# CSE 141L Final Report

Darryl Tayag, A16164193; Mohammed Master, A16136826; Muhammad Imran, A15867501

## 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.

Darryl Tayag
Mohammed Master
Muhammad Imran

# 0. Team

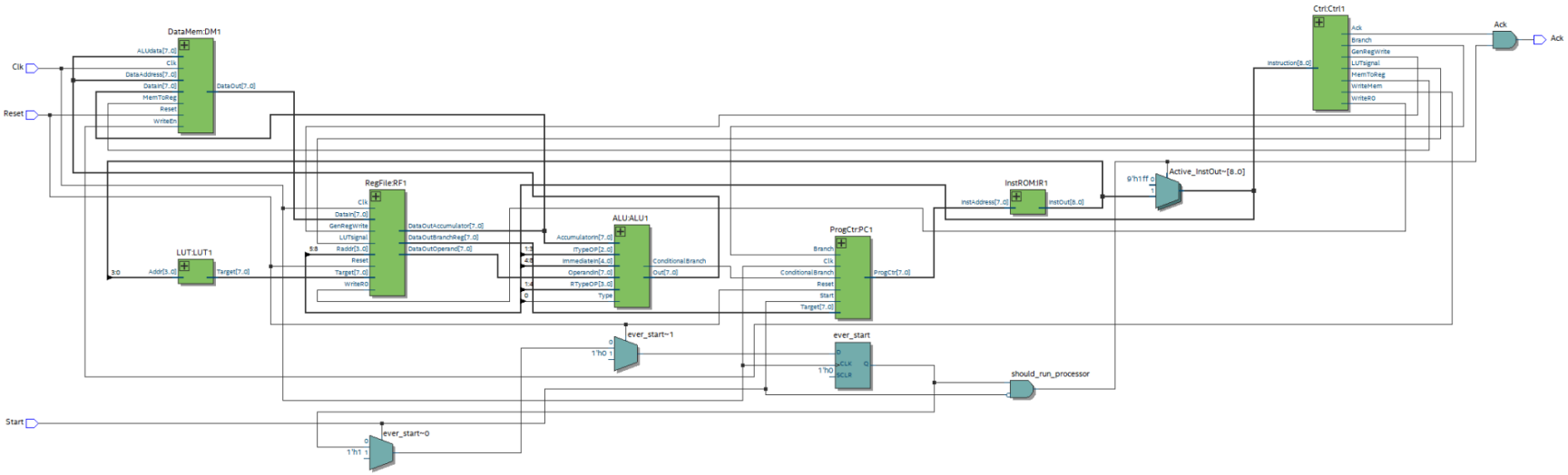
Darryl Tayag
Mohammed Master
Muhammad Imran

# 1. Introduction

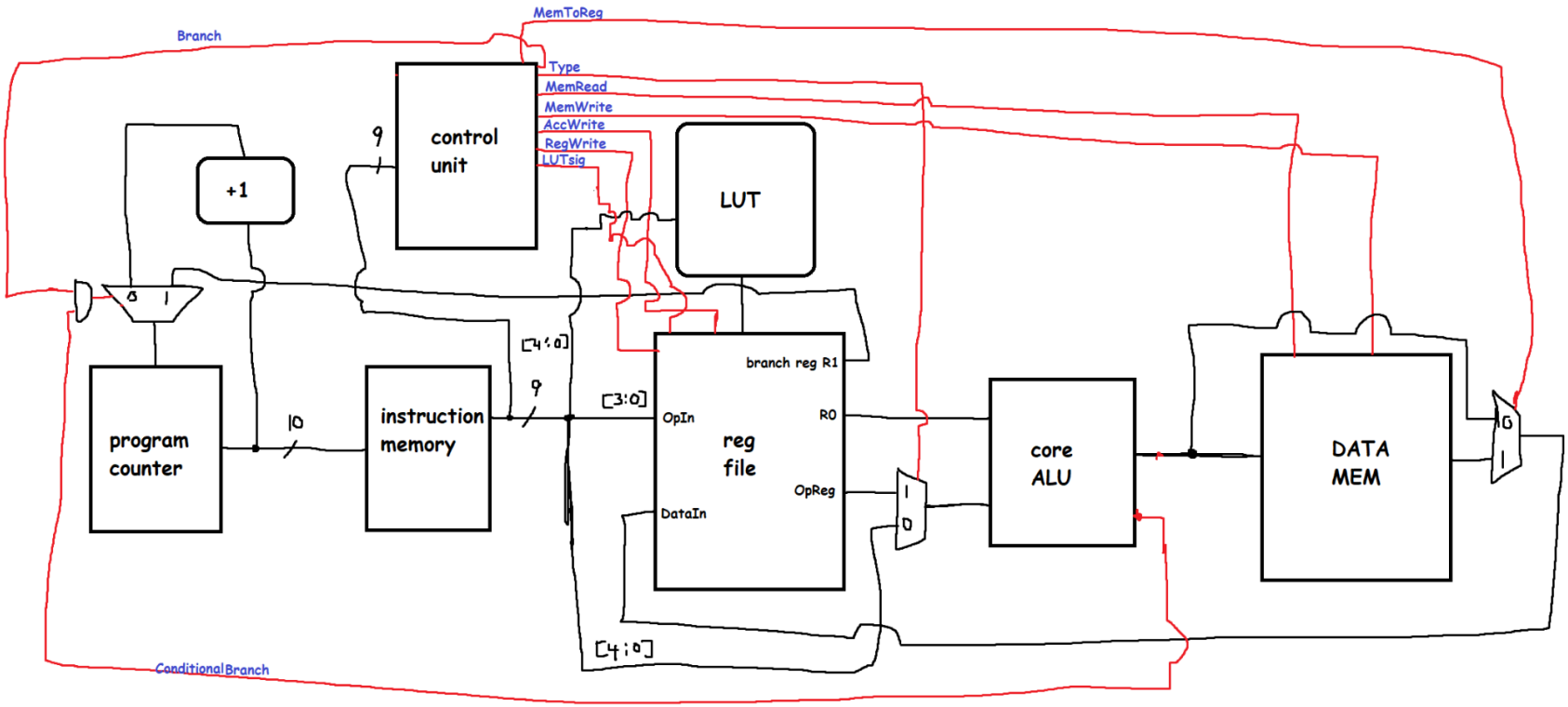
The name of our architecture is WWFC (Weijia & Wesley Fan Club) 😎 Our philosophy is to keep everything short and simple. To follow this philosophy, we have decided to use an accumulator machine type for our architecture. By having an implicitly dedicated accumulator register, we can keep our instructions *short* because we only need to define one other operand in our instruction. We are also able to free up more bits in our 9-bit instruction to use more registers, use more operations, and leave more bits open for the immediate. To handle *simplicity*, we are making one register to all the heavy lifting, R0, the accumulator register. This register will be the only register that gets updated, all operations will have the accumulator register implicitly defined as the operand and destination register. The only exception to this is when we move the value of the accumulator into a general purpose register; this is the only case where we would write to another register other than the accumulator. Finally, the most important aspect of our ISA… our love for Weijia and Wesley is what provides power to architecture.

# 2. Architectural Overview

From Quartus:



From MS Paint (totally not scuffed):



# 3. Machine Specification

## Instruction formats

| TYPE | FORMAT | CORRESPONDING INSTRUCTIONS |
|---|---|---|
| R | 9'b0_xxxx_xxxx<br><br>Specifies R Type<br>Opcode<br>Operand Register | ADD, LOAD, MVFR, MVTO, OR, XOR, XORR, AND, STR, SLT, SEQ, BTRU, SUB, NOT<br><br>14 R-type instructions |
| I | 9'b1_xxxx_xxxx<br><br>Specifies I Type<br>Opcode<br>Immediate | LUT, ADDI, SUBI, B, LSLI, LSRI, HALT<br><br>5 I-type instructions and 2 special instructions B and HALT that don't deal with immediates |

## Operations

| NAME | TYPE | BIT BREAKDOWN | EXAMPLE | NOTES |
|---|---|---|---|---|
| ADD | R | 0_0000_xxxx | ADD R2<br>0_0000_0010<br>R0 = R0 + R2 | Adds the value in the operand register (xxxx) to the accumulator register (R0) |
| LOAD | R | 0_0001_xxxx | LOAD R2<br>0_0001_0010<br>R0 = Mem[R2] | Loads byte from memory at the address specified by the operand register (xxxx) into the accumulator register (R0) |
| MVFR | R | 0_0010_xxxx | MVFR R2<br>0_0010_0010<br>R2 = R0 | Moves the contents FROM the accumulator register (R0) into the operand register (xxxx) |
| MVTO | R | 0_0011_xxxx | MVTO R2<br>0_0011_0010<br>R0 = R2 | Moves the contents in the operand register (xxxx) TO the accumulator register (R0) |
| OR | R | 0_0100_xxxx | OR R2<br>0_0100_0010<br>R0 = R0 \| R2 | Bitwise OR the contents of the operand register (xxxx) with the accumulator register (R0) and store the resulting byte into R0 |
| XOR | R | 0_0101_xxxx | XOR R2<br>0_0101_0010<br>R0 = R0 ^ R2 | Bitwise XOR the contents of the operand register (xxxx) with the accumulator register (R0) and store the resulting byte into R0 |
| XORR | R | 0_0110_xxxx | XORR R2 | Reduction XOR the contents of |

| | | | | |
|---|---|---|---|---|
| | | | 0_0110_xxxx<br>R0 = ^[R2] | the operand register (xxxx) and put result into accumulator (R0) |
| AND | R | 0_0111_xxxx | XOR R2<br>0_0111_0010<br>R0 = R0 & R2 | Bitwise AND the contents of the operand register (xxxx) with the accumulator register (R0) and store the resulting byte into R0 |
| STR | R | 0_1000_xxxx | STR R2<br>0_1000_0010<br>Mem[R2] = R0 | Store the contents of the accumulator register (R0) into the address in memory specified by the operand register (xxxx) |
| SLT | R | 0_1001_xxxx | SLT R2<br>0_1001_0010<br>if R0 < R2:<br>  R0 = 1<br>else:<br>  R0 = 0 | Compares value in operand register (xxxx) to accumulator register (R0), if R0 is less than operand, store 1 in R0, otherwise, store 0 in R0 |
| SEQ | R | 0_1010_xxxx | SEQ R2<br>0_1010_0010<br><br>if R0 == R2:<br>  R0 = 1<br>else:<br>  R0 = 0 | Compares value in operand register (xxxx) to accumulator register (R0), if R0 is equal to operand, store 1 in R0, otherwise, store 0 in R0 |
| BTRU | R | 0_1011_xxxx | BTRU<br>0_1011_xxxx<br>if R0 == 1:<br>  branch to branch reg | If the accumulator is “true” - equal to 1 - then we branch to the address specified in the branch register (R1) |
| SUB | R | 0_1100_xxxx | SUB R0<br>0_1100_0000 | Clear out the register specified by the operand |

| | | | | |
|---|---|---|---|---|
| | | | R0 = R0 - R0 = 0 | register (xxxx); usually going to be the accumulator register |
| ..<br>NOT | ..<br>R | ..<br>0_1101_xxxx | ..<br>NOT R2<br>0_1101_0010<br>R0 = R0[;~R2] | ..<br>Flips the bit specified by the operand register (xxxx) within the byte in the accumulator register (R0) and stores the result back into the accumulator |
| LUT | I | 1_000x_xxxx | LUT #10<br>0_0000_1010<br><br>R0 = LUT[10] | Load into the accumulator register (R0), the value stored in the LUT at the address specified by the immediate |
| ADDI | I | 1_001x_xxxx | ADDI #3<br>0_0010_0011<br>R0 = R0 + 3 | Add the immediate value to the accumulator register (R0) and store the result back into the accumulator register |
| SUBI | I | 1_010x_xxxx | SUBI #8<br>1_0100_1000<br>R0 = R0 - 8 | Subtract the specified immediate value from the accumulator register (R0) and store the result back into the accumulator |
| B | I | 1_011x_xxxx | B<br>1_011x_xxxx | Branch to the address specified in the branch register (R1) |
| LSLI | I | 1_100x_xxxx | LSLI #4<br>1_1000_0100<br>R0 = R0 << 4 | Left shift byte in accumulator register by value specified by immediate and store back into accumulator register (R0) |
| LSRI | I | 1_101x_xxxx | LSRI #4 | Right shift byte in accumulator |

| | | | | |
|---|---|---|---|---|
| | | | 1_1010_0100<br>R0 = R0 >> 4 | register by value specified by immediate and store back into accumulator register (R0) |
| HALT | I | 1_1111_1111 | HALT<br>1_1111_1111 | Stops program |

## Internal Operands

Our ISA supports a total of 16 registers. R0 will be the accumulator register, and R1 will be the branch register. The rest of the registers are general purpose registers.

R0 = accumulator; R1 = branch; R2-R15 = general purpose

## Control Flow (branches)

Our ISA supports conditional branches using `SLT, SEQ,` and `BTRU` instructions and will branch to whatever target is in the branch register on a condition. We also have an unconditional branch instruction `B` that will just branch to whatever is in the branch register under no condition. The necessary branch targets will be calculated by looking at which lines in the assembly code the labels land and these will be stored into a LUT. Our ISA will allow branching up to 2^8 addresses away because our LUT will be 8 bits wide.

## Addressing Modes

We support indirect addressing. Addresses in memory are calculated by incrementing the accumulator register to the desired memory index. For a `LOAD` operation, you first increment the accumulator to the desired index and then call the load operation on the register with the index – in our example, R0 has the index – then R0 would get loaded with the memory value at index specified by the operand register. For a `STR` operation, you would do mostly the same thing. Increment accumulator to desired store address, then you would move that address into a general purpose register, reload the accumulator with the contents you want to store, and then call the store operation with the general purpose register that holds the index as an operand.

Example (`LOAD`):

```
ADDI  #31         // R0 = 41
ADDI  #25         // R0 = 31 + 25 = 56
LOAD  R0          // R0 = Mem[56]
```

Example (`STR`):

```
ADDI  #31         // R0 = 41
ADDI  #25         // R0 = 31 + 25 = 56
MVFR  R2          // R2 = R0
MVTO  R5          // R0 = R5
LOAD  R2          // Mem[R2] = R0
```

# 4. Programmer's Model [Lite]

4.1 Since our machine is an accumulator with general purpose registers R2-R15, the programmer should prioritize loading all the necessary values from memory through the accumulator and into as many general purpose registers as possible. After all the necessary values are loaded into general purpose registers, you can move in between the accumulator and general purpose registers to execute arithmetic operations, the results would be placed into the accumulator and then moved again into one of the many general purpose registers to later be accessed and stored somewhere in memory. So a basic rundown of how the program should think about how the machine operates: Load necessary data from memory into accumulator step by step → put this data from the accumulator into general purpose registers → conduct arithmetic operations using the general purpose registers and accumulator → store result back into memory

4.2 No, we cannot. We first tried doing a classic load-store architecture like MIPS/ARM with 2 operands, but we ran into some issues down the line because we were restricted with the number of bits and therefore number of registers we could use for operands. To overcome this obstacle, we decided to make the transition to the accumulator design where one of the operand registers were implied. This allowed us to expand to 16 total registers instead of being stuck with 4 which was very nice.

4.3 Our ALU will *NOT* be used for non-arithmetic instructions. All instructions that use the ALU will be arithmetic instructions. So no complications here. We won't be calculating any memory pointer addresses or using PC relative branch computations… All of the stuff that has to deal with branch targets will be preloaded into the LUT.

# 5. Individual Component Specification

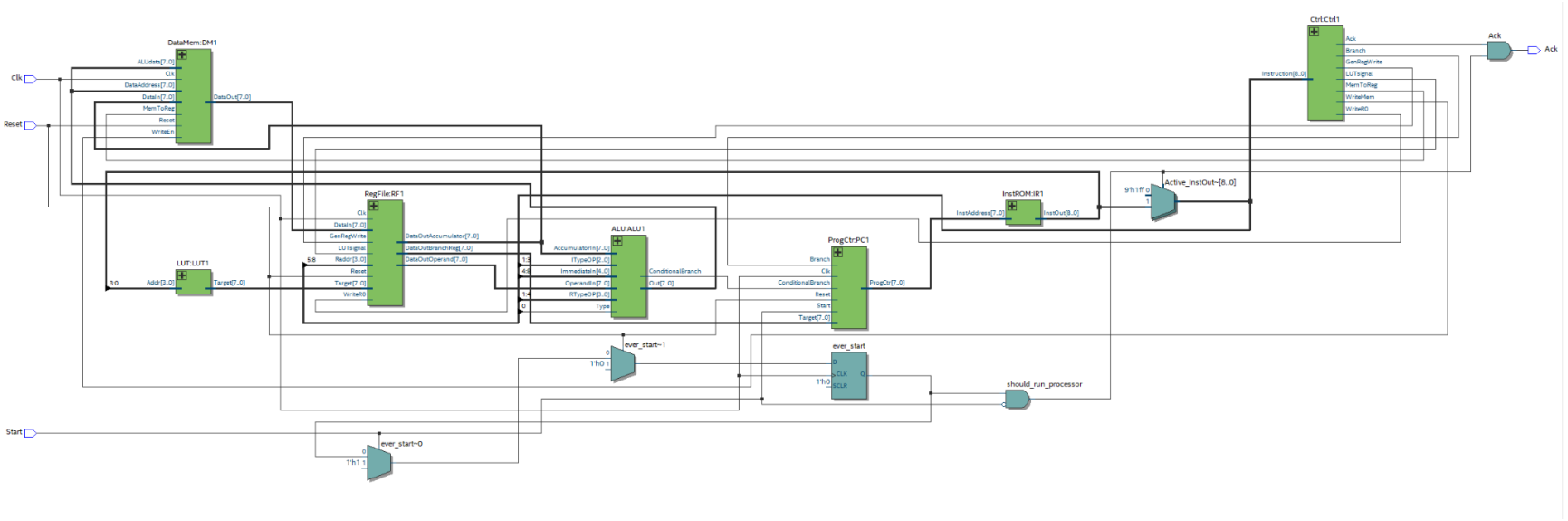
## Top Level

Module file name: TopLevel.sv

### Functionality Description

Responsible for instantiating all components of the ISA and hooking up all the connections together. Will output an *Ack* signal when the program is finished (`HALT` instruction).

### Schematic



# Program Counter

Module file name: ProgCtr.sv
Module testbench file name: ProgCtr_tb.sv

## Functionality Description

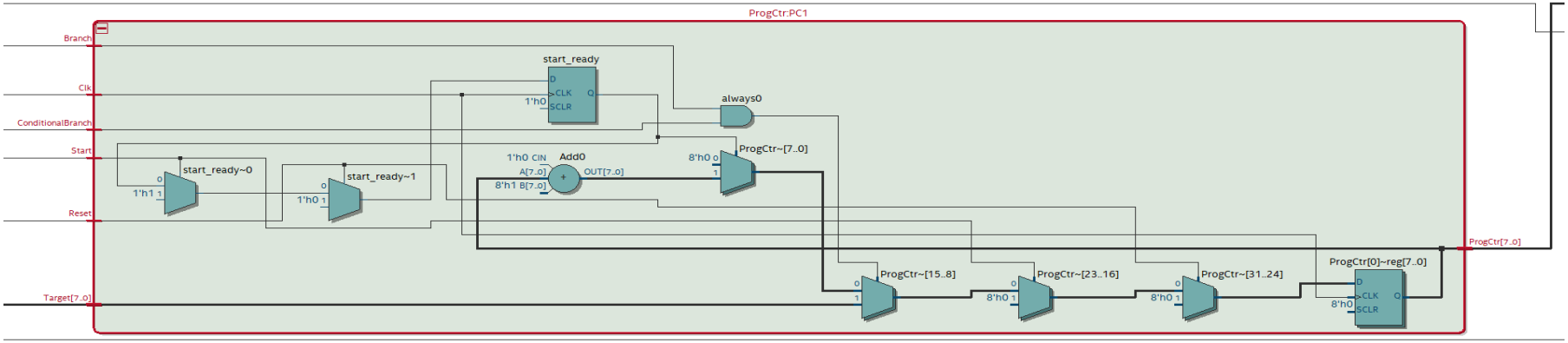
The program counter is responsible for keeping track of which instruction we are running in Instruction Memory. If the program counter receives the *Branch* and *Unconditional* branch signals, then the program counter will be set to the input branch target, otherwise the program counter just gets incremented by 1 every time.

## Testbench Description

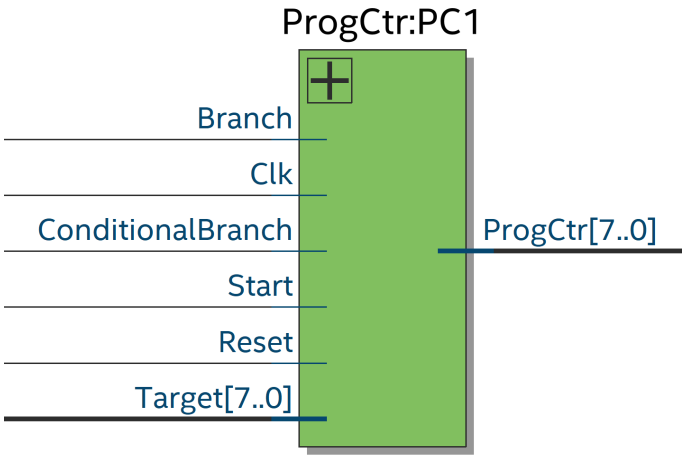
In our test bench, we first set the *reset* bit to high, then we lower it. In this period, we should not see the Program Counter get incremented at all. Then, we raise the *start* signal, and it is not until we lower the *start* signal that the Program Counter should start incrementing. After observing that, we test the Program Counter's functionality with branching. So we set a target address and raise *Branch* and *Unconditional* branch signals to high. We observe that the Program Counter correctly jumps to the branch address and starts incrementing from there. Then, we also test wanting to branch back to a previous address. At the end, we raise *reset* and lower *reset* to set the Program Counter back to 0 and start it up again.

## Schematic

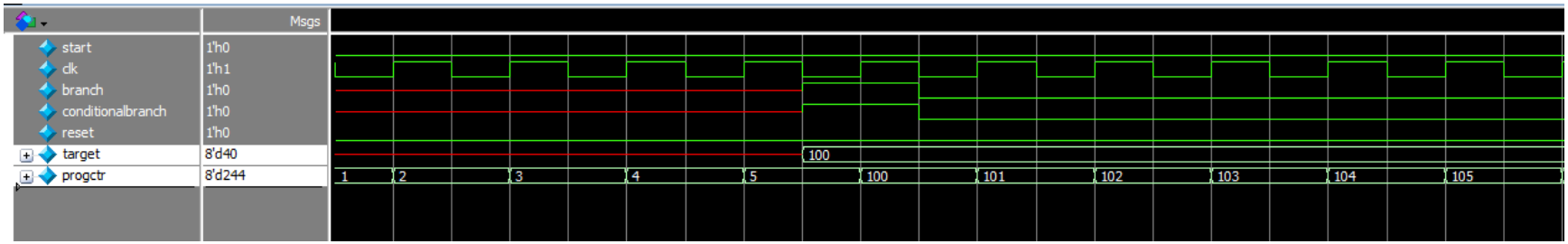
Expansive View:




High-Level View:



## Timing Diagram




Program Counter correctly increments from 0 and then branches to the target address when the branch signals are high and increments from there.

# Instruction Memory

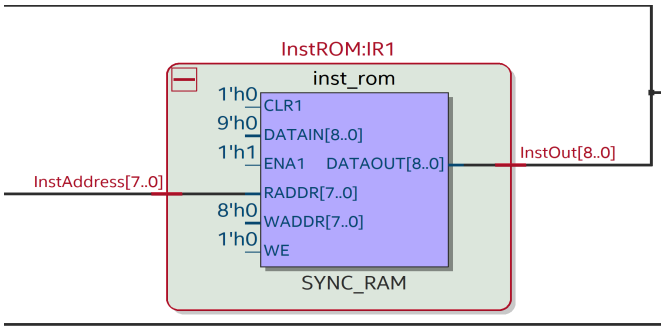
Module file name: InstROM.sv

## Functionality Description

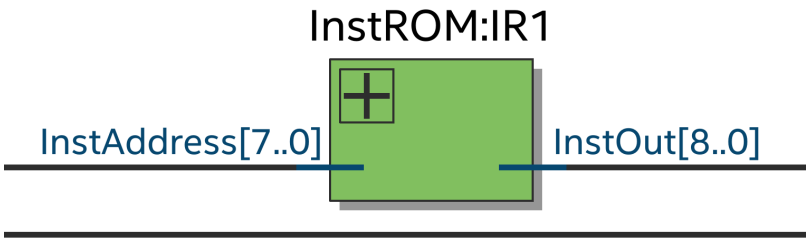
9-bit wide, 256 entries long. Holds all the machine code instructions for a program. Outputs an InstOut which is determined by the index of InstAddress fetched from the Program Counter.

## Schematic

Expansive View:



High-Level View:



# Control Decoder

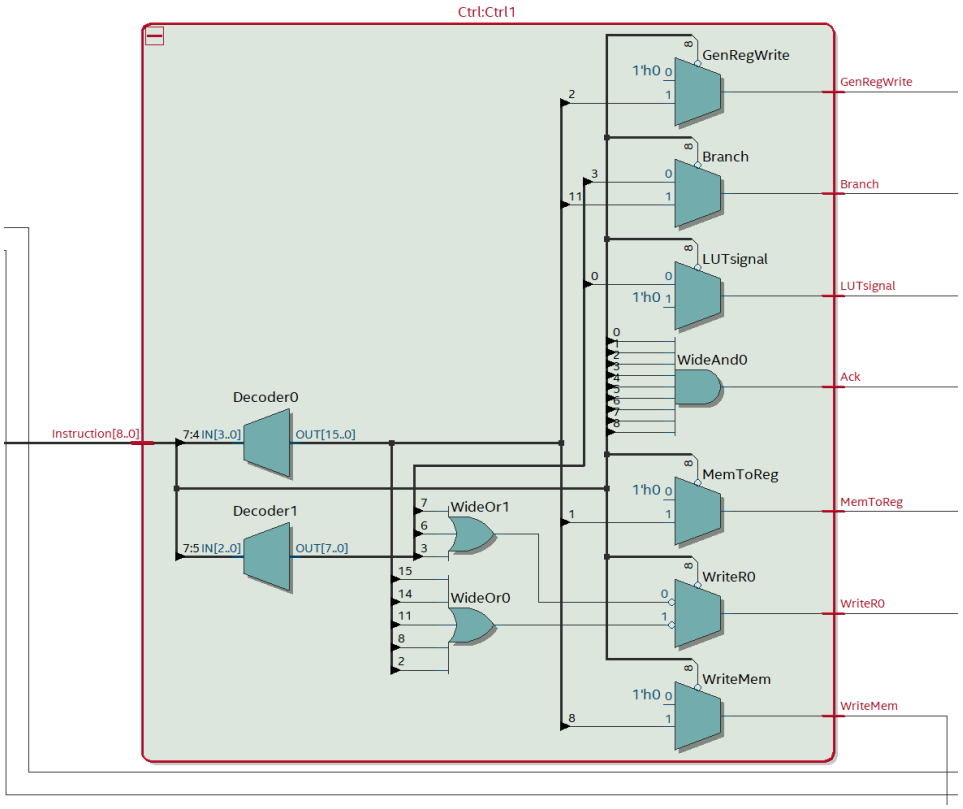
Module file name: Ctrl.sv

## Functionality Description

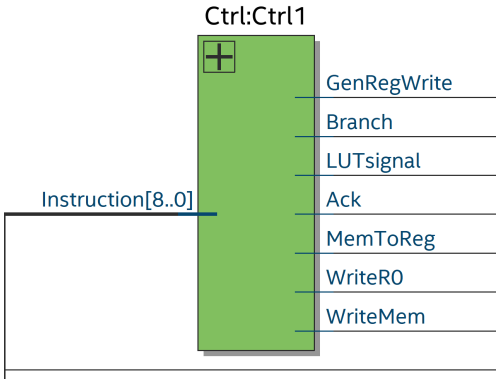
Deconstructs the active 9-bit instruction into signal outputs.

## Schematic

Expansive View:



High-Level View:




# Register File

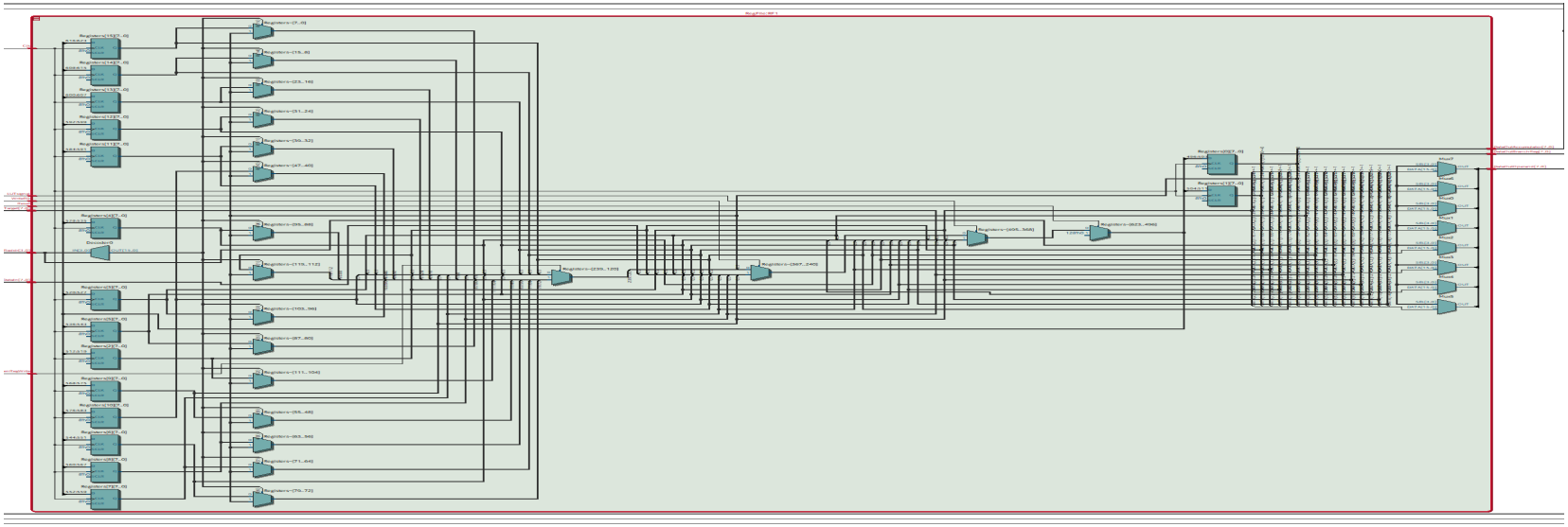
Module file name: RegFile.sv

## Functionality Description

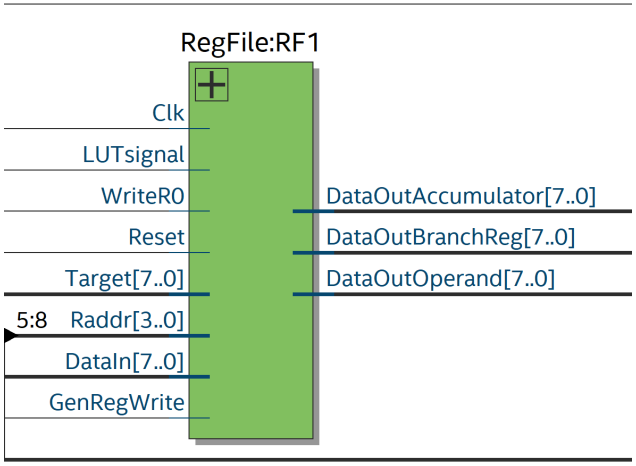
Holds all the registers: accumulator register, branch register, and general purpose registers. The output of the branch register is connected to the Program Counter, there are also outputs from the Accumulator register and the Operand register that connect to the ALU.

# Schematic

## Expansive View:



## High-level View:



# ALU (Arithmetic Logic Unit)

Module file name: ALU.sv
Module testbench file name: ALU_tb.sv

## Functionality Description

Breaks down the 9-bit active instruction to determine what type of operation we want to perform. Executes desired operation and outputs result. If the operation is a branch operation, then the ALU will raise a branching signal.

## Testbench Description

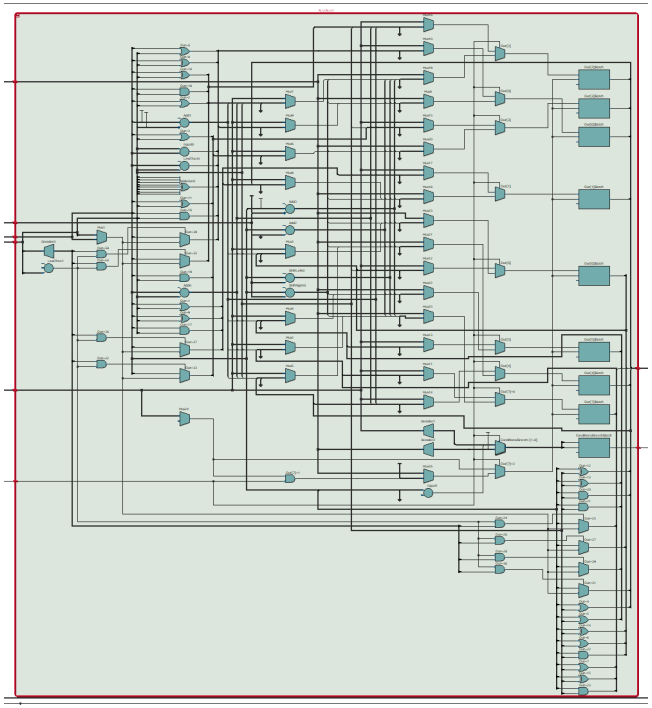
In the testbench, basically what we do is we manually set the instruction we want to observe and also manually set the operand values of the accumulator and operand register. We have display statements with the expected output and the output of our ALU for debugging purposes.

## ALU Operations

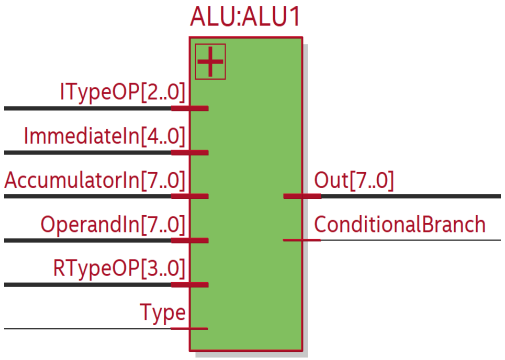
In the testbench, we tested *ALL* ALU operations. ADD, OR, XOR, AND, ADDI, LSRI, etc.

# Schematic

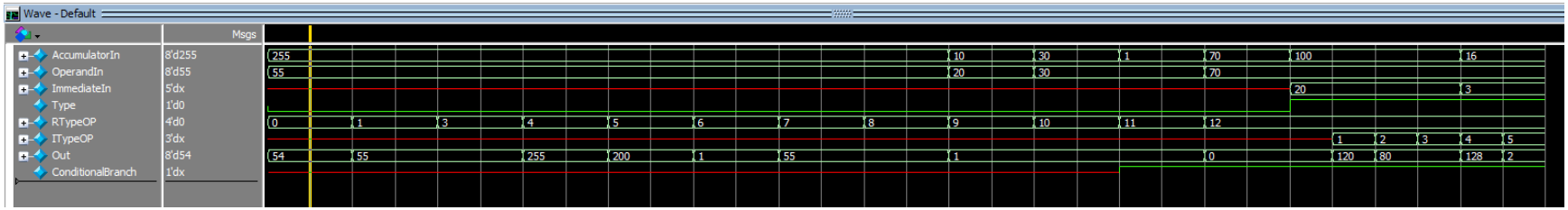
## Expansive View:



## High-Level View:



## Timing Diagram




Debugging Transcript Output:

```
# add Out = 54?  54
# load Out = 55?  55
# mvto Out = 55?  55
# or Out = 255? 255
# xor Out = 200? 200
# xorr Out = 1?   1
# and Out = 55?  55
# str Out = 55?  55
# slt Out = 1?   1
#  seq Out = 1?   1
# btru Out = 1?   1
# sub Out = 0?   0
#  ADDI Out = 120? 120
# SUBI Out = 80?  80
# B (unconditional) Conditional = 1? 1
# LSLI sub Out = 128? 128
# LSRI sub Out = 2?   2
```

# Data Memory

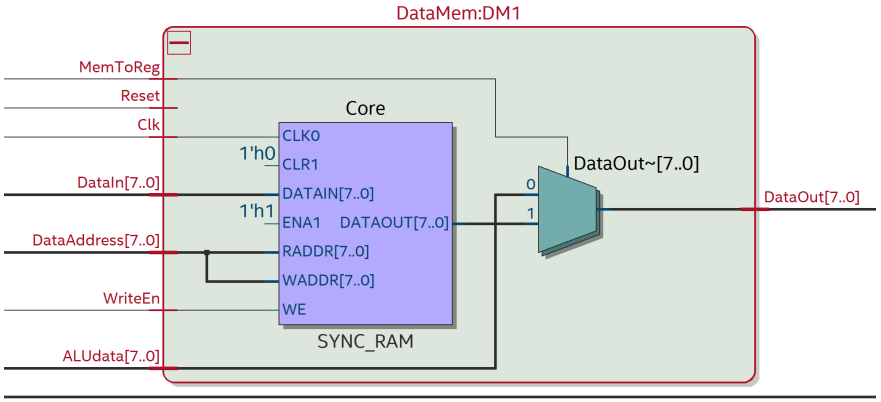
Module file name: DataMem.sv

## Functionality Description

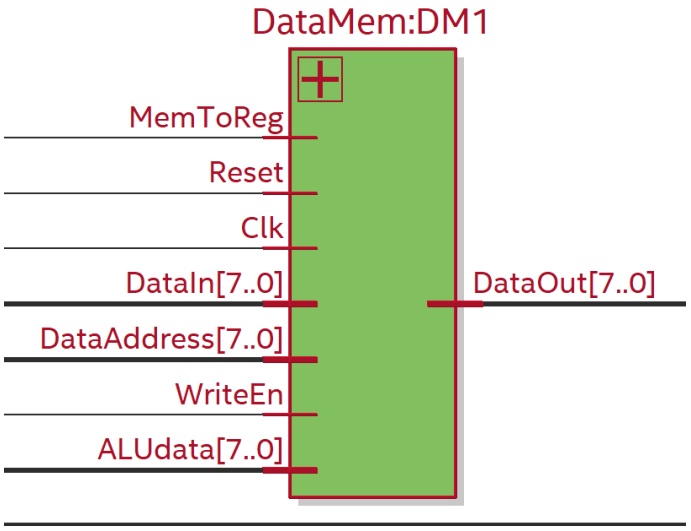
Holds an 8-bit wide, 256 entry long memory Core. *MemToReg* input signal determines whether or not we output the ALU data or Memory Data from this component. When *WriteEn* is high, we write the input data into the Core at the specified index.

# Schematic

## Expansive View:




## High-Level View:



# Lookup Tables

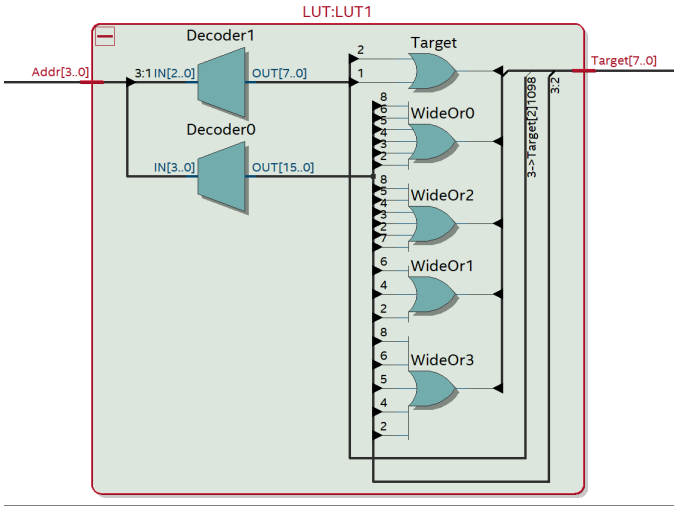
Module file name: LUT.sv

## Functionality Description

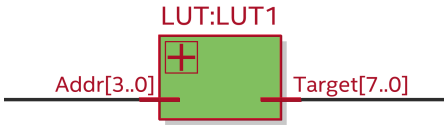
8-bit wide, 16 entry long Lookup Table. Holds all the branching targets for each program. Takes in *Addr* as an input that specifies which index in the LUT to point to. Outputs the branch *Target* that is located at the specified index.

## Schematic

Expansive View:




High-Level View:



# 6. Assembler

Our assembler is written in python. It translates our human-readable assembly code into the machine code 1s and 0s (beep boop). The assembler maps registers and operations to their binary representations, reads an assembler.txt file and outputs the binary into machine_code.txt file. Here is the usage.txt file that describes how to use our assembler:

> How to use assembler.py
>
> 1. Install python
>
> 2. Place the assembler.py and your assembly text file in the same directory.
> NOTE: The assembly text file must be named 'assembly.txt'
>
> 3. Using the command line/your favorite IDE, navigate to the directory that
> the assembler and assembly text file is in.
>
> 4. In the command line, type: 'python assembler.py'
>
> 5. The assembler should run and a machine_code.txt file should be outputted
> within the same directory.

Example:

Assembly Code → assembler.py → Machine Code:

```
LOAD R2                    000010010
MVFR R4                    000100100
SUB R0                     011000000
ADDI #1                    011000000
```

# 7. Program Implementation

## Program 1 Pseudocode

```
void program1(unsigned char mem[]) {
  for (int i = 0; i < 30; i = i + 2) {
    unsigned char MSW = mem[i + 1];
    unsigned char LSW = mem[i];

    mem[i + 31] = 0;
    mem[i + 30] = 0;

    for (int j = 0; j < 3; j++) {
      mem[i + 31] = mem[i + 31] ^ ((MSW >> j) & 1);
    }

    for (int j = 0; j < 4; j++) {
      mem[i + 31] = mem[i + 31] ^ ((LSW >> (4 + j)) & 1);
    }

    unsigned char p4 = 0;
    for (int j = 0; j < 3; j++) {
      p4 = p4 ^ ((MSW >> j) & 1);
    }

    unsigned char mask = 0b10001110;
    unsigned char temp = mask & LSW;

    for (int j = 0; j < 8; j++) {
      p4 = p4 ^ ((temp >> j) & 1);
    }
```

```
  mem[i + 30] = m[i + 30] | (p4 << 4);
}

unsigned char p2 = 0;
for (int j = 0; j < 2; j++) {
  p2 = p2 ^ ((MSW >> j + 1) & 1);
}
mask = 0b01101101;
temp = mask & lsw;

for (int j = 0; j < 8; j++) {
  p2 = p2 ^ ((temp >> j) & 1);
}
for (int j = 0; j < 8; j++) {
  p2 = p2 ^ ((temp >> j + 1) & 1);
}
mem[i + 30] = m[i + 30] | (p2 << 2);

unsigned char p1 = 0;
mask = 0b00000101;
temp = mask & MSW;
for (int j = 0; j < 8; j++) {
  p1 = p1 ^ ((temp >> j) & 1);
}
mask = 0b1011011;
temp = mask & LSW;
for (int j = 0; j < 2; j++) {
  p1 = p1 ^ ((temp >> j + 1) & 1);
}
mem[i + 30] = m[i + 30] | (p1 << 1);
```

```
unsigned char p0 = 0;
for (int j = 0; j < 3; j++) {
  p0 = p0 ^ ((MSW >> j) & 1);
}
for (int j = 0; j < 8; j++) {
  p0 = p0 ^ ((temp >> j) & 1);
}
p0 = p0 ^ mem[i + 31];
p0 = p0 ^ p4;
p0 = p0 ^ p2;
p0 = p0 ^ p1;

mem[i + 30] = mem[i + 30] | p0;

mem[i + 31] = mem[i + 31] | (MSW << 5);

mask = 0b11110000;
temp = mask & LSW;
temp = temp >> 3;

mem[i + 31] = mem[i + 31] | temp;

mask = 0b00001110;
temp = mask & LSW;
temp = temp << 4;
mem[i + 30] = mem[i + 30] | temp;

mask = 0b00000001;
temp = mask & LSW;
temp = temp << 3;
mem[i + 30] = mem[i + 30] | temp;
```

```
}
```

## Program 1 Assembly Code

```
Note: R0 accumulator register; R1 branch register; R2-R15 General Purpose
LUT[0] = &[Loop]

R2 = looping variable (i)
R3 = input MSW
R4 = input LSW
R11 = p8
R12 = p4
R13 = p2
R14 = p1
R15 = p0

Loop:
      // Loading registers with input MSW and input LSW
      LOAD  R2              # R0 = Mem[i]
      MVFR  R4              # R3 = Mem[i]; LSW
      SUB   R0              # R0 = 0
      ADDI  #1              # R0 = 1
      ADD   R2              # R0 = i + 1
      LOAD  R0              # R0 = Mem[i+1]
      MVFR  R3              # R4 = Mem[i+1]; MSW

      // Calculating p8
      MVTO  R4              # R0 = R4 (LSW)
      LSRI  #4              # R0 = R0 >> 4
      XOR   R3              # R0 = R0 ^ R3
      XORR  R0              # R0 = ^[R0] (parity8)
      MVFR  R11             # R11 = p8
```

```
// Calculating p4
SUB   R0              # R0 = 0
ADDI  #31             # R0 = 31
ADDI  #31             # R0 = 62
ADDI  #31             # R0 = 93
ADDI  #31             # R0 = 124
ADDI  #18             # R0 = 142 (8'b1000_1110)
AND   R4              # R0 = R0 & R4
XOR   R3              # R0 = R0 ^ R3
XORR  R0              # R0 = ^[R0] (parity4)
MVFR  R12             # R12 = p4

// Calculating p2
SUB   R0              # R0 = 0
MVTO  R3              # R0 = R3
LSRI  #1              # R0 = R0 >> 1
MVFR  R5              # R5 = R0
SUB   R0              # R0 = 0
ADDI  #31             # R0 = 31
ADDI  #31             # R0 = 62
ADDI  #31             # R0 = 93
ADDI  #16             # R0 = 109 (8'b0110_1101)
AND   R4              # R0 = R0 & R4
XOR   R5              # R0 = R0 ^ R5
XORR  R0              # R0 = ^[R0] (parity2)
MVFR  R13             # R13 = p2

// Calculating p1
SUB   R0              # R0 = 0
ADDI  #5              # R0 = 5 (8'b0000_0101)
AND   R3              # R0 = R0 & R3
```

```
MVFR R5              # R5 = R0
SUB  R0              # R0 = 0
ADDI #31             # R0 = 31
ADDI #31             # R0 = 62
ADDI #29             # R0 = 91 (8'b0101_1011)
AND  R4              # R0 = R0 & R4
XOR  R5              # R0 = R0 ^ R5
XORR R0              # R0 = ^[R0] (parity1)
MVFR R14             # R14 = p1
```

**// Calculating p0**

```
MVTO R3              # R0 = R3 (MSW)
XOR  R4              # R0 = R0 ^ R4
XORR R0              # R0 = ^[R0]
MVFR R5              # R5 = R0
MVTO R14             # R0 = R14 (p1)
XOR  R13             # R0 = R0 ^ R13 (p2)
XOR  R12             # R0 = R0 ^ R12 (p4)
XOR  R11             # R0 = R0 ^ R11 (p8)
XOR  R5              # R0 = R0 ^ R5 (parity0)
MVFR R15             # R15 = p0
```

**// Formatting output MSW**

```
MVTO R3              # R0 = R3 (MSW)
LSLI #5              # R0 = R3 << 5
MVFR R5              # R5 = R0
MVTO R4              # R0 = R4 (LSW)
LSRI #4              # R0 = R0 >> 4
LSLI #1              # R0 = R0 << 1
OR   R11             # R0 = R0 | R11 (parity8)
OR   R5              # R0 = R0 | R5 (output MSW)
```

```
MVFR  R5              # R5 = output MSW
SUB   R0              # R0 = 0
ADDI  #31             # R0 = 31
ADD   R2              # R0 = 31 + i
MVFR  R6              # R6 = R0
MVTO  R5              # R0 = R5
STR   R6              # Mem[R6] = R0
```

**// Formatting output LSW**

```
SUB   R0              # R0 = 0
ADDI  #14             # R0 = 14 (8'b0000_1110)
AND   R4              # R0 = R0 & R4 (LSW)
LSLI  #4              # R0 = R0 << 4
MVFR  R5              # R5 = R0
SUB   R0              # R0 = 0
ADDI  #1              # R0 = 1 (8'b0000_0001)
AND   R4              # R0 = R0 & R4 (LSW)
LSLI  #3              # R0 = R0 << 3
OR    R5              # R0 = R0 | R5 (message bits in correct position)
MVFR  R5              # R5 = R0
MVTO  R14             # R0 = R14
LSLI  #1              # R0 = R0 << 1
MVFR  R14             # R14 = R0
MVTO  R13             # R0 = R13
LSLI  #2              # R0 = R0 << 2
MVFR  R13             # R13 = R0
MVTO  R12             # R0 = R12
LSLI #4               # R0 = R0 << 4
MVFR  R12             # R12 = R0
MVTO  R5              # R0 = R5
OR    R12             # R0 = R0 | R12
OR    R13             # R0 = R0 | R13
```

```
OR    R14             # R0 = R0 | R14
OR    R15             # R0 = R0 | R15
MVFR  R5              # R5 = R0
SUB   R0              # R0 = 0
ADDI  #30             # R0 = 30
ADD   R2              # R0 = 30 + i
MVFR  R6              # R6 = R0
MVTO  R5              # R0 = R5
STR   R6              # Mem[R6] = R0

// Handle looping
LUT   #0              # R0 = LUT[0]; holds the address of Loop label
MVFR  R1              # R1 = R0; branch reg has address of Loop label
SUB   R0              # R0 = 0
ADDI  #30             # R0 = 30
MVFR  R5              # R5 = R0
MVTO  R2              # R0 = R2
ADDI  #2              # R0 = R0 + 2
MVFR  R2              # R2 = R0
SLT   R5              # R0 = R0 < R5 (1 or 0)
BTRU                  # If R0 = 1; branch to address specified in branch reg (Loop)
HALT
```

## Program 2 Pseudocode

```
// will store parity bit "correctness"; 8'b0000_p8p4p2p1
parity_tracker = mem[60]

// Looping through the encoded data values
for (int i = 0; i < 30; i+2) {
```

```
val_p0 = 0
p_counter = 0
int p1 = 0;
int p2 = 0;
int p4 = 0;
int p8 = 0;

encoded_MSW = Mem[30+i]
encoded_LSW = Mem[31+i]

// calculate the correctness of each parity bit in the encoded message

// checking p0
MSW_p0 = ^(encoded_MSW[7:0])
LSW_p0 = ^(encoded_LSW[7:0])
p0 = MSW_p0 ^ LSW_p0
if (p0 == 0) {
     val_p0 = 0
} else {
     // this means that p0 is wrong
     val_p0 = 1
}

// checking p1

// xor bits in positions 7, 5, 3, and 1 together
MSW_p1 = ^(encoded_MSW[7;5;3;1])

// xor bits in positions 7, 5, 3, 1 together
LSW_p1 = ^(encoded_LSW[7;5;3;1])

val_p1 = MSW_p1 ^ LSW_p1
```

```
if (val_p1 != 0) {
     parity_tracker[;0] = 1          // 8'b0000_0001
     p1 = 1
     counter++
}

// checking p2

MSW_p2 = ^(encoded_MSW[7;6;3;2])
LSW_p2 = ^(encoded_LSW[7;6;3;2])
val_p2 = MSW_p2 ^ LSW_p2

if (val_p2 != 0) {
     parity_tracker[;1] = 1          // 8'b0000_0010
     p2 = 1
     counter++
}

// checking p4

MSW_p4 = ^(encoded_MSW[7:4])
LSW_p4 = ^(encoded_LSW[7:4])
val_p4 = MSW_p4 ^ LSW_p4


if (val_p4 != 0) {
     parity_tracker[;2] = 1          // 8'b0000_0100
     p4 = 1
     counter++
}

// checking p8
```

```
MSW_p8 = ^(encoded[7:0])
val_p8 = ^(MSW_p8)

if (val_p8 != 0) {
     parity_tracker[;3] = 1          // 8'b0000_1000
     p8 = 1
     counter++
}

// check each error scenario
if (val_p0 == 1 and counter == 0){
     // single bit error in p0 itself; the d's represent the incoming message
     Mem[i+1:i] = 16'b0100_0ddd_dddd_dddd
     continue
}

if (val_p0 == 0 and counter == 0) {
     // no errors
     Mem[i+1:i] = 16'b0000_0ddd_dddd_dddd
     continue
}


if (val_p0 == 1 and counter == 1) {
     Mem[i+1:i] = 16'b0100_0ddd_dddd_dddd
     continue
}

if (val_p0 == 1 and counter >= 2) {
     // one data error, we must correct
     if (parity_tracker < 8) {
```

```
                // error is in LSW
                encoded_LSW[;parity_tracker] = !encoded_LSW[;parity_tracker]
                Mem[i] = encoded_LSW
                Mem[i+1] = encoded_MSW
          } else {
                // error is in MSW
                parity_tracker = parity_tracker - 8
                encoded_MSW[;parity_tracker] = !encoded_MSW[;parity_tracker]
                Mem[i] = encoded_LSW
                Mem[i+1] = encoded_MSW
          }
     }

     if (val_p0 == 0 and counter >= 1) {
          // found 2 errors, cannot correct original data
          Mem[i+1:i] = 16'b1000_0000_0000_0000
     }
}
```

## Program 2 Assembly Code

```
Note: R0 accumulator register; R1 branch register; R2-R15 General Purpose
LUT[1] = &[Loop]
LUT[2] = &[handle_loop]
LUT[3] = &[p0_correct]
LUT[4] = &[all_correct]
LUT[5] = &[one_or_none_wrong]
LUT[6] = &[err_MSW]
```

```
R2 = looping variable (i)
R3 = MSW_encoded
R4 = LSW_encoded
R13 = p0
R14 = 8'b0000_p8p4p2p1 (tracks error bit)
R15 = counter (stores number of incorrect parity bits - excluding p0)
```

```
      // Loading registers with encoded MSW and LSW
Loop: SUB  R0              # R0 = 0
      MVFR R14             # R14 = 0
      MVFR R15             # R15 = 0
      ADDI #31             # R0 = 30
      ADD  R2              # R0 = 30 + i
      LOAD R0              # R0 = Mem[30 + i]; MSW_encoded
      MVFR R3              # R3 = R0
      SUB  R0              # R0 = 0
      ADDI #30             # R0 = 31
      ADD  R2              # R0 = 31 + i
      LOAD R0              # R0 = Mem[31 + i]; LSW_encoded
      MVFR R4              # R4 = R0

      // Verify correctness of parity0
      MVTO R3              # R0 = R3; MSW_encoded
      XOR  R4              # R0 = R0 ^ R4; MSW_encoded[7:0] ^ LSW_encoded[7:0]
      XORR R0              # R0 = ^[R0]
      MVFR R5              # R5  = R0
      SUB  R0              # R0 = 0
      ADDI #1              # R0 = 1
```

```
SEQ  R5              # R0 = (R0 == R5)
MVFR R13             # R13 = R0

// Verify correctness of parity1
SUB  R0              # R0 = 0
ADDI #31             #
ADDI #31             #
ADDI #31             #
ADDI #31             #
ADDI #31             #
ADDI #15             # R0 = 170 (8'b1010_1010)
MVFR R5              # R5 = R0
AND  R3              # R0 = R0 & R3
MVFR R6              # R6 = R0 (MSW_encoded[7;5;3;1])
MVTO R5              # R0 = R5
AND  R4              # R0 = R0 & R4 (LSW_encoded[7;5;3;1])
XOR  R6              # R0 = R0 ^ R6
XORR R0              # R0 = ^[R0]
MVFR R5              # R5 = R0
SUB  R0              # R0 = 0
ADDI #1              # R0 = 1
SEQ  R5              # R0 = (R0 == R5)
MVFR R14             # R14 = R0
MVFR R15             # R15 = R0

// Verifying correctness of parity2
SUB  R0              # R0 = 0
ADDI #31             #
ADDI #31             #
ADDI #31             #
ADDI #31             #
ADDI #31             #
```

```
ADDI  #31             #
ADDI  #18             # R0 = 204 (8'b1100_1100)
MVFR  R5              # R5 = R0
AND   R3              # R0 = R0 & R3
MVFR  R6              # R6 = R0 (MSW_encoded[7;6;3;2])
MVTO  R5              # R0 = R5
AND   R4              # R0 = R0 & R4 (LSW_encoded[7;6;3;2])
XOR   R6              # R0 = R0 ^ R6
XORR  R0              # R0 = ^[R0]
MVFR  R5              # R5 = R0
SUB   R0              # R0 = 0
ADDI #1               # R0 = 1
SEQ   R5              # R0 = (R0 == R5)
MVFR  R6              # R6 = R0
LSLI  #1              # R0 = R0 << 1
OR    R14             # R0 = R0 | R14
MVFR  R14             # R14 = R0
MVTO  R15             # R0 = R15
ADD   R6              # R0 = R0 + R6
MVFR  R15             # R15 = R0
```

**// Verifying correctness of parity4**

```
MVTO  R3              # R0 = R3
LSRI  #4              # R0 = R0 >> 4
MVFR  R5              # R6 = R0 (MSW_encoded[7:4])
MVTO  R4              # R0 = R4 (LSW_encoded)
LSRI  #4              # R0 = R0 >> 4 (LSW_encoded[7:4])
XOR   R5              # R0 = R0 ^ R5
XORR  R0              # R0 = ^[R0]
MVFR  R5              # R5 = R0
SUB   R0              # R0 = 0
ADDI  #1              # R0 = 1
```

```
SEQ  R5              # R0 = (R0 == R5)
MVFR R6              # R6 = R0
LSLI #2              # R0 = R0 << 2
OR   R14             # R0 = R0 | R14
MVFR R14             # R14 = R0
MVTO R15             # R0 = R15
ADD  R6              # R0 = R0 + R6
MVFR R15             # R15 = R0
```

**// Verifying correctness of parity8**

```
MVTO R3              # R0 = R3
XORR R0              # R0 = ^[R0]
MVFR R5              # R5 = R0
SUB  R0              # R0 = 0
ADDI #1              # R0 = 1
SEQ  R5              # R0 = (R0 == R5)
MVFR R6              # R6 = R0
LSLI #3              # R0 = R0 << 3
OR   R14             # R0 = R0 | R14
MVFR R14             # R14 = R0
MVTO R15             # R0 = R15
ADD  R6              # R0 = R0 + R6
MVFR R15             # R15 = R0
```

**// First, let's load R5 and R6 to have the original data in the correct spots**

```
MVTO R3              # R0 = R3
LSRI #5              # R0 = R0 >> 5
MVFR R5              # R5 = R0 (8'b0000_0b11b10b09)
MVTO R3              # R0 = R3
LSRI #1              # R0 = R0 >> 1
```

```
LSLI  #4              # R0 = R0 << 4 (8'b8b7b6b5_0000)
MVFR  R6              # R6 = R0
MVTO  R4              # R0 = R4
LSRI  #5              # R0 = R0 >> 5
LSLI  #1              # R0 = R0 << 1
OR    R6              # R0 = R0 | R6 (8'b8b7b6b5_b4b3b20)
MVFR  R6              # R6 = R0
SUB   R0              # R0 = 0
ADDI  #8              # R0 = 8
AND   R4              # R0 = R0 & R4
LSRI  #3              # R0 = R0 >> 3
OR    R6              # R0 = R0 | R6 (8'b8b7b6b5_b4b3b2b1)
MVFR  R6              # R6 = R0

// Check each error scenario
LUT   #3              # R0 = LUT[3]
MVFR  R1              # R1 = R0
SUB   R0              # R0 = 0
SEQ   R13             # R0 = (R0 == R13)
BTRU                  # if p0 is correct, branch to p0_correct

// p0 is wrong
LUT   #5              # R0 = LUT[5]
MVFR  R1              # R1 = R0
SUB   R0              # R0 = 0
ADDI  #1              # R0 = 1
SEQ   R15             # R0 = (R0 == R15)
BTRU                  # one other p is wrong, branch to one_or_none_wrong
SUB   R0              # R0 = 0
SEQ   R15             # R0 = (R0 == R15)
BTRU                  # if no other p is wrong; also branch to one_or_none_wrong
```

```
// p0 wrong and 2+ other p wrong
LUT   #6              # R0 = LUT[6]
MVFR  R1              # R1 = R0
SUB   R0              # R0 = 0
ADDI  #7              # R0 = 7
SLT   R14             # R0 = (7 < R14)
BTRU                  # if 7 < error_tracker; branch to err_MSW

// error in LSW
LUT   #2              # R0 = LUT[2]
MVFR  R1              # R1 = R0
MVTO  R4              # R0 = R4 (LSW_encoded)
NOT   R14             # R0 = R0[;~R14] (flip the bit specified by R14 in R0)
MVFR  R10             # R10 = R0
SUB   R0              # R0 = 0
ADDI  #1              # R0 = 1
LSLI  #6              # R0 = R0 << 6 (8'b0100_0000)
OR    R5              # R0 = R0 | R5
MVFR  R7
MVTO  R2
ADDI  #1
MVFR  R8
MVTO  R7
STR   R8              # Mem[i] = R0
MVTO  R6              # R0 = R6
LSRI  #4              # R0 = R0 >> 4
LSLI  #4              # R0 = R0 << 4
MVFR  R9              # R9 = R0
MVTO  R10             # R0 = R10
LSRI  #5              # R0 = R0 >> 5
LSLI  #1              # R0 = R0 << 1
OR    R9              # R0 = R0 | R9
```

```
MVFR  R9              # R9 = R0
MVTO  R10             # R0 = R10
LSLI  #4              # R0 = R0 << 4
LSRI  #7              # R0 = R0 >> 7
OR    R9              # R0 = R0 | R9
STR   R2              # Mem[i + 1] = R0
B                     # branch to handle_loop
```

**// error in MSW**

```
err_MSW:
      LUT   #2        # R0 = LUT[2]
      MVFR  R1        # R1 = R0
      MVTO  R14       # R0 = R14
      SUBI  #8        # R0 = R0 - 8
      MVFR  R14       # R14 = R0
      MVTO  R3        # R0 = R3
      NOT   R14       # R0 = R0[;~R14] (flip the bit specified by R14 in R0)
      MVFR  R3        # R3 = R0
      LSRI  #5        # R0 = R0 >> 5
      MVFR  R5        # R5 = R0
      SUB   R0        # R0 = 0
      ADDI  #1        # R0 = 1
      LSLI  #6        # R0 = R0 << 6
      OR    R5        # R0 = R0 | R5
      MVFR  R7
      MVTO  R2
      ADDI  #1
      MVFR  R8
      MVTO  R7
      STR   R8        # Mem[i] = R0
      MVTO  R3        # R0 = R3
      LSRI  #1        # R0 = R0 >> 1
```

```
      LSLI  #4          # R0 = R0 << 4
      MVFR  R7          # R7 = R0
      MVTO  R6          # R0 = R6
      LSLI  #4          # R0 = R0 << 4
      LSRI  #4          # R0 = R0 >> 4
      OR    R7          # R0 = R0 | R7
      STR   R2
      B                 # branch to handle_loop
```

**// p0 wrong and one other p is wrong OR no other p is wrong**

```
one_or_none_wrong:
      LUT   #2          # R0 = LUT[2]
      MVFR  R1          # R1 = R0
      SUB   R0          # R0 = 0
      ADDI  #1          # R0 = 1
      LSLI  #6          # R0 = (8'b0100_0000)
      OR    R5          # R0 = (8'b0100_0ddd; d = original data)
      MVFR  R7
      MVTO  R2
      ADDI  #1
      MVFR  R8
      MVTO  R7
      STR   R8
      MVTO  R6          # R0 = R6
      STR   R2          # Mem[i + 1] = R0
      B                 # branch to handle_loop
```

**// p0 is correct**

```
p0_correct:
      LUT   #4          # R0 = LUT[4]
      MVFR  R1          # R1 = R0
```

```
SUB   R0          # R0 = 0
SEQ   R15         # R0 = (R0 == R15)
BTRU              # all 5 parities are correct, branch to all_correct
```

**// one or more other parity look wrong, we store (10xx_xxxx_xxxx_xxxx)**

```
LUT   #2          # R0 = LUT[2]
MVFR  R1          # R1 = R0
SUB   R0          # R0 = 0
ADDI  #1          # R0 = 1
LSLI  #7          # R0 = R0 << 7 (8'b1000_0000)
MVFR  R7
MVTO  R2
ADDI  #1
MVFR  R8
MVTO  R7
STR   R8          # Mem[i] = R0
                  # We don't care about what gets stored in Mem[i + 1] in this case

B                 # branch to handle_loop
```

**// all parities are correct, we store (0000_0ddd_dddd_dddd; d = original data)**

```
all_correct:
      SUB   R0    # R0 = 0
      ADDI  #1    # R0 = 1
      ADD   R2    # R0 = R0 + R2
      MVFR  R7    # R7 = R0
      MVTO  R6    # R0 = R5
      STR   R2    # Mem[i] = R0
      MVTO  R5    # R0 = R6
      STR   R7    # Mem[i + 1] = R0
```

**// handle looping**

```
handle_loop:
      LUT   #1          # R0 = LUT[1]
      MVFR  R1          # R1 = R0
      SUB   R0          # R0 = 0
      ADDI  #30         # R0 = 30
      MVFR  R8          # R8 = R0
      MVTO  R2          # R0 = R2
      ADDI  #2          # R0 = R0 + 2
      MVFR  R2          # R2 = R0
      SLT   R8          # R0 = (R0 < R8)
      BTRU              # branch to Loop if i < 30
      HALT
```

## Program 3 Pseudocode

```
//Lines 4-17 are working with ith byte only. checking if the pattern exists in the ith byte,
and //if it does, how many times it exists. So for that, it starts by comparing the least
significant //5 bits and shifts and compares till it reaches the most significant 5 bits.
That's what the for //loop on line 6 is doing. Shifting by 0,1,2 and 3 in different iterations.
For c we are doing the //same thing but checking 2 bytes at a time, and shifting upto 7 times

unsigned char pattern = mem[32] >> 3
  mask = 8'b00011111;
for(i=0 to i=32){
```

```
 unsigned char byte = mem[i]
 occurrences = 0;
 for(j= 0 to j=3){
   shift_pattern = left shift pattern by 'j'
   xor_out = shift_pattern ^ byte
   shift_mask = left shift mask by 'j'
   if((shift_mask & xor_out)== 0)
     occurrences++;
 }

 if(occurrences > 0){
   increment mem[34]
 }
 mem[33] = mem[33] + occurrences


 unsigned short twobytes=bytes[i]
   if (i!=31)
      twobytes= (bytes[i+1] << 8) | bytes[i]
// copy bytes from index 'i' and 'i+1' into twobytes

 occurrences=0

 unsigned short pattern2=pattern;

 for(i=0 to j=7){
   shift_pattern = left shift pattern2 by 'j'
   xor_out = shift_pattern ^ two byte
   shift_mask = left shift mask by 'j'
   if((shift_mask & xor_out)== 0)
     occurrences++;
 }
```

```
  mem[35] = mem[35] + occurrences;
}
```

## Program 3 Assembly Code

```
Note: R0 accumulator register; R1 branch register; R2-R15 General Purpose
R2 = i; looping variable
R3 = pattern
R4 = respect boundaries count
R5 = num bytes within which pattern occurs
R6 = count in between boundaries (bit[3:0] of first string and bit[7:4] of next string)
R7 = Mem[i]

LUT[7] = &[Loop]
LUT[8] = &[last_entry]


ADDI  #31                    # R0 = 31
ADDI  #1                     # R0 = 32
LOAD  R0                     # R0 = Mem[32]
LSRI  #3
LSLI  #3
MVFR  R3                     # R3 = R0; pattern

Loop:
      // Load registers with message byte at Mem[i] with correct shifts
      Notes:
            R7 = Mem[i] = b7 b6 b5 b4 _ b3 b2 b1 b0

            R8    =     b7 b6 b5 b4 _ b3 0 0 0
```

```
      R9    =     b6 b5 b4 b3 _ b2 0 0 0
      R10   =     b5 b4 b3 b2 _ b1 0 0 0
      R11   =     b4 b3 b2 b1 _ b0 0 0 0

      R15   =     count within byte

SUB   R0                  # R0 = 0
MVFR  R15                 # R15 = 0
LOAD  R2                  # R0 = Mem[i]
MVFR  R7                  # R7 = R0
LSRI  #3                  # R0 = R0 >> 3
LSLI  #3                  # R0 = R0 << 3
MVFR  R8                  # R8 = R0 (b7 b6 b5 b4 _ b3 0 0 0)
MVTO  R7                  # R0 = R7
LSRI  #2                  # R0 = R0 >> 2
LSLI  #3                  # R0 = R0 << 3
MVFR  R9                  # R9 = R0 (b6 b5 b4 b3 _ b2 0 0 0)
MVTO  R7                  # R0 = R7
LSRI  #1                  # R0 = R0 >> 1
LSLI  #3                  # R0 = R0 << 3
MVFR  R10                 # R10 = R0
MVTO  R7                  # R0 = R7
LSLI  #3                  # R0 = R0 << 3
MVFR  R11                 # R11 = R0

// checking R8
MVTO  R8                  # R0 = R8
SEQ   R3                  # R0 = (R0 == R3) checking pattern occurrence
MVFR  R12                 # R12 = 1 or 0 if pattern occurred
ADD   R15                 # R0 = R0 + R15
MVFR  R15                 # R15 = R0
MVTO  R12                 # R0 = R12
```

```
ADD  R4              # R0 = R0 + R4
MVFR R4              # R4 = R0

// checking R9
MVTO R9              # R0 = R9
SEQ  R3              # R0 = (R0 == R3) checking pattern occurrence
MVFR R12             # R12 = 1 or 0 if pattern occurred
ADD  R15             # R0 = R0 + R15
MVFR R15             # R15 = R0
MVTO R12             # R0 = R12
ADD  R4              # R0 = R0 + R4
MVFR R4              # R4 = R0

// checking R10
MVTO R10             # R0 = R10
SEQ  R3              # R0 = (R0 == R3) checking pattern occurrence
MVFR R12             # R12 = 1 or 0 if pattern occurred
ADD  R15             # R0 = R0 + R15
MVFR R15             # R15 = R0
MVTO R12             # R0 = R12
ADD  R4              # R0 = R0 + R4
MVFR R4              # R4 = R0

// checking R11
MVTO R11             # R0 = R11
SEQ  R3              # R0 = (R0 == R3) checking pattern occurrence
MVFR R12             # R12 = 1 or 0 if pattern occurred
ADD  R15             # R0 = R0 + R15
MVFR R15             # R15 = R0
MVTO R12             # R0 = R12
ADD  R4              # R0 = R0 + R4
MVFR R4              # R4 = R0
```

```
// check if the pattern occurs within the byte, if yes, increment R5
SUB   R0              # R0 = 0
MVFR  R14             # R14 = 0
SEQ   R15             # R0 = (R0 == R15)
SEQ   R14             # R0 = (R0 == R14)
ADD   R5              # R0 = R0 + R5
MVFR  R5              # R5 = R0

// Check in between boundaries
Note: DO NOT TOUCH R4-R7; recall R7 = Mem[i]

General Setup:
     Mem[i+1:i] = b7 b6 b5 b4 _ b3 b2 b1 b0 b7 b6 b5 b4 _ b3 b2 b1 b0


// Check for Mem[31] because Mem[31] is the last message with no adjacent byte in Mem[32]
LUT   #8              # R0 = LUT[8]
MVFR  R1              # R1 = R0
SUB   R0              # R0 = 0
ADDI  #31             # R0 = 31
SEQ   R2              # R0 = (R0 == 31)
BTRU                  # if (i = 31), branch to last_entry

// Checking between boundaries Mem[0:30]
// R9       =     b3 b2 b1 b0 b7 0 0 0
// R10      =     b2 b1 b0 b7 b6 0 0 0
// R11      =     b1 b0 b7 b6 b5 0 0 0
// R12      =     b0 b7 b6 b5 b4 0 0 0

MVTO  R2              # R0 = R2
ADDI  #1              # R0 = R0 + 1
```

```
LOAD  R0              # R0 = Mem[i + 1]
MVFR  R8              # R8 = R0

// Case 1
// Mem[i+1:i] = b7 b6 b5 b4 _ b3 b2 b1 b0 b7 b6 b5 b4 _ b3 b2 b1 b0
MVTO  R7              # R0 = R7
LSLI  #4              # R0 = R0 << 4
MVFR  R9              # R9 = R0
MVTO  R8              # R0 = R8
LSRI  #7              # R0 = R0 >> 7
LSLI #3               # R0 = R0 << 3
OR    R9              # R0 = R0 | R9
MVFR  R9              # R9 = R0

// Case 2
// Mem[i+1:i] = b7 b6 b5 b4 _ b3 b2 b1 b0 b7 b6 b5 b4 _ b3 b2 b1 b0
MVTO  R7              # R0 = R7
LSLI  #5              # R0 = R0 << 5
MVFR  R10             # R10 = R0
MVTO  R8              # R0 = R8
LSRI  #6              # R0 = R0 >> 6
LSLI #3               # R0 = R0 << 3
OR    R10             # R0 = R0 | R10
MVFR  R10             # R10 = R0

// Case 3
// Mem[i+1:i] = b7 b6 b5 b4 _ b3 b2 b1 b0 b7 b6 b5 b4 _ b3 b2 b1 b0
MVTO  R7              # R0 = R7
LSLI  #6              # R0 = R0 << 6
MVFR  R11             # R11 = R0
MVTO  R8              # R0 = R8
LSRI  #5              # R0 = R0 >> 5
```

```
LSLI #3                 # R0 = R0 << 3
OR   R11                # R0 = R0 | R11
MVFR R11                # R11 = R0
```

**// Case 4**

// Mem[i+1:i] = b7 b6 b5 b4 _ b3 b2 b1 **b0** **b7 b6 b5 b4** _ b3 b2 b1 b0

```
MVTO R7                 # R0 = R7
LSLI #7                 # R0 = R0 << 7
MVFR R12                # R12 = R0
MVTO R8                 # R0 = R8
LSRI #4                 # R0 = R0 >> 4
LSLI #3                 # R0 = R0 << 3
OR   R12                # R0 = R0 | R12
MVFR R12                # R12 = R0
```

**// Now, let's check each case against the pattern, recall R6 = count between boundaries**

```
MVTO R9                 # R0 = R9
SEQ  R3                 # R0 = (R0 == R3)
ADD  R6                 # R0 = R0 + R6
MVFR R6                 # R6 = R0
MVTO R10                # R0 = R10
SEQ  R3                 # R0 = (R0 == R3)
ADD  R6                 # R0 = R0 + R6
MVFR R6                 # R6 = R0
MVTO R11                # R0 = R11
SEQ  R3                 # R0 = (R0 == R3)
ADD  R6                 # R0 = R0 + R6
MVFR R6                 # R6 = R0
MVTO R12                # R0 = R12
SEQ  R3                 # R0 = (R0 == R3)
ADD  R6                 # R0 = R0 + R6
```

```
MVFR R6              # R6 = R0

// Handle the loop
LUT  #7              # R0 = LUT[7]
MVFR R1              # R1 = R0
SUB  R0              # R0 = 0
ADDI #31             # R0 = 31
ADDI #1              # R0 = 32
MVFR R8              # R8 = R0
MVTO R2              # R0 = R2
ADDI #1              # R0 = R0 + 1
MVFR R2
SLT  R8              # R0 = (R0 < R8)
BTRU                 # branch to Loop if i < 32

// Mem[31]; no adjacent message in Mem[32]

last_entry:
     // Now, we just store the registers into memory
     SUB  R0         # R0 = 0
     ADDI #31        # R0 = 31
     ADDI #2         # R0 = 33
     MVFR R8         # R8 = 33
     MVTO R4         # R0 = R4
     STR  R8         # Mem[33] = R0
     MVTO R8         # R0 = R8
     ADDI #1         # R0 = R0 + 1 (34)
     MVFR R8         # R8 = 34
     MVTO R5         # R0 = R5
     STR  R8         # Mem[34] = R0
     MVTO R8         # R0 = R8
     ADDI #1         # R0 = R0 + 1 (35)
```

```
MVFR  R8          # R8 = R0
MVTO  R4          # R0 = R4
ADD   R6          # R0 = R0 + R6
STR   R8          # Mem[35] = R0
HALT
```

# 8. Changelog

- **Milestone 1:**
  - Initial version
  - Created barebones layout of ISA
  - Wrote introduction, rough whiteboard sketch of architecture, specified instruction formats and operations
  - Pseudocode and assembly code for Programs 1 and 2
- **Milestone 2:**
  - Got rid of original reg-to-reg ISA design with operand and dest registers and switched to accumulator register implementation
  - Redefined instruction formats and operations/bit breakdowns
  - Rewrote assembly code for Program 1 and Program 2 to fit new accumulator design
  - Wrote the assembly code for Program 3
- **Milestone 3:**
  - Designed python assembler to translate assembly code to machine code
  - Formatted the assembly code to be readable by our assembler
    - Deleted comments, removed blank lines in between assembly instructions
  - Obtained all machine code text files for each program
  - Did some testing with our ISA
    - Realized that we needed to make a HALT instruction
    - Added unconditional and conditional branch signals
    - There were some issues we were facing with our program counter not incrementing at the correct time/not incrementing at all!

            - Introduced a start_ready boolean in the ProgCtr.sv that would indicate when the ProgCtr is ready to be incremented
        - Issues with ProgCtr not incrementing when a branch signal was raised → FIXED
    - Determined what lines labels were on for branch targets and manually populated LUT to be preloaded with the targets
    - Created usage.txt file for assembler
- **Milestone 4:**
    - Did significant testing and frequently went to Weijia's office hours for guidance (Darryl)
    - There were some bugs with Program 2 with incorrect masks → FIXED
    - Issue with TopLevel_tb, simulation stopping prematurely in Questa → FIXED
        - Ack signal being raised earlier than expected
    - Once everything was cleaned up, did a final compile on Quartus and copied over the schematics into the final report
    - Pasted TopLevel schematic along with the whiteboard, scuffed MS paint schematic
    - Created README.txt file