CSE 141L Milestone 3

Tyler Le, A16527713; Alexander G. Arias, A16525320; Aiko Coanaya, A16397455

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.

Tyler Le
Alexander G. Arias
Aiko Coanaya

0. Team

Tyler Le, A16527713; Alexander G. Arias, A16525320; Aiko Coanaya, A16397455

1. Introduction

Name your architecture. What is your overall philosophy? What specific goals did you strive to achieve? Can you classify your machine in any of the standard ways (e.g., stack machine, accumulator, register-register/load-store, register-memory)? If so, which? If not, devise a name for your class of machine. Word limit: 200 words.

Our architecture is called *The Super Simple Architecture* (*SSA*), and our motivations for the design are efficiency, clarity, and simplicity.

SSA aims to execute the three programs efficiently. The principle behind this architecture is **minimalism**. By eliminating unnecessary complexities, we enhance the clarity of the system's operations, making it both user-friendly and effective.

Specific Goals:

Simplicity: Rather than over-complicating the design, *SSA's* design prioritizes straightforwardness so that anyone can understand it.

Functionality: While simplicity is key, it should not come at the expense of functionality. *SSA* guarantees that despite its minimalist design, it remains adept at accomplishing the three tasks it's set out to achieve.

Machine Classification:

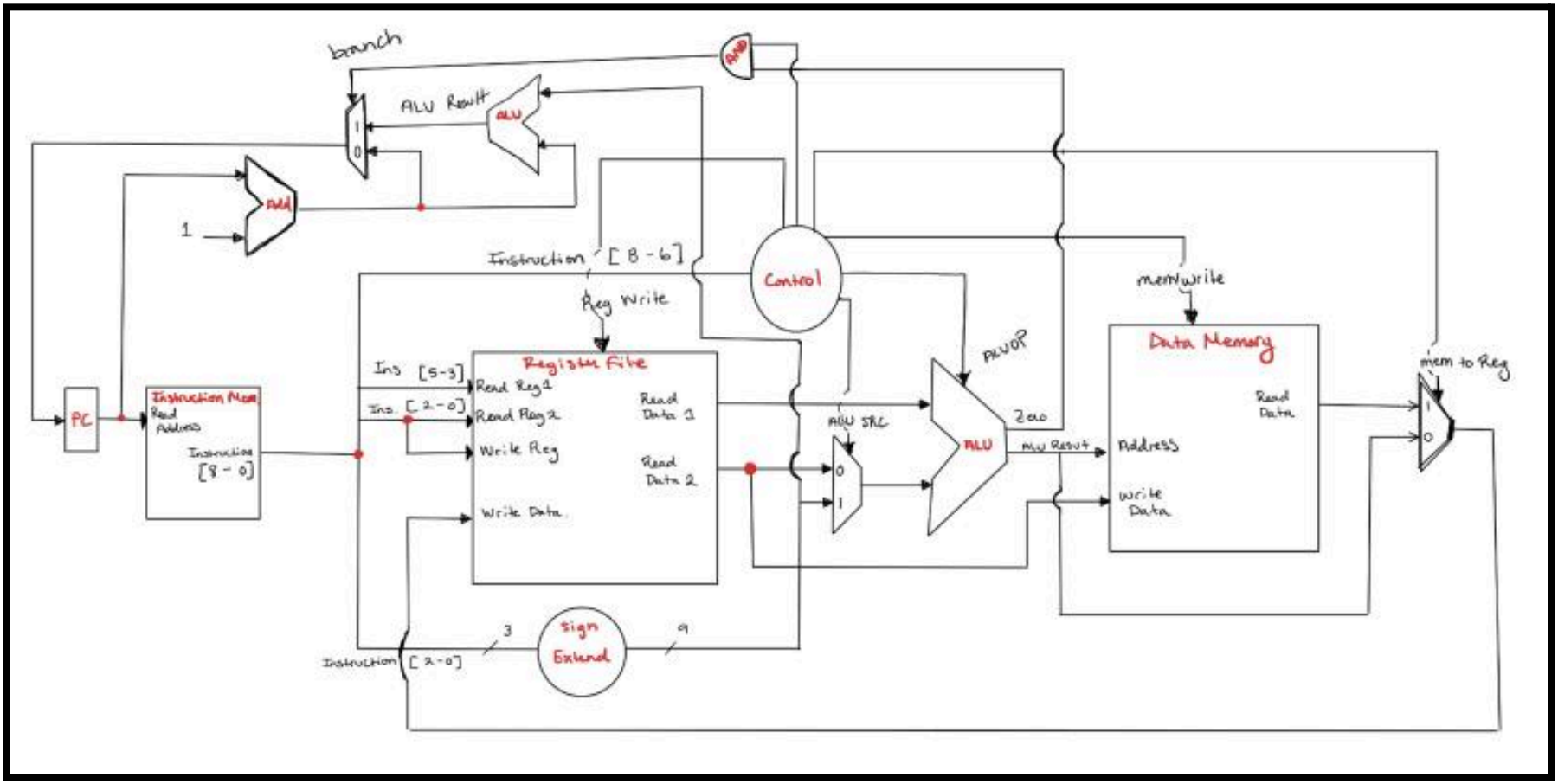
In terms of classification, the *SSA* falls under the 'register-register / load-store' category. This classification indicates that operations are primarily executed between registers with distinct instructions dedicated to loading and storing data from and to memory.

By adhering to these principles and goals, we believe *SSA* offers a perfect blend of simplicity and functionality while being able to run the given programs efficiently.

2. Architectural Overview

This must be in picture form. What are the major building blocks you expect your processor to be made up of? You must have data memory in your architecture. (Example of MIPS: https://www.researchgate.net/figure/The-MIPS-architecture_fig1_251924531)

DIAGRAM 1, VERSION 2:



Omitted all control signals for clarity, and better visuals.

3. Machine Specification

Instruction formats

Two example rows have been filled for you. When you submit, do not include the example types. Add rows as necessary. In your submission, please delete this paragraph.

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	Opcode - 3 bits Source Register1 - 3 bits (also Destination Reg for XOR and LD) Source Register2 or Destination Address - 3 bits (Destination Reg for ST)	xor, load, store
I	Opcode - 3 bits Source Register - 3 bits Immediate - 3 bits (Destination is the same as Source Register) *** This layout only applies to addi, andi, and logical shift *** —---------- Or Opcode - 3 bits Source Register - 1 bit	addi, andi, logical shift, beq

	Immediate - 1 bit Target Address - 4 bits *** This layout only applies to beq ***	
J	Opcode - 3 bits Target - 6 bits	jump

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
		Assume rs is the first source register, rt is the second source register, and rd is the destination register.		This row is an explanation of abbreviations.

xor	R	3 bits opcode (000)
		3 bits rs (XXX)
		3 bits rt (XXX)
			# Assume R0 has `0b0001_0001`
			# Assume R1 has `0b1001_0000`
			
			(Assume unsigned values, thus omitting signed bit (9th bit))
			
			`XOR  R0  R1 ⇔`
			`000 000  001`
			
			# after **xor** instruction, R0 now holds `0b1000_0001 since it is a bitwise XOR`
				R[rs] = R[rs] ^ R[rt]

beq	I	3 bits opcode (001) 1 bits rs (X)	# Assume R1 has `0b0001_0001`	If R[r1] == imm then we branch to the 9-bit memory address held in R[r0].

		1 bits imm (X)
		4 bits rd (XXXX)
			`beq R1  0  TARGET ⇔`
			`001  1  0  0000`
			
			# After **beq** instruction, if R1 and 0 are the same, then we branch out to the target. This target will set the pc to point to the target. Some targets will be mapped to specific pc values and operations.	

addi	I	3 bits opcode (010)
		3 bits rs (XXX)
		3 bits imm (XXX)
		(The destination register is the same as the source.)
			# Assume R0 has `0b0000_0000`
			# Assume IMM is `0b0000_0001`
			
			The immediate is signed
			
			`addi R0  imm ⇔`
			`010 000 001`
			
			# after addi instruction, R0 now holds `0b0000_0001`
				R0 would be the destination register and would be the first source register.
				IMM would be the immediate

andi	I	3 bits opcode (011)
		3 bits rs (XXX)
		3 bits imm (XXX)
		(The destination register is the same as the source.)
			# Assume R0 has `0b0000_0011`
			# Assume IMM is `0b0000_0001`
			
			`andi R0  imm ⇔`
			`011 000 001`
			
			# after andi instruction, R0 now holds `0b0000_0001`
				This operation simply isolates the lsb into a register of choice.

logical shift (ls)	I	3 bits opcode (100)
		3 bits rs (XXX)
		3 bits imm
			# Assume R0 has `0b1000_0000`
			
			`NOTE: imm msb is signed!`
			
			`ls  R0  imm ⇔`
				R0 would be the destination register and would be the first source register.
				IMM would be the immediate.
				NOTE:
				**********

			`100 000 010` # after ls instruction, R0 now holds `0b0010_0000`	**Negative value for left shift, positive value for right shift.** **********

Load (ld)	R	3 bits opcode (101)
		3 bits rs (XXX)
		3 bits data_mem_reg (XXX)
			# Assume R0 has `0bXXXX_XXXX`
			
			`ld R0 data_mem_reg ⇔`
			`101 000 001`
			
			# after load instruction, R0 now holds the memory contents stored at the address held by R1
				Load the memory contents at a specified address in r1, and put that content into register 0.
				r0 = M[r1]

Store (st)	R	3 bits opcode (101)
		3 bits rs (XXX)
		3 bits data_mem_reg (XXX)
			# Assume R0 has `0b0000_0001`
			# Assume R1 is `0bXXXX_XXXX`
			
			`st R0 data_mem_reg ⇔`
			`110 000 001`
			
			# After store instruction, The memory addres held in R1 `M[r1]` now contains the contents of register R0.
				M[r1] = r0.

Jump (j)	J	3 bits opcode (111)
		6 bits target (XXXXXX)
			`j target ⇔`
			`111 XXXXXX`
			
			# This instruction jumps to the specified target….
				Note: This target will be mapped to a memory address via the PC Look Up Table

Internal Operands

How many registers are supported? Is there anything special about any of the registers (e.g. constant, accumulator), or are all of them general purpose?

- We have eight general-purpose registers, thus there are no dedicated accumulators or constant registers in our architecture.
- One things to note is that certain operations are dedicated to specific registers for better program behavior and optimality. This includes instructions like BEQ, which can only be used with registers 0 or 1. For load and store operations, we used registers as place holders for constant memory addresses we need access to as well.

Control Flow (branches)

What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported? How do you accommodate large jumps?

- We support two control flow operations, beq and jump. The maximum branch distance is 2^9 or 2^10 because we predict our assembly files will be about this much in size. We will map particular values to certain locations in memory to support branching in the required place in our programs.
- For the beq instruction, the branch is taken if the two specified registers contain the same value. The target address is typically derived by an immediate used for mapping to an address specified during our fetch operation, which is then made into the current program counter (PC) or some kind of PC operation. If the condition is met, the PC is updated to this new address, effectively causing a branch.
- The jump instruction allows for unconditional branching. It uses an absolute address which is mapped to a memory address in the fetch unit and we set the program counter (PC) to a specified target mapping, effectively redirecting the flow of execution to that address.

Addressing Modes

What memory addressing modes are supported, e.g. direct, indirect? How are addresses calculated? Give examples.

- **Register Direct Addressing**: In this mode, the operand is located in a register, and the instruction specifies the register
  Example:
  xor r1, r2 // this xors th values in r1 and r2, and places the result in r1

- **Direct Memory Adressing**: In this mode, a register acts a pointer to memory.
  Example:
  ld r0, r1

- **Immediate addressing**

    Example:

        Addi r0 1

    These are the only addressing modes we anticipate to maintain.

4. 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.

- Programmers working with our ISA should note that it is a streamlined, efficient system focused on direct operations. Since our ISA is limited to 8 simple instructions and our register count is all general purpose and limited to 8 registers total, its utilization should be to optimize data lookup times by minimizing memory access times. As a result, programmers should try to load all essential values into registers before they commence any calculations or algorithms. They will ensure that all calculations are made via registers, which could potentially decrease execution time. An additional point to address is that our system would work better with reduced jump and branch instructions, so an emphasis on linear code execution could increase our ISAs efficiency. However, branching still works great. All programmers should strive to store values into memory after every completed routine.

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.

- Copying instructions/operations directly from competitor ISAs like MIPS/ARM was not possible. Although our ISAs share foundational concepts, their encodings, operational behavior, and overall architecture design differ significantly. The main point to address is that our SSA ISA is 9-bits fixed. This limitation reduces the complexity and number of instructions we can support. As a result, we can’t copy their instructions. Overall, we designed similar instructions that are tailored to our

architecture, given its constraints. By focusing on our instruction set, and by not copying the instructions of our competitors, we have achieved an efficient system.

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?

- The alu design we have came up with does handle some non-arithmetic instructions. For instance, it performs bitwise operations like XOR, and logical operations like AND, as well as shift operations. These operations are in MIPS and ARM. However, our design does not cover any sort of memory or PC manipulation or updating via the ALU. Incorporating non-arithmetic operations like these into the ALU design increases its functional complexity, necessitating intricate multiplexing and expanded control signals to accommodate these diverse operations. Such a design demands rigorous validation due to the broad range of instruction combinations and may encounter varied complications due to its rise in complexity. It is best to keep our ALU strictly dedicated to conducting as few tasks as possible, considering they are the most important in terms of its capabilities.

5. Individual Component Specification

Top Level

Module file name: **top_level.sv**
Module testbench file name: **top_level_tb.sv**

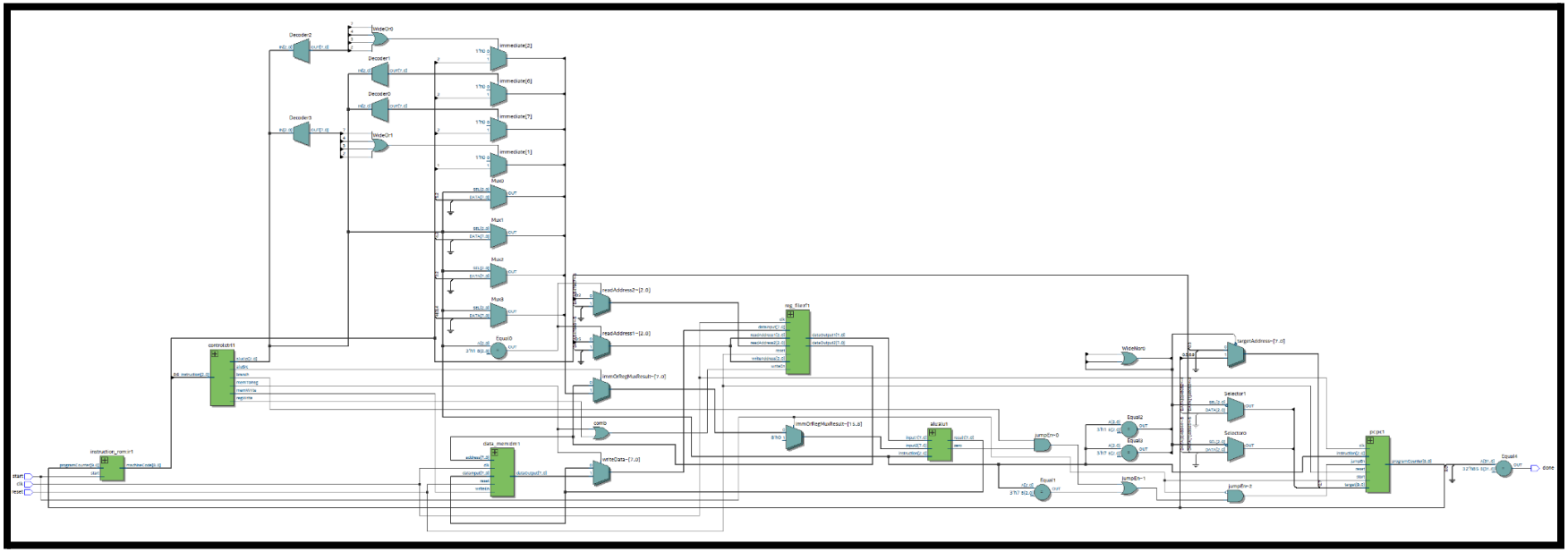
Functionality Description

The top_level module acts as the main integration point for a simple processor that sequentially executes a predefined set of instructions. Its primary function is to manage the flow of data between different sub-modules such as program counter, instruction memory, control unit, register file, ALU, and data memory.

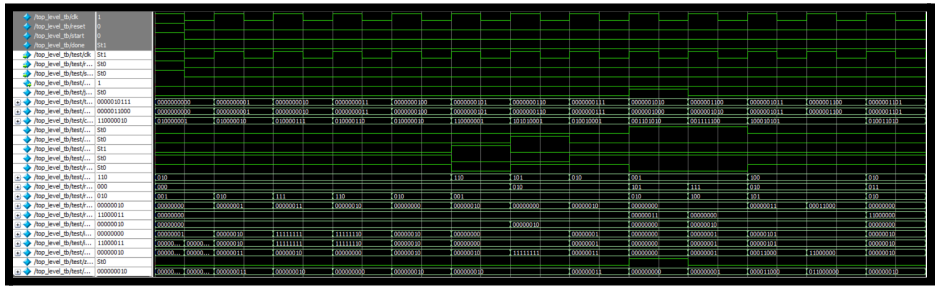
Testbench Description

The testbench for the top_level module is designed to emulate the environment in which the top-level module operates, injecting signals such as clk, reset, and start to initialize and control the flow of the test. It meticulously observes the effects of these inputs on the module's behavior, particularly looking at specific memory locations to verify correct data processing and storage, asserting the integrity of the system. Upon detecting the done signal's assertion, indicative of the module's completion of its task, the testbench reports the simulation time and concludes the testing process, offering insights into the module's performance and reliability.

Schematic



Timing Diagram



Program Counter

Module file name: **pc.sv**
Module testbench file name: **ps_tb.sv**

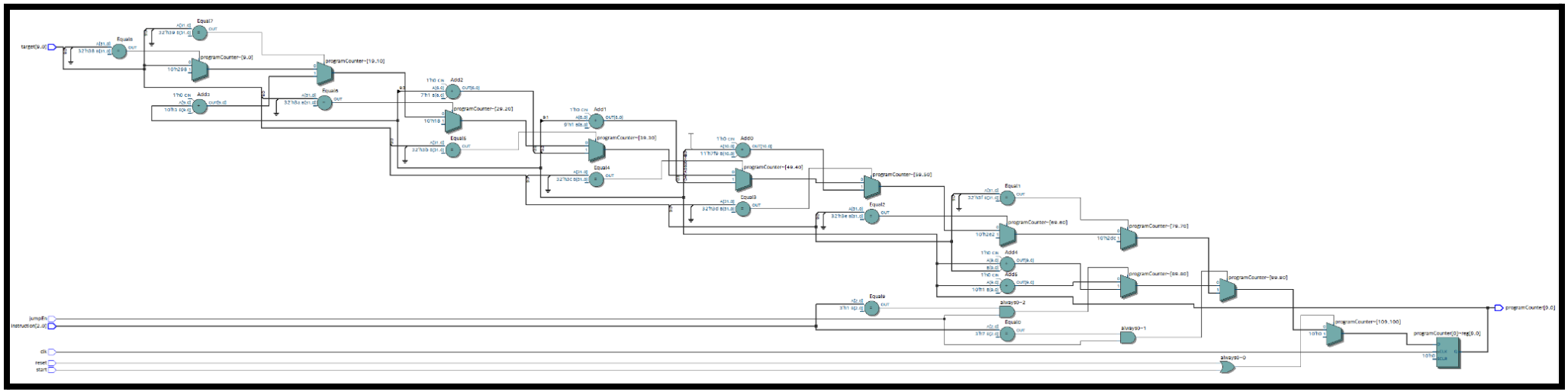
Functionality Description

The Program Counter is responsible for generating and managing addresses that point to locations in instruction memory. It provides sequential execution of instructions, but also support absolute to nonsequential addresses for branch and jump instructions.

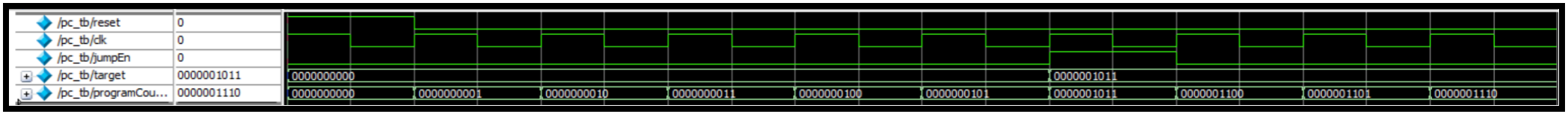
Testbench Description

The testbench pc_tb is designed, currently, to verify the functionality of the pc module, specifically focusing on its counting, jump, and reset features. Since the pc is a pretty direct module, the tests are pretty straightforward.

Schematic



Timing Diagram



Instruction Memory

Module file name: **instruction_rom.sv**
Module testbench file name: **instruction_rom_tb.sv**

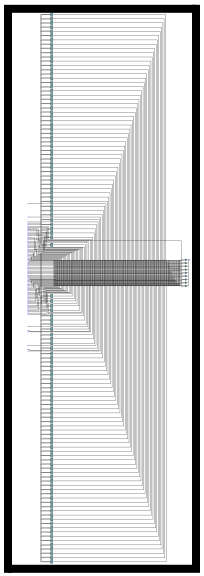
Functionality Description

The instruction module acts as READ-ONLY Memory storage unit for machine code instructions. The module is dependent on the program counter, which acts as an address pointer, which is utilized to fetch machine instructions. The overall size of the module is modifiable, and the overall design allows the process to sequential or selectively access its instruction set, facilitating program execution.

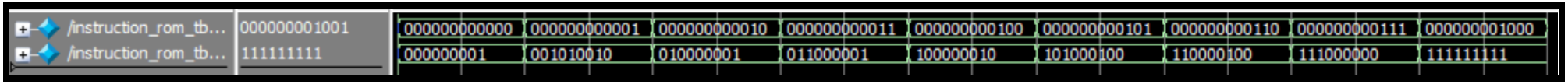
Testbench Description

The instruction_rom_tb is designed to validate theinstruction module, as we have a working assembler. It primarily focuses on evaluating how the module responds to different program counter inputs and retrieves the corresponding machine code instructions.

Schematic



Timing Diagram




```
# Program Counter | Machine Code
#
# ----------------|-------------
#
# PC     0:       | 000000001
#
# PC     1:       | 001010010
#
# PC     2:       | 010000001
#
# PC     3:       | 011000001
#
# PC     4:       | 100000010
#
# PC     5:       | 101000100
#
# PC     6:       | 110000100
#
# PC     7:       | 111000000
#
# PC     8:       | 111111111
```

Control Decoder

Module file name: **control.sv**
Module testbench file name: **control_tb.sv**

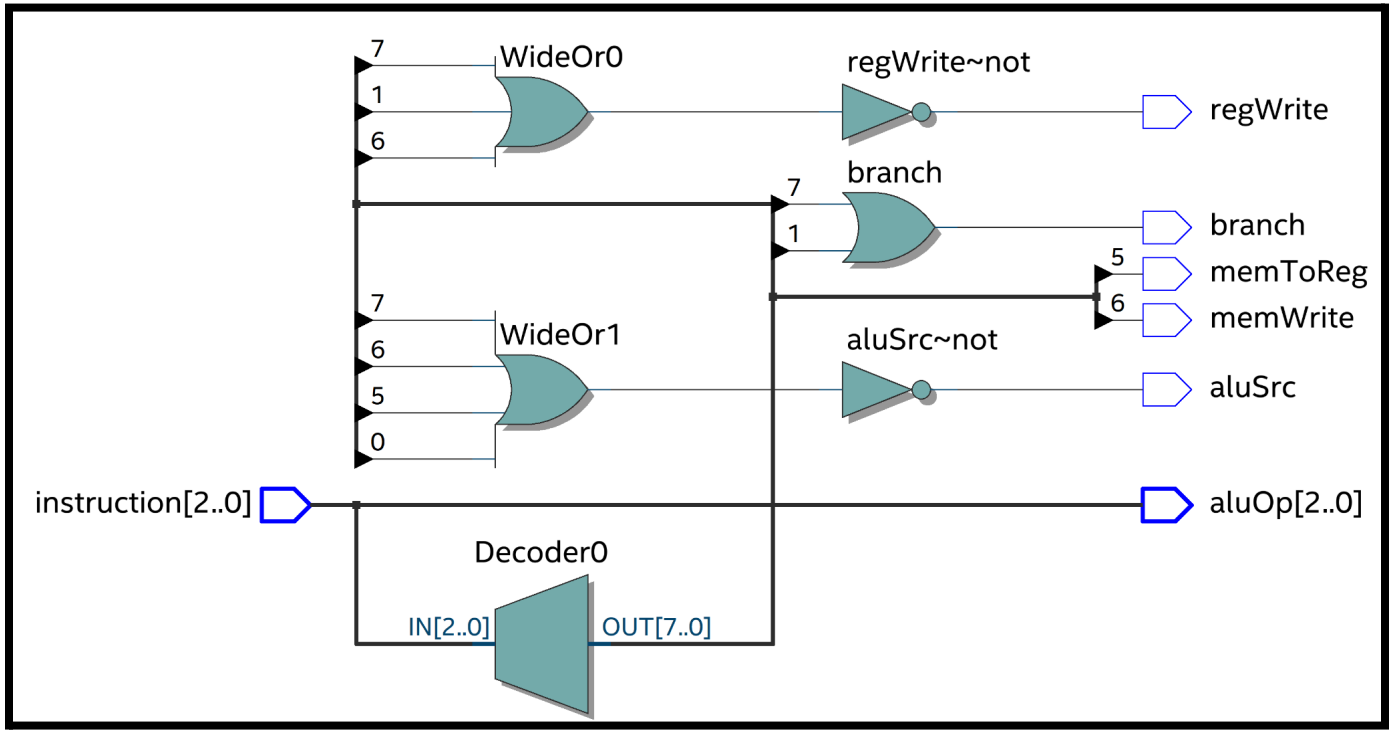
Functionality Description

The control module acts as the central decision-making unit for a custom instruction set, translating a given subset of machine code into relevant control signal for the processors data path. Upon receiving the instruction, it discerns the type of operation and produces corresponding control outputs like branch, memToReg, memWrite and so on. Default control signals are set initially, but are modified based on the type of instruction that would be done.

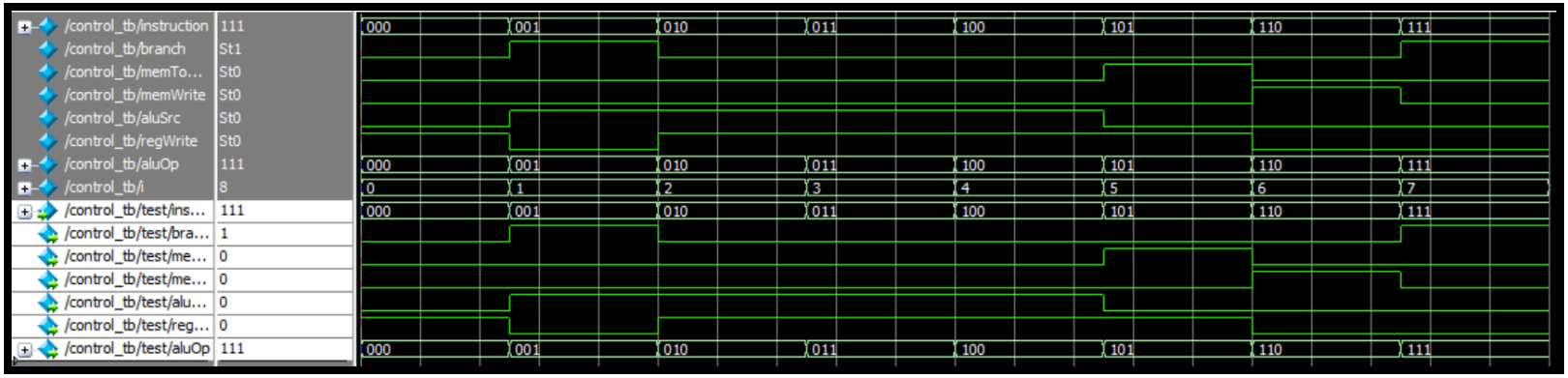
Testbench Description

The control_tb evaluates the control module by iterating through all possible instruction inputs, applying each to the module, and observing the resulting control signals. For each instruction, the testbench displays the instruction value and its corresponding control outputs for clarity. The simulation concludes once all instructions have been assessed.

Schematic



Timing Diagram




```
# Testing control unit...
# ----------------------------------------------------------
# Instruction: 000
#
#       branch: 0
#       memToReg: 0
#       memWrite: 0
#       aluSrc: 0
#       regWrite: 1
#       aluOp: 000
#
# ----------------------------------------------------------
# ----------------------------------------------------------
# Instruction: 001
#
#       branch: 1
#       memToReg: 0
#       memWrite: 0
#       aluSrc: 1
#       regWrite: 0
#       aluOp: 001
#
# ----------------------------------------------------------
# ----------------------------------------------------------
# Instruction: 010
#
#       branch: 0
#       memToReg: 0
#       memWrite: 0
#       aluSrc: 1
#       regWrite: 1
#       aluOp: 010
#
# ----------------------------------------------------------
# ----------------------------------------------------------
# Instruction: 011
#
#       branch: 0
#       memToReg: 0
#       memWrite: 0
#       aluSrc: 1
#       regWrite: 1
#       aluOp: 011
#
# ----------------------------------------------------------
# ----------------------------------------------------------
```

```
# ----------------------------------------------------------
# ----------------------------------------------------------
# Instruction: 100
#
#       branch: 0
#       memToReg: 0
#       memWrite: 0
#       aluSrc: 1
#       regWrite: 1
#       aluOp: 100
#
# ----------------------------------------------------------
# ----------------------------------------------------------
# Instruction: 101
#
#       branch: 0
#       memToReg: 1
#       memWrite: 0
#       aluSrc: 0
#       regWrite: 1
#       aluOp: 101
#
# ----------------------------------------------------------
# ----------------------------------------------------------
# Instruction: 110
#
#       branch: 0
#       memToReg: 0
#       memWrite: 1
#       aluSrc: 0
#       regWrite: 0
#       aluOp: 110
#
# ----------------------------------------------------------
# ----------------------------------------------------------
# Instruction: 111
#
#       branch: 1
#       memToReg: 0
#       memWrite: 0
#       aluSrc: 0
#       regWrite: 0
#       aluOp: 111
#
# ----------------------------------------------------------
```

Register File

Module file name: **regfile.sv**
Module testbench file name: **regfile_tb.sv**

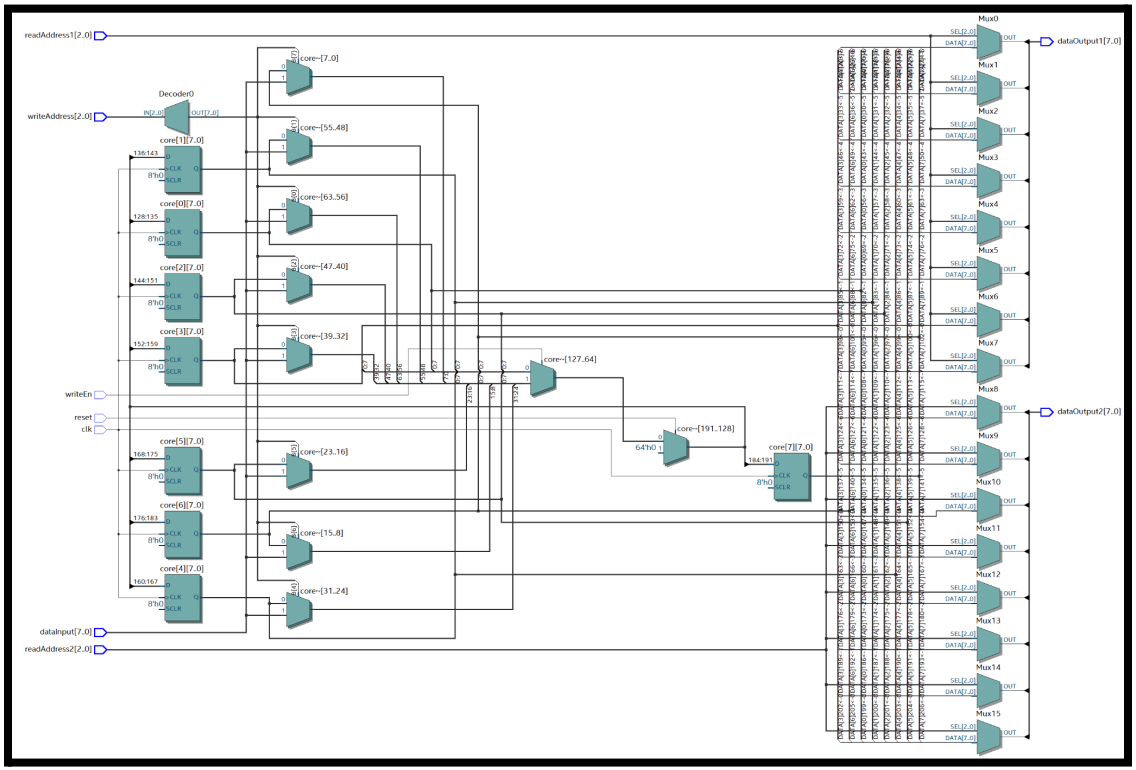
Functionality Description

The register file module represents an 8-register file where each register is 8-bits wide. Given read addresses, it provides the corresponding register values through data output ports within its design. It has the capability to write onto a register if the write enabler is high on the positive edge of the clock. A reset signal is also used to clear out the register file by zeroing out each register entirely.

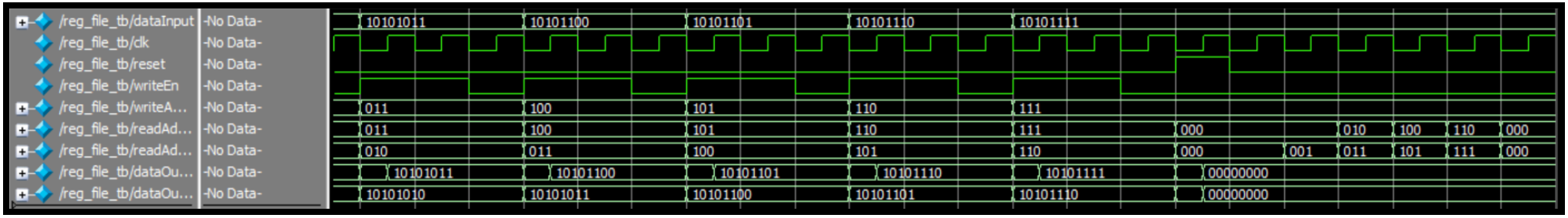
Testbench Description

The reg_file_tb evaluates the reg_file module by first verifying that all registers are cleared upon reset, then sequentially writing and reading from each register to ensure correct data storage, and finally, re-validating that all registers are cleared after another reset. Discrepancies between expected and actual register values trigger error messages, while successful checks result in a "PASS" message.

Schematic



Timing Diagram



ALU (Arithmetic Logic Unit)

Module file name: **alu.sv**
Module testbench file name: **alu_tb.sv**

Functionality Description

The alu module is an 8-bit arithmetic logic unit (ALU) that performs various arithmetic operations based on a 3-bit instruction it receives from other relevant components of our CPU. The module overall produces an 8-bit result and a zero flag that signals if the result is zero in terms of it receiving a beq instruction. If it is provided an unsupported instruction or logical shift value, the module defaults to specific predefined outputs.

Testbench Description

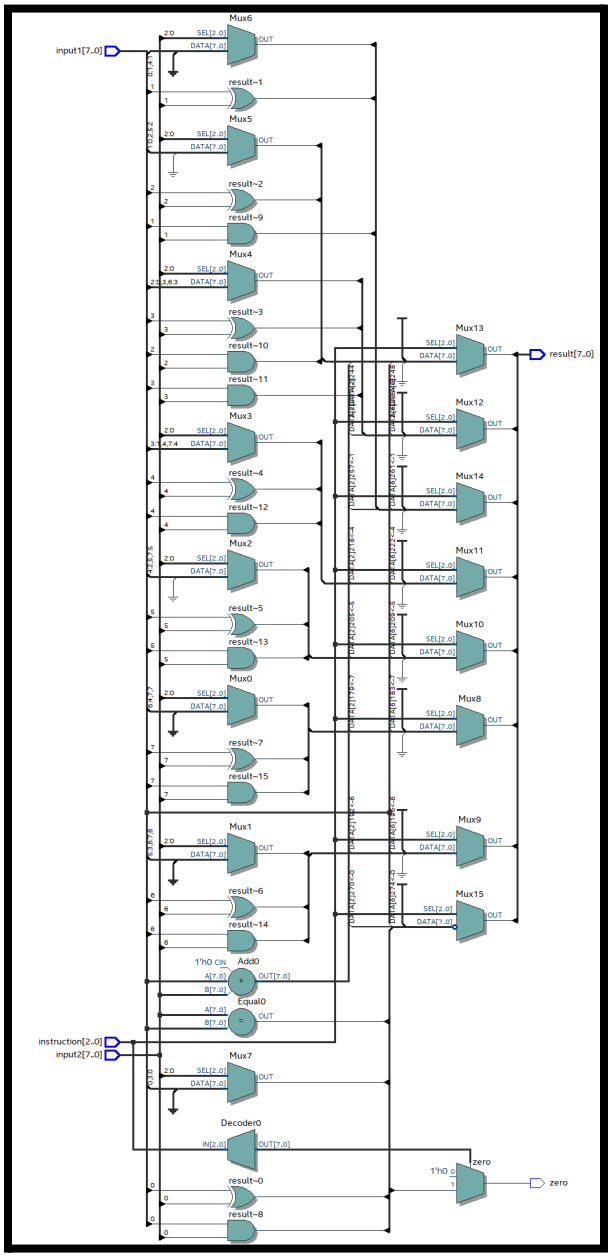
The alu_tb simulates the functionality of the alu module by providing it with a range of instructions and input values. It tests all supported ALU operations such as XOR, BEQ, ADD, AND, and logical shifts, and also verifies the behavior for unsupported instructions. The results are observed after waiting 100 time units between each operation.

ALU Operations

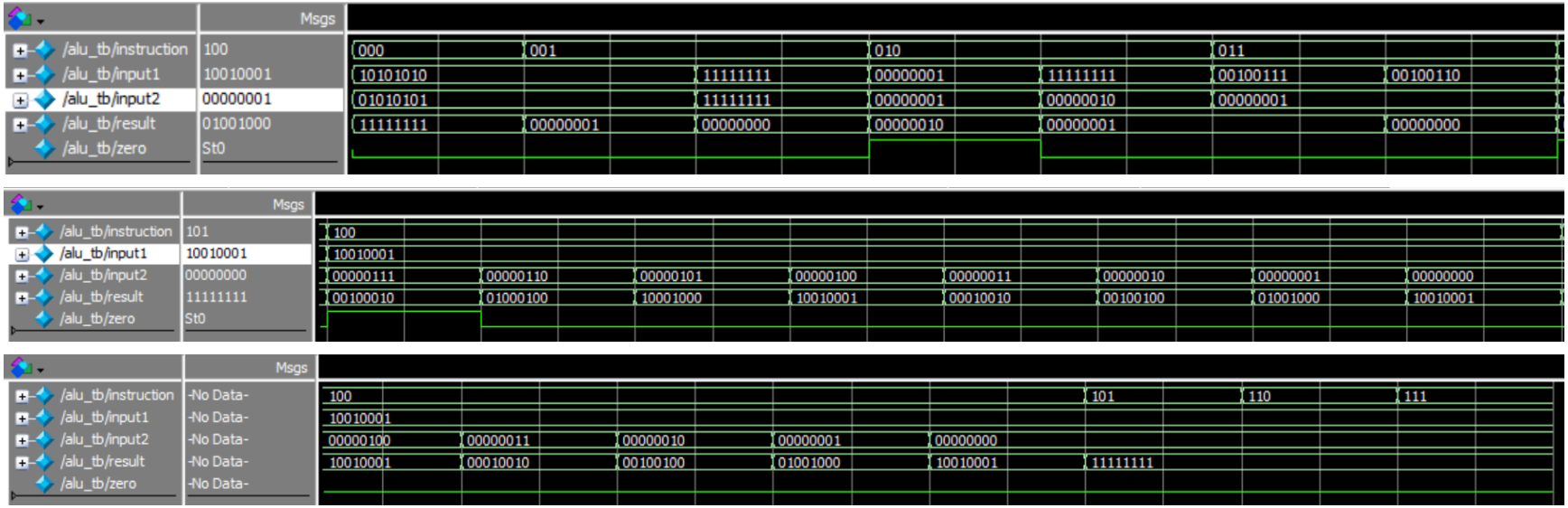
1. Bitwise XOR: This operation performs an EXCLUSIVE OR between two inputs. It is relevant to the instruction 3'b000.
2. Branch if Equal (BEQ): This operation checks if two inputs are equal and sets the result to 8'b00000000 if true, or 8'b00000001 if false. It is relevant to the instruction 3'b001.

3. Add Immediate (ADDI): This operation adds the values of two inputs together. It is relevant to the instruction 3'b010.
4. Bitwise AND (ANDI): This operation performs a bitwise AND between two inputs. It is relevant to the instruction 3'b011.
5. Logical Shift (LS): Depending on the three least significant bits of the second input it recieves, this operation either shifts the first input left or right by 1, 2, or 3 bits. Unsupported shifts result in the output being the same as the first input recieved. It is relevant to the instruction 3'b100.

Schematic



Timing Diagram



Data Memory

Module file name: **data_mem.sv**
Module testbench file name: **data_mem.sv**

Functionality Description

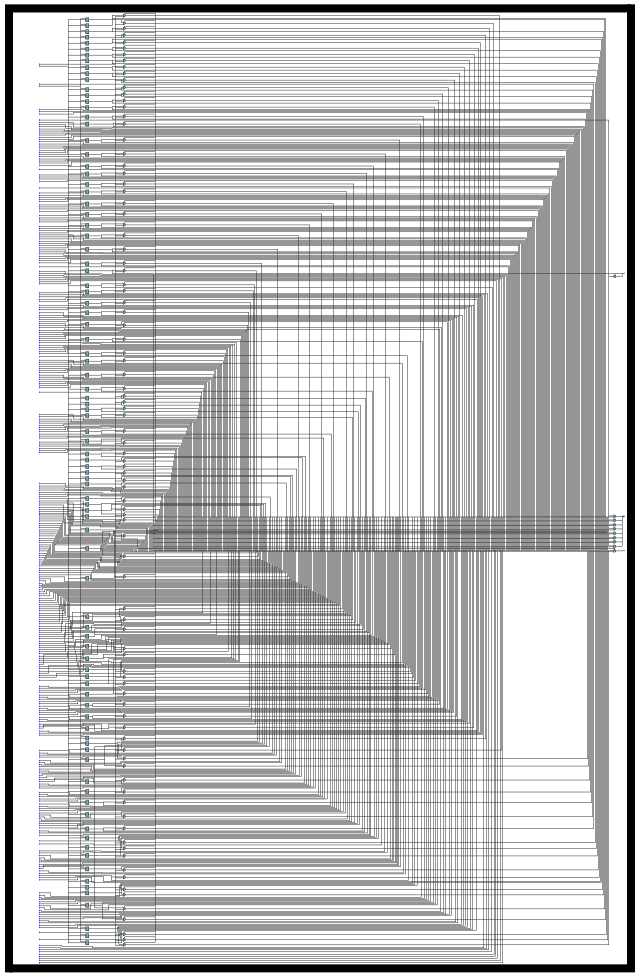
The data memory module represents an 8-bit wide, 256-word deep memory array. It allows combinational reads using the address input it is given, with the data being outputted as “data output”. Sequential writes are triggered by the positive edge of the clock and the write enabler. If the reset signal is high, then it resets the entire memory space with zeros. The memory is addressed using an 8-bit wide address, which accesses all 256 memory locations without any issues.

Testbench Description

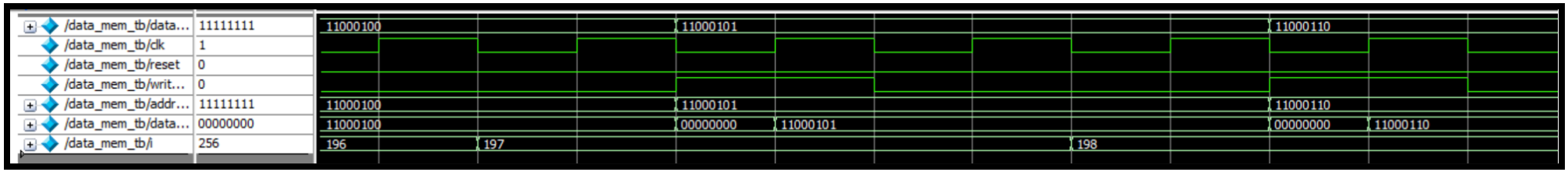
The data_mem_tb module verifies the functionality of the data_mem memory module. The test sequence follows these steps:

1. Initialization: All memory locations are reset to zero.
2. Verification: It checks that after the reset, all memory addresses indeed hold a value of 8'h00.
3. Memory Initialization: It writes values to each memory location, with the address being equal to the data.
4. Verification: After writing, it ensures that each memory location holds the correct value.
5. Clearing: All memory locations are reset again.
6. Verification: It checks again that all memory addresses hold a value of 8'h00 after the reset.
7. The testbench outputs messages indicating whether the memory functions as expected or if there are errors.

Schematic



Timing Diagram



Lookup Tables **(WE DO NOT USE ANY LOOKUP TABLES)**

Muxes (Multiplexers) : **ALL MUXES ARE BUILT INTO THE LOGIC.** No individual Mux modules available.

6. Program Implementation

Program 1 Source Code

File Name: hamming.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_BIT_LENGTH 11  // Define the maximum length for the binary string
#define NUM_PARITIES 5     // Define the number of parities

// Function declarations
void promptBinaryString(char *binaryString, int size);
```

```
int validBinaryString(char *binaryString);
void clearInputBuffer(void);
int binaryStringToBinary(char *binaryString);
void calculateParities(unsigned int binaryNumber, char *p0, char *p1, char *p2,
                       char *p4, char *p8);

// Main function
int main(void) {
    // Variable declarations for parity bits and binary string
    char p0, p1, p2, p4, p8;
    char binaryString[MAX_BIT_LENGTH + 1];
    int arraySize = sizeof(binaryString);

    // Prompting the user for a binary string
    promptBinaryString(binaryString, arraySize);

    // Loop until the user types "quit"
    while (strcmp(binaryString, "quit")) {
        // Validate the binary string
        if (!validBinaryString(binaryString)) {
            // If not valid, prompt again
            promptBinaryString(binaryString, arraySize);
            continue;
        } else {
            // Convert binary string to binary number
            unsigned int binaryNumber = binaryStringToBinary(binaryString);

            // Calculate the parity bits
            calculateParities(binaryNumber, &p0, &p1, &p2, &p4, &p8);

            // Check if parity bits are calculated properly
            if (p0 == '\0' || p1 == '\0' || p2 == '\0' || p4 == '\0' ||
```

```
            p8 == '\0') {
            // Error message and exit if parities are not calculated
            printf("ERROR: Parities not Calculated. Please try again.\n");
            return EXIT_FAILURE;
        }

        // Create the Hamming code string with the parity bits
        char hammingString[17];
        for (int i = 0; i < MAX_BIT_LENGTH - 4; i++) {
            hammingString[i] = binaryString[i];
        }
        hammingString[7] = p8;
        hammingString[8] = binaryString[7];
        hammingString[9] = binaryString[8];
        hammingString[10] = binaryString[9];
        hammingString[11] = p4;
        hammingString[12] = binaryString[10];
        hammingString[13] = p2;
        hammingString[14] = p1;
        hammingString[15] = p0;
        hammingString[16] = '\0';
        char *hammingStringPtr = hammingString;

        // Print the Hamming code
        printf("Hamming Binary String: %s\n", hammingStringPtr);

        // Prompt for another binary string
        promptBinaryString(binaryString, arraySize);
    }
}

// Exit the program successfully
```

```
    return EXIT_SUCCESS;
}

// Function to prompt the user for a binary string
void promptBinaryString(char *binaryString, int size) {
    // Print prompt message
    printf("Enter an 11-bit binary string (or type quit): ");

    // Read the user input
    fgets(binaryString, size, stdin);

    // Clear the input buffer if the binary string is of maximum length, or
    // remove the newline character otherwise
    if (strcspn(binaryString, "\n") == MAX_BIT_LENGTH) {
        clearInputBuffer();  // Clear the buffer only if 11 characters are
                             // entered
    } else {
        binaryString[strcspn(binaryString, "\n")] =
            '\0';  // Removing newline character if it's present
    }
}

// Function to validate the binary string
int validBinaryString(char *binaryString) {
    // Check the length of the binary string
    if (strlen(binaryString) != MAX_BIT_LENGTH) {
        // Print error message if the length is not valid
        printf("The binary string is not 11 bits long, please try again...\n");
        return 0;
    } else {
        // Check each character of the binary string
        for (int i = 0; i < MAX_BIT_LENGTH; i++) {
```

```
            // If any character is not '0' or '1', it's not a valid binary
            // string
            if (binaryString[i] != '1' && binaryString[i] != '0') {
                printf(
                    "you did not enter a valid binary string! Please try "
                    "again...\n");
                return 0;
            }
        }
        // Return 1 if the binary string is valid
        return 1;
    }
}

// Function to convert binary string to binary number
int binaryStringToBinary(char *binaryString) {
    unsigned int binaryNumber = 0;
    for (int i = 0; i < MAX_BIT_LENGTH; i++) {
        if (binaryString[i] == '1') {
            binaryNumber = (binaryNumber << 1) | 1;
        } else {
            binaryNumber = (binaryNumber << 1);
        }
    }
    return binaryNumber;
}

// Function to clear the input buffer
void clearInputBuffer() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF)
        ;  // Clear the input buffer
```

```
}

// Function to calculate the parity bits
void calculateParities(unsigned int binaryNumber, char *p0, char *p1, char *p2,
                       char *p4, char *p8) {
    int p0I = 0;
    int p1I = 0;
    int p2I = 0;
    int p4I = 0;
    int p8I = 0;
    int positionsP1[] = {11, 9, 7, 5, 4, 2, 1};
    int positionsP2[] = {11, 10, 7, 6, 4, 3, 1};
    int positionsP4[] = {11, 10, 9, 8, 4, 3, 2};
    int positionsP8[] = {11, 10, 9, 8, 7, 6, 5};
    int size = sizeof(positionsP1) / sizeof(positionsP1[0]);
    for (int i = 0; i < size; i++) {
        p1I ^= (binaryNumber >> (positionsP1[i] - 1)) & 1;
        p2I ^= (binaryNumber >> (positionsP2[i] - 1)) & 1;
        p4I ^= (binaryNumber >> (positionsP4[i] - 1)) & 1;
        p8I ^= (binaryNumber >> (positionsP8[i] - 1)) & 1;
    }
    *p1 = (p1I == 1 ? '1' : (p1I == 0 ? '0' : '\0'));
    *p2 = (p2I == 1 ? '1' : (p2I == 0 ? '0' : '\0'));
    *p4 = (p4I == 1 ? '1' : (p4I == 0 ? '0' : '\0'));
    *p8 = (p8I == 1 ? '1' : (p8I == 0 ? '0' : '\0'));
    for (int i = 0; i < MAX_BIT_LENGTH; i++) {
        p0I ^= (binaryNumber >> (i)) & 1;
    }
    p0I ^= p1I ^ p2I ^ p4I ^ p8I;
    *p0 = (p0I == 1 ? '1' : (p0I == 0 ? '0' : '\0'));
}
```

Program 1 Assembly Code

```
xor r0 r0 // this is a no op
addi r0 1 // r0 now has        0000 0001
ls r0 -1  // r0 now has        0000 0010
addi r0 1 // r0 now has        0000 0011
ls r0 -1  // r0 now has        0000 0110
addi r0 1 // r0 now has        0000 0111
ls r0 -2  // r0 now has        0001 1100
addi r0 1 // r0 now has        0001 1101 // desired result for r1 is 29  0001 1101
addi r1 1 // r1 now has        0000 0001
ls r1 -1  // r1 now has        0000 0010
addi r1 1 // r1 now has        0000 0011
ls r1 -1  // r1 now has        0000 0110
addi r1 1 // r1 now has        0000 0111
ls r1 -2  // r1 now has        0001 1100
addi r1 1 // r1 now has        0001 1101
ls r1 -1  // r1 now has        0011 1010
addi r1 1 // r1 now has        0011 1011  // desired result for r1 is 59  0011 1011
ld r2 r0  // r2 now has the contents of DM[29]
ls r2 2     // r2 now has 0000 000X... aka bit position 11
xor r4 r2   // r4 now has 0000 000X... aka bit position 11
xor r5 r2   // r5 now has 0000 000X... aka bit position 11
xor r6 r2   // r6 now has 0000 000X... aka bit position 11
xor r7 r2   // r7 now has 0000 000X... aka bit position 11
ld r3 r0    // r3 now has the contents of DM[29]
ls r3 1     // r3 now has the 10th bit as its lsb
andi r3 1   // isolate that lsb
xor r2 r3   // r2 is now b11^b10
xor r4 r3   // r4 is now b11^b10
xor r5 r3   // r5 is now b11^b10
xor r7 r3   // r7 is now b11^b10
```

```
ld r3 r0    // r3 now has the contents of DM[29]
andi r3 1   // isolate that lsb. r3 has the 9th bit
xor r2 r3   // r2 is now b11^b10^b9
xor r4 r3   // r4 is now b11^b10^b9
xor r6 r3   // r6 is now b11^b9
xor r7 r3   // r7 is now b11^b10^b9
addi r0 -1  // r0 now has the value 28
ld r3 r0    // r3 now has the contents of DM[28]
ls r3 3
ls r3 3
ls r3 1     // the msb should now be the lsb : r3 now has the 8th bit
xor r2 r3   // r2 is now b11^b10^b9^b8
xor r4 r3   // r4 is now b11^b10^b9^b8
xor r7 r3   // r7 is now b11^b10^b9^b8
ld r3 r0    // r3 now has the contents of DM[28]
ls r3 3
ls r3 3     // r3 now has the 7th bit positon
andi r3 1   // isolate that lsb
xor r2 r3   // r2 is now b11^b10^b9^b8^b7
xor r5 r3   // r5 is now b11^b10^b7
xor r6 r3   // r6 is now b11^b9^b7
xor r7 r3   // r7 is now b11^b10^b9^b8^b7
ld r3 r0    // r3 now has the contents of DM[28]
ls r3 3
ls r3 2     // r3 now has the 6th bit positon
andi r3 1   // isolate that lsb
xor r2 r3   // r2 is now b11^b10^b9^b8^b7^b6
xor r5 r3   // r5 is now b11^b10^b7^b6
xor r7 r3   // r7 is now b11^b10^b9^b8^b7^b6
ld r3 r0    // r3 now has the contents of DM[28]
ls r3 3
ls r3 1     // r3 now has the 5th bit positon
```

```
andi r3 1   // isolate that lsb
xor r2 r3   // r2 is now b11^b10^b9^b8^b7^b6^b5 ****** PARITY 8 COMPLETE!! ******
xor r6 r3   // r6 is now b11^b9^b7^b5
xor r7 r3   // r7 is now b11^b10^b9^b8^b7^b6^b5
ld r3 r0    // r3 now has the contents of DM[28]
ls r3 3     // r3 now has the 4th bit positon
andi r3 1   // isolate that lsb
xor r4 r3   // r4 is now b11^b10^b9^b8^b4
xor r5 r3   // r5 is now b11^b10^b7^b6^b4
xor r6 r3   // r6 is now b11^b9^b7^b5^b4
xor r7 r3   // r7 is now b11^b10^b9^b8^b7^b6^b5^b4
ld r3 r0    // r3 now has the contents of DM[28]
ls r3 2     // r3 now has the 3rd bit positon
andi r3 1   // isolate that lsb
xor r4 r3   // r4 is now b11^b10^b9^b8^b4^b3
xor r5 r3   // r5 is now b11^b10^b7^b6^b4^b3
xor r7 r3   // r7 is now b11^b10^b9^b8^b7^b6^b5^b4^b3
ld r3 r0    // r3 now has the contents of DM[28]
ls r3 1     // r3 now has the 2nd bit positon
andi r3 1   // isolate that lsb
xor r4 r3   // r4 is now b11^b10^b9^b8^b4^b3^b2 ****** PARITY 4 COMPLETE!! ******
xor r6 r3   // r6 is now b11^b9^b7^b5^b4^b2
xor r7 r3   // r7 is now b11^b10^b9^b8^b7^b6^b5^b4^b3^b2
ld r3 r0    // r3 now has the contents of DM[28]
andi r3 1   // isolate that lsb. r3 has the 1st bit
xor r5 r3   // r5 is now b11^b10^b7^b6^b4^b3^b1 ****** PARITY 2 COMPLETE!! ******
xor r6 r3   // r6 is now b11^b9^b7^b5^b4^b2^b1 ****** PARITY 1 COMPLETE!! ******
xor r7 r3   // r7 is now b11^b10^b9^b8^b7^b6^b5^b4^b3^b2^b1
addi r0 1   // r0 now has 29 again
xor r7 r2   //register 7 has all the xored bits ^ r2 = r2
xor r7 r4   // bits ^ r2 ^ r4
xor r7 r5   // bits ^ r2 ^ r4 ^ r5
```

```
xor r7 r6   // r7 = bits ^ r2 ^ r4 ^ r5 ^ r6 ****** PARITY 0 COMPLETE!! ******
st r7 r1    // store this parity bit at M[r1]
ld r3 r0    // r3 now has the contents of DM[29]
ls r3 -1    // r3 is now 0000 XXX0
addi r0 -1  // r0 now has the value 28
ld r7 r0    // r7 now has the contents of DM[28]
ls r7 3
ls r7 3
ls r7 1     // r7 now has the 8th bit
xor r3 r7   // r3 now has 0000 XXXX
ls r3 -1    // r3 is now 000X XXX0
ld r7 r0    // r7 now has the contents of DM[28]
ls r7 3
ls r7 3     // r7 now has the 7th bit
andi r7 1   // isolate that lsb
xor r3 r7   // r3 now has 000X XXXX
ls r3 -1    // r3 is now 00XX XXX0
ld r7 r0    // r7 now has the contents of DM[28]
ls r7 3
ls r7 2     // r7 now has the 6th bit
andi r7 1   // isolate that lsb
xor r3 r7   // r3 now has 00XX XXXX
ls r3 -1    // r3 is now 0XXX XXX0
ld r7 r0    // r7 now has the contents of DM[28]
ls r7 3
ls r7 1     // r7 now has the 5th bit
andi r7 1   // isolate that lsb
xor r3 r7   // r3 now has 0XXX XXXX
ls r3 -1    // r3 is now XXXX XXX0
xor r3 r2   // r3 is now B11 B10 B9 B8  B7 B6 B5 P8   *** FIRST BYTE COMPLETE ***
xor r2 r2   // clear the register 2 for extra register memory
ld r7 r1    // extract parity 0 from memory
```

```
st r3 r1    // store this string result into the M[r1] (59)
addi r1 -1  // reduce the memory address by 1. (want to store the next string at address 58)
xor r3 r3   // clear r3, it now has 0000 0000
ld r2 r0    // r2 now has the contents of DM[28]
ls r2 3     // r2 now has the 4th bit
andi r2 1   // isolate that lsb
xor r3 r2   // r3 now has 0000 000X
ls r3 -1    // r3 is now 0000 00X0
ld r2 r0    // r2 now has the contents of DM[28]
ls r2 2     // r2 now has the 3rd bit
andi r2 1   // isolate that lsb
xor r3 r2   // r3 now has 0000 00XX
ls r3 -1    // r3 is now 0000 0XX0
ld r2 r0    // r2 now has the contents of DM[28]
ls r2 1     // r2 now has the 2nd bit
andi r2 1   // isolate that lsb
xor r3 r2   // r3 now has 0000 0XXX
ls r3 -1    // r3 is now 0000 XXX0
xor r3 r4   // r3 is now 0000 b4b3b2p4
ls r3 -1    // r3 is now 000X XXX0
ld r2 r0    // r2 now has the contents of DM[28]
andi r2 1   // isolate that lsb
xor r3 r2   // 000b4 b3b2p4b1
ls r3 -1    // r3 is now 00XX XXX0
xor r3 r5   // r3 is now 00b4b3 b2p4b1p2
ls r3 -1
xor r3 r6   // r3 is now 0b4b3b2 p4b1p2p1
ls r3 -1
xor r3 r7   // r3 is now b4b3b2p4 b1p2p1p0
st r3 r1    // store this string result into the M[r1]
beq r0 0 10
addi r0 -1  // decrement the count of our core by 1...should be 27 next iteration
```

```
addi r1 -1  // we want to start storing things at one address down.
xor r2 r2
xor r3 r3
xor r4 r4
xor r5 r5
xor r6 r6
xor r7 r7
j 17
```

Program 1 Machine Code

```
000000000     // xor r0 r0
010000001     // addi r0 1
100000111     // ls r0 -1
010000001     // addi r0 1
100000111     // ls r0 -1
010000001     // addi r0 1
100000110     // ls r0 -2
010000001     // addi r0 1
010001001     // addi r1 1
100001111     // ls r1 -1
010001001     // addi r1 1
100001111     // ls r1 -1
010001001     // addi r1 1
100001110     // ls r1 -2
010001001     // addi r1 1
100001111     // ls r1 -1
010001001     // addi r1 1
101010000     // ld r2 r0
100010010     // ls r2 2
```

```
000100010    // xor r4 r2
000101010    // xor r5 r2
000110010    // xor r6 r2
000111010    // xor r7 r2
101011000    // ld r3 r0
100011001    // ls r3 1
011011001    // andi r3 1
000010011    // xor r2 r3
000100011    // xor r4 r3
000101011    // xor r5 r3
000111011    // xor r7 r3
101011000    // ld r3 r0
011011001    // andi r3 1
000010011    // xor r2 r3
000100011    // xor r4 r3
000110011    // xor r6 r3
000111011    // xor r7 r3
010000111    // addi r0 -1
101011000    // ld r3 r0
100011011    // ls r3 3
100011011    // ls r3 3
100011001    // ls r3 1
000010011    // xor r2 r3
000100011    // xor r4 r3
000111011    // xor r7 r3
101011000    // ld r3 r0
100011011    // ls r3 3
100011011    // ls r3 3
011011001    // andi r3 1
000010011    // xor r2 r3
000101011    // xor r5 r3
000110011    // xor r6 r3
```

```
000111011     // xor r7 r3
101011000     // ld r3 r0
100011011     // ls r3 3
100011010     // ls r3 2
011011001     // andi r3 1
000010011     // xor r2 r3
000101011     // xor r5 r3
000111011     // xor r7 r3
101011000     // ld r3 r0
100011011     // ls r3 3
100011001     // ls r3 1
011011001     // andi r3 1
000010011     // xor r2 r3
000110011     // xor r6 r3
000111011     // xor r7 r3
101011000     // ld r3 r0
100011011     // ls r3 3
011011001     // andi r3 1
000100011     // xor r4 r3
000101011     // xor r5 r3
000110011     // xor r6 r3
000111011     // xor r7 r3
101011000     // ld r3 r0
100011010     // ls r3 2
011011001     // andi r3 1
000100011     // xor r4 r3
000101011     // xor r5 r3
000111011     // xor r7 r3
101011000     // ld r3 r0
100011001     // ls r3 1
011011001     // andi r3 1
000100011     // xor r4 r3
```

```
000110011     // xor r6 r3
000111011     // xor r7 r3
101011000     // ld r3 r0
011011001     // andi r3 1
000101011     // xor r5 r3
000110011     // xor r6 r3
000111011     // xor r7 r3
010000001     // addi r0 1
000111010     // xor r7 r2
000111100     // xor r7 r4
000111101     // xor r7 r5
000111110     // xor r7 r6
110111001     // st r7 r1
101011000     // ld r3 r0
100011111     // ls r3 -1
010000111     // addi r0 -1
101111000     // ld r7 r0
100111011     // ls r7 3
100111011     // ls r7 3
100111001     // ls r7 1
000011111     // xor r3 r7
100011111     // ls r3 -1
101111000     // ld r7 r0
100111011     // ls r7 3
100111011     // ls r7 3
011111001     // andi r7 1
000011111     // xor r3 r7
100011111     // ls r3 -1
101111000     // ld r7 r0
100111011     // ls r7 3
100111010     // ls r7 2
011111001     // andi r7 1
```

```
000011111     // xor r3 r7
100011111     // ls r3 -1
101111000     // ld r7 r0
100111011     // ls r7 3
100111001     // ls r7 1
011111001     // andi r7 1
000011111     // xor r3 r7
100011111     // ls r3 -1
000011010     // xor r3 r2
000010010     // xor r2 r2
101111001     // ld r7 r1
110011001     // st r3 r1
010001111     // addi r1 -1
000011011     // xor r3 r3
101010000     // ld r2 r0
100010011     // ls r2 3
011010001     // andi r2 1
000011010     // xor r3 r2
100011111     // ls r3 -1
101010000     // ld r2 r0
100010010     // ls r2 2
011010001     // andi r2 1
000011010     // xor r3 r2
100011111     // ls r3 -1
101010000     // ld r2 r0
100010001     // ls r2 1
011010001     // andi r2 1
000011010     // xor r3 r2
100011111     // ls r3 -1
000011100     // xor r3 r4
100011111     // ls r3 -1
101010000     // ld r2 r0
```

```
011010001     // andi r2 1
000011010     // xor r3 r2
100011111     // ls r3 -1
000011101     // xor r3 r5
100011111     // ls r3 -1
000011110     // xor r3 r6
100011111     // ls r3 -1
000011111     // xor r3 r7
110011001     // st r3 r1
001001010     // beq r0 0 10
010000111     // addi r0 -1
010001111     // addi r1 -1
000010010     // xor r2 r2
000011011     // xor r3 r3
000100100     // xor r4 r4
000101101     // xor r5 r5
000110110     // xor r6 r6
000111111     // xor r7 r7
111010001     // j 17
```

Program 2 Source Code

File Name: hammingErrorCorrection.c

NOTE THE RESULT STRING IS 17 IN LENGTH, TO LEAVE SPACE FOR THE NULL TERMINATOR IN C.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_BIT_LENGTH 16
void promptBinaryString(char *binaryString, int size);
int validBinaryString(char *binaryString);
void clearInputBuffer(void);
```

```
void detectHammingError(char *binaryString);
int main(void){
    // Variable declarations for parity bits and binary string
    char binaryString[MAX_BIT_LENGTH + 1];
    int arraySize = sizeof(binaryString);
    promptBinaryString(binaryString, arraySize);
    while(strcmp(binaryString, "quit")){
        if(!validBinaryString(binaryString)){
            promptBinaryString(binaryString, arraySize);
        } else {
            // compare the parity bits
            detectHammingError(binaryString);
            promptBinaryString(binaryString, arraySize);
        }
    }
    return EXIT_SUCCESS;
}
// Function to prompt the user for a binary string
void promptBinaryString(char *binaryString, int size) {
    // Print prompt message
    printf("Enter a 16-bit hamming binary string to detect potential errors (or type quit): ");
    // Read the user input
    fgets(binaryString, size, stdin);
    // Clear the input buffer if the binary string is of maximum length, or
    // remove the newline character otherwise
    if (strcspn(binaryString, "\n") == MAX_BIT_LENGTH) {
        clearInputBuffer();  // Clear the buffer only if 16 characters are
                             // entered
    } else {
        binaryString[strcspn(binaryString, "\n")] =
            '\0';  // Removing newline character if it's present
    }
```

```
}
// Function to validate the binary string
int validBinaryString(char *binaryString) {
    // Check the length of the binary string
    if (strlen(binaryString) != MAX_BIT_LENGTH) {
        // Print error message if the length is not valid
        printf("The hamming binary string is not 16 bits long, please try again...\n");
        return 0;
    } else {
        // Check each character of the binary string
        for (int i = 0; i < MAX_BIT_LENGTH; i++) {
            // If any character is not '0' or '1', it's not a valid binary
            // string
            if (binaryString[i] != '1' && binaryString[i] != '0') {
                printf(
                    "you did not enter a valid hamming binary string! Please try "
                    "again...\n");
                return 0;
            }
        }
        // Return 1 if the binary string is valid
        return 1;
    }
}
// Function to clear the input buffer
void clearInputBuffer() {
    int c;
    while ((c = getchar()) != '\n' && c != EOF);  // Clear the input buffer
}
void detectHammingError(char *binaryString){
    int p0IsBad = 0;
    unsigned int incorrectPosition = 0;
```

```
int p1 = 0;
int p2 = 0;
int p4 = 0;
int p8 = 0;
// calculate the parity of p0
for (int i = MAX_BIT_LENGTH - 1; i > -1; i--) {
    int digit = (binaryString[i] == '1') ? 1 : 0;
    p0IsBad ^= digit;
}
int positionsP1[] = {14, 12, 10, 8, 6, 4, 2, 0};
int positionsP2[] = {13, 12, 9, 8, 5, 4, 1, 0};
int positionsP4[] = {11, 10, 9, 8, 3, 2, 1, 0}; // good
int positionsP8[] = {7, 6, 5, 4, 3, 2, 1, 0}; // good
int size = sizeof(positionsP1) / sizeof(positionsP1[0]);
for (int i = 0; i < size; i++) {
    int digit1 = (binaryString[positionsP1[i]] == '1') ? 1 : 0;
    int digit2 = (binaryString[positionsP2[i]] == '1') ? 1 : 0;
    int digit4 = (binaryString[positionsP4[i]] == '1') ? 1 : 0;
    int digit8 = (binaryString[positionsP8[i]] == '1') ? 1 : 0;
    p1 ^= digit1;
    p2 ^= digit2;
    p4 ^= digit4;
    p8 ^= digit8;
}
printf("p8p4p2p1: %d%d%d%d\n", p8, p4, p2, p1);
incorrectPosition = (p8); // 0000 00X0
incorrectPosition = (incorrectPosition << 1);
incorrectPosition |= (p4);
incorrectPosition = (incorrectPosition << 1);
incorrectPosition |= (p2);
incorrectPosition = (incorrectPosition << 1);
incorrectPosition |= (p1);
```

```
    // if p0 looks good, we have no errors, or a two bit error
    if(!p0IsBad){
        // if p0 looks good, and P8P4P2P1 is zero, we have no error
        if(incorrectPosition == 0){
            char noneParityString[12] = {binaryString[0], binaryString[1], binaryString[2], binaryString[3], binaryString[4], binaryString[5],
binaryString[6], binaryString[8], binaryString[9], binaryString[10], binaryString[12], '\0'};
            printf("No Errors Detected: ");
            char resultString[17] = "00000";
            strcat(resultString, noneParityString);
            printf("%s\n", resultString);
        // else we have a two bit error
        } else {
            char noneParityString[12] = {binaryString[0], binaryString[1], binaryString[2], binaryString[3], binaryString[4], binaryString[5],
binaryString[6], binaryString[8], binaryString[9], binaryString[10], binaryString[12], '\0'};
            printf("Two Bit Error Detected: ");
            char resultString[17] = "10000";
            strcat(resultString, noneParityString);
            printf("%s\n", resultString);
        }
    // p0 looks bad, time to determine if p0, or something else
    } else {
        // p0 is bad: THEN WE HAVE A 1 BIT ERROR
        printf("One Bit Error Detected: ");
        printf("%s\n", binaryString);
        printf("bit position: %i: \n", incorrectPosition);
        incorrectPosition = 15 - incorrectPosition;
        char resultString[17] = "01000";
        binaryString[incorrectPosition] = (binaryString[incorrectPosition] == '1') ? '0': '1';
        char noneParityString[12] = {binaryString[0], binaryString[1], binaryString[2], binaryString[3], binaryString[4], binaryString[5],
binaryString[6], binaryString[8], binaryString[9], binaryString[10], binaryString[12], '\0'};
        strcat(resultString, noneParityString);
        printf("Corrected String: %s\n", resultString);
```

```
    }
}
```

Program 2 Assembly Code

```
xor r0 r0 // have a NOP to keep the CPU busy
addi r6 2 // 0000 0010
addi r6 2 // 0000 0100
addi r6 2 // 0000 0110
addi r6 1 // 0000 0111
ls r6 -3  // 0011 1000
addi r6 2 // 0011 1010
addi r6 1 // 0011 1011 --> 59, r6 will be used for loading
addi r7 2 // 0000 0010
addi r7 2 // 0000 0100
addi r7 2 // 0000 0110
addi r7 1 // 0000 0111
ls r7 -2  // 0001 1100
addi r7 1 // 0001 1101 --> 29, r7 will be used for storing
xor r0 r0   // this register will hold our p0 test
xor r1 r1   // this register will hold our p8p4p2p1 result
xor r2 r2
xor r3 r3
xor r4 r4
xor r5 r5   // clear all registers, but our count registers.
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
andi r2 1   // isolate p8
andi r3 1   // isolate p0
xor r0 r2   // p8
```

```
xor r0 r3   // p0 ^ p8
xor r1 r2   // p8
addi r6 1   // 59
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 1     // current top byte 0_d11_d10_d9 d8_d7_d6_d5
ls r3 1     // current bottom byte 0_d4_d3_d2 p4_d1_p2_p1
andi r2 1   // isolate d5
andi r3 1   // isolate p1
xor r0 r2   // p0 ^ p8 ^ d5
xor r0 r3   // p0 ^ p1 ^ p8 ^ d5
xor r1 r2   // p8 ^ d5
addi r6 1   // 59
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 2     // current top byte 0_0_d11_d10_d9 d8_d7_d6
ls r3 2     // current bottom byte 0_0_d4_d3_d2 p4_d1_p2
andi r2 1   // isolate d6
andi r3 1   // isolate p2
xor r0 r2   // p0 ^ p1 ^ p8 ^ d5 ^ d6
xor r0 r3   // p0 ^ p1 ^ p2 ^ p8 ^ d5 ^ d6
xor r1 r2   // p8 ^ d5 ^ d6
xor r5 r2   // d6
xor r5 r3   // p2 ^ d6
addi r6 1   // 59
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 3     // current top byte    0_0_0_d11 d10_d9_d8_d7
ls r3 3     // current bottom byte 0_0_0_d4  d3_d2_p4_d1
```

```
andi r2 1   // isolate d7
andi r3 1   // isolate d1
xor r0 r2   // p0 ^ p1 ^ p2 ^ p8 ^ d5 ^ d6 ^ d7
xor r0 r3   // p0 ^ p1 ^ p2 ^ p8 ^ d1 ^ d5 ^ d6 ^ d7
xor r1 r2   // p8 ^ d5 ^ d6 ^ d7
xor r5 r2   // p2 ^ d6 ^ d7
xor r5 r3   // p2 ^ d1 ^ d6 ^ d7
addi r6 1   // 59
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 3
ls r2 1     // current top byte    0_0_0_0   d11_d10_d9_d8
ls r3 3
ls r3 1     // current bottom byte 0_0_0_0   d4_d3_d2_p4
andi r2 1   // isolate d8
andi r3 1   // isolate p4
xor r0 r2   // p0 ^ p1 ^ p2 ^ p8 ^ d1 ^ d5 ^ d6 ^ d7 ^ d8
xor r0 r3   // p0 ^ p1 ^ p2 ^ p4 ^ p8 ^ d1 ^ d5 ^ d6 ^ d7 ^ d8
xor r1 r2   // p8 ^ d5 ^ d6 ^ d7 ^ d8
xor r4 r2   // d8
xor r4 r3   // p4 ^ d8
addi r6 1   // 59
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 3
ls r2 2     // current top byte    0_0_0_0 0_d11_d10_d9
ls r3 3
ls r3 2     // current bottom byte 0_0_0_0 0_d4_d3_d2
andi r2 1   // isolate d9
andi r3 1   // isolate d2
```

```
xor r0 r2   // p0 ^ p1 ^ p2 ^ p4 ^ p8 ^ d1 ^ d5 ^ d6 ^ d7 ^ d8 ^ d9
xor r0 r3   // p0 ^ p1 ^ p2 ^ p4 ^ p8 ^ d1 ^ d2 ^ d5 ^ d6 ^ d7 ^ d8 ^ d9
xor r1 r2   // p8 ^ d5 ^ d6 ^ d7 ^ d8 ^ d9
xor r4 r2   // p4 ^ d8 ^ d9
xor r4 r3   // p4 ^ d2 ^ d8 ^ d9
addi r6 1   // 59
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 3
ls r2 3     // current top byte    0_0_0_0 0_0_d11_d10
ls r3 3
ls r3 3     // current bottom byte 0_0_0_0 0_0_d4_d3
andi r2 1   // isolate d10
andi r3 1   // isolate d3
xor r0 r2   // p0 ^ p1 ^ p2 ^ p4 ^ p8 ^ d1 ^ d2 ^ d5 ^ d6 ^ d7 ^ d8 ^ d9 ^ d10
xor r0 r3   // p0 ^ p1 ^ p2 ^ p4 ^ p8 ^ d1 ^ d2 ^ d3 ^ d5 ^ d6 ^ d7 ^ d8 ^ d9 ^ d10
xor r1 r2   // p8 ^ d5 ^ d6 ^ d7 ^ d8 ^ d9 ^ d10
xor r5 r2   // p2 ^ d1 ^ d6 ^ d7 ^ d10
xor r5 r3   // p2 ^ d1 ^ d3 ^ d6 ^ d7 ^ d10
xor r4 r2   // p4 ^ d2 ^ d8 ^ d9 ^ d10
xor r4 r3   // p4 ^ d2 ^ d3 ^ d8 ^ d9 ^ d10
addi r6 1   // 59
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 3
ls r2 3
ls r2 1     // current top byte    0_0_0_0 0_0_0_d11
ls r3 3
ls r3 3
ls r3 1     // current bottom byte 0_0_0_0 0_0_0_d4
```

```
andi r2 1   // isolate d11
andi r3 1   // isolate d4
xor r0 r2   // p0 ^ p1 ^ p2 ^ p4 ^ p8 ^ d1 ^ d2 ^ d3 ^ d5 ^ d6 ^ d7 ^ d8 ^ d9 ^ d10 ^ d11
xor r0 r3   // p0 ^ p1 ^ p2 ^ p4 ^ p8 ^ d1 ^ d2 ^ d3 ^ d4 ^ d5 ^ d6 ^ d7 ^ d8 ^ d9 ^ d10 ^ d11  ** P0 COMPLETE ** In Register 0
xor r1 r2   // p8 ^ d5 ^ d6 ^ d7 ^ d8 ^ d9 ^ d10 ^ d11                                  ** P8 COMPLETE ** In Register 1
xor r4 r2   // p4 ^ d2 ^ d3 ^ d8 ^ d9 ^ d10 ^ d11
xor r4 r3   // p4 ^ d2 ^ d3 ^ d4 ^ d8 ^ d9 ^ d10 ^ d11                                  ** P4 COMPLETE ** In Register 4
xor r5 r2   // p2 ^ d1 ^ d3 ^ d6 ^ d7 ^ d10 ^ d11
xor r5 r3   // p2 ^ d1 ^ d3 ^ d4 ^ d6 ^ d7 ^ d10 ^ d11                                  ** P2 COMPLETE ** In Register 5
addi r6 1   // 59
ls r1 -1    // 0000 00p80
xor r1 r4   // 0000 00p8p4
ls r1 -1    // 0000 0p8p40
xor r1 r5   // 0000 0p8p4p2
ls r1 -1    // 0000 p8p4p20                                                  ** P8P4P2_0 ** In Register 1
xor r4 r4
xor r5 r5   // clear registers 4 and 5
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 1     // current top byte 0_d11_d10_d9 d8_d7_d6_d5
ls r3 1     // current bottom byte 0_d4_d3_d2 p4_d1_p2_p1
andi r2 1   // isolate d5
andi r3 1   // isolate p1
xor r4 r2   // d5
xor r4 r3   // p1 ^ d5
addi r6 1   // 59
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 3     // current top byte    0_0_0_d11 d10_d9_d8_d7
ls r3 3     // current bottom byte 0_0_0_d4  d3_d2_p4_d1
```

```
andi r2 1   // isolate d7
andi r3 1   // isolate d1
xor r4 r2   // p1 ^ d5 ^ d7
xor r4 r3   // p1 ^ d1 ^ d5 ^ d7
addi r6 1   // 59
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 3
ls r2 2     // current top byte    0_0_0_0 0_d11_d10_d9
ls r3 3
ls r3 2     // current bottom byte 0_0_0_0 0_d4_d3_d2
andi r2 1   // isolate d9
andi r3 1   // isolate d2
xor r4 r2   // p1 ^ d1 ^ d5 ^ d7 ^ d9
xor r4 r3   // p1 ^ d1 ^ d2 ^ d5 ^ d7 ^ d9
addi r6 1   // 59
ld r2 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r2 3
ls r2 3
ls r2 1     // current top byte    0_0_0_0 0_0_0_d11
ls r3 3
ls r3 3
ls r3 1     // current bottom byte 0_0_0_0 0_0_0_d4
andi r2 1   // isolate d11
andi r3 1   // isolate d4
xor r4 r2   // p1 ^ d1 ^ d2 ^ d5 ^ d7 ^ d9 ^ d11
xor r4 r3   // p1 ^ d1 ^ d2 ^ d4 ^ d5 ^ d7 ^ d9 ^ d11                    ** P1 COMPLETE ** In Register 4
xor r1 r4   // 0000 p8p4p2p1                                  ** P8P4P2P1 ** In Register 1
xor r4 r4   // clear register 4 -- r2, r3, r4, r5 are all available as of now
```

```
addi r6 1   // 59
xor r2 r2   // clear register 2
ld r3 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
ls r3 1     // current top byte 0_d11_d10_d9  d8_d7_d6_d5
ls r3 -3
ls r3 -1    // current top byte d8_d7_d6_d5    0_0_0_0
xor r2 r3   // register now contains d8_d7_d6_d5    0_0_0_0
addi r6 -1  // 58
ld r3 r6    // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r3 3
ls r3 2    // current bottom byte 0_0_0_0 0_d4_d3_d2
ls r3 -1   // current bottom byte 0_0_0_0 d4_d3_d2_0
xor r2 r3  // register now contains d8_d7_d6_d5    d4_d3_d2_0
ld r3 r6   // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r3 3    // current bottom byte 0_0_0_d4 d3_d2_p4_d1
andi r3 1  // isolate d1    --> 0000 000d1
xor r2 r3  // register now contains d8_d7_d6_d5    d4_d3_d2_d1
beq r0 0 7  // P0 TEST, IF LOW, NO ERROR OR TWO BIT ERROR, ELSE A ONE BIT ERROR
xor r0 r0 // no longer care about the result of p0, so zero it out
xor r2 r2
xor r3 r3
xor r4 r4
xor r5 r5 // clear all registers
j 63 // MAPPED TO LINE 237
xor r0 r0   // NO LONGER CARE ABOUT THE P0 PARITY
addi r6 1   // 59
addi r7 -1  // 28
st r2 r7    // store d8_d7_d6_d5    d4_d3_d2_d1 in M[r7]
addi r7 1   // 29
ld r3 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
ls r3 3
ls r3 2       // 0_0_0_0 0_d11_d10_d9
```

```
beq r1 0 12
addi r4 1 // 0000 0001
ls r4 -3
ls r4 -3
ls r4 -1    // 1000 0000
xor r3 r4   // r3 is now 1000 0d11d10d9
st r3 r7    // store 1000 0d11d10d9 in M[r7]
xor r0 r7   // extract the count from r7
beq r0 1 10
addi r6 -2 // 57
addi r7 -2 // 27
j 14       // jump back to the top of our procedure
st r3 r7   // store 0000 0d11d10d9 in M[r7]
xor r0 r7  // extract the count from r7
beq r0 1 4
addi r6 -2 // 57
addi r7 -2 // 27
j 14       // jump back to the top of our procedure
xor r0 r0  // ********************************************** THIS LINE TERMINATES THE PROGRAM
**********************************************
xor r0 r1  // r0 holds a copy of r1 ********************** THIS IS THE START OF OUR 1 BIT ERROR CHECK
*********************************************
ls r0 3   // r0 now holds only the value of p8.
addi r6 1   // 59
addi r2 1   // 0000 0001
ls r2 -3
ls r2 -3    // 0100 0000
beq r0 0 2  // if p8 is zero, then the error is in the lower byte
j 62 // MAPPED TO LINE 285
xor r0 r0
xor r0 r1   // should contain 0000 0XXX
ld r3 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
```

```
ls r3 3
ls r3 2     // load the current top byte 0_0_0_0 0_d11_d10_d9
xor r2 r3   // 0_1_0_0 0_d11_d10_d9
st r2 r7    // store 0100 0d11d10d9 in M[r7]
addi r7 -1  // 28
xor r2 r2
addi r2 1          // 0000 0001
beq r0 0 4         // check the error location
ls r2 -1           // ls once to the left
addi r0 -1         // deduct r0 by 1
j 61               // IS MAPPED TO PC - 3
ld r3 r6           // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
ls r3 1            // current top byte 0_d11_d10_d9 d8_d7_d6_d5
ls r3 -3
ls r3 -1           // d8_d7_d6_d5 0_0_0_0
addi r6 -1         // 58
ld r4 r6           // load the current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
xor r2 r4          // current bottom byte d4_d3_d2_p4 d1_p2_p1_p0 with corrected potential bit
xor r4 r4
xor r4 r2          // copy of corrected data
ls r4 3
ls r4 2            // 0_0_0_0 0_d4_d3_d2
ls r4 -1           // 0_0_0_0 d4_d3_d2_0
xor r3 r4          // d8_d7_d6_d5 d4_d3_d2_0
xor r4 r4
xor r4 r2          // copy of corrected data
ls r4 3            // 0_0_0_d4 d3_d2_p4_d1
andi r4 1          // isolate d1, which is the corrected bit potentially.
xor r3 r4          // d8_d7_d6_d5 d4_d3_d2_d1
st r3 r7
xor r0 r0
xor r0 r7
```

```
beq r0 0 4
addi r6 -1  // 57
addi r7 -1  // 27
j 14        // jump back to the top of our procedure
xor r0 r0   // ********************************************** THIS LINE TERMINATES THE PROGRAM
**********************************************
xor r0 r0   // r0 holds a copy of r1 ********************** THIS IS THE START OF OUR 1 BIT ERROR CHECK
*********************************************
addi r0 1   // 0000 0001
ls r0 -3    // 0000 1000
xor r0 r1   // should contain 0000 0XXX, as i got rid of p8's location
ld r3 r6    // load the current top byte d11_d10_d9_d8 d7_d6_d5_p8
addi r4 1   // 0000 0001
beq r0 0 4          // check the error location
ls r4 -1            // ls once to the left
addi r0 -1          // deduct r0 by 1
j 61                // IS MAPPED TO PC - 3
xor r3 r4           // r3 now contains the corrected bit d11_d10_d9_d8 d7_d6_d5_p8
xor r4 r4
xor r4 r3           // copy of the corrected data d11_d10_d9_d8 d7_d6_d5_p8
ls r4 3
ls r4 2             // copy of the corrected data 0_0_0_0 0_d11_d10_d9
xor r2 r4           // should now contain 0_1_0_0 0_d11_d10_d9
st r2 r7
addi r6 -1  // 58
addi r7 -1  // 28
ld r4 r6    // current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ld r5 r6    // current bottom byte d4_d3_d2_p4 d1_p2_p1_p0
ls r4 3
ls r4 2     // 0_0_0_0 0_d4_d3_d2
ls r4 -1    // 0_0_0_0 d4_d3_d2_0
ls r5 3     // 0_0_0_d4 d3_d2_p4_d1
```

```
andi r5 1   // isolate d1
xor r4 r5   // 0_0_0_0 d4_d3_d2_d1
ls r3 1     // 0_d11_d10_d9 d8_d7_d6_d5
ls r3 -3
ls r3 -1    // d8_d7_d6_d5 0_0_0_0
xor r3 r4   // d8_d7_d6_d5 d4_d3_d2_d1 POTENTIALLY CORRECTED DATA
st r3 r7    // store r3 in M[r7]
xor r0 r0
xor r0 r7
beq r0 0 4
addi r6 -1  // 57
addi r7 -1  // 27
j 14        // jump back to the top of our procedure
xor r0 r0   // ********************************************** THIS LINE TERMINATES THE PROGRAM
**********************************************
```

Program 2 Machine Code

```
000000000     // xor r0 r0
010110010     // addi r6 2
010110010     // addi r6 2
010110010     // addi r6 2
010110001     // addi r6 1
100110101     // ls r6 -3
010110010     // addi r6 2
010110001     // addi r6 1
010111010     // addi r7 2
010111010     // addi r7 2
010111010     // addi r7 2
010111001     // addi r7 1
100111110     // ls r7 -2
010111001     // addi r7 1
```

```
000000000    // xor r0 r0
000001001    // xor r1 r1
000010010    // xor r2 r2
000011011    // xor r3 r3
000100100    // xor r4 r4
000101101    // xor r5 r5
101010110    // ld r2 r6
010110111    // addi r6 -1
101011110    // ld r3 r6
011010001    // andi r2 1
011011001    // andi r3 1
000000010    // xor r0 r2
000000011    // xor r0 r3
000001010    // xor r1 r2
010110001    // addi r6 1
101010110    // ld r2 r6
010110111    // addi r6 -1
101011110    // ld r3 r6
100010001    // ls r2 1
100011001    // ls r3 1
011010001    // andi r2 1
011011001    // andi r3 1
000000010    // xor r0 r2
000000011    // xor r0 r3
000001010    // xor r1 r2
010110001    // addi r6 1
101010110    // ld r2 r6
010110111    // addi r6 -1
101011110    // ld r3 r6
100010010    // ls r2 2
100011010    // ls r3 2
011010001    // andi r2 1
```

```
011011001     // andi r3 1
000000010     // xor r0 r2
000000011     // xor r0 r3
000001010     // xor r1 r2
000101010     // xor r5 r2
000101011     // xor r5 r3
010110001     // addi r6 1
101010110     // ld r2 r6
010110111     // addi r6 -1
101011110     // ld r3 r6
100010011     // ls r2 3
100011011     // ls r3 3
011010001     // andi r2 1
011011001     // andi r3 1
000000010     // xor r0 r2
000000011     // xor r0 r3
000001010     // xor r1 r2
000101010     // xor r5 r2
000101011     // xor r5 r3
010110001     // addi r6 1
101010110     // ld r2 r6
010110111     // addi r6 -1
101011110     // ld r3 r6
100010011     // ls r2 3
100010001     // ls r2 1
100011011     // ls r3 3
100011001     // ls r3 1
011010001     // andi r2 1
011011001     // andi r3 1
000000010     // xor r0 r2
000000011     // xor r0 r3
000001010     // xor r1 r2
```

```
000100010    // xor r4 r2
000100011    // xor r4 r3
010110001    // addi r6 1
101010110    // ld r2 r6
010110111    // addi r6 -1
101011110    // ld r3 r6
100010011    // ls r2 3
100010010    // ls r2 2
100011011    // ls r3 3
100011010    // ls r3 2
011010001    // andi r2 1
011011001    // andi r3 1
000000010    // xor r0 r2
000000011    // xor r0 r3
000001010    // xor r1 r2
000100010    // xor r4 r2
000100011    // xor r4 r3
010110001    // addi r6 1
101010110    // ld r2 r6
010110111    // addi r6 -1
101011110    // ld r3 r6
100010011    // ls r2 3
100010011    // ls r2 3
100011011    // ls r3 3
100011011    // ls r3 3
011010001    // andi r2 1
011011001    // andi r3 1
000000010    // xor r0 r2
000000011    // xor r0 r3
000001010    // xor r1 r2
000101010    // xor r5 r2
000101011    // xor r5 r3
```

```
000100010    // xor r4 r2
000100011    // xor r4 r3
010110001    // addi r6 1
101010110    // ld r2 r6
010110111    // addi r6 -1
101011110    // ld r3 r6
100010011    // ls r2 3
100010011    // ls r2 3
100010001    // ls r2 1
100011011    // ls r3 3
100011011    // ls r3 3
100011001    // ls r3 1
011010001    // andi r2 1
011011001    // andi r3 1
000000010    // xor r0 r2
000000011    // xor r0 r3
000001010    // xor r1 r2
000100010    // xor r4 r2
000100011    // xor r4 r3
000101010    // xor r5 r2
000101011    // xor r5 r3
010110001    // addi r6 1
100001111    // ls r1 -1
000001100    // xor r1 r4
100001111    // ls r1 -1
000001101    // xor r1 r5
100001111    // ls r1 -1
000100100    // xor r4 r4
000101101    // xor r5 r5
101010110    // ld r2 r6
010110111    // addi r6 -1
101011110    // ld r3 r6
```

```
100010001    // ls r2 1
100011001    // ls r3 1
011010001    // andi r2 1
011011001    // andi r3 1
000100010    // xor r4 r2
000100011    // xor r4 r3
010110001    // addi r6 1
101010110    // ld r2 r6
010110111    // addi r6 -1
101011110    // ld r3 r6
100010011    // ls r2 3
100011011    // ls r3 3
011010001    // andi r2 1
011011001    // andi r3 1
000100010    // xor r4 r2
000100011    // xor r4 r3
010110001    // addi r6 1
101010110    // ld r2 r6
010110111    // addi r6 -1
101011110    // ld r3 r6
100010011    // ls r2 3
100010010    // ls r2 2
100011011    // ls r3 3
100011010    // ls r3 2
011010001    // andi r2 1
011011001    // andi r3 1
000100010    // xor r4 r2
000100011    // xor r4 r3
010110001    // addi r6 1
101010110    // ld r2 r6
010110111    // addi r6 -1
101011110    // ld r3 r6
```

```
100010011    // ls r2 3
100010011    // ls r2 3
100010001    // ls r2 1
100011011    // ls r3 3
100011011    // ls r3 3
100011001    // ls r3 1
011010001    // andi r2 1
011011001    // andi r3 1
000100010    // xor r4 r2
000100011    // xor r4 r3
000001100    // xor r1 r4
000100100    // xor r4 r4
010110001    // addi r6 1
000010010    // xor r2 r2
101011110    // ld r3 r6
100011001    // ls r3 1
100011101    // ls r3 -3
100011111    // ls r3 -1
000010011    // xor r2 r3
010110111    // addi r6 -1
101011110    // ld r3 r6
100011011    // ls r3 3
100011010    // ls r3 2
100011111    // ls r3 -1
000010011    // xor r2 r3
101011110    // ld r3 r6
100011011    // ls r3 3
011011001    // andi r3 1
000010011    // xor r2 r3
001000111    // beq r0 0 7
000000000    // xor r0 r0
000010010    // xor r2 r2
```

```
000011011    // xor r3 r3
000100100    // xor r4 r4
000101101    // xor r5 r5
111111111    // j 63
000000000    // xor r0 r0
010110001    // addi r6 1
010111111    // addi r7 -1
110010111    // st r2 r7
010111001    // addi r7 1
101011110    // ld r3 r6
100011011    // ls r3 3
100011010    // ls r3 2
001101100    // beq r1 0 12
010100001    // addi r4 1
100100101    // ls r4 -3
100100101    // ls r4 -3
100100111    // ls r4 -1
000011100    // xor r3 r4
110011111    // st r3 r7
000000111    // xor r0 r7
001011010    // beq r0 1 10
010110110    // addi r6 -2
010111110    // addi r7 -2
111001110    // j 14
110011111    // st r3 r7
000000111    // xor r0 r7
001010100    // beq r0 1 4
010110110    // addi r6 -2
010111110    // addi r7 -2
111001110    // j 14
000000000    // xor r0 r0
000000001    // xor r0 r1
```

```
100000011    // ls r0 3
010110001    // addi r6 1
010010001    // addi r2 1
100010101    // ls r2 -3
100010101    // ls r2 -3
001000010    // beq r0 0 2
111111110    // j 62
000000000    // xor r0 r0
000000001    // xor r0 r1
101011110    // ld r3 r6
100011011    // ls r3 3
100011010    // ls r3 2
000010011    // xor r2 r3
110010111    // st r2 r7
010111111    // addi r7 -1
000010010    // xor r2 r2
010010001    // addi r2 1
001000100    // beq r0 0 4
100010111    // ls r2 -1
010000111    // addi r0 -1
111111101    // j 61
101011110    // ld r3 r6
100011001    // ls r3 1
100011101    // ls r3 -3
100011111    // ls r3 -1
010110111    // addi r6 -1
101100110    // ld r4 r6
000010100    // xor r2 r4
000100100    // xor r4 r4
000100010    // xor r4 r2
100100011    // ls r4 3
100100010    // ls r4 2
```

```
100100111    // ls r4 -1
000011100    // xor r3 r4
000100100    // xor r4 r4
000100010    // xor r4 r2
100100011    // ls r4 3
011100001    // andi r4 1
000011100    // xor r3 r4
110011111    // st r3 r7
000000000    // xor r0 r0
000000111    // xor r0 r7
001000100    // beq r0 0 4
010110111    // addi r6 -1
010111111    // addi r7 -1
111001110    // j 14
000000000    // xor r0 r0
000000000    // xor r0 r0
010000001    // addi r0 1
100000101    // ls r0 -3
000000001    // xor r0 r1
101011110    // ld r3 r6
010100001    // addi r4 1
001000100    // beq r0 0 4
100100111    // ls r4 -1
010000111    // addi r0 -1
111111101    // j 61
000011100    // xor r3 r4
000100100    // xor r4 r4
000100011    // xor r4 r3
100100011    // ls r4 3
100100010    // ls r4 2
000010100    // xor r2 r4
110010111    // st r2 r7
```

```
010110111     // addi r6 -1
010111111     // addi r7 -1
101100110     // ld r4 r6
101101110     // ld r5 r6
100100011     // ls r4 3
100100010     // ls r4 2
100100111     // ls r4 -1
100101011     // ls r5 3
011101001     // andi r5 1
000100101     // xor r4 r5
100011001     // ls r3 1
100011101     // ls r3 -3
100011111     // ls r3 -1
000011100     // xor r3 r4
110011111     // st r3 r7
000000000     // xor r0 r0
000000111     // xor r0 r7
001000100     // beq r0 0 4
010110111     // addi r6 -1
010111111     // addi r7 -1
111001110     // j 14
000000000     // xor r0 r0
```

Program 3 Source Code

File Name: whereIsWaldo.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define PATTERN_SIZE 5
#define BINARY_SIZE 256
```

```
#define EIGHT 8
#define THIRTY_TWO 32
#define FOUR 4
#define FIVE 5
#define MAX_ITERATIONS 252

int countPattern(char *bitPattern, char *binaryString, int bitCount);
int calcNumBytesInWhichPatternOccurs(char *bitPattern, char *binaryString, int bitCount);
int main(int argc, char **argv){
    char bitPattern[PATTERN_SIZE + 1];
    char binaryString[BINARY_SIZE + 1];
    if(argc != 3){
        printf("This program requires 2 arguements: a 5 bit pattern, and a 256 bit binary string.\n");
    } else {
        int bitPatternSize = strlen(*(argv + 1));
        int binaryStringSize = strlen(*(argv + 2));
        if(bitPatternSize != PATTERN_SIZE || binaryStringSize != BINARY_SIZE){
            printf("This program requires 2 arguements: a 5 bit pattern, and a 256 bit binary string.\n");
            return EXIT_FAILURE;
        } else {
            for(int i = 0; i < bitPatternSize; i++){
                if(*(*(argv + 1) + i) != '0' && *(*(argv + 1) + i) != '1'){
                    printf("The 5 bit pattern arguement is not a binary string! Please try again!\n");
                return EXIT_FAILURE;
                } else {
                    bitPattern[i] = *(*(argv + 1) + i);
                }
            }
            for(int i = 0; i < binaryStringSize; i++){
                if(*(*(argv + 2) + i) != '0' && *(*(argv + 2) + i) != '1'){
                    printf("The binary string arguement is not a binary string! Please try again!\n");
                    return EXIT_FAILURE;
```

```
                } else {
                    binaryString[i] = *(*(argv + 2) + i);
                }
            }
        }
    }
    int patternCount1 = countPattern(bitPattern, binaryString, EIGHT);
    int numBytes = calcNumBytesInWhichPatternOccurs(bitPattern, binaryString, EIGHT);
    int patternCount2 = countPattern(bitPattern, binaryString, THIRTY_TWO);
    printf("Number of bytes within which the pattern occurs: %d\n", patternCount1);
    printf("Number of bytes that contain the pattern: %d\n", numBytes);
    printf("Total number of times which the pattern occurs overall: %d\n", patternCount2);
    return EXIT_SUCCESS;
}

int countPattern(char *bitPattern, char *binaryString, int bitsToCount){
    int patternCount = 0;
    int numIterations = 0;
    if(bitsToCount == 8){
        while(numIterations < THIRTY_TWO){
            for(int i = numIterations * bitsToCount; i < (numIterations * bitsToCount) + FOUR; i++){
                int currentBit = 0;
                for(int j = i; j < i + FIVE; j++){
                    if(*(bitPattern + currentBit) != *(binaryString + j)){
                        break;
                    } else {
                        currentBit++;
                    }
                }
                if(currentBit == FIVE){
                    patternCount++;
                }
```

```
            }
            numIterations++;
        }
    return patternCount;
  } else {
    while(numIterations < MAX_ITERATIONS){
          int currentBit = 0;
          for(int i = numIterations; i < numIterations + FIVE; i++){
                if(*(bitPattern + currentBit) != *(binaryString + i)){
              break;
          } else {
              currentBit++;
          }
          }
          if(currentBit == FIVE){
          patternCount++;
        }
          numIterations++;
        }
    return patternCount;
  }
}

int calcNumBytesInWhichPatternOccurs(char *bitPattern, char *binaryString, int bitsToCount){
  int numIterations = 0;
  int byteHasPattern = 0;
  int numBytes = 0;
  while(numIterations < THIRTY_TWO){
        for(int i = numIterations * bitsToCount; i < (numIterations * bitsToCount) + FOUR; i++){
          int currentBit = 0;
          for(int j = i; j < i + FIVE; j++){
            if(*(bitPattern + currentBit) != *(binaryString + j)){
```

```
                break;
            } else {
                currentBit++;
            }
        }
        if(currentBit == FIVE && !byteHasPattern){
                byteHasPattern = 1;
                numBytes++;
        }
      }
        byteHasPattern = 0;
      numIterations++;
    }
  return numBytes;
}
```

Program 3 Assembly Code

```
xor r0 r0            // have a nop at the start of the program
addi r7 1            // 0000 0001
ls r7 -3             // 0000 1000
ls r7 -2             // 0010 0000 -- r7 should contain 32, it will contain the location of our pattern. It will also serve as the pointers to store our information in DM[33-35]
xor r6 r7            // make a copy of r7 into r6
addi r6 -1           // r6 should contain 31, thus it will serve as our main load register for the string bytes
xor r4 r6            // R4 WILL BE A CONSTANT REGISTER TO STRICTLY HOLD THE VALUE 31 AND NOTHING MORE OR NOTHING LESS
ld r5 r7             // load the pattern into r5, this will be our constant register to hold our pattern: a 5-bit pattern in bits [7:3] of datamem[32] X_X_X_X X_0_0_0
ls r5 3              // The pattern is now in bits [4:0] 0_0_0_X X_X_X_X
addi r7 1            // r7 now holds the value 33. which find the number of times a pattern occurs in a byte data mem[33].
ld r0 r6             // r0 now holds our byte X_X_X_X X_X_X_X
```

```
ls r0 -3              // isolate bits [4:0]
ls r0 3               // 0004 3210
xor r0 r5             // check and see if the pattern matches
beq r0 0 2            // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                  // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first load instruction below, ensuring not to increment the pattern count
addi r3 1             // r3 will contain our count for the number of patterns we encounter: increment the pattern count
ld r0 r6              // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 -2              // isolate bits [5:1]
ls r0 3               // 0005 4321
xor r0 r5             // check and see if the pattern matches
beq r0 0 2            // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                  // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first load instruction below, ensuring not to increment the pattern count
addi r3 1             // r3 will contain our count for the number of patterns we encounter: increment the pattern count
ld r0 r6              // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 -1              // isolate bits [6:2]
ls r0 3               // 0006 5432
xor r0 r5             // check and see if the pattern matches
beq r0 0 2            // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                  // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first load instruction below, ensuring not to increment the pattern count
addi r3 1             // r3 will contain our count for the number of patterns we encounter: increment the pattern count
ld r0 r6              // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 3               // isolate bits [7:3]   0007 6543
xor r0 r5             // check and see if the pattern matches
beq r0 0 2            // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                  // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first decrement instruction below, ensuring not to increment the pattern count, but to decrement the memory address as we are done with one byte
addi r3 1             // r3 will contain our count for the number of patterns we encounter: increment the pattern count
xor r0 r0             // clear r0
xor r0 r6             // get a copy of the current memory count into r0
```

```
beq r0 0 3          // the count of memory addresses has reached 0, so we now move onto the next portion of our algorithm - the number of times our pattern occurs per byte
addi r6 -1          // we are down one more memory address
j 10                // jump back to the start of our procedure, as there are more bytes to process.
st r3 r7            // Enter the total number of occurrences of the given 5-bit pattern in any byte into data mem[33]
addi r7 1           // r7 now holds the value 34. The number of bytes within which the pattern occurs data mem[34].
xor r3 r3           // clear our pattern count register. register 0 and 6 should have 0000 0000
xor r6 r4           // go back to data mem[31].
ld r0 r6            // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 -3            // isolate bits [4:0]
ls r0 3             // 0004 3210
xor r0 r5           // check and see if the pattern matches
beq r0 0 2          // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 59                // this jump value maps to PC + 8  AKA the pattern did not match, so jump to the next check, ensuring not to increment the pattern count
addi r3 1           // r3 will contain our count for the number of bytes within which the pattern occurs: increment the pattern count
xor r0 r0           // clear the register r0
xor r0 r6           // get a copy of the current memory count into r0
beq r0 0 3          // if memory is 0, jump out of the loop
addi r6 -1          // we are down one more memory address, we no longer need to check the current byte
j 46                // go back to the top of the current procedure
j 58                // 58 is mapped to the our final and last procedure at line: 96
ld r0 r6            // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 -2            // isolate bits [5:1]
ls r0 3             // 0005 4321
xor r0 r5           // check and see if the pattern matches
beq r0 0 2          // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 59                // this jump value maps to PC + 8  AKA the pattern did not match, so jump to the next check, ensuring not to increment the pattern count
addi r3 1           // r3 will contain our count for the number of bytes within which the pattern occurs: increment the pattern count
xor r0 r0           // clear the register r0
xor r0 r6           // get a copy of the current memory count into r0
```

```
beq r0 0 3          // if memory is 0, jump out of the loop
addi r6 -1          // we are down one more memory address, we no longer need to check the current byte
j 46               // go back to the top of the current procedure
j 58               // 58 is mapped to the our final and last procedure at line: 96
ld r0 r6           // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 -1           // isolate bits [6:2]
ls r0 3            // 0006 5432
xor r0 r5          // check and see if the pattern matches
beq r0 0 2          // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 59               // this jump value maps to PC + 8  AKA the pattern did not match, so jump to the next check, ensuring not to
increment the pattern count
addi r3 1          // r3 will contain our count for the number of bytes within which the pattern occurs: increment the pattern count
xor r0 r0          // clear the register r0
xor r0 r6          // get a copy of the current memory count into r0
beq r0 0 3          // if memory is 0, jump out of the loop
addi r6 -1          // we are down one more memory address, we no longer need to check the current byte
j 46               // go back to the top of the current procedure
j 58               // 58 is mapped to the our final and last procedure at line: 96
ld r0 r6           // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 3            // isolate bits [7:3]   0007 6543
xor r0 r5          // check and see if the pattern matches
beq r0 0 2          // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 59               // this jump value maps to PC + 8  AKA the pattern did not match, so jump to the next check, ensuring not to
increment the pattern count
addi r3 1          // r3 will contain our count for the number of bytes within which the pattern occurs: increment the pattern count
xor r0 r0          // clear the register r0
xor r0 r6          // get a copy of the current memory count into r0
beq r0 0 3          // if memory is 0, jump out of the loop
addi r6 -1          // we are down one more memory address, we no longer need to check the current byte
j 46               // go back to the top of the current procedure
st r3 r7           // Enter the number of bytes within which the pattern occurs in data mem[34].
```

```
addi r7 1            // r7 now holds the value 35. The total number of times the pattern occurs anywhere in the 256 bit string data mem[35]
xor r3 r3            // clear our pattern count register. register 0 and 6 should have 0000 0000
xor r6 r4            // go back to data mem[31].
xor r4 r4            // we no longer need the memory count 31 to be stored in a constant register
ld r0 r6             // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 -3             // isolate bits [4:0]
ls r0 3              // 0004 3210
xor r0 r5            // check and see if the pattern matches
beq r0 0 2           // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                 // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first load instruction below, ensuring not to increment the pattern count
addi r3 1            // r3 will contain our count for the number of patterns we encounter: increment the pattern count
ld r0 r6             // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 -2             // isolate bits [5:1]
ls r0 3              // 0005 4321
xor r0 r5            // check and see if the pattern matches
beq r0 0 2           // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                 // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first load instruction below, ensuring not to increment the pattern count
addi r3 1            // r3 will contain our count for the number of patterns we encounter: increment the pattern count
ld r0 r6             // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 -1             // isolate bits [6:2]
ls r0 3              // 0006 5432
xor r0 r5            // check and see if the pattern matches
beq r0 0 2           // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                 // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first load instruction below, ensuring not to increment the pattern count
addi r3 1            // r3 will contain our count for the number of patterns we encounter: increment the pattern count
ld r0 r6             // r0 now holds our byte X_X_X_X X_X_X_X
ls r0 3              // isolate bits [7:3]   0007 6543
xor r0 r5            // check and see if the pattern matches
```

```
beq r0 0 2            // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                 // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first decrement instruction below,
ensuring not to increment the pattern count, but to decrement the memory address as we are done with one byte
addi r3 1             // r3 will contain our count for the number of patterns we encounter: increment the pattern count
xor r0 r0             // clear r0
xor r0 r6             // get a copy of the current memory count into r0
beq r0 0 2            // the count of memory addresses has reached 0, so store the pattern count into DM[35] and terminate the
program
j 57                 // this jump value maps to PC + 3 AKA the pattern did not match, so jump to the next part of our procdure
st r3 r7              // Store the total number of times the pattern occurs anywhere in the 256 bit string into data mem[35]
xor r2 r2             // ****************************************** THIS LINE TERMINATES THE PROGRAM COMPLETELY
*********************************************************************************
ld r0 r6              // r0 now holds our upper byte B_B_B_B X_X_X_X. WE WANT OUR B'S TO BE THE BOTTOM MOST BITS
addi r6 -1            // 30
ld r1 r6              // r1 now holds our bottom byte X_X_X_X X_X_X_B WE WANT OUR B'S TO BE THE UPPER MOST BITS
addi r6 1             // 31
ls r0 3
ls r0 1               // 0000 BBBB
andi r1 1             // 0000 000B
ls r1 -3
ls r1 -1              // 000B 0000
xor r0 r1             // 000B BBBB -- THIS IS NOW 5 BITS WE CAN COMPARE IN THIS CASE BITS [8:4]
xor r0 r5             // check and see if the pattern matches
beq r0 0 2            // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                 // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first load instruction below, ensuring
not to increment the pattern count
addi r3 1             // r3 will contain our count for the number of patterns we encounter: increment the pattern count
ld r0 r6              // r0 now holds our upper byte B_B_B_X X_X_X_X. WE WANT OUR B'S TO BE THE BOTTOM MOST BITS
addi r6 -1            // 30
ld r1 r6              // r1 now holds our bottom byte X_X_X_X X_X_B_B WE WANT OUR B'S TO BE THE UPPER MOST BITS
addi r6 1             // 31
ls r0 3
```

```
ls r0 2              // 0000 0BBB
ls r1 -3
ls r1 -3             // BB00 0000
ls r1 3              // 000B B000
xor r0 r1             // 000B BBBB -- THIS IS NOW 5 BITS WE CAN COMPARE IN THIS CASE BITS [9:5]
xor r0 r5             // check and see if the pattern matches
beq r0 0 2             // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                 // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first load instruction below, ensuring
not to increment the pattern count
addi r3 1             // r3 will contain our count for the number of patterns we encounter: increment the pattern count
ld r0 r6             // r0 now holds our upper byte B_B_X_X X_X_X_X. WE WANT OUR B'S TO BE THE BOTTOM MOST BITS
addi r6 -1             // 30
ld r1 r6             // r1 now holds our bottom byte X_X_X_X X_B_B_B WE WANT OUR B'S TO BE THE UPPER MOST BITS
addi r6 1             // 31
ls r0 3
ls r0 3              // 0000 00BB
ls r1 -3
ls r1 -2             // BBB0 0000
ls r1 3              // 000B BB00
xor r0 r1             // 000B BBBB -- THIS IS NOW 5 BITS WE CAN COMPARE IN THIS CASE BITS [10:6]
xor r0 r5             // check and see if the pattern matches
beq r0 0 2             // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                 // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first load instruction below, ensuring
not to increment the pattern count
addi r3 1             // r3 will contain our count for the number of patterns we encounter: increment the pattern count
ld r0 r6             // r0 now holds our upper byte B_X_X_X X_X_X_X. WE WANT OUR B'S TO BE THE BOTTOM MOST BITS
addi r6 -1             // 30
ld r1 r6             // r1 now holds our bottom byte X_X_X_X B_B_B_B WE WANT OUR B'S TO BE THE UPPER MOST BITS
addi r6 1             // 31
ls r0 3
ls r0 3
ls r0 1              // 0000 000B
```

```
ls r1 -3
ls r1 -1              // BBBB 0000
ls r1 3               // 000B BBB0
xor r0 r1             // 000B BBBB -- THIS IS NOW 5 BITS WE CAN COMPARE IN THIS CASE BITS [11:7]
xor r0 r5             // check and see if the pattern matches
beq r0 0 2            // if the pattern matches, then the result should be 0 and increment the pattern count by 1
j 60                  // this jump value maps to PC + 2 AKA the pattern did not match, so jump to the first load instruction below, ensuring
not to increment the pattern count
addi r3 1             // r3 will contain our count for the number of patterns we encounter: increment the pattern count
addi r6 -1            // decrement the count no matter what
j 56                  // mapped to line 101
```

Program 3 Machine Code

```
000000000     // xor r0 r0
010111001     // addi r7 1
100111101     // ls r7 -3
100111110     // ls r7 -2
000110111     // xor r6 r7
010110111     // addi r6 -1
000100110     // xor r4 r6
101101111     // ld r5 r7
100101011     // ls r5 3
010111001     // addi r7 1
101000110     // ld r0 r6
100000101     // ls r0 -3
100000011     // ls r0 3
000000101     // xor r0 r5
001000010     // beq r0 0 2
111111100     // j 60
010011001     // addi r3 1
```

```
101000110    // ld r0 r6
100000110    // ls r0 -2
100000011    // ls r0 3
000000101    // xor r0 r5
001000010    // beq r0 0 2
111111100    // j 60
010011001    // addi r3 1
101000110    // ld r0 r6
100000111    // ls r0 -1
100000011    // ls r0 3
000000101    // xor r0 r5
001000010    // beq r0 0 2
111111100    // j 60
010011001    // addi r3 1
101000110    // ld r0 r6
100000011    // ls r0 3
000000101    // xor r0 r5
001000010    // beq r0 0 2
111111100    // j 60
010011001    // addi r3 1
000000000    // xor r0 r0
000000110    // xor r0 r6
001000011    // beq r0 0 3
010110111    // addi r6 -1
111001010    // j 10
110011111    // st r3 r7
010111001    // addi r7 1
000011011    // xor r3 r3
000110100    // xor r6 r4
101000110    // ld r0 r6
100000101    // ls r0 -3
100000011    // ls r0 3
```

```
000000101    // xor r0 r5
001000010    // beq r0 0 2
111111011    // j 59
010011001    // addi r3 1
000000000    // xor r0 r0
000000110    // xor r0 r6
001000011    // beq r0 0 3
010110111    // addi r6 -1
111101110    // j 46
111111010    // j 58
101000110    // ld r0 r6
100000110    // ls r0 -2
100000011    // ls r0 3
000000101    // xor r0 r5
001000010    // beq r0 0 2
111111011    // j 59
010011001    // addi r3 1
000000000    // xor r0 r0
000000110    // xor r0 r6
001000011    // beq r0 0 3
010110111    // addi r6 -1
111101110    // j 46
111111010    // j 58
101000110    // ld r0 r6
100000111    // ls r0 -1
100000011    // ls r0 3
000000101    // xor r0 r5
001000010    // beq r0 0 2
111111011    // j 59
010011001    // addi r3 1
000000000    // xor r0 r0
000000110    // xor r0 r6
```

```
001000011     // beq r0 0 3
010110111     // addi r6 -1
111101110     // j 46
111111010     // j 58
101000110     // ld r0 r6
100000011     // ls r0 3
000000101     // xor r0 r5
001000010     // beq r0 0 2
111111011     // j 59
010011001     // addi r3 1
000000000     // xor r0 r0
000000110     // xor r0 r6
001000011     // beq r0 0 3
010110111     // addi r6 -1
111101110     // j 46
110011111     // st r3 r7
010111001     // addi r7 1
000011011     // xor r3 r3
000110100     // xor r6 r4
000100100     // xor r4 r4
101000110     // ld r0 r6
100000101     // ls r0 -3
100000011     // ls r0 3
000000101     // xor r0 r5
001000010     // beq r0 0 2
111111100     // j 60
010011001     // addi r3 1
101000110     // ld r0 r6
100000110     // ls r0 -2
100000011     // ls r0 3
000000101     // xor r0 r5
001000010     // beq r0 0 2
```

```
111111100     // j 60
010011001     // addi r3 1
101000110     // ld r0 r6
100000111     // ls r0 -1
100000011     // ls r0 3
000000101     // xor r0 r5
001000010     // beq r0 0 2
111111100     // j 60
010011001     // addi r3 1
101000110     // ld r0 r6
100000011     // ls r0 3
000000101     // xor r0 r5
001000010     // beq r0 0 2
111111100     // j 60
010011001     // addi r3 1
000000000     // xor r0 r0
000000110     // xor r0 r6
001000010     // beq r0 0 2
111111001     // j 57
110011111     // st r3 r7
000010010     // xor r2 r2
101000110     // ld r0 r6
010110111     // addi r6 -1
101001110     // ld r1 r6
010110001     // addi r6 1
100000011     // ls r0 3
100000001     // ls r0 1
011001001     // andi r1 1
100001101     // ls r1 -3
100001111     // ls r1 -1
000000001     // xor r0 r1
000000101     // xor r0 r5
```

```
001000010    // beq r0 0 2
111111100    // j 60
010011001    // addi r3 1
101000110    // ld r0 r6
010110111    // addi r6 -1
101001110    // ld r1 r6
010110001    // addi r6 1
100000011    // ls r0 3
100000010    // ls r0 2
100001101    // ls r1 -3
100001101    // ls r1 -3
100001011    // ls r1 3
000000001    // xor r0 r1
000000101    // xor r0 r5
001000010    // beq r0 0 2
111111100    // j 60
010011001    // addi r3 1
101000110    // ld r0 r6
010110111    // addi r6 -1
101001110    // ld r1 r6
010110001    // addi r6 1
100000011    // ls r0 3
100000011    // ls r0 3
100001101    // ls r1 -3
100001110    // ls r1 -2
100001011    // ls r1 3
000000001    // xor r0 r1
000000101    // xor r0 r5
001000010    // beq r0 0 2
111111100    // j 60
010011001    // addi r3 1
101000110    // ld r0 r6
```

```
010110111      // addi r6 -1
101001110      // ld r1 r6
010110001      // addi r6 1
100000011      // ls r0 3
100000011      // ls r0 3
100000001      // ls r0 1
100001101      // ls r1 -3
100001111      // ls r1 -1
100001011      // ls r1 3
000000001      // xor r0 r1
000000101      // xor r0 r5
001000010      // beq r0 0 2
111111100      // j 60
010011001      // addi r3 1
010110111      // addi r6 -1
111111000      // j 56
```

7. Assembler

Description: We built the assembler with python3. The idea is to pass in an instruction and it returns the result as machine code. We have a look-up table that maps an assembly to a 9-bit machine code. Then, depending on the instruction type (J, I, R), we process it.

Example of input and output:

Input : xor r0 r1
Output : 000000001

```
import argparse

"""
Instruction          | Type | Opcode | rs   | rt   | rd   | imm    | target
```

```
----------------------|------|--------|------|------|------|--------|-------
xor                    |  R   | 000    | XXX  | XXX  | -    | -      | -
beq                    |  I   | 001    | X    | -    | -    | XXXX   | -
addi                   |  I   | 010    | XXX  | -    | -    | XXX    | -
andi                   |  I   | 011    | XXX  | -    | -    | XXX    | -
ls                     |  I   | 100    | XXX  | -    | -    | XXX    | -
ld                     |  R   | 101    | XXX  | XXX  | -    | -      | -
st                     |  R   | 110    | XXX  | XXX  | -    | -      | -
j                      |  J   | 111    | -    | -    | -    | -      | XXXXXX
"""

INSTRUCTION_FORMAT = {
    "xor":  {"type": "R", "opcode": "000", "width": 3},
    "beq":  {"type": "I", "opcode": "001", "width": 4},
    "addi": {"type": "I", "opcode": "010", "width": 3},
    "andi": {"type": "I", "opcode": "011", "width": 3},
    "ls":   {"type": "I", "opcode": "100", "width": 3},
    "ld":   {"type": "I", "opcode": "101", "width": 3},
    "st":   {"type": "I", "opcode": "110", "width": 3},
    "j":    {"type": "J", "opcode": "111", "width": 6}
}

TWOS_COMP = {
    '0':  '000',
    '1':  '001',
    '2':  '010',
    '3':  '011',
    '-3': '101',
    '-2': '110',
    '-1': '111'
}
```

```
# Clean up instruction
def tokenize_instruction(instruction):
    return instruction.split('//')[0].strip().split()

# Convert register or imm into binary
def decode_value(value, width):
    try:
        if value.startswith("r"): value = value.replace("r", "")
        bin_value = format(int(value), f'0{width}b')

        if len(bin_value) > width:
            raise ValueError(f"Value '{value}' too large for specified width of {width}.")

        return bin_value

    except ValueError as ve:
        raise ValueError(f"Failed to decode value '{value}': {str(ve)}")

# Process R-Type
def assemble_r_type(tokens, opcode, width):
    rs, rt = tokens[1], tokens[2]
    return opcode + decode_value(rs, width) + decode_value(rt, width)

# Process the two types of I-Type instructions
def assemble_i_type(tokens, opcode, width):
    instruction, rs = tokens[0], tokens[1]

    if len(tokens) == 4:
        if instruction == "beq":
            imm, label = tokens[2], tokens[3]
            return opcode + decode_value(rs, 1) + decode_value(imm, 1) + decode_value(label, 4)
```

```
        else:
            rd, imm = tokens[2], tokens[3]
            return opcode + decode_value(rs, width) + decode_value(rd, width) + decode_value(imm, width)
    else:
        imm = tokens[2]
        if (instruction == "ls") | (instruction == "addi"):
            try: return opcode + decode_value(rs, width) + TWOS_COMP[imm]
            except: return opcode + decode_value(rs, width) + '000'

        else:
            return opcode + decode_value(rs, width) + decode_value(imm, width)

# Process J-Type
def assemble_j_type(tokens, opcode, width):
    target = tokens[1]
    return opcode + decode_value(target, width)

# Convert assembly into machine code
def assemble_instruction(instruction):
    try:
        tokens = tokenize_instruction(instruction)
        op_info = INSTRUCTION_FORMAT.get(tokens[0])
        if not op_info: raise ValueError(f"Invalid instruction '{tokens[0]}'.")

        opcode = op_info["opcode"]
        width = op_info["width"]

        if op_info["type"] == "R": return assemble_r_type(tokens, opcode, width)
        elif op_info["type"] == "I": return assemble_i_type(tokens, opcode, width)
        else: return assemble_j_type(tokens, opcode, width)

    except Exception as e:
```

```
        raise ValueError(f"Failed to assemble instruction '{instruction}': {str(e)}")


# To perform tests, run `python3 assembler.py -t`
def test_assembler():
    test_data = [
        ("xor r0 r1", "000000001"),
        ("xor r0 r1", "000000001"),
        ("beq r1 0 15", "001101111"),
        ("addi r0 1", "010000001"),
        ("andi r0 1", "011000001"),
        ("ls r0 -4", "100000000"),
        ("ls r0 -3", "100000101"),
        ("ls r0 -2", "100000110"),
        ("ls r0 -1", "100000111"),
        ("ls r0 0", "100000000"),
        ("ls r0 1", "100000001"),
        ("ls r0 2", "100000010"),
        ("ls r0 3", "100000011"),
        ("ld r0 r1", "101000001"),
        ("st r0 r1", "110000001"),
        ("j 000000", "111000000"),
        ("j 20", "111010100")
    ]

    for instr, expected in test_data:
        result = assemble_instruction(instr)
        assert result == expected, f"Test failed for instruction '{instr}'. Expected: '{expected}' and got: '{result}'"

    print("All tests passed.")

"""
```

```
Input - A file called `assembly.txt` with the assembly instr. separated by newline
Output - A file called `mach_code.txt` with the corresponding machine code
"""
def process_file(input_filename, output_filename):
    try:
        with open(input_filename, 'r') as infile:
            instructions = infile.readlines()

        machine_codes = [assemble_instruction(instr.strip()) for instr in instructions]

        with open(output_filename, 'w') as outfile:
            for code, instr in zip(machine_codes, instructions):
                outfile.write(f"{code} \t// {instr.split('//')[0].strip()}\n")

    except FileNotFoundError:
        print(f"Error: File '{input_filename}' not found.")

    except Exception as e:
        print(f"Error: {str(e)}")

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Assembler script.")
    parser.add_argument("-t", "--test", action="store_true", help="Run the assembler tests.")
    args = parser.parse_args()

    if args.test:
        try: test_assembler()
        except AssertionError as e: print(f"Test Failed: {str(e)}")
    else:
        process_file("assembly.txt", "mach_code.txt")
```

8. Change Log

- Milestone 3
  - Finalized the ISA
  - Finalized the CPU
  - Finalized all three programs.
  - Updated assembly code from milestone 1 with our finalized assembly files.
  - Assembler is also finalized.
  - All schematics are finalized as well.
  - Machine code was also uploaded.
  - Ran and tested all programs against our test benches, and they all pass.
  - Updated incorrect pseudo code for program 2.
  - Updated all program pseudo code to finalized C files.
  - Updated the names of pseudo sections to Source Code
- Milestone 2
  - Changed various aspects of the ISA, and uploaded our first draft of our processor.
- Milestone 1
  - Initial submission, covered extensive requirements for ISA.

EXPLANATION OF CHANGES TO THE ISA:

addi (name change)	I type: should follow the same structure as noted in milestone 1.
andi (immediate bitwise and)	I type: takes a register, and isolates the lsb.
logical left-shift IS NOW just logical shift (lsl) is now (ls)	I type: same structure as noted in milestone 1, however the overall design uses negative numbers for left shifting and positive numbers for right shifting. # Assume R0 has `0b1000_0000` `ls R0, imm` ⇔

	`011  000  010`
	
	# after shift instruction, R0 now holds `0b0010_0000, as it was a right shift. (2 means move all bits twice to the right.`
	
	# Assume R0 has `0b0010_0000`
	
	`Ls R0 imm ⇔`
	`011  000  110`
	
	# after shift instruction, R0 now holds `0b1000_0000, as it was a left shift (-2 means move all bits twice to the left)`
	
	THE FIRST BIT OF OUR 3 BIT IMM IS A SIGN BIT TO DETERMINE LEFT OR RIGHT SHIFTING.

LOAD AND STORE Reflected on our table in MS 1	These instructions are R types now, to ensure 8 bit address memory can be accessed. This is crucial for Data Memory to work.
BEQ	Changed the overall structure of the instruction. Please refer to our table in MS 1 for updated semantics.