CSE 141L Milestone 3

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

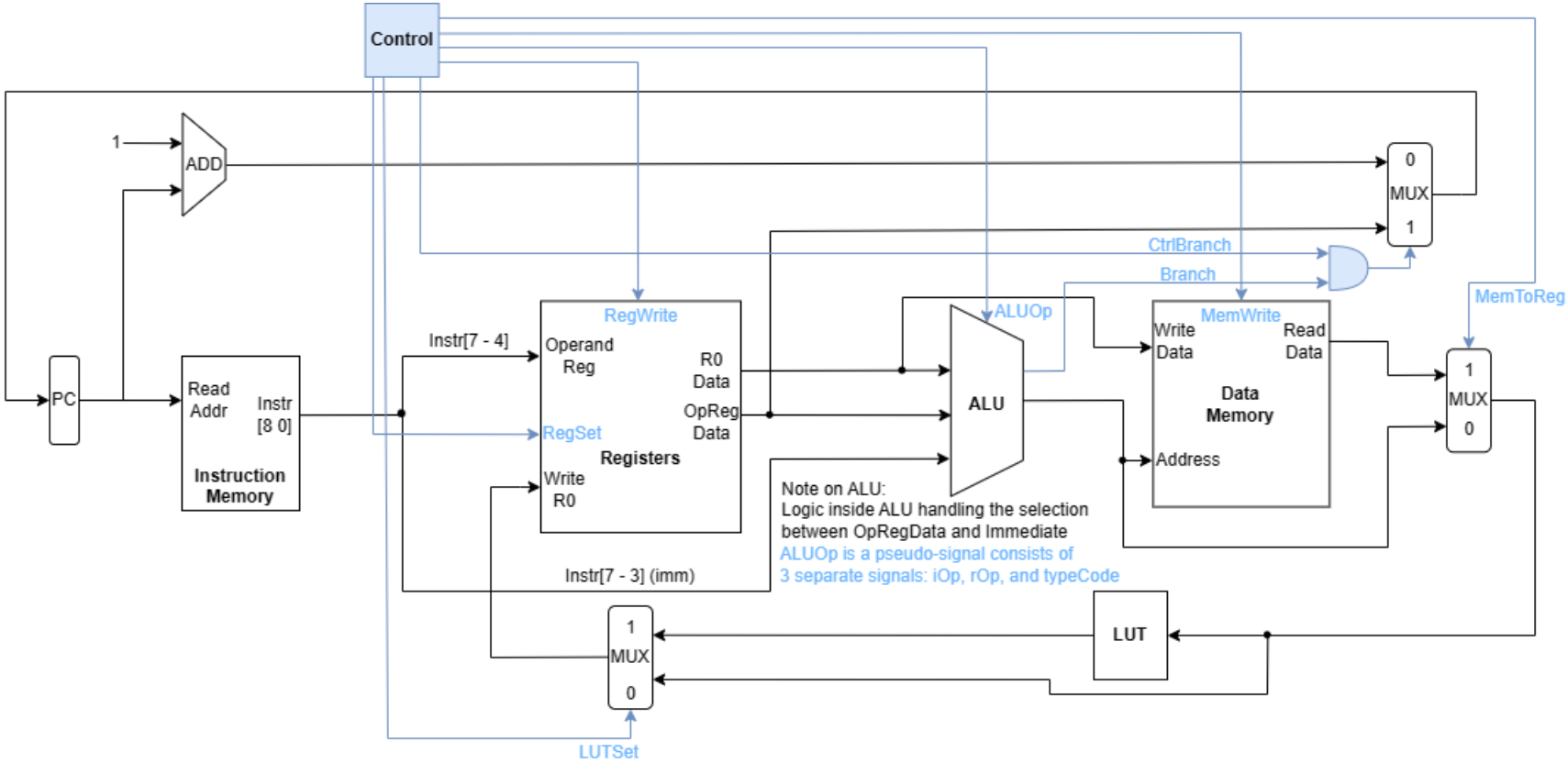
0. Team

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, sub, and, or, xor, rxor, slr, srr, lw, sw, eq, slt, br, j, set, la
I	1 bit type, 5 bits immediate, 3 bits funct	addi, subi, andi, sll, srl, seti, halt, luta

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
add = arithmetic addition	R	1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0000)	# Assume R0 (accumulator) has `0b0001_0001` ⇔ `17` # Assume R2 has `0b0001_0000` ⇔ `16` `add R2` ⇔ `0_0010_0000` # after add instruction, R0 now holds `0b0010_0001` ⇔ `33`	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.
sub = arithmetic subtraction	R	1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0001)	# Assume R0 (accumulator) has `0b0001_0001` ⇔ `17` # Assume R2 has `0b0001_0000` ⇔ `16` `sub R2` ⇔ `0_0010_0001` # 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.

addi = arithmetic addition with immediate	I	1 bit type (1), 5 bit immediate (XXXXX), 3 bit funct (000)	# Assume R0 (accumulator) has `4` `addi 2` ⇔ `1_00010_000` # after addi instruction, R0 now holds `6`	Perform arithmetic addition between the value stored by the accumulator R0 and the immediate. Store the result in the accumulator R0.
subi = arithmetic subtraction with immediate	I	1 bit type (1), 5 bit immediate (XXXXX), 3 bit funct (001)	# Assume R0 (accumulator) has `6` `subi 2` ⇔ `1_00010_001` # after subi instruction, R0 now holds `4`	Perform 2's complement subtraction. Subtract the value of immediate from the value of R0 (IMM - R0). The result will be stored in the accumulator R0.
and = logical and	R	1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0010)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b1001_0000` `and R2` ⇔ `0_0010_0010` # 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 (0011)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b1001_0000` `or R2` ⇔ `0_0010_0011` # 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 (0100)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b1001_0000` `xor R2` ⇔ `0_0010_0100`	Bitwise XOR values stored in accumulator and the given register (e.g. R2) and store the

			# after xor instruction, R0 now holds `0b1000_0001`	result in the destination register R0 (accumulator).
rxor = reduction XOR on register	R	1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0101)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b1001_0000` `rxor R2` ⇔ `0_0010_0101` # after rxor instruction, R0 now holds `0`	Perform reduction XOR between every bits of the operand register (e.g. R2). Store the result in the destination register R0 (accumulator).
andi = logical and with immediate	I	1 bit type (1), 5 bit immediate (XXXXX), 3 bit funct (010)	# Assume R0 (accumulator) has `0b0001_0001` `andi 1` ⇔ `1_00001_010` # after andi instruction, R0 now holds `0b0000_0001`	Bitwise AND values stored in accumulator and the immediate operand. Then store the result in the accumulator. Useful for bit masking.
sll = shift left logical	I	1 bit type (1), 5 bit immediate (XXXXX), 3 bit funct (011)	# Assume R0 (accumulator) has `0b0001_0001` `sll 2` ⇔ `1_00010_011` # 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 (100)	# Assume R0 (accumulator) has `0b0001_0001` `srl 2` ⇔ `1_00010_100` # 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.

slr = shift left register	R	1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0110)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b0000_0010 (2)` `slr R2 ⇔ 0_0010_0110` # after slr instruction, R0 now holds `0b0100_0100`	Shift leftward the value stored in the accumulator R0. The number of bits shifted is designated by the value in the operand register (e.g. R2). Result will be stored in R0.
srr = shift right register	R	1 bit type (0), 4 bit register (XXXX), 4 bit funct code (0111)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b0000_0010 (2)` `srr R2 ⇔ 0_0010_0111` # after srr instruction, R0 now holds `0b0000_0100`	Shift rightward the value stored in the accumulator R0. The number of bits shifted is designated by the value in the operand register (e.g. R2). Result will be stored in R0.
lw = load word	R	1 bit type (0), 4 bit register (XXXX), 4 bit funct code (1000)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b0001_0000 ⇔ 16` `lw R2 ⇔ 0_0010_1000` # 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 (1001)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b0001_0000 ⇔ 16` `sw R2 ⇔ 0_0010_1001` # 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)
eq = conditional equality	R	1 bit type (0), 4 bit register (XXXX), 4 bit funct code (1010)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b0001_0001` `eq R2` ⇔ `0_0010_1010` # after eq instruction, R0 now holds `0b0000_0001` # For any other value of R2, R0 would hold `0b0000_0000`	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.
slt = set on less than	R	1 bit type (0), 4 bit register (XXXX), 4 bit funct code (1011)	# Assume R0 (accumulator) has `0b0000_0001` # Assume R2 has `0b0001_0001` `slt R2` ⇔ `0_0010_1011` # after slt instruction, R0 now holds `0b0000_0001` # 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 (1100)	# Assume R0 (accumulator) has `0b0000_0001` # Assume R2 has value 28 `br R2` ⇔ `0_0010_1100` # After br instruction, PC will be at 28 # 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	# Assume R2 has value 28	Jump to the target address specified by the

		funct (1101)	`j R2` ⇔ `0_0010_1101` # After j instruction, PC will be at 28	value of the operand register. Difference to br: jump regardless of any other conditions.
seti = set imm+-ediate to accumulator	I	1 bit type (1), 5 bit immediate (XXXXX), 3 bit funct (101)	# Assume R0 (accumulator) has `0b0001_0001` `seti 2` ⇔ `1_00010_101` # after seti instruction, R0 now holds `0b0000_0010`	Assign the immediate to the accumulator R0. Let R0 store that value
set = set to register	R	1 bit type (0), 4 bit register (XXXX), 4 bit funct code (1110)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b1001_0000` `set R2` ⇔ `0_0010_1110` # 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 (1111)	# Assume R0 (accumulator) has `0b0001_0001` # Assume R2 has `0b1001_0000` `la R2` ⇔ `0_0010_1111` # 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 (110)	`halt` ⇔ `1_XXXXX_110` Immediate don't care # After halt instruction, program will terminate	Kill the program
luta = Load LUT to accumulator	I	1 bit type (1), 5 bit immediate (XXXXX), 3 bit funct (111)	`luta 15` ⇔ `1_01111_111` # After luta instruction, accumulator will be load with address at key = 15 in the LUT, which is `0b1110_1000`	Load the accumulator with the designated address in the LUT.

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]

4.1 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.

4.2 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.

4.3 Our ALU will not be used for memory or PC address calculation. The branch target address will be stored in the operand register, and the necessary addresses will be in LUT for fetch. However, we have embedded logics in the ALU to select between `opReg` input and immediate input. In other words, our ALU takes in 3 inputs instead of 2 and has logic within to act as a multiplexer that selects between two of the inputs. This makes the internal logic of our ALU more complicated.

5. Hardware Component Specification

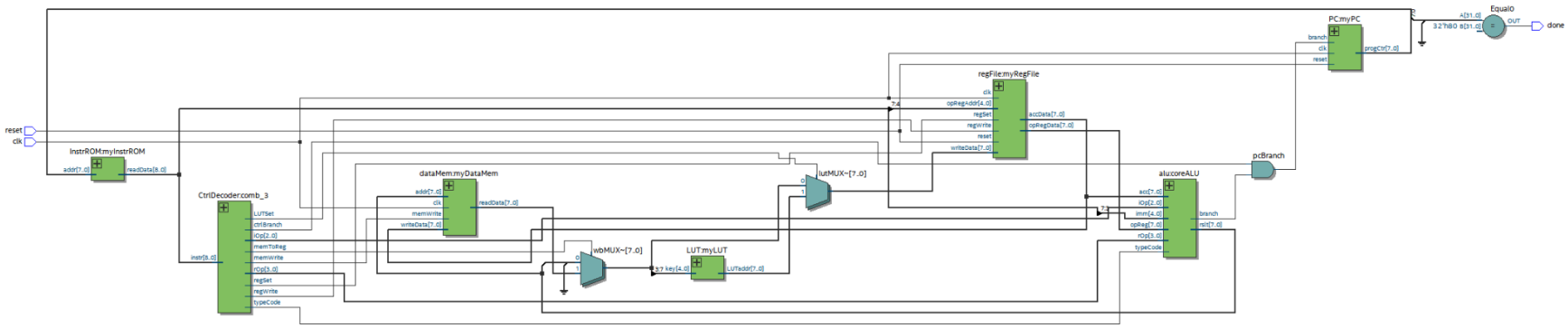
Top Level

Module file name: TopLevel.sv

Functionality Description

The top-level implementation integrates all the hardware components as modules and connect their inputs and outputs using wires and multiplexers. The general idea follows the architectural overview diagram of Section 2 of this report. It takes the clock and reset signal as inputs and outputs done when the PC is 128. In terms of implementation, the output logics are declared and modules are instantiated. The logics are then connected by directing outputs to inputs.

Schematic



Program Counter

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

Functionality Description

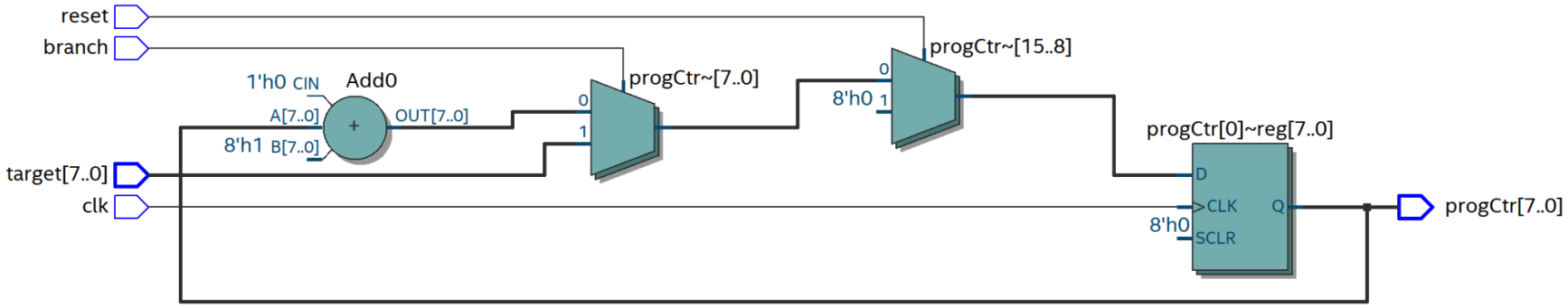
The program counter is a counter for program.The counter will increase the counter value with 1 as the instructions are fetched. That is, when RAM is clocked, do the squential write. When condition is `reset`, reset the counter value to 0, when condition is branch, set the counter value to target(how far/where to jump).

Testbench Description

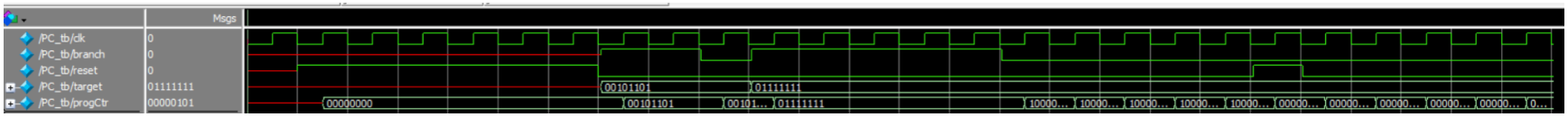
The PC testbench has the following workflow:

1. Reset is set to 1 and see if PC is reset to 0 or not. Then, reset is set to 0 for further testing.
2. Set target to 45 and branch to 1. PC is observed and is expected to jump to 45.
3. Wait for 2 cycles, PC should proceed by 2.
4. Set branch to 0.
5. Set target to 127 and branch to 1, wait for 5 cycles, PC should jump to 127 and proceed by 1 each cycle.
6. Set branch to 0, wait for 5 cycles, PC should continue to proceed by 1 each cycle.
7. Set reset to 1, wait for 1 cycle, PC should be reset to 0.
8. Set reset to 0, wait for 10 cycles, PC shold proceed from 0 by 1 each cycle.

Schematic



Timing Diagram



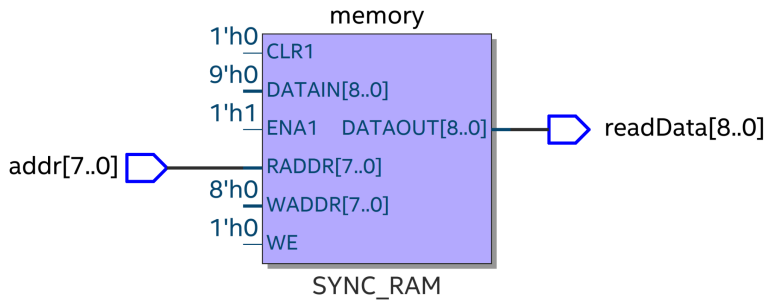
Instruction Memory

Module file name: instrROM.sv
Attached file name: mach_code.txt (dummy file for compile)

Functionality Description

The instruction memory gets the input of the instruction address (`readAddr`) and outputs the 9-bit `instruction`. The size of instruction meomory is 9-bit wide, and can hold a data space of 256-word (byte). It will read the machine code in, and fetch the instruction to feed to other modules (register files and the core ALU) for further operations.

Schematic



Control Decoder

Module file name: CtrlDecoder.sv

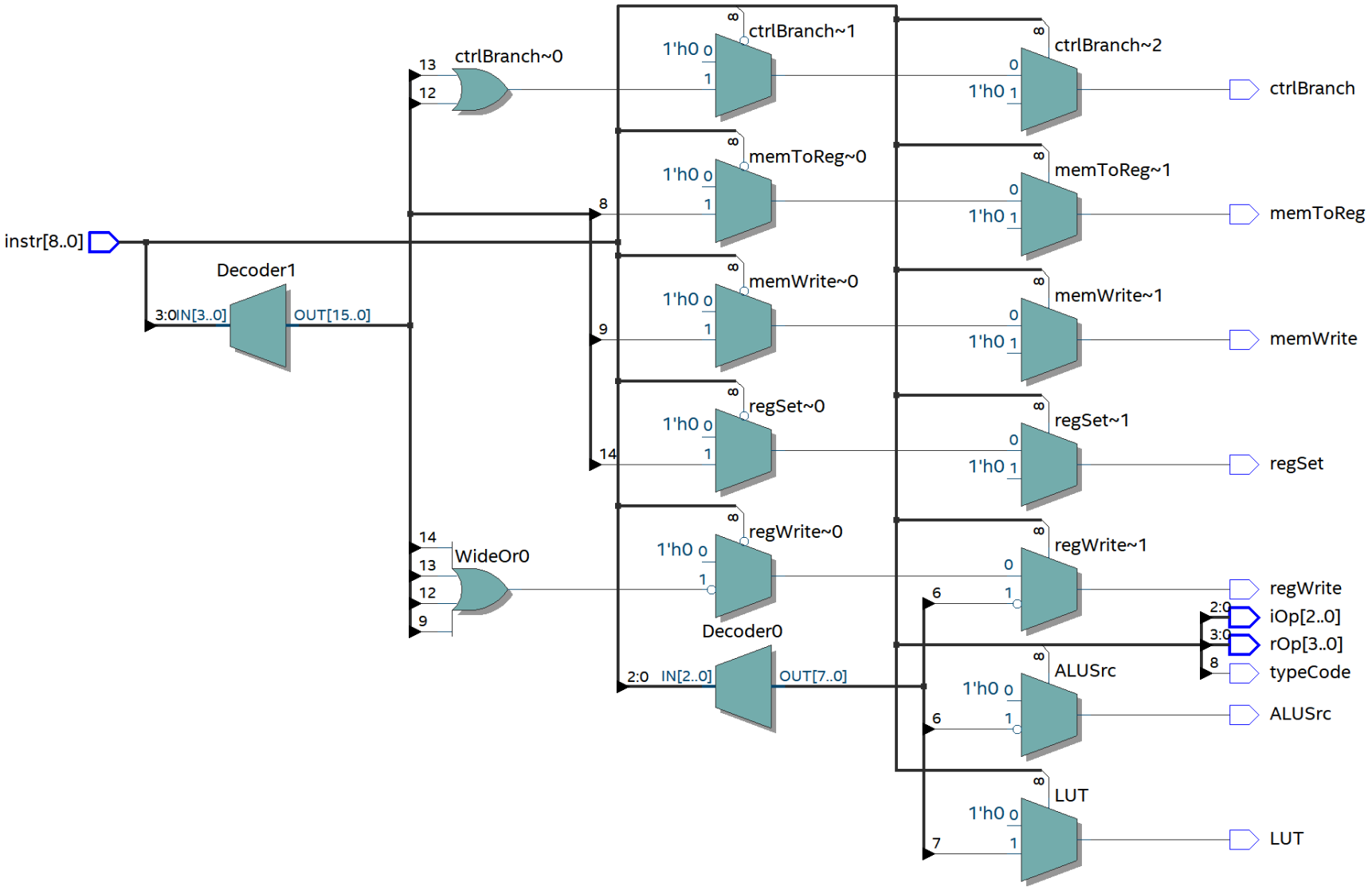
Functionality Description

The Control Decoder takes the instruction as the input and decode it to generate control signals as the output. The signals are:

- `regWrite`: Write enable signal of register files. Set to 1 to allow writing back to the accumulator, otherwise 0;
- `regSet`: Signal for write from accumulator to the operand register. Set to 1 only for SET instruction, otherwise 0;
- `LUTSet`: Signal controlling the MUX for loading LUT address to accumulator. Set to 1 for LUTA instruction, otherwise 0;
- `memWrite`: Write enable signal for data memory. Set to 1 to allow writing into data mem, otherwise 0;
- `ctrlBranch`: ONE OF THE SIGNALS controlling the branch behavior. Generates `PCSrc` together with `ALUBranch` from ALU output. Set to 1 if it is a branch instruction, otherwise 0;
- `memToReg`: Control the MUX to select between dataMem output and ALU output for writing back to the accumulator;
- `typeCode`: Part of ALU signal that tells the ALU whether the operation is R-type or I-type. 0 for R, 1 for I;
- `rOp`: Part of ALU signal that tells the ALU which specific R-type operation to take;
- `iOp`: Part of ALU signal that tells the ALU which specific I-type operation to take.

These output control signal are fed to different modules of the processor to control their behavir in order to perform the desired instruction.

Schematic


instr[8..0]
Decoder1
3:0
IN[3..0]
OUT[15..0]
13
12
ctrlBranch~0
14
13
12
9
WideOr0
8
9
14
2:0
IN[2..0]
Decoder0
OUT[7..0]
ctrlBranch~1
memToReg~0
memWrite~0
regSet~0
regWrite~0
1'h0
0
1
ctrlBranch~2
memToReg~1
memWrite~1
regSet~1
regWrite~1
6
ALUSrc
6
LUT
7
ctrlBranch
memToReg
memWrite
regSet
regWrite
2:0
iOp[2..0]
3:0
rOp[3..0]
8
typeCode
ALUSrc
LUT

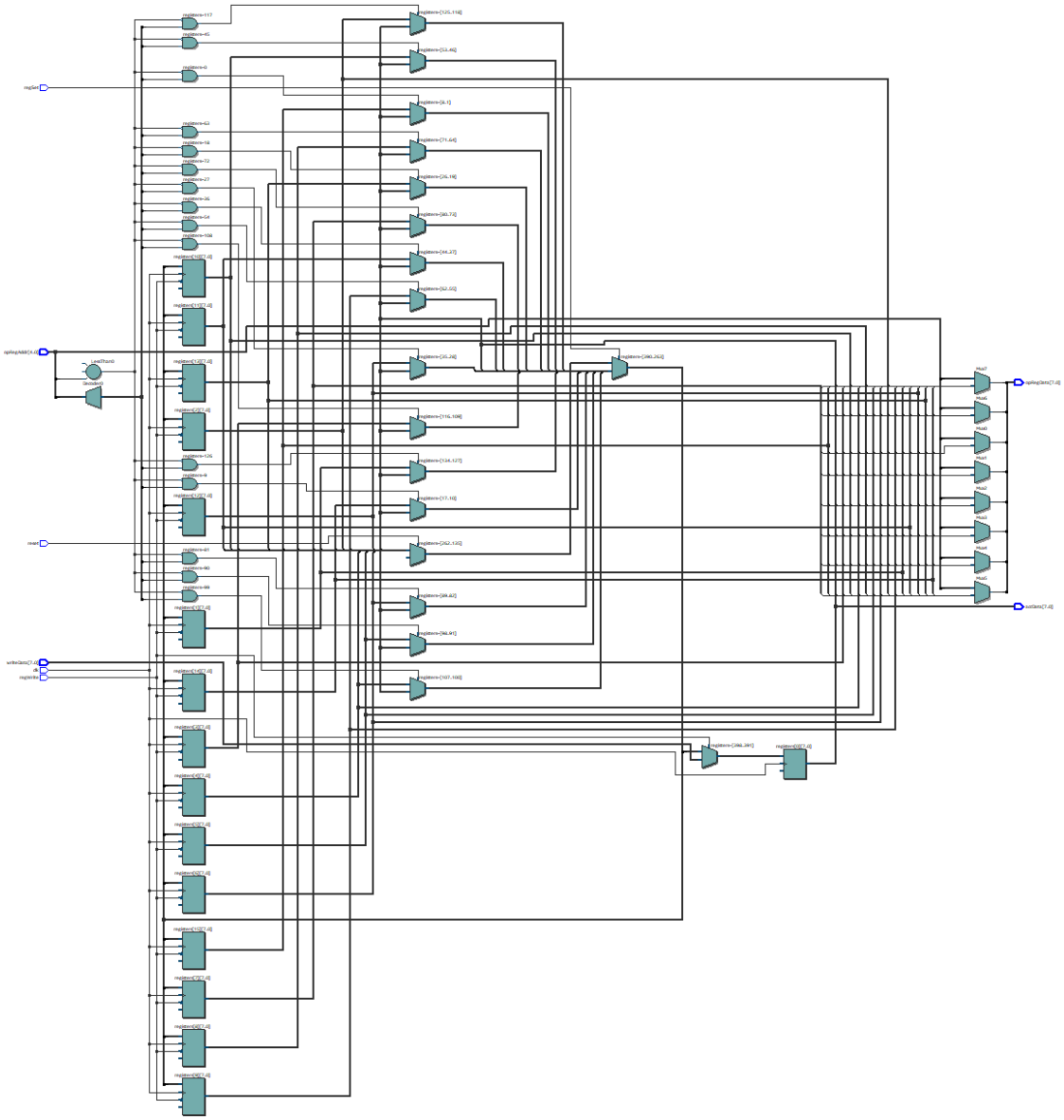
Register File

Module file name: regFile.sv

Functionality Description

The registers will temporarily store the address and value of the instructions. While resiter file will store all these registers' information. The size of register file is 8 bit wide, and 16 byte deep to hold 16 registers. As ARM is clocked, it takes the input of `regWrite, regSet, reset, LUTSet, writeData, opRegAddr`, then output the `accData` and `opRegData`. The register file will calculate the `accData` and `opRegData` with Combinational Read by using register array and oprand register address. Then implement the function of writeback, handle instruction set, and reset with Sequential Write.

Schematic



ALU (Arithmetic Logic Unit)

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

Functionality Description

ALU is the core arithmetic unit of the processor, as it can perform mathematical calculations, logical calculations, and bitwise operations. It takes `acc` and `opReg` as register inputs and `imm` as immediate input. The control signal `typeCode` will clarify the instruction type, `rOp` and `iOp` will specify the instruction to execute. `scIn` is the shift_carry in. The outputs were consists of `rslt`, which is ALU operation results and `scOut` that displays the shift_carry out. There will be a branch signal output which will be combined with the branch signal from control decoder to decide the PC address. When typeCode is equal to 0, the ALU will perform R-type operations (ADD, SUB, AND, OR, XOR, RXOR, SLR, SLR, SRR, LW, SW, EQ, SLT, BR, J, SET, LA), otherwise if the typeCode is 1, the ALU will perform I-type operations (ADDI, SUBI, ANDI, SLL, SRL, SETI, LUTA).

Testbench Description

The PC testbench has the following workflow:

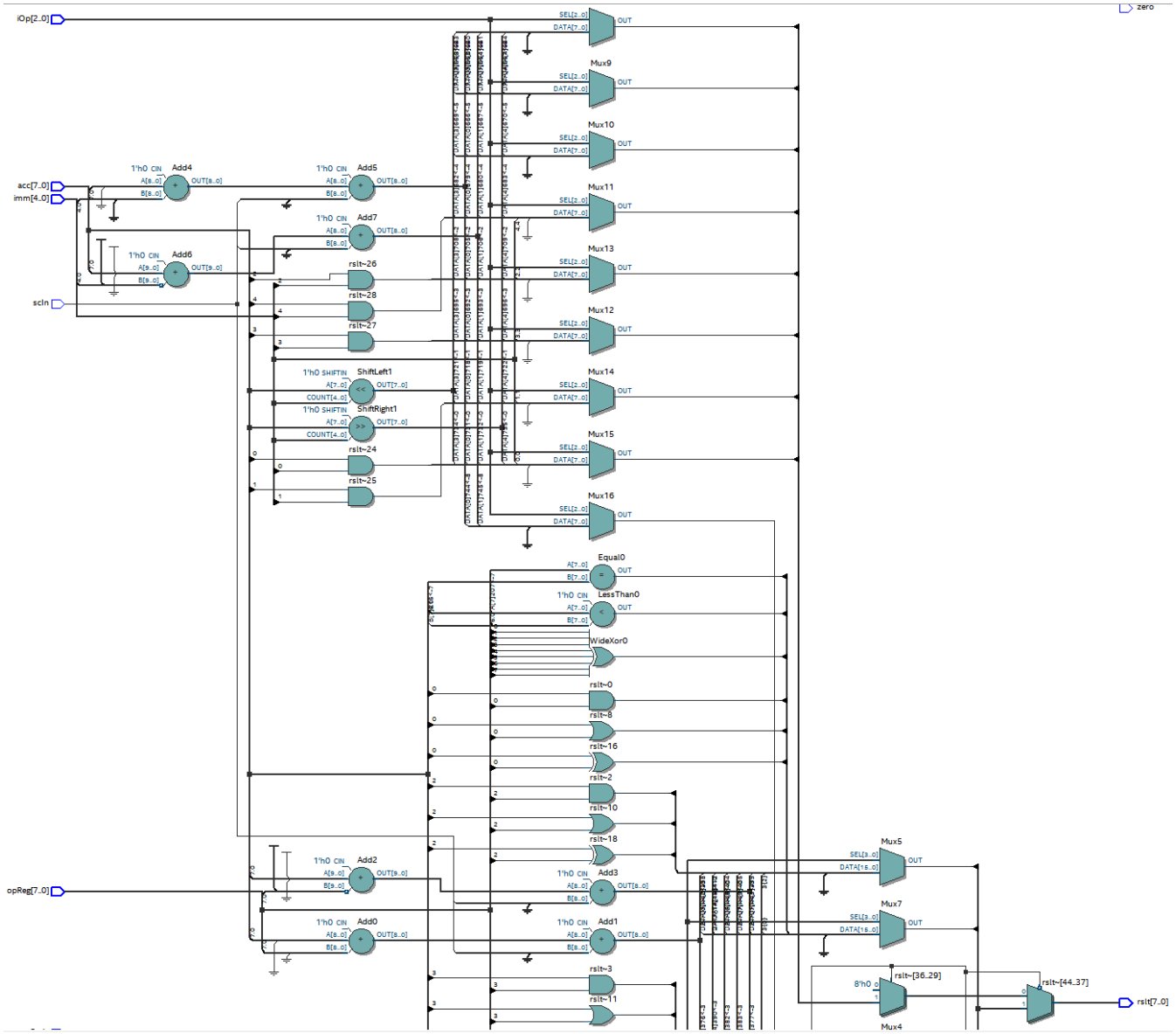
1. `typeCode` is set to 0 for R-type instructions.
2. `rOp` is set to 4'b0000 for ADD instructions. Set `accIn` to 44 and `opIn` to 45. ALU should result in 89.
3. `rOp` is set to 4'b0001 for SUB instructions. Set `accIn` to 45 and `opIn` to 44. ALU should result in 1.
4. `rOp` is set to 4'b0010 for AND instructions. Set `accIn` to 8’b00101100 and `opIn` to 8’b00101101. ALU should result in 8’b00101100.
5. `rOp` is set to 4'b0011 for OR instructions. Set `accIn` to 8’b00101100 and `opIn` to 8’b00101101. ALU should result in 8’b00101101.
6. `rOp` is set to 4'b0100 for XOR instructions. Set `accIn` to 8’b00101100 and `opIn` to 8’b00101101. ALU should result in 8’b00000001.
7. `rOp` is set to 4'b0101 for RXOR instructions. Set `opIn` to 8’b00101101. ALU should result in 8’b00000000.
8. `rOp` is set to 4'b0110 for SLR instructions. Set `accIn` to 8’b00101100 and `opIn` to 2. ALU should result in 8’b10110000.
9. `rOp` is set to 4'b0111 for SRR instructions. Set `accIn` to 8’b00101100 and `opIn` to 2. ALU should result in 8’b00001011.
10. `rOp` is set to 4'b1000 for LW instructions. Set `accIn` to 44 and `opIn` to 127, which is value in `opReg`. ALU should result in 127.

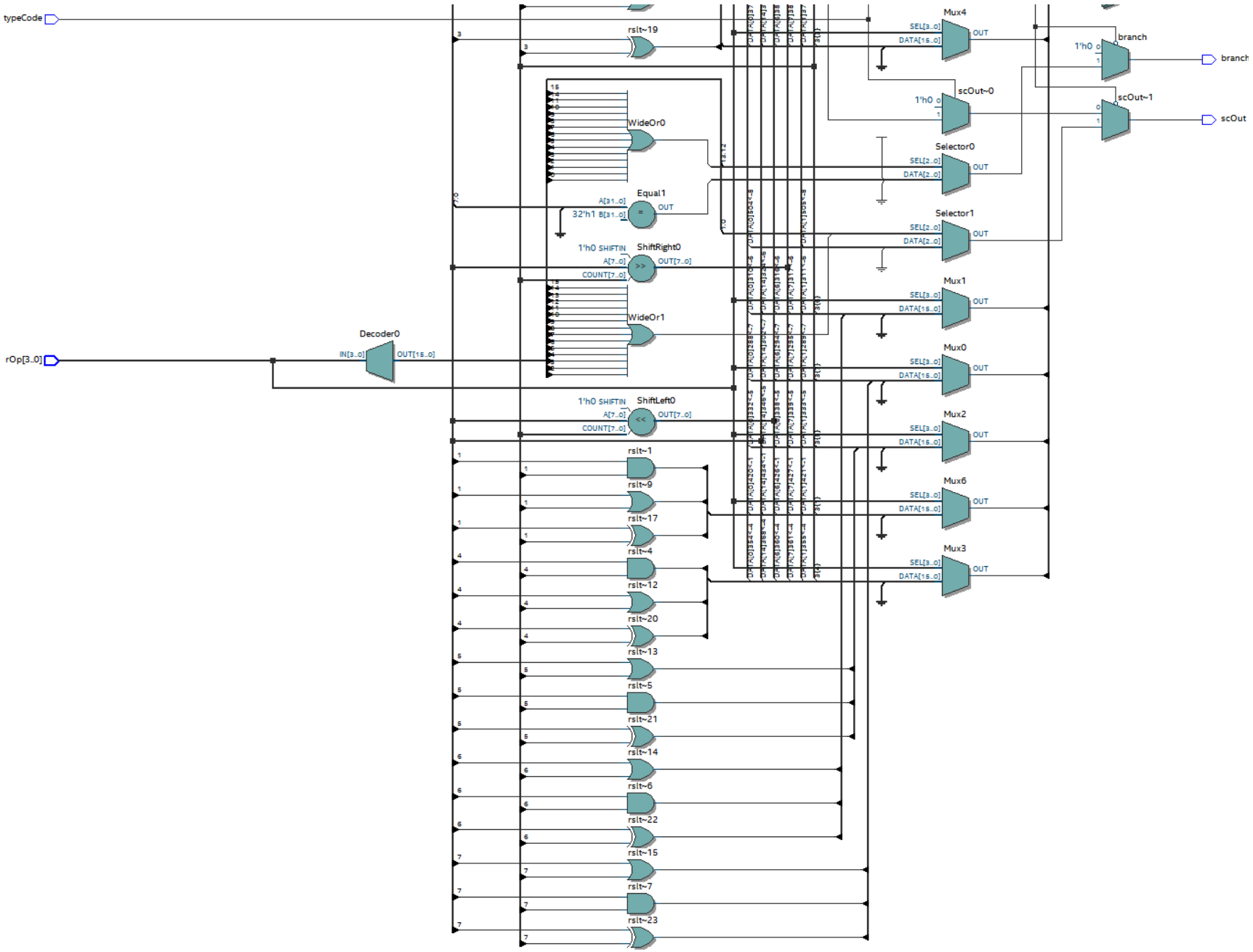
11. `rOp` is set to 4'b1001 for SW instructions. Set `accIn` to 44 and `opIn` to 127, which is value in `opReg`. ALU should result in 127.
12. `rOp` is set to 4'b1010 for EQ instructions. Set `accIn` to 44 and `opIn` to 44. ALU should result in 1.
13. `rOp` is set to 4'b1010 for EQ instructions. Set `accIn` to 0 and `opIn` to 127. ALU should result in 0.
14. `rOp` is set to 4'b1011 for SLT instructions. Set `accIn` to 44 and `opIn` to 45. ALU should result in 1.
15. `rOp` is set to 4'b1011 for SLT instructions. Set `accIn` to 44 and `opIn` to 32. ALU should result in 0.
16. `rOp` is set to 4'b1011 for SLT instructions. Set `accIn` to 44 and `opIn` to 44. ALU should result in 0.
17. `rOp` is set to 4'b1100 for BR instructions. Set `accIn` to 1. ALU should raise `branchFlag` to 1.
18. `rOp` is set to 4'b1100 for BR instructions. Set `accIn` to 0. ALU should lower `branchFlag` to 0.
19. `rOp` is set to 4'b1101 for J instructions. Set `accIn` to 44 and `opIn` to 27. ALU should raise `branchFlag` to 1 for unconditional jump.
20. `rOp` is set to 4'b1110 for SET instructions. Set `accIn` to 44. ALU should result in 44.
21. `rOp` is set to 4'b1111 for LA instructions. Set `opIn` to 45. ALU should result in 45.
22. `typeCode` is set to 1 for I-type instructions.
23. `iOp` is set to 3'b000 for ADDI instructions. Set `accIn` to 44 and `immIn` to 31. ALU should result in 75.
24. `iOp` is set to 3'b001 for SUBI instructions. Set `accIn` to 32 and `immIn` to 31. ALU should result in 1.
25. `iOp` is set to 3'b010 for ANDI instructions. Set `accIn` to 8'b00101100 and `immIn` to 5'b01101. ALU should result in 8'b00001100.
26. `iOp` is set to 3'b011 for SLL instructions. Set `accIn` to 8'b00101100 and `immIn` to 2. ALU should result in 8'b10110000.
27. `iOp` is set to 3'b100 for SRL instructions. Set `accIn` to 8'b00101100 and `immIn` to 2. ALU should result in 8'b00001011.
28. `iOp` is set to 3'b101 for SETI instructions. Set `immIn` to 25. ALU should result in 25.

ALU Operations

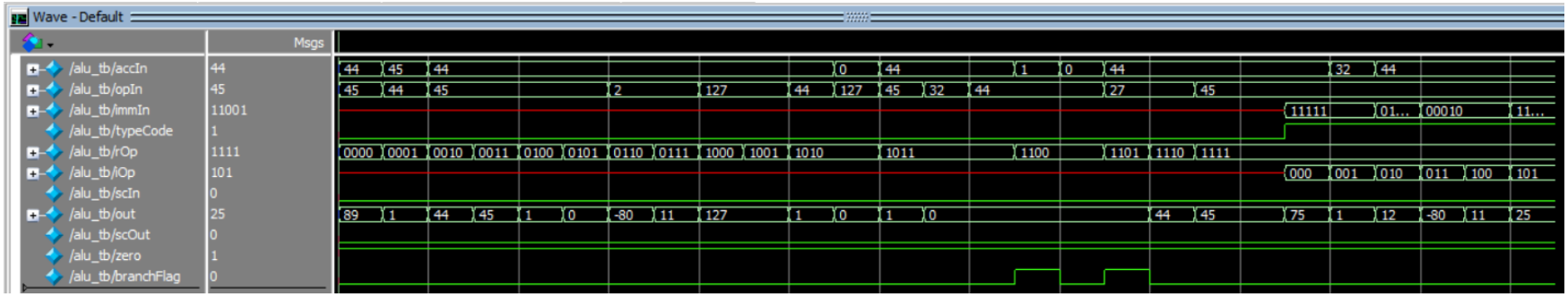
ALU Operation Demonstrated in Testbench	**Corresponding Instruction**
Register addition and subtraction	ADD, SUB
Register bitwise AND	AND
Register bitwise OR	OR
Register bitwise XOR	XOR
Register reduction XOR	RXOR
Register shift left and right	SLR, SRR
Direct value from opReg input to rslt	LW, SW, J, SET, LA
Equality comparison	EQ, BR
Less-than comparison	SLT
Immediate addition and subtraction	ADDI, SUBI
Immediate bitwise AND	ANDI
Immediate shift left and right	SLL, SRL
Direct immediate value to rslt	SETI, LUTA

Schematic


iOp[2..0]
acc[7..0]
imm[4..0]
scIn
opReg[7..0]
zero
rslt[7..0]
Add4
Add5
Add6
Add7
Add0
Add1
Add2
Add3
ShiftLeft1
ShiftRight1
Equal0
LessThan0
WideXor0
Mux4
Mux5
Mux7
Mux9
Mux10
Mux11
Mux12
Mux13
Mux14
Mux15
Mux16
rslt~0
rslt~2
rslt~3
rslt~8
rslt~10
rslt~11
rslt~16
rslt~18
rslt~24
rslt~25
rslt~26
rslt~27
rslt~28
rslt~[36..29]
rslt~[44..37]


typeCode
rOp[3..0]
Decoder0
IN[3..0]
OUT[15..0]
rslt~19
WideOr0
Equal1
A[31..0]
32'h1 B[31..0]
OUT
1'h0 SHIFTIN
ShiftRight0
A[7..0]
COUNT[7..0]
OUT[7..0]
WideOr1
ShiftLeft0
rslt~1
rslt~9
rslt~17
rslt~4
rslt~12
rslt~20
rslt~13
rslt~5
rslt~21
rslt~14
rslt~6
rslt~22
rslt~15
rslt~7
rslt~23
Mux4
Mux1
Mux0
Mux2
Mux6
Mux3
SEL[3..0]
DATA[15..0]
scOut~0
Selector0
Selector1
SEL[2..0]
DATA[2..0]
branch
scOut~1
scOut

Timing Diagram



Wave form segment and corresponding test: ADD SUB AND OR XOR RXOR SHIFT Load/Store Equality Less-than Branch signal test Set LA ADDI SUBI ANDI Imm Shift LUTA

```
VSIM 11> run -all
# R-type operation testing:
# 44 + 45 =   89, 89 expected
# 45 - 44 =    1, 1 expected
# 00101100 AND 00101101 = 00101100, 00101100 expected
# 00101100 OR 00101101 = 00101101, 00101101 expected
# 00101100 XOR 00101101 = 00000001, 00000001 expected
# ^(00101101) = 00000000, 00000000 expected
# 00101100 << 2 = 10110000, 10110000 expected
# 00101100 >> 2 = 00001011, 00001011 expected
# Using value in the opReg for address:  127, 127 expected
# Using value in the opReg for address:  127, 127 expected
# 44 == 44: 00000001, 1 expected
# 0 == 127: 00000000, 0 expected
# 44 < 45: 00000001, 1 expected
# 44 < 32: 00000000, 0 expected
# Edge case 44 < 44: 00000000, 0 expected
# ACC = 1, branch flag: 1, 1 expected
# ACC = 0, branch flag: 0, 0 expected
# Unconditional jump, branch flag: 1, 1 expected
# Set ACC value to reg, output:   44, 44 expected
# Load opReg value to ACC, output:   45, 45 expected
#
#
# I-type operations testing:
# 44 + 31 =   75, 75 expected
# 32 - 31 =    1, 1 expected
# 00101100 AND {000, 01101} = 00001100, 00001100 expected
# 00101100 << 2 = 10110000, 10110000 expected
# 00101100 >> 2 = 00001011, 00001011 expected
# Set immediate input to acc, output:   25, 25 expected
```

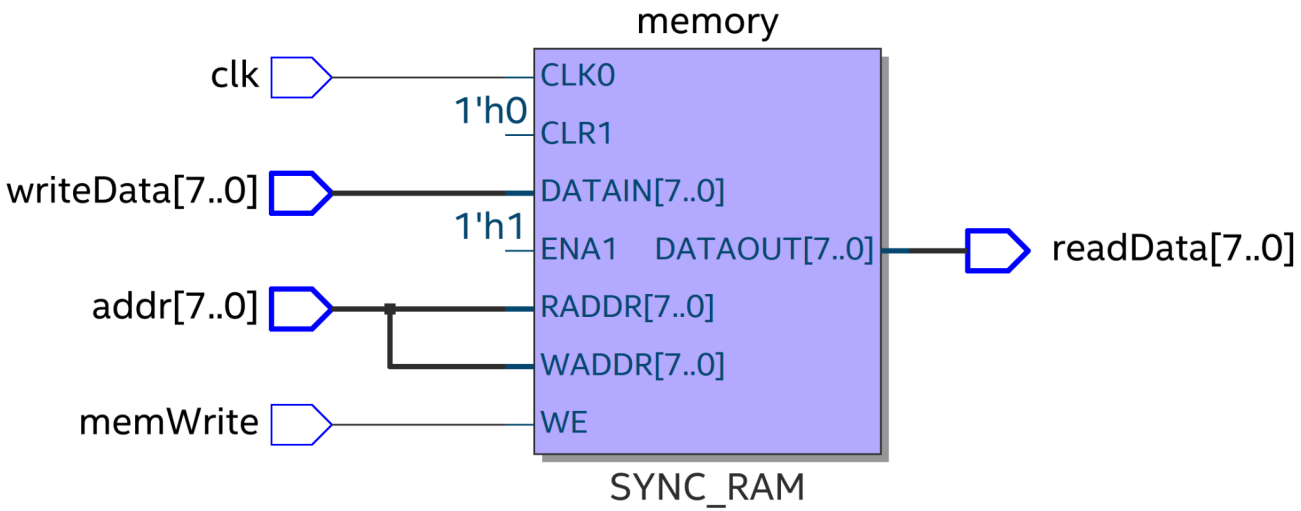
Data Memory

Module file name: dataMem.sv

Functionality Description

Data memory plays the role of storing, reading, and writing the data. This data memory is 8-bit wide, and holds a data space of 256-word (byte). It takes `writeData`, `clk`, `memWrite` and `addr` as inputs. It reads from the data and ouptuts `readData` from the address with combinational read. To write data, we do sequential write at posedge of the clk signal based on `addr` and `writeData`.

Schematic



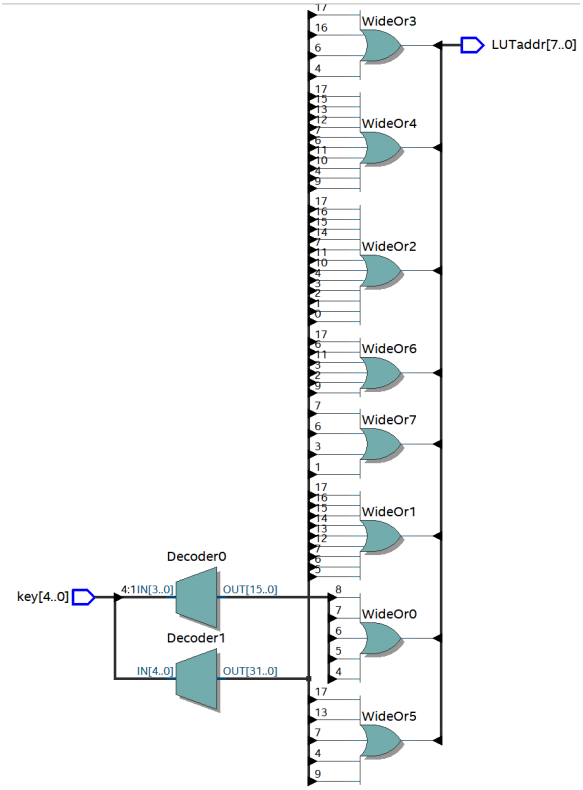
Lookup Tables

Module file name: LUT.sv

Functionality Description

The Lookup Tables is more like a dictionary, that maps the input values with the output values. In our processor, the lookup table takes the inut numbers `key`, and output the target address `LUTaddr`.

Schematic



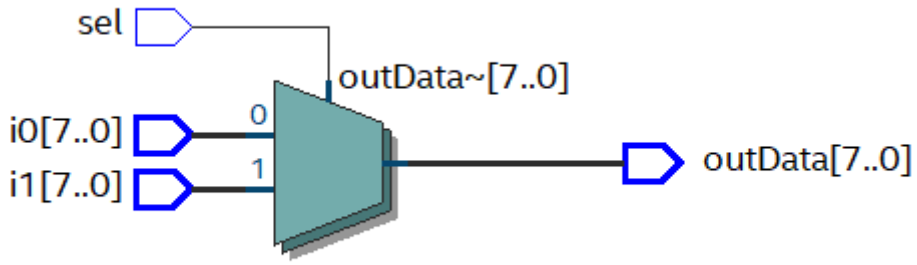
Muxes (Multiplexers)

Module file name: MUX.sv

Functionality Description

This is a 2 to 1 mux, which consists two inputs(i1 and i0), a selector, and an output data. Based on the selector to choose the input, the output Data will be either 1 or 0.

Schematic



6. 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
```

7. Assembler and Machine Code

Assembly Syntax

Assembler overview

1. The programming language we used: Python
2. Usage: Translate assembly code to machine code by the Assembler
3. Implementation Method: op reg imm translate to code
4. Instruction of Using Steps:
    a. Place the assembly code that is going to use for conversion into the folder where the Assembler.py is located
    b. python Assembler.py
    c. Export the output into the file mach_code.txt

Assembly translation syntax

Assembly code format:
R-type: 1-bit type code + 4-bit register + 4-bit op code
I-type: 1-bit bype code + 5-bit immediate + 3-bit op code
The machine code follows the syntax: `type (R or I) + regOrImm + op (e.g. opADD, opXOR, op SLL) + '\n'`

Operation	Type Code	Machine Code
ADD	0	0000
SUB	0	0001
AND	0	0010
OR	0	0011
XOR	0	0100

RXOR	0	0101
SLR	0	0110
SRR	0	0111
LW	0	1000
SW	0	1001
EQ	0	1010
SLT	0	1011
BR	0	1100
J	0	1101
SET	0	1110
LA	0	1111
ADDI	1	000
SUBI	1	001
ANDI	1	010
SLL	1	011
SRL	1	100
SETI	1	101
HALT	1	110
LUTA	1	111

Assembler

Source code in a separate file

Example Input and Output of the Assembler

Example input of assembly code

```
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
```

Example output machine code (of the above assembly code)

```
000101110
100001101
000010000
000011000
000111110
```

Program 1 Machine Code

```
100000101
000011110
000011000
000101110
100001101
000010000
```

000011000
000111110
100001101
011111110
101111101
010011110
100001101
001101110
000011111
001100000
000011110
000000001
010001110
000111111
011110010
010000100
010001110
000111111
100001100
011110010
010000100
010001110
000111111
100010100
011110010
010000100
010001110
000000001
000101111
100100100
011110010
010000100

010001110
000101111
100101100
011110010
010000100
010001110
000101111
100110100
011110010
010000100
010001110
000101111
100111100
011110010
010000100
010001110
000000001
001001110
000111111
011110010
001000100
001001110
000111111
100001100
011110010
001000100
001001110
000111111
100010100
011110010
001000100
001001110

000000001
000101111
100001100
011110010
001000100
001001110
000101111
100010100
011110010
001000100
001001110
000101111
100011100
011110010
001000100
001001110
000101111
100111100
011110010
001000100
001001110
000000001
011001110
000111111
100001100
011110010
011000100
011001110
000111111
100010100
011110010
011000100

011001110
000000001
000101111
011110010
011000100
011001110
000101111
100010100
011110010
011000100
011001110
000101111
100011100
011110010
011000100
011001110
000101111
100101100
011110010
011000100
011001110
000101111
100110100
011110010
011000100
011001110
000000001
010111110
000111111
011110010
010110100
010111110

000111111
100001100
011110010
010110100
010111110
000000001
000101111
011110010
010110100
010111110
000101111
100001100
011110010
010110100
010111110
000101111
100011100
011110010
010110100
010111110
000101111
100100100
011110010
010110100
010111110
000101111
100110100
011110010
010110100
010111110
000000001
010101110

000111111
011110010
010100100
010001110
000111111
100001100
011110010
010100100
010101110
000111111
100010100
011110010
010100100
010101110
000000001
000101111
011110010
010100100
010101110
000101111
100001100
011110010
010100100
010101110
000101111
100010100
011110010
010100100
010101110
000101111
100011100
011110010

010100100
010101110
000101111
100100100
011110010
010100100
010101110
000101111
100101100
011110010
010100100
010101110
000101111
100110100
011110010
010100100
010101110
000101111
100111100
011110010
010100100
010101110
010101111
010000100
001000100
011000100
010110100
010101110
000000001
111111101
000010000
011101110

110000101
001011110
000000001
000111111
100101011
001101110
000101111
001010010
100011100
001100011
010000011
011101001
000000001
111110101
000010000
011011110
000000001
101110101
001011110
000101111
001010010
100100011
001101110
001001111
100100011
001100011
001101110
000101111
011110010
100111011
001111110
011001111

100010011
011001110
010111111
100001011
010111110
001101111
001110011
011000011
010110011
010100011
011011001
000011111
010011011
100000110

Program 2 Machine Code

```
111110101
000011110
000010000
000101110
000011111
000101011
000111110
100001101
000110100
000011000
000111110
100001101
000010000
000001000
001001110
000111111
001000100
001011110
100000101
001101110
100000101
001011010
100001101
001010010
001100000
001011111
100001100
001011110
100010101
001101010
```

100000101
001101110
001101111
001111110
101111101
100100011
010001110
101110101
010000000
000110010
010001110
100000101
001000010
010011110
010001111
010010100
001011110
100000101
001101110
100000101
001011010
100001101
001010010
001100000
001011111
100001100
001011110
100010101
001101010
100000101
001101110
100000101

010101110
100001101
000110010
001100100
100011011
010100000
010101110
101111101
100100011
000110010
010001110
101110101
100100011
001000010
010011110
010001111
010010100
001011110
100000101
001101110
100000101
001011010
100001101
001010010
001100000
001011111
100001100
001011110
100010101
001101010
100000101
001101110

110000101
001000010
100100100
001100100
100010011
010100000
010101110
101100101
100100011
010001110
101100101
010000000
000110010
010001110
101100101
100100011
010011110
101000101
010010000
001000010
010011110
010001111
010010100
001011110
100000101
001101110
100000101
001011010
100001101
001010010
001100000
001011111

100001100
001011110
100010101
001101010
100000101
001101110
100100101
001000010
100010100
001100100
100001011
010100000
010101110
101010101
100100011
010001110
101010101
010000000
000110010
010001110
101010101
100100011
010011110
101000101
010010000
001000010
010011110
010001111
010010100
001011110
100000101
001101110

100000101
001011010
100001101
001010010
001100000
001011111
100001100
001011110
100010101
001101010
100000101
001101110
100010101
001000010
100001100
001100100
010100000
010101110
101110101
100100011
000110010
100101100
010001110
101110101
100100011
010011110
101000101
010010000
001000010
100011100
010011110
100001101

010010010
010111110
010011111
100001100
010110000
010011110
111110101
000110010
100011011
010010000
010011110
100001101
001111010
100100101
100100011
010000000
010001110
100011101
010101010
100101101
010101010
100110101
010101010
100111101
010101010
101001101
010101010
101010101
010101010
101011101
010101010
101100101

010101010
101101101
010101010
101110101
010101010
101111101
010101010
100001101
010010100
010011110
100010101
010010100
010011110
100100101
010010100
010011110
101000101
010010100
010011110
110000101
010010100
010011110
100010101
100100011
010010100
010011110
100100101
100100011
010010100
010011110
101000101
100100011

010010100
010011110
100001101
010000100
010001110
100010101
010000100
010001110
100100101
010000100
010001110
010101010
101000101
100100011
010000000
010001110
111110101
010111110
000011111
010110001
010111110
010001111
010111001
111101101
010111110
000011111
010110001
010111110
010011111
010111001
100010101
000010000

000011110
100000110

Program 3 Machine Code

```
111110101
000011110
000010000
000101110
000011111
000101011
000111110
100001101
000110100
000011000
000111110
100001101
000010000
000001000
001001110
000111111
001000100
001011110
100000101
001101110
100000101
001011010
100001101
001010010
001100000
001011111
100001100
001011110
100010101
001101010
```

100000101
001101110
001101111
001111110
101111101
100100011
010001110
101110101
010000000
000110010
010001110
100000101
001000010
010011110
010001111
010010100
001011110
100000101
001101110
100000101
001011010
100001101
001010010
001100000
001011111
100001100
001011110
100010101
001101010
100000101
001101110
100000101

010101110
100001101
000110010
001100100
100011011
010100000
010101110
101111101
100100011
000110010
010001110
101110101
100100011
001000010
010011110
010001111
010010100
001011110
100000101
001101110
100000101
001011010
100001101
001010010
001100000
001011111
100001100
001011110
100010101
001101010
100000101
001101110

110000101
001000010
100100100
001100100
100010011
010100000
010101110
101100101
100100011
010001110
101100101
010000000
000110010
010001110
101100101
100100011
010011110
101000101
010010000
001000010
010011110
010001111
010010100
001011110
100000101
001101110
100000101
001011010
100001101
001010010
001100000
001011111

100001100
001011110
100010101
001101010
100000101
001101110
100100101
001000010
100010100
001100100
100001011
010100000
010101110
101010101
100100011
010001110
101010101
010000000
000110010
010001110
101010101
100100011
010011110
101000101
010010000
001000010
010011110
010001111
010010100
001011110
100000101
001101110

100000101
001011010
100001101
001010010
001100000
001011111
100001100
001011110
100010101
001101010
100000101
001101110
100010101
001000010
100001100
001100100
010100000
010101110
101110101
100100011
000110010
100101100
010001110
101110101
100100011
010011110
101000101
010010000
001000010
100011100
010011110
100001101

010010010
010111110
010011111
100001100
010110000
010011110
111110101
000110010
100011011
010010000
010011110
100001101
001111010
100100101
100100011
010000000
010001110
100011101
010101010
100101101
010101010
100110101
010101010
100111101
010101010
101001101
010101010
101010101
010101010
101011101
010101010
101100101

010101010
101101101
010101010
101110101
010101010
101111101
010101010
100001101
010010100
010011110
100010101
010010100
010011110
100100101
010010100
010011110
101000101
010010100
010011110
110000101
010010100
010011110
100010101
100100011
010010100
010011110
100100101
100100011
010010100
010011110
101000101
100100011

010010100
010011110
100001101
010000100
010001110
100010101
010000100
010001110
100100101
010000100
010001110
010101010
101000101
100100011
010000000
010001110
111110101
010111110
000011111
010110001
010111110
010001111
010111001
111101101
010111110
000011111
010110001
010111110
010011111
010111001
100010101
000010000

000011110
100000110

Changelog

Milestone 2:

- Modified the Architectural Overview:
  - PC increments by one
  - Control
  - ALU input
  - LUT
- Modified ISA Operations by changing the function code of every instruction and appending the following instructions:
  - slr (R) shift left register
  - srr (R) shift right register
  - rxor (R) reduction xor
  - addi (I) add immediate
  - subi (I) sub immediate
  - andi (I) and immediate
  - luta (I) look up table accumulator
- Added Hardware Component Specification:
  - Top Level: Functionality Description and Schematic
  - Program Counter: Functionality Description, Testbench Description, Schematic, and Timing Diagram
  - Instruction Memory: Functionality Description and Schematic
  - Control Decoder: Functionality Description and Schematic
  - Register File: Functionality Description and Schematic
  - ALU (Arthimetic Logic Unit): Functionality Description, Testbench Description, ALU Operations, Schematic, and Timing Diagram
  - Data Memory: Functionality Description and Schematic
  - Lookup Tables: Functionality Description and Schematic
  - Muxes (Multiplexers): Functionality Description and Schematic

Milestone 3:

- Modified ISA for LUT-related instructions
- Implemented the assembler and performed testings
- Constructed two dictionaries for R-type instructions and I-type instructions
- Designed machine code and assembly code for testing (mach_code.txt and assembly.txt)

- Considered how to handle HALT and BR conditions
- Commented out J and BR instruction part from the original assembly file
- Improved the assembly code part by editing the assembly syntax
- Fixed LUT and LUTA functions
- Architecture diagram modified in Milestone 2