CSE 141L Milestone 2

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

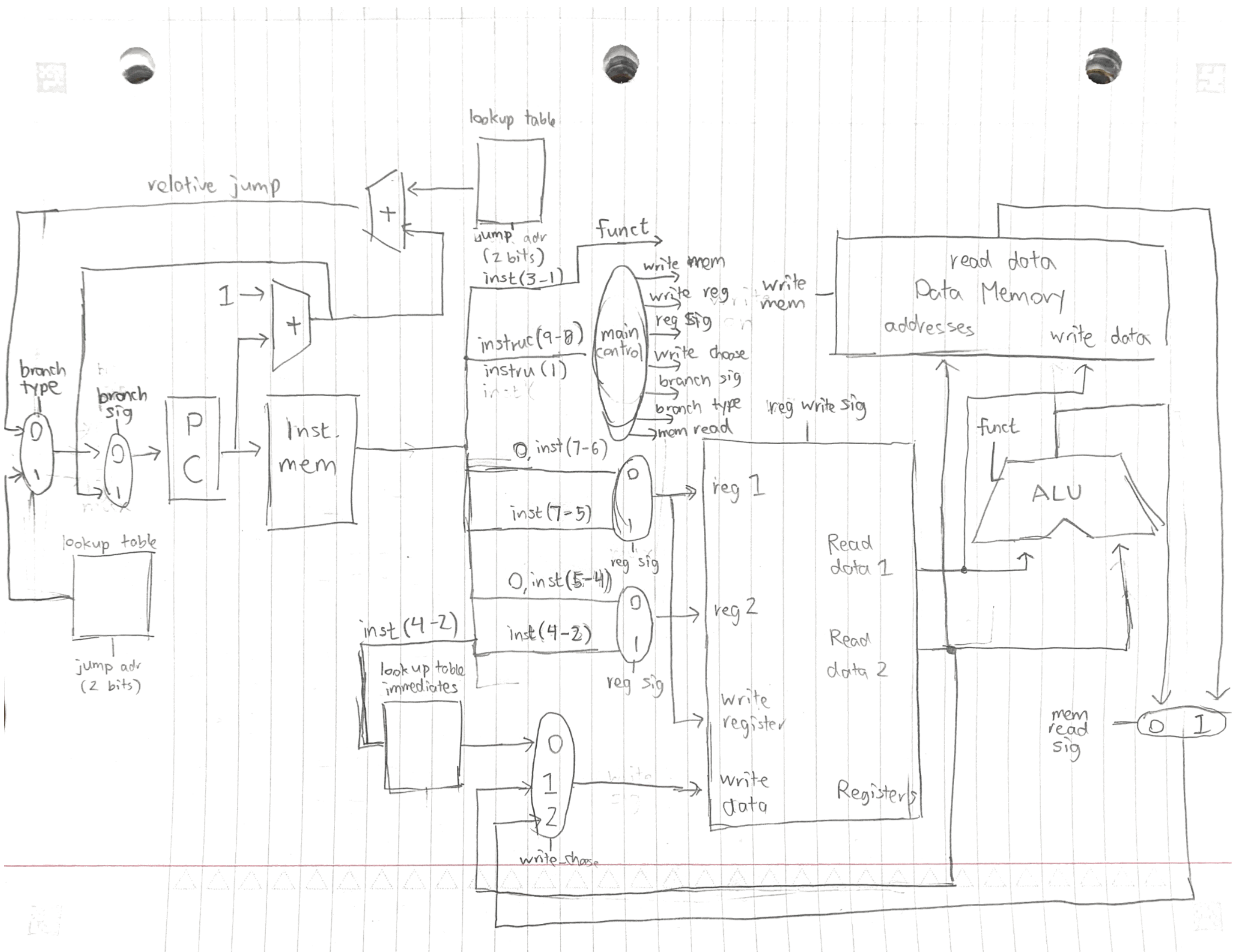
0. Team

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 (no ALU instruction is 000), 2 bit operand 1, 2 bit operand 2	add, shftl, shftr, and, or, xor, par
A	6 bit opcode, 3 bit register	inc, dec,
J	3 bit opcode, 2 bit branch target, 2 bit operand 1, 2 bit operand 2	bne
I	3 bit opcode, 1 bit funct, 3 bit operand 1, 3 bit operand 2	load, store, movr
L	3 bit opcode, 6 bit lookup table index	movi

Changelog

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
Inc = Increment	R	6 bits opcode (000000), 3 bit register (XXX)	# Assume R0 = 0b0000_0000 inc R0 = 000000_000	Will increment the specified register, unlike other R types, increment takes a 3 bit register

			# After: R0 = 0b0000_0001	
Dec = Decrement	R	6 bits opcode (000001), 3 bit register (XXX)	# Assume R0 = 0b0000_0010 dec R0 = 000001_000 # After: R0 = 0b0000_0001	Will decrement the specified register, unlike other R types, increment takes a 3 bit register Shares the same funct with inc
shftl = Shift Left	R	2 bits opcode (00),3 bits funct(001), 2 bits operand register(XX), 2 bits operand register (XX)	# Assume R0 = 0b0000_0010 # Assume R1 = 0b0000_0011 shiftl R0 R1 = 00_001_00_01 # After: R0 = 0b0001_0000	Shift will use a shift in value of 0
shftr = Shift Right	R	2 bits opcode (00), 3 bits funct(010), 2 bits operand register(XX), 2 bits operand register (XX)	# Assume R0 = 0b0000_0010 # Assume R1 = 0b0000_0001 shiftr R0 R1 = 00_010_00_01 # After: R0 = 0b0000_0001	Shift will use a shift in value of 0
And = Bitwise And	R	2 bits opcode (00), 3 bits funct(011), 2 bits operand register(XX), 2 bits operand register (XX)	# Assume R0 = 0b0110_0010 # Assume R1 = 0b1011_1100 and R0 R1 = 00_011_00_01 # 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), 3 bits funct(100), 2 bits operand register(XX), 2 bits operand register (XX)	# Assume R0 = 0b0110_0010 # Assume R1 = 0b1011_0000 or R0 R1 = 00_100_00_01	The first operand register is also the destination register or r0 r1 -> r0 = r0 or r1

			# After: R0 = 0b1111_0010 # R1 = 0b1011_0000	
XOR = Bitwise XOR	R	2 bits opcode (00), 3 bits funct(101), 2 bits operand register(XX), 2 bits operand register (XX)	# Assume R0 = 0b0110_0010 # Assume R1 = 0b1011_0000 xor R0 R1 = 00_101_00_01 # After: R0 = 0b1101_0010 # R1 = 0b1011_0000	The first operand register is also the destination register xor r0 r1 -> r0 = r0 xor r1
Noop	R	2 bits opcode (00), 3 bits funct(110), 2 bits operand register(XX), 2 bits operand register (XX)	noop = 00_110_XX_XX	Does no operation
par = Parity Bit	R	2 bits opcode (00), 3 bits funct(111), 2 bits destination register(XX), 2 bits operand register (XX)	# Assume R0 = 0b0110_0010 # Assume R1 = 0b1011_0000 par R0 R1 = 00_111_00_01 # 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 (relative jump)	J	3 bits opcode (100), 2 bit branch target (XX), 2 bits operand register(XX), 2 bits operand register (XX)	# Assume R0 = 0b0110_0010 # Assume R1 = 0b0110_0010 beq R0 R1 6 = 100_00_00_01 # After: R0 = 0b0110_0010 # R1 = 0b0110_0010	Will perform a jump that is x instructions forward or backward where the value of x is retrieved from a lookup table
bne = branch not equal (absolute	J	3 bits opcode (101), 2 bit branch target (XX), 2 bits operand register(XX), 2 bits operand register (XX),	# Assume R0 = 0b0110_0010 # Assume R1 = 0b0110_0010 beq R0 R1 endloop = 101_00_00_01	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

jump)			# After: R0 = 0b0110_0010 # R1 = 0b0110_0010	
load	I	3 bits opcode(010), 3 bits operand register (XXX), 3 bits operand address register (XXX)	# Assume mem[0] = 0b0110_1000 # Assume r1 = 0b0000_0000 load r0 r1 = 010_000_001 # After : r0 = 0b0110_1000	Load and Store have the same opcode but are instead controlled by a control bit.
store	I	3 bits opcode(011), 3 bits operand register (XXX), 3 bits operand address register (XXX)	# Assume r0 = 0b0110_1000 # Assume r1 = 0b0000_0000 store r0 r1 = 011_000_001 # After :mem[0] = 0b0110_1000	
movI = move immediat e	L	3 bits opcode(110), 6 bit immediate lookup (XXXXXX)	movi r3 3 = 110_000_011_0 #After : r0 = 0b0000_0011	Will load into r3 an immediate value given by the lookup table. r3 is the only argument that movi accepts
movr = move registers	I	3 bits opcode(111), 3 bit destination register (XXX), 3 bit source register (XXX)	# Assume r1 = 0b0011_1111 movr r0 r1 = 111_000_001 # 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. For example, in program 2, the output has to be written at mem[31]. Therefore, at the start of the program, r7 will be initialized to 31, and whenever an output is written 31 will be incremented. If we have to temporarily write to memory from running out of registers, we would write to r7+1, then when we want to use the value again, we would “pop” it by loading the value into a register, then decrementing r7 back. 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. The 2 bit will be part of the branch instruction. 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. The decision for the type of jump will be specified by the 3rd last bit of the branch instruction.

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]

4.1
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.

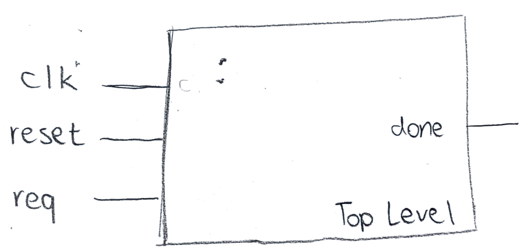
4.2
Since our architecture is very similar to MIPS, most basic MIPS instructions can be copied over with some adjustments. The main adjustment is that our instructions only take two registers where one register is both the operand and the destination register. However, all the instructions in our architecture have a similar instruction in MIPS.

4.3
Our ALU will only perform arithmetic operations. Branch calculation will be done in the PC, but comparison will be done in the ALU where subtraction will be performed, and the zero flag will be used as an indication of equality.

5. Individual Component Specification

Top Level (Module: top_level.sv)

At the top level, the processor has three inputs and one input and it will interact with the test bench. The clock input will be used to synchronize all the components of the processor, the reset flag and req flags will be used to start the program.



Program Counter (Module: PC.sv)

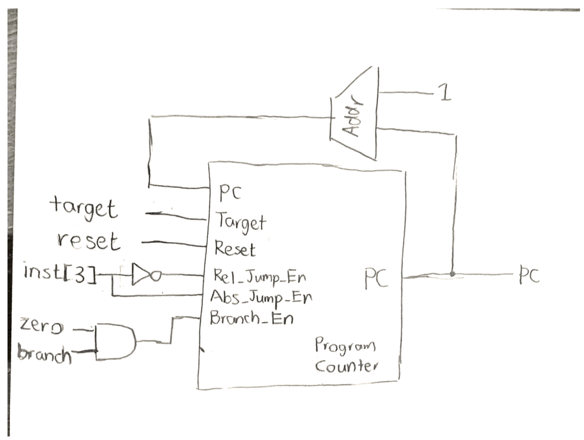
The program counter is the component whose output is the next address in instruction memory that our program is on. If no branching is done, it will increment PC by 1. If branching is done, both relative branching and absolute branching are possible and is decided by the last bit of the instruction. The target will be the output of a LUT which looks at 3 bits in the instruction.
Input:

- Program Counter: The current value of Program Counter + 1.
- Target: The distance to jump by, this will be the output Branch LUT.
- Reset: When set, we will set Program Counter back to 0.
- Rel_Jump_En: When set, we will perform Relative Branching. This will be NOT Inst[1].
- Abs_Jump_En: When set, we will perform Absolute Branching. This will be Inst[1].
- Branch_En: When set, we will perform branching. This will be (NOT ALU's Zero Flag AND Branch control signal)

Output:

- Program Counter: The new value of Program Counter

Branch _En	Rel_ Jump _En	Abs_ Jump _En	Program Counter
0	x	x	PC = PC + 1
1	1	0	PC = PC + target+1
1	0	1	PC = target+1



Instruction Memory (Module: instr_ROM.sv)

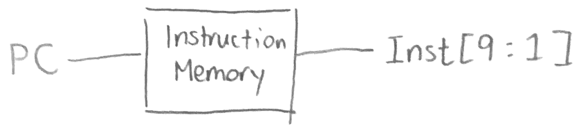
The instruction memory takes as input PC which is an address and outputs the instruction in instruction memory at that particular address.

Input:

- Program Counter (PC): The location of the requested instruction in instruction Memory

Output:

- Instruction = $I_9I_8I_7I_6I_5I_4I_3I_2I_1$ at location PC in instruction Memory



PC
Instruction Memory
Inst[9:1]

Control Decoder (Module: Control.sv)

Control deciphers the instruction using the OpCode, $Funct_1$ and $Funct_2$ and sets the control bits accordingly.

Inputs:

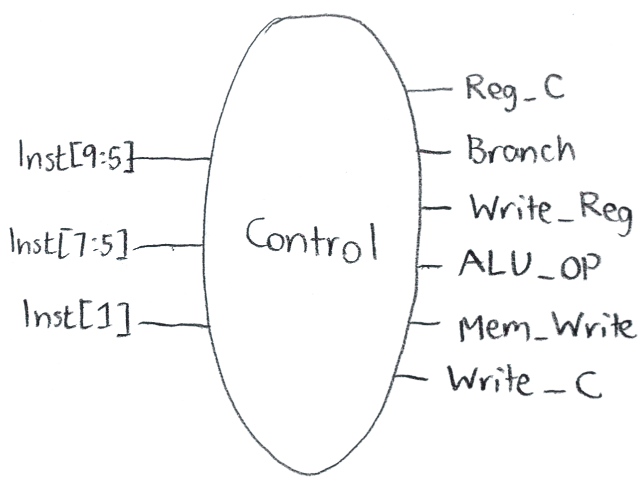
- OP Code: Inst[9:4]
- $Funct_1$: Inst[7:5]
- $Funct_2$: Inst[1]

Outputs:

- Reg_C : Signal to choose between four different register schemes (which bits are register bits)
- Branch : Signal to perform a branching operation
- Write_Reg : Signal to determine if we write data to the write register
- ALU_OP : Signal for ALU to choose instruction
- Mem_Write : Signal to determine if we write to memory
- Write_C : Signal to choose between what data to write to memory

Instr	Input		Output (Control Signals)					
	OP Code Inst[9:4]	$Funct_1$ Inst[7:5]	Reg_C	Branch	Write_Reg	ALU_OP	Mem_Write	Write_C
inc	000000	xxx	00	0	1	10	0	00
dec	000001	xxx	00	0	1	01	0	00
shftl	00xxxx	001	01	0	1	00	0	00
shftr	00xxxx	010	01	0	1	00	0	00
and	00xxxx	011	01	0	1	00	0	00
or	00xxxx	100	01	0	1	00	0	00
xor	00xxxx	101	01	0	1	00	0	00
noop	00xxxx	110	01	0	0	00	0	00

par	00xxxx	111	01	0	1	00	0	00
load	010xxx	xxx	10	0	1	xx	0	01
store	011xxx	xxx	10	0	0	xx	1	xx
bne(rel)	100xxx	xxx	01	1	0	11	0	xx
bne(abs)	101xxx	xxx	01	1	0	11	0	xx
movr	111xxx	xxx	10	0	1	xx	0	11
movl	110xxx	xxx	11	0	1	xx	0	10



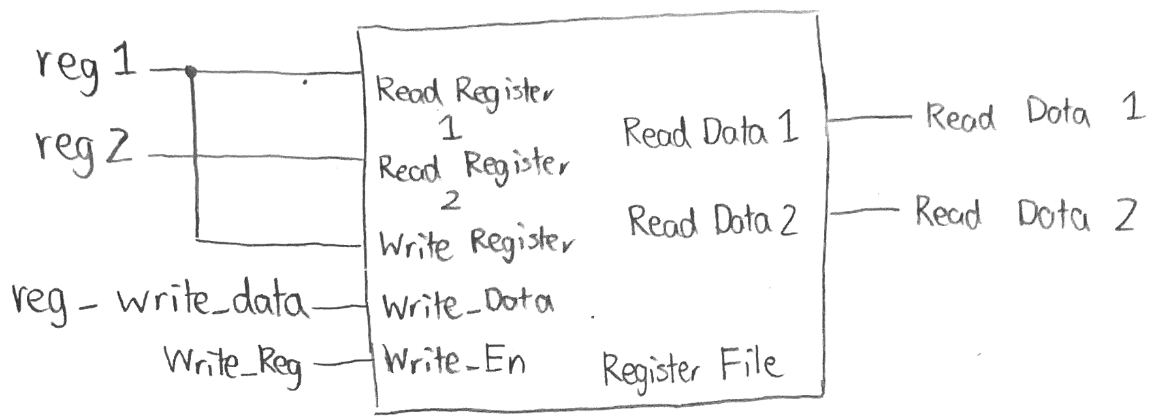
Register File (Module: reg_file.sv)

Inputs:

- Read Register 1 : the register number to read from, this is the output of Register Read Choose Mux 1
- Read Register 2 : the register number to read from, this is the output of Register Read Choose Mux 2
- Write Register: the register number to write to, this is the output of Register Read Choose Mux 1
- Write_Data: the data to write into register, this is the output of Register Write Choose Mux
- Write_En : if enabled, we will write back to register determined by Write Register. This is the Write_Reg control signal.

Outputs:

- Read Data 1: The contents of the register specified by input Read Register 1
- Read Data 2: The contents of the register specified by input Read Register 2



ALU (Module: alu.sv)

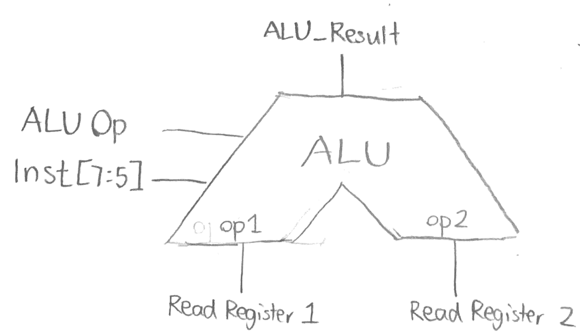
Inputs:

- Operand 1 : the first operand, this is the output of Read Register 1
- Operand 2 : the second operand, this is the output of Read Register 2
- ALUOp : part of the control signal to decide the ALU Operation
- Funct (Inst [7:5]) : part of the control signal to decide the ALU Operation

Outputs:

- ALU Result : Result of the operation that was performed
- Zero Flag : Flag which indicates if result was zero

ALUOp	Inst [7:5]	ALU Operation
10	000	Increment, return op1 + 1
01	000	Decrement, return op1 - 1
00	001	Shift Left, return op1 << op2
00	010	Shift Right, return op1 >> op2
00	011	And, return op1 & op2
00	100	Or, return op1 | op2
00	101	Xor, return op1 xor op2
00	110	No Operation
00	111	Parity, return ^(op1)
11	xxx	Subtract, return op1 - op2



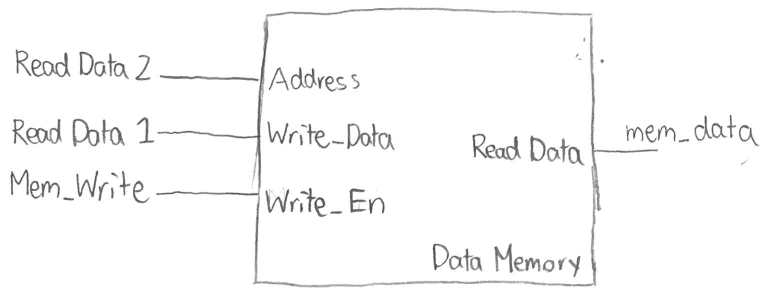
Data Memory (Module: dat_mem.sv)

Inputs:

- Address: Address in Data Memory that we want to read or write to. This is the output of Read Data 2.
- Write Data: The data that would be written to memory. This is the output of Read Data 1.
- Write_En: If set, we will write to memory. This will be the control signal Mem_Write.

Outputs

- Read Data: If Mem_Write = 0, this will output the data at the location in memory specified by Address, otherwise this can return any value

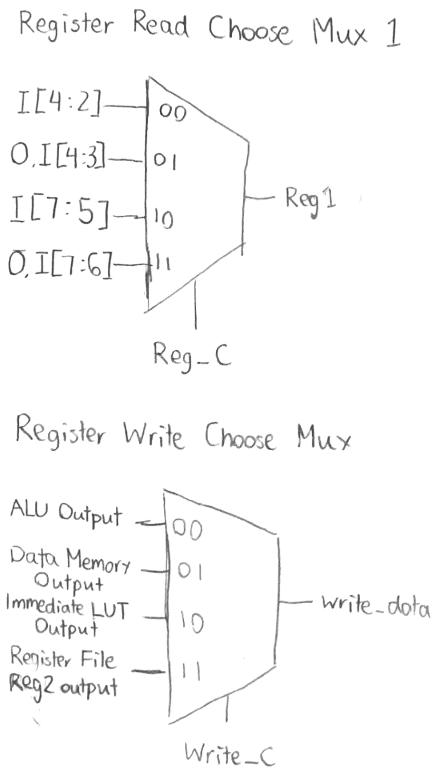


Lookup Tables (Module: PC_Lut.sv, IMM_Lut.sv)

Immediates Lookup Table (6 to 64)	
Input (Inst [4:2])	Output
000	0000 0000 (0)
001	0000 0001 (1)
010	0001 1110 (30)
011	0011 1100 (60)
100	
101	
110	
111	1111 1111 (255)

Jumps Lookup Table (3 to 8) 0-3 : Relative Jumps 4-7 : Absolute Jumps	
Input (Inst[3:1])	Output
000	3
001	
010	
011	
100	3
101	
110	
111	

Muxes (Module: register_c_mux.sv, write_c_mux.sv)



ALU Testing (alu_tb.sv)

```
VSIM 17> run -all
# ALU Instruction: 3A << 03
# Expected: 11010000
# Result: 11010000
# Zero: 0
# ALU Instruction: 3A >> 03
# Expected: 00000111
# Result: 00000111
# Zero: 0
# ALU Instruction: 3A & 03
# Expected: 00000010
# Result: 00000010
# Zero: 0
# ALU Instruction: 3A | 03
# Expected: 00111011
# Result: 00111011
# Zero: 0
# ALU Instruction: 3A ^ 03
# Expected: 00111001
# Result: 00111001
# Zero: 0
# ALU Instruction: No Op
# Expected: 00000000
# Result: 00000000
# Zero: 1
# ALU Instruction: ^(03)
# Expected: 00000000
# Result: 00000000
# Zero: 1
# ALU Instruction: ^(04)
# Expected: 00000001
# Result: 00000001
# Zero: 0
# ALU Instruction: 3A--
# Expected: 00111001
# Result: 00111001
# Zero: 0
# ALU Instruction: 3A++
# Expected: 00111011
# Result: 00111011
# Zero: 0
# ALU Instruction: 3A - 03
# Expected: 00110111
# Result: 00110111
# Zero: 0
# ALU Instruction: 3A - 3A
# Expected: 00000000
# Result: 00000000
# Zero: 1
```

PC Testing (pc_tb.sv)

```
# Initial PC:     0
# Performing Relative Jump, +4
# Current PC:     4
# Performing Absolute Jump To 8
# Current PC:     8
# No Jump Performed
# Current PC:     9
# Current PC:    10
# Current PC:    11
# Program Finished
# ** Note: $stop    : C:/Users/david/OneDrive/Desktop/CSE 141L Milestone 2/tests/pc_tb.sv(81)
#    Time: 60 ns  Iteration: 0  Instance: /PC_tb
# Break at C:/Users/david/OneDrive/Desktop/CSE 141L Milestone 2/tests/pc_tb.sv line 81
```

6. Program Implementation

Program 1 Pseudocode

```
i = 0
while i < 30:
        parities = 0000 0000
        topHalf = mem[i+1];                     // topHalf = 0000 0b_{11}b_{10}b_9
        botHalf = mem[i];                       // botHalf = b_8b_7b_6b_5 b_4b_3b_2b_1
        botHalf = botHalf AND 1111 0000         // botHalf = b_8b_7b_6b_5 0000
        topHalf = topHalf XOR botHalf           // topHalf = b_8b_7b_6b_5 0b_{11}b_{10}b_9
        topHalf = parity(topHalf);              // topHalf = 0000 000p_8
        parities = parities | topHalf;          // parities = 0000 000p_8
        parities = parities << 1;

        topHalf = mem[i+1];                     // topHalf = 0000 0b_{11}b_{10}b_9
        botHalf = mem[i];
        botHalf = botHalf AND 1000 1110;        // botHalf = b_8000 b_4b_3b_20
        topHalf = topHalf XOR botHalf;          // topHalf = b_8000 b_4(b_{11}⊕b_3)(b_{10}⊕b_2)b_9
        topHalf = parity(topHalf);              // topHalf = 0000 000p_4
        topHalf = topHalf << 4;
        parities = parities | topHalf;          // parities = 000p_4 000p_8

        topHalf = mem[i+1];
        botHalf = mem[i];
        botHalf = botHalf AND 0110 1101;        // botHalf = 0b_7b_60 b_4b_30b_1
        topHalf = topHalf AND 0000 0110;        // topHalf = 0000 0b_{11}b_{10}0
        topHalf = topHalf XOR botHalf;          // topHalf = 0b_7b_60 b_4(b_3⊕b_{11})b_{10}b_1
        topHalf = parity(topHalf);              // topHalf = 0000 000p_2
        topHalf = topHalf << 2;
        parities = parities | topHalf;          // parities = 000p_4 0p_20p_8
```

```
topHalf = mem[i+1];
botHalf = mem[i];
botHalf = botHalf AND 0101 1011;        // botHalf = $0b_70b_5\ b_40b_2b_1$
topHalf = topHalf AND 0000 0101;        // topHalf = $0000\ 0b_{11}0b_9$
topHalf = topHalf XOR botHalf;          // topHalf = $0b_70b_50\ b_4b_{11}b_2(b_1 \oplus b_9)$
topHalf = parity(topHalf);              // topHalf = $0000\ 000p_1$
topHalf = topHalf << 1;
parities = parities | topHalf;          // parities = $000p_4\ 0p_2p_1p_8$

topHalf = mem[i+1];                     // topHalf = $0000\ 0b_{11}b_{10}b_9$
botHalf = mem[i];                       // botHalf = $b_8b_7b_6b_5\ b_4b_3b_2b_1$
topHalf = topHalf << 5;                 // topHalf = $b_{11}b_{10}b_90\ 0000$
topHalf = topHalf OR parities;          // topHalf = $b_{11}b_{10}b_9p_4\ 0p_2p_1p_8$
topHalf = topHalf AND 1110 0001;        // topHalf = $b_{11}b_{10}b_90\ 000p_8$
botHalf = botHalf >> 4 << 1;            // botHalf = $000b_8\ b_7b_6b_50$
topHalf = topHalf OR botHalf;           // topHalf = $b_{11}b_{10}b_9b_8\ b_7b_6b_5p_8$
mem[30 + i + 1] = topHalf;
parities = parities AND 1111 1110;      // parities = $000p_4\ 0p_2p_10$

botHalf = mem[i];                       // botHalf = $b_8b_7b_6b_5\ b_4b_3b_2b_1$
botHalf = botHalf << 4;                 // botHalf = $b_4b_3b_2b_1\ 0000$
temp = botHalf;
temp = temp << 3;                       // temp = $b_1000\ 0000$
temp = temp >> 4;                       // temp = $0000\ b_1000$
botHalf = botHalf AND 1110 0000         // botHalf = $b_4b_3b_20\ 0000$
botHalf = botHalf OR temp               // botHalf = $b_4b_3b_20\ b_1000$
botHalf = botHalf OR parities           // botHalf = $b_4b_3b_2p_4\ b_1p_2p_10$

topHalf = mem[30 + i + 1];              // topHalf = $b_{11}b_{10}b_9b_8\ b_7b_6b_5p_8$
topHalf = parity(topHalf);              // topHalf = $0000\ 000(\ \hat{}(b_{11}:b_5,\ p_8)\ )$
parities = parity(botHalf);             // parities = $0000\ 000(\ \hat{}(b_4:b_1,\ p_4,\ p_2,\ p_1)\ )$
```

```
parities = parities XOR topHalf                    // parities = 0000 000p0
botHalf = botHalf OR parities                // botHalf = b4b3b2p4 b1p2p1p0

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

Reference:

mem[1] = 00000101 -- 5 b11:b9

mem[0] = 01010101 -- b8:b1

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 $0b_{11}b_{10}b_9$
Movi r3 r4
Inc r3
Load r1 r3 //topHalf = r1
    botHalf = mem[i];
Load r2 r4 // botHalf = $b_8b_7b_6b_5$ $b_4b_3b_2b_1$
    botHalf = botHalf AND 1111 0000 // botHalf = $b_8b_7b_6b_5$ 0000
And r2 240
    topHalf = topHalf XOR botHalf // topHalf = $b_8b_7b_6b_5$ $0b_{11}b_{10}b_9$
Xor r1 r2
    topHalf = parity(topHalf); // topHalf = 0000 $000p_8$
Par r1 r1
    parities = parities | topHalf; // parities = 0000 $000p_8$
Or r0 r3
    parities = parities << 1;
Shftl r0 r0
    topHalf = mem[i+1]; // topHalf = 0000 $0b_{11}b_{10}b_9$
Load r1 r3
    botHalf = mem[i];
Load r2 r4
    botHalf = botHalf AND 1000 1110; // botHalf = b_8000 $b_4b_3b_20$
And r2 142
    topHalf = topHalf XOR botHalf; // topHalf = b_8000 $b_4(b_{11}\oplus b_3)(b_{10}\oplus b_2)b_9$

```
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 = $0b_7 0b_5 0\ b_4 b_{11} b_2 (b_1 \oplus b_9)$

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 = b1000 0000
Shftl r5 3
        temp = temp >> 4;                    // temp = 0000 b1000
Shftr r5 4
        botHalf = botHalf AND 1110 0000      // botHalf = b4b3b20 0000
And r2 224
        botHalf = botHalf OR temp            // botHalf = b4b3b20 b1000
Or r2 r5
        botHalf = botHalf OR parities        // botHalf = b4b3b2p4 b1p2p10
Or r2 r0

        topHalf = mem[30 + i + 1];           // topHalf = b11b10b9b8 b7b6b5p8
load r1 r7
        topHalf = parity(topHalf);           // topHalf = 0000 000( ^(b11:b5, p8) )
Par r1 r1
        parities = botHalf;
Movi r0 r2
        parities = parity(botHalf);          // parities = 0000 000( ^(b4:b1, p4, p2, p1) )
Par r0 r2
        parities = parities XOR topHalf      // parities = 0000 000p0
Xor r0 r1
        botHalf = botHalf XOR parities       // botHalf = b4b3b2p4 b1p2p1p0
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];                      // topHalf = b_{11}b_{10}b_9b_8 b_7b_6b_5p_8
        botHalf = mem[i];                        // botHalf = b_4b_3b_2p_4 b_1p_2p_1p_0
        topHalf = parity(topHalf);               // topHalf = 0000 000s_8
        parities = parities | topHalf;           // parities = 0000 000s_8
        parities = parities << 1;                // parities = 0000 00s_80

        topHalf = mem[i+1];                      // topHalf = b_{11}b_{10}b_9b_8 b_7b_6b_5p_8
        botHalf = mem[i];                        // botHalf = b_4b_3b_2p_4 b_1p_2p_1p_0
        botHalf = botHalf AND 1111 0000          // topHalf = b_{11}b_{10}b_9b_8 0000
        topHalf = topHalf AND 1111 0000          // botHalf = b_4b_3b_2p_4 0000
        topHalf = topHalf XOR botHalf
        topHalf = parity(topHalf);               // topHalf = 0000 000s_4
        parities = parities | topHalf;           // parities = 0000 00s_8s_4
        parities = parities << 1;                // parities = 0000 0s_8s_40

        topHalf = mem[i+1];                      // topHalf = b_{11}b_{10}b_9b_8 b_7b_6b_5p_8
        botHalf = mem[i];                        // botHalf = b_4b_3b_2p_4 b_1p_2p_1p_0
        botHalf = botHalf AND 1100 1100          // topHalf = b_{11}b_{10}00 b_7b_600
        topHalf = topHalf AND 1100 1100          // botHalf = b_4b_300 b_1p_200
        topHalf = topHalf XOR botHalf
        topHalf = parity(topHalf);               // topHalf = 0000 000s_2
        parities = parities | topHalf;           // parities = 0000 0s_8s_4s_2
        parities = parities << 1;                // parities = 0000 s_8s_4s_20

        topHalf = mem[i+1];                      // topHalf = b_{11}b_{10}b_9b_8 b_7b_6b_5p_8
        botHalf = mem[i];                        // botHalf = b_4b_3b_2p_4 b_1p_2p_1p_0
```

```
botHalf = botHalf AND 1010 1010              // topHalf = b_{11}0b_{9}0 b_{7}0b_{5}0
topHalf = topHalf AND 1010 1010              // botHalf = b_{4}0b_{2}0 b_{1}0p_{1}0
topHalf = topHalf XOR botHalf
topHalf = parity(topHalf);                   // topHalf = 0000 000s_{1}
parities = parities | topHalf;               // parities = 0000 s_{8}s_{4}s_{2}s_{1}
parities = parities << 1;                    // parities = 000s_{8} s_{4}s_{2}s_{1}0

topHalf = mem[i+1];                          // topHalf = b_{11}b_{10}b_{9}b_{8} b_{7}b_{6}b_{5}p_{8}
botHalf = mem[i];                            // botHalf = b_{4}b_{3}b_{2}p_{4} b_{1}p_{2}p_{1}p_{0}
topHalf = topHalf XOR botHalf
topHalf = parity(topHalf);                   // topHalf = 0000 000s_{0}
parities = parities | topHalf;               // parities = 000s_{8} s_{4}s_{2}s_{1}s_{0}

topHalf = mem[i];                            // topHalf = b_{11}b_{10}b_{9}b_{8} b_{7}b_{6}b_{5}p_{8}
botHalf = mem[i+1];                          // botHalf = b_{4}b_{3}b_{2}p_{4} b_{1}p_{2}p_{1}p_{0}

errorBit = parities >> 1;                    // errorBit = 0000 s_{8}s_{4}s_{2}s_{1} = errorLocation
temp = 0000 0001;
temp = temp << (errorBit);                   // temp will be a one hot encoding of the value of errorLocation
if (temp == 0):                              // errorLocation > 8
        errorBit = errorBit AND 0000 1000    // temp = temp - 8
        temp = 0000 0001;
        temp = temp << (errorBit);           // temp will be a one hot encoding of the value of error
        topHalf = topHalf XOR temp;          // flip topHalf's bit by errorLocation value - 8
else:
        botHalf = botHalf XOR temp;          // flip botHalf's bit by errorLocation value
errorCount = 0000 0000;
if (errorBits != 0):
        errorCount ++;
parrities = parities AND 0000 0001;
if (parrities != 0):
        errorCount ++;
```

errorCount = errorCount << 6 // errorCount = $f_1f_0 00\ 0000$

toSave = topHalf // toSave = $b_{11}b_{10}b_9b_8\ b_7b_6b_5p_8$ with incorrect bit flipped

toSave = toSave >> 5; // toSave = $0000\ 0b_{11}b_{10}b_9$ with incorrect bit flipped

toSave = toSave OR errorCount // toSave = $f_1f_0 00\ 0b_{11}b_{10}b_9$ with incorrect bit flipped

mem[i-31] = toSave;

botHalf = botHalf >> 3; // botHalf = $000b_4\ b_3b_2p_4b_1$ with incorrect bit flipped

temp = botHalf;

botHalf = botHalf AND 0000 0001; // botHalf = $0000\ 000b_1$ with incorrect bit flipped

temp = botHalf >> 1; // temp = $0000\ b_4b_3b_2p_4$ with incorrect bit flipped

temp = temp AND 0000 1110; // temp = $0000\ b_4b_3b_2 0$ with incorrect bit flipped

botHalf = botHalf AND temp; // botHalf = $0000\ b_4b_3b_2b_1$ with incorrect bit flipped

topBits = topHalf; // topBits = $b_{11}b_{10}b_9b_8\ b_7b_6b_5p_8$ with incorrect bit flipped

topBits = topBits << 3; // topBits = $b_8b_7b_6b_5\ p_8 000$ with incorrect bit flipped

topBits = topBits AND 1111 0000;

botHalf = botHalf AND topBits; // botHalf = $b_8b_7b_6b_5\ b_4b_3b_2b_1$

mem[i-30] = botHalf;

i++;

i++;

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 = $b_{11}b_{10}b_9b_8$ $b_7b_6b_5p_8$
Movi r3 r4
Inc r3
Load r1 r3
        botHalf = mem[i];                           // botHalf = $b_4b_3b_2p_4$ $b_1p_2p_1p_0$
Load r2 r4
        topHalf = parity(topHalf);                  // topHalf = 0000 $000s_8$
Par r1 r1
        parities = parities | topHalf;              // parities = 0000 $000s_8$
Or r0 r1
        parities = parities << 1;                   // parities = 0000 $00s_80$
Shftl r0 1
        topHalf = mem[i+1];                         // topHalf = $b_{11}b_{10}b_9b_8$ $b_7b_6b_5p_8$
Load r1 r3
        botHalf = mem[i];                           // botHalf = $b_4b_3b_2p_4$ $b_1p_2p_1p_0$
Load r2 r4
        botHalf = botHalf AND 1111 0000             // topHalf = $b_{11}b_{10}b_9b_8$ 0000
And r2 240
        topHalf = topHalf AND 1111 0000             // botHalf = $b_4b_3b_2p_4$ 0000
And r1 240
        topHalf = topHalf XOR botHalf
Xor r1 r2
        topHalf = parity(topHalf);                  // topHalf = 0000 $000s_4$
Par r1 r1
        parities = parities | topHalf;              // parities = 0000 $00s_8s_4$
Or r0 r1
```

parities = parities << 1; // parities = 0000 $0s_8s_40$

Shftl r0 1

topHalf = mem[i+1]; // topHalf = $b_{11}b_{10}b_9b_8$ $b_7b_6b_5p_8$

Load r1 r3

botHalf = mem[i]; // botHalf = $b_4b_3b_2p_4$ $b_1p_2p_1p_0$

Load r2 r4

botHalf = botHalf AND 1100 1100 // topHalf = $b_{11}b_{10}00$ b_7b_600

And r2 204

topHalf = topHalf AND 1100 1100 // botHalf = b_4b_300 b_1p_200

And r1 204

topHalf = topHalf XOR botHalf

Xor r1 r2

topHalf = parity(topHalf); // topHalf = 0000 $000s_2$

Par r1

parities = parities | topHalf; // parities = 0000 $0s_8s_4s_2$

Or r0 r1

parities = parities << 1; // parities = 0000 $s_8s_4s_20$

Shftl r0 1

topHalf = mem[i+1]; // topHalf = $b_{11}b_{10}b_9b_8$ $b_7b_6b_5p_8$

Load r1 r3

botHalf = mem[i]; // botHalf = $b_4b_3b_2p_4$ $b_1p_2p_1p_0$

Load r2 r4

botHalf = botHalf AND 1010 1010 // topHalf = $b_{11}0b_90$ b_70b_50

And r2 170

topHalf = topHalf AND 1010 1010 // botHalf = b_40b_20 b_10p_10

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 $s_8s_4s_2s_1$

```
Or r0 r1
        parities = parities << 1;                    // parities = 000s8 s4s2s10
Shftl r0 1

        topHalf = mem[i+1];                          // topHalf = b11b10b9b8 b7b6b5p8
Load r1 r3
        botHalf = mem[i];                            // botHalf = b4b3b2p4 b1p2p1p0
Load r2 r4
        topHalf = topHalf XOR botHalf
Xor r1 r2
        topHalf = parity(topHalf);                   // topHalf = 0000 000s0
Par r1
        parities = parities | topHalf;               // parities = 000s8 s4s2s1s0
Or r0 r1
                topHalf = mem[i];                            // topHalf = b11b10b9b8 b7b6b5p8
            Load r1 r4
                botHalf = mem[i+1];                          // botHalf = b4b3b2p4 b1p2p1p0
            Load r2 r3
        errorBit = parities >> 1;                    // errorBit = 0000 s8s4s2s1 = 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 = b11b10b9b8 b7b6b5p8
Load r2 r4
                    topHalf = topHalf XOR temp;                 // flip topHalf's bit by errorLocation value - 8
Xor r1 r2
                    botHalf = mem[i+1];                                   // botHalf = b4b3b2p4 b1p2p1p0
Load r3 r3
Bne 0 1 endElse
          else:
Else:
                    botHalf = mem[i+1];                                   // botHalf = b4b3b2p4 b1p2p1p0
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 s8s4s2s1 = errorLocation
movr r2 r0;
shiftr r2 1;
          if (errorBits != 0):
bne r2 0 endif;
                    errorCount ++;
inc r1; // r1 is errorCount
```

endif;
if (parrities != 0):
bne r0 0 endif;
errorCount ++;
inc r1;
endif;
errorCount = errorCount << 6 // errorCount = $f_1f_0$00 0000
shiftl r1 6;
toSave = topHalf // toSave = $b_{11}b_{10}b_9b_8$ $b_7b_6b_5p_8$ with incorrect bit flipped
mov r2 r5;
toSave = toSave >> 5; // toSave = 0000 $0b_{11}b_{10}b_9$ with incorrect bit flipped
shiftr r2 5;
toSave = toSave OR errorCount // toSave = $f_1f_0$00 $0b_{11}b_{10}b_9$ with incorrect bit flipped
or r2 r1;
mem[i-31] = toSave;
store r2 r7
inc r7
botHalf = botHalf >> 3; // botHalf = $000b_4$ $b_3b_2p_4b_1$ with incorrect bit flipped
mov r0 r6;
shiftr r0 3;
temp = botHalf;
mov r1 r6;
botHalf = botHalf AND 0000 0001; // botHalf = 0000 $000b_1$ with incorrect bit flipped
and r0 1;
temp = botHalf >> 1; // temp = 0000 $b_4b_3b_2p_4$ with incorrect bit flipped
shiftr r1 1;
temp = temp AND 0000 1110; // temp = 0000 $b_4b_3b_20$ with incorrect bit flipped
and r1 14
botHalf = botHalf AND temp; // botHalf = 0000 $b_4b_3b_2b_1$ with incorrect bit flipped
and r0 r1
topBits = topHalf; // topBits = $b_{11}b_{10}b_9b_8$ $b_7b_6b_5p_8$ with incorrect bit flipped
movr r2 r5;

```
        topBits = topBits << 3;                          // topBits = b8b7b6b5 p8000  with incorrect bit flipped
shiftr r2 3;
        topBits = topBits AND 1111 0000;
and r2 240;
        botHalf = botHalf AND topBits;                   // botHalf = b8b7b6b5 b4b3b2b1
and r0 r2;
        mem[i-30] = botHalf;
store r0 r7;
inc r7;
        i++;
Inc r4
        i++;
Inc r4
```

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];                                 //firstHalfByte = b_8b_7b_6b_5 b_4b_3b_2b_1
	secondHalfByte = mem[i+1];                              //secondHalfByte = b_16b_15b_14b_13 b_12b_11b_10b_9
	firstHalfByte = (firstHalfByte >> 4) ;                  //firstHalfByte = 0000 b_8b_7b_6b_5
	secondHalfByte = (secondHalfByte << 4) & 0001 0000;     //secondHalfByte = 000b_9 0000
	firstHalfByte = firstHalfByte | secondHalfByte;         //firstHalfByte = 000b_9 b_8b_7b_6b_5
	if (firstHalfByte == pattern):
		accumulatorA++;
	firstHalfByte = firstHalfByte >> 1                      //firstHalfByte = 0000 b_9b_8b_7b_6
	secondHalfByte = mem[i+1];
	secondHalfByte = (secondHalfByte << 3) & 0001 0000;     //secondHalfByte = 000b_10 0000
	firstHalfByte = firstHalfByte | secondHalfByte;         //firstHalfByte = 000b_10 b_9b_8b_7b_6
	 if (firstHalfByte == pattern):
		accumulatorA++;
	firstHalfByte = firstHalfByte >> 1                      //firstHalfByte = 0000 b_10b_9b_8b_7
	secondHalfByte = mem[i+1];
	secondHalfByte = (secondHalfByte << 2) & 0001 0000;     //secondHalfByte = 000b_11 0000
	firstHalfByte = firstHalfByte | secondHalfByte;         //firstHalfByte = 000b_11 b_10b_9b_8b_7
	 if (firstHalfByte == pattern):
		accumulatorA++;
	firstHalfByte = firstHalfByte >> 1                      //firstHalfByte = 0000 b_11b_10b_9b_8
	secondHalfByte = mem[i+1];
	secondHalfByte = (secondHalfByte << 1) & 0001 0000;     //secondHalfByte = 000b_12 0000
	firstHalfByte = firstHalfByte | secondHalfByte;         //firstHalfByte = 000b_12 b_11b_10b_9b_8
	 if (firstHalfByte == pattern):
		accumulatorA++;
	i++;
mem[35] = accumulatorA;
```

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
shiftl 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;
shiftl 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;
```

7. Changelog

- Milestone 3
    - 3. Machine Specification Changes
        - Added new instruction type L type for immediate loading which can now load up to 64 different immediate values
        - changed movi from a I type to L type and changed functionality to only load into register 3
- Milestone 2
    - 3. Machine Specification Changes
        - Added new instruction type A type for increment and decrement instruction which has opcode 00000
        - Adjusted R type instruction format to be opcode(00), funct(xxx), reg1(xx), reg2(xx)
        - Renamed control bits to funct bits to specify that they are for controlling functionality
        - Added example for r7 register usage
        - Added specification on how branch decision will be made based on branch instruction format
    - Added Section 4.3
    - Added Section 5. Individual Component Specification
- Milestone 1
    - Initial version