CPU Processor (changelog page 12)

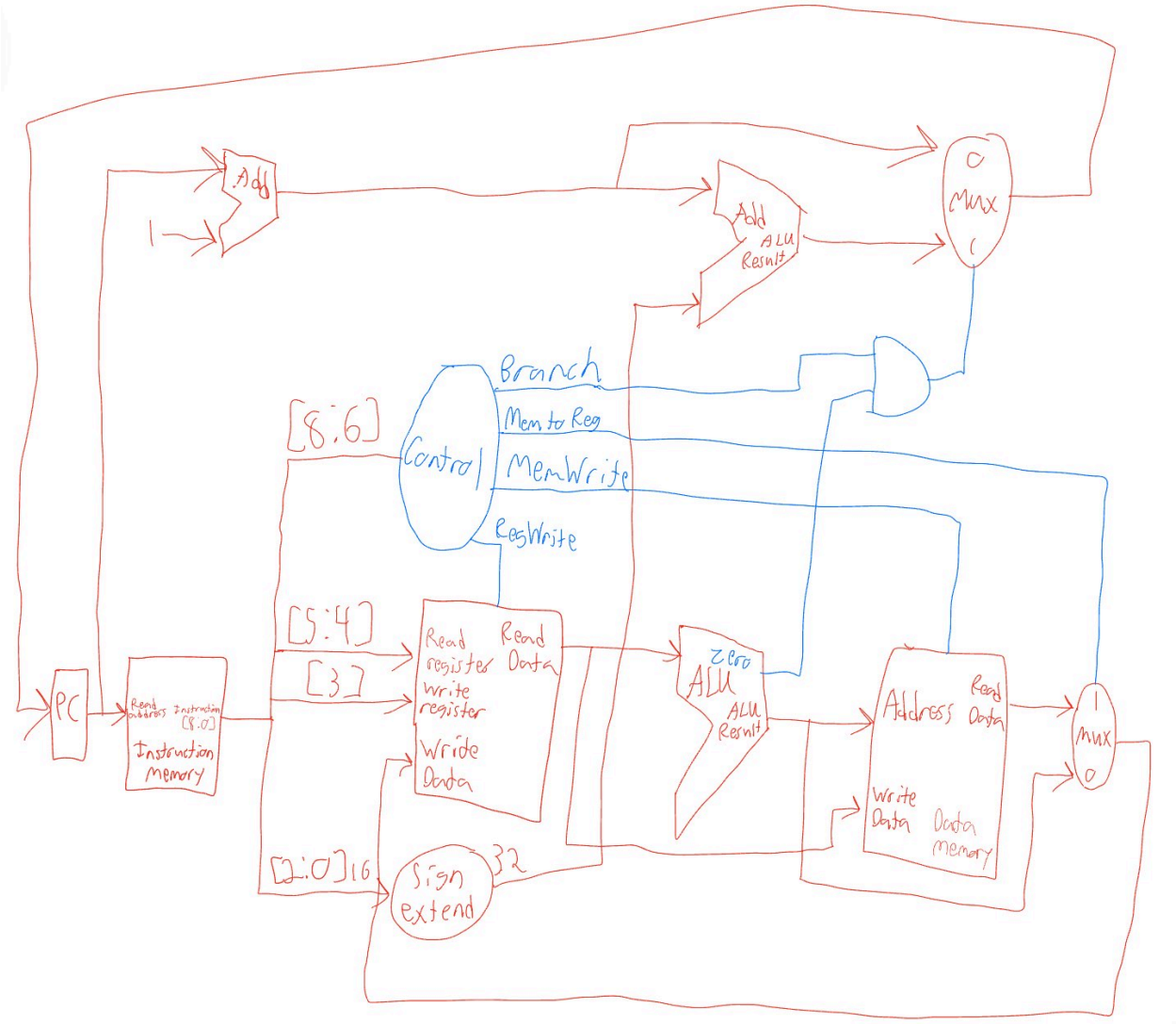
0. Team

Thomas Limperis, Yilin Song, Richard Nguyen

1. Introduction

We are calling the design of our ISA Asturias. The goal of the ISA was to make things as simple as possible. We used a register-register/load-store design. We wanted to reduce complexity so we have a very simple instruction set that allows us to perform the required calculations.

2. Architectural Overview



3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	3 bit opcode, 2 bit reg1, 2 bit reg2, 2 bit reg3	Xor, or, and, add,

R	3 bit opcode 2 bit reg1 2 bit reg2, 2 bit immediate	Srl
I	3 bit opcode, 2 bit reg1, 2 bit reg2, 2bit reg3 We are using flags here for li	Beq, sb, lbu
I	8 bit immediate	li
J	None	None

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
and = logical and	R	Opcode 000	#R0 = 00000001 #R1 = 00000000 And R0 ,R1, R0 #machine: 000000100 #R0 = 00000000	
Or = logical or	R	Opcode 001	#R0 = 00100001 #R1 = 00000000 Or r1 r1 r0 #machine 101010100 #R1 = 00000001	
add	R	Opcode 010	#R0 = 00000001 #R1 = 00000000 add r1 r1 r0 #machine: 010010100 #R1 = 00000001	
srl	R	Opcode 011	#R1 = 00000010 srl r1 r1 1 #machine: 011010101 #R1 = 00000001	

xor	R	Opcode 100	#R0 = 00000001 #R1 = 00000010 xor r1 r1 r0 #machine: 100010100 #R1 = 00000011	
beq	I	Opcode 101	#R0 = 00000001 #R1 = 00000000 #r2 = 3 Beq r1 r0, r2 #machine: 101010010 #If r1 == r0: jump to line 3 We use a flag here if the jump register is a 00 then we know that the jump is a 9 bit integer and we will instead read the next line as a 9 bit integer and go there Example: #machine 101000**00** # 111111111	This beq does not behave as a normal beq. It will jump to the line register r2 has We know here we need to jump to a 9 bit integer because of the mem flag 00, read next line as go to jump line
sb	I	Opcode 110	#T1 = 1 #mem[t2] = 0 sb $t1, mem($t2) #machine: 110010010 #mem[t2] = 1	
Lbu	I	Opcode 111	#T1 = 1 #mem[t2] = 5 Lbu $t1, mem($t2) #machine: 111010010 #t1 = 5	
li	I	Opcode 00000	Li t1, 15 #machine: 000000100 000001111	The opcode is 00000, we know in our code we will never use 00000 because 00 represents mem and we will never be working with “and” and “mem” in the same line of code. So the address of mem here works as a flag we know that we will be reading a 8 bit integer value as the next 9 bits and will be

				storing it into the 3-4 bit position register. Only way we could figure out how to use 8 bit integer values

Internal Operands

3 registers are supported T1,T2,T3, we wanted to make sure we could easily access mem and wasnt sure how that was going to work so we decided to label mem 00
Mem 00
T1 01
T2 10
T3 11

The purpose of the 3 registers are simply to load/store values into mem

Control Flow (branches)

We branch by finding the line we need to jump to and storing that value into a register, when we jump we simply make sure we set a beq to true and jump to the targeted line. This is only for values of 0-255. For values greater we use a flag in our

Max jump value is 511

For 1-8 bit values:
We branch by finding the line we need to jump to and storing that value into a register, when we jump we simply make sure we set a beq to true and jump to the targeted line.

T1 = 35
Beq 0 0, T1

This jumps to the line 35.

For 256-511 values
Example if we wanted to jump to line 511
Beq t1 t1, 00 (read next line as jump to integer value)

#111111111

Addressing Modes

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

We use direct and indirect addressing.
Examples:

Direct:
Li t1 , 1
Add t1, t1, t2

Indirect:
Sb t1, mem(t2)
Lbu t1, mem(60)

4. Programmer's Model [Lite]

TODO. 4.1 How should a programmer think about how your machine operates?

If someone wanted to code using our ISA, they should always think about what they need to perform on a single calculation. They should load the value they need into a t1 register, load a second value into a t2 register, and perform the operations they need. Then take that value and store it back into t1 or t2. If they need to store it later they should always store it somewhere in mem. They should be able to accomplish all of their needs if they perform calculations on 1-2 registers at a time only. Always should be done in this fashion,

1.load,
2. Calculate
3. store.

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

If the question is asking if the ISA can be copied to run other programs, the answer is yes. Our instruction set should be able to run any other program, though it has to be remembered that we slightly changed what some of the instructions actually mean.

5. Individual Component Specification

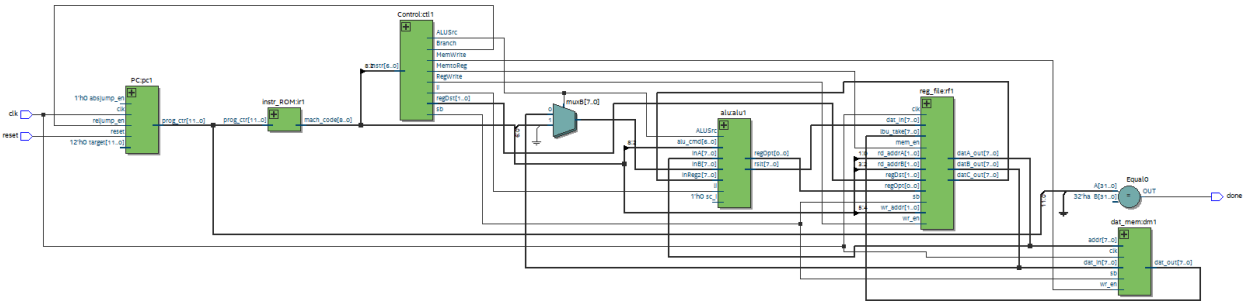
Top Level

Module file name: top_level.sv

Functionality Description

This module holds has all the file names and input/output logic so things can be instantiated and called during the program

Schematic



Program Counter

Module file name: PC.sv
Module testbench file name: milestone2_quicktest_tb.sv

Functionality Description

The PC module provides functionality to manage the program counter value in a typical computing system. It has the following features:

- Reset: Upon a reset signal, the program counter is set to 0.
- Absolute Jump: If the absjump_en (Absolute Jump Enable) signal is active, the program counter is set directly to the value provided by the target input.
- Increment: In the absence of any jump signals, the program counter is incremented by 1 at every positive edge of the clock.

(Optional) Testbench Description

TODO. Describe your testbench. How does it work? What test cases does it test?

Schematic

TODO. Show us your schematic for the fetch unit.

(Optional) Timing Diagram

TODO. Show us a screenshot of the timing diagram that demonstrates all relevant functions of the fetch unit.

Instruction Memory

Module file name: instr_ROM.sv

Functionality Description

The instr_ROM module serves as a memory unit to store and fetch machine code instructions. It provides the following features:

- Memory Core: The module contains a memory array core that can store up to 2^D machine code instructions, where D is a parameter that specifies the width of the program counter. Each instruction is 9 bits wide.
- Initialization: At initialization, the module loads machine code instructions from the "mach_code.txt" file into the core memory array.
- Fetching: The module fetches a machine code instruction from the core memory based on the address provided by the prog_ctr input and outputs it through the mach_code output port.
- Display: The module displays the program counter value and the corresponding fetched machine code instruction for debugging purposes.

Schematic

TODO. Show us your schematic for the fetch unit.

Control Decoder

Module file name: TODO

Functionality Description

The Control module decodes instructions (specifically their opcode and function parts) and produces control signals for the rest of the CPU. Its main functionalities include:

1. Instruction Decoding: The module examines a subset of the machine code (particularly the opcode) to determine the type of instruction being executed.
2. Control Signal Generation: Depending on the instruction type, the module sets or clears various control signals, including:
    a. Branch: Determines whether a branch (conditional jump) should be taken.
    b. MemWrite: Activates writing to memory, used for store operations.
    c. MemtoReg: Determines whether the result from the memory or the ALU should be written to the register file.
    d. ALUSrc: Selects between immediate value and the second register file output as input to the ALU.
    e. RegWrite: Activates writing to the register file.
    f. ALUOp: Specifies the operation the ALU should perform.
    g. regDst: Specifies which register should be written to.
    h. li: Control for load immediate operation.
    i. sb: Control for set bit operation.
3. Special Case Handling: Some specific instructions or conditions might require additional logic or overriding default behaviors, such as the logic handling the li instruction, which seems to be a load immediate operation.

The module's behavior is defined mainly through the case statement that examines the opcode of the instruction and sets control signals accordingly. After this primary decoding, there are additional checks and adjustments to the control signals, especially related to the li operation.

Schematic

TODO. Show us your schematic for the control decoder.

Register File

Module file name: TODO

Functionality Description

The reg_file module simulates a cache memory or a register file with the following functionalities:

1. Storage: The module contains a storage array core which can store up to 2^pw 8-bit wide values. By default, pw is 4, meaning the core can store 16 8-bit values, simulating 16 registers. But since the max bit size to represent a register is 2-bits in our machine code, we are only using 3 registers in total.
2. Reading: Register values can be read from three different read addresses:

    a. rd_addrA to datA_out
    b. wr_addr to datB_out
    c. rd_addrB to datC_out The read operations are combinational.
3. Writing: Writes to the registers are clocked and occur on the rising edge of the clk signal. The register to be written is determined by the wr_addr or regDst input, based on conditions, and the data to be written comes from the dat_in or lbu_take input.
4. Control Signals: The module has several control signals:
    a. wr_en: When high, it enables writing to the register file.
    b. mem_en: When high, data from lbu_take is written to the specified register.
    c. regOpt: Seems to be an overriding control signal, forcing a write to the address specified by wr_addr.
    d. sb: Appears in the code but its functionality isn't clear from the provided snippet.

Schematic

TODO. Show us your schematic for the register file.

ALU (Arithmetic Logic Unit)

Module file name: TODO
Module testbench file name: TODO

Functionality Description

TODO. Write a brief description of the functionality of this module.

(Optional) Testbench Description

TODO. Describe your testbench. How does it work? What test cases does it test?

ALU Operations

TODO. What ALU operations will you be demonstrating? What instructions are they relevant to?

Schematic

TODO. Show us your schematic for the register file.

(Optional) Timing Diagram

TODO. Show us a screenshot of the timing diagram that demonstrates all relevant operations you mentioned in the ALU Operations section.

Data Memory

Module file name: TODO

Functionality Description

TODO. Write a brief description of the functionality of this module.

Schematic

TODO. Show us your schematic for the data memory.

Lookup Tables

Module file name: TODO

Functionality Description

TODO. Write a brief description of the functionality of this module.

Schematic

TODO. Show us your schematic(s) for the lookup table(s).

Muxes (Multiplexers)

Module file name: TODO

Functionality Description

TODO. Write a brief description of the functionality of this module.

Schematic

TODO. Show us your schematic for your mux(es).

Other Modules (if necessary)

Module file name: TODO

Functionality Description

TODO. Write a brief description of the functionality of this module.

Schematic

TODO. Show us your schematic for your module.

7. Changelog

TODO. have a bulleted list of your changes here. Example below:

Milestone 3:

Slightly modified a few lines of the assembly code to be compatible with our assembler, no other changes were made. P3 was modified and used as the assembler example in the text files in the project folder.

The files for the assembly code are now text files and added in project folder.
The assembly input and output examples are given in project folder.

The functions and loops are left in the text file so we can change to mach code manually of where to jump to

6. Milestone 2
    a. Introduction
        i. edited to change from a load/store architecture to accumulator architecture.
    b. TODO: add bullet points as necessary
7. Milestone 1
    a. Initial version

8. Program Implementation

Program 1 Pseudocode

```
# Given a series of fifteen 11-bit message blocks in data mem[0:29], generate the corresponding
# 16-bit encoded versions and store these in data mem[30:59].
# Input and output formats are as follows:
# input MSW = 0 0 0 0 0 b11 b10 b09
# LSW = b8 b7 b6 b5 b4 b3 b2 b1, where bx denotes a data bit
# output MSW = b11 b10 b9 b8 b7 b6 b5 p8
# LSW = b4 b3 b2 p4 b1 p2 p1 p0, where px denotes a parity bit
# Example, to clarify “endianness”: binary data value = 101_0101_0101
# mem[1] = 00000101 -- 5 bits zero pad followed by b11:b9 = 00000_101
# mem[0] = 01010101 -- lower 8 data bits b8:b1
# You would generate and store:
# mem[31] = 10101010 -- b11:b5, p8 = 1010101_0
```

```
# mem[30] = 01011010 -- b4:b2, p4, b1, p2:p1, p0 = 010_1_1_01_0
# p8 = ^(b11:b5) = 0;
# p4 = ^(b11:b8,b4,b3,b2) = 1;
# p2 = ^(b11,b10,b7,b6,b4,b3,b1) = 0;
# p1 = ^(b11,b9,b7,b5,b4,b2,b1) = 1;
# p0 = ^(b11:1,p8,p4,p2,p1) = 0;

mem = ['00000000'] *32
val = '10101010101'
mem[1] = '00000' + val[0:3]
mem[0] = val[3:]


# p8 = ^(b11:b5) = 0;
#mem[1]         b11 = 5, b10 =6, b9 = 7
p8 = int(mem[1][5])
p8 = p8 ^ int(mem[1][6])
p8 = p8 ^ int(mem[1][7])
#mem[0]         b8 = 0, b7 =1, b6 = 2, b5 = 3, b4 =4, b3 = 5, b2 = 6, b1 = 7
p8 = p8 ^ int(mem[0][0])
p8 = p8 ^ int(mem[0][1])
p8 = p8 ^ int(mem[0][2])
p8 = p8 ^ int(mem[0][3])
print(p8)

# p4 = ^(b11:b8,b4,b3,b2) = 1;
p4 = int(mem[1][5])
p4 = p4 ^ int(mem[1][6])
p4 = p4 ^ int(mem[1][7])
p4 = p4 ^ int(mem[0][0])
p4 = p4 ^ int(mem[0][4])
p4 = p4 ^ int(mem[0][5])
p4 = p4 ^ int(mem[0][6])
print(p4)

#p2 = ^(b11,b10,b7,b6,b4,b3,b1) = 0;
p2 = int(mem[1][5])
p2 = p2 ^ int(mem[1][6])
p2 = p2 ^ int(mem[0][1])
p2 = p2 ^ int(mem[0][2])
p2 = p2 ^ int(mem[0][4])
p2 = p2 ^ int(mem[0][5])
p2 = p2 ^ int(mem[0][7])
print(p2)
```

```
#mem[1]         b11 = 5, b10 =6, b9 = 7
#mem[0]         b8 = 0, b7 =1, b6 = 2, b5 = 3, b4 =4, b3 = 5, b2 = 6, b1 = 7
# p1 = ^(b11,b9,b7,b5,b4,b2,b1) = 1;
p1 = int(mem[1][5])
p1 = p1 ^ int(mem[1][7])
p1 = p1 ^ int(mem[0][1])
p1 = p1 ^ int(mem[0][3])
p1 = p1 ^ int(mem[0][4])
p1 = p1 ^ int(mem[0][6])
p1 = p1 ^ int(mem[0][7])
print(p1)

# p0 = ^(b11:1,p8,p4,p2,p1) = 0;
p0 = int(mem[1][5])
p0 = p0 ^ int(mem[1][6])
p0 = p0 ^ int(mem[1][7])
p0 = p0 ^ int(mem[0][0])
p0 = p0 ^ int(mem[0][1])
p0 = p0 ^ int(mem[0][2])
p0 = p0 ^ int(mem[0][3])
p0 = p0 ^ int(mem[0][4])
p0 = p0 ^ int(mem[0][5])
p0 = p0 ^ int(mem[0][6])
p0 = p0 ^ int(mem[0][7])
p0 = p0 ^ p1
p0 = p0 ^ p2
p0 = p0 ^ p4
p0 = p0 ^ p8
print(p0)

# You would generate and store:
# mem[31] = 10101010 -- b11:b5, p8 = 1010101_0
# mem[30] = 01011010 -- b4:b2, p4, b1, p2:p1, p0 = 010_1_1_01_0

#mem[1]         b11 = 5, b10 =6, b9 = 7
#mem[0]         b8 = 0, b7 =1, b6 = 2, b5 = 3, b4 =4, b3 = 5, b2 = 6, b1 = 7
mem[31] = (mem[1][5:])
mem[31] =mem[31] + mem[0][0:4]
mem[31] = mem[31] + str(p8)
mem[30] = mem[0][4:7]
mem[30] = mem[30] + str(p4)
mem[30] = mem[30] + mem[0][7:8]
```

```
mem[30] = mem[30] + str(p2)
mem[30] = mem[30] + str(p1)
mem[30] = mem[30] + str(p0)
print(mem[30])
print(mem[31])
```

Program 1 Assembly Code

```
main:
    li $t1, 1
    xor $t2, $t2, $t1 # p8 = int(mem[1][5])
    li $t1, 0
    xor $t2, $t2, $t1 # p8 = p8 ^ int(mem[1][6])
    li $t1, 1
    xor $t2, $t2, $t1 # p8 = p8 ^ int(mem[1][7])
    li $t1, 0
    xor $t2, $t2, $t1 # p8 = p8 ^ int(mem[0][0])
    li $t1, 1
    xor $t2, $t2, $t1 # p8 = p8 ^ int(mem[0][1])
    li $t1, 0
    xor $t2, $t2, $t1 # p8 = p8 ^ int(mem[0][2])
    li $t1, 1
    xor $t2, $t2, $t1 # p8 = p8 ^ int(mem[0][3])
    li $t1, 60
    sb $t2, 0($t1) # Store p8 in mem[60]

    li $t1, 1
    xor $t2, $t2, $t1 # p4 = int(mem[1][5])
    li $t1, 0
    xor $t2, $t2, $t1 # p4 = p4 ^ int(mem[1][6])
    li $t1, 1
    xor $t2, $t2, $t1 # p4 = p4 ^ int(mem[1][7])
    li $t1, 0
    xor $t2, $t2, $t1 # p4 = p4 ^ int(mem[0][0])
    li $t1, 0
    xor $t2, $t2, $t1 # p4 = p4 ^ int(mem[0][4])
    li $t1, 1
    xor $t2, $t2, $t1 # p4 = p4 ^ int(mem[0][5])
    li $t1, 0
    xor $t2, $t2, $t1 # p4 = p4 ^ int(mem[0][6])
    li $t1, 64
    sb $t2, 0($t1) # Store p4 in mem[64]

    li $t1, 1
```

```
xor $t2, $t2, $t1 # p2 = int(mem[1][5])
li $t1, 0
xor $t2, $t2, $t1 # p2 = p2 ^ int(mem[1][6])
li $t1, 1
xor $t2, $t2, $t1 # p2 = p2 ^ int(mem[0][1])
li $t1, 0
xor $t2, $t2, $t1 # p2 = p2 ^ int(mem[0][2])
li $t1, 1
xor $t2, $t2, $t1 # p2 = p2 ^ int(mem[0][4])
li $t1, 0
xor $t2, $t2, $t1 # p2 = p2 ^ int(mem[0][5])
li $t1, 1
xor $t2, $t2, $t1 # p2 = p2 ^ int(mem[0][7])
li $t1, 68
sb $t2, 0($t1) # Store p2 in mem[68]

li $t1, 1
xor $t2, $t2, $t1 #p1 = int(mem[1][5])
li $t1, 1
xor $t2, $t2, $t1 #p1 = p1 ^ int(mem[1][7])
li $t1, 1
xor $t2, $t2, $t1 #p1 = p1 ^ int(mem[0][1])
li $t1, 1
xor $t2, $t2, $t1 #p1 = p1 ^ int(mem[0][3])
li $t1, 0
xor $t2, $t2, $t1 #p1 = p1 ^ int(mem[0][4])
li $t1, 0
xor $t2, $t2, $t1 #p1 = p1 ^ int(mem[0][6])
li $t1, 1
xor $t2, $t2, $t1 #p1 = p1 ^ int(mem[0][7])
li $t1, 72
sb $t2, 0($t1) # Store p1 in mem[72]

li $t1, 1
xor $t2, $t2, $t1#p0 = int(mem[1][5])
li $t1, 0
xor $t2, $t2, $t1 #p0 = p0 ^ int(mem[1][6])
li $t1, 1
xor $t2, $t2, $t1 #p0 = p0 ^ int(mem[1][7])
li $t1, 0
xor $t2, $t2, $t1 #p0 = p0 ^ int(mem[0][0])
li $t1, 1
xor $t2, $t2, $t1 #p0 = p0 ^ int(mem[0][1])
li $t1, 0
```

```
xor $t2, $t2, $t1 #p0 = p0 ^ int(mem[0][2])
li $t1, 1
xor $t2, $t2, $t1 #p0 = p0 ^ int(mem[0][3])
li $t1, 0
xor $t2, $t2, $t1 #p0 = p0 ^ int(mem[0][4])
li $t1, 1
xor $t2, $t2, $t1 #p0 = p0 ^ int(mem[0][5])
li $t1, 0
xor $t2, $t2, $t1 #p0 = p0 ^ int(mem[0][6])
li $t1, 1
xor $t2, $t2, $t1 #p0 = p0 ^ int(mem[0][7])
li $t1, 72
lbu $t3, 0($t1)
xor $t2, $t2, $t3 #p0 = p0 ^ p1
li $t1, 68
lbu $t3, 0($t1)
xor $t2, $t2, $t3 #p0 = p0 ^ p2
li $t1, 64
lbu $t3, 0($t1)
xor $t2, $t2, $t3 #p0 = p0 ^ p4
li $t1, 60
lbu $t3, 0($t1)
xor $t2, $t2, $t3 #p0 = p0 ^ p8
li $t1, 76
sb $t2, 0($t1) # Store p0 in mem[76]

li $t1, 101       # mem[31] = (mem[1][5:])
li $t2, 0101
add $t1, $t1, $t2 # mem[31] =mem[31] + mem[0][0:4]
li $t2, 60
lbu $t3, 0($t1)
add $t2, $t2, $t3 # mem[31] = mem[31] + str(p8)
li $t2, 31
sb $t1, 0($t2)      # Store in mem[31]

li $t1, 0101              # mem[30] = mem[0][4:7]
li $t2, 64
lbu $t3, 0($t2)
add $t1, $t1, $t3 # mem[30] = mem[30] + str(p4)
li $t3, 01
add $t1, $t1, $t3 # mem[30] = mem[30] + mem[0][7:8]
li $t2, 68
lbu $t3, 0($t2)
add $t1, $t1, $t3 # mem[30] = mem[30] + str(p2)
```

```
    li $t2, 72
    lbu $t3, 0($t2)
    add $t1, $t1, $t3 # mem[30] = mem[30] + str(p1)
    li $t2, 76
    lbu $t3, 0($t2)
    add $t1, $t1, $t3 # mem[30] = mem[30] + str(p0)
    li $t2, 31
    sb $t1, 0($t2)          # Store in mem[30]
```

Program 2 Pseudocode

Modified slightly

```
def extract_data_and_parity_bits(encoded_msw, encoded_lsw):
        # MSW: b11  b10  b9  b8  b7  b6  b5  p8
        # LSW: b4  b3  b2  p4  b1  p2  p1  p0
        data_msw = encoded_msw[:7]
        data_lsw = encoded_lsw[0:3] + encoded_lsw[4]
        p8 = int(encoded_msw[7])
        p4 = int(encoded_lsw[3])
        p2 = int(encoded_lsw[5])
        p1 = int(encoded_lsw[6])
        p0 = int(encoded_lsw[7])
        return data_msw, data_lsw, p8, p4, p2, p1, p0

def calculate_expected_parity_bits(data_msw, data_lsw):
        # MSW: b11  b10  b9  b8  b7  b6  b5  p8
        # LSW: b4  b3  b2  p4  b1  p2  p1  p0
        # p8 = ^(b11:b5) = 0;
        # p4 = ^(b11:b8,b4,b3,b2) = 1;
        # p2 = ^(b11,b10,b7,b6,b4,b3,b1) = 0;
        # p1 = ^(b11,b9,b7,b5,b4,b2,b1) = 1;
        # p0 = ^(b11:1,p8,p4,p2,p1) = 0;
        sum_for_p4 = int(data_msw[0]) + int(data_msw[1]) + int(data_msw[2]) +
int(data_msw[3])
        sum_for_p4 += int(data_lsw[0]) + int(data_lsw[1]) + int(data_lsw[2])

        sum_for_p2 = int(data_msw[0]) + int(data_msw[1]) + int(data_msw[4]) +
int(data_msw[5])
        sum_for_p2 += int(data_lsw[0]) + int(data_lsw[1]) + int(data_lsw[3])

        sum_for_p1 = int(data_msw[0]) + int(data_msw[2]) + int(data_msw[4]) +
int(data_msw[6])
```

```
        sum_for_p1 += int(data_lsw[0]) + int(data_lsw[2]) + int(data_lsw[3])

        sum_for_p0 = sum(int(bit) for bit in data_msw + data_lsw)
        sum_for_p0 += p8 + p4 + p2 + p1

        p8 = 0
        for bit in data_msw:
        p8 ^= int(bit)  #XOR
        p4 = sum_for_p4 % 2
        p2 = sum_for_p2 % 2
        p1 = sum_for_p1 % 2
        p0 = sum_for_p0 % 2
        return p8, p4, p2, p1, p0

def decode_one_block(data_msw, data_lsw, received_parity):
        expected_parity = calculate_expected_parity_bits(data_msw, data_lsw)
        errors = []
        for r, e in zip(received_parity, expected_parity):
        if r != e:
        errors.append(True)
        else:
        errors.append(False)
        error_count = 0
        for error in errors:
        if error:
        error_count += 1
        if error_count == 0:
        F1F0 = '00'
        elif error_count == 1:
        F1F0 = '01'
        else:
        F1F0 = '1X'
        return F1F0, data_msw, data_lsw

mem = ['00000000'] * 60
mem[30] = '01011010'
mem[31] = '10101010'
for i in range(30, 60, 2):
        data_msw, data_lsw, p8, p4, p2, p1, p0 = extract_data_and_parity_bits(mem[i+1],
mem[i])
        F1F0, corrected_msw, corrected_lsw = decode_one_block(data_msw, data_lsw, [p8, p4,
p2, p1, p0])
        mem[(i-30)//2 + 1] = F1F0 + '00000' + corrected_msw
        mem[(i-30)//2] = corrected_lsw
```

result_check = mem[:30]

Program 2 Assembly Code

```
# Inputs: None (starts with default values)
# Outputs: Decoded data in mem[0:119]

main:
        li $t3, 120       # Starting point at mem[120] for encoded data

loop:
        # Checking if we've reached mem[240], which is the end
        li $t1, 240
        beq $t3, $t1, end

        # Load encoded_lsw and encoded_msw
        lbu $t1, 0($t3)           # Load encoded_lsw
        add $t3, $t3, 4           # Move to next block
        lbu $t2, 0($t3)           # Load encoded_msw

        # Call the function to extract data and parity bits
        beq $t3, $t3, extract_data_and_parity_bits

loop_continue_first:
        # Call the function to decode the block and determine errors
        beq $t3, $t3, decode_one_block

loop_continue_second:
        # Store the result in mem[0:119]
        sb $t1, 0($t3)  # Store corrected_lsw
        add $t3, $t3, 4
        sb $t2, 0($t3)            # Store F1F0 + '00000' + corrected_msw

        # Move to the next block of encoded data
        add $t3, $t3, 4
        beq $t3, $t3, loop  # Continue the loop

end:
        nop               # End of the program


# Inputs: $t1 (encoded_msw), $t2 (encoded_lsw)
```

```
# Outputs: $t1 (data_msw), $t2 (data_lsw), mem[244] (p8), mem[248] (p4), mem[252] (p2), mem[256] (p1), mem[260] (p0)

extract_data_and_parity_bits:
        # Extracting p8 from encoded_msw
        li $t3, 128
        and $t3, $t1, $t3
        srl $t3, $t3, 7
        li $t1, 244
        sb $t3, 0($t1)  # Store p8 to mem[244]

        # Extracting data_msw from encoded_msw
        li $t3, 127
        and $t1, $t1, $t3

        # Extracting p4 from encoded_lsw
        li $t3, 8
        and $t3, $t2, $t3
        srl $t3, $t3, 3
        li $t1, 248
        sb $t3, 0($t1)  # Store p4 to mem[248]

        # Extracting p2 from encoded_lsw
        li $t3, 32
        and $t3, $t2, $t3
        srl $t3, $t3, 5
        li $t1, 252
        sb $t3, 0($t1)  # Store p2 to mem[252]

        # Extracting p1 from encoded_lsw
        li $t3, 64
        and $t3, $t2, $t3
        srl $t3, $t3, 6
        li $t1, 256
        sb $t3, 0($t1)  # Store p1 to mem[256]

        # Extracting p0 from encoded_lsw
        li $t3, 128
        and $t3, $t2, $t3
        srl $t3, $t3, 7
        li $t1, 260
        sb $t3, 0($t1)  # Store p0 to mem[260]

        # Extracting data_lsw from encoded_lsw
```

```
        li $t3, 23   # This mask gets bits 0, 1, 2, and 4
        and $t2, $t2, $t3

        beq $t3, $t3, loop_continue_first



# Inputs: $t1 (data_msw), $t2 (data_lsw)
# Outputs: mem[264] (p4), mem[268] (p2), mem[272] (p1), mem[276] (p0), mem[280] (p8)

calculate_expected_parity_bits:
        # For p4
        li $t3, 241
        and $t3, $t1, $t3       # Extract relevant bits from data_msw
        li $t3, 7
        and $t3, $t2, $t3       # Add relevant bits from data_lsw (bits b4, b3, and b2)
        xor $t1, $t1, $t3       # XOR operation for all bits
        li $t3, 264
        sb $t1, 0($t3)  # Store the calculated p4 in mem[264]

        # For p2
        li $t3, 195
        and $t3, $t1, $t3       # Extract relevant bits from data_msw
        li $t3, 11
        and $t3, $t2, $t3       # Add relevant bits from data_lsw (bits b4, b3, and b1)
        xor $t1, $t1, $t3       # XOR operation for all bits
        li $t3, 268
        sb $t1, 0($t3)  # Store the calculated p2 in mem[268]

        # For p1
        li $t3, 165
        and $t3, $t1, $t3       # Extract relevant bits from data_msw
        li $t3, 10
        and $t3, $t2, $t3       # Add relevant bits from data_lsw (bits b4, b2, and b1)
        xor $t1, $t1, $t3       # XOR operation for all bits
        li $t3, 272
        sb $t1, 0($t3)  # Store the calculated p1 in mem[272]

        # For p0
        or $t3, $t1, $t2        # Combine both data_msw and data_lsw
        li $t1, 255
        xor $t3, $t3, $t1       # XOR operation for all bits
        li $t1, 276
        sb $t3, 0($t1)  # Store the calculated p0 in mem[276]
```

```
        # p8 is always 0, so no calculation required
        li $t1, 0
        li $t3, 280
        sb $t1, 0($t3)  # Store 0 for p8 in mem[280]

        # Return to the main routine
        beq $t3, $t3, loop_continue_second



# Inputs: $t1 (data_msw), $t2 (data_lsw), mem[244:260] (received parity bits)
# Outputs: $t1 (F1F0 + '00000' + corrected_msw), $t2 (corrected_lsw)

decode_one_block:
        # Calculate the expected parity bits for the given data
        beq $t3, $t3, calculate_expected_parity_bits

decode_continue:
        # Load the received parity bits
        li $t3, 244
        lbu $t1, 0($t3)   # Load p8
        li $t3, 248
        lbu $t2, 0($t3)   # Load p4
        li $t3, 252
        lbu $t3, 0($t3)   # Load p2

        li $t1, 284
        sb $t2, 0($t1)
        li $t1, 288
        sb $t2, 0($t1)
        li $t1, 292
        sb $t3, 0($t1)

        # Assuming we stored the received p1 at mem[296] and p0 at mem[300]
        li $t3, 256
        lbu $t1, 0($t3)   # Load p1
        li $t3, 296
        sb $t1, 0($t3)
        li $t3, 260
        lbu $t1, 0($t3)   # Load p0
        li $t3, 300
        sb $t1, 0($t3)
```

```
        # Initialize error counter
        li $t3, 0

        # Compare received parity p8 to expected parity in mem[280]
        li $t1, 280
        lbu $t1, 0($t1)
        li $t2, 284
        lbu $t2, 0($t2)
        beq $t1, $t2, no_error_p8
        add $t3, $t3, $t3
        add $t3, $t3, 1
no_error_p8:

        # Compare received parity p4 to expected parity in mem[264]
        li $t1, 264
        lbu $t1, 0($t1)
        li $t2, 288
        lbu $t2, 0($t2)
        beq $t1, $t2, no_error_p4
        add $t3, $t3, $t3
        add $t3, $t3, 1
no_error_p4:

        # Compare received parity p2 to expected parity in mem[268]
        li $t1, 268
        lbu $t1, 0($t1)
        li $t2, 292
        lbu $t2, 0($t2)
        beq $t1, $t2, no_error_p2
        add $t3, $t3, $t3
        add $t3, $t3, 1
no_error_p2:

        # Compare received parity p1 to expected parity in mem[272]
        li $t1, 272
        lbu $t1, 0($t1)
        li $t2, 296
        lbu $t2, 0($t2)
        beq $t1, $t2, no_error_p1
        add $t3, $t3, $t3
        add $t3, $t3, 1
no_error_p1:

        # Compare received parity p0 to expected parity in mem[276]
```

```
        li $t1, 276
        lbu $t1, 0($t1)
        li $t2, 300
        lbu $t2, 0($t2)
        beq $t1, $t2, no_error_p0
        add $t3, $t3, $t3
        add $t3, $t3, 1
no_error_p0:

        # Set F1F0 based on the number of errors
        li $t1, 0           # Default F1F0 to '00'
        beq $t3, 1, set_F1F0_one_error
        beq $t3, 0, set_F1F0_no_error

        # If two or more errors, set F1F0 to '1X'
        li $t1, 2           # 10 in binary
        beq $t3, $t3, set_F1F0_end

set_F1F0_one_error:
        li $t1, 1           # If one error, set F1F0 to '01'

set_F1F0_no_error:
        # This step is added to ensure continuity. No operations are done here.

set_F1F0_end:
        # Shift the F1F0 value to make space for the data
        add $t1, $t1, $t1
        add $t1, $t1, $t1
        add $t1, $t1, $t1
        add $t1, $t1, $t1
        add $t1, $t1, $t1
        add $t1, $t1, $t1       # Shifted left 6 times

        # Now, OR the data_msw with the shifted F1F0
        or $t1, $t1, $t2        # This will create F1F0 + '00000' + corrected_msw

        # At this point, $t1 has the desired format: F1F0 + '00000' + corrected_msw
        # And $t2 has the corrected_lsw

        # Return to the main routine
        beq $t3, $t3, decode_continue
```

Program 3 Pseudocode

Modified slightly so that there is no inner loop J

```
mem = ['00000000'] *32  #01010101010101010101010101010101

p3 = '00000'
total_occurence = 0
byte_occurrence = 0
continuous_occurrence = 0
#change to p1 or p2 for diff results
p = p3
for i in range(32):
        byte_flag = True
        j = 0
        while j + 5 <= len(mem[i]):
        curr = mem[i]
        if curr[j:j+5] == p:
        total_occurence += 1
        if (byte_flag):
                byte_occurrence = byte_occurrence +1
                byte_flag = False
        j += 1
# a. Enter the total number of occurrences of the given 5-bit pattern in any byte into data
# mem[33]. Do not cross byte boundaries for this count.
mem33 = total_occurence
print(mem33)
# b. Write the number of bytes within which the pattern occurs into data mem[34].
mem34 = byte_occurrence #we no longer need byte_occurence can re use
print(mem34)

for i in range(len(mem) -1) :
        curr = mem[i]
        for j in range(4,8):
        byte_occurence = mem[i+1]  #reuse byte_occurence to save on variables
        byte_occurence = curr[j:] + byte_occurence[:5-(8-j)]
        if (byte_occurence == p):
        continuous_occurrence +=1
        i +=1
```

```
# c. Write the total number of times it occurs anywhere in the string into data mem[35].
mem34 = continuous_occurrence + total_occurence
print(mem34)
# For this total count, consider the 32 bytes to comprise one continuous 256-bit message,
# such that the 5-bit pattern could span adjacent portions of two consecutive bytes.
```

Program 3 Assembly Code

The following 3 programs are 99% complete and are being used to debug, once we are confident they work properly in assembly we are changing a few lines to match our ISA.

```
.data
        mem:            .byte 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA,
                .byte 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA
                # Fillers for indices 16-59 (total 44 bytes)
                .space 44
                # Empty values from indices 60-255 (total 196 bytes)
                .space 196
        p3:             .byte 0x15
        total_occurrence:   .word 0
        byte_occurrence:    .word 0


.text
main:

        #op
  # or $t1, $t1, 1        # Load the value 1 into $t1
        li $t1, 1
  #li $t2, 61            # Load the value 61 into $t2
        li $t2, 61
        sb $t1, mem ($t2)
  # sb $t1, mem + 61          # Store the value in $t1 at the calculated memory address
        #this code is more like, store t1 into mem + t2


outer_loop:

        #say t1 is 000, say mem is 00
        li $t2, 60
        lbu $t1, mem($t2) #this is my i value
        li $t2, 16
        beq $t1, $t2, end       # If i >= 32, exit loop
```

```
        lbu $t3, p3             # Load address of p3 into $t3

        li $t2, 61
        lbu $t1 , mem($t2)
        li $t2, 62
        sb $t1, mem($t2)
        #j check_1 can see if continous is working

c1:
        li $t2, 60
        lbu $t1, mem($t2)
        lbu $t1, mem($t1)
        srl $t1, $t1, 3   # Shift bits 4-8 to the rightmost position
        beq $t1, $t3, m1

c2:
        li $t2, 60
        lbu $t1, mem($t2)
        lbu $t1, mem($t1)
        li $t2, 248
        #and $t1, $t1, 0xF8
        and $t1, $t1, $t2
        srl $t1, $t1, 3   # Shift bits 4-8 to the rightmost position
        beq $t1, $t3, m2

c3:
        li $t2, 60
        lbu $t1, mem($t2)
        lbu $t1, mem($t1)
        li $t2, 124
        #and $t1, $t1, 0x7C
        and $t1, $t1, $t2
        srl $t1, $t1, 2   # Shift bits 4-8 to the rightmost position
        beq $t1, $t3, m3
c4:
        li $t2, 60
        lbu $t1, mem($t2)
        lbu $t1, mem($t1)
        li $t2, 62
  # and $t1, $t1, 0x3E
        and $t1, $t1, $t2
        beq $t1, $t3, m4
  # j check_1
        beq $t1, $t1, check_1
```

```
m1:

        #add $t4, $t4, $s5      # Increment total_occurrence
        li $t2, 63
        lbu $t1, mem($t2)
        li $t2, 61
        lbu $t2, mem($t2)
        add $t1, $t1, $t2
        li $t2, 63
        sb $t1, mem($t2)

        li $t2, 62
        lbu $t2, mem($t2)
        li $t3, 0
        beq $t2, $t3, b1
 #  add $t5, $t5, $s5            # Increment byte_occurrence
        li $t2, 64
        lbu $t2 , mem($t2)

        li $t3, 61
        lbu $t1, mem($t3)
        add $t2, $t2, $t1
        li $t3, 64
        sb $t2, mem($t3)
        #li $t2, 0               # Set byte_flag to false
        li $t3, 62
        li $t2, 0
        sb $t2, mem($t3)
        lbu $t3, p3
        beq $t3, $t3, c2
  # j c2
m2:

        #add $t4, $t4, $s5      # Increment total_occurrence
        li $t2, 63
        lbu $t1, mem($t2)
        li $t2, 61
        lbu $t2, mem($t2)
        add $t1, $t1, $t2
        li $t2, 63
        sb $t1, mem($t2)

        li $t2, 62
```

```
        lbu $t2, mem($t2)
        li $t3, 0
        beq $t2, $t3, b2
 #  add $t5, $t5, $s5          # Increment byte_occurrence
        li $t2, 64
        lbu $t2 , mem($t2)

        li $t3, 61
        lbu $t1, mem($t3)
        add $t2, $t2, $t1
        li $t3, 64
        sb $t2, mem($t3)
        #li $t2, 0              # Set byte_flag to false
        li $t3, 62
        li $t2, 0
        sb $t2, mem($t3)
        lbu $t3, p3
        #j c3
        beq $t3, $t3, c3
m3:
        #add $t4, $t4, $s5      # Increment total_occurrence
        li $t2, 63
        lbu $t1, mem($t2)
        li $t2, 61
        lbu $t2, mem($t2)
        add $t1, $t1, $t2
        li $t2, 63
        sb $t1, mem($t2)

        li $t2, 62
        lbu $t2, mem($t2)
        li $t3, 0
        beq $t2, $t3, b3
 #  add $t5, $t5, $s5          # Increment byte_occurrence
        li $t2, 64
        lbu $t2 , mem($t2)

        li $t3, 61
        lbu $t1, mem($t3)
        add $t2, $t2, $t1
        li $t3, 64
        sb $t2, mem($t3)
        #li $t2, 0              # Set byte_flag to false
        li $t3, 62
```

```
        li $t2, 0
        sb $t2, mem($t3)
        lbu $t3, p3
  # j c4
  beq $t3, $t3, c4
m4:
        #add $t4, $t4, $s5      # Increment total_occurrence
        li $t2, 63
        lbu $t1, mem($t2)
        li $t2, 61
        lbu $t2, mem($t2)
        add $t1, $t1, $t2
        li $t2, 63
        sb $t1, mem($t2)

        li $t2, 62
        lbu $t2, mem($t2)
        li $t3, 0
        beq $t2, $t3, b4
 #  add $t5, $t5, $s5            # Increment byte_occurrence
        li $t2, 64
        lbu $t2 , mem($t2)

        li $t3, 61
        lbu $t1, mem($t3)
        add $t2, $t2, $t1
        li $t3, 64
        sb $t2, mem($t3)
        #li $t2, 0                # Set byte_flag to false
        li $t3, 62
        li $t2, 0
        sb $t2, mem($t3)
        lbu $t3, p3
        #j check_1
        beq $t3, $t3, check_1
b1:
        #j c2
        beq $t3, $t3, c2
b2:
        #j c3
        beq $t3, $t3, c3
b3:
        #j c4
        beq $t3, $t3, c4
```

```
b4:
        #j check_1
        beq $t3, $t3, check_1

#below  this line  is for continous only
 check_1:
        #comparison checking

        #add $s1, $t0, $t1      # Calculate address of mem[i +1]
        #lbu $s2, 0($s1)        # Load a byte from mem[i] + j into $s2
        #lbu $t8, 1($t0)        # Load the first byte of p3 into $t8
        li $t2, 60
        lbu $t1, mem($t2)
        li $t2, 61
        lbu $t2, mem($t2)
        add $t1, $t1, $t2
        lbu $t2, mem($t1) #loading mem[i] into t2


        lbu $t3, mem($t1)
        #now have mem[i] in t2  and mem [i+1] in t4


        # Extract bits 4-7 from mem[i] and bit 0 from mem[i+1]
        li $t1, 30
  # andi $t2, $t2, 0x1E   # Keep only bits 1-4 of $s2
  and $t2, $t2, $t1
  # sll $t2, $t2, 1  # Shift the value in $s2 left by one position
        add $t2, $t2, $t2
        li $t1, 128
        #andi $t3, $t3, 0x80   # Keep only bit 8 of $t8
        and $t3, $t3, $t1
        #srl $t3, $t3, 7 #shift bit 7 to the right most position of bit 1
        srl $t3, $t3, 3
        srl $t3, $t3, 3
        srl $t3, $t3, 1

  # sll $t8, $t8, 4        # Shift bit 7 to the leftmost position (bit 4)
        or $t2, $t2, $t3          # Combine bits 0-3 from $s2 with bit 4 from $t8
        lbu $t3 , p3
        beq $t2, $t3, match_1
 check_2:
        #comparison checking
        #add $s1, $t0, $t1      # Calculate address of mem[i + 1]
```

```
        #lbu $s2, 0($s1)        # Load a byte from mem[i] + j into $s2
        #lbu $t8, 1($t0)        # Load the first byte of p3 into $t8
        li $t2, 60
        lbu $t1, mem($t2)
        li $t2, 61
        lbu $t2, mem($t2)
        add $t1, $t1, $t2
        lbu $t2, mem($t1) #loading mem[i] into t2
        lbu $t3, mem($t1)


        # Extract bits 5-7 from mem[i] and bits 0-1 from mem[i+1]
  # andi $t2, $t2, 0x0E   # Keep only bits 1-3 of $s2
  li $t1, 14
  and $t2,$t2,$t1
  # andi $t3, $t3, 0xC0   # Keep only bits 7-8 of $t8
  li $t1, 192
  and $t3 $t3 $t1
  # srl $t3, $t3, 6       # Shift bits 7-8 of $t8 to the rightmost posit
        srl $t3, $t3, 3
        srl $t3, $t3, 3
        or $t2, $t2, $t3
        lbu $t3, p3
        beq $t2, $t3, match_2
check_3:
        #comparison checking
        #add $s1, $t0, $t1      # Calculate address of mem[i] + j
  # lbu $s2, 0($s1)     # Load a byte from mem[i] + j into $s2
  # lbu $t8, 1($t0)       # Load the first byte of p3 into $t8
        li $t2, 60
        lbu $t1, mem($t2)
        li $t2, 61
        lbu $t2, mem($t2)
        add $t1, $t1, $t2
        lbu $t2, mem($t1) #loading mem[i] into t2
        lbu $t3, mem($t1)


        # Extract bits 1-2 from mem[i] and bits 6-8 from mem[i+1]
        li $t1, 6
        and $t2, $t2, $t1   # Keep only bits 1-2 of $s2
        #sll $t2, $t2, 3
        add $t2, $t2, $t2
        add $t2, $t2, $t2
```

```
        add $t2, $t2, $t2
        li $t1 224
        and $t3, $t3, $t1   # Keep only bits 6-8 of $t8
 # srl $t3, $t3, 5        # Shift bits 6-8 of $t8 to the rightmost position
 srl $t3, $t3, 3          # Shift bits 6-8 of $t8 to the rightmost position
 srl $t3, $t3, 2
        or $t2, $t2, $t3
        lbu $t3, p3
        beq $t2, $t3, match_3

check_4:
        #comparison checking
 # add $s1, $t0, $t1             # Calculate address of mem[i] + j
#  lbu $s2, 0($s1)     # Load a byte from mem[i] + j into $s2
        #lbu $t8, 0($t0)          # Load the first byte of p3 into $t8
        li $t2, 60
        lbu $t1, mem($t2)
        li $t2, 61
        lbu $t2, mem($t2)
        add $t1, $t1, $t2
        lbu $t2, mem($t1) #loading mem[i] into t2
        lbu $t3, mem($t1)

 # Extract bit 1 from mem[i] and bits 5-8 from mem[i+1]
        li $t1, 2
        and $t2, $t2, $t1   # Keep only bit 1 of $s2
#  sll $t2, $t2, 4        # Shift bit 1 to the left 4 times

        add $t2, $t2, $t2
        add $t2, $t2, $t2
        add $t2, $t2, $t2
        add $t2, $t2, $t2
        li $t1 ,240
        and $t3, $t3, $t1   # Keep only bits 5-8 of $t8
        srl $t3, $t3, 3
        srl $t3, $t3, 1
        or $t2, $t2,$t3
        lbu $t3, p3
        beq $t2, $t3, match_4

        li $t3, 60
        lbu $t1, mem($t3)
        li $t3, 61
        lbu $t2 mem($t3)
```

```
        add $t1, $t1, $t2
        li $t3, 60
        sb $t1, mem($t3)
        lbu $t3, p3
        #add $t1, $t1, $s5      # Increment i

        #j outer_loop
        beq $t3, $t3, outer_loop
match_1:


        #add $t4, $t4, $s5      # Increment total_occurrence
        li $t3, 63
        lbu $t1, mem ($t3)
        li $t3, 61
        lbu $t2, mem($t3)
        add $t1, $t1, $t2
        li $t3, 63
        sb $t1, mem($t3)
        li $t3, 62
        lbu $t2, mem ($t3)
        li $t3, 0
        beq $t2, $t3, byte_not_occurred1
 #  add $t5, $t5, $s5            # Increment byte_occurrence
 li $t3, 64
        lbu $t2 , mem($t3)
        li $t3, 61
        lbu $t1, mem($t3)
        add $t2, $t2, $t1
        li $t3, 64
        sb $t2, mem($t3)
        #li $t2, 0              # Set byte_flag to false
        li $t3, 62
        li $t2, 0
        sb $t2, mem($t3)
        #j check_2
        beq $t3, $t3, check_2
 match_2:


        #add $t4, $t4, $s5      # Increment total_occurrence
        li $t3, 63
        lbu $t1, mem ($t3)
        li $t3, 61
```

```
        lbu $t2, mem($t3)
        add $t1, $t1, $t2
        li $t3, 63
        sb $t1, mem($t3)
        li $t3, 62
        lbu $t2, mem ($t3)
        li $t1, 0
        beq $t2, $t1, byte_not_occurred2
#  add $t5, $t5, $s5            # Increment byte_occurrence
li $t3, 64
        lbu $t2 , mem($t3)
        li $t3, 61
        lbu $t1, mem($t3)
        add $t2, $t2, $t1
        li $t3, 64
        sb $t2, mem($t3)
        #li $t2, 0                # Set byte_flag to false
        li $t3, 62
        li $t2, 0
        sb $t2, mem($t3)
        #j check_3
        beq $t3, $t3, check_3
match_3:


        #add $t4, $t4, $s5      # Increment total_occurrence
        li $t3, 63
        lbu $t1, mem ($t3)
        li $t3, 61
        lbu $t2, mem($t3)
        add $t1, $t1, $t2
        li $t3, 63
        sb $t1, mem($t3)
        li $t3, 62
        lbu $t2, mem ($t3)
        li $t1, 0
        beq $t2, $t1, byte_not_occurred3
#  add $t5, $t5, $s5            # Increment byte_occurrence
li $t3, 64
        lbu $t2 , mem($t3)
        li $t3, 61
        lbu $t1, mem($t3)
        add $t2, $t2, $t1
        li $t3, 64
```

```
        sb $t2, mem($t3)
        #li $t2, 0              # Set byte_flag to false
        li $t3, 62
        li $t2, 0
        sb $t2, mem($t3)
        #j check_4
        beq $t3, $t3, check_4
match_4:
  # add $t4, $t4, $s5           # Increment total_occurrence
        li $t3, 63
        lbu $t1, mem($t3)
        li $t3, 63
        lbu $t2, mem($t3)
        add $t1, $t1, $t2
        li $t3, 63
        sb $t1, mem($t3)

        li $t3, 63
        lbu $t2, mem($t3)
        li $t1, 0
        beq $t2, $t1, byte_not_occurred4
  # add $t5, $t5, $s5           # Increment byte_occurrence
  li $t3, 63
        lbu $t2 , mem($t3)
        li $t3, 63
        lbu $t1, mem($t3)
        add $t2, $t2, $t1
        li $t3, 63
        sb $t2, mem($t3)

  # li $t2, 0             # Set byte_flag to false
 #  add $t1, $t1, $s5  # Increment i
        li $t3, 60
        lbu $t1, mem($t3)
        li $t3, 61
        lbu $t2 mem($t3)
        add $t1, $t1, $t2
        li $t3, 60
        sb $t1, mem($t3)
  # j outer_loop
  beq $t3, $t3, outer_loop
byte_not_occurred1:
        #j check_2
        beq $t3, $t3, check_2
```

```
byte_not_occurred2:
  # j check_3
  beq $t3, $t3, check_3
byte_not_occurred3:
  # j check_4
  beq $t3, $t3, check_4
byte_not_occurred4:
  # add $t1, $t1, $s5            # Increment i

        li $t3, 60
        lbu $t1, mem + 60
        li $t3, 61
        lbu $t2 mem($t3)
        add $t1, $t1, $t2
        li $t3, 60
        sb $t1, mem($t3)
        #j outer_loop
        beq $t3, $t3, outer_loop

end:
        lbu $t0, mem + 63
        lbu $t1, mem + 64
        sw $t0, total_occurrence       # Store total_occurrence back to memory
        sw $t1, byte_occurrence        # Store byte_occurrence back to memory

        # Print the result message
        la $a0, total_occurrence_msg          # Load the address of the total_occurrence
message
        li $v0, 4                      # Load syscall code for printing string
        syscall

        # Print the value of total_occurrence
        lw $a0, total_occurrence       # Load the value of total_occurrence into $a0
        li $v0, 1                      # Load syscall code for printing integer
        syscall

        # Print the result message for byte_occurrence
        la $a0, byte_occurrence_msg           # Load the address of the byte_occurrence
message
        li $v0, 4                      # Load syscall code for printing string
        syscall

        # Print the value of byte_occurrence
        lw $a0, byte_occurrence        # Load the value of byte_occurrence into $a0
```

```
    li $v0, 1                   # Load syscall code for printing integer
    syscall

    # Exit the program
    li $v0, 10                  # Load syscall code for exit
    syscall

.data
    total_occurrence_msg: .asciiz "Total Occurrences: "
    byte_occurrence_msg: .asciiz "\nByte Occurrences: "
```