

CSE 141L Milestone 1

Aung Kyaw, A17178766; Myo Zaw Win, A16241709; Ojeen Gammah A17169134

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.

Aung Kyaw
Myo Zaw Win
Ojeen Gammah

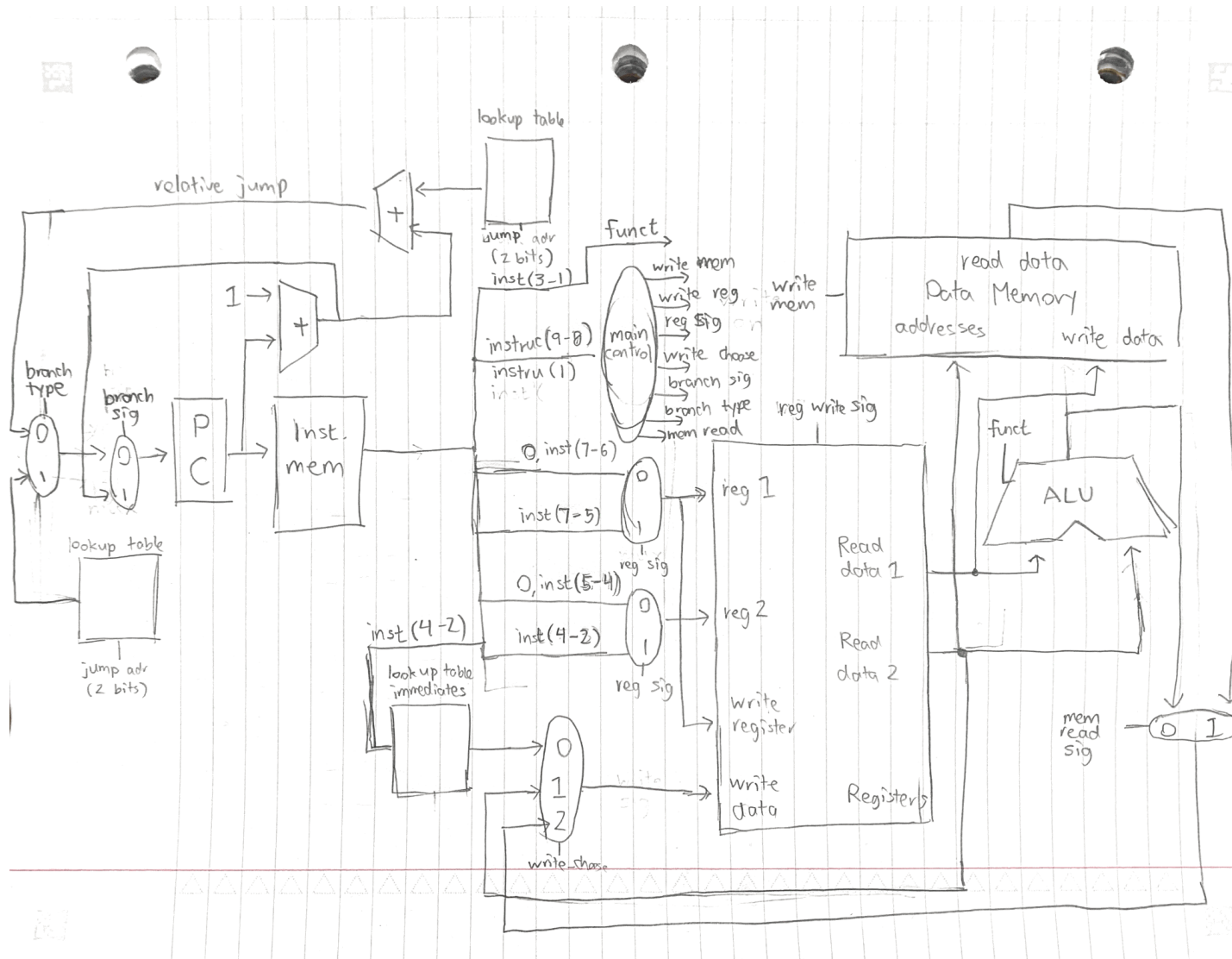
0. Team

Aung Kyaw
Myo Zaw Win
Ojeen Gammah

1. Introduction

The name of our architecture is Basic V1. The philosophy of our architecture is a load store model that provides a simple set of instructions that although are specialized to solve the 3 program given can also be applied to any general program. Our main goal was to keep the hardware architecture simple and general purpose while also providing all the tools needed for a processor. For example, most the ALU instructions we use are a subset of those that are found in a MIPS ALU. Our processor prioritizes ease of use and programming capabilities over speed, and we rely on memory operations to store information because there are only 8 registers, of which 4 can be used for instructions. Similar to modern machine, we reserve a register to act as a stack pointer which allows us extra memory over the 8 registers.

2. Architectural Overview



3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	2 bit opcode, 3 bit ALU instruction, 2 bit operand 1, 2 bit operand 2	inc, dec, add, shftl, shftr, and, or, xor, par
J	2 bit opcode, 2 bit operand 1, 2 bit operand 2, 3 bit branch target	beq
I	2 bit opcode, 3 bit operand 1, 3 bit operand 2, 1 bit control	load, store, movl, movr

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
Inc = Increment	R	2 bits opcode (00), 3 bit register (XXX), (0) 1 bit control, 3 bits funct(000)	# Assume R0 = 0b0000_0000 inc R0 = 00_000_0_000 # After: R0 = 0b0000_0001	Will increment the specified register, unlike other R types, increment takes a 3 bit register Shares the same funct with dec
Dec = Decrement	R	2 bits opcode (00), 3 bit register (XXX), (1) 1 bit control, 3 bits funct(000)	# Assume R0 = 0b0000_0010 dec R0 = 00_000_1_000	Will decrement the specified register, unlike other R types, increment takes a 3 bit register

			# After: R0 = 0b0000_0001	Shares the same funct with inc
shftl = Shift Left	R	2 bits opcode (00), 2 bits operand register(XX), 2 bits operand register (XX), 3 bits funct(001)	# Assume R0 = 0b0000_0010 # Assume R1 = 0b0000_0011 shftl R0 R1 = 00_00_01_001 # After: R0 = 0b0001_0000	Shift will use a shift in value of 0
shftr = Shift Right	R	2 bits opcode (00), 2 bits operand register(XX), 2 bits operand register (XX), 3 bits funct(010)	# Assume R0 = 0b0000_0010 # Assume R1 = 0b0000_0001 shftr R0 R1 = 00_00_01_010 # After: R0 = 0b0000_0001	Shift will use a shift in value of 0
And = Bitwise And	R	2 bits opcode (00), 2 bits operand register(XX), 2 bits operand register (XX), 3 bits funct(011)	# Assume R0 = 0b0110_0010 # Assume R1 = 0b1011_1100 and R0 R1 = 00_00_01_011 # After: R0 = 0b0010_0000 # R1 = 0b1011_1100	The first operand register is also the destination register and r0 r1 -> r0 = r0 and r1
Or = Bitwise Or	R	2 bits opcode (00), 2 bits operand register(XX), 2 bits operand register (XX), 3 bits funct(100)	# Assume R0 = 0b0110_0010 # Assume R1 = 0b1011_0000 or R0 R1 = 00_00_01_100 # After: R0 = 0b1111_0010 # R1 = 0b1011_0000	The first operand register is also the destination register or r0 r1 -> r0 = r0 or r1
XOR = Bitwise XOR	R	2 bits opcode (00), 2 bits operand register(XX), 2 bits operand register (XX),	# Assume R0 = 0b0110_0010 # Assume R1 = 0b1011_0000	The first operand register is also the destination register xor r0 r1 -> r0 = r0 xor r1

		3 bits funct(101)	xor R0 R1 = 00_00_01_101 # After: R0 = 0b1101_0010 # R1 = 0b1011_0000	
PAR = Parity Bit	R	2 bits opcode (00), 2 bits destination register(XX), 2 bits operand register (XX), 3 bits funct(111)	# Assume R0 = 0b0110_0010 # Assume R1 = 0b1011_0000 par R0 R1 = 00_00_01_111 # After: R0 = 0b0000_0001 # R1 = 0b1011_0000	Stores into the destination register the parity of the operand register in the format 0b0000_000p where p is the XOR of all the bits in the operand
bne = branch not equal	J	2 bits opcode (11), 2 bits operand register(XX), 2 bits operand register (XX), 2 bit branch target (XXX), 1 bit control (X)	# Assume R0 = 0b0110_0010 # Assume R1 = 0b0110_0010 beq R0 R1 endloop = 11_00_01_00_0 # After: R0 = 0b0110_0010 # R1 = 0b0110_0010	If R0 and R1 are not equal, then the PC will be updated to be at the location that is given by a lookup table
load	I	2 bits opcode(01), 3 bits operand register (XXX), 3 bits operand address register (XXX), (0)1 bit control	# Assume mem[0] = 0b0110_1000 # Assume r1 = 0b0000_0000 load r0 r1 = 01_000_001_0 # After : r0 = 0b0110_1000	Load and Store have the same opcode but are instead controlled by a control bit.
store	I	2 bits opcode(01), 3 bits operand register (XXX), 3 bits operand address register (XXX), (1) 1 bit control	# Assume r0 = 0b0110_1000 # Assume r1 = 0b0000_0000 store r0 r1 = 01_000_001_1 # After :mem[0] = 0b0110_1000	
movl =	I	2 bits opcode(11), 3 bit	movl r0 3 = 11_000_011_0	The 3 bits will be used on a lookup tablet

move immediate		destination register (XXX), 3 bit immediate lookup (XXX), (0) 1 bit control	#After : r0 = 0b0000_0011	that contains the corresponding immediate values
movr = move registers	1	2 bits opcode(11), 3 bit destination register (XXX), 3 bit source register (XXX), (1) 1 bit control	# Assume r1 = 0b0011_1111 movr r0 r1 = 11_000_001_1 # After: r0 = 0b0011_1111	

Internal Operands

There will be 8 registers. The first 4 registers are general purpose registers, r0-r3 and the only registers that can be accessed most ALU operations except for the increment which can access all registers. All 8 registers can have values loaded from memory or stored to memory and their values can be overwritten by either another register or a set of predefined immediate values. r7 will be a special register that will be used as a pseudo stack-counter and it will be used to point to the place in data memory where the output has to be written. r4 will primarily be used as an accumulator for for loops. r5-r6 are registers that will primarily be used for the purpose of branching or for accumulators but they can also be used for temporary storage.

Control Flow (branches)

The structure will support both relative and absolute jumps. For each jump type, there will be four addresses or jump sizes that are supported. The idea behind this is that for loops, we will use absolute jumps because it will be large jumps. To get the target address, we will use a 2 bit lookup table to search for addresses which matches with the four while loops that we have in our program. As for relative jumps, we will use it for if statements. Similar to absolute jumps, we will use a lookup table to get the relative jump distance.

Addressing Modes

For addressing we will use indirect addressing where the address of memory is stored inside a register. Indirect addressing is the only mode that is available so to get to a certain address, we have to modify the contents of the register. For example, to get the contents of mem[20], we would do movi r1 20, then load r0 r1.

4. Programmer's Model [Lite]

A programmer can think of the machine as a simple MIPS with less instructions and less registers. We are limited to 4 registers for operations and all operations can only be done on registers so registers have to be initialized so one register can be used as a temp register. The programmer has to make use of r7, the stack pointer register, and updates its value as values are stored and loaded from memory. This design allows the programmer to exceed the capabilities of having only 8 registers. The programmer also has free reign with incrementing all registers and moving values around all registers so the programmer can choose their own registers to use for various operations. The instructions that are available are simple and basic in nature so to perform more complicated tasks, it requires multiple instructions.

Since our architecture is very similar to MIPS, most basic MIPS instructions can be copied over with some adjustments. The main adjustment is that our instructions only take two registers where one register is both the operand and the destination register. However, all the instructions in our architecture have a similar instruction in MIPS.

5. Program Implementation

Program 1 Pseudocode

i = 0

while i < 30:

 parities = 0000 0000

 topHalf = mem[i+1];

 botHalf = mem[i];

 botHalf = botHalf AND 1111 0000

 topHalf = topHalf XOR botHalf

 topHalf = parity(topHalf);

 parities = parities | topHalf;

 parities = parities << 1;

// topHalf = 0000 0b₁₁b₁₀b₉

// botHalf = b₈b₇b₆b₅ b₄b₃b₂b₁

// botHalf = b₈b₇b₆b₅ 0000

// topHalf = b₈b₇b₆b₅ 0b₁₁b₁₀b₉

// topHalf = 0000 000p₈

// parities = 0000 000p₈


```

topHalf = mem[i+1];
botHalf = mem[i];
botHalf = botHalf AND 1000 1110;
topHalf = topHalf XOR botHalf;
topHalf = parity(topHalf);
topHalf = topHalf << 4;
parities = parities | topHalf;

```

```

// topHalf = 0000 0b11b10b9
// botHalf = b8000 b4b3b20
// topHalf = b8000 b4(b11⊕b3)(b10⊕b2)b9
// topHalf = 0000 000p4
// parities = 000p4 000p8

```

```

topHalf = mem[i+1];
botHalf = mem[i];
botHalf = botHalf AND 0110 1101;
topHalf = topHalf AND 0000 0110;
topHalf = topHalf XOR botHalf;
topHalf = parity(topHalf);
topHalf = topHalf << 2;
parities = parities | topHalf;

```

```

// botHalf = 0b7b60 b4b30b1
// topHalf = 0000 0b11b100
// topHalf = 0b7b60 b4(b3⊕b11)b10b1
// topHalf = 0000 000p2
// parities = 000p4 0p20p8

```

```

topHalf = mem[i+1];
botHalf = mem[i];
botHalf = botHalf AND 0101 1011;
topHalf = topHalf AND 0000 0101;
topHalf = topHalf XOR botHalf;
topHalf = parity(topHalf);
topHalf = topHalf << 1;
parities = parities | topHalf;

```

```

// botHalf = 0b70b5 b40b2b1
// topHalf = 0000 0b110b9
// topHalf = 0b70b50 b4b11b2(b1⊕b9)
// topHalf = 0000 000p1
// parities = 000p4 0p2p1p8

```

```

topHalf = mem[i+1];
botHalf = mem[i];
topHalf = topHalf << 5;
topHalf = topHalf OR parities;
topHalf = topHalf AND 1110 0001;
botHalf = botHalf >> 4 << 1;

```

```

// topHalf = 0000 0b11b10b9
// botHalf = b8b7b6b5 b4b3b2b1
// topHalf = b11b10b90 0000
// topHalf = b11b10b9p4 0p2p1p8
// topHalf = b11b10b90 000p8
// botHalf = 000b8 b7b6b50

```

```

topHalf = topHalf OR botHalf;
mem[30 + i + 1] = topHalf;
parities = parities AND 1111 1110;

```

```

botHalf = mem[i];
botHalf = botHalf << 4;
temp = botHalf;
temp = temp << 3;
temp = temp >> 4;
botHalf = botHalf AND 1110 0000
botHalf = botHalf OR temp
botHalf = botHalf OR parities

```

```

topHalf = mem[30 + i + 1];
topHalf = parity(topHalf);
parities = botHalf;
parities = parity(botHalf);
parities = parities XOR topHalf
botHalf = botHalf XOR parities

```

```

mem[30 + i] = botHalf;
i++;
i++;

```

```
// topHalf =  $b_{11}b_{10}b_9b_8 \ b_7b_6b_5p_8$ 
```

```
// parities =  $000p_4 \ 0p_2p_10$ 
```

```
// botHalf =  $b_8b_7b_6b_5 \ b_4b_3b_2b_1$ 
```

```
// botHalf =  $b_4b_3b_2b_1 \ 0000$ 
```

```
// temp =  $b_1000 \ 0000$ 
```

```
// temp =  $0000 \ b_1000$ 
```

```
// botHalf =  $b_4b_3b_20 \ 0000$ 
```

```
// botHalf =  $b_4b_3b_20 \ b_1000$ 
```

```
// botHalf =  $b_4b_3b_2p_4 \ b_1p_2p_10$ 
```

```
// topHalf =  $b_{11}b_{10}b_9b_8 \ b_7b_6b_5p_8$ 
```

```
// topHalf =  $0000 \ 000( \wedge(b_{11}:b_5, p_8) )$ 
```

```
// parities =  $0000 \ 000( \wedge(b_4:b_1, p_4, p_2, p_1) )$ 
```

```
// parities =  $0000 \ 000p_0$ 
```

```
// botHalf =  $b_4b_3b_2p_4 \ b_1p_2p_1p_0$ 
```

Reference:

mem[1] = 00000101 -- 5 $b_{11}:b_9$

mem[0] = 01010101 -- $b_8:b_1$

```
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;
```

Program 1 Assembly Code

i = 0

Movi r4 0 // r4 is i

movi r7 31

while i < 30:

startloop:

 parities = 0000 0000

Movi r0 0 // r0 is parities

 topHalf = mem[i+1]; // topHalf = 0000 0b₁₁b₁₀b₉

Movi r3 r4

Inc r3

Load r1 r3 //topHalf = r1

 botHalf = mem[i];

Load r2 r4 // botHalf = b₈b₇b₆b₅ b₄b₃b₂b₁

 botHalf = botHalf AND 1111 0000

 // botHalf = b₈b₇b₆b₅ 0000

And r2 240

 topHalf = topHalf XOR botHalf

 // topHalf = b₈b₇b₆b₅ 0b₁₁b₁₀b₉

Xor r1 r2

 topHalf = parity(topHalf);

 // topHalf = 0000 000p₈

Par r1 r1

 parities = parities | topHalf;

 // parities = 0000 000p₈

Or r0 r3

 parities = parities << 1;

Shftl r0 r0

 topHalf = mem[i+1];

 // topHalf = 0000 0b₁₁b₁₀b₉

Load r1 r3

 botHalf = mem[i];

Load r2 r4

 botHalf = botHalf AND 1000 1110;

 // botHalf = b₈000 b₄b₃b₂0

And r2 142

 topHalf = topHalf XOR botHalf;

 // topHalf = b₈000 b₄(b₁₁⊕b₃)(b₁₀⊕b₂)b₉

```

Xor r1 r2
    topHalf = parity(topHalf);                // topHalf = 0000 000p4
Par r1 r1
    topHalf = topHalf << 4;
Shftl r1 4
    parities = parities | topHalf;            // parities = 000p4 000p8
Or r0 r1
    topHalf = mem[i+1];
Load r1 r3
    botHalf = mem[i];
Load r2 r4
    botHalf = botHalf AND 0110 1101;          // botHalf = 0b7b60 b4b30b1
And r2 109
    topHalf = topHalf AND 0000 0110;          // topHalf = 0000 0b11b100
And r1 6
    topHalf = topHalf XOR botHalf;            // topHalf = 0b7b60 b4(b3⊕b11)b10b1
Xor r1 r2
    topHalf = parity(topHalf);                // topHalf = 0000 000p2
Par r1 r1
    topHalf = topHalf << 2;
Shftl r1 2
    parities = parities | topHalf;            // parities = 000p4 0p20p8
Or r0 r1
    topHalf = mem[i+1];
Load r1 r3
    botHalf = mem[i];
Load r2 r4
    botHalf = botHalf AND 0101 1011;          // botHalf = 0b70b5 b40b2b1
And r2 91
    topHalf = topHalf AND 0000 0101;          // topHalf = 0000 0b110b9
And r1 9

```

topHalf = topHalf XOR botHalf;	// topHalf = 0b ₇ 0b ₅ 0 b ₄ b ₁₁ b ₂ (b ₁ ⊕b ₉)
Xor r1 r2	
topHalf = parity(topHalf);	// topHalf = 0000 000p ₁
Par r1 r1	
topHalf = topHalf << 1;	
Shftl r1 1	
parities = parities topHalf;	// parities = 000p ₄ 0p ₂ p ₁ p ₈
Or r0 r1	
topHalf = mem[i+1];	// topHalf = 0000 0b ₁₁ b ₁₀ b ₉
Load r1 r3	
botHalf = mem[i];	// botHalf = b ₈ b ₇ b ₆ b ₅ b ₄ b ₃ b ₂ b ₁
Load r2 r4	
topHalf = topHalf << 5;	// topHalf = b ₁₁ b ₁₀ b ₉ 0 0000
Shftl r1 5	
topHalf = topHalf OR parities;	// topHalf = b ₁₁ b ₁₀ b ₉ p ₄ 0p ₂ p ₁ p ₈
Or r1 r0	
topHalf = topHalf AND 1110 0001;	// topHalf = b ₁₁ b ₁₀ b ₉ 0 000p ₈
And r1 225	
botHalf = botHalf >> 4 << 1;	// botHalf = 000b ₈ b ₇ b ₆ b ₅ 0
Shftl r2 1	
Shftr r2 4	
topHalf = topHalf OR botHalf;	// topHalf = b ₁₁ b ₁₀ b ₉ b ₈ b ₇ b ₆ b ₅ p ₈
Or r1 r2	
mem[30 + i + 1] = topHalf;	
store r1 r7	
parities = parities AND 1111 1110;	// parities = 000p ₄ 0p ₂ p ₁ 0
And r0 254	
botHalf = mem[i];	// botHalf = b ₈ b ₇ b ₆ b ₅ b ₄ b ₃ b ₂ b ₁
Load r2 r0	
botHalf = botHalf << 4;	// botHalf = b ₄ b ₃ b ₂ b ₁ 0000
Shftl r2 4	

temp = botHalf;	
Load r5 r2 //r5 is temp	
temp = temp << 3;	// temp = b ₁ 000 0000
Shftl r5 3	
temp = temp >> 4;	// temp = 0000 b ₁ 000
Shftr r5 4	
botHalf = botHalf AND 1110 0000	// botHalf = b ₄ b ₃ b ₂ 0 0000
And r2 224	
botHalf = botHalf OR temp	// botHalf = b ₄ b ₃ b ₂ 0 b ₁ 000
Or r2 r5	
botHalf = botHalf OR parities	// botHalf = b ₄ b ₃ b ₂ p ₄ b ₁ p ₂ p ₁ 0
Or r2 r0	
topHalf = mem[30 + i + 1];	// topHalf = b ₁₁ b ₁₀ b ₉ b ₈ b ₇ b ₆ b ₅ p ₈
load r1 r7	
topHalf = parity(topHalf);	// topHalf = 0000 000(\wedge (b ₁₁ :b ₅ , p ₈))
Par r1 r1	
parities = botHalf;	
Movi r0 r2	
parities = parity(botHalf);	// parities = 0000 000(\wedge (b ₄ :b ₁ , p ₄ , p ₂ , p ₁))
Par r0 r2	
parities = parities XOR topHalf	// parities = 0000 000p ₀
Xor r0 r1	
botHalf = botHalf XOR parities	// botHalf = b ₄ b ₃ b ₂ p ₄ b ₁ p ₂ p ₁ p ₀
Xor r2 r0	
mem[30 + i] = botHalf;	
dec r7;	
store r2 r7	
i++;	
Inc r4	
inc r7	

i++;

Inc r4

inc r7

Bne r4 30 startloop

Program 2 Pseudocode

i = 30

while i < 60:

// calculating discrepancy vector

parities = 0000 0000

topHalf = mem[i+1];

botHalf = mem[i];

topHalf = parity(topHalf);

parities = parities | topHalf;

parities = parities << 1;

topHalf = mem[i+1];

botHalf = mem[i];

botHalf = botHalf AND 1111 0000

topHalf = topHalf AND 1111 0000

topHalf = topHalf XOR botHalf

topHalf = parity(topHalf);

parities = parities | topHalf;

parities = parities << 1;

topHalf = mem[i+1];

botHalf = mem[i];

botHalf = botHalf AND 1100 1100

topHalf = topHalf AND 1100 1100

topHalf = topHalf XOR botHalf

topHalf = parity(topHalf);

parities = parities | topHalf;

parities = parities << 1;

topHalf = mem[i+1];

botHalf = mem[i];

// topHalf = $b_{11}b_{10}b_9b_8 b_7b_6b_5p_8$

// botHalf = $b_4b_3b_2p_4 b_1p_2p_1p_0$

// topHalf = 0000 000s₈

// parities = 0000 000s₈

// parities = 0000 00s₈0

// topHalf = $b_{11}b_{10}b_9b_8 b_7b_6b_5p_8$

// botHalf = $b_4b_3b_2p_4 b_1p_2p_1p_0$

// topHalf = $b_{11}b_{10}b_9b_8$ 0000

// botHalf = $b_4b_3b_2p_4$ 0000

// topHalf = 0000 000s₄

// parities = 0000 00s₈s₄

// parities = 0000 0s₈s₄0

// topHalf = $b_{11}b_{10}b_9b_8 b_7b_6b_5p_8$

// botHalf = $b_4b_3b_2p_4 b_1p_2p_1p_0$

// topHalf = $b_{11}b_{10}$ 00 b_7b_6 00

// botHalf = b_4b_3 00 b_1p_2 00

// topHalf = 0000 000s₂

// parities = 0000 0s₈s₄s₂

// parities = 0000 s₈s₄s₂0

// topHalf = $b_{11}b_{10}b_9b_8 b_7b_6b_5p_8$

// botHalf = $b_4b_3b_2p_4 b_1p_2p_1p_0$

```

botHalf = botHalf AND 1010 1010
topHalf = topHalf AND 1010 1010
topHalf = topHalf XOR botHalf
topHalf = parity(topHalf);
parities = parities | topHalf;
parities = parities << 1;

```

```

// topHalf =  $b_{11}b_9b_7b_5b_3b_1$ 
// botHalf =  $b_4b_2b_0b_{-1}b_{-3}b_{-5}$ 

// topHalf = 0000 000s1
// parities = 0000 s8s4s2s1
// parities = 000s8 s4s2s10

```

```

topHalf = mem[i+1];
botHalf = mem[i];
topHalf = topHalf XOR botHalf
topHalf = parity(topHalf);
parities = parities | topHalf;

```

```

// topHalf =  $b_{11}b_{10}b_9b_8b_7b_6b_5b_4$ 
// botHalf =  $b_3b_2b_1b_0b_{-1}b_{-2}b_{-3}b_{-4}$ 

// topHalf = 0000 000s0
// parities = 000s8 s4s2s1s0

```

```

topHalf = mem[i];
botHalf = mem[i+1];

```

```

// topHalf =  $b_{11}b_{10}b_9b_8b_7b_6b_5b_4$ 
// botHalf =  $b_3b_2b_1b_0b_{-1}b_{-2}b_{-3}b_{-4}$ 

```

```

errorBit = parities >> 1;
temp = 0000 0001;
temp = temp << (errorBit);
if (temp == 0):
    errorBit = errorBit AND 0000 1000
    temp = 0000 0001;
    temp = temp << (errorBit);
    topHalf = topHalf XOR temp;
else:
    botHalf = botHalf XOR temp;
errorCount = 0000 0000;
if (errorBits != 0):
    errorCount ++;
parities = parities AND 0000 0001;
if (parities != 0):
    errorCount ++;

```

```

// errorBit = 0000 s8s4s2s1 = errorLocation

// temp will be a one hot encoding of the value of errorLocation
// errorLocation > 8
// temp = temp - 8

// temp will be a one hot encoding of the value of error
// flip topHalf's bit by errorLocation value - 8

// flip botHalf's bit by errorLocation value

```

```
errorCount = errorCount << 6
```

```
toSave = topHalf
```

```
toSave = toSave >> 5;
```

```
toSave = toSave OR errorCount
```

```
mem[i-31] = toSave;
```

```
botHalf = botHalf >> 3;
```

```
temp = botHalf;
```

```
botHalf = botHalf AND 0000 0001;
```

```
temp = botHalf >> 1;
```

```
temp = temp AND 0000 1110;
```

```
botHalf = botHalf AND temp;
```

```
topBits = topHalf;
```

```
topBits = topBits << 3;
```

```
topBits = topBits AND 1111 0000;
```

```
botHalf = botHalf AND topBits;
```

```
mem[i-30] = botHalf;
```

```
i++;
```

```
i++;
```

```
// errorCount = f1f000 0000
```

```
// toSave = b11b10b9b8 b7b6b5p8 with incorrect bit flipped
```

```
// toSave = 0000 0b11b10b9 with incorrect bit flipped
```

```
// toSave = f1f000 0b11b10b9 with incorrect bit flipped
```

```
// botHalf = 000b4 b3b2p4b1 with incorrect bit flipped
```

```
// botHalf = 0000 000b1 with incorrect bit flipped
```

```
// temp = 0000 b4b3b2p4 with incorrect bit flipped
```

```
// temp = 0000 b4b3b20 with incorrect bit flipped
```

```
// botHalf = 0000 b4b3b2b1 with incorrect bit flipped
```

```
// topBits = b11b10b9b8 b7b6b5p8 with incorrect bit flipped
```

```
// topBits = b8b7b6b5 p8000 with incorrect bit flipped
```

```
// botHalf = b8b7b6b5 b4b3b2b1
```

Program 2 Assembly Code

```
i = 30
```

```
Movi r4 30
```

```

movi r7 0
while i < 60:
startloop:
    // calculating discrepancy vector
    parities = 0000 0000
Movi r0 0
    topHalf = mem[i+1]; // topHalf = b11b10b9b8 b7b6b5p8
Movi r3 r4
Inc r3
Load r1 r3
    botHalf = mem[i]; // botHalf = b4b3b2p4 b1p2p1p0
Load r2 r4
    topHalf = parity(topHalf); // topHalf = 0000 000s8
Par r1 r1
    parities = parities | topHalf; // parities = 0000 000s8
Or r0 r1
    parities = parities << 1; // parities = 0000 00s80
Shftl r0 1
    topHalf = mem[i+1]; // topHalf = b11b10b9b8 b7b6b5p8
Load r1 r3
    botHalf = mem[i]; // botHalf = b4b3b2p4 b1p2p1p0
Load r2 r4
    botHalf = botHalf AND 1111 0000 // topHalf = b11b10b9b8 0000
And r2 240
    topHalf = topHalf AND 1111 0000 // botHalf = b4b3b2p4 0000
And r1 240
    topHalf = topHalf XOR botHalf
Xor r1 r2
    topHalf = parity(topHalf); // topHalf = 0000 000s4
Par r1 r1
    parities = parities | topHalf; // parities = 0000 00s8s4
Or r0 r1

```

parities = parities << 1;	// parities = 0000 0s ₈ s ₄ 0
Shftl r0 1	
topHalf = mem[i+1];	// topHalf = b ₁₁ b ₁₀ b ₉ b ₈ b ₇ b ₆ b ₅ p ₈
Load r1 r3	
botHalf = mem[i];	// botHalf = b ₄ b ₃ b ₂ p ₄ b ₁ p ₂ p ₁ p ₀
Load r2 r4	
botHalf = botHalf AND 1100 1100	// topHalf = b ₁₁ b ₁₀ 00 b ₇ b ₆ 00
And r2 204	
topHalf = topHalf AND 1100 1100	// botHalf = b ₄ b ₃ 00 b ₁ p ₂ 00
And r1 204	
topHalf = topHalf XOR botHalf	
Xor r1 r2	
topHalf = parity(topHalf);	// topHalf = 0000 000s ₂
Par r1	
parities = parities topHalf;	// parities = 0000 0s ₈ s ₄ s ₂
Or r0 r1	
parities = parities << 1;	// parities = 0000 s ₈ s ₄ s ₂ 0
Shftl r0 1	
topHalf = mem[i+1];	// topHalf = b ₁₁ b ₁₀ b ₉ b ₈ b ₇ b ₆ b ₅ p ₈
Load r1 r3	
botHalf = mem[i];	// botHalf = b ₄ b ₃ b ₂ p ₄ b ₁ p ₂ p ₁ p ₀
Load r2 r4	
botHalf = botHalf AND 1010 1010	// topHalf = b ₁₁ 0b ₉ 0 b ₇ 0b ₅ 0
And r2 170	
topHalf = topHalf AND 1010 1010	// botHalf = b ₄ 0b ₂ 0 b ₁ 0p ₁ 0
And r1 170	
topHalf = topHalf XOR botHalf	
Xor r1 r2	
topHalf = parity(topHalf);	// topHalf = 0000 000s ₁
Par r1 r1	
parities = parities topHalf;	// parities = 0000 s ₈ s ₄ s ₂ s ₁

Or r0 r1	
parities = parities << 1;	// parities = 000s ₈ s ₄ s ₂ s ₁ 0
Shftl r0 1	
topHalf = mem[i+1];	// topHalf = b ₁₁ b ₁₀ b ₉ b ₈ b ₇ b ₆ b ₅ p ₈
Load r1 r3	
botHalf = mem[i];	// botHalf = b ₄ b ₃ b ₂ p ₄ b ₁ p ₂ p ₁ p ₀
Load r2 r4	
topHalf = topHalf XOR botHalf	
Xor r1 r2	
topHalf = parity(topHalf);	// topHalf = 0000 000s ₀
Par r1	
parities = parities topHalf;	// parities = 000s ₈ s ₄ s ₂ s ₁ s ₀
Or r0 r1	
topHalf = mem[i];	// topHalf = b ₁₁ b ₁₀ b ₉ b ₈ b ₇ b ₆ b ₅ p ₈
Load r1 r4	
botHalf = mem[i+1];	// botHalf = b ₄ b ₃ b ₂ p ₄ b ₁ p ₂ p ₁ p ₀
Load r2 r3	
errorBit = parities >> 1;	// errorBit = 0000 s ₈ s ₄ s ₂ s ₁ = errorLocation
store r0 r7;	//store parities in mem[0]
inc r7;	
Shftr r0 1//errorBit is r0	
temp = 0000 0001;	
Movi r1 1 // temp is r1	
temp = temp << (errorBit);	// temp will be a one hot encoding of the value of errorLocation
Shftl r1 r0	
if (temp == 0):	// errorLocation > 8
bne r1 0 else	
errorBit = errorBit AND 0000 1000	// temp = temp - 8
And r0 16	
temp = 0000 0001;	
Movi r1 1	

	temp = temp << (errorBit);	// temp will be a one hot encoding of the value of error
Shftl r1 r0	topHalf = mem[i];	// topHalf = $b_{11}b_{10}b_9b_8b_7b_6b_5p_8$
Load r2 r4	topHalf = topHalf XOR temp;	// flip topHalf's bit by errorLocation value - 8
Xor r1 r2	botHalf = mem[i+1];	// botHalf = $b_4b_3b_2p_4b_1p_2p_1p_0$
Load r3 r3		
Bne 0 1 endElse		
else:		
Else:	botHalf = mem[i+1];	// botHalf = $b_4b_3b_2p_4b_1p_2p_1p_0$
Load r3 r3	botHalf = botHalf XOR temp;	// flip botHalf's bit by errorLocation value
xor r3 r1	topHalf = mem[i];	
Load r2 r4		
EndElse:		
movr r5 r2	//save r2 (topHalf) in r5	
movr r6 r3	//save r3 (botHalf) in r6	
dec r7;		
load r0 r7;	//restore parities as r0	
errorCount = 0000 0000;		
movi r1 0;		
errorBit = parities >> 1;		// errorBit = 0000 $s_8s_4s_2s_1$ = errorLocation
movr r2 r0;		
shiftr r2 1;		
if (errorBits != 0):		
bne r2 0 endif;		
errorCount ++;		
inc r1; // r1 is errorCount		

```

endif;
    if (parities != 0):
bne r0 0 endif;
        errorCount ++;
inc r1;
endif;
    errorCount = errorCount << 6 // errorCount = f1f000 0000
shiftr r1 6;
    toSave = topHalf // toSave = b11b10b9b8 b7b6b5p8 with incorrect bit flipped
mov r2 r5;
    toSave = toSave >> 5; // toSave = 0000 0b11b10b9 with incorrect bit flipped
shiftr r2 5;
    toSave = toSave OR errorCount // toSave = f1f000 0b11b10b9 with incorrect bit flipped
or r2 r1;
    mem[i-31] = toSave;
store r2 r7
inc r7
    botHalf = botHalf >> 3; // botHalf = 000b4 b3b2p4b1 with incorrect bit flipped
mov r0 r6;
shiftr r0 3;
    temp = botHalf;
mov r1 r6;
    botHalf = botHalf AND 0000 0001; // botHalf = 0000 000b1 with incorrect bit flipped
and r0 1;
    temp = botHalf >> 1; // temp = 0000 b4b3b2p4 with incorrect bit flipped
shiftr r1 1;
    temp = temp AND 0000 1110; // temp = 0000 b4b3b20 with incorrect bit flipped
and r1 14
    botHalf = botHalf AND temp; // botHalf = 0000 b4b3b2b1 with incorrect bit flipped
and r0 r1
    topBits = topHalf; // topBits = b11b10b9b8 b7b6b5p8 with incorrect bit flipped
movr r2 r5;

```



```

    topBits = topBits << 3;
shiftr r2 3;
    topBits = topBits AND 1111 0000;
and r2 240;
    botHalf = botHalf AND topBits;
and r0 r2;
    mem[i-30] = botHalf;
store r0 r7;
inc r7;
    i++;
Inc r4
    i++;
Inc r4

```

// topBits = $b_8b_7b_6b_5$ p₈000 with incorrect bit flipped

// botHalf = $b_8b_7b_6b_5$ $b_4b_3b_2b_1$

Program 3 Pseudocode

```
accumulatorA = 0;
accumulatorB = 0;
pattern = mem[32] >> 3;           //pattern = 000p5 p4p3p2p1
countByte = 0;

i = 0;
while (i < 32):
    singleByte = mem[i];
    singleByte = singleByte & 0001 1111;           //singleByte = 000b5 b4b3b2b1
    if (singleByte == pattern):
        accumulatorA++;
        countByte = 1;
    singleByte = mem[i] >> 1;
    singleByte = singleByte & 0001 1111;           //singleByte = 000b6 b5b4b3b2
    if (singleByte == pattern):
        accumulatorA++;
        countByte = 1;
    singleByte = mem[i] >> 2;
    singleByte = singleByte & 0001 1111;           //singleByte = 000b7 b6b5b4b3
    if (singleByte == pattern):
        accumulatorA++;
        countByte = 1;
    singleByte = mem[i] >> 3;
    singleByte = singleByte & 0001 1111;           //singleByte = 000b8 b7b6b5b4
    if (singleByte == pattern):
        accumulatorA++;
        countByte = 1;
    if (countByte == 1):
        accumulatorB++;
    i++;
```

```
mem[33] = accumulatorA;
```

```
mem[34] = accumulatorB;
```

```
i = 0
```

```
while( i < 31):
```

```
    firstHalfByte = mem[i];
```

```
    secondHalfByte = mem[i+1];
```

```
    firstHalfByte = (firstHalfByte >> 4) ;
```

```
    secondHalfByte = (secondHalfByte << 4) & 0001 0000;
```

```
    firstHalfByte = firstHalfByte | secondHalfByte;
```

```
    if (firstHalfByte == pattern):
```

```
        accumulatorA++;
```

```
    firstHalfByte = firstHalfByte >> 1
```

```
    secondHalfByte = mem[i+1];
```

```
    secondHalfByte = (secondHalfByte << 3) & 0001 0000;
```

```
    firstHalfByte = firstHalfByte | secondHalfByte;
```

```
    if (firstHalfByte == pattern):
```

```
        accumulatorA++;
```

```
    firstHalfByte = firstHalfByte >> 1
```

```
    secondHalfByte = mem[i+1];
```

```
    secondHalfByte = (secondHalfByte << 2) & 0001 0000;
```

```
    firstHalfByte = firstHalfByte | secondHalfByte;
```

```
    if (firstHalfByte == pattern):
```

```
        accumulatorA++;
```

```
    firstHalfByte = firstHalfByte >> 1
```

```
    secondHalfByte = mem[i+1];
```

```
    secondHalfByte = (secondHalfByte << 1) & 0001 0000;
```

```
    firstHalfByte = firstHalfByte | secondHalfByte;
```

```
    if (firstHalfByte == pattern):
```

```
        accumulatorA++;
```

```
    i++;
```

```
mem[35] = accumulatorA;
```

```
//firstHalfByte = b8b7b6b5 b4b3b2b1
```

```
//secondHalfByte = b16b15b14b13 b12b11b10b9
```

```
//firstHalfByte = 0000 b8b7b6b5
```

```
//secondHalfByte = 000b9 0000
```

```
//firstHalfByte = 000b9 b8b7b6b5
```

```
//firstHalfByte = 0000 b9b8b7b6
```

```
//secondHalfByte = 000b10 0000
```

```
//firstHalfByte = 000b10 b9b8b7b6
```

```
//firstHalfByte = 0000 b10b9b8b7
```

```
//secondHalfByte = 000b11 0000
```

```
//firstHalfByte = 000b11 b10b9b8b7
```

```
//firstHalfByte = 0000 b11b10b9b8
```

```
//secondHalfByte = 000b12 0000
```

```
//firstHalfByte = 000b12 b11b10b9b8
```

Program 3 Assembly Code

```
    accumulatorA = 0;
movi r5 0;
    accumulatorB = 0;
movi r6 0;
    pattern = mem[32] >> 3;           //pattern = 000p5 p4p3p2p1
movi r3 32;
load r0 r1;
movi r3 3;
shiftr r0 r3;
    i = 0;
movi r4 0;
movi r7 33;
    while (i < 32):
startloop:
    countByte = 0;
movr r2 = 0;
    singleByte = mem[i];
movr r3 r4;
load r1 r3;
    singleByte = singleByte & 0001 1111;           //singleByte = 000b5 b4b3b2b1
movr r3 31;
and r1 r3;
    if (singleByte == pattern):
bne r1 r0 endif
    accumulatorA++;
inc r5;
    countByte = 1;
```

```
mov r2 1;
endif:
```

```
    singleByte = mem[i] >> 1;
```

```
movr r3 r4;
load r1 r3;
movi r3 1;
shiftr r1 r3;
```

```
    singleByte = singleByte &
```

```
//singleByte = 000b6 b5b4b3b2
```

```
movi r3 31;
and r1 r3;
```

```
    if (singleByte == pattern):
```

```
bne r1 r0 endif;
```

```
        accumulatorA++;
```

```
inc r5;
```

```
        countByte = 1;
```

```
mov r2 1;
endif:
```

```
    singleByte = mem[i] >> 2;
```

```
movr r3 r4;
load r1 r3;
movi r3 2;
shiftr r1 r3;
```

```
    singleByte = singleByte & 0001 1111;
```

```
//singleByte = 000b7 b6b5b4b3
```

```
movi r3 31;
and r1 r3;
```

```
    if (singleByte == pattern):
```

```
bne r1 r0 endif;
```

```
        accumulatorA++;
```

```
inc r5;
```

```
        countByte = 1;
```

```
mov r2 1;
```

```

endif:
    singleByte = mem[i] >> 3;
movr r3 r4;
load r1 r3;
movi r3 3;
shiftr r1 r3;
    singleByte = singleByte & 0001 1111;           //singleByte = 000b8 b7b6b5b4
movi r3 31;
and r1 r3;
    if (singleByte == pattern):
bne r1 r0 endif;
        accumulatorA++;
inc r5;
        countByte = 1;
mov r2 1;
endif:
    if (countByte == 1):
movi r1 1
bne r1 r2 endif;
        accumulatorB++;
inc r6;
        i++;
inc r4;
movi r1 32;
bne r4 r1 startloop;
    mem[33] = accumulatorA;
store r5 r7;
inc r7;
    mem[34] = accumulatorB;
store r6 r7;
inc r7;
    i = 0

```

```

movi r4 0;
    while( i < 31):
startloop2:
    firstHalfByte = mem[i];                //firstHalfByte = b8b7b6b5 b4b3b2b1
    movr r3 r4;
    load r1 r3;
    secondHalfByte = mem[i+1];            //secondHalfByte = b16b15b14b13 b12b11b10b9
    inc r3;
    load r2 r3;
    firstHalfByte = (firstHalfByte >> 4) ;    //firstHalfByte = 0000 b8b7b6b5
    load r3 4;
    shiftr r1 r3;
    secondHalfByte = (secondHalfByte << 4) & 0001 0000;    //secondHalfByte = 000b9 0000
    shiftr r2 r3;
    movi r3 16;
    and r2 r3;
    firstHalfByte = firstHalfByte | secondHalfByte;    //firstHalfByte = 000b9 b8b7b6b5
    or r1 r2;
    if (firstHalfByte == pattern):
    bne r1 r0 endif;
        accumulatorA++;
    inc r5;
    endif:
    firstHalfByte = firstHalfByte >> 1    //firstHalfByte = 0000 b9b8b7b6
    mov r3 1;
    shiftr r1 r3;
    secondHalfByte = mem[i+1];
    movr r3 r4;
    inc r3;
    load r2 r3;
    secondHalfByte = (secondHalfByte << 3) & 0001 0000;    //secondHalfByte = 000b10 0000
    movi r3 3;

```

```

shiftl r2 r3;
movi r3 16;
and r2 r3;
    firstHalfByte = firstHalfByte | secondHalfByte;           //firstHalfByte = 000b10 b9b8b7b6
or r1 r2;
    if (firstHalfByte == pattern):
bne r1 r0 endif;
    accumulatorA++;

inc r5;
endif:
    firstHalfByte = firstHalfByte >> 1                       //firstHalfByte = 0000 b10b9b8b7
movi r3 1;
shiftr r1 r3;
    secondHalfByte = mem[i+1];
movr r3 r4;
inc r3;
load r2 r3;
    secondHalfByte = (secondHalfByte << 2) & 0001 0000;      //secondHalfByte = 000b11 0000
movi r3 2;
shiftl r2 r3;
movi r3 16;
and r2 r3;
    firstHalfByte = firstHalfByte | secondHalfByte;           //firstHalfByte = 000b11 b10b9b8b7
or r1 r2;
    if (firstHalfByte == pattern):
bne r1 r0 endif;
    accumulatorA++;

inc r5;
endif:
    firstHalfByte = firstHalfByte >> 1                       //firstHalfByte = 0000 b11b10b9b8
movi r3 1;
shiftr r1 r3;

```



```

        secondHalfByte = mem[i+1];
movr r3 r4;
inc r3;
load r2 r3;

        secondHalfByte = (secondHalfByte << 1) & 0001 0000;    //secondHalfByte = 000b12 0000
movi r3 1;
shifl r2 r3;
movi r3 16;
and r2 r3;

        firstHalfByte = firstHalfByte | secondHalfByte;        //firstHalfByte = 000b12 b11b10b9b8
or r1 r2;

        if (firstHalfByte == pattern):
bne r1 r0 endif;

                accumulatorA++;

inc r5;
endif:
inc r4;

        i++;
movi r3 31;
bne r4 r3 startloop2;
        mem[35] = accumulatorA;
store r5 r7;
inc r7;

```