# CSE 141L Milestone 1

Sky (Ho Tin) Hung, A15909216; Yizhou Wang, A16145420; Shuo Wang, A16622869

## 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.

Sky Hung
Yizhou Wang
Shuo Wang

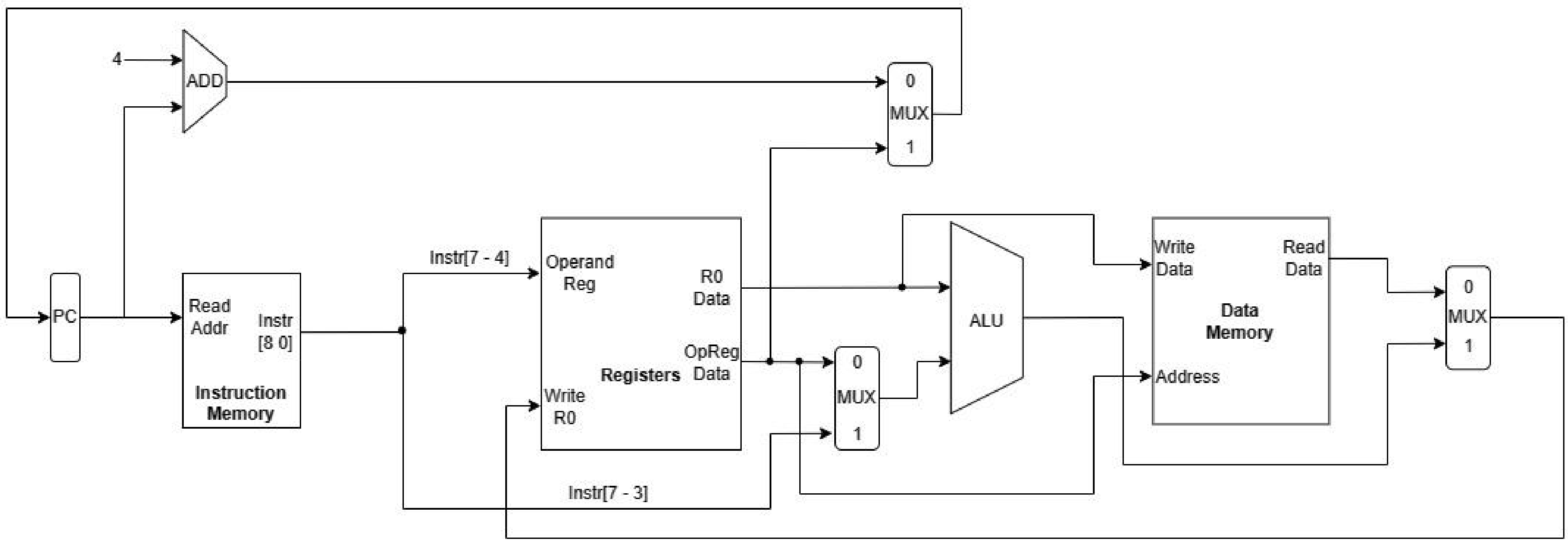
# 0. Team

Sky Hung
Yizhou Wang
Shuo Wang

# 1. Introduction

Name: **SIAA - Short Instruction Accumulator Architecture**
Our overall philosophy is to keep the entire architecture simple and straightforward. Our goal is to accomplish the functionality of the processor with a reduced-size and fully-functional instruction set. To accomplish this, we decided to use an accumulator machine where R0 is reserved as the accumulator. The accumulator will handle all the jobs of the destination register and one of the registers in R-type instructions. By doing this, we are able to keep the instructions short (within 9 bits) without having to reduce the number of registers or use varied-length register expressions in the machine code, which accomplishes our philosophy of being simple and straightforward. The name of the architecture is yet another place where our tenet of simplicity.

## 2. Architectural Overview



# 3. Machine Specification

## Instruction formats

| TYPE | FORMAT | CORRESPONDING INSTRUCTIONS |
|---|---|---|
| R | 1 bit type, 4 bits register, 4 bits funct | and, or, xor, lw, sw, add, sub, eq, slt, br, j, set, la |
| I | 1 bit type, 5 bits immediate, 3 bits funct | sll, srl, seti, halt |

## Operations

| NAME | TYPE | BIT BREAKDOWN | EXAMPLE | NOTES |
|---|---|---|---|---|
| and = logical and | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0000) | # Assume R0 (accumulator) has `0b0001_0001`<br># Assume R2 has `0b1001_0000`<br><br>`and R2 ⇔ 0_0010_0000`<br><br># after and instruction, R0 now holds `0b0001_0000` | Bitwise AND values stored in accumulator and the given register (e.g. R2) and store the result in the destination register R0 (accumulator). |
| or = logical or | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0001) | # Assume R0 (accumulator) has `0b0001_0001`<br># Assume R2 has `0b1001_0000`<br><br>`or R2 ⇔ 0_0010_0001`<br><br># after or instruction, R0 now holds `0b1001_0001` | Bitwise OR values stored in accumulator and the given register (e.g. R2) and store the result in the destination register R0 (accumulator). |

| | | | | |
|---|---|---|---|---|
| xor = logical xor | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0010) | # Assume R0 (accumulator) has `0b0001_0001`<br># Assume R2 has `0b1001_0000`<br><br>`xor R2 ⇔ 0_0010_0010`<br><br># after xor instruction, R0 now holds `0b1000_0001` | Bitwise XOR values stored in accumulator and the given register (e.g. R2) and store the result in the destination register R0 (accumulator). |
| lw = load word | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0011) | # Assume R0 (accumulator) has `0b0001_0001`<br># Assume R2 has `0b0001_0000 ⇔ 16`<br><br>`lw R2 ⇔ 0_0010_0011`<br><br># after lw instruction, R0 now holds dataMem[16] | Load word from data memory to the accumulator register R0. The memory address is specified by the value stored in the operand register (e.g. R2) |
| sw = store word | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0100) | # Assume R0 (accumulator) has `0b0001_0001`<br># Assume R2 has `0b0001_0000 ⇔ 16`<br><br>`sw R2 ⇔ 0_0010_0100`<br><br># after sw instruction, dataMem[16] now holds `0b0001_0001` | Store word in the accumulator R0 into the data memory. The destination memory address is specified by the value held by the operand register (e.g. R2) |
| add = arithmetic addition | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0101) | # Assume R0 (accumulator) has `0b0001_0001 ⇔ 17`<br># Assume R2 has `0b0001_0000 ⇔ 16`<br><br>`add R2 ⇔ 0_0010_0101` | Perform arithmetic addition between the value stored by the accumulator R0 and the operand register (e.g. R2). Store the result in the accumulator R0. |

| | | | | |
|---|---|---|---|---|
| | | | # after add instruction, R0 now holds `0b0010_0001` ⇔ `33` | |
| sub = arithmetic subtraction | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0110) | # Assume R0 (accumulator) has `0b0001_0001` ⇔ `17`<br># Assume R2 has `0b0001_0000` ⇔ `16`<br><br>`sub R2` ⇔ `0_0010_0110`<br><br># after sub instruction, R0 now holds `0b0000_0001` ⇔ `1` | Perform 2's complement subtraction. Subtract value of R2 from value of R0 (R2 - R0). The result will be stored in the accumulator R0. |
| sll = shift left logical | I | 1 bit type (1), 5 bit immediate (XXXXX), 3 bit funct (000) | # Assume R0 (accumulator) has `0b0001_0001`<br><br>`sll 2` ⇔ `1_00010_000`<br><br># after sll instruction, R0 now holds `0b0100_0100` | Shift leftward the value stored in the accumulator R0. The number of bits shifted is designated by the immediate. Result will be stored in R0. |
| srl = shift right logical | I | 1 bit type (1), 5 bit immediate (XXXXX), 3 bit funct (001) | # Assume R0 (accumulator) has `0b0001_0001`<br><br>`srl 2` ⇔ `1_00010_001`<br><br># after srl instruction, R0 now holds `0b0000_0100` | Shift rightward the value stored in the accumulator R0. The number of bits shifted is designated by the immediate. Result will be stored in R0. |
| eq = conditional equality | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0111) | # Assume R0 (accumulator) has `0b0001_0001`<br># Assume R2 has `0b0001_0001`<br><br>`eq R2` ⇔ `0_0010_0111`<br><br># after eq instruction, R0 now holds | Compare the value held by the accumulator R0 and the operand register (e.g. R2). If they are the same, R0 will store value 1 otherwise 0. |

| | | | | |
|---|---|---|---|---|
| | | | `0b0000_0001`<br><br># For any other value of R2, R0 would hold `0b0000_0000` | |
| slt = set on less than | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (1000) | # Assume R0 (accumulator) has `0b0000_0001`<br># Assume R2 has `0b0001_0001`<br><br>`slt R2 ⇔ 1_0010_1000`<br><br># after slt instruction, R0 now holds `0b0000_0001`<br><br># If R0 is larger or equal to R2, R0 would hold `0b0000_0000` | Compare the values held by the accumulator R0 and the operand register (e.g. R2). If R0 < R2, R0 will store 1, otherwise, R0 will store 0. |
| br = branch | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct (1001) | # Assume R0 (accumulator) has `0b0000_0001`<br><br># Assume R2 has value 28<br><br>`br R2 ⇔ 0_0010_1001`<br><br># After br instruction, PC will be at 28<br><br># If R0 is 0, proceed without branching | If the accumulator R0 stores the value logic TRUE (1), the program counter will branch to the destination designated by the value of the operand register. Otherwise, proceed without branching. |
| j = jump | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct (1010) | # Assume R2 has value 28<br><br>`j R2 ⇔ 0_0010_1010`<br><br># After j instruction, PC will be at 28 | Jump to the target address specified by the value of the operand register. Difference to br: jump regardless of any other conditions. |
| seti = | I | 1 bit type (1), 5 bit | # Assume R0 (accumulator) has | Assign the immediate to the accumulator |

| | | | | |
|---|---|---|---|---|
| set immediate to accumulator | | immediate (XXXXX), 3 bit funct (010) | `0b0001_0001`<br><br>`seti 2` ⇔ `1_00010_010`<br><br># after stl instruction, R0 now holds `0b0000_0010` | R0. Let R0 store that value |
| set = set to register | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (1011) | # Assume R0 (accumulator) has `0b0001_0001`<br># Assume R2 has `0b1001_0000`<br><br>`set R2` ⇔ `0_0010_1011`<br><br># after set instruction, R2 now holds `0b0001_0001` | Assign the value of the accumulator R0 to the operand register (e.g. R2) |
| la = load to accumulator | R | 1 bit type (0), 4 bit register (XXXX), 4 bit funct code (1100) | # Assume R0 (accumulator) has `0b0001_0001`<br># Assume R2 has `0b1001_0000`<br><br>`la R2` ⇔ `0_0010_1100`<br><br># after la instruction, R0 now holds `0b1001_0000` | Load the value from the operand register (e.g. R2) to the accumulator R0. |
| halt = halt program | I | 1 bit type (1), 5 bit immediate (XXXXX), 3 bit funct (011) | `halt` ⇔ `0_XXXXX_011`<br>Immediate don’t care<br><br># After halt instruction, program will terminate | Kill the program |

## Internal Operands

There will be 16 registers in our design. R0 will be the accumulator. R1 - R15 will be general-purpose registers.

## Control Flow (branches)

Both conditional and unconditional branches are supported. Instruction ‘br’ is the conditional branch. It executes the branching if the accumulator R0 stores the value 1, and does nothing if R0 stores the value 0. Instruction ‘j’ is the unconditional branch. It jumps the program counter to the destination regardless of any condition. The target address will be specified using a register. The operand register will store an 8-bit memory address. The maximum branch distance will be 2^7 addresses because a register will hold an 8-bit value. We have considered using I-type instruction at the beginning but later switched to R-type in order to accommodate larger jumps. For I-type, the maximum address length will be only 5 bits, but using R-type allows us to expand the address to 8 bits. Jumps larger than this range are not supported in the current version.

## Addressing Modes

We adopt indirect addressing. The target address of instruction ‘lw’ and ‘sw’ are specified by a register. The value of that register will be incremented to the destination address. After executing the ‘lw’ instruction, R0 will be loaded with the word at the memory address specified by the operand register. The process for ‘sw’ would be similar except that it would be writing from accumulator R0 to the memory address specified by the operand register.

Example:

```
seti  15    # R0 = 15
set   R2    # R2 = 15
add   R2    # R0 = R0 + R2 = 30
lw    R1    # Read from memory address 30; R0 = mem[30]
```

# 4. Programmer's Model [Lite]

Our machine is an accumulator machine, which utilizes a special register R0 to perform operations. While this architecture allows a simple and straightforward instruction format, it brings the cost of increasing the number of instructions. Therefore, the programmer must keep in mind that a clever use of the accumulator would significantly increase the efficiency of the program. Also, the programmer must be careful not to corrupt the accumulator data by overriding it with something else: store the current accumulator data into a general-purpose register before you overwrite it. Programmers should smartly organize R1-R15 by assigning them special purposes before programming. In this architecture, there are fifteen general-purpose registers whose functionality in a specific program is up to the programmers' decisions. Assigning them special purposes would help keep the code clean and significantly reduce the probability of corrupting data. A recommended workflow would be: 1) read all the data from memory; 2) process data and assign to general-purpose registers; 3) do operations store all data back to the memory.

MIPS or ARM instructions CANNOT be copied directly to our ISA. The use of MIPS/ARM-like instructions was abandoned because of the limited instruction length. With the instruction length being only 9 bits, we would either have to limit the number of registers to 4 or adopt varied-length register expressions in our machine code, which will induce other serious difficulties. We instead used the accumulator architecture. In all R-type instructions, the default operand and destination register will be R0, the accumulator. With this design, the number of registers can be expanded to 16 with a simpler instruction format.

# 5. Program Implementation

## Program 1 Pseudocode

```
# Forward error correction block encoder
# Read from mem[0:29], Write to mem[30:59]
function hammingEncoding:
	for i from 0 to 14:
		# Read the 11-bit message from data mem
		LSW = [i]
		MSW = [i + 1]

		# Calculate p8 MSW part b11, b10, b9
		p8 = 0
		for j from 0 to 2:
			currBit = (MSW >> j) & 1
			p8 = p8 ^ currBit

		# Calculate p8 LSW part: b8, b7, b6, b5
		fiveToEight = LSW >> 4
		for j from 0 to 3:
			currBit = (fiveToEight >> j) & 1
			p8 = p8 ^ currBit

		# Calculate p4 MSW part: b11, b10, b9
		p4 = 0
		for j from 0 to 2:
			currBit = (MSW >> j) & 1
			p4 = p4 ^ currBit

		# Calculate p4 LSW part: b8, b4, b3, b2
```

```
maskedLSW = LSW & (0b1000_1110)
for j from 0 to 7:
     currBit = (MSW >> j) & 1
     p4 = p4 ^ currBit

# Calculate p2 MSW part: b11, b10
p2 = 0
tenEleven = MSW >> 1
for j from 0 to 1:
     currBit = (tenEleven >> j) & 1
     p2 = p2 ^ currBit

# Calculate p2 LSW part: b7, b6, b4, b3, b1
maskedLSW = LSW & (0b0110_1101)
for j from 0 to 7:
     currBit = (MSW >> j) & 1
     p4 = p4 ^ currBit

# Calculate p1 MSW part: b11, b9
p1 = 0
maskedMSW = MSW & (0b0000_0101)
for j from 0 to 7:
     currBit = (maskedMSW  >> j) & 1
     p1 = p1 ^ currBit

# Calculate p1 LSW part: b7, b5, b4, b2, b1
maskedLSW = LSW & (0b0101_1011)
for j from 0 to 7:
     currBit = (maskedLSW >> j) & 1
     p1 = p1 ^ currBit

# Calculate p0
```

```
p0 = 0
for j from 0 to 2:
     currBit = (MSW >> j) & 1
     p0 = p0 ^ currBit

for j from 0 to 7:
     currBit = (LSW >> j) & 1
     p0 = p0 ^ currBit

p0 = p0 ^ p8 ^ p4 ^ p2 ^ p1

# Insert parities into the message and write to memory
elevenToFive =  (MSW << 5) | ((LSW & 0b1111_0000) >> 3)
newMSW = elevenToFive | p8
mem[31 + i] = newMSW

fourToTwo = (LSW & 0b0000_1110) << 4
newLSW = fourToTwo | (p4 << 4)
one = (LSW & 0b0000_0001) << 4
newLSW = newLSW | (one << 3)
newLSW = newLSW | (p2 << 2)
newLSW = newLSW | (p1 << 1)
newLSW = newLSW | p0
mem[30 + i] = newLSW
```

## Program 1 Assembly Code

```
# R1: loop counter i
# R2: MSW
# R3: LSW
# R4: p4
# R8: p8
# R9: constant 15
# R10: p0
# R11: p1
# R12: p2
# R15: bit mask 0b0000_0001

seti  0     # R0 = 0
set   R1    # R1 = 0
LOOP:
      # Read data from data mem
      lw    R1    # R0 = mem[i]
      set   R2    # R2 = mem[i] = LSW
      seti  1     # R0 = 1
      add   R1    # R0 = i + 1
      lw    R1    # R1 = mem[i + 1]
      set   R3    # R3 = mem[i + 1] = MSW

      # Constant set
      seti 0b0000_0001           # R0 = 1
      set   R15   # R15 = 1
```

```
seti  15    # R0 = 15
set   R9    # R9 = 15
```

**# Loop condition modify**

```
seti  1     # R0 = 1
set   R6    # R6 = 1
la    R1    # R0 = R1
add   R6    # R0 = R0 + R6
set   R1    # R1 = R0  (equivalent to R1 += 1)
```

**# Calculate p8 MSW part (11, 10, 9)**

```
sub   R0    # R0 = 0 (clear)
set   R8    # R8 = 0; p8 = 0
la    R3    # R0 = R3 = MSW
and   R15   # R0 = b9
xor   R8    # R0 = R0 ^ r8 (b9 ^ R8)
set   R8    # R8 = R0

la    R3    # R0 = R3 = MSW
srl   1     # R0 = R0 >> 1
and   R15   # R0 = b10
xor   R8    # R0 = R0 ^ r8 (b10 ^ R8)
set   R8    # R8 = R0

la    R3    # R0 = R3 = MSW
srl   2     # R0 = R0 >> 2
and   R15   # R0 = b11
xor   R8    # R0 = R0 ^ r8 (b11 ^ R8)
set   R8    # R8 = R0
```

**# Calculate p8 LSW part (8, 7, 6, 5)**

```
sub   R0    # R0 = 0 (clear)
```

```
la    R2    # R0 = R2 = LSW
srl   4     # R0 = R0 >> 4
and   R15   # R0 = b5
xor   R8    # R0 = R0 ^ r8 (b5 ^ R8)
set   R8    # R8 = R0

la    R2    # R0 = R2 = LSW
srl   5     # R0 = R0 >> 5
and   R15   # R0 = b6
xor   R8    # R0 = R0 ^ r8 (b6 ^ R8)
set   R8    # R8 = R0

la    R2    # R0 = R2 = LSW
srl   6     # R0 = R0 >> 6
and   R15   # R0 = b7
xor   R8    # R0 = R0 ^ r8 (b7 ^ R8)
set   R8    # R8 = R0

la    R2    # R0 = R2 = LSW
srl   7     # R0 = R0 >> 7
and   R15   # R0 = b8
xor   R8    # R0 = R0 ^ r8 (b8 ^ R8)
set   R8    # R8 = R0

# Calculate p4 MSW part (11, 10, 9)
sub   R0    # R0 = 0 (clear)
set   R4    # R4 = 0; p4 = 0
la    R3    # R0 = R3 = MSW
and   R15   # R0 = b9
xor   R4    # R0 = R0 ^ R4 (b9 ^ Rr)
set   R4    # R4 = R0
```

```
la    R3    # R0 = R3 = MSW
srl   1     # R0 = R0 >> 1
and   R15   # R0 = b10
xor   R4    # R0 = R0 ^ R4 (b10 ^ R4)
set   R4    # R4 = R0

la    R3    # R0 = R3 = MSW
srl   2     # R0 = R0 >> 2
and   R15   # R0 = b11
xor   R4    # R0 = R0 ^ R4 (b11 ^ R4)
set   R4    # R4 = R0

# Calculate p4 LSW part (8, 4, 3, 2)
sub   R0    # R0 = 0 (clear)
la    R2    # R0 = R2 = LSW
srl   1     # R0 = R0 >> 1
and   R15   # R0 = b2
xor   R4    # R0 = R0 ^ R4 (b2 ^ R4)
set   R4    # R4 = R0

la    R2    # R0 = R2 = LSW
srl   2     # R0 = R0 >> 2
and   R15   # R0 = b3
xor   R4    # R0 = R0 ^ R4 (b3 ^ R4)
set   R4    # R4 = R0

la    R2    # R0 = R2 = LSW
srl   3     # R0 = R0 >> 3
and   R15   # R0 = b4
xor   R4    # R0 = R0 ^ R4 (b4 ^ R4)
set   R4    # R4 = R0
```

```
la   R2   # R0 = R2 = LSW
srl  7    # R0 = R0 >> 7
and  R15  # R0 = b8
xor  R4   # R0 = R0 ^ R4 (b8 ^ R4)
set  R4   # R8 = R0
```

**# Calculate p2 MSW part (11, 10)**

```
sub  R0   # R0 = 0 (clear)
set  R12  # R12 = 0; R12 = 0
la   R3   # R0 = R3 = MSW
srl  1    # R0 = R0 >> 1
and  R15  # R0 = b10
xor  R12  # R0 = R0 ^ R12 (b10 ^ R12)
set  R12  # R12 = R0

la   R3   # R0 = R3 = MSW
srl  2    # R0 = R0 >> 2
and  R15  # R0 = b11
xor  R12  # R0 = R0 ^ R12 (b11 ^ R12)
set  R12  # R12 = R0
```

**# Calculate p2 LSW part (7, 6, 4, 3, 1)**

```
sub  R0   # R0 = 0 (clear)
la   R2   # R0 = R2 = LSW
and  R15  # R0 = b1
xor  R12  # R0 = R0 ^ R12 (b1 ^ R12)
set  R12  # R12 = R0

la   R2   # R0 = R2 = LSW
srl  2    # R0 = R0 >> 2
and  R15  # R0 = b3
xor  R12  # R0 = R0 ^ R12 (b3 ^ R12)
```

```
set   R12   # R12 = R0

la    R2    # R0 = R2 = LSW
srl   3     # R0 = R0 >> 3
and   R15   # R0 = b4
xor   R12   # R0 = R0 ^ R12 (b4 ^ R12)
set   R12   # R12 = R0

la    R2    # R0 = R2 = LSW
srl   5     # R0 = R0 >> 5
and   R15   # R0 = b6
xor   R12   # R0 = R0 ^ R12 (b6 ^ R12)
set   R12   # R12 = R0

la    R2    # R0 = R2 = LSW
srl   6     # R0 = R0 >> 6
and   R15   # R0 = b7
xor   R12   # R0 = R0 ^ R12 (b7 ^ R12)
set   R12   # R12 = R0

# Calculate p1 MSW part (11, 9)
sub   R0    # R0 = 0 (clear)
set   R11   # R12 = 0; R12 = 0
la    R3    # R0 = R3 = MSW
and   R15   # R0 = b9
xor   R11   # R0 = R0 ^ R11 (b9 ^ R11)
set   R11   # R11 = R0

la    R3    # R0 = R3 = MSW
srl   1     # R0 = R0 >> 2
and   R15   # R0 = b11
xor   R11   # R0 = R0 ^ R11 (b11 ^ R11)
```

```
set   R11   # R11 = R0
```

**# Calculate p1 LSW part (7, 5, 4, 2, 1)**

```
sub   R0    # R0 = 0 (clear)
la    R2    # R0 = R2 = LSW
and   R15   # R0 = b1
xor   R11   # R0 = R0 ^ R11 (b1 ^ R11)
set   R11   # R11 = R0

la    R2    # R0 = R2 = LSW
srl   1     # R0 = R0 >> 1
and   R15   # R0 = b2
xor   R11   # R0 = R0 ^ R11 (b2 ^ R11)
set   R11   # R11 = R0

la    R2    # R0 = R2 = LSW
srl   3     # R0 = R0 >> 3
and   R15   # R0 = b4
xor   R11   # R0 = R0 ^ R11 (b4 ^ R11)
set   R11   # R11 = R0

la    R2    # R0 = R2 = LSW
srl   4     # R0 = R0 >> 4
and   R15   # R0 = b5
xor   R11   # R0 = R0 ^ R11 (b5 ^ R11)
set   R11   # R11 = R0

la    R2    # R0 = R2 = LSW
srl   6     # R0 = R0 >> 6
and   R15   # R0 = b7
xor   R11   # R0 = R0 ^ R11 (b7 ^ R11)
set   R11   # R11 = R0
```

```
# Calculate p0 MSW part (11, 10, 9)
sub  R0   # R0 = 0 (clear)
set  R10  # R10 = 0; p0 = 0
la   R3   # R0 = R3 = MSW
and  R15  # R0 = b9
xor  R10  # R0 = R0 ^ R10 (b9 ^ R10)
set  R8   # R10 = R0

la   R3   # R0 = R3 = MSW
srl  1    # R0 = R0 >> 1
and  R15  # R0 = b10
xor  R10  # R0 = R0 ^ R10 (b10 ^ R10)
set  R10  # R10 = R0

la   R3   # R0 = R3 = MSW
srl  2    # R0 = R0 >> 2
and  R15  # R0 = b11
xor  R10  # R0 = R0 ^ R10 (b11 ^ R10)
set  R10  # R10 = R0

# Calculate p0 LSW part (8, 7, 6, 5, 4, 3, 2, 1)
sub  R0   # R0 = 0 (clear)
la   R2   # R0 = R2 = LSW
and  R15  # R0 = b1
xor  R10  # R0 = R0 ^ R10 (b1 ^ R10)
set  R10  # R10 = R0

la   R2   # R0 = R2 = LSW
srl  1    # R0 = R0 >> 1
and  R15  # R0 = b2
xor  R10  # R0 = R0 ^ R10 (b2 ^ R10)
```

```
set  R10  # R10 = R0

la   R2   # R0 = R2 = LSW
srl  2    # R0 = R0 >> 2
and  R15  # R0 = b3
xor  R10  # R0 = R0 ^ R10 (b3 ^ R10)
set  R10  # R10 = R0

la   R2   # R0 = R2 = LSW
srl  3    # R0 = R0 >> 3
and  R15  # R0 = b4
xor  R10  # R0 = R0 ^ R10 (b4 ^ R10)
set  R10  # R10 = R0

la   R2   # R0 = R2 = LSW
srl  4    # R0 = R0 >> 4
and  R15  # R0 = b5
xor  R10  # R0 = R0 ^ R10 (b5 ^ R10)
set  R10  # R10 = R0

la   R2   # R0 = R2 = LSW
srl  5    # R0 = R0 >> 5
and  R15  # R0 = b6
xor  R10  # R0 = R0 ^ R10 (b6 ^ R10)
set  R10  # R10 = R0

la   R2   # R0 = R2 = LSW
srl  6    # R0 = R0 >> 6
and  R15  # R0 = b7
xor  R10  # R0 = R0 ^ R10 (b7 ^ R10)
set  R10  # R10 = R0
```

```
la    R2    # R0 = R2 = LSW
srl   7     # R0 = R0 >> 7
and   R15   # R0 = b8
xor   R10   # R0 = R0 ^ R10 (b8 ^ R10)
set   R10   # R10 = R0

# Calculate p0 partiy part
la    R10   # R0 = R10 (p0)
xor   R8    # R0 = R0 ^ R8 (p8)
xor   R4    # R0 = R0 ^ R4 (p4)
xor   R12   # R0 = R0 ^ R12 (p2)
xor   R11   # R0 = R0 ^ R11 (p1)
set   R10   # R10 = R0 (p0)


# New MSW address
sub   R0    # R0 = 0 (clear)
seti  31    # R0 = 31
add   R1    # R0 = R0 + i (loop counter)
set   R14   # R14 = R0 (31 + i, the new MSW destination address)

# New MSW
seti  0b1111_0000     # R0 = 0b1111_0000 (temporary mask)
set   R5    # R5 = R0 = 0b1111_0000
sub   R0    # R0 = 0 (clear)
la    R3    # R0 = R3 (MSW)
sll   5     # R0 << 5
set   R6    # R6 = R0
la    R2    # R0 = R2 (LSW)
and   R5    # R0 = R0 & R5 (R0 & 0b1111_0000)
srl   3     # R0 = R0 >> 3
or    R6    # R0 = R0 | R6; (MSW << 5) | ((LSW & 0b1111_0000) >> 3))
```

```
or   R8   # R0 = R0 | R8 (R0 | p8)
sw   R14  # datamem[31 + i] = R0

# New LSW address
sub  R0   # R0 = 0 (clear)
seti 30   # R0 = 30
add  R1   # R0 = R0 + i (loop counter)
set  R13  # R13 = R0 (30 + i, the new LSW destination address)

# New LSW
sub  R0   # R0 = 0 (clear)
seti 0b0000_1110     # R0 = 0b0000_1110 (temporary mask)
set  R5   # R5 = R0 = 0b0000_1110
la   R2   # R0 = R2 (LSW)
and  R5   # R0 = R0 & R5 (R0 & 0b1111_0000)
sll  4    # R0 = R0 << 4
set  R6   # R6 = R0

la   R4   # R0 = R4 (p4)
sll  4    # R0 = R0 << 4
or   R6   # R0 = R0 | R6
set  R6   # R6 = R0  (newLSW)

la   R2   # R0 = R2 (LSW)
and  R15  # R0 = R0 & R15 (R0 & 0b0000_0001)
sll  7    # R0 = R0 << 7
set  R7   # R7 = R0

la   R12  # R0 = R12 (p2)
sll  2    # R0 = R0 << 2 (p2 << 2)
set  R12  # R12 = R0 (p2 after shift)
la   R11  # R0 = R11 (p1)
```

```
sll   1     # R0 = R0 << 1 (p1 << 1)
set   R11   # R11 = R0 (p1 after shift)

la    R6    # R0 = R6
or    R7    # R0 = R0 | R7
or    R12   # R0 = R0 | R12 (newLSW | (p2 << 2))
or    R11   # R0 = R0 | R11 (newLSW | (p1 << 1))
or    R10   # R0  = R0 | R10 (newLSW | p0)
sw    R13   # datamem[30 + i] = R0

# Loop condition check
la    R1    # R0 = R1
slt   R9    # R0 = (R0 < 15)
br    LOOP  # Iterate from the start (LOOP flag)
halt        # Otherwise, terminate the program
```

## Program 2 Pseudocode

```
# forward error correction block decoder/receiver
# write into mem[0:29]
decoder():
  index = 30
  while index < 60:
    # Take parity of all 16 incoming bits
    # mem[index]:    b11 b10  b9  b8  b7  b6  b5  p8
    # mem[index + 1]: b4  b3  b2  p4  b1  p2  p1  p0
    num_for_xor_digits = mem[index] ^ mem[index + 1]
    ones = 0
    while num_for_xor_digits != 0:
      digit = num_for_xor_digits & 1
      ones += digit
      num_for_xor_digits >>= 1
      if ones == 2:
        ones = 0
    one_error_sign = ones
    # p8 = ^(b11:b5)
    mem_index = mem[index] & 8'b11111110
    mem_index1 = mem[index + 1] & 0
    num_for_xor_digits = mem_index ^ mem_index1
    ones = 0
    while num_for_xor_digits != 0:
      digit = num_for_xor_digits & 1
      ones += digit
      num_for_xor_digits >>= 1
      if ones == 2:
        ones = 0
    # affected parity analysis: p8
    parity_affected = 0
```

```
parity_check = mem[index] & 1 # real p8
parity_check = ones ^ parity_check
parity_check <<= 3
parity_affected += parity_check
# p4 = ^(b11:b8,b4,b3,b2)
mem_index = mem[index] & 8'b11110000
mem_index1 = mem[index + 1] & 8'b11100000
num_for_xor_digits = mem_index ^ mem_index1
ones = 0
while num_for_xor_digits != 0:
  digit = num_for_xor_digits & 1
  ones += digit
  num_for_xor_digits >>= 1
  if ones == 2:
    ones = 0
# affected parity analysis: p4
parity_check = mem[index + 1] & 8'b00010000
parity_check >>= 4 # real p4
parity_check = ones ^ parity_check
parity_check <<= 2
parity_affected += parity_check
# p2 = ^(b11,b10,b7,b6,b4,b3,b1)
mem_index = mem[index] & 8'b11001100
mem_index1 = mem[index + 1] & 8'b11001000
num_for_xor_digits = mem_index ^ mem_index1
ones = 0
while num_for_xor_digits != 0:
  digit = num_for_xor_digits & 1
  ones += digit
  num_for_xor_digits >>= 1
  if ones == 2:
    ones = 0
```

```
# affected parity analysis: p2
parity_check = mem[index + 1] & 8'b00000100
parity_check >>= 2 # real p2
parity_check = ones ^ parity_check
parity_check <<= 1
parity_affected += parity_check
# p1 = ^(b11,b9,b7,b5,b4,b2,b1)
mem_index = mem[index] & 8'b10101010
mem_index1 = mem[index + 1] & 8'b10101000
num_for_xor_digits = mem_index ^ mem_index1
ones = 0
while num_for_xor_digits != 0:
  digit = num_for_xor_digits & 1
  ones += digit
  num_for_xor_digits >>= 1
  if ones == 2:
    ones = 0
# affected parity analysis: p1
parity_check = mem[index + 1] & 8'b00000010
parity_check >>= 1 # real p1
parity_check = ones ^ parity_check
parity_affected += parity_check
# parity_affected: 4 bits represent error for p8, p4, p2, p1
# prepare b11:b1
mem_index = mem[index] & 8'b11100000
mem_index >>= 5 # 0 0 0 0 0 b11 b10 b9
mem_index1 = mem[index + 1] & 8'b11101000
mem_index1 >>= 3
temp = mem_index1 & 1
mem_index1 >>= 1
mem_index1 += temp # 0 0 0 0 b4 b3 b2 b1
temp = mem[index] & 8'b00011110
```

```
temp <<= 3
mem_index1 += temp # b8 b7 b6 b5 b4 b3 b2 b1
# deal with one_error_sign
if one_error_sign == 1: # 1 error
  mem_index += 8’b01000000
  if parity_affected == 8’b00000011: # b1
    mem_index1 ^= 8’b00000001
  if parity_affected == 8’b00000101: # b2
    mem_index1 ^= 8’b00000010
  if parity_affected == 8’b00000110: # b3
    mem_index1 ^= 8’b00000100
  if parity_affected == 8’b00000111: # b4
    mem_index1 ^= 8’b00001000
  if parity_affected == 8’b00001001: # b5
    mem_index1 ^= 8’b00010000
  if parity_affected == 8’b00001010: # b6
    mem_index1 ^= 8’b00100000
  if parity_affected == 8’b00001011: # b7
    mem_index1 ^= 8’b01000000
  if parity_affected == 8’b00001100: # b8
    mem_index1 ^= 8’b10000000
  if parity_affected == 8’b00001101: # b9
    mem_index ^= 8’b00000001
  if parity_affected == 8’b00001110: # b10
    mem_index ^= 8’b00000010
  if parity_affected == 8’b00001111: # b11
    mem_index ^= 8’b00000100
else: # 0 or 2 errors
  if parity_affected != 0: # 2 errors
    mem_index += 8’b10000000
mem[index - 30] = mem_index
mem[index - 29] = mem_index1
```

```
    index += 2
```

## Program 2 Assembly Code

```
start:
  seti 30                           # R0 = 30
  set  R1                           # R1 = 30 (index)
  add  R1                           # R0 = 60
  set  R2                           # R2 = 60 (range of index)

loop_overall_start:
  la   R1                           # R0 = index
  slt  R2                           # R0 = R0 < R2 = index < 60
  set  R3                           # R3 = index < 60
  seti 1                            # R0 = 1
  xor  R3                           # R0 = R0 ^ R3 = 1 ^ (index < 60) = index >= 60
  br   loop_done                    # if index >= 60, go to loop_done

  lw   R1                           # R0 = mem[index]
  set  R3                           # R3 (mem[index])
  seti 1                            # R0 = 1
  add  R1                           # R0 = index + 1
  lw   R0                           # R0 = mem[index + 1]
  set  R4                           # R4 (mem[index + 1])
  la   R3                           # R0 = R3 (mem[index])
  xor  R4                           # R0 = R3 ^ R4 (mem[index] ^ mem[index + 1])
  set  R5                           # R5 = mem[index] ^ mem[index + 1] (num_for_xor_digits)
  seti 0                            # R0 = 0
  set  R6                           # R6 = 0 (ones)

loop_xor_digits_start:
  seti 0                            # R0 = 0
```

```
  eq   R5                          # R0 = (R0 == R5)
  br   loop_overall_p8             # if num_for_xor_digits == 0, go to loop_overall_p8
  seti 1                           # R0 = 1
  and  R5                          # R0 = R0 & R5 = 1 & num_for_xor_digits (digit)
  add  R6                          # R6 = R0 + R6 (ones += digit)
  la   R5                          # R0 = num_for_xor_digits
  srl  1                           # R0 = (num_for_xor_digits >> 1)
  set  R5                          # R5 = num_for_xor_digits = (num_for_xor_digits >> 1)
  set  2                           # R0 = 2
  eq   R6                          # R0 = (R0 == R6) = (ones == 2)
  br   loop_xor_digits_start_set_ones # if ones == 2, ones = 0
  j    loop_xor_digits_start       # loop

loop_xor_digits_start_set_ones:
  seti 0                           # R0 = 0
  set  R6                          # R6 = ones = 0
  j    loop_xor_digits_start       # loop

loop_overall_p8:
  la   R6                          # R0 = R6 = ones
  set  R7                          # R7 = ones (one_error_sign)

  seti 15                          # R0 = 15 = 8'b00001111
  sll  4                           # R0 = 8'b11110000
  set  R8                          # R8 = 8'b11110000
  seti 14                          # R0 = 14 = 8'b00001110
  add  R8                          # R0 = R0 + R8 = 8'b11111110
  and  R3                          # R0 = R0 & R3 = mem[index] & 8'b11111110
  set  R8                          # R8 = mem[index] & 8'b11111110 (mem_index)
  seti 0                           # R0 = 0
  and  R4                          # R0 = R0 & R4 = 0 & mem[index + 1]
  set  R9                          # R9 = mem[index + 1] & 0 (mem_index1)
```

```
  la   R8                           # R0 = R8
  xor  R9                           # R0 = R8 ^ R9 (mem_index ^ mem_index1)
  set  R5                           # R5 = mem_index ^ mem_index1 (num_for_xor_digits)
  seti 0                            # R0 = 0
  set  R6                           # R6 = 0 (ones)

loop_xor_digits_p8:
  seti 0                            # R0 = 0
  eq   R5                           # R0 = (R0 == R5)
  br   loop_overall_p4              # if num_for_xor_digits == 0, go to loop_overall_p4
  seti 1                            # R0 = 1
  and  R5                           # R0 = R0 & R5 = 1 & num_for_xor_digits (digit)
  add  R6                           # R6 = R0 + R6 (ones += digit)
  la   R5                           # R0 = num_for_xor_digits
  srl  1                            # R0 = (num_for_xor_digits >> 1)
  set  R5                           # R5 = num_for_xor_digits = (num_for_xor_digits >> 1)
  set  2                            # R0 = 2
  eq   R6                           # R0 = (R0 == R6) = (ones == 2)
  br   loop_xor_digits_p8_set_ones  # if ones == 2, ones = 0
  j    loop_xor_digits_p8           # loop

loop_xor_digits_p8_set_ones:
  seti 0                            # R0 = 0
  set  R6                           # R6 = ones = 0
  j    loop_xor_digits_p8           # loop

loop_overall_p4:
  seti 0                            # R0 = 0
  set  R10                          # R10 = 0 (parity_affected)
  seti 1                            # R0 = 1
  and  R3                           # R0 = R0 & R3 = mem[index] & 1 (parity_check)
  xor  R6                           # R0 = parity_check = R0 ^ R6 = ones ^ parity_check
```

```
  sll  3                            # R0 = (parity_check << 3)
  add  R10                          # R0 = R0 + R10 = parity_affected + parity_check
  set  R10                          # R10 = parity_affected = R0

  seti 15                           # R0 = 15 = 8'b00001111
  sll  4                            # R0 = 8'b11110000
  and  R3                           # R0 = R0 & R3 = mem[index] & 8'b11110000
  set  R8                           # R8 = mem[index] & 8'b11110000 (mem_index)
  seti 14                           # R0 = 14 = 8'b00001110
  sll  4                            # R0 = 8'b11100000
  and  R4                           # R0 = R0 & R4 = 8'b11100000 & mem[index + 1]
  set  R9                           # R9 = mem[index + 1] & 8'b11100000 (mem_index1)
  la   R8                           # R0 = R8
  xor  R9                           # R0 = R8 ^ R9 (mem_index ^ mem_index1)
  set  R5                           # R5 = mem_index ^ mem_index1 (num_for_xor_digits)
  seti 0                            # R0 = 0
  set  R6                           # R6 = 0 (ones)

loop_xor_digits_p4:
  seti 0                            # R0 = 0
  eq   R5                           # R0 = (R0 == R5)
  br   loop_overall_p2              # if num_for_xor_digits == 0, go to loop_overall_p2
  seti 1                            # R0 = 1
  and  R5                           # R0 = R0 & R5 = 1 & num_for_xor_digits (digit)
  add  R6                           # R6 = R0 + R6 (ones += digit)
  la   R5                           # R0 = num_for_xor_digits
  srl  1                            # R0 = (num_for_xor_digits >> 1)
  set  R5                           # R5 = num_for_xor_digits = (num_for_xor_digits >> 1)
  set  2                            # R0 = 2
  eq   R6                           # R0 = (R0 == R6) = (ones == 2)
  br   loop_xor_digits_p4_set_ones  # if ones == 2, ones = 0
  j    loop_xor_digits_p4           # loop
```

```
loop_xor_digits_p4_set_ones:
  seti 0                              # R0 = 0
  set  R6                             # R6 = ones = 0
  j    loop_xor_digits_p4             # loop

loop_overall_p2:
  seti 16                             # R0 = 16 = 8’b00010000
  and  R4                             # R0 = R0&R4 = mem[index+1] & 8’b00010000 (parity_check)
  srl  4                              # R0 = (parity_check >> 4)
  xor  R6                             # R0 = parity_check = R0 ^ R6 = ones ^ parity_check
  sll  2                              # R0 = (parity_check << 2)
  add  R10                            # R0 = R0 + R10 = parity_affected + parity_check
  set  R10                            # R10 = parity_affected = R0

  seti 12                             # R0 = 12 = 8’b00001100
  sll  4                              # R0 = 8’b11000000
  set  R8                             # R8 = R0 = 8’b11000000
  seti 12                             # R0 = 12 = 8’b00001100
  add  R8                             # R0 = R0 + R8 = 8’b11001100
  and  R3                             # R0 = R0 & R3 = mem[index] & 8’b11001100
  set  R8                             # R8 = mem[index] & 8’b11001100 (mem_index)
  seti 12                             # R0 = 12 = 8’b00001100
  sll  4                              # R0 = 8’b11000000
  set  R9                             # R9 = R0 = 8’b11000000
  seti 8                              # R0 = 8 = 8’b00001000
  add  R9                             # R0 = R0 + R9 = 8’b11001000
  and  R4                             # R0 = R0 & R4 = 8’b11001000 & mem[index + 1]
  set  R9                             # R9 = mem[index + 1] & 8’b11001000 (mem_index1)
  la   R8                             # R0 = R8
  xor  R9                             # R0 = R8 ^ R9 (mem_index ^ mem_index1)
  set  R5                             # R5 = mem_index ^ mem_index1 (num_for_xor_digits)
```

```
  seti 0                              # R0 = 0
  set  R6                             # R6 = 0 (ones)

loop_xor_digits_p2:
  seti 0                              # R0 = 0
  eq   R5                             # R0 = (R0 == R5)
  br   loop_overall_p1                # if num_for_xor_digits == 0, go to loop_overall_p1
  seti 1                              # R0 = 1
  and  R5                             # R0 = R0 & R5 = 1 & num_for_xor_digits (digit)
  add  R6                             # R6 = R0 + R6 (ones += digit)
  la   R5                             # R0 = num_for_xor_digits
  srl  1                              # R0 = (num_for_xor_digits >> 1)
  set  R5                             # R5 = num_for_xor_digits = (num_for_xor_digits >> 1)
  set  2                              # R0 = 2
  eq   R6                             # R0 = (R0 == R6) = (ones == 2)
  br   loop_xor_digits_p2_set_ones    # if ones == 2, ones = 0
  j    loop_xor_digits_p2             # loop

loop_xor_digits_p2_set_ones:
  seti 0                              # R0 = 0
  set  R6                             # R6 = ones = 0
  j    loop_xor_digits_p2             # loop

loop_overall_p1:
  seti 4                              # R0 = 4 = 8’b00000100
  and  R4                             # R0 = R0&R4 = mem[index+1] & 8’b00000100 (parity_check)
  srl  2                              # R0 = (parity_check >> 2)
  xor  R6                             # R0 = parity_check = R0 ^ R6 = ones ^ parity_check
  sll  1                              # R0 = (parity_check << 1)
  add  R10                            # R0 = R0 + R10 = parity_affected + parity_check
  set  R10                            # R10 = parity_affected = R0
```

```
  seti 10                            # R0 = 10 = 8'b00001010
  sll  4                             # R0 = 8'b10100000
  set  R8                            # R8 = R0 = 8'b10100000
  seti 10                            # R0 = 10 = 8'b00001010
  add  R8                            # R0 = R0 + R8 = 8'b10101010
  and  R3                            # R0 = R0 & R3 = mem[index] & 8'b10101010
  set  R8                            # R8 = mem[index] & 8'b10101010 (mem_index)
  seti 10                            # R0 = 10 = 8'b00001010
  sll  4                             # R0 = 8'b10100000
  set  R9                            # R9 = R0 = 8'b10100000
  seti 8                             # R0 = 8 = 8'b00001000
  add  R9                            # R0 = R0 + R9 = 8'b10101000
  and  R4                            # R0 = R0 & R4 = 8'b10101000 & mem[index + 1]
  set  R9                            # R9 = mem[index + 1] & 8'b10101000 (mem_index1)
  la   R8                            # R0 = R8
  xor  R9                            # R0 = R8 ^ R9 (mem_index ^ mem_index1)
  set  R5                            # R5 = mem_index ^ mem_index1 (num_for_xor_digits)
  seti 0                             # R0 = 0
  set  R6                            # R6 = 0 (ones)

loop_xor_digits_p1:
  seti 0                             # R0 = 0
  eq   R5                            # R0 = (R0 == R5)
  br   loop_overall_prepare          # if num_for_xor_digits == 0, go to loop_overall_prepare
  seti 1                             # R0 = 1
  and  R5                            # R0 = R0 & R5 = 1 & num_for_xor_digits (digit)
  add  R6                            # R6 = R0 + R6 (ones += digit)
  la   R5                            # R0 = num_for_xor_digits
  srl  1                             # R0 = (num_for_xor_digits >> 1)
  set  R5                            # R5 = num_for_xor_digits = (num_for_xor_digits >> 1)
  set  2                             # R0 = 2
  eq   R6                            # R0 = (R0 == R6) = (ones == 2)
```

```
  br   loop_xor_digits_p1_set_ones    # if ones == 2, ones = 0
  j    loop_xor_digits_p1             # loop

loop_xor_digits_p1_set_ones:
  seti 0                              # R0 = 0
  set  R6                             # R6 = ones = 0
  j    loop_xor_digits_p1             # loop

loop_overall_prepare:
  seti 2                              # R0 = 2 = 8’b00000010
  and  R4                             # R0 = R0&R4 = mem[index+1] & 8’b00000010 (parity_check)
  srl  1                              # R0 = (parity_check >> 1)
  xor  R6                             # R0 = parity_check = R0 ^ R6 = ones ^ parity_check
  add  R10                            # R0 = R0 + R10 = parity_affected + parity_check
  set  R10                            # R10 = parity_affected = R0

  seti 14                             # R0 = 14 = 8’b00001110
  sll  4                              # R0 = 8’b11100000
  and  R3                             # R0 = R0 & R3 = mem[index] & 8’b11100000
  srl  5                              # R0 = (mem[index] & 8’b11100000) >> 5
  set  R8                             # R8 = (mem[index] & 8’b11100000) >> 5 (mem_index)

  seti 14                             # R0 = 14 = 8’b00001110
  sll  4                              # R0 = 8’b11100000
  set  R9                             # R9 = R0 = 8’b11100000
  seti 8                              # R0 = 8 = 8’b00001000
  add  R9                             # R0 = R0 + R9 = 8’b11101000
  and  R4                             # R0 = R0 & R4 = 8’b11101000 & mem[index + 1]
  srl  3                              # R0 = (mem[index + 1] & 8’b11101000) >> 3
  set  R9                             # R9 = (mem[index + 1] & 8’b11101000) >> 3 (mem_index1)
  seti 1                              # R0 = 1
  and  R9                             # R0 = mem_index1 & 1
```

```
  set  R11                          # R11 = R0 (temp)
  la   R9                           # R0 = R9
  srl  1                            # R0 >>= 1
  add  R11                          # R0 = R0 + R11
  set  R9                           # R9 = mem_index1 -> 0 0 0 0 b4 b3 b2 b1
  seti 30                           # R0 = 30 = 8'b00011110
  and  R3                           # R0 = R0 & R3 = 8'b00011110 & mem[index]
  sll  3                            # R0 <<= 3
  add  R9                           # R0 = R0 + R9 -> b8 b7 b6 b5 b4 b3 b2 b1
  set  R9                           # R9 -> b8 b7 b6 b5 b4 b3 b2 b1

  seti 1                            # R0 = 1
  eq   R7                           # R0 = (R7 == R0) = (one_error_sign == 1)
  br   loop_overall_one_error       # one_error_sign == 1, go to loop_overall_one_error
  j    loop_overall_not_one_error   # 0 or 2 errors, go to loop_overall_not_one_error

loop_overall_one_error:
  seti 4                            # R0 = 4 = 8'b00000100
  sll  4                            # R0 = 8'b01000000
  add  R8                           # R0 = R0 + R8 = 8'b01000000 + mem_index
  set  R8                           # R8 = mem_index = R0

  seti 3                            # R0 = 3 = 8'b00000011
  eq   R10                          # R0 = (R0 == R10) = (parity_affected == 8'b00000011)
  br   loop_overall_one_error_b1    # R0 = 1, b1 error

  seti 5                            # R0 = 5 = 8'b00000101
  eq   R10                          # R0 = (R0 == R10) = (parity_affected == 8'b00000101)
  br   loop_overall_one_error_b2    # R0 = 1, b2 error

  seti 6                            # R0 = 6 = 8'b00000110
  eq   R10                          # R0 = (R0 == R10) = (parity_affected == 8'b00000110)
```

```
br   loop_overall_one_error_b3      # R0 = 1, b3 error

seti 7                              # R0 = 7 = 8’b00000111
eq   R10                            # R0 = (R0 == R10) = (parity_affected == 8’b00000111)
br   loop_overall_one_error_b4      # R0 = 1, b4 error

seti 9                              # R0 = 9 = 8’b00001001
eq   R10                            # R0 = (R0 == R10) = (parity_affected == 8’b00001001)
br   loop_overall_one_error_b5      # R0 = 1, b5 error

seti 10                             # R0 = 10 = 8’b00001010
eq   R10                            # R0 = (R0 == R10) = (parity_affected == 8’b00001010)
br   loop_overall_one_error_b6      # R0 = 1, b6 error

seti 11                             # R0 = 11 = 8’b00001011
eq   R10                            # R0 = (R0 == R10) = (parity_affected == 8’b00001011)
br   loop_overall_one_error_b7      # R0 = 1, b7 error

seti 12                             # R0 = 12 = 8’b00001100
eq   R10                            # R0 = (R0 == R10) = (parity_affected == 8’b00001100)
br   loop_overall_one_error_b8      # R0 = 1, b8 error

seti 13                             # R0 = 13 = 8’b00001101
eq   R10                            # R0 = (R0 == R10) = (parity_affected == 8’b00001101)
br   loop_overall_one_error_b9      # R0 = 1, b9 error

seti 14                             # R0 = 14 = 8’b00001110
eq   R10                            # R0 = (R0 == R10) = (parity_affected == 8’b00001110)
br   loop_overall_one_error_b10     # R0 = 1, b10 error

seti 15                             # R0 = 15 = 8’b00001111
eq   R10                            # R0 = (R0 == R10) = (parity_affected == 8’b00001111)
```

```
  br   loop_overall_one_error_b11     # R0 = 1, b11 error

loop_overall_one_error_b1:
  seti 1                              # R0 = 1 = 8’b00000001
  xor  R9                             # R0 = R0 ^ R9 = 8’b00000001 ^ mem_index1
  set  R9                             # R9 = mem_index1 = mem_index1 ^ 8’b00000001
  j    loop_overall_end

loop_overall_one_error_b2:
  seti 2                              # R0 = 2 = 8’b00000010
  xor  R9                             # R0 = R0 ^ R9 = 8’b00000010 ^ mem_index1
  set  R9                             # R9 = mem_index1 = mem_index1 ^ 8’b00000010
  j    loop_overall_end

loop_overall_one_error_b3:
  seti 4                              # R0 = 4 = 8’b00000100
  xor  R9                             # R0 = R0 ^ R9 = 8’b00000100 ^ mem_index1
  set  R9                             # R9 = mem_index1 = mem_index1 ^ 8’b00000100
  j    loop_overall_end

loop_overall_one_error_b4:
  seti 8                              # R0 = 8 = 8’b00001000
  xor  R9                             # R0 = R0 ^ R9 = 8’b00001000 ^ mem_index1
  set  R9                             # R9 = mem_index1 = mem_index1 ^ 8’b00001000
  j    loop_overall_end

loop_overall_one_error_b5:
  seti 16                             # R0 = 16 = 8’b00010000
  xor  R9                             # R0 = R0 ^ R9 = 8’b00010000 ^ mem_index1
  set  R9                             # R9 = mem_index1 = mem_index1 ^ 8’b00010000
  j    loop_overall_end
```

```
loop_overall_one_error_b6:
  seti 2                              # R0 = 2 = 8’b00000010
  sll  4                              # R0 = 8’b00100000
  xor  R9                             # R0 = R0 ^ R9 = 8’b00100000 ^ mem_index1
  set  R9                             # R9 = mem_index1 = mem_index1 ^ 8’b00100000
  j    loop_overall_end

loop_overall_one_error_b7:
  seti 4                              # R0 = 2 = 8’b00000100
  sll  4                              # R0 = 8’b01000000
  xor  R9                             # R0 = R0 ^ R9 = 8’b01000000 ^ mem_index1
  set  R9                             # R9 = mem_index1 = mem_index1 ^ 8’b01000000
  j    loop_overall_end

loop_overall_one_error_b8:
  seti 8                              # R0 = 8 = 8’b00001000
  sll  4                              # R0 = 8’b10000000
  xor  R9                             # R0 = R0 ^ R9 = 8’b10000000 ^ mem_index1
  set  R9                             # R9 = mem_index1 = mem_index1 ^ 8’b10000000
  j    loop_overall_end

loop_overall_one_error_b9:
  seti 1                              # R0 = 1 = 8’b00000001
  xor  R8                             # R0 = R0 ^ R8 = 8’b00000001 ^ mem_index
  set  R8                             # R8 = mem_index = mem_index ^ 8’b00000001
  j    loop_overall_end

loop_overall_one_error_b10:
  seti 2                              # R0 = 2 = 8’b00000010
  xor  R8                             # R0 = R0 ^ R8 = 8’b00000010 ^ mem_index
  set  R8                             # R8 = mem_index = mem_index ^ 8’b00000010
  j    loop_overall_end
```

```
loop_overall_one_error_b11:
  seti 4                            # R0 = 4 = 8’b00000100
  xor  R8                           # R0 = R0 ^ R8 = 8’b00000100 ^ mem_index
  set  R8                           # R8 = mem_index = mem_index ^ 8’b00000100
  j    loop_overall_end

loop_overall_not_one_error:
  eq   R10                          # R0 = (R0 == R10) = (parity_affected == 0)
  br   loop_overall_end             # if parity_affected == 0, no error
  seti 8                            # R0 = 8 = 8’b00001000
  sll  4                            # R0 = 8’b10000000
  add  R8                           # R0 = R0 + R8 = mem_index + 8’b10000000
  set  R8                           # R8 = mem_index = mem_index + 8’b10000000

loop_overall_end:
  seti 30                           # R0 = 30
  set  R11                          # R11 = 30
  la   R1                           # R0 = R1 = index
  sub  R11                          # R0 = R0 - R11 = index - 30
  set  R11                          # R11 = R0 = index - 30
  la   R8                           # R0 = R8 = mem_index
  sw   R11                          # mem[R11] = R0, mem[index - 30] = mem_index

  seti 29                           # R0 = 29
  set  R11                          # R11 = 29
  la   R1                           # R0 = index
  sub  R11                          # R0 = R0 - R11 = index - 29
  set  R11                          # R11 = R0 = index - 29
  la   R9                           # R0 = R9 = mem_index1
  sw   R11                          # mem[R11] = R0, mem[index - 29] = mem_index1
```

```
  seti 2                                 # R0 = 2
  add  R1                                # R0 = R0 + R1 = index + 2
  set  R1                                # R1 = index = index + 2

  j    loop_overall_start                # loop

loop_done:
  halt                                   # terminate the program
```

# Program 3 Pseudocode

```
patternSearch(msg, pattern):
  pattern = mem[32] >> 3   # right shift by 3 so that our current bit index is 0
  mem_33 = 0
  mem_34 = 0
  mem_35 = 0
  for i in range (0, 32):
    occNum = 0
    currByte = mem[i]
    for j in range(0, 4):
      curr_5bit = currByte & 0b00011111
      xor_output = pattern ^ curr_5bit
      if xor_output == 0:
        occNum += 1
        mem_33 += 1
      currByte >>= 1
    mem_34 += (occNum > 0)
  for i in range(0, 32):
    mem_i = mem[i]
    mem_i1 = mem[i + 1]
    for j in range(7, -1, -1): # [7, 6, 5, 4, 3, 2, 1, 0]
      if j > 3: # in byte
        curr_5bit = 0b00011111 << (j - 4)
        curr_5bit = curr_5bit & mem_i
        curr_5bit >>= (j - 4)
      else: # cross bytes
        if i == 31:
          break
        # Comment for the following lines: assume j = 1, so we need mem[i][1:0]:mem[i+1][7:5]
        curr_5bit_1 = 0b00001111 >> (3 - j) # bit[1:0]
        curr_5bit_1 = curr_5bit_1 & mem_i # mem[i][1:0]
```

```
        curr_5bit_1 <<= (4 - j) # mem[i][1:0]:0b000
        curr_5bit_2 = 0b00001111 >> j # bit[2:0]
        curr_5bit_2 <<= (j + 4) # bit[7:5]
        curr_5bit_2 = curr_5bit_2 & mem_i1 # mem[i+1][7:5]:0b00000
        curr_5bit_2 >>= (j + 4) # mem[i+1][7:5]
        curr_5bit = curr_5bit1 + curr_5bit2 # mem[i][1:0]:mem[i+1][7:5]
      xor_output = pattern ^ curr_5bit
      if xor_output == 0:
        mem_35 += 1
  mem[33] = mem_33
  mem[34] = mem_34
  mem[35] = mem_35
```

## Program 3 Assembly Code

```
# R1: loop counter i
# R2: pattern
# R3 : mem_33
# R4 : mem_34
# R5 : mem_35
# R6: loop boundary 32
# R7: store mem[i]
# R8: store mem[i+1]
# R9: occNum
# R10: loop counter j
# R11: loop boundary for j
# R12: curr_5bit1
# R13: curr_5bit2, curr_5bit
# R14: shifting number
# R15: bit mask 0b0000_0001

seti 0b10000    # R0 = 0b10000
```

```
sll  1     # R0 = 0b100000 = 32
set  R6    # R6 = 32 (loop boundary)
lw   R0    # R0 = mem[32]
srl  3     # R0 = mem[32] >> 3
set  R2    # R2 = mem[32] >> 3 (pattern)

seti 0     # R0 = 0
set  R1    # R1 = 0 (loop counter i)

set  R3    # R3 = mem_33
set  R4    # R4 = mem_34
set  R5    # R5 = mem_35

seti 1     # R0 = 1
set  R15   # R15 = 1 (bit mask)

LOOP1a:
     la   R1   # R0 = loop counter i
     eq   R6   # R0 = (R0 == R6) = (i == 32)
     br   after_LOOP1     # if i == 32, leave loop

     seti 0    # R0 = 0
     set  R9   # R9 = 0 (occNum)
     lw   R1   # R0 = mem[i]
     set  R7   # R7 = mem[i] (currByte)
     seti 0    # R0 = 0
     set  R10  # R10 = 0 (loop counter j)
     seti 4    # R0 = 4
     set  R11  # R11 = 4 (loop boundary j)


LOOP1j_a:
```

```
    la    R10   # R0 = loop counter j
    eq    R11   # R0 = (R0 == R11) = (j == 4)
    br    LOOP1b     # if j == 4, leave loop

    seti  0b11111    # R0 = 0b11111
    and   R7    # R0 = R0 & R7 = 0b11111 & currByte = curr_5bit
    xor   R2    # R0 = R0 ^ R2 = curr_5bit ^ pattern = xor_output

    set   R15   # R15 = xor_output [BORROW]
    seti  0     # R0 = 0
    eq    R15   # R0 = (0 == xor_output)
    br    LOOP1_addOccNum # match pattern
    j     LOOP1j_b

LOOP1_addOccNum:
    seti  1     # R0 = 1
    add   R9    # R0 = 1 + R9
    set   R9    # R9 += 1

    seti  1     # R0 = 1
    add   R3    # R0 = 1 + R3
    set   R3    # R3 += 1

LOOP1j_b:
    seti  1     # R0 = 1
    set   R15   # R15 = 1 (bit mask) [RETURN]

    la    R7    # R0 = currByte
    srl   1     # R0 = currByte >> 1
    set   R7    # currByte >>= 1

    set   0     # R0 = 0
```

```
        slt   R9    # R0 = (R0 < R9) = (occNum > 0)
        add   R4    # R0 = R4 + (occNum > 0)
        set   R4    # R4 += (occNum > 0)

        la    R15   # R0 = 1
        add   R10   # R0 = R10 + 1
        set   R10   # R10 = R10 + 1
        j     LOOP1j_a   # loop

LOOP1b:
        la    R15   # R0 = 1
        add   R1    # R0 = R1 + 1
        set   R1    # R1 = R1 + 1
        j     LOOP1a     # loop

after_LOOP1:
        seti  0     # R0 = 0
        set   R1    # loop counter i = 0

LOOP2a:
        la    R1    # R0 = loop counter i
        eq    R6    # R0 = (R0 == R6) = (i == 32)
        br    after_LOOP2     # if i == 32, leave loop

        lw    R1    # R0 = mem[i]
        set   R7    # R7 = mem[i] (mem_i)

        set   1     # R0 = 1
        add   R1    # R0 = i + 1
        lw    R0    # R0 = mem[i+1]
        set   R8    # R8 = mem[i+1] (mem_i1)
```

```
      seti 0     # R0 = 0
      set  R10   # R10 = 0 (loop counter j)
      set  R11   # R11 = 0 (loop boundary j)

LOOP2j_a:
      seti 3     # R0 = 3
      slt  R10   # R0 = R0 < R10 = 3 < j = j > 3
      br   LOOP2j_inByte   # if j > 3, in byte

      seti 31    # R0 = 31
      eq   R1    # R0 = (i == 31)
      br   LOOP2b     # no more

      seti 0b1111     # R0 = 0b1111
      set  R12   # R12 = curr_5bit1 = 0b1111
      seti 3     # R0 = 3
      sub  R10   # R0 = 3 - j
      set  R14   # R14 = shifting number = 3 - j
      la   R12   # R0 = 0b1111
      srl  R14   # R0 = 0b1111 >> 3 - j
      and  R7    # R0 = R0 & mem_i
      set  R12   # R12 = curr_5bit1 = (0b1111 >> 3 - j) & mem_i
      seti 1     # R0 = 1
      add  R14   # R14 = 4 - j
      la   R12   # R0 = curr_5bit1
      sll  R14   # R0 = curr_5bit1 << 4 - j
      set  R12   # R12 = curr_5bit1

      seti 0b1111     # R0 = 0b1111
      set  R13   # R13 = curr_5bit2 = 0b1111
      la   R13   # R0 = 0b1111
      srl  R10   # R0 = 0b1111 >> j
```

```
        set   R13   # R13 = 0b1111 >> j
        seti  4     # R0 = 4
        add   R10   # R0 = j + 4
        set   R14   # R14 = j + 4
        la    R13   # R0 = curr_5bit2
        sll   R14   # R0 = curr_5bit2 << j + 4
        and   R8    # R0 = curr_5bit2 << j + 4 & mem_i1
        srl   R14   # R0 = ((curr_5bit2 << j + 4) & mem_i1) >> j + 4
        set   R13   # R13 = curr_5bit2

        la    R12   # R0 = curr_5bit1
        add   R13   # R0 = curr_5bit1 + curr_5bit2
        set   R13   # R13 = curr_5bit = curr_5bit1 + curr_5bit2

        j     LOOP2j_b

LOOP2j_inByte:
        seti  0b11111    # R0 = 0b11111
        set   R13   # R13 = 0b11111
        seti  4     # R0 = 4
        set   R14   # R14 = 4
        la    R10   # R0 = j
        sub   R14   # R0 = j - 4
        set   R14   # R14 = shifting number = j - 4
        la    R13   # R0 = R13
        sll   R14   # R0 = R0 << R15 = 0b11111 << (j - 4)
        and   R7    # R0 = R0 & R7 = curr_5bit & mem_i = curr_5bit
        srl   R14   # R0 = curr_5bit >> (j - 4)
        set   R13   # R13 = curr_5bit

LOOP2j_b:
        seti  0     # R0 = 0
```

```
        set   R15   # R15 = 0 [BORROW]
        la    R2    # R0 = R2 = pattern
        xor   R13   # R0 = pattern ^ curr_5bit
        eq    R15   # if xor_output == 0
        br    LOOP2_addMem35
        j     LOOP2j_c

LOOP2_addMem35:
        seti  1     # R0 = 1
        add  R5     # R0 = 1 + R5
        set  R5     # R5 += 1

LOOP2j_c:
        la    R10   # R0 = j
        eq    R15   # whether j == 0
        br    LOOP2b     # go out of the j loop
        seti  1     # R0 = 1
        set   R15   # R15 = 1 [RETURN]
        la    R10   # R0 = j
        sub   R15   # R0 = j - 1
        set   R10   # j = j - 1
        j     LOOP2j_a

LOOP2b:
        la    R15   # R0 = 1
        add   R1    # R0 = R1 + 1
        set   R1    # R1 = R1 + 1
        j     LOOP2a     # loop

after_LOOP2:
        seti 2      # R0 = 2 = 8'b00000010
        sll 4       # R0 = 8'b00100000
```

```
set R15    # R15 = 8'b00100000
seti 1     # R0 = 1 = 8'b00000001
add R15    # R0 += R15 = 8'b00100001 = 33
set R15    # R15 = 33
la R3      # la mem_33 to R0
sw R15     # mem[R15] = R0, mem[33] = mem_33

seti 1     # R0 = 1
add R15    # R0 = R0 + R15 = 1 + 33 = 34
set R15    # R15 = 34
la R4      # la mem_34 to R0
sw R15     # mem[R15] = R0, mem[34] = mem_34

seti 1     # R0 = 1
add R15    # R0 = R0 + R15 = 1 + 34 = 35
set R15    # R15 = 35
la R5      # la mem_35 to R0
sw R15     # mem[R15] = R0, mem[35] = mem_35

halt # terminate
```