CSE 141L Final Report

Junhua (Michael) Ma, A16299193; Juhak Lee, A16056117

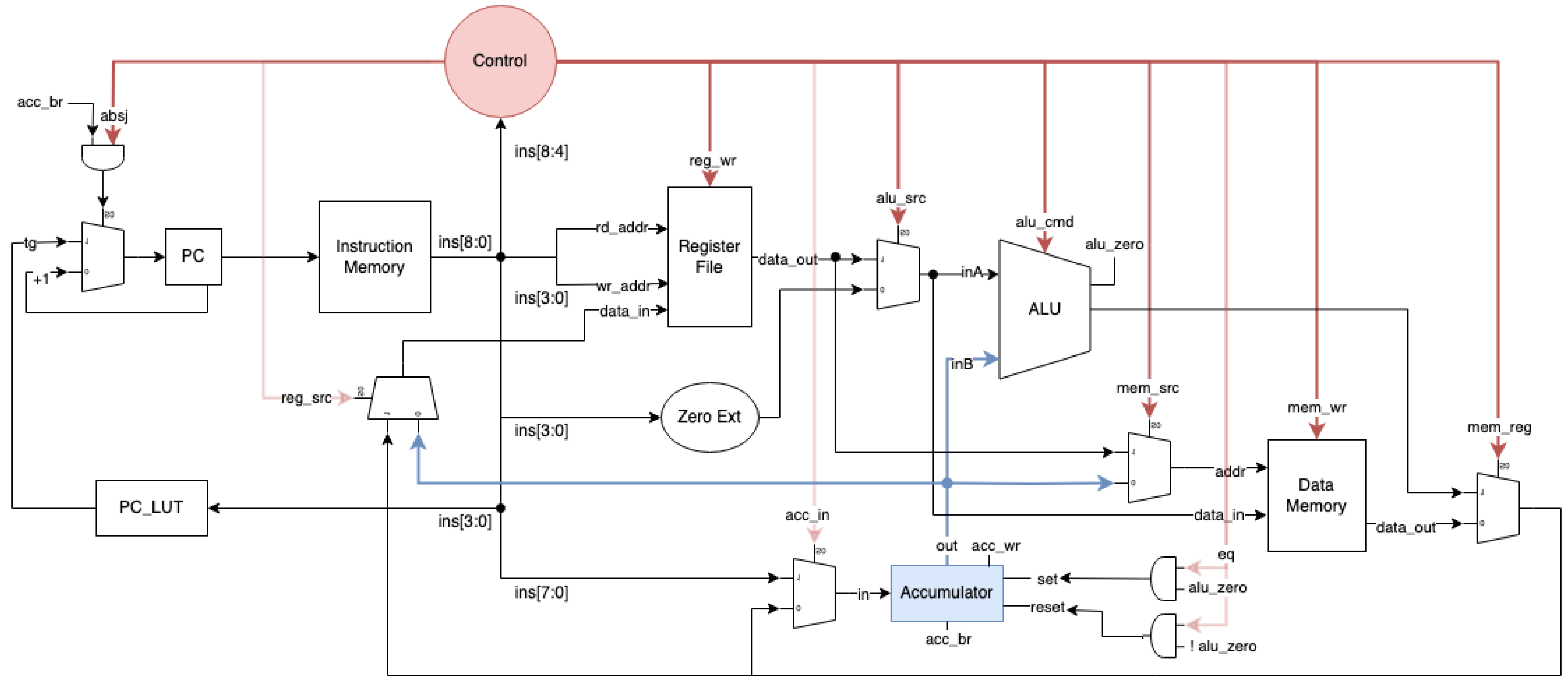
Team

Junhua (Michael) Ma
Juhak Lee

Introduction

The architecture is named MIHAK. It aims to achieve this by having a larger set of operations (4 bits for opcode) to make writing assembly code more straightforward and efficient. Additionally, MIHAK includes more registers to simplify assembly coding further and reduce the total number of instructions needed. This architecture can be classified as an "Accumulator" type of machine, which means it performs computations by using a single accumulator register.

Architectural Overview



Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	1 bit type, 4 bits opcode, 4 bits operand register	AND, ADD, PUT, etc
A	1 bit type, 1 bit funct, 7 bits address	SETI
I	1 bit type, 4 bits opcode, 4 bits immediate	ADDI, SUBI, etc

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
SETI	A	1 bit type (1), 8 bits immediate	`SETI 0001_0001` # after SETI instruction, accumulator now holds `0b0001_0001`	SETI X = Acc <= X X is signed: -128 to 127
PUT	R	1 bit type (0), 4 bits opcode (0000), 4 bits operand	# Assume accumulator has `0b0001_0001` `PUT R1` # after PUT instruction, R1 now holds `0b0001_0001`	PUT R1 = R1 <= Acc

EQ	R	1 bit type (0), 4 bits opcode (0010), 4 bits operand	`EQ R1` # if Acc = R1, then Acc = 1; otherwise, Acc = 0	For R1 = 1, EQ R1 flips
GET	R	1 bit type (0), 4 bits opcode (0011), 4 bits operand	# Assume R1 has `0b0001_0001` `GET R1` # after GET instruction, accumulator now holds `0b0001_0001`	GET R1 = Acc <= R1
ADDI	I	1 bit type (0), 4 bits opcode (0100), 4 bits imm	# Assume accumulator has 0 `ADDI 2` # after ADDI instruction, accumulator now holds 2	ADDI imm = Acc <= Acc + imm Imm unsigned: Range: 0 to 15
LOAD	R	1 bit type (0), 4 bits opcode (0101), 4 bits operand	# Assume R1 = 0 and mem[0] = 1 `LOAD R1` # after LOAD instruction, accumulator now holds 1	LOAD R1 = Acc <= mem[R1]
LOAD A	R	1 bit type (0), 4 bits opcode (0110), 4 bits operand	# Assume Acc = 0 and mem[0] = 1 `LOADA R1` # after LOAD instruction, R1 now holds 1	LOADA R1 = R1 <= mem[Acc]
SUBI	I	1 bit type (0), 4 bits opcode (0111), 4 bits imm	# Assume Acc = 5 `SUBI 1`	SUBI imm = Acc <= Acc - imm Imm unsigned: range 0 - 15

			# after SUBI instruction, accumulator now holds 4	
ADD	R	1 bit type (0), 4 bits opcode (1000), 4 bits operand	# Assume R1 = 1 and Acc = 1 `ADD R1` # after ADD instruction, accumulator now holds 2	ADD R1 = Acc <= Acc + R1
RS	R	1 bit type (0), 4 bits opcode (1001), 4 bits operand	# Assume R1 = 2 and Acc = `0b0001_0001` `RS R1` # after RS instruction, accumulator now holds `0b0000_0100`	RS R1 = Acc <= Acc >> R1 Sign-extended shift
AND	R	1 bit type (0), 4 bits opcode (1010), 4 bits operand	# Assume R1 = `0b0001_1000` and Acc = `0b0001_0001` `AND R1` # after RS instruction, accumulator now holds `0b0001_0000`	AND R1 = Acc <= Acc & R1
SAVE A	R	1 bit type (0), 4 bits opcode (1011), 4 bits operand	# Assume Acc = 0, R1 = 2, and mem[0] = 1 `SAVEA R1` # after SAVEA instruction, mem[0] now holds 2	SAVEA R1 = mem[Acc] = R1
LSI	I	1 bit type (0), 4 bits opcode (1100), 4 bits imm	# Assume Acc = `0b0001_0001`	LSI R1 = Acc <= Acc << imm Imm unsigned: range 0 - 15

			`LSI 2` # after RS instruction, accumulator now holds `0b0100_0100`	
XOR	R	1 bit type (0), 4 bits opcode (1101), 4 bits operand	# Assume R1 = `0b0001_0001` and Acc = `0b0000_0000` `XORS R1` # after XORS instruction, R1 now holds `0b0001_0001`	XOR R1 = Acc <= Acc ^ R1
OR	R	1 bit type (0), 4 bits opcode (1110), 4 bits operand	# Assume R1 = `0b0001_0001` and Acc = `0b0000_0000` `OR R1` # after XORS instruction, R1 now holds `0b0001_0001`	OR R1 = Acc <= Acc | R1
BZ	I	1 bit type (0), 4 bits opcode (1111), 4 bits imm	`BZ label` # if Acc = 0, go to label	Assign label to 4-bit value from 0 to 15, PC_LUT has mapping from label value to the actual PC address to jump to

Internal Operands

The MIHAK architecture supports 17 registers, including 16 normal general purpose registers and 1 special accumulator register.

Control Flow (branches)

The MIHAK architecture supports two forms of branch instructions, one form only allows branch to jump backward but offering a larger jump distance, while the other form allows the branch to jump both forward and backward but offering less jump distance. The maximum branch distance supported is 255 instructions. To accommodate large jumps, we used full 8-bit unsigned values to maximize the relative jump distance. When computing the target address, the value is simply added to the PC with ALU to derive the new PC value.

Addressing Modes

Our instruction supports input of full 7-bit memory address values, and our instruction can load from memory using the value of accumulator as address. So we can directly access 2^7 bytes of memory using indirect access where we can compute the desired memory address in the accumulator and then directly read or write to memory at the address specified. Example:

```
# Assume Acc = 0
ADD 3
SAVEA R3
# mem[3] = R3
```

We can also support indirect addressing, where we save the 8-bit memory address to access in the accumulator or a register, and then access memory at that address. This allows us to specify memory addresses up to 8 bits, which can cover all 256 memory positions in data memory. Example of our memory accessing instruction is as follows:

```
# Assume mem[200] = 1
SETI 200
LOADA R1
# R1 = 1
```

Programmer's Model [Lite]

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

For our accumulator architecture, the programmer should strive to utilize the registers as much as possible to improve the efficiency. Because many of our instructions are acted on the accumulator using values of registers, and the accumulator, especially in the case of a chain of operations, could continue on without disruption provided that the values are all within registers. If values in a computation are yet to be stored in register or provided halfway through a computation, then providing that value would cause the accumulator to lose its current value, and continuing the computation would require additional repetitive setups. So it’s most optimal to prepare the values in advance and maximize the utilization of registers so that the computation is most efficient. This is also due to the limitations of bits in instructions that prevent the possibility of directly providing numbers to the accumulator, while registers hae 8-bits that allows easy access of larger values.

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

We can’t copy MIPS or ARM because they are 32-bit architectures whereas we are limited to 9-bit instructions. To overcome this, we have to cut down the number of supported operations which makes programming assembly more difficult. We also limit the types of instructions to just 2 types. We also utilized the strengths of accumulator architecture to directly store and use 8-bit register values and make 1-operand instructions possible.

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

Based on our ISA, there are no non-arithmetic instructions that utilize the ALU.

Individual Component Specification

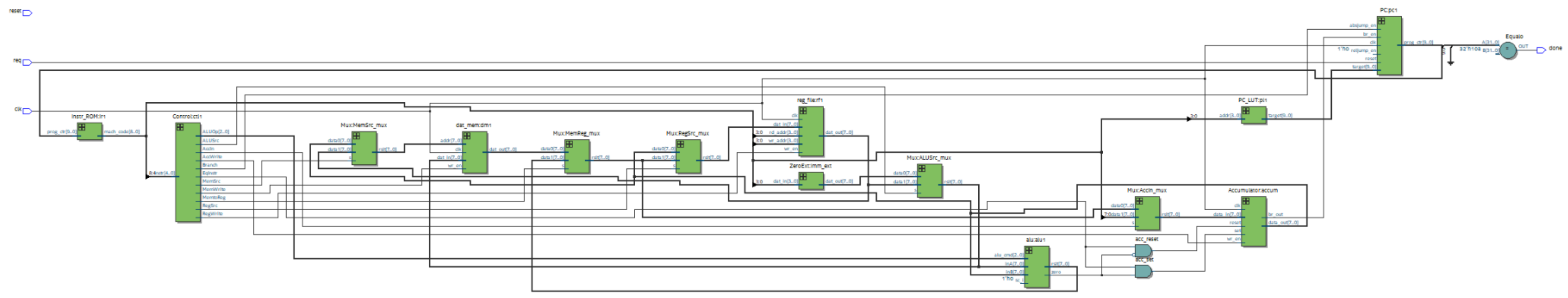
Top Level

Module file name: top_level.sv

Functionality Description

It combines all the modules to form the final processor, starting with using a PC to locate instructions in the instruction memory, to reading from a register file, running ALU, and writing to data memory. For our architecture, the accumulator module is used as a special purpose register that's connected directly to ALU and register file.

Schematic



Program Counter

Module file name: PC.sv
Module testbench file name: PC_tb.sv

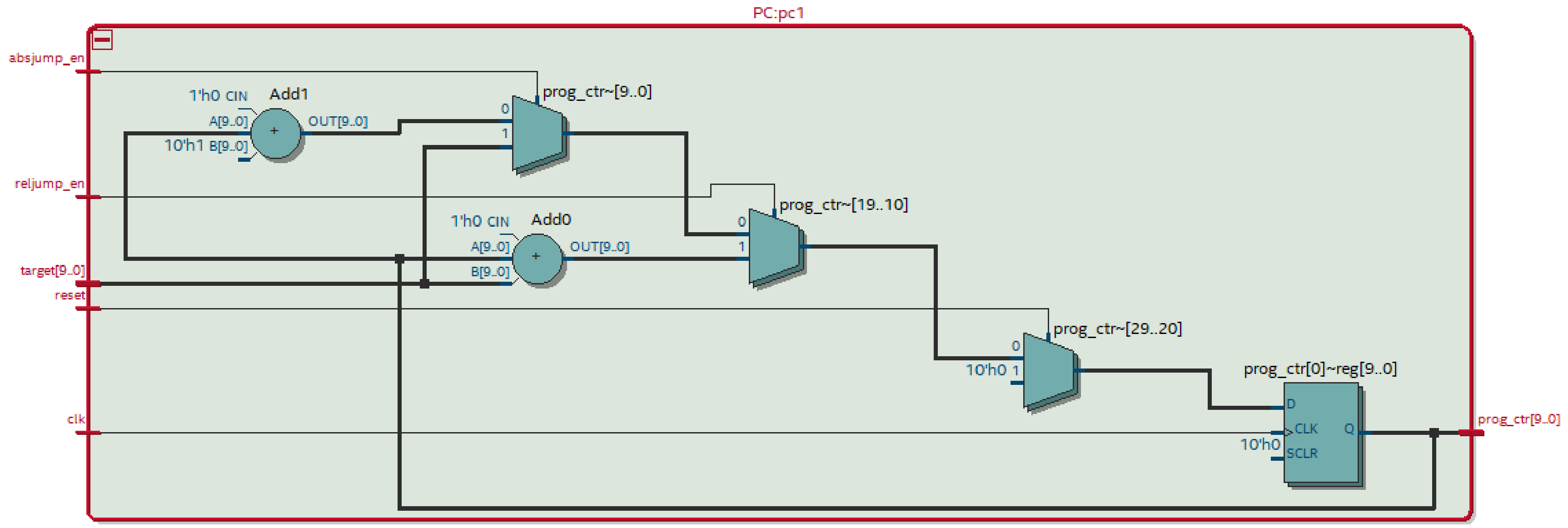
Functionality Description

This module keeps track of the address of instructions, so that the instructions in a program can be executed line by line. The program counter increments by 1 at every clock cycle, and has the option to perform relative or absolute jump. The target input to PC is used for either relative or absolute jump to determine the new PC after the jump. The PC can be reset to 0 with reset input.

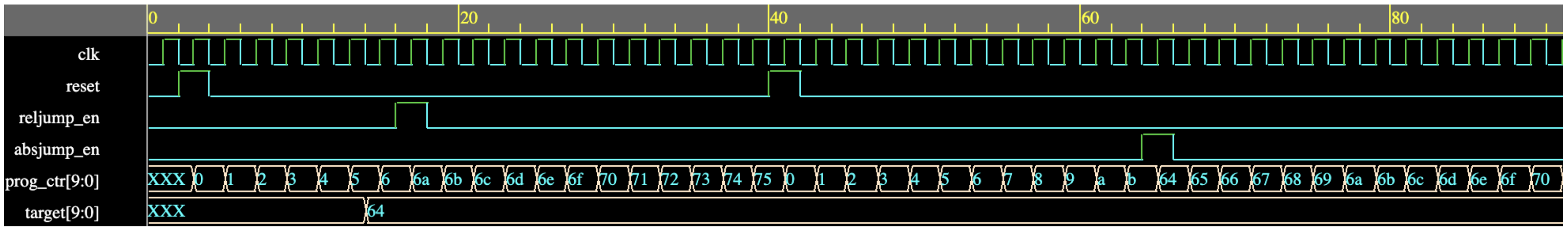
Testbench Description

The testbench tests that the PC can increment at each clock cycle, and can also perform both relative and absolute jumps correctly.

Schematic



Timing Diagram



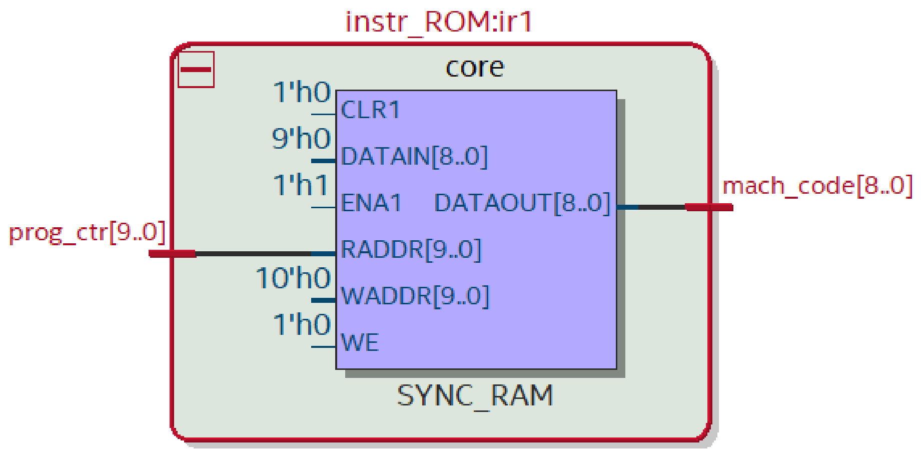
Instruction Memory

Module file name: instr_ROM.sv

Functionality Description

This module takes the PC value, which specifies the address of an instruction, and outputs the corresponding 9-bits instruction binary based on the given machine code instructions in mach_code.txt (converted from assembly).

Schematic



Control Decoder

Module file name: Control.sv

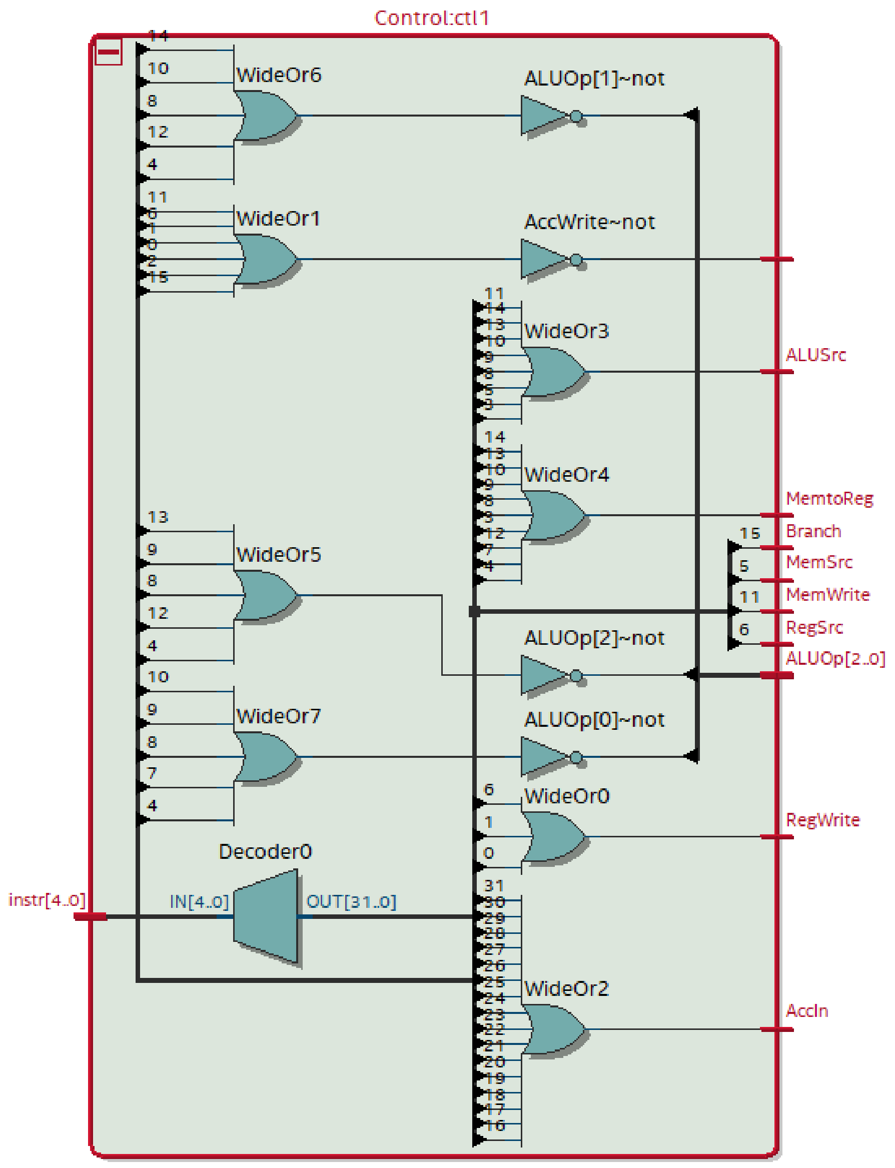
Functionality Description

This module is responsible for taking the opcode from each instruction and setting the control bits for other modules in the processor and ensuring the correct datapath for each instruction in our ISA. The table that maps instruction opcode to the exact control bit settings in our processor design is shown as follows:

Ins	Branch	MemWrite	RegWrite	AccWrite	ALUSrc	MemtoReg	RegSrc	AccIn	MemSrc	ALUop	EqInstr
SETI	0	0	0	1	X	X	X	1	X	X	0
PUT	0	0	1	0	X	X	0	X	X	X	0

EQ	0	0	0	0	1	X	X	X	X	110	1
GET	0	0	0	1	1	1	X.	0	X	111	0
ADDI SUBI LSI	0	0	0	1	0	1	X	0	X	000 110 001	0
LOAD	0	0	0	1	1	0	X	0	1	X	0
LOADA	0	0	1	0	X	0	1	X	0	X	0
ADD RS AND OR XOR	0	0	0	1	1	1	X	0	X	000 010 100 101 011	0
SAVEA	0	1	0	0	1	X	X	X	0	X	0
BZ	1	0	0	0	X	X	X	X	X	X	0

Schematic


Control:ctl1
WideOr6
ALUOp[1]~not
WideOr1
AccWrite~not
WideOr3
ALUSrc
WideOr4
MemtoReg
Branch
MemSrc
MemWrite
RegSrc
ALUOp[2..0]
WideOr5
ALUOp[2]~not
WideOr7
ALUOp[0]~not
WideOr0
RegWrite
Decoder0
instr[4..0]
IN[4..0]
OUT[31..0]
WideOr2
AccIn

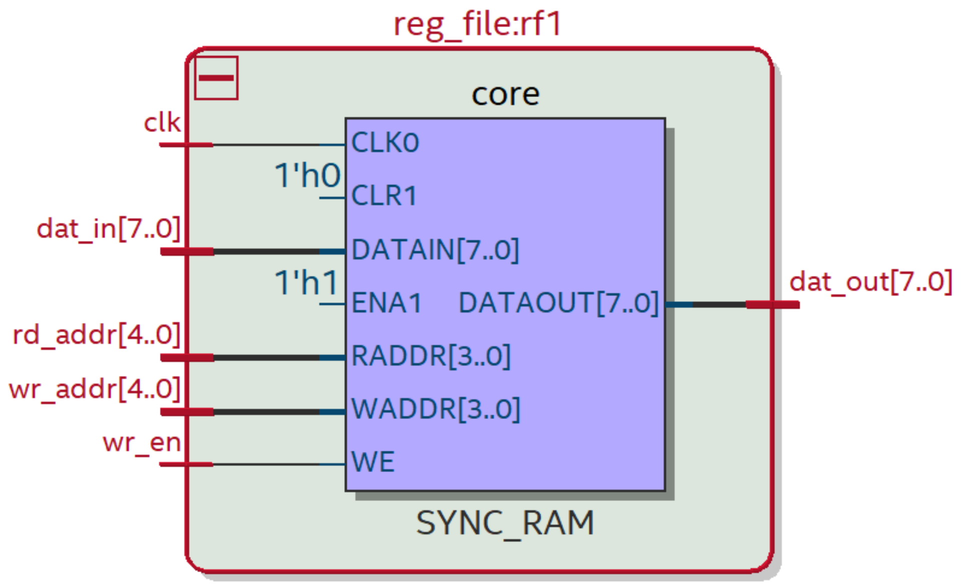
Register File

Module file name: reg_file.sv

Functionality Description

This module consists of 16 registers that can store and access values quickly. To read the value of a register, the rd_addr field specifies the register to read from, and dat_out is the value of that register. To modify the value of a register, the wr_addr and dat_in field specifies the register to modify and the new value it should take. The “wr_en” control bit must be true to modify register value once per clock cycle to protect the values stored in registers.

Schematic



ALU (Arithmetic Logic Unit)

Module file name: alu.sv

Module testbench file name: ALU_tb.sv

Functionality Description

This module performs all the key arithmetic operations for the processor. It’s fully combinational so it’s always computing values. The operation performed is based on the 3-bit “alu_cmd” opcode. With each operation, the ALU also sets flags based on the result. The zero flag is set to true if the result is 0, the parity flag is set to true if the number of 1’s in the result is odd, and so on.

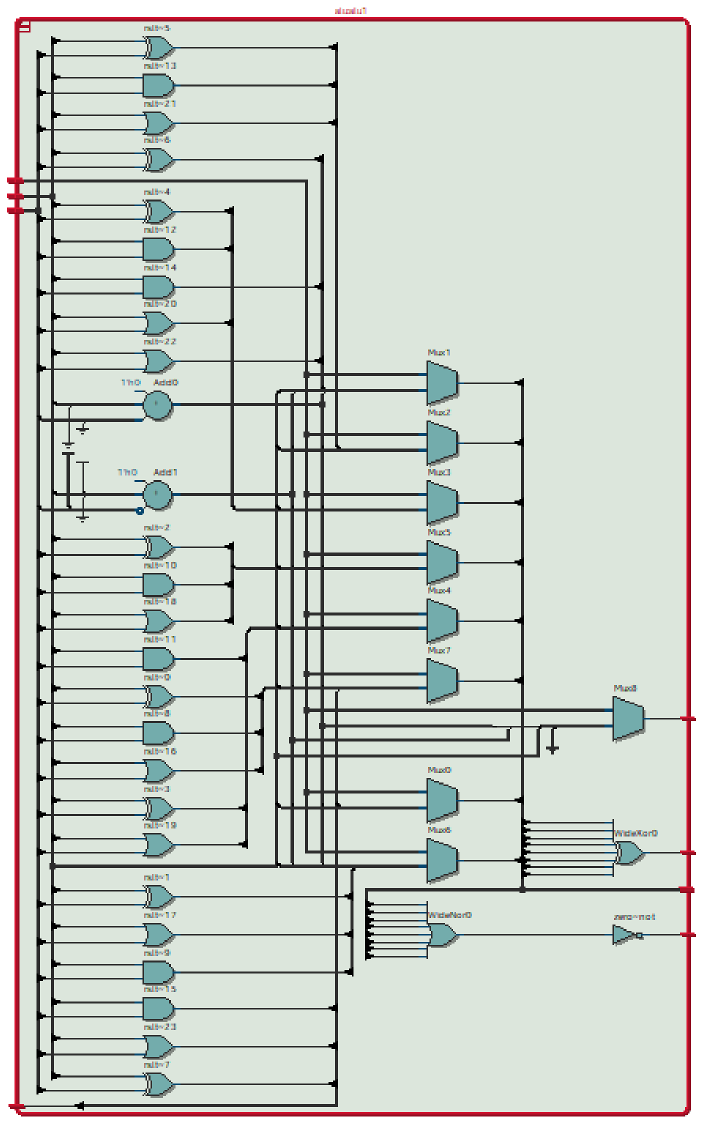
Testbench Description

The testbench tests that the ALU can correctly perform all the arithmetic operations needed for our ISA. Each operation is performed sequentially and the result is shown as waveform and checked.

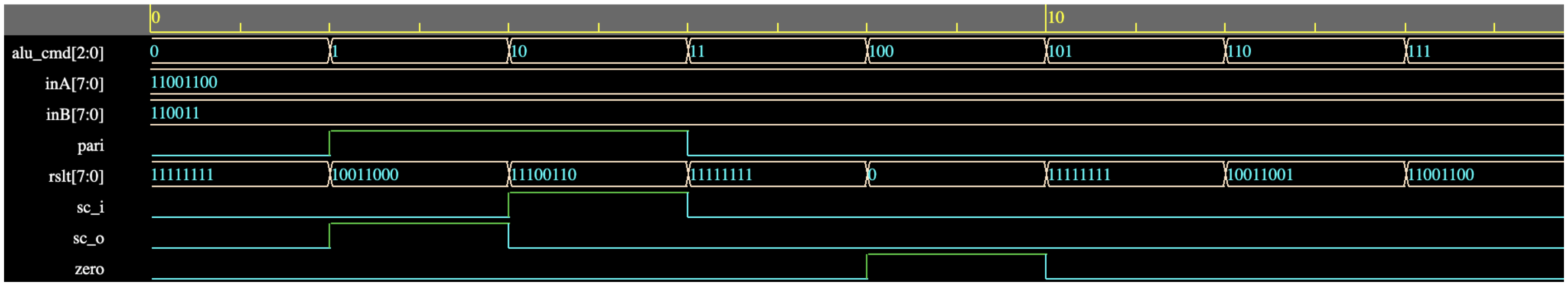
ALU Operations

Add, subtract, and, or, xor, left shift, right shift (sign extend), pass A

Schematic


alualu1
rslt~5
rslt~13
rslt~21
rslt~6
rslt~4
rslt~12
rslt~14
rslt~20
rslt~22
1'h0
Add0
1'h0
Add1
rslt~2
rslt~10
rslt~18
rslt~11
rslt~0
rslt~8
rslt~16
rslt~3
rslt~19
rslt~1
rslt~17
rslt~9
rslt~15
rslt~23
rslt~7
Mux1
Mux2
Mux3
Mux5
Mux4
Mux7
Mux8
Mux0
Mux6
WideXor0
WideNor0
zero~not

Timing Diagram



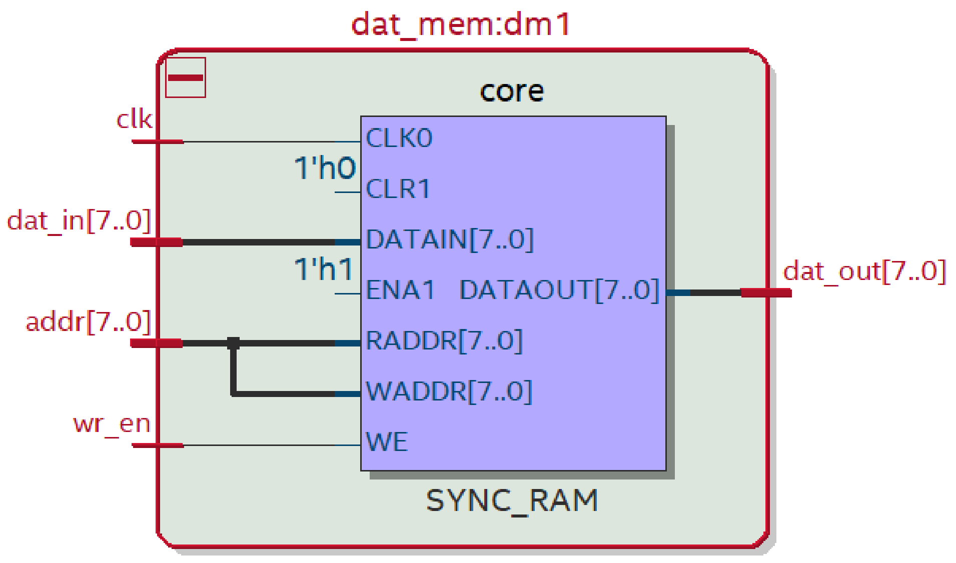
Data Memory

Module file name: dat_mem.sv

Functionality Description

This module is the main data storage of the processor that is slower to access but can store more data than the register file. Data in data memory can be read or modified. To read data, the “addr” field specifies the memory address to read, the resulting data at the memory location will be at “dat_out” field. To modify data, the same “addr” field specifies the memory address to write to, and the data is specified in “dat_in” field. The “clk” input ensures that only one read or write can happen per clock cycle.

Schematic



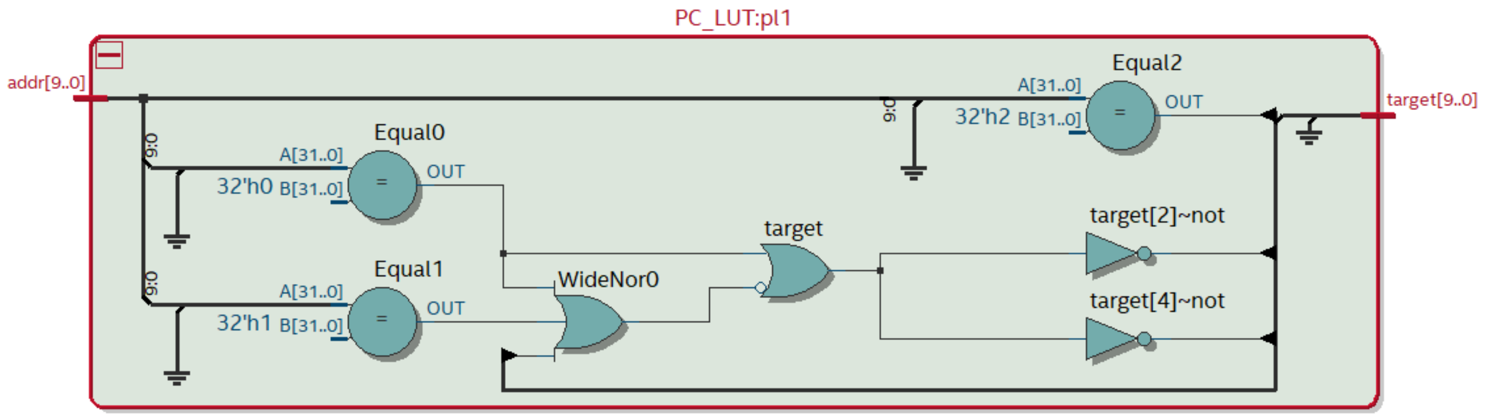
Lookup Tables

Module file name: PC_LUT.sv

Functionality Description

This module is a look up table that maps the current PC value to any address that we want to jump to. In the case that the current PC is pointing to a branch instruction, the look up table will provide the next PC value based on the address of the label to jump to.

Schematic



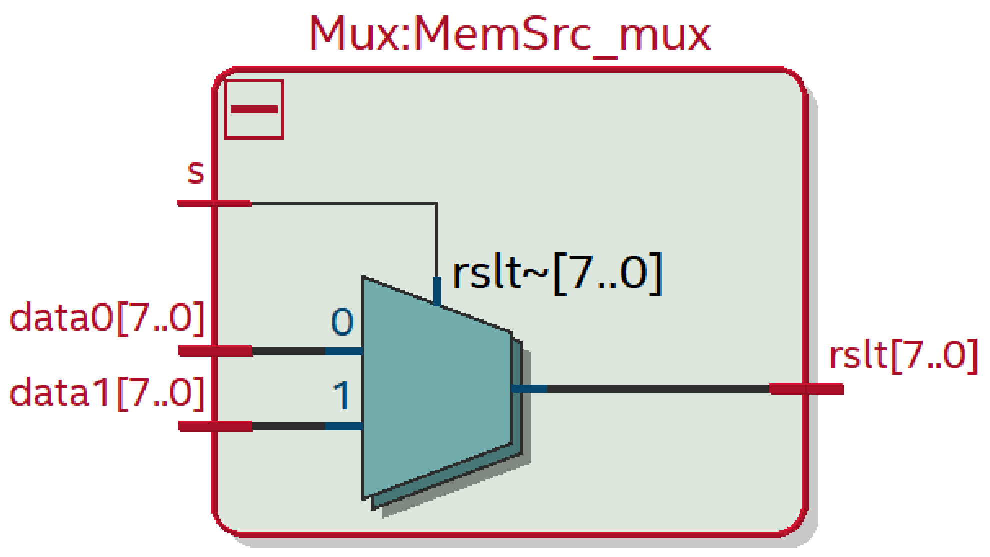
Muxes (Multiplexers)

Module file name: mux.sv

Functionality Description

This module selects 1 of 2 values based on a selector input. This is used heavily to construct the datapath so that data can flow to the correct modules.

Schematic



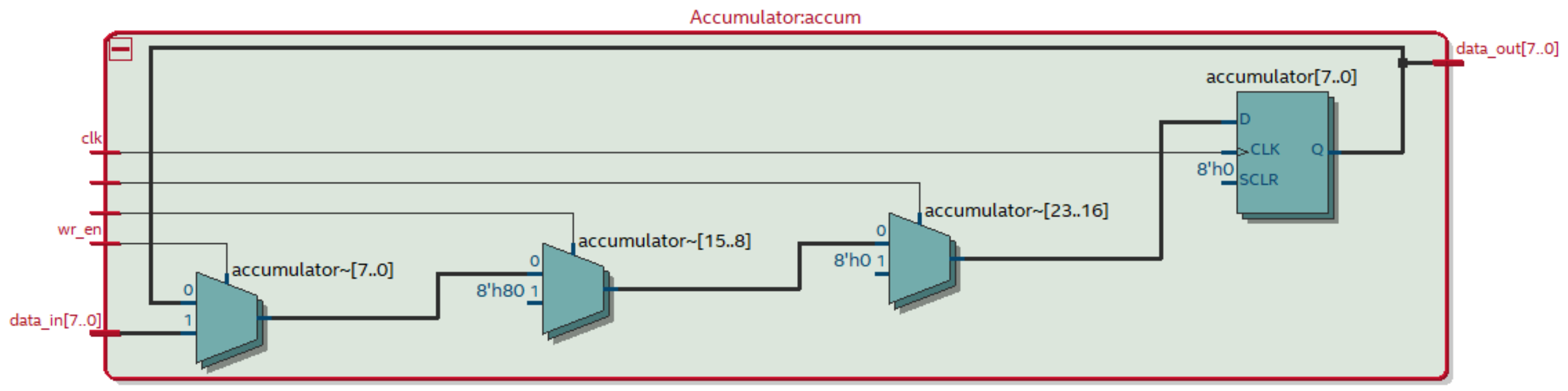
Accumulator

Module file name: Accumulator.sv

Functionality Description

This module is a special purpose register that is closely connected to the ALU, data memory, and the register file. The accumulator holds exactly one 8-bit value. To change the value, the new value is provided in “data_in” field, and the “wr_en” control bit must be set to true to change the accumulator value. The accumulator value can be accessed anytime through the “data_out” field. The “reset” field can be set to true to change the accumulator value to 0, while the “set” field can be set to true to change the value to 1.

Schematic



Accumulator:accum
data_out[7..0]
accumulator[7..0]
D
CLK
Q
8'h0
SCLR
clk
wr_en
accumulator~[23..16]
0
8'h0 1
accumulator~[15..8]
0
8'h80 1
accumulator~[7..0]
0
1
data_in[7..0]

Program Implementation

Program 1 Pseudocode

```
/*
  Hemming Encoding (needs to be converted to Assembly)
  Input: mem[0:29] - 15 original messages
  Output: mem[30:59] - 15 corresponding encoded messages
*/
for (int i = 0; i < 30; i += 2)
{
  // Load message
  char firstHalf = mem[i + 1];
  char secondHalf = mem[i];

  // Get bits
  char b[17];
  for (int j = 7; j >= 0; j--)
  {
    b[j + 8 + 1] = (firstHalf >> j) & 1;
    b[j + 1] = (secondHalf >> j) & 1;
  }

  // Calculate parity bits
  char p8 = b[11] ^ b[10] ^ b[9] ^ b[8] ^ b[7] ^ b[6] ^ b[5];
  char p4 = b[11] ^ b[10] ^ b[9] ^ b[8] ^ b[4] ^ b[3] ^ b[2];
  char p2 = b[11] ^ b[10] ^ b[7] ^ b[6] ^ b[4] ^ b[3] ^ b[1];
  char p1 = b[11] ^ b[9] ^ b[7] ^ b[5] ^ b[4] ^ b[2] ^ b[1];
  char p0 = b[11] ^ b[10] ^ b[9] ^ b[8] ^ b[7] ^ b[6] ^ b[5] ^ b[4] ^ b[3] ^ b[2] ^ b[1] ^ p8
^ p4 ^ p2 ^ p1;
```

```
  // Set parity bits
  char result[] = {b[11], b[10], b[9], b[8], b[7], b[6], b[5], p8,
                   b[4], b[3], b[2], p4, b[1], p2, p1, p0};

  // Write result to memory
  char firstHalfEncoded = 0;
  char secondHalfEncoded = 0;
  for (int j = 7; j >= 0; j--)
  {
    firstHalfEncoded |= result[7 - j] << j;
    secondHalfEncoded |= result[7 - j + 8] << j;
  }
  mem[i + 1 + 30] = firstHalfEncoded;
  mem[i + 30] = secondHalfEncoded;
}
```

Program 1 Assembly Code

```
// For loop

SETI 0
PUT R1        // R1 = i = 0

SETI 30
PUT R5        // R5 = 30

main_loop:

LOAD R1
PUT R2        // R2 = mem[i] = secondHalf

GET R1
```

```
ADDI 1
LOADA R3        // R3 = mem[i + 1] = firstHalf

SETI 80
PUT R4          // R4 = b = 80 (addr to mem)


// Inner For loop

SETI 7
PUT R6          // R6 = j = 7

SETI -1
PUT R7          // R7 = -1

SETI 1
PUT R8          // R8 = 1

get_bit_loop:

GET R3
RS R6
AND R8
PUT R9          // R9 = (firstHalf >> j) & 1

GET R2
RS R6
AND R8
PUT R10         // R10 = (secondHalf >> j) & 1

GET R4
ADD R6
```

```
ADDI 9          // b + j + 9

SAVEA R9        // mem[b + j + 8 + 1] = (firstHalf >> j) & 1;

SUBI 8          // b + j + 1
SAVEA R10       // mem[b + j + 1] = (secondHalf >> j) & 1;

GET R6
SUBI 1          // j--
PUT R6
EQ R7
BZ get_bit_loop  // j == -1 ?


// Calculate parity
GET R4
ADDI 11
LOADA R10       // R10 = mem[b + 11]
SUBI 1
LOADA R11       // R11 = mem[b + 10]
SUBI 1
LOADA R12       // R12 = mem[b + 9]
SUBI 1
LOADA R13       // R13 = mem[b + 8]
SUBI 1
LOADA R14       // R14 = mem[b + 7]
SUBI 1
LOADA R15       // R15 = mem[b + 6]
SUBI 1
LOADA R16       // R16 = mem[b + 5]

GET R10
```

```
XOR R11
PUT R8          // R8 = p2 = 11 ^ 10
XOR R12
XOR R13
PUT R7          // R7 = p4 = 11 ^ 10 ^ 9 ^ 8
XOR R14
XOR R15
XOR R16
PUT R6          // R6 = p8 = 11 ^ 10 ^ 9 ^ 8 ^ 7 ^ 6 ^ 5

GET R4
ADDI 4
LOADA R9         // R9 = mem[b + 4]
SUBI 1
LOADA R10        // R10 = mem[b + 3]
SUBI 1
LOADA R11        // R11 = mem[b + 2]

GET R7
XOR R9
XOR R10
XOR R11
PUT R7          // R7 = p4 = 11 ^ 10 ^ 9 ^ 8 ^ 4 ^ 3 ^ 2

GET R4
ADDI 1
LOADA R13        // R13 = mem[b + 1]

GET R8
XOR R14
XOR R15
XOR R9
XOR R10
XOR R13
PUT R8          // R8 = p2 = 11 ^ 10 ^ 7 ^ 6 ^ 4 ^ 3 ^ 1

GET R4
ADDI 11
LOADA R10       // R10 = p1 = mem[b + 11]

GET R10
XOR R12
XOR R14
XOR R16
XOR R9
XOR R11
XOR R13
PUT R10         // R10 = p1 = 11 ^ 9 ^ 7 ^ 5 ^ 4 ^ 2 ^ 1

GET R4
ADDI 3
LOADA R14       // R14 = mem[b + 3]

GET R6
XOR R9
XOR R14
XOR R11
XOR R13
XOR R6
XOR R7
XOR R8
XOR R10
PUT R12         // R12 = p0 = p8 ^ 4 ^ 3 ^ 2 ^ 1 ^ p8 ^ p4 ^ p2 ^ p1
```

```
// Put bits together as result

SETI 0
PUT R15       // firstHalfEncoded = 0
PUT R16       // secondHalfEncoded = 0

GET R9        // R9 = mem[b + 4]
LSI 7
OR R16
PUT R16       // set b4

GET R14       // R14 = mem[b + 3]
LSI 6
OR R16
PUT R16       // set b3

GET R11       // R11 = mem[b + 2]
LSI 5
OR R16
PUT R16       // set b2

GET R13       // R13 = mem[b + 1]
LSI 3
OR R16
PUT R16       // set b1

GET R6        // R6 = p8
OR R15
PUT R15       // set p8

GET R7        // R7 = p4
LSI 4
```

```
OR R16
PUT R16       // set p4

GET R8       // R8 = p2
LSI 2
OR R16
PUT R16       // set p2

GET R10       // R10 = p1
LSI 1
OR R16
PUT R16       // set p1

GET R12       // R12 = p0
OR R16
PUT R16       // set p0

// All values in registers are covered, need to access mem for remaining bits
GET R4
ADDI 11
LOADA R6      // R6 = mem[b + 11]
SUBI 1
LOADA R7      // R7 = mem[b + 10]
SUBI 1
LOADA R8      // R8 = mem[b + 9]
SUBI 1
LOADA R9      // R9 = mem[b + 8]
SUBI 1
LOADA R10     // R10 = mem[b + 7]
SUBI 1
LOADA R11     // R11 = mem[b + 6]
SUBI 1
```

```
LOADA R12      // R12 = mem[b + 5]

GET R6         // R6 = b11
LSI 7
OR R15
PUT R15        // set b11

GET R7         // R7 = b10
LSI 6
OR R15
PUT R15        // set b10

GET R8         // R8 = b9
LSI 5
OR R15
PUT R15        // set b9

GET R9         // R9 = b8
LSI 4
OR R15
PUT R15        // set b8

GET R10        // R10 = b7
LSI 3
OR R15
PUT R15        // set b7

GET R11        // R11 = b6
LSI 2
OR R15
PUT R15        // set b6
```

```
GET R12       // R12 = b5
LSI 1
OR R15
PUT R15       // set b5

// Write result to memory

GET R1
ADD R5
SAVEA R16     // mem[i + 30] = secondHalfEncoded;
ADDI 1
SAVEA R15     //  mem[i + 1 + 30] = firstHalfEncoded;


GET R1
ADDI 2        // i += 2
PUT R1
EQ R5
BZ main_loop    // i == 30 ?
```

Program 2 Pseudocode

```
/*
  Hemming Decoding (needs to be converted to Assembly)
  Input: mem[30:59] - 15 encoded messages with no error, or 1 or 2 bit errors
  Output: mem[0:29] - 15 corresponding original messages (decoded and corrected if necessary)
*/
for (int i = 30; i < 60; i += 2)
{
  // Load message
  char firstHalf = mem[i + 1];
  char secondHalf = mem[i];
```

```
    // Get bits
    char b[17];
    for (int j = 7; j >= 0; j--)
    {
      b[j + 8 + 1] = (firstHalf >> j) & 1;
      b[j + 1] = (secondHalf >> j) & 1;
    }

    // Compare parity bits
    char p8_e = b[16] ^ b[15] ^ b[14] ^ b[13] ^ b[12] ^ b[11] ^ b[10] ^ b[9];
    char p4_e = b[16] ^ b[15] ^ b[14] ^ b[13] ^ b[8] ^ b[7] ^ b[6] ^ b[5];
    char p2_e = b[16] ^ b[15] ^ b[12] ^ b[11] ^ b[8] ^ b[7] ^ b[4] ^ b[3];
    char p1_e = b[16] ^ b[14] ^ b[12] ^ b[10] ^ b[8] ^ b[6] ^ b[4] ^ b[2];
    char p0_e = b[16] ^ b[15] ^ b[14] ^ b[13] ^ b[12] ^ b[11] ^ b[10] ^ b[9] ^ b[8] ^ b[7] ^
b[6] ^ b[5] ^ b[4] ^ b[3] ^ b[2] ^ b[1];

    printf("p8_e: %d\n", p8_e);
    printf("p4_e: %d\n", p4_e);
    printf("p2_e: %d\n", p2_e);
    printf("p1_e: %d\n", p1_e);
    printf("p0_e: %d\n", p0_e);

    // Analyze parity bits for error
    char result[17] = {0};

    if (p0_e)
    {
      // 1-bit error
      result[16] = 0;
      result[15] = 1;
```

```
  // Correction
  int error_index = p8_e * 8 + p4_e * 4 + p2_e * 2 + p1_e;
  error_index += 1;
  b[error_index] = !b[error_index];
}
else
{
  // No error or 2-bit error
  if (p8_e || p4_e || p2_e || p1_e)
  {
    // 2-bit error
    result[16] = 1;
    result[15] = 1;
  }
  else
  {
    // No error
    result[16] = 0;
    result[15] = 0;
  }
}

// Store decoded message
result[11] = b[16];
result[10] = b[15];
result[9] = b[14];
result[8] = b[13];
result[7] = b[12];
result[6] = b[11];
result[5] = b[10];
result[4] = b[8];
result[3] = b[7];
```

```
    result[2] = b[6];
    result[1] = b[4];

    // Write result to memory
    char firstHalfEncoded = 0;
    char secondHalfEncoded = 0;
    for (int j = 7; j >= 0; j--)
    {
      firstHalfEncoded |= result[j + 8 + 1] << j;
      secondHalfEncoded |= result[j + 1] << j;
    }
    mem[i - 30 + 1] = firstHalfEncoded;
    mem[i - 30] = secondHalfEncoded;
  }
```

Program 2 Assembly Code

```
// For loop

SETI 30
PUT R1          // R1 = i = 30

SETI 60
PUT R5          // R5 = 60

main_loop:

LOAD R1
PUT R2          // R2 = mem[i] = secondHalf

GET R1
ADDI 1
```

```
LOADA R3        // R3 = mem[i + 1] = firstHalf

SETI 80
PUT R4          // R4 = b = 80 (addr to mem)

// Inner For loop

SETI 7
PUT R6          // R6 = j = 7

SETI -1
PUT R7          // R7 = -1

SETI 1
PUT R8          // R8 = 1

get_bit_loop:

GET R3
RS R6
AND R8
PUT R9          // R9 = (firstHalf >> j) & 1

GET R2
RS R6
AND R8
PUT R10         // R10 = (secondHalf >> j) & 1

GET R4
ADD R6
ADDI 9          // b + j + 9
```

```
SAVEA R9        // mem[b + j + 8 + 1] = (firstHalf >> j) & 1;

SUBI 8          // b + j + 1
SAVEA R10       // mem[b + j + 1] = (secondHalf >> j) & 1;

GET R6
SUBI 1          // j--
PUT R6
EQ R7
BZ get_bit_loop  // j == -1 ?

// Compare parity bits

GET R4
ADDI 8
ADDI 8
LOADA R9        // R9 = mem[b + 16]
SUBI 1
LOADA R10       // R10 = mem[b + 15]
SUBI 1
LOADA R11       // R11 = mem[b + 14]
SUBI 1
LOADA R12       // R12 = mem[b + 13]
SUBI 1
LOADA R13       // R13 = mem[b + 12]
SUBI 1
LOADA R14       // R14 = mem[b + 11]
SUBI 1
LOADA R15       // R15 = mem[b + 10]
SUBI 1
LOADA R16       // R16 = mem[b + 9]
```

```
GET R9          // R9 = p1 = 16
XOR R10
PUT R8          // R8 = p2 = 16 ^ 15
XOR R11
XOR R12
PUT R7          // R7 = p4 = 16 ^ 15 ^ 14 ^ 13
XOR R13
XOR R14
XOR R15
XOR R16
PUT R6          // R6 = p8 = 16 ^ 15 ^ 14 ^ 13 ^ 12 ^ 11 ^ 10 ^ 9

GET R9
XOR R11
PUT R9          // R9 = p3 = 16 ^ 14

GET R4
ADDI 8
LOADA R10       // R10 = mem[b + 8]
SUBI 1
LOADA R11       // R11 = mem[b + 7]
SUBI 1
LOADA R12       // R12 = mem[b + 6]
SUBI 1
LOADA R16       // R16 = mem[b + 5]

GET R16
XOR R10
XOR R11
XOR R12
PUT R16         // R16 = 8 ^ 7 ^ 6 ^ 5
```

```
GET R7
XOR R16
PUT R7          // R7 = p4 = 16 ^ 15 ^ 14 ^ 13 ^ 8 ^ 7 ^ 6 ^ 5

GET R8
XOR R13
XOR R14
XOR R10
XOR R11
PUT R8          // R8 = p2 = 16 ^ 15 ^ 12 ^ 11 ^ 8 ^ 7

GET R9
XOR R13
XOR R15
XOR R10
XOR R12
PUT R9          // R9 = p1 = 16 ^ 14 ^ 12 ^ 10 ^ 8 ^ 6

GET R4
ADDI 4
LOADA R11       // R11 = mem[b + 4]
SUBI 1
LOADA R12       // R12 = mem[b + 3]
SUBI 1
LOADA R13       // R13 = mem[b + 2]
SUBI 1
LOADA R14       // R14 = mem[b + 1]

GET R8
XOR R11
XOR R12
PUT R8         // R8 = p2 = 16 ^ 15 ^ 12 ^ 11 ^ 8 ^ 7 ^ 4 ^ 3
```

```
GET R9
XOR R11
XOR R13
PUT R9        // R9 = p1 = 16 ^ 14 ^ 12 ^ 10 ^ 8 ^ 6 ^ 4 ^ 2

GET R6
XOR R16
XOR R11
XOR R12
XOR R13
XOR R14
PUT R10       // R10 = p0 = p8 ^ 8 ^ 7 ^ 6 ^ 5 ^ 4 ^ 3 ^ 2 ^ 1

// Analyze parity bits for error

SETI 1
PUT R11         // R11 = 1
GET R10         // Acc = p0_e
EQ R11          // if p0_e == 1, Acc = 1
BZ else

// 1-bit error
SETI 1
PUT R14         // R14 = F0 = 1
SETI 0
PUT R13         // R13 = F1 = 0

// Correction
GET R6
LSI 3          // Acc = p8_e * 8
PUT R6
```

```
GET R7
LSI 2
PUT R7

GET R8
LSI 1
ADD R9
ADD R7
ADD R6
PUT R9
ADDI 1
ADD R4
PUT R11       // R11 = b + p8_e * 8 + p4_e * 4 + p2_e * 2 + p1_e * 1 + 1
LOAD R11      // Acc = mem[b + p8_e * 8 + p4_e * 4 + p2_e * 2 + p1_e * 1 + 1]
EQ R13        // if Acc == 0, Acc = 1
PUT R12       // R12 = flipped bit
GET R11
SAVEA R12     // mem[b + p8_e * 8 + p4_e * 4 + p2_e * 2 + p1_e * 1 + 1] = flipped bit

SETI 0
BZ exit

else:

// No error or 2-bit
GET R6
OR R7
OR R8
OR R9
EQ R11
BZ inner_else
```

```
// 2-bit error
SETI 1
PUT R14        // R14 = F0 = 1
PUT R13        // R13 = F1 = 1
SETI 0
BZ exit

inner_else:
// no error
SETI 0
PUT R14        // R14 = F0 = 0
PUT R13        // R13 = F1 = 0

exit:

GET R14
LSI 6
PUT R14

GET R13
LSI 7
ADD R14
PUT R15

SETI 0
PUT R16

GET R4
ADDI 8
ADDI 8
LOADA R6       // R6 = mem[b + 16]
```

```
SUBI 1
LOADA R7       // R7 = mem[b + 15]
SUBI 1
LOADA R8       // R8 = mem[b + 14]
SUBI 1
LOADA R9       // R9 = mem[b + 13]
SUBI 1
LOADA R10      // R10 = mem[b + 12]
SUBI 1
LOADA R11      // R11 = mem[b + 11]
SUBI 1
LOADA R12      // R12 = mem[b + 10]

GET R6
LSI 2
OR R15
PUT R15

GET R7
LSI 1
OR R15
PUT R15

GET R8
OR R15
PUT R15

GET R9
LSI 7
OR R16
PUT R16
```

```
GET R10
LSI 6
OR R16
PUT R16

GET R11
LSI 5
OR R16
PUT R16

GET R12
LSI 4
OR R16
PUT R16

GET R4
ADDI 8
LOADA R6       // R6 = mem[b + 8]
SUBI 1
LOADA R7       // R7 = mem[b + 7]
SUBI 1
LOADA R8       // R8 = mem[b + 6]
SUBI 2
LOADA R9       // R9 = mem[b + 4]

GET R6
LSI 3
OR R16
PUT R16

GET R7
LSI 2
```

```
OR R16
PUT R16

GET R8
LSI 1
OR R16
PUT R16

GET R9
OR R16
PUT R16

// Write result to memory

GET R1
SUBI 15
SUBI 15
SAVEA R16      // mem[i - 30] = secondHalfEncoded;
ADDI 1
SAVEA R15      //  mem[i + 1 + 30] = firstHalfEncoded;


GET R1
ADDI 2         // i += 2
PUT R1
EQ R5
BZ main_loop    // i == 60 ?
```

Program 3 Pseudocode

```
/*
```

```
  Pattern Finding (needs to be converted to Assembly)
  Input: mem[0:31] - 32 byte strings
  Output:
    mem[32] - 5-bit pattern (first 5 bits of byte)
    mem[33] - # bytes containing pattern
    mem[34] - # times pattern appears within each byte
    mem[35] - # times pattern appears across all bytes
*/
char pattern = mem[32]; // first 5 bits

char prevString = mem[0];
char numBytesWithPattern = 0;
char numPatternInBytes = 0;
char numPatternAcrossBytes = 0;

for (int i = 1; i <= 32; i++)
{
  // Load message
  char leftString = prevString;
  char rightString = mem[i];

  // Count occurances of pattern within each byte (leftString and rightString)
  char leftStringHasPattern = 0;
  unsigned char patternMask = 0b11111000;
  unsigned char patternSh = pattern;
  for (int j = 0; j < 4; j++)
  {
    char left = leftString & patternMask;
    if (left == patternSh)
    {
      leftStringHasPattern = 1;
      numPatternInBytes++;
    }
    patternMask = patternMask >> 1;
    patternSh = patternSh >> 1;
  }

  if (leftStringHasPattern)
  {
    numBytesWithPattern++;
  }

  if (i == 32)
  {
    break;
  }

  // Count occurances of pattern across the two bytes
  char patternMaskL = 0b00001111;
  char patternMaskR = 0b10000000;

  for (int j = 0; j < 4; j++)
  {
    char left = leftString & patternMaskL;
    char right = rightString & patternMaskR;
    char patternL = pattern >> (j + 4);
    patternL = patternL & patternMaskL;
    char patternR = pattern << (4 - j);

    if (left == patternL && right == patternR)
    {
      numPatternAcrossBytes++;
    }
    patternMaskL = patternMaskL >> 1;
```

```
      patternMaskR = patternMaskR >> 1;
    }

    prevString = rightString;
  }

  // Update memory
  mem[33] = numBytesWithPattern;
  mem[34] = numPatternInBytes;
  mem[35] = numPatternAcrossBytes + numPatternInBytes;
```

Program 3 Assembly Code

```
SETI 32
LOADA R13        // R13 = mem[32]

ADDI 1
PUT R5          // R5 = 33

SETI 0
LOADA R12        // R12 = mem[0] = prevString
PUT R14          // numBytesWithPattern = 0
PUT R15          // numPatternInBytes = 0
PUT R16          // numPatternAcrossBytes = 0

// For loop

SETI 1
PUT R1         // R1 = i = 1

main_loop:
```

```
GET R12
PUT R3          // R3 = leftString = prevString

GET R1
LOADA R2        // R2 = mem[i]

// Count occurances of pattern within each byte

SETI 0
PUT R6          // R6 = leftStringHasPattern = 0

SETI 0
PUT R10

GET R13
PUT R8        // R8 = patternSh = pattern

SETI -8       // patternMask = 11111000
AND R3        // Acc = left = leftString & patternMask
EQ R8
BZ exit       // left != patternSh go to else

SETI 1
PUT R6        // leftStringHasPattern = 1
GET R15
ADDI 1
PUT R15       // numPatternInBytes ++

exit:

SETI 1
PUT R9
```

```
GET R8
RS R9
PUT R8          // patternSh = patternSh >> 1

SETI 124
PUT R7          // R7 = patternMask = 01111100

AND R8
PUT R8

SETI 1
PUT R4          // j = 1

SETI 4
PUT R11

inner_for:

GET R7
AND R3        // Acc = left = leftString & patternMask
EQ R8
BZ inner_for_exit       // left != patternSh go to else

SETI 1
PUT R6
GET R15
ADDI 1
PUT R15

inner_for_exit:
```

```
GET R8
RS R9
PUT R8

GET R7
RS R9
PUT R7

GET R4
ADDI 1          // j ++
PUT R4
EQ R11
BZ inner_for    // j == 4 ?


GET R6
EQ R9
BZ hasPatternExit   // leftStringHasPattern == 0 then go to else

GET R14
ADDI 1
PUT R14          // numBytesWithPattern ++

hasPatternExit:

SETI 32
EQ R1
BZ skip_break    // i != 32 go to else

SETI 0
BZ end_for
```

```
skip_break:

// Count occurances of pattern across bytes

SETI 15
PUT R7          // R7 = patternMaskL = 00001111

SETI -128
PUT R8          // R8 = patternMaskR = 10000000

SETI 0
PUT R4          // R4 = j = 0

SETI 4
PUT R6

inner_for_across:

SETI 2
PUT R11

SETI 0
PUT R10          // z = 0

GET R13
PUT R9

left_shift_for:
GET R9
LSI 1
PUT R9
```

```
GET R10
ADDI 1          // Z ++
PUT R10
EQ R6
BZ left_shift_for    // z == (4 - j) ?

GET R2
AND R8          // Acc = right = rightString & patternMaskR

EQ R9
PUT R10         // R10 = 1 if right == patternR, else 0

GET R4
ADDI 4
PUT R5          // R5 = j + 4
GET R13
RS R5
AND R7
PUT R9          // R9 = patternL = pattern >> (j + 4) & patternMaskL

GET R3
AND R7          // Acc = left = leftString & patternMaskL
EQ R9          // Acc = 1 if left == patternL, else 0
ADD R10         // Acc = 2 iff right == patternR && left == patternL
EQ R11
BZ exit_count   // if right == pattern && left == pattern, go to else

GET R16
ADDI 1
PUT R16

exit_count:
```

```
SETI 1
PUT R9

GET R7
RS R9
PUT R7          // patternMaskL >>= 1

GET R8
RS R9
PUT R8          // patternMaskR >>= 1

SETI 4
PUT R11          // reset R11 = 4

GET R6
SUBI 1
PUT R6

GET R4
ADDI 1          // j ++
PUT R4
EQ R11
BZ inner_for_across    // j == 4 ?

GET R2
PUT R12         // prevString = rightString

SETI 33
PUT R5          // reset R5 = 33

end_for:
```

```
GET R1
ADDI 1          // i ++
PUT R1
EQ R5
BZ main_loop    // i == 33 ?

SETI 33
SAVEA R14       // mem[33] = numBytesWithPattern
SETI 34
SAVEA R15       // mem[34] = numPatternInBytes

GET R15
ADD R16
PUT R16        // numPatternAcrossBytes += numPatternInBytes

SETI 35
SAVEA R16       // mem[35] = numPatternAcrossBytes
```

Changelog

- Final report
  - Architecture Overview: Updated architecture diagram with PC_LUT changes
  - Individual Component Specification: Updated top_level schematic
  - Program Implementation: Updated all pseudocode and assembly code that fully works
- Milestone 3
  - Assembly: updated assembly code based on changes to ISA in Milestone 2
- Milestone 2
  - Architectural Overview: added control unit, LUT, and generally more details to the architecture diagram
  - Machine formats:
    - Added A-type instruction to more clearly distinguish between instruction formats, especially instructions dealing with immediates
    - Removed SET instruction because we didn’t use it in our assembly code and seems hard to implement
    - Removed BZU instructions because we no longer need it with the PC look up table
    - Changed XORS to XOR instruction because it’s easier to implement
- Milestone 1
  - Initial version