations which is useful for incrementing/decrementing memory addresses that we manipulate. This module just takes a signed 3-bit input and assigns it to a signed 8-bit output. The sign extension is handled automatically by system verilog.

Schematic:



6. Assembler

TODO. Explain your assembler. What language is it written in? Describe what it does. Provide a usage description. If you were to give your assembler to someone else, what would be the exact steps to generate machine code from your assembly? Give us an example of input to and output from your assembler. If your assembler does anything beyond what a 'normal' assembler would be expected to do, explain this as well.

Our assembler is written in Python. The assembler works by parsing command line arguments. The first command line argument is the source file, this is where we will be putting our assembly code in a text format. The second command line argument is the destination file which is where the assembler will be saving the results of our machine code translations.

First, our assembler assumes that every single line of assembly code will also have a corresponding comment next to it.
It also assumes that there the instructions are laid out as designated as in the Machine Specification section above where instructions are spaced as separators.

Assuming all constraints are satisfied, the assembler will parse the input assembly code line by line. It will separate the assembly code into an instruction and a comment. The instruction is then split up using python's .split and the opcode and operands are split and translated separately. The assembler will look at what the instruction opcode is and then translate it to its corresponding opcode and append it to a writeline which starts as an empty string and will hold the translated machine code. For all of our instructions the next three bits will always be some register operand so the next section of the instruction will be translated to the corresponding opcode. Then based on the instruction type the last three bits are either: a register, an immediate, or a select bit + shift amount. A series of conditional statements are used to identify which of these it is and the correct translations are made and then appended to writeline. Lastly the comment that was split off earlier gets translated into a system verilog comment and appended to writeline and then writeline is written to the newly opened file.

Usage:

In Command Line put: python assembler.py assembly_file.txt machine_code1

xor r0 r1 #asdfasdfasdf	000000001 //asdfasdfasdf
rxor r0 r1 #asdfaasdfsdf	001000001 //asdfaasdfsdf
or r0 r1 #asdfasdfqweraqwer	010000001 //asdfasdfqweraqwer
beq r0 r1 #my comment here	011000001 //my comment here
land r1 0 #my comment here	100001000 //my comment here
ma r1 l #my comment here	101001100 //my comment here
ma r1 s #my comment here	101001100 //my comment here
bs r1 l 3 #my comment here	110001011 //my comment here
bs r1 r 3 #my comment here	110001111 //my comment here
addi r0 3 #my comment here	111000011 //my comment here

7. Program Implementation

Program 1 Pseudocode

```
//Read and Write Address hold the addresses of the LSW, address of MSW is off by 1
ReadAddress = 0
WriteAddress = 30
1
For (readAddress < 30):
      LSW = datamem[ReadAddres]
      MSW = datamem[ReadAddress + 1]

      //Select the bits you need from the MSW and LSw, reduction xor them, and then xor the results
      P8 = ^(MSW[2:0]) ^ (^(LSW[7:4]))
      P4 = ^(MSW[2:0]) ^ (^(LSW[7])) ^ (^(LSW[3:1]))
      P2 = ^(MSW[2:1]) ^ (^(LSW[6:5])) ^ (^(LSW[3:2])) ^ (^(LSW[0]))
      P1 = (MSW[2]) ^ (^(MSW[0])) (^(LSW[6]) ^ (^(LSW[4:3])) ^ (^(LSW[1:0]))

      //Since P8 =  ^(b11:b5) this 'cancels' with the b11:b5 needed in P0.
      //We only need b11:b4 and P1,P2,P4
      P0 = ^(LSW[3:0]) ^ P4 ^ P2 ^ P1

      MSW_Write = MSW << 5                      //MSW_Write = b11 b10 b9 0 0 0 0 0
      MSW_Write[0] = P8                        //MSW_Write = b11 b10 b9 0 0 0 0 p8
      MSW_Write[4:1] = LSW[7:4]                 //MSW_Write = b11 b10 b9 b8 b7 b6 b5 p8
      datamem[WriteAddress + 1] = MSW_Write     //Write to datamem

      LSW_Write = LSW << 4                      //LSW_Write = b4 b3 b2 b1 0 0 0 0
      LSW_Write[0] = P0
      LSW_Write[1] = P1
      LSW-Write[2] = P2
      LSW_Write[4] = P4
      LSW_Write[3] = LSW[0]                  //LSW_Write = b4 b3 b2 p4 b1 p2 p1 p0
      datamem[WriteAddress] = LSW_Write      //Write to datamem
```

```
//Add 2 to move onto the next message.
WriteAddress += 2
ReadAddress += 2
```

Program 1 Assembly Code

```
//Can Pre-Load registers with these values during initialization of program
R1 = 1                      //contains address of LSW of current message, MSW is off by 1
R2 = 31                     //contains the write address of LSW of the current message.

TOP OF LOOP (INSTRUCTION 0):          //Save a branch destination to go back to here when we
loop
//Get p8
ma    R1, l                           //load MSW into R0
rxor  R6, R0                          //R6 = ^(b11:b9)
addi  R1, -1                           //R1 = 0. R1 now contains LSW address
ma    R1, l                           //load LSW
bs    R0, r, 3                        //
bs    R0, r, 1                        //R0 = 0 0 0 0 b8 b7 b6 b5
rxor  R4, R0                          //R4 = ^(b8:b5)
xor   R4, R6                          //R4 = R4 ^ R6 = ^(b11:b5) = p8

//Write msw out
addi  R1, 1                           //R1 = 1.
ma    R1, l                           //Load MSW
bs    R0, l, 3                        //
bs    R0, l, 2                        //R0 = b11 b10 b9 0 0 0 0 0
xor   R5, R5                          //Clear R5
xor   R5, R0                          //R5 = b11 b10 b9 0 0 0 0 0
addi  R1, -1                          //R1 = 0
ma    R1, l                           //Load LSW
bs    R0, r, 3                        //
bs    R0, r, 1                        //R0 = 0 0 0 0 b8 b7 b6 b5
bs    R0, l, 1                        //R0 = 0 0 0 b8 b7 b6 b5 0
or    R0, R5                          //R0 = b11 b10 b9 b8 b7 b6 b5 0
or    R0, R4                          //R0 = b11 b10 b9 b8 b7 b6 b5 p8
ma    R2, s                           //Store MSW
addi  R2, -1                          //R2 = 30 (address of LSW)
```

```
//Get p4
ma    R1, l                    //Load LSW, r0 = b8:1--------------35
land  R6, 0                    //R6 = b8 0 0 0 b4 b3 b2 0 (MASKING)
rxor  R6, R6                   //Parity flag reduction XOR. R6 = ^(b8,b4,b3,b2)
rxor r5, r5                    //R5 = ^(b11:9)
xor   R5, R6                   //R5 = ^(b11:8,b4,b3,b2) = p4
bs    R5, l, 3
bs    R5, l, 1                 //R5 =  0 0 0 p4 0 0 0 0

//Get p2
addi r1 1                      //R1 = 1 (MSW address)
ma r1 l                        //R0 = 0 0 0 0 0 b11 b10 b9
bs    R0, r, 1                 //R0 = 0 0 0 0 0 0 b11 b10
rxor  R6, R0                   //R6 = ^(b11:10)
addi  R1, -1                   //R1 = 0
ma    R1, l                    //Load LSW
land  R4, 1                    //R4 = 0 b7 b6 0 b4 b3 0 b1
rxor  R4, R4                   //R4 = ^(b7,b6,b4,b3,b1)
xor   R4, R6                   //R4 = ^(b11,b10,b7,b6,b4 b3,b1) = p2
bs    R4, l, 2                 //R4 = 0 0 0 0 0 p2 0 0
or    R4, R5                   //R4 = 0 0 0 p4 0 p2 0 0

//Get p1
land  R5, 2                    //R5 = 0 b7 0 b5 b4 0 b2 b1
rxor  R5, R5                   //R5 = ^(b7,b5,b4,b2,b1)
addi  R1, 1                    //R1 = 1 (MSW)
ma    R1, l                    //Load MSW
land  R6, 3                    //R6 = 0 0 0 0 0 b11 0 b9
rxor  R6, R6                   //R6 = ^(b11,b9)
xor   R5, R6                   //R5 = ^(b11,b9,b7,b5,b4,b2,b1) = p1
bs    R5 l 1
or    R4, R5                   //R4 = 0 0 0 P4 0 p2 p1 0

//Get p0
addi  R1, -1                   //R1 = 0
```

```
ma    R1, l                         //Load LSW
bs    R0, l, 3
bs    R0, l, 1                      //R0 = b4 b3 b2 b1 0 0 0 0
rxor  R6, R0                        //R6 = ^(b4,b3,b2,b1)
rxor  R5, R4                        //R5 = ^(p4,p2,p1)
xor   R5, R6                        //R5 = p0
or    R4, R5                        //R4 = 0 0 0 p4 0 p2 p1 p0 (ALL PARITY BITS ARE IN R4)

//Write LSW out
bs    R0, r, 3
bs    R0, r, 1                      // 0 0 0 0 b4 b3 b2 b1
land  R3, 5                         //R3 = 0 0 0 0 0 0 b1
Bs    R0, r, 1                      //R0 = 0 0 0 0 0 b4 b3 b2 (
xor   R6, R6                        //Clear R6
xor   R6, R3                        //R6 = R3 = b1
bs    R6, l, 3                      //R6 = 0 0 0 0 b1 0 0 0
or    R4, R6                        //R4 = 0 0 0 p4 b1 p2 p1 p0
bs    R0, l, 3
bs    R0, l, 2                      //R0 = b4 b3 b2 0 0 0 0 0
or    R0, R4                        //R0 = b4 b3 b2 p4 b1 p2 p1 p0
ma    R2, s                         //Store LSW

//Increment read/write address to next LSW
addi  R1, 3
addi  R2, 3

//End of loop
xor   R6, R6                        //clear R6
addi  R6, 3                         //R6 = 0000 0011
bs    R6, l, 2                      //R6 = 0000 1100
addi  R6, 3                         //R6 = 0000 1111
bs    R6, l, 1                      //R6 = 0001 1110
addi R6, 1
xor   R7, R7                        //clear R7
Addi R7 1                           //0000 0001
Bs R7 l 3                           //0000 1000
```

```
Bs R7 l 3                       //0100 0000

beq  R1, R6                     //If R1 = R6, jump to address in R7
xor  R7, R7                     //Clear R7
beq  R7, R7                     //Compare with itself, unconditional jump to top of loop
```

INSTRUCTION 255:
done

Program 2 Pseudocode

```
//Read and Write Address hold the addresses of the LSW, address of MSW is off by 1
ReadAddress = 30
WriteAddress = 0

For (readAddress < 60):
      encodedLSW = datamem[ReadAddres]
      encodedMSW = datamem[ReadAddress + 1]

      decodedMSW = encodedMSW >> 5
      decodedLSW = {encodedMSW[4:1], encodedLSW[7:5], encodedLSW[3]

      //Select the bits you need from the decodedMSW and decodedLSW, reduction xor them, and then xor the
      //results. This is same as program 1 for this part.

      P8 = ^(decodedMSW[2:0]) ^ (^(decodedLSW[7:4]))
      P4 = ^(decodedMSW[2:0]) ^ (^(decodedLSW[7])) ^ (^(decodedLSW[3:1]))
      P2 = ^(decodedMSW[2:1]) ^ (^(decodedLSW[6:5])) ^ (^(decodedLSW[3:2])) ^ (^(decodedLSW[0]))
      P1 = (decodedMSW[2]) ^ (^(decodedMSW[0])) (^(decodedLSW[6]) ^ (^(decodedLSW[4:3])) ^
      (^(decodedLSW[1:0]))

      //Since P8 =  ^(b11:b5) this 'cancels' with the b11:b5 needed in P0.
      //We only need b11:b4 and P1,P2,P4
      P0 = ^(encodedLSW[3:0]) ^ P4 ^ P2 ^ P1


      givenParities = {encodedMSW[0],encodedLSW[4],encodedLSW[2:0]}   //{P8, P4, P2, P1, P0}
      generatedParities = {P8,P4,P2,P1,P0)
      differenceVector = givenParities ^ generatedParities


      //No Error
      if(differenceVector = 0)
            Do Nothing
```

```
     //PARITY ERROR: Case where P0 is wrong, and 1 other parity is wrong
     Else if(differenceVector[0] == 1 && (differenceVector[4:1] == 0001, 0010, 0100, 1000,
0000))
          decodedMSW[6] = 1

     //DOUBLE ERROR: Case where P0 is right, but one or more other parities are wrong
     Else if(differenceVector[0] == 0 && differenceVector[4:1] != 0000)
          decodedMSW[7] = 1

     //DATA ERROR: Else we have a data error that we need to fix
     //case where P0 looks wrong, but two or more other P look wrong, report fixed
     Else
          decodedMSW[6] = 1

          //differenceVector[4:1] = [p8,p4,p2,p1]
          case(differenceVector[4:1])

               //0011 means that bit position 3 is wrong, which means d1 needs to be fixed
               //d1 is stored in decodedLSW[0] because we have restored the format to
               // 0 0 0 0 0 b11  b10 b9 b8 b7 b6 b5 b4 b3 b2 b1

               0011: decodedLSW[0] = decodedLSW[0] ^ 1         //d1
               0101: decodedLSW[1] = decodedLSW[1] ^ 1         //d2
               0110: decodedLSW[2] = decodedKSW[2] ^ 1         //d3
               0111: decodedLSW[3] = decodedLSW[3] ^ 1         //d4
               1001: decodedLSW[4] = decodedLSW[4] ^ 1         //d5
               1010: decodedLSW[5] = decodedLSW[5] ^ 1         //d6
               1011: decodedLSW[6] = decodedLSW[6] ^ 1         //d7
               1100: decodedLSW[7] = decodedLSW[7] ^ 1         //d8
               1101: decodedMSW[0] = decodedMSW[0] ^ 1         //d9
               1110: decodedMSW[1] = decodedMSW[1] ^ 1         //d10
               1111: decodedMSW[2] = decodedMSW[2] ^ 1         //d11


     //At this  point, the decoded MSW/LSW have been fixed or was already correct
```

```
datamem[writeAddress] = decodedLSW
datamem[writeAddress + 1] = decodedMSW

ReadAddress += 2
WriteAddress += 2
```

Program 2 Assembly Code

```
// Register preload
R1 = 30              //Contains read address of LSW of current message, MSW is LSW + 1
R2 = 0               //Contains the write address of LSW of current message

//TOP OF LOOP (INSTRUCTIOn 0)
addi R1 1                              //R1 = LSW + 1 (MSW) 31 in first iteration
ma R1 l                                //R0 = b11 b10 b9 b8 b7 b6 b5 p8
land R7 5                              //Extract P8: R7 = p8
bs R7 l 3
bs R7 l 2                              //R7 = 0 0 p8 0 0 0 0 0

//Save b8 b7 b6 b5 because we are about to overwrite it
xor R5 R5                              //Clear R5
bs R0 r  1                             //R0 = 0 b11 b10 b9 b8 b7 b6 b5
bs R0 l, 3
bs R0 l, 1                             //R0 = b8 b7 b6 b5 0 0 0 0
xor R5 R0                              //R5 = b8 b7 b6 b5 0 0 0 0

//Get decoded MSW
ma R1 l                                //Load MSW
bs r0 r 3
bs r0 r 2                              //R0 = 0 0 0 0 0 b11 b10 b9
ma R1 s                                //STORE DECODED MSW TO READ ADDRESS (31 initially)

//Extract the other generated parities store in R7
addi  R1 -1                            //R1 =  address of LSW (30 initially)
ma R1 l                                //Load LSW, R0 = b4  b3  b2  p4  b1  p2  p1  p0
land  R6 4                             //R6 = 0 0 0 p4 0 p2 p1 p0
or R7 R6                               //R7 = 0 0 p8 p4 0 p2 p1 p0
```

```
//Get Decoded LSW part 1: get b4 through b1
xor R0 R6                              //R0 = b4 b3 b2 0 b1 0 0 0 (clear the parities)
bs R0 r 3                              //R6 = 0 0 0 b4 b3 b2 0 b1
land R3 5                              //R3 = 0 0 0 0  0  0  0 b1
bs R0 r 1                              //R6 = 0 0 0 0 b4 b3 b2 0
or R0 R3                               //R6 = 0 0 0 0 b4 b3 b2 b1

//Get Decoded LSW part 2: Finish decoded LSW and store
or R5 R0                               //R5 = b8 b7 b6 b5 b4 b3 b2 b1
xor R0 R0                              //Clear R0
xor R0 R5                              //Copy over R5 into R0
ma R1 s                                //store decodedLSW to readAddress (initially 30)

addi r1 1                              //R1 = MSW (initially 31)

-------------------------------------------------------------------------------------------
---TOP OF LOOP (INSTRUCTION 0):        //Save a branch destination to go back to here when we
loop
//Get p8
ma    R1, l                            //load MSW into R0
rxor  R6, R0                           //R6 = ^(b11:b9)
addi  R1, -1                            //R1 = 0. R1 now contains LSW address (initially 30)
ma    R1, l                            //load LSW
bs    R0, r, 3                         //
bs    R0, r, 1                         //R0 = 0 0 0 0 b8 b7 b6 b5
rxor  R4, R0                           //R4 = ^(b8:b5)
xor   R4, R6                           //R4 = R4 ^ R6 = ^(b11:b5) = 0 0 0 0 0 0 0 p8
bs R4 l 3
bs R4 l 2                              //R4 = 0 0 p8 0 0 0 0 0

//Get p4
addi  R1, 1                            //R1 = 1. (initially 31)
ma    R1, l                            //Load MSW
bs    R0, l, 3                         //
bs    R0, l, 2                         //R0 = b11 b10 b9 0 0 0 0 0
```

```
xor   R5, R5                    //Clear R5
xor   R5, R0                    //R5 = b11 b10 b9 0 0 0 0 0
addi  R1, -1                    //R1 = address of LSW (30 initially)
ma    R1, l                     //Load LSW, r0 = b8:1--------------35
land  R6, 0                     //R6 = b8 0 0 0 b4 b3 b2 0 (MASKING)
rxor  R6, R6                    //Parity flag reduction XOR. R6 = ^(b8,b4,b3,b2)
rxor r5, r5                     //R5 = ^(b11:9)
xor   R5, R6                    //R5 = ^(b11:8,b4,b3,b2) = p4
bs    R5, l, 3
bs    R5, l, 1                  //R5 =  0 0 0 p4 0 0 0 0
or    R4 R5                     //R4 =  0 0 p8 p4 0 0 0 0

//Get p2
addi r1 1                       //R1 = MSW address (31 initially)
ma r1 l                         //R0 = 0 0 0 0 0 b11 b10 b9
bs    R0, r, 1                  //R0 = 0 0 0 0 0 0 b11 b10
rxor  R6, R0                    //R6 = ^(b11:10)
addi  R1, -1                    //R1 = 0
ma    R1, l                     //Load LSW
land  R5, 1                     //R4 = 0 b7 b6 0 b4 b3 0 b1
rxor  R5, R5                    //R4 = ^(b7,b6,b4,b3,b1)
xor   R5, R6                    //R4 = ^(b11,b10,b7,b6,b4 b3,b1) = p2
bs    R5, l, 2                  //R4 = 0 0 0 0 0 p2 0 0
or    R4, R5                    //R4 = 0 0 0 p4 0 p2 0 0

//Get p1
land  R5, 2                     //R5 = 0 b7 0 b5 b4 0 b2 b1
rxor  R5, R5                    //R5 = ^(b7,b5,b4,b2,b1)
addi  R1, 1                     //R1 = 1 (MSW)
ma    R1, l                     //Load MSW
land  R6, 3                     //R6 = 0 0 0 0 0 b11 0 b9
rxor  R6, R6                    //R6 = ^(b11,b9)
xor   R5, R6                    //R5 = ^(b11,b9,b7,b5,b4,b2,b1) = p1
bs    R5 l 1
or    R4, R5                    //R4 = 0 0 p8 P4 0 p2 p1 0
```

```
//Get p0

rxor R3 R0                          //R3 = ^b11:b9
addi R1 -1                          //Get address of LSW
ma R1 1                             //Load LSW
rxor r0 r0                          //R0 = ^b8:b1
xor R3 R0                           //R3 = ^b11:b1
xor r5 r5                           // clear r5
xor r5 r7                           // r5 = old parities
bs r5 r 1                           // r5 = p8 p4 p2 p1
rxor r5 r5                          //R5 = ^p8,p4,p2,p1
xor r5 r3                           //R5 = p0 = ^b11:b1 ^ (^parities)
or r4 r5                            //R4 = 0 0 p8 p4 0 p2 p1 p0

------------------------------------------------------------------------------------------
---

// difference vector
xor r4 r7                           //R4 = difference of parities

//bit prep for error
xor   R5, R5                        //Clear R5
addi  R5, 1                         //R5 = 0 0 0 0 0 0 0 1
bs    R5, 1, 3
bs    R5, 1, 2
bs    R5, 1, 1                      //R5 = 0 1 0 0 0 0 0 0

//load MSW
Addi R1 1                           //R1 = MSW (31 initially)
Ma R1 1                             //Load MSW
Addi R2 1                           //R2 = MSW (1 initially)

//Jump to Case 1: NO ERROR
xor   R6, R6                        //Clear R6
xor   R7, R7                        //Clear R7
```

```
addi  R7, 1                          //R7 = 0 0 0 0 0 0 0 1
bs    R7 l 3                         //
bs    R7 l 3                         //R7 = 0 1 0 0 0 0 0 0
Beq   R4, R6                         //If R4 = 0, then jump to instruction 256



//Jump to Case 2: P0 is wrong
addi R7 3
addi R7 1                        //R7 = 68 (new address we are jumping to)
addi R6 1                        //R6 = 0 0 0 0 0 0 0 1
beq r4 r6                        //If P0 is the only one that doesn't match then P0 is wrong.

//Jump to Case 3: P1 is wrong
Addi  R6, 2                          //R6 = 0 0 0 0 0 0 1 1
Beq   R4, R6                         //If p0 and p1 is wrong, then jump to INSTRUCTION 272

//Jump to Case 4: P2 is WRONG
Addi  R6, 2                          //R6 = 0 0 0 0 0 1 0 1
Beq   R4, R6                         //If p0 and p2 is wrong, then  jump to INSTRUCTIOn 272

//Jump to Case 5: P4 is WRONG
bs    R6, l, 2                       //R6 = 0 0 0 1 0 1 0 0
Addi  R6, -3                         //R6 = 0 0 0 1 0 0 0 1
Beq   R4, R6                         //If p0 and p4 is wrong, then jump to INSTRUCTION 272

//Jump to Case 6: P8 is WRONG
bs    R6, l, 1                       //R6 = 0 0 1 0 0 0 1 0
Addi  R6, -1                         //R6 = 0 0 1 0 0 0 0 1
Beq   R4, R6                         //If p0 and p8 is wrong, then jump to INSTRUCTION 272


//Jump to Case 7: D1 is wrong
```

```
xor  R6, R6                          //clear R6
addi  R6, 3                          //R6 = 0 0 0 0 0 0 1 1
addi  R6, 3                          //R6 = 0 0 0 0 0 1 1 0
addi  R6, 1                          //R6 = 0 0 0 0 0 1 1 1
addi R7 3
addi R7 1                            //R7 = 72 (new address we are jumping to is 288)
Beq  R4, R6                          //If p0, p1, p2  is wrong then jump to INSTRUCTION 288



//Jump to Case 8: D2 is wrong
bs   R6, l, 1                        //R6 = 0 0 0 0 1 1 1 0
addi  R6, 3
add  R6, 2                           //R6 = 0 0 0 1 0 0 1 1
addi R7 3
addi R7 1                            //R7 = 76 (new address we are jumping to is 304)
Beq  R4, R6                          //If p0, p1, p4  is wrong then jump to INSTRUCTION 304

//Jump to Case 9: D3 is wrong
addi  R6, 2                          //R6 = 0 0 0 1 0 1 0 1
addi R7 3
addi R7 1                            //R7 = 80 (new address we are jumping to is 320)
Beq  R4, R6                          //If p0, p2, p4  is wrong then jump to INSTRUCTION 320

//Jump to Case 10: D4 is wrong
addi  R6, 2                          //R6 = 0 0 0 1 0 1 1 1
addi R7 3
addi R7 1                            //R7 = 84 (new address we are jumping to is 336)
Beq  R4, R6                          //If p0, p1, p2, p4  is wrong then jump to INSTRUCTION 336


//Jump to Case 11: D5 is wrong
xor r6 r6                            //Clear R6
addi r6 1                            //R6 = 0 0 0 0 0 0 0 1
bs r6 l 3
bs r6 l 2                            //R6 = 0 0 1 0 0 0 0 0
```

```
addi r6 3                          //R6 =   0 0 1 0 0 0 1 1
addi R7 3
addi R7 1                          //R7 = 88 (new address we are jumping to is 352)
Beq  R4, R6                        //If p0, p1, p8  is wrong then jump to INSTRUCTION 352)



//Jump to Case  12: D6 is wrong
addi r6 2                          //R6 = 0 0 1 0 0 1 0 1
addi R7 3
addi R7 1                          //R7 = 92 (new address we are jumping to is 368)
Beq  R4, R6                        //If p0, p2, p8  is wrong then jump to INSTRUCTION 368)

//Jump toCase 13: D7 is wrong
addi r6 2                          //R6 = 0 0 1 0 0 1 1 1
addi R7 3
addi R7 1                          //R7 = 96 (new address we are jumping to is 384)
Beq  R4, R6                        //If p0, p1, p2, p8  is wrong then jump to INSTRUCTION
384)

//Jump to Case 14: D8 is wrong
xor r6 r6                          //R6 = 0
addi r6 3                          //R6 = 0 0 0 0 0 0 1 1
bs r6 l 3                          //R6 = 0 0 0 1 1 0 0 0
bs r6 l 1                          //R6 = 0 0 1 1 0 0 0 0
addi r6 1                          //R6 = 0 0 1 1 0 0 0 1
addi R7 3
addi R7 1                          //R7 = 100 (new address we are jumping to is 400)
Beq  R4, R6                        //If p0, p4, p8  is wrong then jump to INSTRUCTION 400)

//Jump to Case 15: D9 is wrong
addi r6 2                          //R6 = 0 0 1 1 0 0 1 1
addi R7 3
```

```
addi R7 1                           //R7 = 104 (new address we are jumping to is 416)
Beq  R4, R6                         //If p0, p1, p4, p8  is wrong then jump to INSTRUCTION
416)

//Jump to Case 16: D10 is wrong
addi r6 2                           //R6 = 0 0 1 1 0 1 0 1
addi R7 3
addi R7 1                           //R7 = 108 (new address we are jumping to is 432)
Beq  R4, R6                         //If p0, p2, p4, p8  is wrong then jump to INSTRUCTION
432)



//Jamp to Case 17: D11 is wrong
addi r6 2                           //R6 = 0 0 1 1 0 1 1 1
addi R7 3
addi R7 1                           //R7 = 112 (new address we are jumping to is 448)
Beq  R4, R6                         //If p0, p1, p2, p4, p8  is wrong then jump to INSTRUCTION
448)
1
//Jump to Case 18: Double error detected
addi R7 3
addi R7 1                           //R7 = 116 (new address we are jumping to is 464)
Beq  R7, R7                         //Jump to INSTRUCTIOn 464


-------------------------------------------------------------------------------------
--
//Case 1: NO ERROR (INSTRUCTION 256)
Ma R2 s                             //Store MSW as is NO MODS needed
Addi R1 -1                          //R1 = LSW (30 initially)
Addi R2 -1                          //R2 = LSW (1 initially)
Ma R1 l                             //Load LSW
ma R2 s                             //Store LSW as is
Bs R7 l 1                           //R7 = 128  (JUMP TO 128*4)
Beq R7 R7                           //Jump to END of loop (512)
```

```
//Case 2 - 6: PARITY ERROR (INSTRUCTION 272)
Or R0 R5                              //0 1 0 0 0 b11 b10 b9
Ma R2 s                               //Store MSW
Addi R1 -1                            //R1 = LSW (30 initially)
Addi R2 -1                            //R2 = LSW (1 initially)
Ma R1 l                               //Load LSW
ma R2 s                               //Store LSW as is
Xor R7 R7                             //Clear R7
Addi R7 1                             //R1 = 0 0 0 0 0 0 0 1
Bs R7 l 3
Bs R7 l 3
Bs R7 l 1                             //R1 = 1 0 0 0 0 0 0 0
beq r7 r7                             //Jump to END of loop

//Case 7: D1 is WRONG (INSTRUCTION 288)
Or R0 R5                              //0 1 0 0 0 b11 b10 b9
Ma R2 s                               //Store MSW
Addi R1 -1                            //R1 = LSW (30 initially)
Addi R2 -1                            //R2 = LSW (1 initially)
Ma R1 l                               //Load LSW
Xor R7 R7                             //Clear R7
Addi R7 1                             //R7 = 0 0 0 0 0 0 0 1
Xor R0 R7                             //R0 = b8 b7 b6 b5 b 4 b3 b2 !b1
Ma R2 s                               //STORE LSW
Bs R7 l 3
Bs R7 l 3
Bs R7 l 1                             //R1 = 1 0 0 0 0 0 0 0
beq r7 r7                             //Jump to END of loop

//Case 8: D2 is wrong (INSTRUCTIOn 304)
Or R0 R5                              //0 1 0 0 0 b11 b10 b9
Ma R2 s                               //Store MSW
Addi R1 -1                            //R1 = LSW (30 initially)
Addi R2 -1                            //R2 = LSW (1 initially)
```

```
Ma R1 l                                 //Load LSW
Xor R7 R7                               //Clear R7
Addi R7 1                               //R7 = 0 0 0 0 0 0 0 1
Bs R7 l 1                               //R7 = 0 0 0 0 0 0 1 0
Xor R0 R7                               //R0 = b8 b7 b6 b5 b 4 b3 !b2 b1
Ma R2 s                                 //STORE LSW
Bs R7 l 3
Bs R7 l 3                               //R1 = 1 0 0 0 0 0 0 0
beq r7 r7                               //Jump to END of loop
//Case 9: D3 is wrong (INSTRUCTIOn 320)
Or R0 R5                                //0 1 0 0 0 b11 b10 b9
Ma R2 s                                 //Store MSW
Addi R1 -1                              //R1 = LSW (30 initially)
Addi R2 -1                              //R2 = LSW (1 initially)
Ma R1 l                                 //Load LSW
Xor R7 R7                               //Clear R7
Addi R7 1                               //R7 = 0 0 0 0 0 0 0 1
Bs R7 l 2                               //R7 = 0 0 0 0 0 1 0 0
Xor R0 R7                               //R0 = b8 b7 b6 b5 b4 !b3 b2 b1
Ma R2 s                                 //STORE LSW
Bs R7 l 2
Bs R7 l 3                               //R1 = 1 0 0 0 0 0 0 0
beq r7 r7                               //Jump to END of loop

//Case 10: D4 is wrong (INSTRUCTIOn 336)
Or R0 R5                                //0 1 0 0 0 b11 b10 b9
Ma R2 s                                 //Store MSW
Addi R1 -1                              //R1 = LSW (30 initially)
Addi R2 -1                              //R2 = LSW (1 initially)
Ma R1 l                                 //Load LSW
Xor R7 R7                               //Clear R7
Addi R7 1                               //R7 = 0 0 0 0 0 0 0 1
Bs R7 l 3                               //R7 = 0 0 0 0 1 0 0 0
Xor R0 R7                               //R0 = b8 b7 b6 b5 !b4 b3 b2 b1
Ma R2 s                                 //STORE LSW
Bs R7 l 1
```

```
Bs R7 l 3                           //R1 = 1 0 0 0 0 0 0 0
beq r7 r7                           //Jump to END of loop

//Case 11: D5 is wrong (INSTRUCTIOn 352)
Or R0 R5                            //0 1 0 0 0 b11 b10 b9
Ma R2 s                             //Store MSW
Addi R1 -1                          //R1 = LSW (30 initially)
Addi R2 -1                          //R2 = LSW (1 initially)
Ma R1 l                             //Load LSW
Xor R7 R7                           //Clear R7
Addi R7 1                           //R7 = 0 0 0 0 0 0 0 1
Bs R7 l 3                           //R7 = 0 0 0 0 1 0 0 0
Bs R7 l 1                           //R7 = 0 0 0 1 0 0 0 0
Xor R0 R7                           //R0 = b8 b7 b6 !b5 b4 b3 b2 b1
Ma R2 s                             //STORE LSW
Bs R7 l 3                           //R1 = 1 0 0 0 0 0 0 0
beq r7 r7                           //Jump to END of loop

//Case 12: D6 is wrong (INSTRUCTIOn 368)
Or R0 R5                            //0 1 0 0 0 b11 b10 b9
Ma R2 s                             //Store MSW
Addi R1 -1                          //R1 = LSW (30 initially)
Addi R2 -1                          //R2 = LSW (1 initially)
Ma R1 l                             //Load LSW
Xor R7 R7                           //Clear R7
Addi R7 1                           //R7 = 0 0 0 0 0 0 0 1
Bs R7 l 3                           //R7 = 0 0 0 0 1 0 0 0
Bs R7 l 2                           //R7 = 0 0 1 0 0 0 0 0
Xor R0 R7                           //R0 = b8 b7 !b6 b5 b4 b3 b2 b1
Ma R2 s                             //STORE LSW
Bs R7 l 2                           //R1 = 1 0 0 0 0 0 0 0
beq r7 r7                           //Jump to END of loop

//Case 13: D7 is wrong (INSTRUCTIOn 384)
Or R0 R5                            //0 1 0 0 0 b11 b10 b9
Ma R2 s                             //Store MSW
```

```
Addi R1 -1                          //R1 = LSW (30 initially)
Addi R2 -1                          //R2 = LSW (1 initially)
Ma R1 l                             //Load LSW
Xor R7 R7                           //Clear R7
Addi R7 1                           //R7 = 0 0 0 0 0 0 0 1
Bs R7 l 3                           //R7 = 0 0 0 0 1 0 0 0
Bs R7 l 3                           //R7 = 0 1 0 0 0 0 0 0
Xor R0 R7                           //R0 = b8 !b7 b6 b5 b4 b3 b2 b1
Ma R2 s                             //STORE LSW
Bs R7 l 1                           //R1 = 1 0 0 0 0 0 0 0
beq r7 r7                           //Jump to END of loop

//Case 14: D8 is wrong (INSTRUCTIOn 400)
Or R0 R5                            //0 1 0 0 0 b11 b10 b9
Ma R2 s                             //Store MSW
Addi R1 -1                          //R1 = LSW (30 initially)
Addi R2 -1                          //R2 = LSW (1 initially)
Ma R1 l                             //Load LSW
Xor R7 R7                           //Clear R7
Addi R7 1                           //R7 = 0 0 0 0 0 0 0 1
Bs R7 l 3                           //R7 = 0 0 0 0 1 0 0 0
Bs R7 l 3                           //R7 = 0 1 0 0 0 0 0 0
Bs R7 l 1                           //R1 = 1 0 0 0 0 0 0 0
Xor R0 R7                           //R0 = !b8 b7 b6 b5 b4 b3 b2 b1
Ma R2 s                             //STORE LSW
beq r7 r7                           //Jump to END of loop

//Case 15: D9 is wrong (instruction 416)
Or R0 R5                            //0 1 0 0 0 b11 b10 b9
Xor R7 R7                           //Clear R7
Addi R7 1                           //R7 = 0 0 0 0 0 0 0 1
Xor R0 R7                           //R0 = 0 1 0 0 0 b11 b10 !b9
Ma R2 s                             //Store MSW
Addi R1 -1                          //R1 = LSW (30 initially)
Addi R2 -1                          //R2 = LSW (1 initially)
Ma R1 l                             //Load LSW
```

```
Ma R2 s                                 //STORE LSW without changing
Bs R7 l 3                               //R7 = 0 0 0 0 1 0 0 0
Bs R7 l 3                               //R7 = 0 1 0 0 0 0 0 0
Bs R7 l 1                               //R1 = 1 0 0 0 0 0 0 0
beq r7 r7                               //Jump to END of loop



//Case 16: D10 is wrong (instruction 432)
Or R0 R5                                //0 1 0 0 0 b11 b10 b9
Xor R7 R7                               //Clear R7
Addi R7 1                               //R7 = 0 0 0 0 0 0 0 1
Bs R7 l 1                               //R7 = 0 0 0 0 0 0 1 0
Xor R0 R7                               //R0 = 0 1 0 0 0 b11 !b10 b9
Ma R2 s                                 //Store MSW
Addi R1 -1                              //R1 = LSW (30 initially)
Addi R2 -1                              //R2 = LSW (1 initially)
Ma R1 l                                 //Load LSW
Ma R2 s                                 //STORE LSW without changing
Bs R7 l 3                               //R7 = 0 0 0 0 1 0 0 0
Bs R7 l 3                               //R7 = 0 1 0 0 0 0 0 0
beq r7 r7                               //Jump to END of loop

//Case 17: D11 is wrong (instruction 448)
Or R0 R5                                //0 1 0 0 0 b11 b10 b9
Xor R7 R7                               //Clear R7
Addi R7 1                               //R7 = 0 0 0 0 0 0 0 1
Bs R7 l 2                               //R7 = 0 0 0 0 0 1 0 0
Xor R0 R7                               //R0 = 0 1 0 0 0 !@b11 b10 b9
Ma R2 s                                 //Store MSW
Addi R1 -1                              //R1 = LSW (30 initially)
Addi R2 -1                              //R2 = LSW (1 initially)
Ma R1 l                                 //Load LSW
Ma R2 s                                 //STORE LSW without changing
Bs R7 l 2                               //R7 = 0 0 0 0 1 0 0 0
```

```
Bs R7 l 3                           //R7 = 1 0 0 0 0 0 0 0
beq r7 r7                           //Jump to END of loop

//Case 18: Double bit error (instruction 464)
Bs R5 l 1                           //R5 = 1 0 0 0 0 0 0 0
Or R0 R5                            //R0 = 1 0 0 0 0 x x x
ma   R2, s                          //Store MSW
addi R1, -1                         //Get address of LSW
Addi R2, -1                         //Get address of LSW
ma   R1, l                          //Load LSW
ma   R2, s                          //Store LSW
Xor R7 R7                           //Clear R7
Addi R7 1                           //R7 = 0 0 0 0 0 0 0 1
Bs R7 l 3                           //R7 = 0 0 0 0 1 0 0 0
Bs R7 l 3                           //R7 = 0 1 0 0 0 0 0 0
Bs R7 l 1                           //R7 = 1 0 0 0 0 0 0 0
beq r7 r7                           //Jump to END of loop


-------------------------------------------------------------------------------------
---
//END OF LOOP (INSTRUCTION 512)

//should be in lsw so R1 = 30 and R2 = 0  initially
//need to add 2 to get to next LSW in the loop.
addi R1 2                           //LSW read address of next message
addi R2 2                           //LSW write address of next message

xor  R6, R6                         //Clear R6
addi R6, 3
bs   R6, l, 2
addi R6, 3
bs   R6, l, 2                       //R6 = 0 0 1 1 1 1 0 0 = 60

//If R1 = 60 jump to end
addi R7 3
```

```
addi R7 2                          //R7 = 1 0 0 0 0 1 0 1
beq R1 R6                          //If r1 = 60 jump to 532
xor R7 R7                          //R7 = 0
beq R7, R7                         //else jump to 0

//END (INSTRUCTION 532)
done
```

Program 3 Pseudocode

```
//Read address preloaded with first in data mem
ReadAddress = 0
patternOccurences = 0
bytesWithPattern =0


//SOLVE PART A and PART B
For (readAddress < 32):
     message = datamem[readAddress]
     bitPattern = datamem[32]
     didWeFindPattern = 0

     //Every time you find the pattern, patternOccurences++ for part A)
     if(message[7:3] == bitPattern[7:3])
          patternOccurences++
          didWeFindPattern = 1
     if(message[6:2] == bitPattern[7:3])
          patternOccurences++
          didWeFindPattern = 1
     if(message[5:1] == bitPattern[7:3])
          patternOccurences++
          didWeFindPattern = 1
     if(message[4:0] == bitPattern[7:3])
          patternOccurences++
          didWeFindPattern = 1

     //If we found the pattern at any point then increment bytes with pattern for part b)
     if(didWeFindPattern = True)
          bytesWithPattern++

     readAddresses++
```

```
//SOLVE PART C
For (readAddress < 32):
     message1 = datamem[readAddress]
     message2 = datamem[readAddress+1]
     bitPattern = datamem[32]
     didWeFindPattern = 0

     //Every time you find the pattern, pattern
     if(message1[7:3] == bitPattern[7:3])
          patternOccurences++
          didWeFindPattern = 1
     if(message1[6:2] == bitPattern[7:3])
          patternOccurences++
          didWeFindPattern = 1
     if(message1[5:1] == bitPattern[7:3])
          patternOccurences++
          didWeFindPattern = 1
     if(message1[4:0] == bitPattern[7:3])
          patternOccurences++
          didWeFindPattern = 1

     message1 << 4         //bottom half of first byte
     message2 >> 4         //upper half of second byte
     message2 = message1 | message2             //combine two

     //Every time you find the pattern, pattern
     if(message2[7:3] == bitPattern[7:3])
          patternOccurences++
     if(message2[6:2] == bitPattern[7:3])
          patternOccurences++
     if(message2[5:1] == bitPattern[7:3])
          patternOccurences++
     if(message2[4:0] == bitPattern[7:3])
          patternOccurences++
```

Program 3 Assembly Code

```
//Preloaded into registers
//R1 = 0 (read address of message)
//R2 = 32 (bit pattern is stored in datamem[32]
//R3 = 0 (Constant for comparison)

//INSTRUCTION 0: (SETUP)
ma    R2, l                //Read in bit pattern from datamem[32]
xor   R4, R0               //Copy bit pattern into R4 = p5 p4 p3 p2 p1 0 0 0
xor   r2 r2                //Clear R2
addi r7 1                  //R7 = 1

//TOP OF LOOP
//Case 1: Match b8:b4 (INSTRUCTION 4):
ma    R1, l                //R0 = message = datamem[readAddress]
Bs    R0, r, 3             //R0 = 0 0 0 b8 b7 b6 b5 b4 (clear bottom 3 bits)
bs    R0, l, 3             //R0 = b8 b7 b6 b5 b4 0 0 0 (shift into correct spot)
xor   R0, R4               //compare message with pattern (if perfect match R0 = all 0)
addi R7, 2                 //R7 = 3 (Jump to 12)
beq   R0, R3               //If pattern match jump to increment, otherwise, jump to case 2
addi  R7, 1                //R7 = 4, (Jump to 16)
beq   R7, R7               //Jump to case 2

//If we find pattern in case 1 (INSTRUCTION 12):
addi  R5, 1                //increment patternOccurrences (Part a)
Xor   R2, R2               //Clear R2
Addi  R2, 1                //Set it equal to 1, indicate that we have found occurence
addi R7, 1                 //R7 = 4
```

```
//Case 2 match b7:b3 (INSTRUCTION 16):
ma    R1, l                 //R0 = message = datamem[readAddress]
bs    R0, r, 2              //R0 = 0 0 b8 b7 b6 b5 b4 b3
bs    R0, l, 3              //R0 = b7 b6 b5 b4 b3 0 0 0
xor   R0, R4                //compare message with pattern
addi R7, 3                  //R7 = 7 (Jump to 28)
beq   R0, R3                //If pattern match jump to increment, otherwise, jump to case 2
addi  R7, 1                 //R7 = 8 (Jump to 32)
beq   R7, R7                //Jump to case 3

//If we found a pattern in CASE 2 (INSTRUCTION 28)
addi  R5, 1                 //increment patternOccurrences (Part a)
Xor   R2, R2                //Clear R2
Addi  R2, 1                 //Set it equal to 1, indicate that we have found occurence
addi R7, 1                  //R7 = 8

//Case 3  match b6:b2 (INSTRUCTION 32):
ma    R1, l                 //R0 = message = datamem[readAddress]
bs    R0, r, 1              //R0 = 0 b8 b7 b6 b5 b4 b3 b2
bs    R0, l, 3              //R0 = b6 b5 b4 b3 b2 0 0 0
xor   R0, R4                //compare message with pattern
addi R7, 3                  //R7 = 11 (Jump to 44)
beq   R0, R3                //If pattern match jump to increment, otherwise, jump to case 2
addi  R7, 1                 //R7 = 12 (Jump to 48)
beq   R7, R7                //Jump to case 4


//If we found a pattern in CASE 3 (INSTRUCTION 44)
addi  R5, 1                 //increment patternOccurrences (Part a)
Xor   R2, R2                //Clear R2
Addi  R2, 1                 //Set it equal to 1, indicate that we have found occurence
addi R7, 1                  //R7 = 12
```

```
//Case 4  match b5:b1 (INSTRUCTION 48):
ma    R1, l                 //R0 = message = datamem[readAddress]
bs    R0, l, 3              //R0 = b5 b4 b3 b2 b1 0 0 0
xor   R0, R4                //compare message with pattern
addi  R7, 3                 //R7 = 15 (Jump  to 60)
beq   R0, R3                //If pattern match jump to increment, otherwise, jump to case 4
addi  R7, 1                 //R7 = 16 (Jump to 64)
beq   R7, R7                //Jump to End of loop

//If we found a pattern in CASE 4 (INSTRUCTION 60)
addi  R5, 1                 //increment patternOccurrences (Part a)
Xor   R2, R2                //Clear R2
Addi  R2, 1                 //Set it equal to 1, indicate that we have found occurence
addi R7 1                   //R7 = 16


//FInal case: If a pattern was found in the byte(INSTRUCTION 64) PART B:
addi  R7, 1                 //R7 = 17 (Jump to 68)
beq   r2, r3                //Check if R2 = 0, if it does go to end of loop (68)
addi  R6, 1                 //If we didn't jump, increment bytesWithPattern in R6

// End of loop (INSTRUCTION 68)
xor r2 r2                   //clear r2
addi r1 1                   //move on to message byte
xor r0 r0                   //clear r0
addi r0 1
bs r0 l 3
bs r0 l 2                   //r0 = 32
addi r7 3                   //r7 = 20 (jump to 80)
beq r1 r0                   //if current byte = 32, then jump to END OF PART A AND B
xor r7 r7                   //Clear R7
addi r7 1                   //r7 = 1
beq r7 r7                   //Jump to top of loop (instruction 4)
```

```
// END OF PART A and B (INSTRUCTION 80)
xor  R2, R2               //Clear R2
xor r2 r0                 //r2 = 32
xor  R0, R0               //Clear R0
or   R0, R5               //R0 = R5 (patternOccurences) need to write to 33
addi R2, 1                //R2 = 33
ma   R2, s                //datamem[33] = patternOccurences
xor  R0, R0               //clear R0
or   R0, R6               //R0 = R6 (bytesWithPattern) need to write to 34
addi R2, 1                //R2 = 34
ma   R2, s                //datamem[34] = bytesWithPattern

//Prep for PART C
xor r1 r1                 //reset r1 to 0
addi r7 3                 //R7 = 23
addi r7 1                 //R7 = 24 (JUMP TO 96)

//TOP OF LOOP (INSTRUCTION 96)
ma R1 l                   //Load FIRST BYTE
xor r2  r2                //Clear R2
xor r2 r0                 //R2 = first byte
bs r2 l 3
bs r2 l 1                 //R2 = b4 b3 b2 b1 0 0 0 0
addi r1 1                 //R1 = second byte address
ma R1 l                   //Load SECOND BYTE
bs r0 r 3
bs r0 r 1                 // r0 = 0 0 0 0 b8 b7 b6 b5
or r2 r0                  // r0 = b4 b3 b2 b1 b8 b7 b6 b5
addi r7 3                 //R7 = 27 (Jump to 108)
```

```
//Case 1: b4 b3 b2 b1 b8 matches (108)
xor r0 r0                  //Clear R0
xor r0 r2                  //Get a copy of the word we constructed into R0
bs r0 r 3                  //r0 = 0 0 0 b4 b3 b2 b1 b8
bs r0 l 3                  //r0 = b4 b3 b2 b1 b8 0 0 0
xor r0 r4                  //compare message
addi r7 3                  //R7 = 30 (Jump to 120)
beq r0 r3                  //if pattern matches jump to increment (120)
addi r7 1                  //R7 = 31 (Jump to 124)
beq r7 r7                  //Jump to Case 2

//Increment for  Case 1 (120)
addi r5 1                  //Increment patternOccurences
addi r7 1                  //R7 = 31 (Jump to 124)

//Case 2: b3 b2 b1 b8 b7 matches (INSTRUCTION 124)
xor r0 r0                  //Clear R0
xor r0 r2                  //Copy over constructed bitstring in R2 into R0
bs r0 r 2                  //r0 = 0 0 b4 b3 b2 b1 b8 b7
bs r0 l 3                  //r0 = b3 b2 b1 b8 b7 0 0 0
xor r0 r4
addi r7 3                  //r7 = 34 (Jump to 136)
beq r0 r3                  //if pattern matches jump to increment (136)
addi r7 1                  //r7 = 35 (Jump to 140)
beq r7 r7                  // Jump to case 3

//Increment for case 2 (136)
addi r5 1                  //Increment pattern occurences
addi r7 1                  //R7 = 35
```

```
//Case 3: b2 b1 b8 b7 b6 matches (INSTRUCTION 140)
xor r0 r0                  //Clear R0
xor r0 r2                  //Copy over constructed bitstring in R2 into R0
bs r0 r 1                  //r0 = 0 b4 b3 b2 b1 b8 b7 b6
bs r0 l 3                  //r0 = b3 b2 b1 b8 b7 0 0 0
xor r0 r4
addi r7 3                  //r7 = 38 (Jump to 152)
beq r0 r3                  //if pattern matches jump to increment (152)
addi r7 1                  //r7 = 39
beq r7 r7                  // Jump to case 4

//Increment for case 3 (152)
addi r5 1
addi r7 1

//Case 4: b1 b8 b7 b6 b5 matches (INSTRUCTION 156)
xor r0 r0                  //Clear R0
xor r0 r2                  //Copy over constructed bitstring in R2 into R0
bs r0 l 3                  //r0 =  b1 b8 b7 b6 b5 0 0 0
xor r0 r4
addi r7 3                  //r7 = 42 (Jump to 168)
beq r0 r3                  //if pattern matches jump to increment (168)
addi r7 1                  //r7 = 43 (Jump to 172)
beq r7 r7                  //Jump to END OF LOOP 2

//Increment for case 4 (168)
addi r5 1
addi r7 1                  //r7 = 43

//END OF LOOP (INSTRUCTION 172)
addi r7 3                  //R7 = 46 (Jump to end of part c inst 184)
xor r2 r2                  // clear r2
addi r2 3                  //R2 = 0 0 0 0 0 0 1 1
bs r2 l 2                  //R2 = 0 0 0 0 1 1 0 0
addi r2 3                  //R2 = 0 0 0 0 1 1 1 1
bs r2 l 1                  //R2 = 0 0 0 1 1 1 1 0
```

```
addi r2 1                   //R2 = 31
beq r1 r2                   //If r1 = 31, jump to END OF PART C
bs r7 r 1                   //R7 = 23
addi r7 1                   //r7 = 24 (jump to top of loop inst 96)
beq r7 r7                   //Jump to top of loop 2

//END OF PART C (INSTRUCTION 184)
addi r2 3                   //R2 = 34
addi r2 1                   //R2 = 35
xor r0 r0                   //clear r0
xor r0 r5                   //r0 = total occurrences
ma r2 s                     //mem[35] = total occurences
done
```

8. Changelog

- Milestone 4
  - Individual Component Specification:
    - Decided to add start input to ProgCtr in order to not start reading from data memory before it was completely loaded. Program counter now requires to start high in order to progress counter, otherwise nothing happens.
    - Rewrote the ProgCtr_tb as well to accommodate for the changes described above.
    - Updated the Instruction Memory (InstMem) and Register File (RegFile) modules with an initial begin block so that the machine code and register file presets can be loaded for each respective program.
    - Removed initial begin block from DataMem because we are not preloading any constants into DataMem.
    - Changed name of data memory to 'Core' to match the autograder testbench
    - Control Decoding where we assign control signals happens primarily within the TopLevel now.
    - New functionality that sets all control signals to 0 in order to prevent changes in permanent state before the testbench allows us to proceed (lower start flag).
    - Re-synthesized TopLevel module with new changes.
  - Machine Specification
    - Changed the LUT patterns on the LAND instruction. This module is used to extract a certain pattern of bits from a register. The pattern for LUT input (100) was changed from 0010_0111 to 0001_0111.
  - Program Implementation
    - Made significant changes to program 1's assembly code to accommodate for the shift-left 2 changes as well as to fix correctness
    - Made significant changes to program 2's assembly code for the same reasons.
    - Made significant changes to program 3's assembly code for the same reasons.
    - Wrote testbenches for each program to verify correctness
  - Assembler
    - Added support for special instruction 'done' that raises the done flag within our processor. This instruction uses an unused LUT encoding for our land instruction so there are no conflicts with other instructions.

- Milestone 3
  - Architectural Overview
    - Added in a shift left 2 in order to increase the range of instructions from 2^8 to 2^10 within our instruction memory
    - This was done in the top level with logic assigning the TargetBranch to register 7 (which holds the target address from branching) and performing an arithmetic shift by 2 to the left.

    - Assembler
        - We wrote an assembler in python that takes in the file assembly.txt and outputs a machinecode.txt that contains the binary machine code corresponding to the instruction in the assembly file.
- Milestone 2
    - Architectural Overview
        - Changed input logic for datamem, there is now an additional AND gate which takes a control signal from the control decoder and instruction[2]. Both of these must be high in order for the store word operation to write to memory.
        - Removed ALU OP module since it was extraneous and simply using the bottom 2 bits of the instruction microcode as an input to the ALU is sufficient.
        - Added some extra control signals to control the register file and datamem
        - Added an additional mux that allowed you to change the write address logic to the register file. Loads and stores will always write to R0. All other operations that write to the register file will “destructively” write to the register that is used as the first operand to the instruction.
        - Added additional write_data port on datamem and a corresponding output from reg0 that always maps to it.
    - Machine Specification
        - **INSTRUCTOR FEEDBACK:** Followed instructor feedback and mentioned that we support 8 registers under the internal operands section.
        - Instruction Format Change - added an additional LUT pattern to our LAND instruction. This instruction takes a 3-bit input and maps it to an 8-bit mask that can be used to AND with the contents of another register to extract certain bits. We added a pattern to extract LSB.
        - R3 is now a general purpose register, we are no longer implementing logic that would write back the shift out flag to the register file. Instead, we are using the new LUT pattern to extract the LSB when we need it.
    - Programmer’s Model
        - Added the response to question 4.3 on the writeup.
    - Program Implementation
        - Changed Program Implementation number from 5 to 6 on writeup.
    - Individual Component Specification
        - Added a new 5th chapter in the writeup called individual component specification.
        - Added specifications for the following modules: Top level, program counter, instruction memory, control decoder, register file, ALU, data memory, lookup tables, mux, done flag comparator, sign extend.
- Milestone 1
    - Initial Version