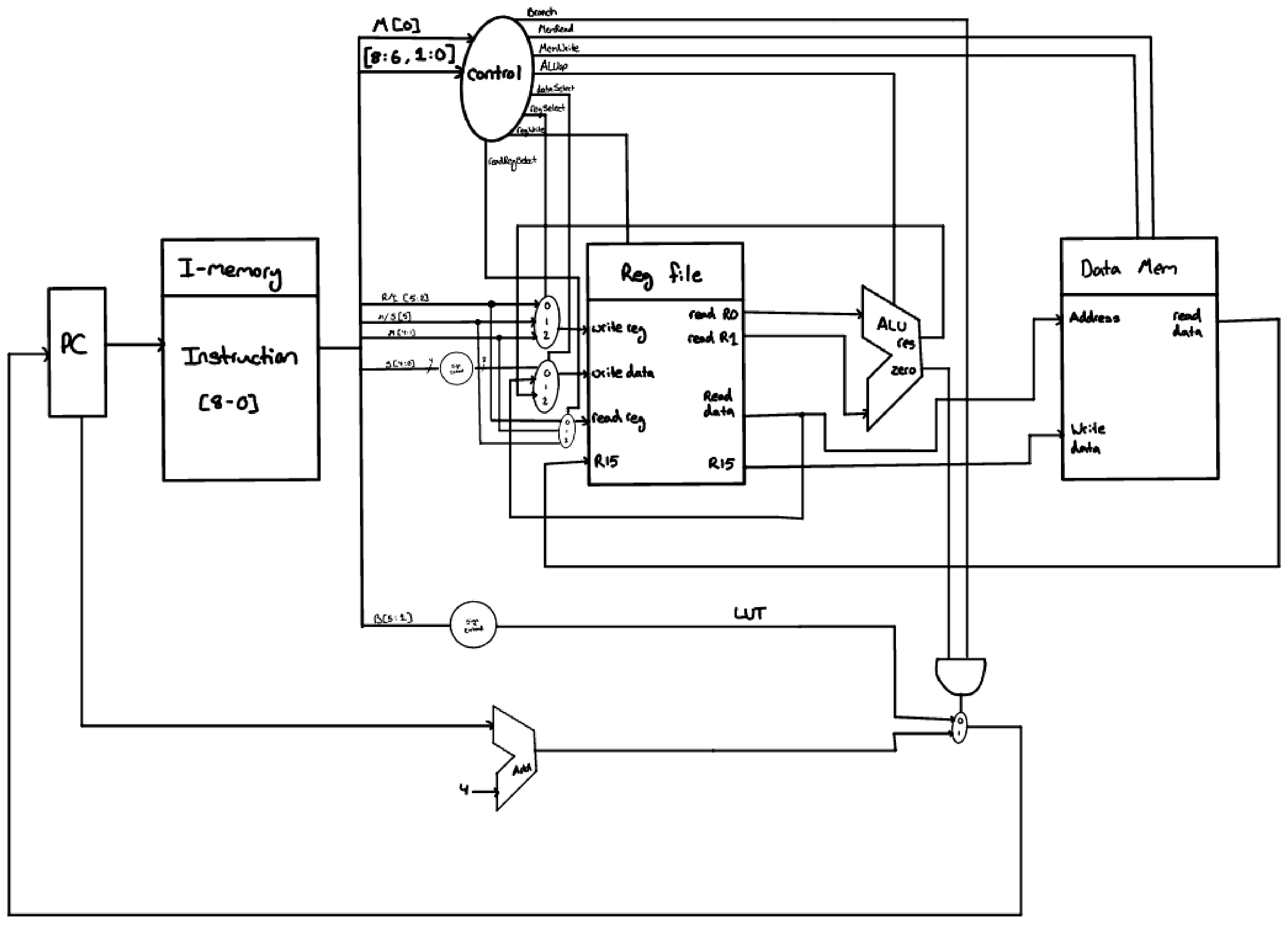
CSE 141L Milestone 2

Benjamin Scott, A16380204; Osama Al-Shuaili, A16341658; Samuel Liu, A15793529

Updated Architecture Diagram



Architecture Diagram Change Log

- MOV1, MOV2 instructions added
  - Changed the input wires to write reg and write data
- Removed an ALU from branching since we are only using Direct Branching and not Relative Branching
- Connected MemRead and MemWrite directly to the DataMem block
- Added a regReadSelect Mux to facilitate reading from r0 ↔ rs

Required Question

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 use our ALU to compute branch instructions as well as arithmetic instructions
- The branch instructions are BEQ, BNE, and BLT. Our ALU will compare R0 and R1 and set the Zero output flag appropriately
- This does not complicate our design too much. We just need to add an extra wire output to the ALU and feed that wire into the PC where we will determine if we need to branch.

Architecture Change Log

- MOV Type Change

    - Add a func bit to our M type to specify which direction the MOV instruction is going in. Now we will have MOV1 and MOV2. So for example, MOV1 R0 R15 will move R15 into R0 and MOV2 R0 R15 will move R0 into R15. They will be compiled down to the same machine code except the last func bit will be different for each and our hardware will have to handle these two cases
- We got rid of the Data Memory LUT because each LUT is limited to 32 entries and we need at least 64 entries for each memory location we are going to access. Instead, we will just have a register value feed into the data memory as the address. The register value can hold up to 8 bits of information which should be plenty to specify a data memory location

Individual Component Specification

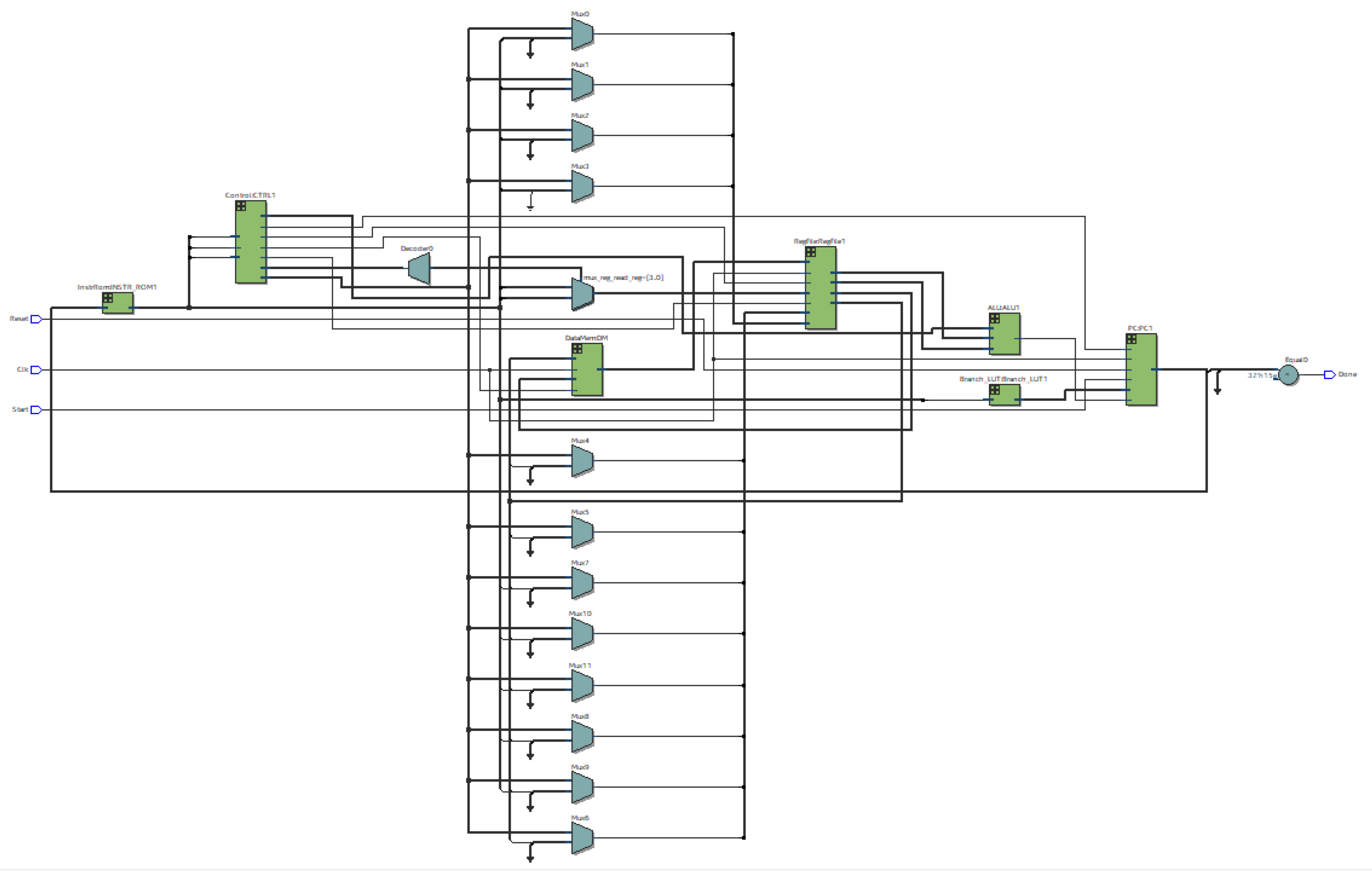
Top Level

Module file name: top_level.sv

Functionality Description

This is the top level of our entire microprocessor. This module is responsible for connecting all the modules we wrote with wires and taking as input the Reset, Start, Clk flags that we need for when the final testbenches get run on it. It is also responsible for setting a Done flag once its computation is done.

Schematic



Program Counter

Module file name: PC.sv

Module testbench file name: PC_tb.sv

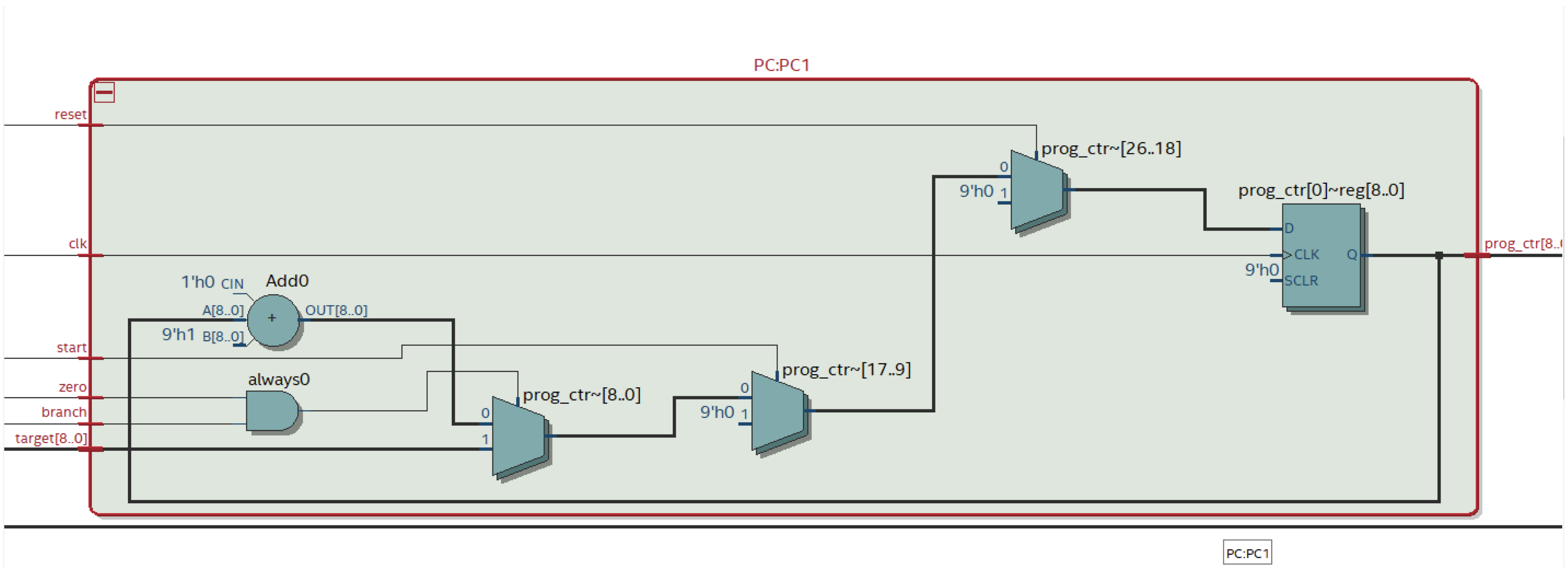
Functionality Description

This is the program counter it is used to keep track of the PC value and by default it would increment the PC value by 1. In a branching scenario it would change the PC value to the target.

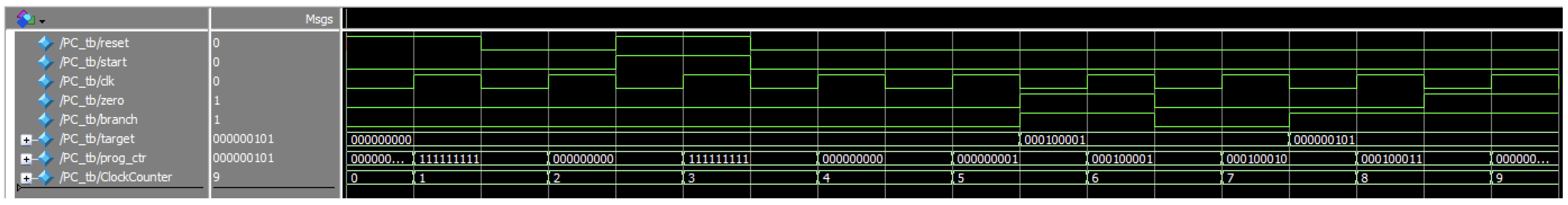
Testbench Description

Our testbench tests that our program will only start when start flag gets turned on then off. We also test our absolute branching when both the zero and branch flags are on. We test that we don't branch when only one of those flags are on. We also test that when we don't absolute branch, we are just incrementing PC by 1. We test our code via assert statements that compare the outputted program counter with an expected value.

Schematic

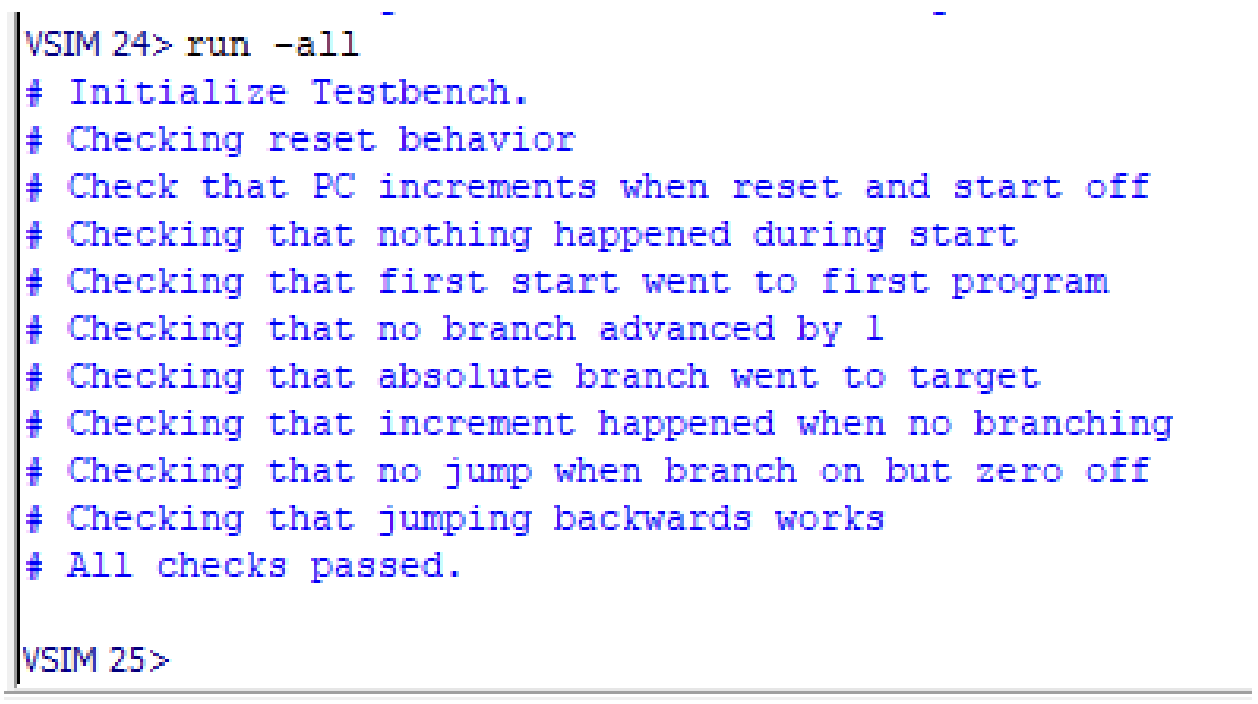


Timing Diagram



Timing Diagram Testing

Below is all the cases we are checking with asserts. If an assert fails, there will be an error message, but as you can see, there are none.



```
VSIM 24> run -all
# Initialize Testbench.
# Checking reset behavior
# Check that PC increments when reset and start off
# Checking that nothing happened during start
# Checking that first start went to first program
# Checking that no branch advanced by 1
# Checking that absolute branch went to target
# Checking that increment happened when no branching
# Checking that no jump when branch on but zero off
# Checking that jumping backwards works
# All checks passed.

VSIM 25>
```

Instruction Memory

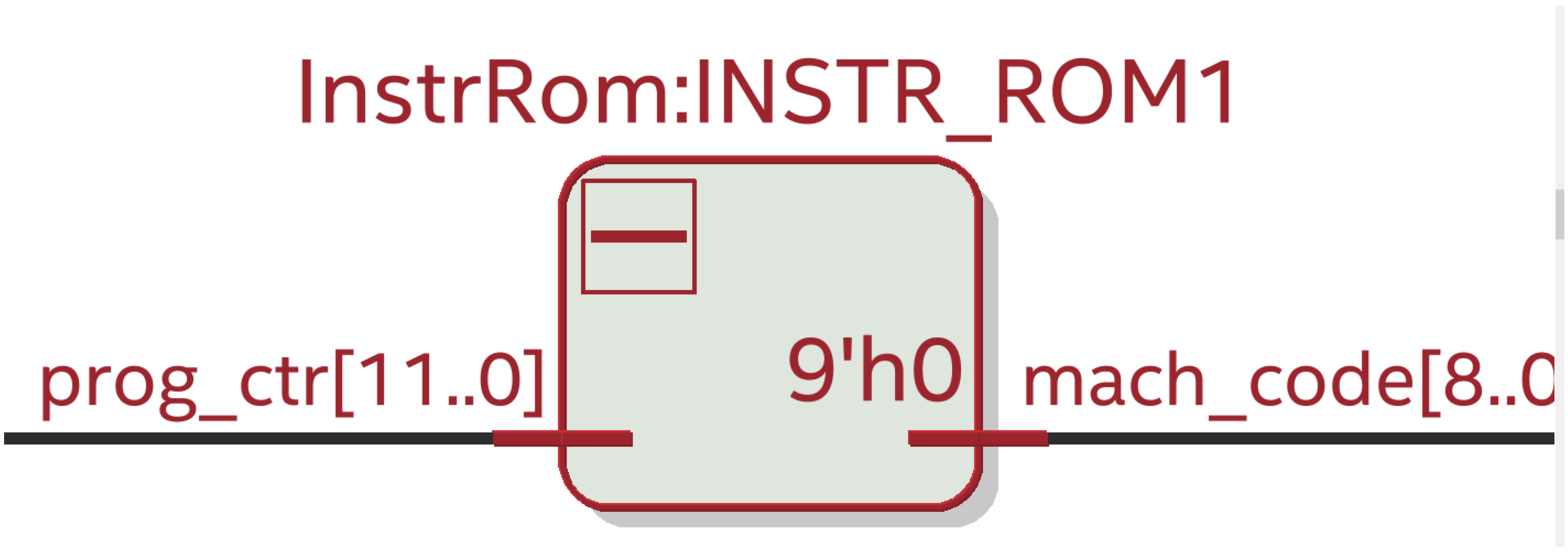
Module file name: InstrRom.sv

Functionality Description

This is the Instruction Memory file that takes in the PC value and outputs the machine code of that code. It is basically used to fetch new instructions to be decoded and executed.

Schematic

See next page



Control Decoder

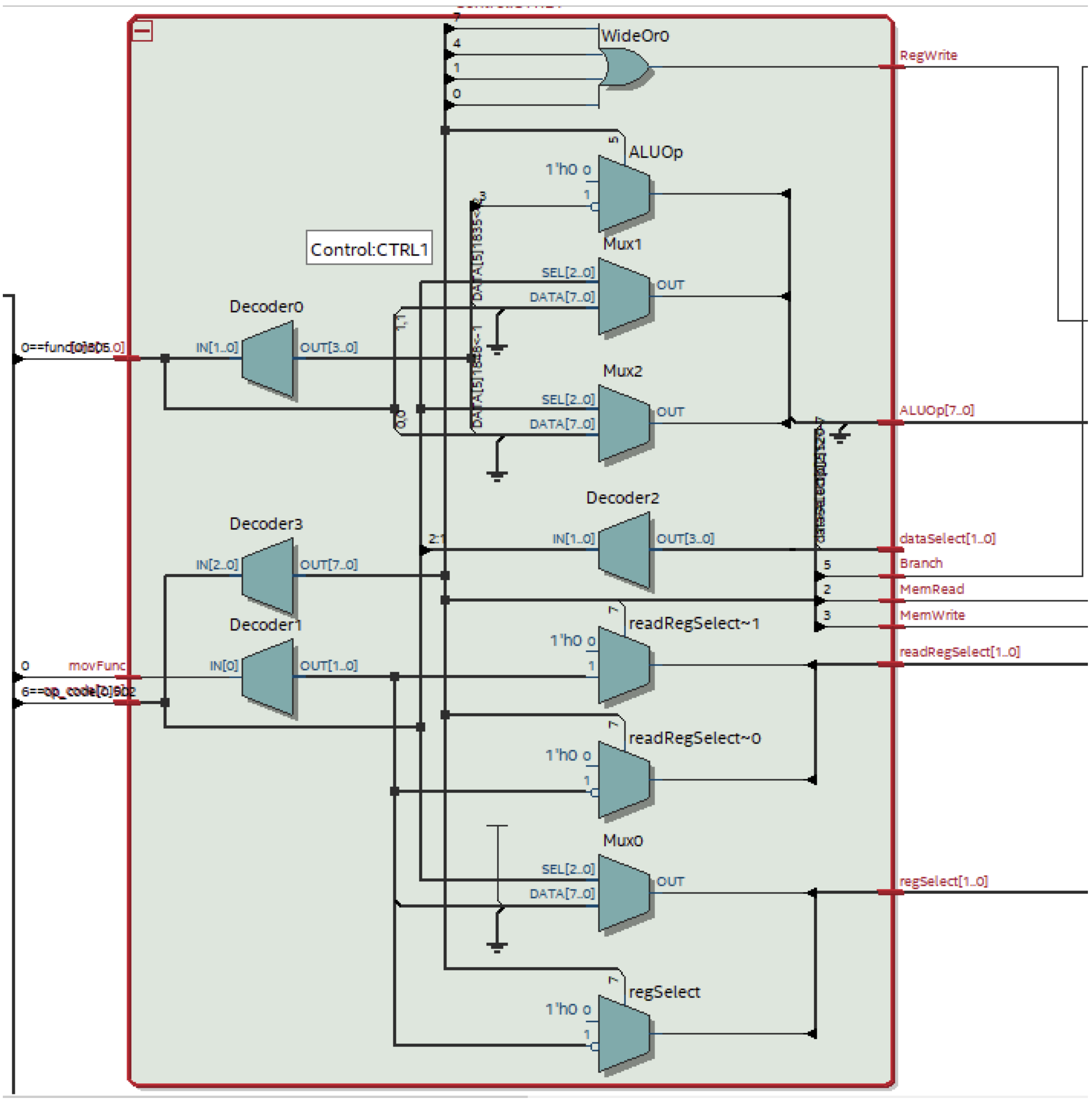
Module file name: Control.sv

Functionality Description

The Control File is the control block of the diagram. It takes in the opcode and decodes it to the specific instruction that our architecture should execute. It sends signals to different parts of the architecture to ensure that the correct instruction is executed.

Schematic

See next page



Register File

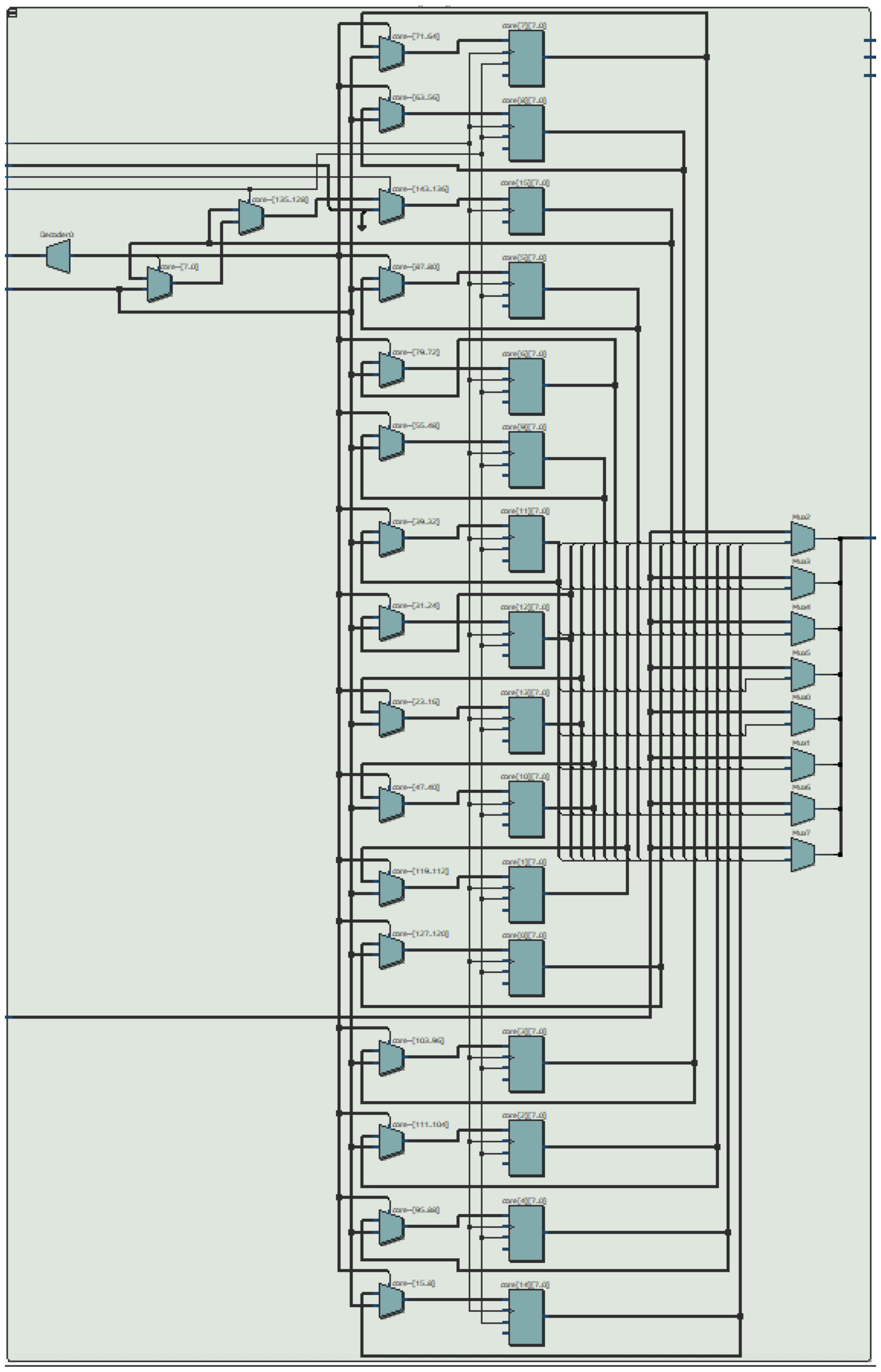
Module file name: RegFile.sv

Functionality Description

This is our Register File, it stores all our registers and controls their data flow. It takes read and write flags and assigns the data to the specific registers.

Schematic

See next page



Definitions

Module file name: Definitions.sv

Functionality Description

This is our Definitions file. It basically converts our ALU operations to their OpCode, this makes our architecture more readable.

Schematic

N/A

ALU (Arithmetic Logic Unit)

Module file name: ALU.sv

Module testbench file name: ALU_tb.sv

Functionality Description

The ALU handles most of the arithmetic computation for our processor as well as calculates if we should branch for a branch instruction. It takes in an inputA and an inputB and computes the output based on the given ALU_OP code which specifies which operation to do (e.g. ADD, SUB, …).

Testbench Description

Our testbench creates the ALU module and feeds in two arbitrary specified inputs as inputA and inputB (4 and 3) respectively. We then change the ALU_OP input (which specifies which ALU operation to do: ADD, SUB, etc) and see if we get the expected output. We have an expected wire that will always hold the correct output depending on the ALU_OP wire. We will change the ALU_OP and test all the operations on inputA and inputB. We verified the correctness by running this on ModelSim and looking at the waveform to make sure that the output of the ALU and the expected wire match.

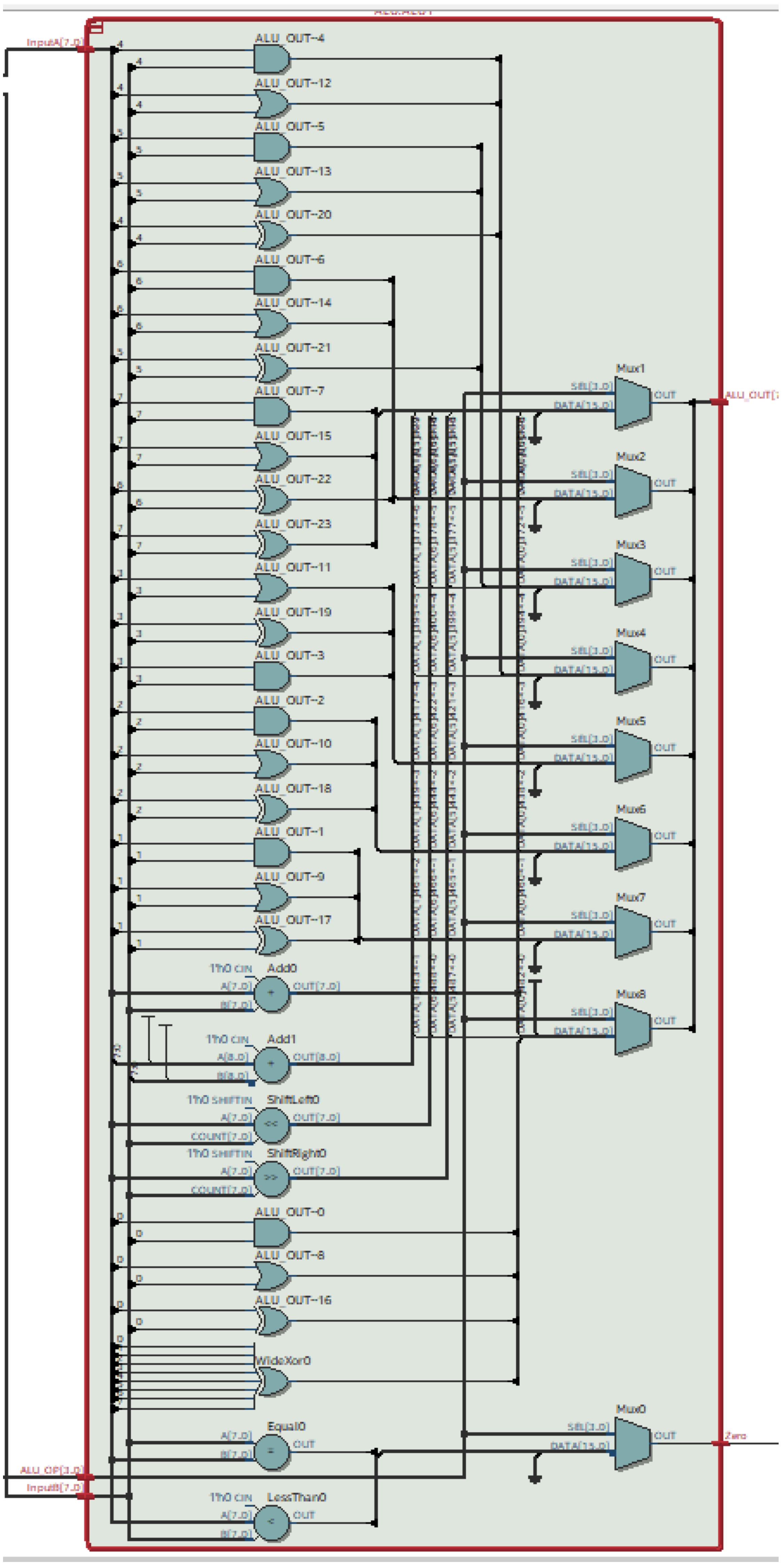
ALU Operations

We will be demonstrating all of our ALU operations. Below is the list of all of them:

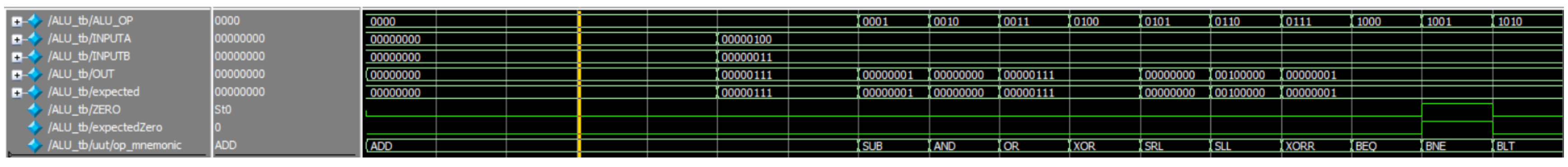
- ADD
- SUB
- AND
- OR
- XOR
- SRL
- SLL
- XORR
- BEQ
- BNE
- BLT

Schematic

See next page



Timing Diagram


```
/ALU_tb/ALU_OP            0000       0000                 0001     0010     0011     0100 0101     0110     0111     1000 1001 1010
/ALU_tb/INPUTA            00000000   00000000  00000100
/ALU_tb/INPUTB            00000000   00000000  00000011
/ALU_tb/OUT               00000000   00000000  00000111   00000001 00000000 00000111      00000000 00100000 00000001
/ALU_tb/expected          00000000   00000000  00000111   00000001 00000000 00000111      00000000 00100000 00000001
/ALU_tb/ZERO              St0
/ALU_tb/expectedZero      0
/ALU_tb/uut/op_mnemonic   ADD        ADD                  SUB      AND      OR       XOR  SRL      SLL      XORR     BEQ  BNE  BLT
```

- Please look at `/ALU_tb/OUT` and `/ALU_tb/expected`
- `/ALU_tb/OUT` represents the output of the ALU
- `/ALU_tb/expected` represents what we expect the output of the ALU to be

- You can see in the waveform that they match so that means our ALU is outputting the correct values!

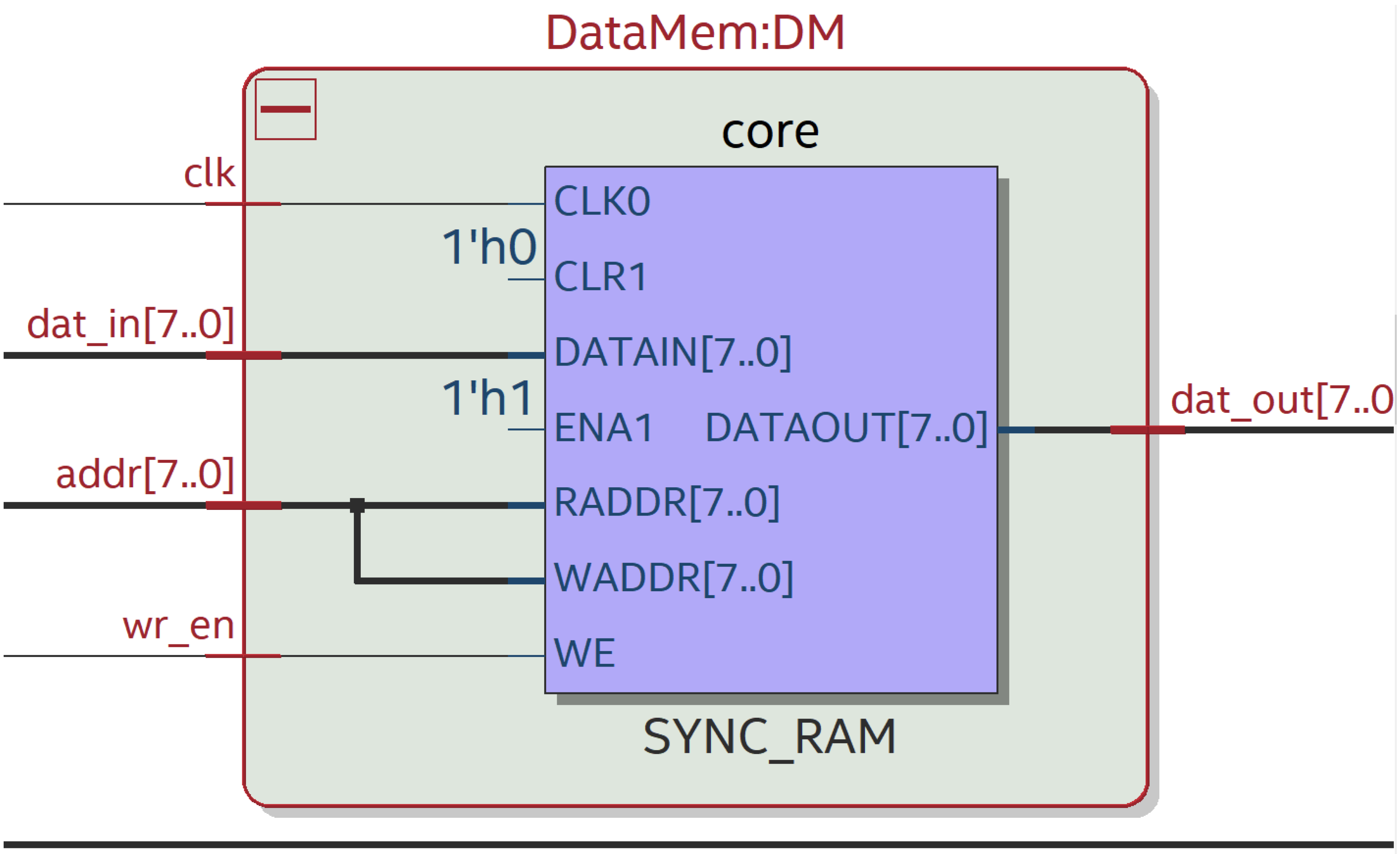
Data Memory

Module file name: DataMem.sv

Functionality Description

Data Memory represents kind of our “main memory”. It holds our inputs for all of the program testbenches and it takes in a 8 bit address and outputs the data at that address. Additionally, it also supports writing if the wr_en flag is turned on.

Schematic



Lookup Tables

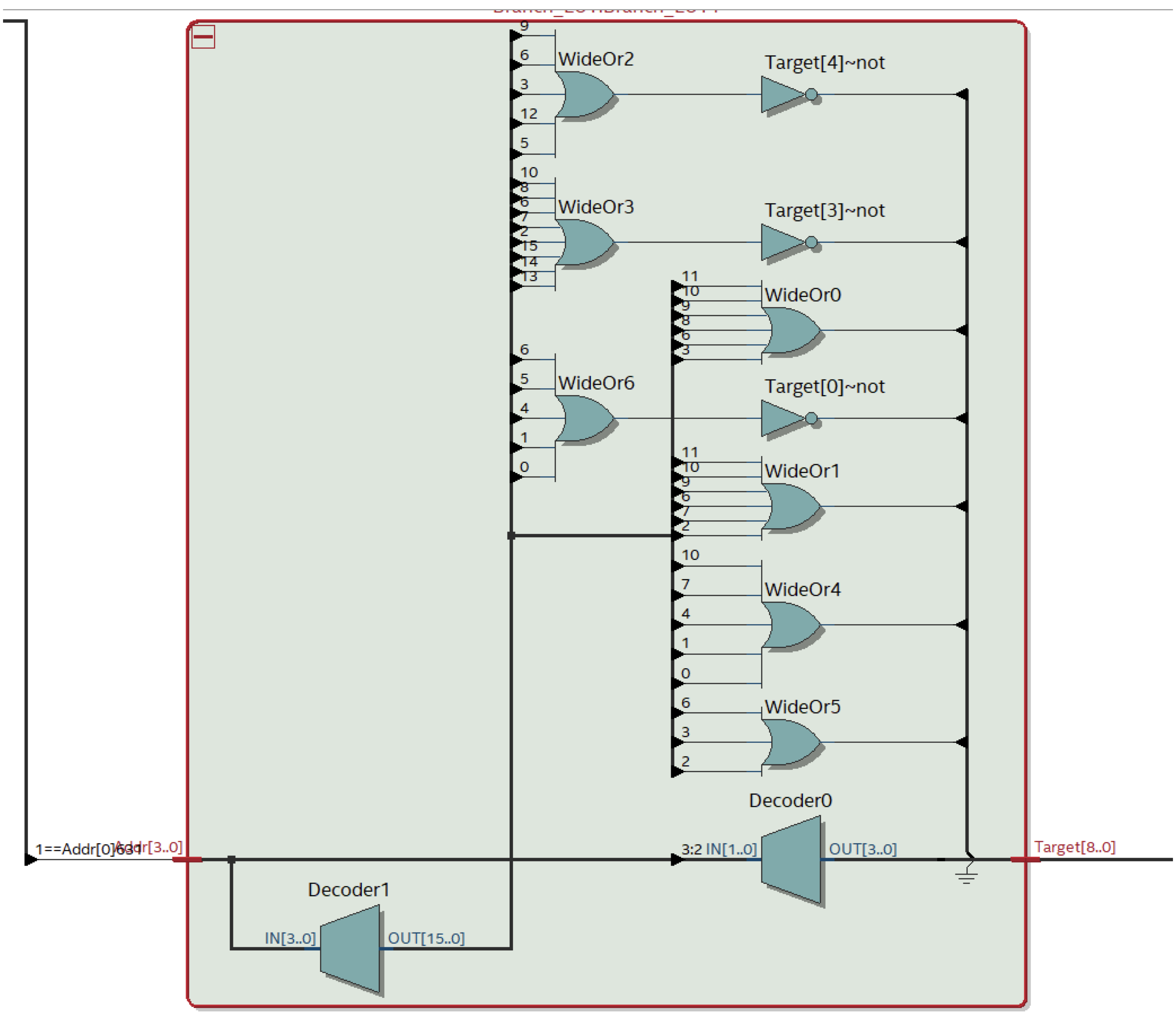
Module file name: Branch_LUT.sv

Functionality Description

The Branch LUT takes in a 4 bit immediate and outputs a 9 bit PC value. It is used for when we want to branch to another place in the code. The PC values outputted by the Branch LUT are all absolute targets.

Schematic

See next page



Other Modules

- N/A

Milestone 1

Important Note

- You may also view the document here for a page-less experience. Link below
    - https://emphasized-sociology-1ab.notion.site/CSE-141L-Milestone-1-b0ca2c6dc18f4dc095da0842e88467f3

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.

Digital Signatures

- Samuel Liu
- Benjamin Scott
- Osama Al-Shuaili

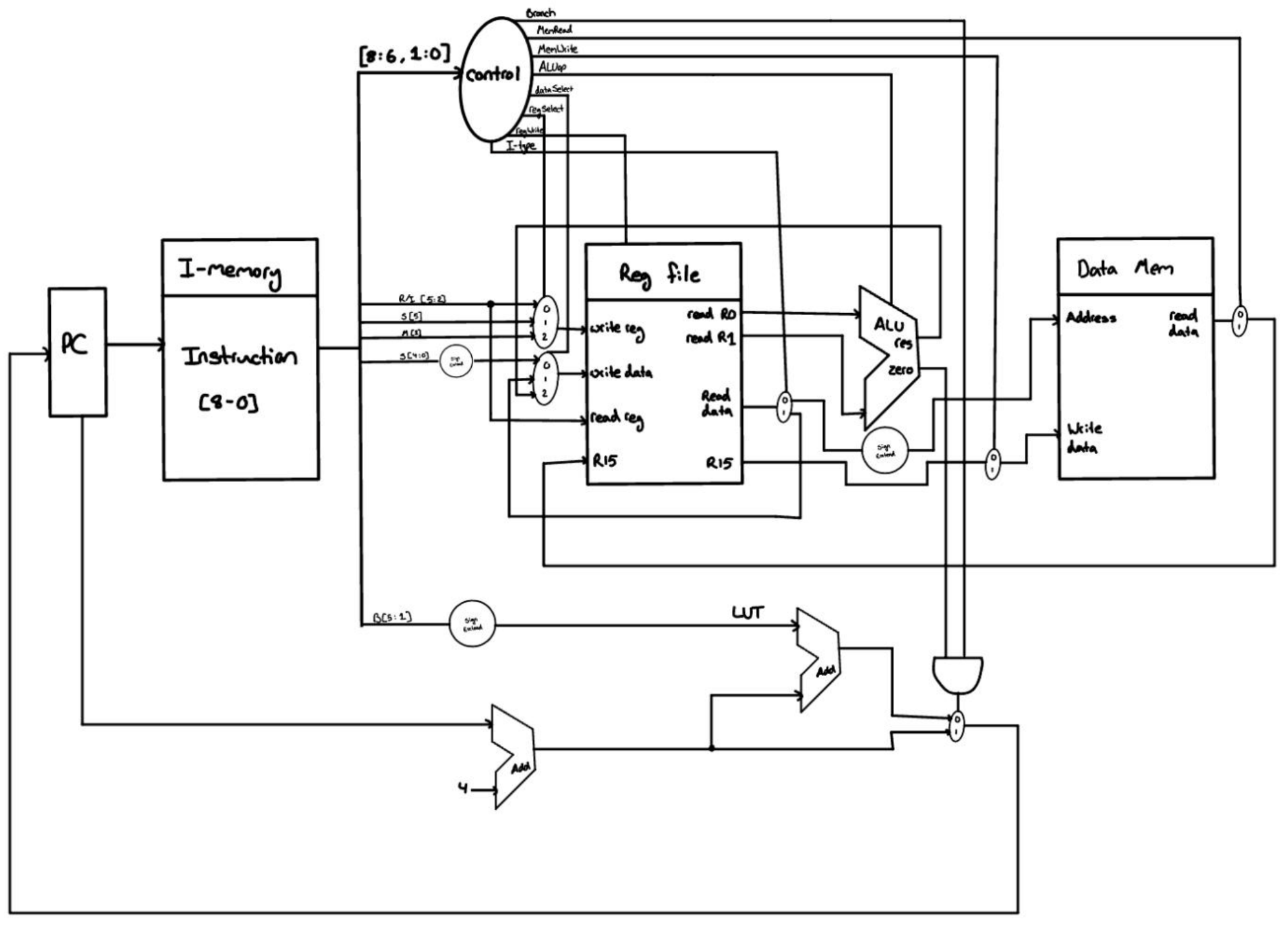
0. Team

- Benjamin Scott
- Osama Al-Shuaili
- Samuel Liu

1. Introduction

- Name of Architecture: HighRISC
- Overall Philosophy
    - Use register 0 and register 1 for all ALU operations
        - Use MOV to move data between registers to prep R0 and R1 before an ALU operation
    - Use register 15 for load and store
- Specific goals
    - Save instructions bits!
    - Readable assembly code
    - Light on memory access
- Architecture Style
    - Reg-reg (Load-Store)

Architectural Overview



3. Machine Specification

Instruction Formats

Type	Format	Corresponding Instructions	Example
R (register)	opcode(3’b) rd(4’b) funct(2’b)	ADD, SUB, AND, OR, XOR, SRL, SLL, XORR	ADD r4
I (immediate)	opcode(3’b) register(4’b) unused(2’b)	LW, SW	LW R14
B (branch)	opcode(3’b) imm(4’b) funct(2’b)	BEQ, BNE, BLT	BNE #8
M (move)	opcode(3’b) rt(1’b) rs(4’b) funct(1’b)	MOV1, MOV2	MOV1 r0 r14
S (set)	opcode(3’b) rd(1’b) immediate(5’b)	SET	SET R0 #20

Operations

Name	Type	Usage	Corresponding Operation	Opcode	Funct
ADD	R	add R[rd]	R[rd] = R[0] + R[1]	0	0
SUB	R	sub R[rd]	R[rd] = R[0] - R[1]	0	1
AND	R	and R[rd]	R[rd] = R[0] & R[1]	0	2
OR	R	or R[rd]	R[rd] = R[0] | R[1]	0	3
XOR	R	xor R[rd]	R[rd] = R[0] ^ R[1]	1	0
SRL	R	srl R[rd]	R[rd] = R[0] >> R[1]	1	1
SLL	R	sll R[rd]	R[rd] = R[0] << R[1]	1	2
XORR	R	xorr rt	R[rt] = ^(R[rt]) xor of all bits inside R[rt])	1	3
LW	I	load rs	R[15] = Mem[lsl_LUT[R[rs]]]	2	N/A
SW	I	store rs	Mem[sl_LUT[R[rs]]] = R[15]	3	N/A
SET	S	set r0/1, imm	R[r0/r1] = imm	4	N/A
BEQ	B	beq LABEL	if (R[0] == R[1]) pc = b_LUT[LABEL]	5	0

Name	Type	Usage	Corresponding Operation	Opcode	Funct
BNE	B	bne LABEL	if (R[0] != R[1]) pc = b_LUT[LABEL]	5	1
BLT	B	blt LABEL	if (R[0] < R[1]) pc = b_LUT[LABEL]	5	2
MOV1	M	mov1 rs r0/1	R[rs] = R[r0/r1]	6	0
MOV2	M	mov2 r0/1 rs	R[r0/r1] = R[rs]	6	1
NOOP	N	NOOP	does nothing	7	N/A

Extra Space:

- Consider NOOP instruction

Internal Operands

- 16 registers, each register holds 8 bits of data
- R0 and R1 are always the operands for R and B type instructions
- R15 is reserved for LOAD and STORE
- The other registers are general purpose

Control Flow (branches)

What types of branches are supported?

- BEQ - Branch if equal
- BNE - Branch if not equal
- BLT - Branch if less than

How are the target addresses calculated?

- Immediate → lookup table

What is the maximum branch distance supported?

- Unbound because we have lookup table

How do you accommodate large jumps?

- Lookup table

Addressing Modes

- Data Memory
    - Indirect via lookup tables
    - An immediate is passed into the load and store instructions and that immediate feeds into a lookup table to get the actual memory addresses
    - 6 bits for LUT, 64 possible data memory locations
- Branch Instructions
    - Indirect via lookup table
    - An immediate is passed into the branch instruction and that immediate feeds into a lookup table to get the actual memory addresses of where to branch to
    - 4 bits for LUT, 16 possible branch locations

4. Programmer’s Model [Lite]

4.1 How does the machine operate?

- All R and B type instructions are done in R0 and R1
- You should use the SET, MOV, and other operations to move data between the registers and put things into R0 and R1 as needed
- Since all the registers excluding R0 and R1 are general purpose and should be used as intermediary space for your calculations, it is best to load only what you need from memory to avoid register clutter; using too many registers will make your code harder to read and maintain.

4.2 Can we copy instructions/operation from MIPS or ARM ISA?

- Though our architecture is very similar to MIPS and ARM, it is different because of the 9-bit constraint. Thus, our instruction set is a bit more nuanced.
- For example, in MIPS and ARM, R type instructions have three register operands. However, in our architecture we do not have room for that so instead, we only specify a destination register and the operands are always supposed to be in R0 and R1 before you call the instruction
- That being said, we share a lot of similar instructions to MIPS and ARM. There will just have to be more work done in preparation to calling those instructions (like putting the operands into R0 and R1 before calling an ADD instruction)

5. Program Implementation

Program 1 Pseudocode

```
Input 1: 0  0  0  0  0  b11 b10 b9
Input 2: b8 b7 b6 b5 b4 b3  b2  b1

Output 1: b11 | b10 | b9 | b8 | b7 | b6 | b5 | p8
Output 2: b4  | b3  | b2 | p4 | b1 | p2 | p1 | p0

Loop 15 times through all 2-byte messages
  R0 <- input 1 (00000 b11 b10 b9)
  R1 <- input 2 (b8 b7 b6 b5 b4 b3 b2 b1)
  calculate p8
  Build output 1 (b11:b5, p8)
  calculate p4
  Build up Output 2 -> (b4, b3, b2, p4, b1, 0, 0, 0)
  calculate p2
  Build up Output 2 -> (b4, b3, b2, p4, b1, p2, 00)
  calculate p1
  Build up Output 2 -> (b4, b3, b2, p4, b1, p2, p1 0)
  calculate p0
  Build up Output 2 -> (b4, b3, b2, p4, b1, p2, p1 p0)
  store output 1 & 2 into data memory
```

Program 1 Assembly Code

```
// TODO: FLIP THE MOV OPERANDS
SET R14 #0              // R14 = 0, used for looping through the 15 2-byte messages

LOOP:

// read from data mem into R1 R0
LW R14                  // R15 = Mem[LUT[0]]
MOV R0 R15              // R0 = R15

// increment R14
MOV R13 R0
MOV R0 R14
SET R1 #1
ADD R14
MOV R0 R13

LW R14                  // R15 = Mem[LUT[1]]
MOV R1 R15              // R1 = R15

// NOTE: I will use R10 as a temp register to hold (p8,p4,p2,p1) this will be used for p0 calculation


// calculate p8
MOV R2 R0               // R2 = R0, temp storage for 0000 0b11:b9
MOV R3 R1               // R3 = R1, temp storage for b8:b1
MOV R0 R1               // R0 = R1
SET R1 b'1111 0000      // R1 = b'1111 0000
AND R0                  // R0 = R0 & R1 = b8:b1 & 1111 0000 = b8:b5 0000
MOV R1 R2               // R1 = R2 = 0000 0b11:b9
OR R0                   // R0 = R0 | R1 = b8:b5 0000 | 0000 0b11:b9 = b8:b5 0b11:b9
XORR R0                 // R0 = ^(R0) = p8
MOV R13 R0              // R13 = R0, save p8
SET R10 R13              // R10 = 0000 000p8
MOV R0 R10              // R0 = R10
SET R1 #1               // R1 = 1
SLL R10                 // R10 = R0 << R1 = 0000 00p8 0


// build output 1: (b11:b5, p8)
MOV R4 R2
```

```
// R4 << 5
MOV R0 R4
SET R1 #5
SLL R4
// MOV R5 R3
MOV R5 R3
// R4 >> 3
MOV R0 R4
SET R1 #3
SRL R5
// OR R4 and R5
MOV R0 R4
MOV R1 R5
OR R5
// AND to ge rid of last bit
MOV R0 R5
SET R1 b'1111 1110
AND R5
// OR to put p8 into the output 1
MOV R0 R5
MOV R1 R13
OR R5

// restore input 1 & 2
MOV R0 R2               // R0 = R2, restore 0000 0b11:b9
MOV R1 R3               // R1 = R3, restore b8:b1

// calculate p4
MOV R2 R0               // R2 = R0, temp storage for 0000 0b11:b9
MOV R3 R1               // R3 = R1, temp storage for b8:b1
MOV R0 R1               // R0 = R1
SET R1 b'1000 1110      // R1 = b'1000 1110
AND R4                  // R4 = R0 & R1
MOV R0 R4               // R0 = R4
XORR R0                 // R0 = ^(R0) = ^(b8, b4, b3, b2)
SET R1 #0               // R1 = 0
OR R1                   // R1 = R0 | R1
MOV R0 R2               // R0 = R2, restore R0
XORR R0                 // R0 = ^(R0) = ^(b11:b9)
XOR R0                  // R0 = R0 ^ R1 = p4
MOV R13 R0              // R13 = R0, save p4
MOV R1 R10              // R1 = R10 = 0000 00p8 0
XOR R0                  // R0 = R0 ^ R1 = 0000 00p8 p4
MOV R10 R0              // R10 = R0, save 0000 00p8 p4

// Build up Output 2 -> (b4, b3, b2, p4, b1, 0, 0, 0)
// MOV R4 R3, copy value of R3
MOV R4 R3

// R4 << 4
MOV R0 R4
SET R1 #4
SLL R4

// AND R4 1110 0000
MOV R0 R4
SET R1 b'1110 0000
AND R4

// OR R4 and R13, which is p4
MOV R0 R4
MOV R1 R13
OR R4

// make copy of R3 and put into R5
MOV R5 R3

// AND R5 0000 0001
MOV R0 R5
SET R1 b'0000 0001
AND R5

// R5 << 3
MOV R0 R5
SET R1 #3
SLL R5

// OR R4 R5, R4 is (b4:b2, p4, 0000), R5 is (0000 000 b1)
MOV R0 R4
MOV R1 R5
OR R4

// after this, R4 is (b4:b2, p4, b1, 000)

MOV R0 R2               // R0 = R2, restore 0000 0b11:b9
MOV R1 R3               // R1 = R3, restore b8:b1

// calculate p2
MOV R2 R0               // R2 = R0, temp storage for 0000 0b11:b9
MOV R3 R1               // R3 = R1, temp storage for b8:b1
MOV R0 R1               // R0 = R1
SET R1 b'0110 1101      // R1 = b'0110 1101
AND R5                  // R5 = R0 & R1
MOV R0 R5               // R0 = R5
XORR R0                 // R0 = ^(R0) = ^(b7, b6, b4, b3, b1)
SET R1 #0               // R1 = 0, clear R1
OR R6                   // R6 = R0 | R1, copy R0 into R6
MOV R0 R2               // R0 = R2, restore R0
SET R1 b'1111 1110      // R1 = b'1111 1110
AND R0                  // R0 = R0 & R1 = 0000 0b11:b9 & 1111 1110 = 0000 0b11:b10 0
XORR R0                 // R0 = ^(R0) = ^(b11:b9)
MOV R1 R6               // R1 = R6, restore ^(b7, b6, b4, b3, b1) back into R1
XOR R0                  // R0 = R0 ^ R1 = p2

// Goal: make R4 (b4:b2, p4, b1, p2, 00)
MOV R13 R0
MOV R1 R10            // R1 = R10 = 0000 00p8 p4
XOR R0                // R0 = R0 ^ R1 = 0000 00p8 p4 p2
MOV R10 R0            // R10 = R0, save 0000 00p8 p4 p2

// make R13 0000 0 p2 00
MOV R0 R13
SET R1 #2
SRL R13

// OR R13 and R4 to make (b4:b2, p4, b1, p2, 00)
MOV R0 R13
MOV R1 R4
OR R4

// after this, R4 is (b4:b2, p4, b1, p2, 00)

MOV R0 R2               // R0 = R2, restore 0000 0b11:b9
MOV R1 R3               // R1 = R3, restore b8:b1

// calculate p1
MOV R2 R0               // R2 = R0, temp storage for R0
MOV R3 R1               // R3 = R1, temp storage for R1
MOV R0 R1               // R0 = R1
SET R1 b'0110 1101      // R1 = b'0110 1101
AND R5                  // R5 = R0 & R1
MOV R0 R5               // R0 = R5
XORR R0                 // R0 = ^(R0) = ^(b7, b6, b4, b3, b1)
SET R1 #0               // R1 = 0, clear R1
OR R6                   // R6 = R0 | R1, copy R0 into R6
MOV R0 R2               // R0 = R2, restore R0
SET R1 b'1111 1110      // R1 = b'1111 1110
AND R0                  // R0 = R0 & R1 = 0000 0b11:b9 & 1111 1110 = 0000 0b11:b10 0
XORR R0                 // R0 = ^(R0) = ^(b11:b9)
MOV R1 R6               // R1 = R6, restore ^(b7, b6, b4, b3, b1) back into R1
XOR R0                  // R0 = R0 ^ R1 = p1

// Goal: make R4 (b4:b2, p4, b1, p2, p1, 0)
MOV R13 R0
MOV R1 R10              // R1 = R10 = 0000 00p8 p4 p2
XOR R0                  // R0 = R0 ^ R1 = 0000 00p8 p4 p2 p1
MOV R10 R0              // R10 = R0, save 0000 00p8 p4 p2 p1

// make R13 0000 00 p1 0
MOV R0 R13
SET R1 #1
SRL R13

// OR R13 and R4 to make (b4:b2, p4, b1, p2, p1, 0)
MOV R0 R13
MOV R1 R4
OR R4

// after this, R4 is (b4:b2, p4, b1, p2, p1, 0)

MOV R0 R2               // R0 = R2, restore 0000 0b11:b9
MOV R1 R3               // R1 = R3, restore b8:b1

// calculate p0
XORR R0                 // R0 = ^(R0) = ^(b11:b9)
XORR R1                 // R1 = ^(R1) = ^(b8:b1)
// TODO: xor with p1, p2, p4, p8 as well (NOTE: Check this logic)
XOR R0                  // R0 = R0 ^ R1 = p0
MOV R1 R10              // R1 = R10 = 0000 00p8 p4 p2 p1
XORR R1                 // R1 = ^(R1) = ^(0000 00p8 p4 p2 p1) -> (p8^p4^p2^p1)
XOR R0                  // R0 = R0 ^ R1 = ^(b11:1) ^ (p8^p4^p2^p1)


// Goal: make R4 (b4:b2, p4, b1, p2, p1, 0)
MOV R13 R0

// OR R13 and R4 to make (b4:b2, p4, b1, p2, p1, p0)
MOV R0 R13
MOV R1 R4
OR R4
```

```
// after this, R4 is (b4:b2, p4, b1, p2, p1, 0)

// decrement R14
MOV R0 R14
SET R1 #1
SUB R14

// store output 1 (R5) into data memory
MOV R15 R5
SW R14

// increment iterator variable R14
MOV R0 R14
SET R1 #1
ADD R14

// store output 2 (R4) into data memory
MOV R15 R4
SW R14

// increment iterator variable R14
MOV R0 R14
SET R1 #1
ADD R14

// branch if iterator is 16 (loop has executed 15 times)
SET R0 #32
MOV R1 R14
BEQ LOOP
```

Program 2 Pseudocode

```
word1: b11 | b10 | b9 | b8 | b7 | b6 | b5 | p8
word2: b4  | b3  | b2 | p4 | b1 | p2 | p1 | p0

Output 1: F1 F0 0  0  0  D11 D10 D9 (stored in R5)
Output 2: D8 D7 D6 D5 D4 D3  D2  D1 (stored in R6)

q0 <- ^(b11:1, p8, p4, p2, p1)
q1 <- ^(b11, b9, b7, b5, b4, b2, b1)
q2 <- ^(b11, b10, b7, b6, b4, b3, b1)
q4 <- ^(b11:b8, b4, b3, b2)
q8 <- ^(b11:b5)
loop 15 times through each msg m:
  R0 <- word 1
  R1 <- word 2
  syndrome <- s1, s2, s4, s8
  q0 <- ^(word1, word2)

  // overall parity matches
  if q0 == p0:
    // syndrome bit is 0, no error detected
    if syndrome == 0:
      no error!
      build up output 1 & 2 without any changes to the input bits
    // syndrome bit is non-zero, error detected. but overall parity matches... must be 2 bit corruption
    else:
      2 bit corruption! set F1 = 1
      build up output 1 & 2 without any changes to the input bits
  // overall partiy does not match
  else:
    1 bit error! set F0 = 1
    // correct the error
    flip the bit at msg[syndrome]
```

Program 2 Assembly Code

```
Input 1: b11 b10 b9 b8 b7 b6 b5 p8
Input 2: b4  b3  b2 p4 b1 p2 p1 p0

Output 1: F1 F0 0  0  0  D11 D10 D9 (stored in R5)
Output 2: D8 D7 D6 D5 D4 D3  D2  D1 (stored in R6)

F1F0 = 00 if no error
F1F0 = 01 if 1 error
F1F0 = 1X if 2 error

q0 <- ^(b11:1, p8, p4, p2, p1)
q1 <- ^(b11, b9, b7, b5, b4, b2, b1)
q2 <- ^(b11, b10, b7, b6, b4, b3, b1)
q4 <- ^(b11:b8, b4, b3, b2)
```

```
q8 <- ^(b11:b5)
loop 15 times through each msg m:
  R0 <- word 1
  R1 <- word 2
  syndrome <- q1, q2, q4, q8
  q0 <- ^(word1, word2)

  // overall parity matches
  if q0 == p0:
    // syndrome bit is 0, no error detected
    if syndrome == 0:
      no error!
    // syndrome bit is non-zero, error detected. but overall parity matches... must be 2 bit corruption
    else:
      2 bit corruption! set F2 = 1
  // overall partiy does not match
  else:
    1 bit error! set F1 = 1

    // correct the error
    flip the bit at msg[syndrome]

// TODO: FLIP THE MOV OPERANDS
SET R14 #0              // R14 = 0, used for looping through the 15 2-byte messages

LOOP:

// read from data mem into R1 R0
LW R14                  // R15 = Mem[LUT[0]]
MOV R0 R15              // R0 = R15

// increment R14
MOV R13 R0
MOV R0 R14
SET R1 #1
ADD R14
MOV R0 R13

LW R14                  // R15 = Mem[LUT[1]]
MOV R1 R15              // R1 = R15

// calculate q8 and store in R13
MOV R2 R0               // R2 = R0, temp storage for 0000 0b11:b9
MOV R3 R1               // R3 = R1, temp storage for b8:b1
MOV R0 R1               // R0 = R1
SET R1 b'1111 0000      // R1 = b'1111 0000
AND R0                  // R0 = R0 & R1 = b8:b1 & 1111 0000 = b8:b5 0000
MOV R1 R2               // R1 = R2 = 0000 0b11:b9
OR R0                   // R0 = R0 | R1 = b8:b5 0000 | 0000 0b11:b9 = b8:b5 0b11:b9
XORR R0                 // R0 = ^(R0) = q8
MOV R13 R0

// restore input 1 & 2
MOV R0 R2               // R0 = R2, restore b11 b10 b9 b8 b7 b6 b5 p8
MOV R1 R3               // R1 = R3, restore b4  b3  b2 p4 b1 p2 p1 p0

// calculate q4
MOV R2 R0               // R2 = R0, temp storage for 0000 0b11:b9
MOV R3 R1               // R3 = R1, temp storage for b8:b1
MOV R0 R1               // R0 = R1
SET R1 b'1000 1110      // R1 = b'1000 1110
AND R4                  // R4 = R0 & R1
MOV R0 R4               // R0 = R4
XORR R0                 // R0 = ^(R0) = ^(b8, b4, b3, b2)
SET R1 #0               // R1 = 0
OR R1                   // R1 = R0 | R1
MOV R0 R2               // R0 = R2, restore R0
XORR R0                 // R0 = ^(R0) = ^(b11:b9)
XOR R0                  // R0 = R0 ^ R1 = q4

// Make sure the ordering of the syndrome bits is correct either MSB or LSB
// store q4 into R13 so it looks like 0000 00 q4 q8
SET R1 #1               // R1 = 1
SLL R0                  // Left shift R0 -> 0000 00q40
MOV R1 R13              // R1 = R13, R13 holds q8
OR R13                  // OR R0 and R1 = 0000 00 q4 0 | 0000 000 q8 = 0000 00 q4 q8

// restore input 1 & 2
MOV R0 R2               // R0 = R2, restore b11 b10 b9 b8 b7 b6 b5 p8
MOV R1 R3               // R1 = R3, restore b4  b3  b2 p4 b1 p2 p1 p0

// calculate q2
MOV R2 R0               // R2 = R0, temp storage for 0000 0b11:b9
MOV R3 R1               // R3 = R1, temp storage for b8:b1
MOV R0 R1               // R0 = R1
SET R1 b'0110 1101      // R1 = b'0110 1101
AND R5                  // R5 = R0 & R1
MOV R0 R5               // R0 = R5
XORR R0                 // R0 = ^(R0) = ^(b7, b6, b4, b3, b1)
SET R1 #0               // R1 = 0, clear R1
```

```
OR R6                   // R6 = R0 | R1, copy R0 into R6
MOV R0 R2               // R0 = R2, restore R0
SET R1 b'1111 1110      // R1 = b'1111 1110
AND R0                  // R0 = R0 & R1 = 0000 0b11:b9 & 1111 1110 = 0000 0b11:b10 0
XORR R0                 // R0 = ^(R0) = ^(b11:b9)
MOV R1 R6               // R1 = R6, restore ^(b7, b6, b4, b3, b1) back into R1
XOR R0                  // R0 = R0 ^ R1 = q2

// store q2 into R13 so it looks like 0000 0 q2 q4 q8
SET R1 #2               // R1 = 2
SLL R0                  // Left shift R0 -> 0000 0 q2 00
MOV R1 R13              // R1 = R13, R13 holds -> 0000 00 q4 q8
OR R13                  // OR R0 and R1 = 0000 0 q2 00 | 0000 00 q4 q8 = 0000 0 q2 q4 q8

// restore input 1 & 2
MOV R0 R2               // R0 = R2, restore b11 b10 b9 b8 b7 b6 b5 p8
MOV R1 R3               // R1 = R3, restore b4  b3  b2 p4 b1 p2 p1 p0

// calculate q1
MOV R2 R0               // R2 = R0, temp storage for R0
MOV R3 R1               // R3 = R1, temp storage for R1
MOV R0 R1               // R0 = R1
SET R1 b'0110 1101      // R1 = b'0110 1101
AND R5                  // R5 = R0 & R1
MOV R0 R5               // R0 = R5
XORR R0                 // R0 = ^(R0) = ^(b7, b6, b4, b3, b1)
SET R1 #0               // R1 = 0, clear R1
OR R6                   // R6 = R0 | R1, copy R0 into R6
MOV R0 R2               // R0 = R2, restore R0
SET R1 b'1111 1110      // R1 = b'1111 1110
AND R0                  // R0 = R0 & R1 = 0000 0b11:b9 & 1111 1110 = 0000 0b11:b10 0
XORR R0                 // R0 = ^(R0) = ^(b11:b9)
MOV R1 R6               // R1 = R6, restore ^(b7, b6, b4, b3, b1) back into R1
XOR R0                  // R0 = R0 ^ R1 = q1

// store q1 into R13 so it looks like 0000 q1 q2 q4 q8
SET R1 #3               // R1 = 3
SLL R0                  // Left shift R0 -> 0000 q1 000
MOV R1 R13              // R1 = R13, R13 holds -> 0000 0 q2 q4 q8
OR R13                  // OR R0 and R1 = 0000 q1 000 | 0000 0 q2 q4 q8 = 0000 q1 q2 q4 q8

// restore input 1 & 2
MOV R0 R2               // R0 = R2, restore b11 b10 b9 b8 b7 b6 b5 p8
MOV R1 R3               // R1 = R3, restore b4  b3  b2 p4 b1 p2 p1 p0

// calculate q0
XORR R0                 // R0 = ^(R0) = ^(b11:b9)
XORR R1                 // R1 = ^(R1) = ^(b8:b1)
XOR R0                  // R0 = R0 ^ R1 = q0
MOV R8 R0               // NOTE: Save q0 into R8 for later

// store q0 into R13 so it looks like 000 q0 q1 q2 q4 q8
SET R1 #4               // R1 = 4
SLL R0                  // Left shift R0 -> 000 q0 0000
MOV R1 R13              // R1 = R13, R13 holds -> 0000 q1 q2 q4 q8
OR R13                  // OR R0 and R1 = 000 q0 0000 | 0000 q1 q2 q4 q8 = 000 q0 q1 q2 q4 q8

// restore input 1 & 2
MOV R0 R2               // R0 = R2, restore b11 b10 b9 b8 b7 b6 b5 p8
MOV R1 R3               // R1 = R3, restore b4  b3  b2 p4 b1 p2 p1 p0

// compute syndrome bits = (q0 q1 q2 q4 q8) ^ (p0 p1 p2 p4 p8)
// we have (q0 q1 q2 q4 q8) inside of R13, WE NEED TO EXTRACT THE Ps into R12
// extract p8
SET R1 b'0000 0001      // R1 -> 0000 0001
AND R12                 // R12 -> 0000 0001 & b11 b10 b9 b8 b7 b6 b5 p8 -> 0000 000p8

// extract p4
SET R1 b'0001 0000      // R1 -> 0001 0000
MOV R0 R1               // R0 -> b4  b3  b2 p4 b1 p2 p1 p0
AND R11                 // R11 -> 000p4 0000
MOV R0 R11              // R0 -> 000p4 0000
SET R1 #3               // R1 == 3
SRL R11                 // R11 -> 0000 00p40
MOV R0 R11              // R0 -> 0000 00p40
MOV R1 R12              // R1 -> 0000 000p8
OR R12                  // R12 = R0 | R1 = 0000 00p40 | 0000 000p8 = 0000 00 p4 p8

// extract p2
SET R1 b'0000 0100      // R1 -> 0000 0100
MOV R0 R1               // R0 -> b4  b3  b2 p4 b1 p2 p1 p0
AND R11                 // R11 -> 0000 0p200
MOV R0 R11              // R0 -> 0000 0p200
MOV R1 R12              // R1 -> 0000 00 p4 p8
OR R12                  // R12 = R0 | R1 = 0000 0p200 | 0000 00 p4 p8 = 0000 0 p2 p4 p8

// extract p1
SET R1 b'0000 0010      // R1 -> 0000 0010
MOV R0 R1               // R0 -> b4  b3  b2 p4 b1 p2 p1 p0
AND R11                 // R11 -> 0000 00p10
```

```
MOV R0 R11              // R0 -> 0000 00p10
SET R1 #2               // R1 == 2
SLL R11                 // R11 -> 0000 p1000
MOV R0 R11              // R0 -> 0000 p1000
MOV R1 R12              // R1 -> 0000 0 p2 p4 p8
OR R12                  // R12 = R0 | R1 = 0000 p1000 | 0000 0 p2 p4 p8 = 0000 p1 p2 p4 p8

// extract p0
SET R1 b'0000 0001      // R1 -> 0000 0001
MOV R0 R1               // R0 -> b4  b3  b2 p4 b1 p2 p1 p0
AND R11                 // R11 -> 0000 000p0
MOV R0 R11              // R0 -> 0000 000p0
MOV R9 R0               // NOTE: R9 holds p0 for later
SET R1 #2               // R1 == 4
SLL R11                 // R11 -> 000p0 0000
MOV R0 R11              // R0 -> 000p0 0000
MOV R1 R12              // R1 -> 0000 p1 p2 p4 p8
OR R12                  // R12 = R0 | R1 = 000p0 0000 | 0000 p1 p2 p4 p8 = 000p0 p1 p2 p4 p8

// now R12 holds (p0, p1, p2, p4, p8) and R13 holds (q0, q1, q2, q4, q8)
// create the syndrome bits which is R12 ^ R13 and store into R11
MOV R0 R12
MOV R1 R13
XOR R11
MOV R0 R11              // R0 -> 000s0  s1 s2 s4 s8
SET R1 b'0000 1111      // R1 -> 0000 1111
AND R11                 // R13 -> 000s0  s1 s2 s4 s8 & 0000 1111 -> 0000 s1 s2 s4 s8

// check if overall parity matches. R8 holds q0, R9 holds p0
MOV R0 R8               // R0 = q0
MOV R1 R9               // R1 = p0
BNE OVERALL_DONT_MATCH  // if overall parity don't match, go to else block (labeled by OVERALL_DONT_MATCH)

// overall partiy matches hurray, but what if it is 2 bit corruption. Check syndrome bit non-zero
// R11 holds (s1, s2, s4, s8)
MOV R0 R11              // R0 -> 0000 s1 s2 s4 s8
SET R1 #0               // R1 = 0
BEQ NO_ERROR            // Syndrome == 0 -> no error detected

// syndrome is non-zero, oh no, that means 2 bit corruption. need to set F1 to 1
// Output 1: F1 F0 0  0  0  D11 D10 D9 (stored in R5)
// Output 2: D8 D7 D6 D5 D4 D3  D2  D1 (stored in R6)

// 2-bit corruption
SET R5 b'1000 0000      // R5 -> F1 F0 0  0  0  D11 D10 D9 -> where F1 = 1


// done with this case! Branch to END to avoid entering OVERALL_DONT_MATCH block!
SET R0 #1
SET R1 #1
BEQ NO_ERROR

OVERALL_DONT_MATCH:
// overall parity doesn't match, this means we have corruption somewhere! R11 holds the syndrome
// 1 bit error! set F0 = 1
SET R5 b'0100 0000      // R5 -> F1 F0 0  0  0  D11 D10 D9 -> where F0 = 1

// correct the error
// flip the bit at msg[syndrome]


// TODO: fix corrupt bits within original input (either R2 or R3)
// if syndrome < 8, then we know it is in R2, otherwise it is in R3
MOV R0 R11              // R0 -> 0000 s1 s2 s4 s8
SET R1 #8               // R1 = 8
BLT SYNDROME_IN_R3      // if syndrome < 8, go to SYNDROME_IN_R3
SET R0 'b0000 0001    // R0 = 1
MOV R1 R11        // R1 = syndrome
SLL R0          // R0 << syndrome (Eg. if syn == 3 -> R0 = 0000 1000)
MOV R0 R2         // R0 = R2 -> restore b11 b10 b9 b8 b7 b6 b5 p8
XOR R2          // This should flip the syndome indexed bit in R2


// done with this case! Branch to END to avoid entering SYNDROME_IN_R3 block!
SET R0 #1
SET R1 #1
BEQ NO_ERROR

SYNDROME_IN_R3:
MOV R0 R11        // R0 -> 0000 s1 s2 s4 s8
SET R1 #8       // R1 = 8
SUB R0          // R0 = syn - 8 (Eg. if syn == 11 -> syn - 8 = 3)
SET R1 'b0000 0001    // R1 = 1
SLL R0          // R0 << syndrome (Eg. if syn == 3 -> R0 = 0000 1000)
MOV R0 R3         // R0 = R3 -> restore b4 b3 b2 p4 b1 p2 p1 p0
XOR R3          // This should flip the syndome indexed bit in R3


NO_ERROR:
// build up R5, we have R5 as F1 F0 00 0000, we want R5 to be F1 F0 00 0 b11 b10 b9
```

```
MOV R0 R2               // R0 = R2, restore b11 b10 b9 b8 b7 b6 b5 p8
SET R1 #5               // R1 = 5
SRL R0                  // R0 -> 0000 0 b11 b10 b9
MOV R1 R5               // R1 -> F1 F0 00 0000
OR R5                   // R5 = R0 | R1 = 0000 0 b11 b10 b9 | F1 F0 00 0000 = F1 F0 00 0 b11 b10 b9

// build up R6, we want it to be b8:b1
// add b8:b5
MOV R0 R2               // R0 -> b11 b10 b9 b8 b7 b6 b5 p8
SET R1 b'0001 1110      // R1 = 0001 1110
AND R0                  // R0 = 000 b8 b7 b6 b5 0
SET R1 #3               // R1 = 3
SLL R1                  // R1 = R0 << 3 = b8 b7 b6 b5 0000
MOV R6 R1               // R6 = b8 b7 b6 b5 0000
// add b4:b2
MOV R0 R3               // R0 = b4 b3 b2 p4 b1 p2 p1 p0
SET R1 b'1110 0000      // R1 = 1110 0000
AND R0                  // R0 = b4 b3 b2 0 0000
SET R1 #4               // R1 = 4
SRL R0                  // R0 = R0 >> 4 = 0000 b4 b3 b2 0
MOV R1 R6               // R1 = R6 = b8 b7 b6 b5 0000
OR R6                   // R6 = R0 | R1 = b8 b7 b6 b5 b4 b3 b2 0
// add b1
MOV R0 R3               // R0 = b4 b3 b2 p4 b1 p2 p1 p0
SET R1 b'0000 1000      // R1 = 0000 1000
AND R0                  // R0 = 0000 b1000
SET R1 #3               // R1 = 3
SRL R0                  // R0 = R0 >> 3 = 0000 000b1
MOV R1 R6               // R1 = R6 = b8 b7 b6 b5 b4 b3 b2 0
OR R6                   // R6 = R0 | R1 = b8 b7 b6 b5 b4 b3 b2 b1

// Now R5, R6 holds output 1 & 2

// decrement R14
MOV R0 R14
SET R1 #1
SUB R14


// store output 1 (R5) into data memory
MOV R15 R5
SW R14

// increment iterator variable R14
MOV R0 R14
SET R1 #1
ADD R14

// store output 2 (R4) into data memory
MOV R15 R4
SW R14

// increment iterator variable R14
MOV R0 R14
SET R1 #1
ADD R14

// branch if iterator is 32 (loop has executed 15 times)
SET R0 #32
MOV R1 R14
BEQ LOOP
```

Program 3 Pseudocode

```
Input:
32 1-byte messages in mem[0:31]: (b7 b6 b5 b4 b3 b2 b1 b0)
1 5-bit pattern in mem[32]: (p4 p3 p2 p1 p0 000)

For part a & b:

// get the 5 bit pattern first
R3 <- 5-bit pattern

// variable to keep track of total number of occurences
R4 <- 0

// variable to keep track of number of bytes within which the pattern occurs
R5 <- 0

// loop through all messages and check for 5 bit pattern
Loop 32 times:
  // keep track of number of occurences before this pattern. we will check later
  // whether it stayed the same to determine whether or not the pattern was found in this message
  R8 <- R4

  // find pattern
  R0 <- 1-byte message

  Loop 4 times:
    R6 <- R0 & R9 = first 5 bits of message
    compare R6 and R0. if equal, then increment R4
    left shift R0 by 1

  compare R4 and R8. if not equal, then message was found in this message, increment R5

write R4 into mem[33]
write R5 into mem[34]

For part c:
// get the 5 bit pattern first
R3 <- 5-bit pattern
R10 <- data mem[0]

// variable to keep track of total number of occurences
R4 <- 0

// loop through all messages and check for 5 bit pattern
Loop 31 times:
  R7 <- data mem[iterator+1]
  Loop 8 times:
    // find pattern R10 holds first msg, R7 holds next msg
    R6 <- R10 & b'1111 1000 = first 5 bits of message
    compare R6 and R3. if equal, then increment R4
    left shift R10 by 1
    R11 <- leftmost bit of R7
    right shift R11 by 7 to move bit all the way right
    OR R10 and R11
    left shift R7 by 1

write R4 into mem[35]
```

Program 3 Assembly Code

```
Input:
mem[0:31]: 32 1-byte messages in : (b7 b6 b5 b4 b3 b2 b1 b0)
mem[32]: 1 5-bit pattern in: (p4 p3 p2 p1 p0 000)

Output:
mem[33]: total number of occurences of pattern without byte boundary crossing
mem[34]: total number of bytes in which pattern occurs (without byte boundary crossing I think)
mem[35]: total number of occurences of pattern with byte boundary crossing

//==============================================================================================================
//==============================================================================================================
// PART A & B: Number of Pattern Occurences without Byte Boundary Crossing and Number of Bytes With Message
//==============================================================================================================
//==============================================================================================================

// TODO: FLIP THE MOV OPERANDS
// set loop iterator variable
SET R14 #0              // R14 = 0, used for looping through the 32 1-byte messages

// get the 5 bit pattern first
SET R13 #32             // R13 = 32, used to get the 5-bit pattern in mem[32]
LW R13                  // R15 = Mem[LUT[32]]
MOV R3 R15              // R3 = R15, R3 now holds the 5-bit pattern

// variable to keep track of total number of occurences
SET R4 #0               // R4 = 0, this line is not necessary but is here for clarity

// variable to keep track of number of bytes within which the pattern occurs
SET R5 #0               // R5 = 0, this line is not necessary but is here for clarity

LOOP:

// keep track of number of pattern occurences before analyzing this message. we will check later
// whether it stayed the same to determine if pattern was found in this byte
MOV R8 R4               // R8 = R4

// get the 1-byte message from data mem
LW R14                  // R15 = Mem[LUT[0]]
MOV R0 R15              // R0 = R15
MOV R10 R0              // R10 = R0, make a copy of the 1-bit message in R10

// create mask
SET R9 b'1111 1000      // R9 = b'1111 1000, used to get 5 bits of a pattern at a time

// Loop 4 times to look through the message byte
SET R13 #0              // R13 = 0, start iterator variable off as 0
FIND_PATTERN:

// look at first 5 bits of message
```

```
MOV R0 R10              // R0 = R10 = message
MOV R1 R9               // R1 = R9 = mask
AND R6                  // R6 = R0 & R1 = message & b'1111 1000

// see if masked message and pattern are equal
MOV R0 R6               // move masked message into R0
MOV R1 R3              // move pattern into R1
BNE PATTERN_NOT_FOUND   // branch to PATTERN_NOT_FOUND if masked message is not the pattern

// pattern found!, increment R4
MOV R0 R4               // move total num of occurrences into R0
SET R1 #1               // R1 = 1
ADD R4                  // R4 = R0 + R1 = total num of occurences + 1

PATTERN_NOT_FOUND:

// left shift R10 by 1
MOV R0 R10              // R0 = R10 = message
SET R1 #1               // R1 = 1
SLL R10                 // R10 = R0 << 1, do this so the mask will capture the next 5 bits of the message in the next iteration

// increment R13 loop iterator
MOV R0 R13
SET R1 #1
ADD R13

// check if loop is done
MOV R0 R13
SET R1 #3
BNE FIND_PATTERN        // if loop has not run for 4 times, we are not done checking for the pattern in the message

// compare R4 and R8. if not equal, then message was found in this message, increment R5
MOV R0 R4               // R0 = R4 = total number of occurences, potentially updated after the loop
MOV R1 R8               // R1 = R8 = old total number of occurences
BEQ PATTERN_NOT_IN_BYTE // branch to PATTERN_NOT_IN_BYTE if pattern is not found in the message

// pattern was found in this message!, increment R5
MOV R0 R5               // move total num of occurrences into R0
SET R1 #1               // R1 = 1
ADD R5                  // R5 = R0 + R1 = total num of occurences + 1

PATTERN_NOT_IN_BYTE:

// increment iterator variable R14
MOV R0 R14
SET R1 #1
ADD R14

// branch if iterator is 32 (loop has executed 32 times)
SET R0 #32
MOV R1 R14
BEQ LOOP

// store mem[33]: total number of occurences of pattern without byte boundary crossing
SET R13 #33             // R13 = 32, used to get the 5-bit pattern in mem[32]
MOV R15 R4              // R15 = R4, R4 holds total num pattern occurences without boundary crossing
SW R13                  // Mem[LUT[33]] = R15

// store mem[34]: total number of bytes in which pattern occurs (without byte boundary crossing I think)
SET R13 #34             // R13 = 32, used to get the 5-bit pattern in mem[32]
MOV R15 R5              // R15 = R5, R5 holds total num of bytes in which pattern occurs
SW R13                  // Mem[LUT[34]] = R15

//=========================================================================================================
//=========================================================================================================
// PART C: Number of Pattern Occurences with Byte Boundary Crossing
//=========================================================================================================
//=========================================================================================================

// set loop iterator variable
SET R14 #1              // R14 = 1, used for looping through the 32 1-byte messages

// get the 5 bit pattern first
SET R13 #32             // R13 = 32, used to get the 5-bit pattern in mem[32]
LW R13                  // R15 = Mem[LUT[32]]
MOV R3 R15              // R3 = R15, R3 now holds the 5-bit pattern

// get the first message byte
SET R13 #0              // R13 = 0, used to get the first message byte in mem[0]
LW R13                  // R15 = Mem[LUT[0]]
MOV R10 R15             // R10 = R15, R10 now holds the first message byte

// variable to keep track of total number of occurences
SET R4 #0               // R4 = 0, this line is not necessary but is here for clarity

LOOP_31_TIMES:

// load next message byte into R7
LW R14                  // R15 = Mem[LUT[0]]
MOV R7 R15              // R7 = R15, R10 now holds the first message byte
```

```
// find pattern with crossing boundaries, iterator variable is R13
SET R13 #0
FIND_PATTERN_CB:

// look at first 5 bits of message
MOV R0 R10              // R0 = R10 = message
SET R1 b'1111 1000      // R1 = b'1111 1000 = mask
AND R6                  // R6 = R0 & R1 = message & b'1111 1000

// see if masked message and pattern are equal
MOV R0 R6               // move masked message into R0
MOV R1 R3               // move pattern into R1
BNE PATTERN_NOT_FOUND_CB   // branch to PATTERN_NOT_FOUND if masked message is not the pattern

// pattern found!, increment R4
MOV R0 R4               // move total num of occurrences into R0
SET R1 #1               // R1 = 1
ADD R4                  // R4 = R0 + R1 = total num of occurences + 1

PATTERN_NOT_FOUND_CB:

// left shift R10 by 1
MOV R0 R10              // R0 = R10 = message
SET R1 #1               // R1 = 1
SLL R10                 // R10 = R0 << 1, do this so the mask will capture the next 5 bits of the message in the next iteration

// make R11 hold the leftmost bit of R7, it is the bit across the boundary
MOV R0 R7
SET R1 b'0000 0001
AND R11

// right shift R11 by 7 to move bit all the way right, b000 0000 -> 0000 000b
MOV R0 R11
SET R1 #7
SRL R11

// OR R10 and R11, move the bit across the boundary
MOV R0 R10
MOV R1 R11
OR R10

// left shift R7 by 1
MOV R0 R7
SET R1 #1
SLL R7

// increment R13 loop iterator
MOV R0 R13
SET R1 #1
ADD R13

// check if loop is done
MOV R0 R13
SET R1 #7
BNE FIND_PATTERN_CB     // if loop has not run for 8 times, we are not done checking for the pattern in the message

// increment iterator variable R14
MOV R0 R14
SET R1 #1
ADD R14

// branch if iterator is 32 (loop has executed 31 times)
SET R0 #32
MOV R1 R14
BEQ LOOP_31_TIMES

// Get the last remaining 4 patterns checked
SET R13 #0              // R13 = 0, start iterator variable off as 0
FIND_PATTERN_CB_LAST:

// look at first 5 bits of message
MOV R0 R10              // R0 = R10 = message
MOV R1 R9               // R1 = R9 = mask
AND R6                  // R6 = R0 & R1 = message & b'1111 1000

// see if masked message and pattern are equal
MOV R0 R6               // move masked message into R0
MOV R1 R3               // move pattern into R1
BNE PATTERN_NOT_FOUND   // branch to PATTERN_NOT_FOUND if masked message is not the pattern

// pattern found!, increment R4
MOV R0 R4               // move total num of occurrences into R0
SET R1 #1               // R1 = 1
ADD R4                  // R4 = R0 + R1 = total num of occurences + 1

PATTERN_NOT_FOUND:

// left shift R10 by 1
MOV R0 R10              // R0 = R10 = message
```

```
SET R1 #1               // R1 = 1
SLL R10                 // R10 = R0 << 1, do this so the mask will capture the next 5 bits of the message in the next iteration

// increment R13 loop iterator
MOV R0 R13
SET R1 #1
ADD R13

// check if loop is done
MOV R0 R13
SET R1 #3
BNE FIND_PATTERN        // if loop has not run for 4 times, we are not done checking for the pattern in the message

// store mem[35]: total number of occurences of pattern with byte boundary crossing
SET R13 #35             // R13 = 35, used to get the 5-bit pattern in mem[32]
MOV R15 R4              // R15 = R4, R4 holds total num pattern occurences without boundary crossing
SW R13                  // Mem[LUT[33]] = R15
```