CSE 141L Final Report

Daryl Foo, A16535281; Michael Sun, A16192094; Yuke Liu, A15675903

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.

Daryl Foo
Michael Sun
Yuke Liu

0. Team

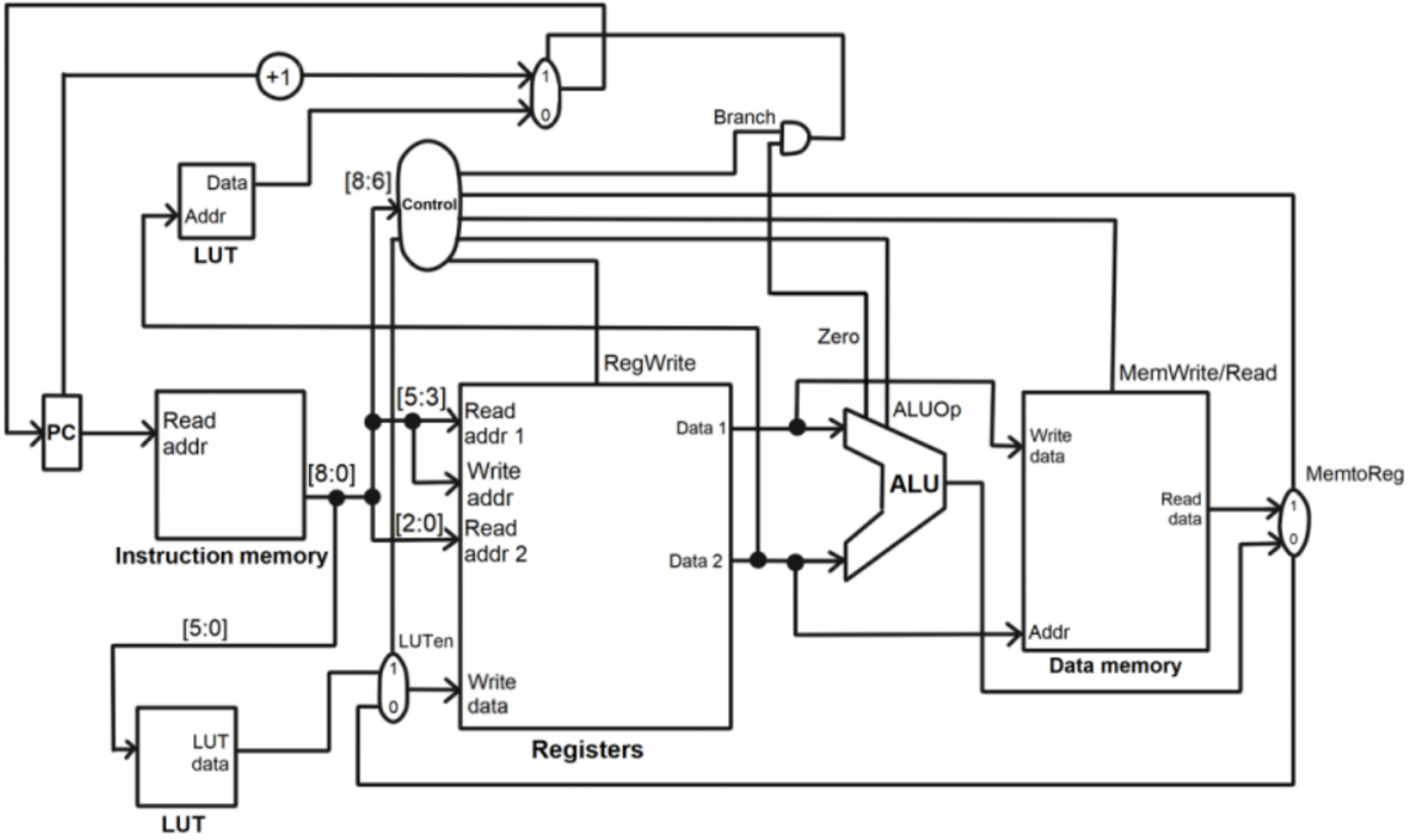
Daryl Foo, Michael Sun, Yuke Liu

1. Introduction

TODO. Name your architecture. What is your overall philosophy? What specific goals did you strive to achieve? Can you classify your machine in any of the standard ways (e.g., stack machine, accumulator, register-register/load-store, register-memory)? If so, which? If not, devise a name for your class of machine. Word limit: 200 words.

Our architecture name is Reg-Reg Accumulator. Our overall philosophy is a balance of hardware and software. The goal we want to achieve is to make the implementation of the hardware and ISA simple (due to the 9 bit wide instruction constraint), but not too constrained such that it makes the programming incredibly tedious. Our machine is a hybrid of accumulator and reg-reg architecture, hence we classify it as a reg-reg accumulator architecture.

2. Architectural Overview



3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	3 bits opcode, 3 bit destination register, 3 bit operand register	add, and, xor, srl, lb, sb
B	3 bit opcode, 3 bit operand register, 3 bit register to store immediate to access lookup table	beq
I	3 bit opcode, 6 bit immediate to access lookup table	lut

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
add = arithmetic add	R	3 bit opcode (000), 3 bit destination register, 3 bit operand register	# Assume R0 has `0b0000_0101` # Assume R1 has `0b0000_0110` `add R0, R1` ⇔ `000_000_001` # after add instruction, R0 now holds `0b0000_1011`	R0 (the destination register) is an implied operand register
and = logical and	R	3 bit opcode (001), 3 bit destination register, 3 bit operand register	# Assume R0 has `0b0101_0100` # Assume R1 has `0b1001_1100`	R0 is the implied destination register.

			`and R0, R1 ⇔ 001_000_001` # after and instruction, R0 now holds `0b0001_0100`	
xor = logical xor	R	3 bit opcode (010), 3 bit destination register, 3 bit operand register	# Assume R0 has `0b0101_0100` # Assume R1 has `0b1001_1100` `xor R0, R1 ⇔ 010_000_001` # after xor instruction, R0 now holds `0b1100_1000`	R0 is the implied destination register.
srl = logical shift right	R	3 bit opcode (011), 3 bit destination register, 3 bit operand register	# Assume R0 has `0b0101_1110` # Assume R1 has `0b0000_0011` `srl R0, R1 ⇔ 011_000_001` # after srl instruction, R0 now holds `0b0000_1011`	R0 is the implied destination register.
lb = load byte	R	3 bit opcode (100), 3 bit destination register, 3 bit operand register	# Assume R0 has `0b0101_1110` # Assume R1 has `0b0000_0100` # Assume that the byte stored at memory address `0b0000_0100` is `0b1111_0000` `lb R0, R1 ⇔ 100_000_001` # after lb instruction, R0 now holds `0b1111_0000`	
sb = store byte	R	3 bit opcode (101), 3 bit operand register, 3 bit destination register	# Assume R0 has `0b0101_1110` # Assume R1 has `0b0000_0100`	

			`sb R0, R1` ⇔ `101_000_001` # after sb instruction, the byte stored at memory address `0b0000_0100` is `0b0101_1110`	
beq = branch equal	B	3 bit opcode (110), 3 bit operand register, 3 bit register to store immediate to access lookup table	# Assume R0 has `0b0000_0001` # Assume R1 has `0b0010_1010` # Assume `LUT[0b0010_1010]` stores the address `0x35` `beq R0, R1` ⇔ `110_000_001` # after beq instruction, PC now stores `0x35`	PC is the implied destination. If the operand register has a nonzero value, the branch will be taken, else it will not.
lut = access lookup table	I	3 bit opcode (111), 6 bit immediate to access lookup table	# Assume `LUTI[27]` stores the value `263` `LUT #27` ⇔ `111_011011` # after lut instruction, R7 now stores `263`	R7 is an implied immediate register. Immediates will always be loaded into R7.

Internal Operands

TODO. How many registers are supported? Is there anything special about any of the registers (e.g. constant, accumulator), or all of them general purpose?

8 registers are supported. We have one special register (R7), which is the register where immediates will always be loaded into. The other 7 registers are general purpose registers. All registers start with the value 0.

We have 2 lookup tables, each of which store 32 8-bit wide entries. One lookup table, LUT, accessed by the beq instruction will store the addresses used for branches, while the other lookup table, LUTI, accessed by the LUT instruction stores the immediates.

Control Flow (branches)

TODO. What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported? How do you accommodate large jumps?

We are doing conditional absolute branching. If the value in the operand register is nonzero, we take the branch, if it is zero, we do not take the branch. Since we are using absolute branching, the target addresses do not have to be calculated, but are instead hard coded into the lookup table. The maximum branch distance is the maximum program count. We do not have to worry about large jumps since we are using absolute branching, so we can just hard code addresses into the lookup table.

Addressing Modes

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

Our addressing mode is indirect. The addresses are obtained from the lookup table(s).

Example 1: we want to load a byte from address `0x64` into r0.

1) lut #0 // Assuming `0x64` is stored at `LUT[0]` in the lookup table, r7 will now store the value `0x64`
2) lb r0, r7 // Load a byte from address `0x64` into r0

Example 2: we want to store a byte from r0 into address `0x64`.

1) lut #0 // Assuming `0x64` is stored at `LUT[0]` in the lookup table, r7 will now store the value `0x64`
2) sb r0, r7 // Store a byte from r0 into address `0x64`

4. Programmer's Model [Lite]

TODO. 4.1 How should a programmer think about how your machine operates? Provide a description of the general strategy a programmer should use to write programs with your machine. For example, one could say that the programmer should prioritize loading in the necessary values from memory into as many registers as possible, then perform calculations. Another approach could be loading and writing to memory in between every calculation step. Word limit: 200 words.

The programmer should prioritize loading in the necessary values from memory into as many registers as possible because our machine has 7 registers (excluding the special immediate register r7) to work with. Since we only have 7 general purpose registers, we allow for intermediate storing to and loading from memory, in the case that more than 7 variables have to be used at once. If the programmer wants to use the bitwise and operation or the bitwise or operation, the “and” and “xor” gate can be used to form these operations because “and” and “xor” together form a universal gate.

TODO. 4.2 Can we copy the instructions/operation from MIPS or ARM ISA? If no, explain why not? How did you overcome this or how do you deal with this in your current design? Word limit: 100 words.

Some instructions from MIPS or ARM ISA are adapted into our ISA. However, we cannot directly copy the instructions because our instructions are 9 bits wide, whereas MIPS or ARM instructions are 32 bits wide. Since we only have 9 bits to work with, we are limited on the number of operands we can put into our instruction set. We overcome this by getting rid of one operand and making the result from operations to be always stored in the same operand. (e.g. X = X + Y), hence the “accumulator” part of our reg-reg accumulator architecture.

TODO. 4.3 Will your ALU be used for non-arithmetic instructions (e.g., MIPS or ARM-like memory address pointer calculations, PC relative branch computations, etc.)? If so, how does that complicate your design?

The main ALU is only used for arithmetic instructions. Since we use absolute branching, we do not need to use the ALU for relative branch computation. However, other simple arithmetic such as adding 1 to our program counter and xor 0 for our branch instructions, but we do not consider those as our main ALU.

5. Individual Component Specification

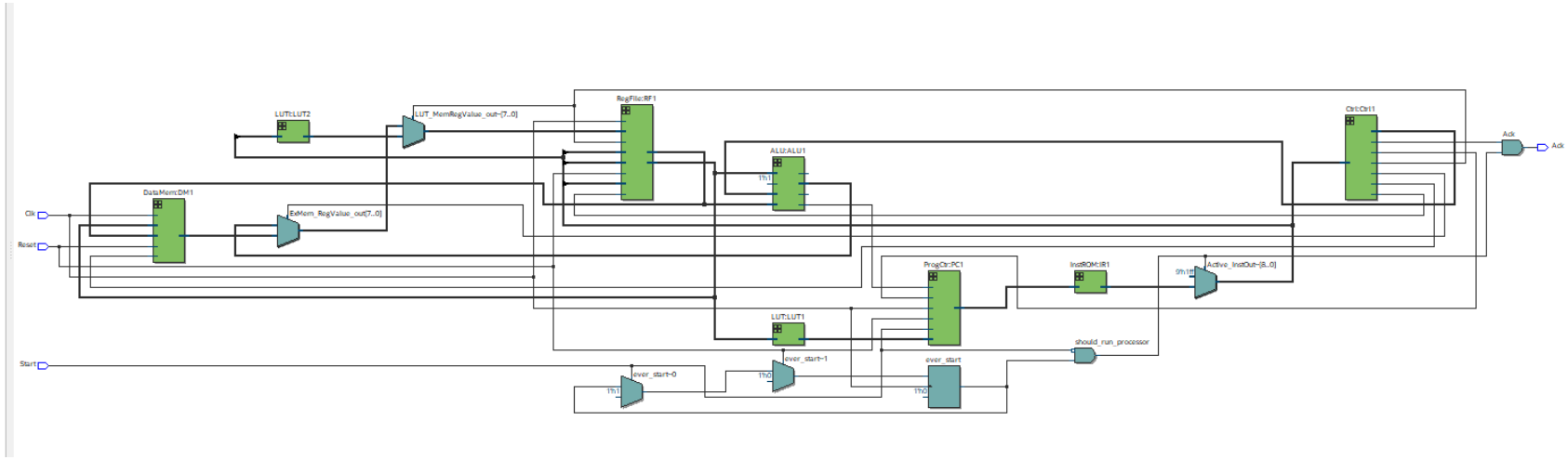
Top Level

Module file name: TopLevel.sv

Functionality Description

TODO. Write a brief description of the functionality of this module.
Declares all the logic and wire inputs and outputs, as well as initializing and hooking up all the modules necessary.

Schematic



Program Counter

Module file name: ProgCtr.sv
Module testbench file name: ProgCtr_tb.sv

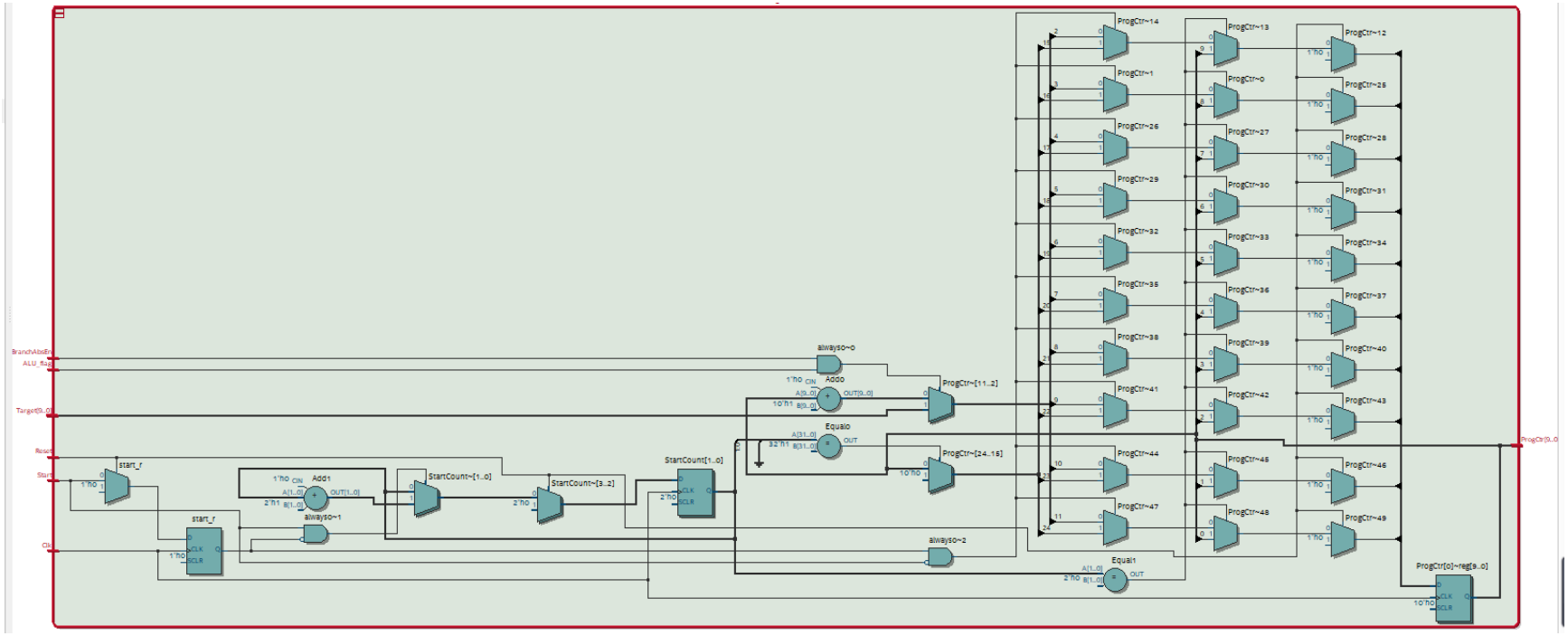
Functionality Description

TODO. Write a brief description of the functionality of this module.
Updates the program counter depending on whether we want to reset, increment, or jump.

Testbench Description

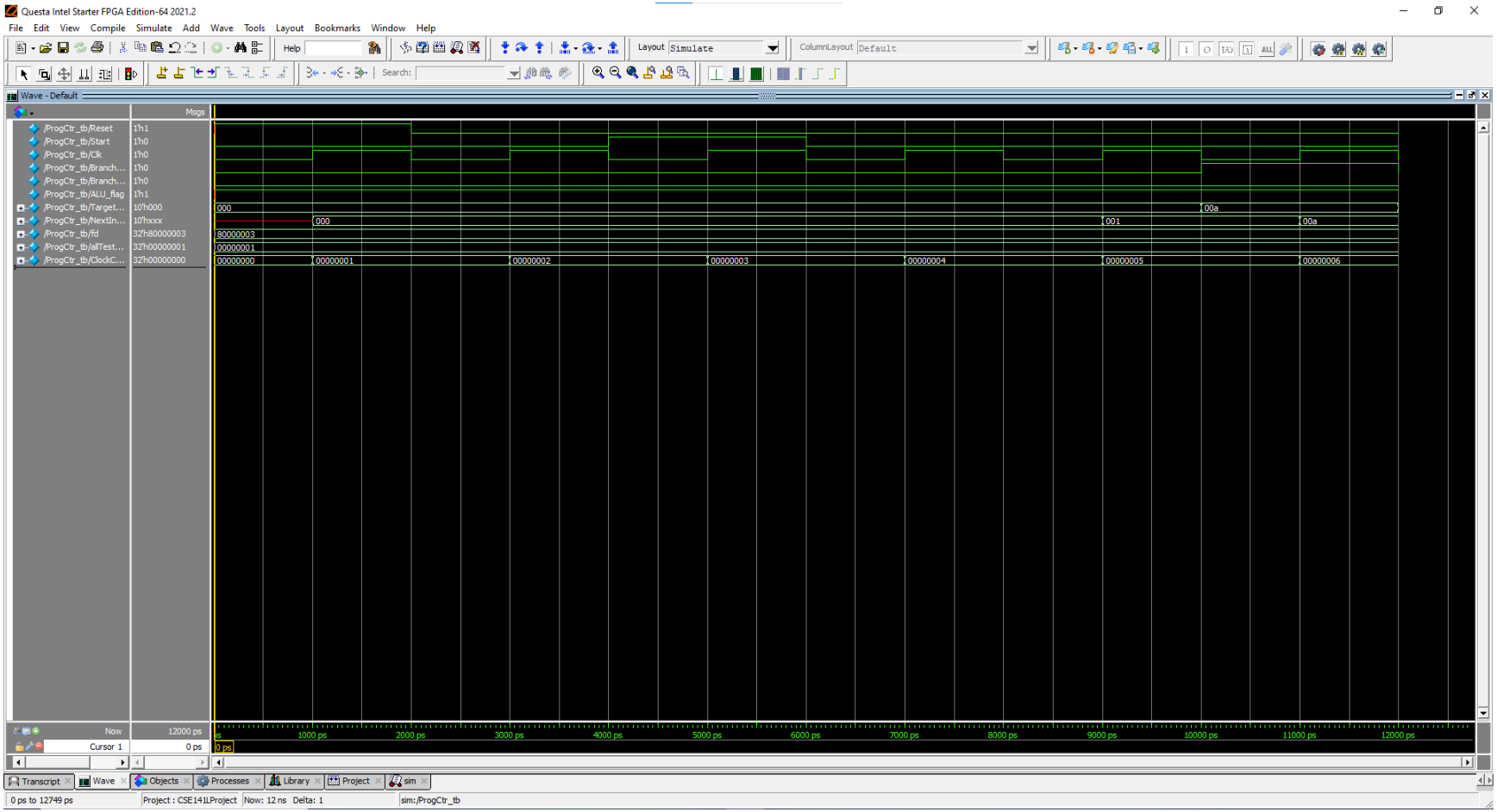
TODO. Describe your testbench. How does it work? What test cases does it test?
The testbench sets the relevant flags and advances time, then checks whether the next instruction index is what is to be expected. The test cases that it tests are reset, increment, and branch, as well as ensuring that the next instruction index is not changed when it is not supposed to be.

Schematic



Timing Diagram

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


Questa Intel Starter FPGA Edition-64 2021.2
File Edit View Compile Simulate Add Wave Tools Layout Bookmarks Window Help
Wave - Default
Msgs
/ProgCtr_tb/Reset
/ProgCtr_tb/Start
/ProgCtr_tb/Clk
/ProgCtr_tb/Branch...
/ProgCtr_tb/Branch...
/ProgCtr_tb/ALU_flag
/ProgCtr_tb/Target...
/ProgCtr_tb/NextIn...
/ProgCtr_tb/fd
/ProgCtr_tb/allTest...
/ProgCtr_tb/ClockC...
1'h1
1'h0
1'h0
1'h0
1'h0
1'h1
10'h000
10'hxxx
32'h80000003
32'h00000001
32'h00000000
000
00a
001
00a
80000003
00000001
00000000
00000001
00000002
00000003
00000004
00000005
00000006
Now
12000 ps
Cursor 1
0 ps
1000 ps
2000 ps
3000 ps
4000 ps
5000 ps
6000 ps
7000 ps
8000 ps
9000 ps
10000 ps
11000 ps
12000 ps
Transcript
Wave
Objects
Processes
Library
Project
sim
0 ps to 12749 ps
Project : CSE141LProject
Now: 12 ns Delta: 1
sim:/ProgCtr_tb

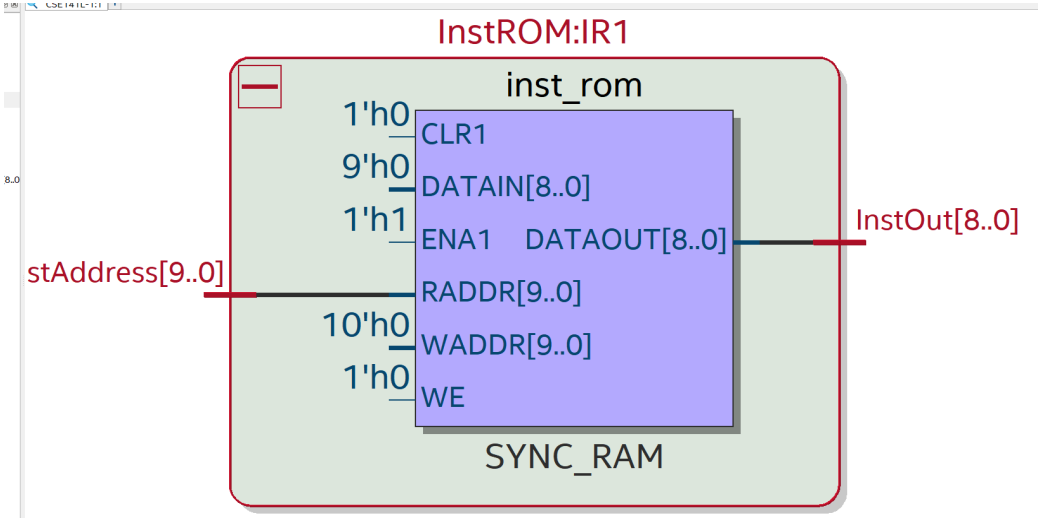
Instruction Memory

Module file name: InstROM.sv

Functionality Description

TODO. Write a brief description of the functionality of this module.
Creates an instruction memory and populates it using an external file.

Schematic



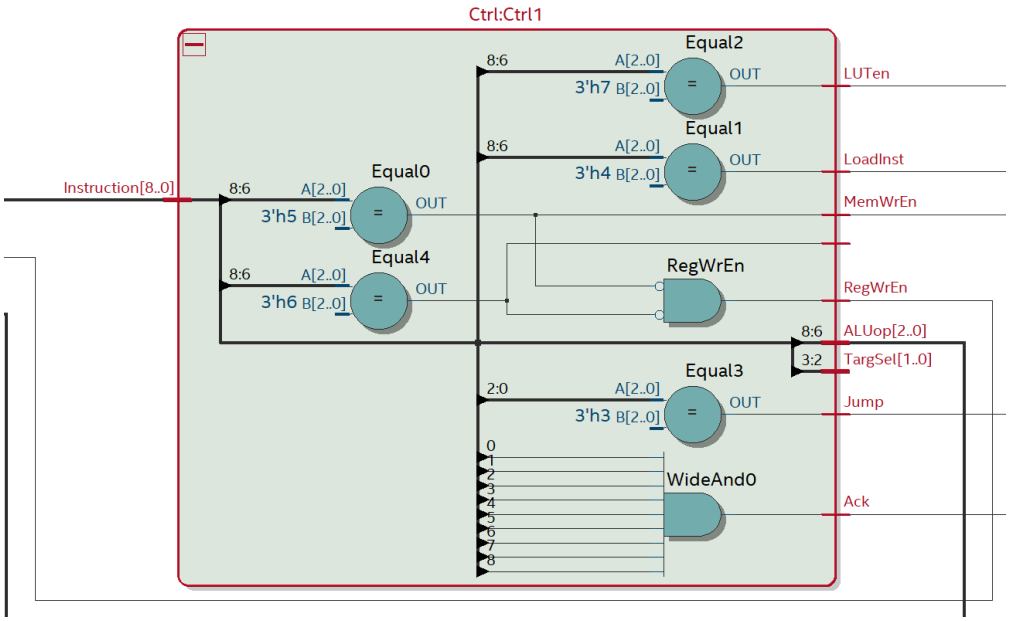
Control Decoder

Module file name: Ctrl.sv

Functionality Description

TODO. Write a brief description of the functionality of this module.
Decodes and initializes control signals used for the architecture.

Schematic



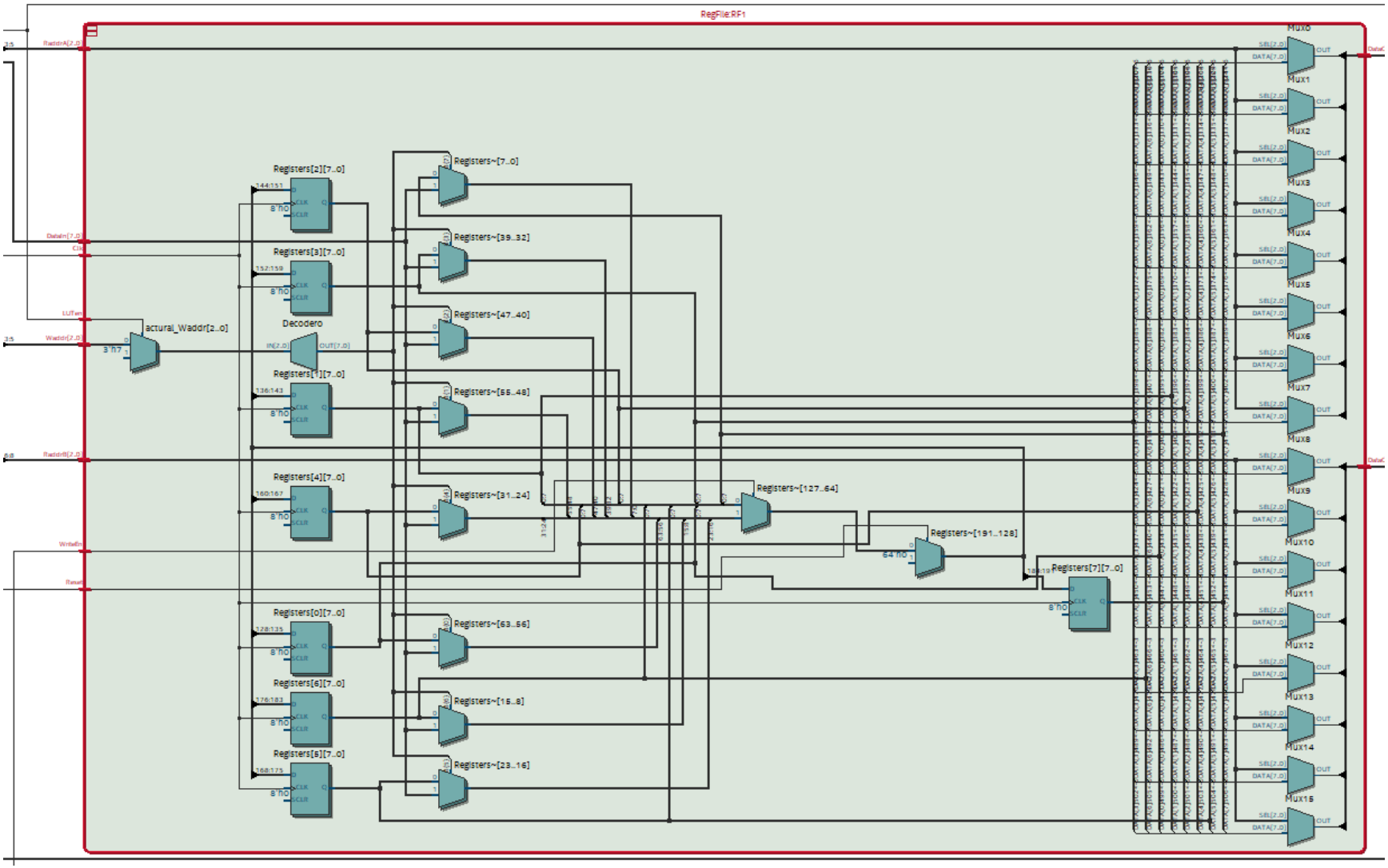
Register File

Module file name: RegFile.sv

Functionality Description

TODO. Write a brief description of the functionality of this module.
Defines a register module that takes in and stores data like a register.

Schematic


RegFile:RF1
Registers[2][7..0]
Registers[3][7..0]
Registers[1][7..0]
Registers[4][7..0]
Registers[0][7..0]
Registers[6][7..0]
Registers[5][7..0]
Registers[7][7..0]
Registers~[7..0]
Registers~[39..32]
Registers~[47..40]
Registers~[55..48]
Registers~[31..24]
Registers~[63..56]
Registers~[15..8]
Registers~[23..16]
Registers~[127..64]
Registers~[191..128]
actural_Waddr[2..0]
Decodero
Mux0
Mux1
Mux2
Mux3
Mux4
Mux5
Mux6
Mux7
Mux8
Mux9
Mux10
Mux11
Mux12
Mux13
Mux14
Mux15

ALU (Arithmetic Logic Unit)

Module file name: ALU.sv
Module testbench file name: ALU_tb.sv

Functionality Description

TODO. Write a brief description of the functionality of this module.
Takes in two input data and outputs, does the appropriate arithmetic, and outputs the result.

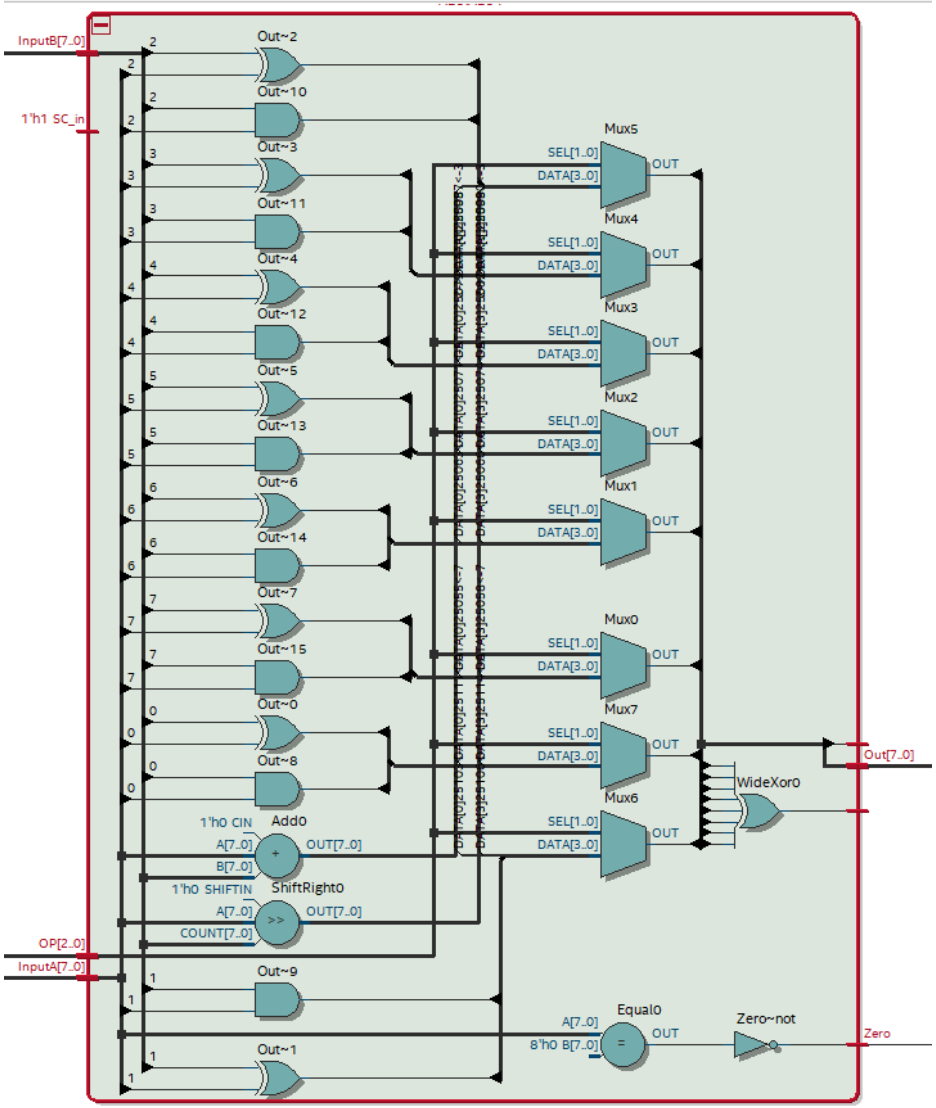
Testbench Description

TODO. Describe your testbench. How does it work? What test cases does it test?
The testbench takes in 2 inputs and produces the corresponding arithmetic output for each of the 4 (ADD, AND, XOR, SRL) ALU operations.

ALU Operations

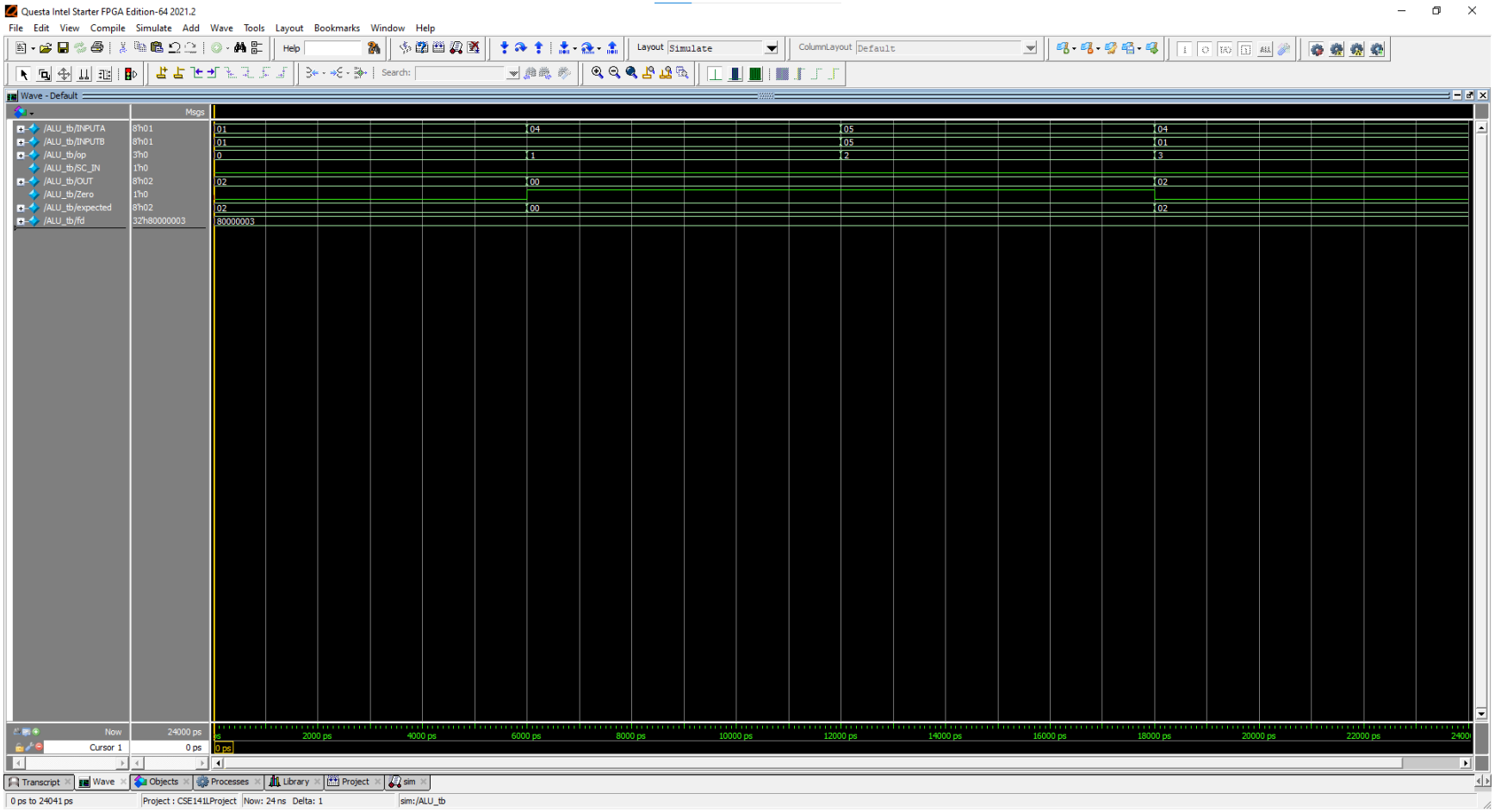
TODO. What ALU operations will you be demonstrating? What instructions are they relevant to?
ADD (add), AND (logical and), XOR (logical xor), SRL (logical shift right).

Schematic


InputB[7..0]
1'h1 SC_in
OP[2..0]
InputA[7..0]
Out~2
Out~10
Out~3
Out~11
Out~4
Out~12
Out~5
Out~13
Out~6
Out~14
Out~7
Out~15
Out~0
Out~8
Out~9
Out~1
Mux5
Mux4
Mux3
Mux2
Mux1
Mux0
Mux7
Mux6
SEL[1..0]
DATA[3..0]
OUT
1'h0 CIN
Add0
A[7..0]
B[7..0]
OUT[7..0]
1'h0 SHIFTIN
ShiftRight0
COUNT[7..0]
WideXor0
Out[7..0]
Equal0
8'h0 B[7..0]
Zero~not
Zero

Timing Diagram

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



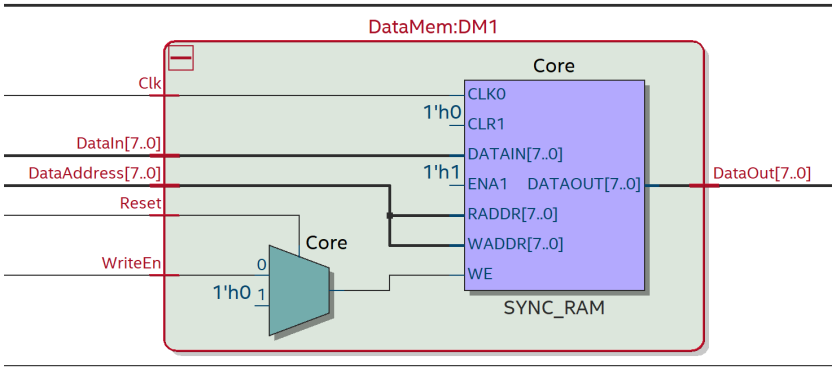
Data Memory

Module file name: DataMem.sv

Functionality Description

TODO. Write a brief description of the functionality of this module.
Initializes memory, writes and reads data to and from memory.

Schematic



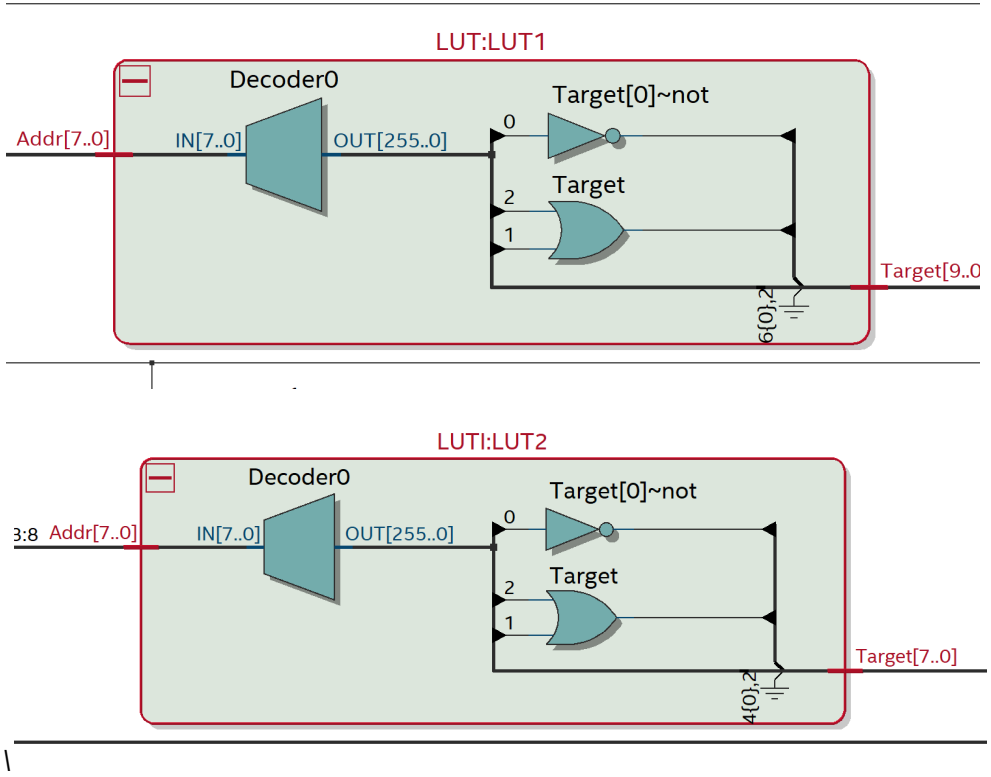
Lookup Tables

Module file name: LUT.sv

Functionality Description

TODO. Write a brief description of the functionality of this module.
Initializes immediate and address values in our lookup table.

Schematic



6. Assembler

TODO. Explain your assembler. What language is it written in? Describe what it does. Provide a usage description. If you were to give your assembler to someone else, what would be the exact steps to generate machine code from your assembly? Give us an example of input to and output from your assembler. If your assembler does anything beyond what a 'normal' assembler would be expected to do, explain this as well.

The assembler is written in Python. The assembler converts given assembly code into corresponding machine code.

For Windows
1. Install Python 3.10.4 from https://www.python.org/downloads/release/python-3104/
2. Navigate to the directory that contains "assembler.py"
3. Create/copy the assembly file you want to convert and name it "assembly.txt", and ensure it is in the same directory as "assembler.py"
4. From there, run "python3 assembler.py" on the terminal
5. The corresponding machine code "machinecode.txt" will be generated

Example input assembly code:

```
xor r0 r0  // r0 = r0 ^ r0 = 0
xor r1 r1  // r1 = r1 ^ r1 = 0

lut #20    // r7 = 1
add r1 r7  // r1 = r1 + r7 = 0 + 1 = 1

// Assume mem[0] = 01011010 in 0x0
lut #0     // r7 = 0x0 = &mem[0]
lb  r2 r7  // r2 = mem[0] = 01011010

add r2 r1  // r2 = r2 + r1 = 01011010 + 00000001 = 01011011
xor r2 r1  // r2 = r2 ^ r1 = 01011011 ^ 00000001 = 01011010
srl r2 r1  // r2 = r2 >> r1 = 01011010 >> 1 = 00101101
and r2 r1  // r2 = r2 & r1 = 00101101 & 00000001 = 00000001
```

```
// Assume mem[1] = 01011010 in 0x1
lut #1     // r7 = 0x1 = &mem[1]
sb  r2 r7  // mem[1] = r2
```

Corresponding output machine code:

```
010000000
010001001
111010100
000001111
111000000
100010111
000010001
010010001
011010001
001010001
111000001
101010111
```

7. Program Implementation

Program 1 Pseudocode

```
# Given the input MSW_in and LSW_in, we return MSW_out and LSW_out

encode(MSW_in, LSW_in):
  p8 = ^MSW_in[2:0] ^ ^LSW_in[7:4]                              // ^(b11:b5)
  p4 = ^MSW_in[2:0] ^ LSW_in[7] ^ ^LSW_in[3:1]                  // ^(b11:b8, b4, b3, b2)
  p2 = ^MSW_in[2:1] ^ ^LSW_in[6:5] ^ ^LSW_in[3:2] ^ LSW_in[0]   // ^(b11, b10, b7, b6, b4, b3,
b1)
  p1 = MSW_in[2] ^ MSW_in[0] ^ LSW_in[6] ^ ^LSW_in[4:3] ^ ^LSW_in[1:0]     // ^(b11, b9, b7, b5,
b4,
                                                                          b2, b1)
  p0 = ^MSW_in[2:0] ^ ^LSW_in[7:0] ^ p8 ^ p4 ^ p2 ^ p1              // ^(b11:1, p8, p4, p2, p1)

  MSW_out = { MSW_in[2:0], LSW_in[7:4], p8 }                    // b11:b5, p8
  LSW_out = { LSW_in[3:1], p4, LSW_in[0], p2, p1, p0 }              // b4:b2, p4, b1, p2:p1, p0

  return MSW_out, LSW_out

test_bench():
  for count in range(0, 30, 2):
    mem[count + 31], mem[count + 30] = encode(mem[count + 1], mem[count])
```

Program 1 Assembly Code

Assume mem[0] is at address 0x0, mem[1] at address 0x1, …, mem[30] at address 0x1E, mem[31] at address 0x1F, …
Assume code starts at address 0x000

LUT[0] = 0x001

LUTI[0] stores 2^0 = 1 = 0b0000 0001
LUTI[1] stores 2^1 = 2 = 0b0000 0010
LUTI[2] stores 2^2 = 4 = 0b0000 0100
LUTI[3] stores 2^3 = 8 = 0b0000 1000
LUTI[4] stores 2^4 = 16 = 0b0001 0000
LUTI[5] stores 2^5 = 32 = 0b0010 0000
LUTI[6] stores 2^6 = 64 = 0b0100 0000
LUTI[7] stores 2^7 = 128 = 0b1000 0000
LUTI[8] stores 0 = 0b0000 0000
LUTI[9] stores 30 = 0b0001 1110

```
xor r0 r0  // r0 = r0 ^ r0 = 0 (count = 0), address 0x000

LOOP:      // Assume label LOOP is at address 0x001
xor r1 r1  // r1 = r1 ^ r1 = 0
xor r2 r2  // r2 = r2 ^ r2 = 0
xor r3 r3  // r3 = r3 ^ r3 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r6 r6  // r6 = r6 ^ r6 = 0
```

```
lut #0     // r7 = 1
add r1 r7  // r1 = 1
lut #8     // r7 = 0x0 = &mem[0]
add r4 r7  // r4 = r4 + r7 = 0 + &mem[0] = &mem[0]
add r4 r0  // r4 = r4 + r0 = &mem[0] + count = &mem[0 + count]
lb r2 r4   // r2 = mem[0 + count] = LSW

lut #2     // r7 = 4
add r5 r7  // r5 = r5 + r7 = 0 + r7 = 4

// Calculating p8
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 4 = LSW >> 4
and r4 r1  // r4 = r4 & 1 = LSW[4]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = LSW[4]

add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = LSW >> 5
and r4 r1  // r4 = r4 & 1 = LSW[5]
xor r6 r4  // r6 = r6 ^  r4 = ^LSW[5:4]

add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = LSW >> 6
and r4 r1  // r4 = r4 & 1 = LSW[6]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[6:4]

add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
```

```
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = LSW >> 7
and r4 r1  // r4 = r4 & 1 = LSW[7]
xor r6 r4  // r6 = r6 ^  r4 = ^LSW[7:4]

lut #8     // r7 = 0x0 = &mem[0]
add r3 r1  // r3 = r3 + r1 = 0 + 1 = 1
add r3 r7  // r3 = r3 + r7 = 1 + 0x0 = 0x1 = &mem[1]
add r3 r0  // r3 = r3 + r0 = &mem[1] + count = &mem[1 + count]
lb r3 r3   // r3 = mem[1 + count] = MSW

xor r4 r4  // r4 = r4 ^ r4 = 0
xor r5 r5  // r5 = r5 ^ r5 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
and r4 r1  // r4 = r4 & 1 = MSW[0]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[7:4] ^ MSW[0]

add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = MSW >> 1
and r4 r1  // r4 = r4 & 1 = MSW[1]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[7:4] ^ ^MSW[1:0]

add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = MSW >> 2
and r4 r1  // r4 = r4 & 1 = MSW[2]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[7:4] ^ ^MSW[2:0] = p8
```

```
// Storing p8 in mem[68]
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #3     // r7 = 8
add r4 r7  // r4 = r4 + r7 = 30 + 8 = 38
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 38 = 0x44 = &mem[68]
sb r6 r4   // mem[68] = r6 = p8

// Constructing MSW_out
add r5 r5  // r5 = r5 + r5 = 2 + 2 = 4
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 4 = LSW >> 4
and r4 r1  // r4 = r4 & 1 = LSW[4]
add r4 r4  // r4 = r4 + r4 = 2 * LSW[4]
add r6 r4  // r6 = { LSW[4], p8 }

add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = LSW >> 5
and r4 r1  // r4 = r4 & 1 = LSW[5]
add r4 r4  // r4 = r4 + r4 = 2 * LSW[5]
add r4 r4  // r4 = r4 + r4 = 4 * LSW[5]
add r6 r4  // r6 = { LSW[5:4], p8 }

add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = LSW >> 6
```

```
and r4 r1  // r4 = r4 & 1 = LSW[6]
add r4 r4  // r4 = r4 + r4 = 2 * LSW[6]
add r4 r4  // r4 = r4 + r4 = 4 * LSW[6]
add r4 r4  // r4 = r4 + r4 = 8 * LSW[6]
add r6 r4  // r6 = { LSW[6:4], p8 }

add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = LSW >> 7
and r4 r1  // r4 = r4 & 1 = LSW[7]
add r4 r4  // r4 = r4 + r4 = 2 * LSW[7]
add r4 r4  // r4 = r4 + r4 = 4 * LSW[7]
add r4 r4  // r4 = r4 + r4 = 8 * LSW[7]
add r4 r4  // r4 = r4 + r4 = 16 * LSW[7]
add r6 r4  // r6 = { LSW[7:4], p8 }

xor r4 r4  // r4 = r4 ^ r4 = 0
xor r5 r5  // r5 = r5 ^ r5 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
and r4 r1  // r4 = r4 & 1 = MSW[0]
add r4 r4  // r4 = r4 + r4 = 2 * MSW[0]
add r4 r4  // r4 = r4 + r4 = 4 * MSW[0]
add r4 r4  // r4 = r4 + r4 = 8 * MSW[0]
add r4 r4  // r4 = r4 + r4 = 16 * MSW[0]
add r4 r4  // r4 = r4 + r4 = 32 * MSW[0]
add r6 r4  // r6 = { MSW[0], LSW[7:4], p8 }

add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r3 = r4 >> 1 = MSW >> 1
```

```
and r4 r1  // r4 = r4 & 1 = MSW[1]
add r4 r4  // r4 = r4 + r4 = 2 * MSW[1]
add r4 r4  // r4 = r4 + r4 = 4 * MSW[1]
add r4 r4  // r4 = r4 + r4 = 8 * MSW[1]
add r4 r4  // r4 = r4 + r4 = 16 * MSW[1]
add r4 r4  // r4 = r4 + r4 = 32 * MSW[1]
add r4 r4  // r4 = r4 + r4 = 64 * MSW[1]
add r6 r4  // r6 = { MSW[1:0], LSW[7:4], p8 }

add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r3 = r4 >> 2 = MSW >> 2
and r4 r1  // r4 = r4 & 1 = MSW[2]
add r4 r4  // r4 = r4 + r4 = 2 * MSW[2]
add r4 r4  // r4 = r4 + r4 = 4 * MSW[2]
add r4 r4  // r4 = r4 + r4 = 8 * MSW[2]
add r4 r4  // r4 = r4 + r4 = 16 * MSW[2]
add r4 r4  // r4 = r4 + r4 = 32 * MSW[2]
add r4 r4  // r4 = r4 + r4 = 64 * MSW[2]
add r4 r4  // r4 = r4 + r4 = 128 * MSW[2]
add r6 r4  // r6 = { MSW[2:0], LSW[7:4], p8 } = MSW_out

// Storing MSW_out into mem[31 + count]
lut #9     // r7 = 0x1E = &mem[30]
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r0  // r4 = r4 + r0 = count
add r4 r1  // r4 = r4 + r1 = count + 1
add r4 r7  // r4 = r4 + r7 = count + 1 + 0x1E = count + 0x1F = &mem[31 + count]
sb r6 r4   // mem[31 + count] = MSW_out

// Calculating p4
```

```
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r6 r6 // r6 = r6 ^ r6 = 0

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = LSW >> 1
and r4 r1  // r4 = r4 & 1 = LSW[1]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = LSW[1]

add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = LSW >> 2
and r4 r1  // r4 = r4 & 1 = LSW[2]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[2:1]

add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = LSW >> 3
and r4 r1  // r4 = r4 & 1 = LSW[3]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[3:1]

add r5 r5  // r5 = r5 + r5 = 3 + 3 = 6
add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = LSW >> 7
and r4 r1  // r4 = r4 & 1 = LSW[7]
xor r6 r4  // r6 = r6 ^ r4 = LSW[7] ^ ^LSW[3:1]
```

```
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
and r4 r1  // r4 = r4 & 1 = MSW[0]
xor r6 r4  // r6 = r6 ^ r4 = MSW[0] ^ LSW[7] ^ ^LSW[3:1]

add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = MSW >> 1
and r4 r1  // r4 = r4 & 1 = MSW[1]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[1:0] ^ LSW[7] ^ ^LSW[3:1]

add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = MSW >> 2
and r4 r1  // r4 = r4 & 1 = MSW[2]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[2:0] ^ LSW[7] ^ ^LSW[3:1] = p4

// Storing p4 in mem[64]
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #2     // r7 = 4
add r4 r7  // r4 = r4 + r7 = 30 + 4 = 34
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 34 = 0x40 = &mem[64]
sb r6 r4   // mem[64] = r6 = p4

// Calculating p2
xor r5 r5  // r5 = r5 ^ r5 = 0
```

```
xor r6 r6 // r6 = r6 ^ r6 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
and r4 r1  // r4 = r4 & 1 = LSW[0]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = LSW[0]

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r5 r5  // r5 = r5 + r5 = 1 + 1 = 2
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = LSW >> 2
and r4 r1  // r4 = r4 & 1 = LSW[2]
xor r6 r4  // r6 = r6 ^ r4 = LSW[2] ^ LSW[0]

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = LSW >> 3
and r4 r1  // r4 = r4 & 1 = LSW[3]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[3:2] ^ LSW[0]

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 3 + 1 = 4
add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = LSW >> 5
and r4 r1  // r4 = r4 & 1 = LSW[5]
xor r6 r4  // r6 = r6 ^ r4 = LSW[5] ^ ^LSW[3:2] ^ LSW[0]

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
```

```
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = LSW >> 6
and r4 r1  // r4 = r4 & 1 = LSW[6]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[6:5] ^ ^LSW[3:2] ^ LSW[0]

xor r4 r4  // r4 = r4 ^ r4 = 0
xor r5 r5  // r5 = r5 ^ r5 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = MSW >> 1
and r4 r1  // r4 = r4 & 1 = MSW[1]
xor r6 r4  // r6 = r6 ^ r4 = MSW[1] ^LSW[6:5] ^ LSW[3:2] ^ LSW[0]

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = MSW >> 2
and r4 r1  // r4 = r4 & 1 = MSW[2]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[2:1] ^LSW[6:5] ^ LSW[3:2] ^ LSW[0] = p2

// Storing p2 in mem[62]
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #1     // r7 = 2
add r4 r7  // r4 = r4 + r7 = 30 + 2 = 32
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 32 = 0x3E = &mem[62]
sb r6 r4   // mem[62] = r6 = p2

// Calculating p1
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r6 r6 // r6 = r6 ^ r6 = 0
```

```
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
and r4 r1  // r4 = r4 & 1 = LSW[0]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = LSW[0]

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = LSW >> 1
and r4 r1  // r4 = r4 & 1 = LSW[1]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[1:0]

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = LSW >> 3
and r4 r1  // r4 = r4 & 1 = LSW[3]
xor r6 r4  // r6 = r6 ^ r4 = LSW[3] ^ ^LSW[1:0]

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 3 + 1 = 4
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 4 = LSW >> 4
and r4 r1  // r4 = r4 & 1 = LSW[4]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[4:3] ^ ^LSW[1:0]

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = LSW >> 6
```

```
and r4 r1  // r4 = r4 & 1 = LSW[6]
xor r6 r4  // r6 = r6 ^ r4 = LSW[6] ^ ^LSW[4:3] ^ ^LSW[1:0]

xor r4 r4  // r4 = r4 ^ r4 = 0
xor r5 r5  // r5 = r5 ^ r5 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
and r4 r1  // r4 = r4 & 1 = MSW[0]
xor r6 r4  // r6 = r6 ^ r4 = MSW[0] ^ LSW[6] ^ ^LSW[4:3] ^ ^LSW[1:0]

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = MSW >> 2
and r4 r1  // r4 = r4 & 1 = LSW[2]
xor r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = MSW[2] ^ MSW[0] ^ LSW[6] ^ ^LSW[4:3] ^ ^LSW[1:0] = p1

// Storing p1 in mem[61]
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
add r4 r1  // r4 = r4 + r1 = 30 + 1 = 31
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 31 = 0x3D = &mem[61]
sb r6 r4   // mem[61] = r6 = p1

// Calculating p0
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #1     // r7 = 2
```

```
add r4 r7  // r4 = r4 + r7 = 30 + 2 = 32
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 32 = 0x3E = &mem[62]
lb r5 r4   // r5 = mem[62] = p2
xor r6 r5  // r6 = r6 ^ r5 = p2 ^ p1

xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #2     // r7 = 4
add r4 r7  // r4 = r4 + r7 = 30 + 4 = 34
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 34 = 0x40 = &mem[64]
lb r5 r4   // r5 = mem[64] = p4
xor r6 r5  // r6 = r6 ^ r5 = p4 ^ p2 ^ p1

xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #3     // r7 = 8
add r4 r7  // r4 = r4 + r7 = 30 + 8 = 38
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 38 = 0x44 = &mem[68]
lb r5 r4   // r5 = mem[68] = p8
xor r6 r5  // r6 = r6 ^ r5 = p8 ^ p4 ^ p2 ^ p1

xor r5 r5  // r5 = r5 ^ r5 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
and r4 r1  // r4 = r4 & 1 = LSW[0]
xor r6 r4  // r6 = r6 ^ r4 = LSW[0] ^ p8 ^ p4 ^ p2 ^ p1
```

```
xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = LSW >> 1
and r4 r1  // r4 = r4 & 1 = LSW[1]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[1:0] ^ p8 ^ p4 ^ p2 ^ p1

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = LSW >> 2
and r4 r1  // r4 = r4 & 1 = LSW[2]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[2:0] ^ p8 ^ p4 ^ p2 ^ p1

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = LSW >> 3
and r4 r1  // r4 = r4 & 1 = LSW[3]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[3:0] ^ p8 ^ p4 ^ p2 ^ p1

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 3 + 1 = 4
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 4 = LSW >> 4
and r4 r1  // r4 = r4 & 1 = LSW[4]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[4:0] ^ p8 ^ p4 ^ p2 ^ p1

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = LSW >> 5
```

```
and r4 r1  // r4 = r4 & 1 = LSW[5]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[5:0] ^ p8 ^ p4 ^ p2 ^ p1

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = LSW >> 6
and r4 r1  // r4 = r4 & 1 = LSW[6]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[6:0] ^ p8 ^ p4 ^ p2 ^ p1

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = LSW >> 2
and r4 r1  // r4 = r4 & 1 = LSW[7]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[7:0] ^ p8 ^ p4 ^ p2 ^ p1

xor r5 r5  // r5 = r5 ^ r5 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
and r4 r1  // r4 = r4 & 1 = MSW[0]
xor r6 r4  // r6 = r6 ^ r4 = MSW[0] ^ ^LSW[7:0] ^ p8 ^ p4 ^ p2 ^ p1

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r1 = r4 >> 1 = MSW >> 1
and r4 r1  // r4 = r4 & 1 = MSW[1]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[1:0] ^ ^LSW[7:0] ^ p8 ^ p4 ^ p2 ^ p1

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
```

```
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r1 = r4 >> 2 = MSW >> 2
and r4 r1  // r4 = r4 & 1 = MSW[2]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[2:0] ^ ^LSW[7:0] ^ p8 ^ p4 ^ p2 ^ p1 = p0

// Constructing LSW_out
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
add r4 r1  // r4 = r4 + r1 = 30 + 1 = 31
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 31 = 0x3D =  &mem[61]
lb r5 r4   // r5 = mem[61] = p1
add r5 r5  // r5 = r5 + r5 = 2 * p1
add r6 r5  // r6 = r6 + r5 = { p1, p0 }

xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #1     // r7 = 2
add r4 r7  // r4 = r4 + r7 = 30 + 2 = 32
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 32 = 0x3E = &mem[62]
lb r5 r4   // r5 = mem[62] = p2
add r5 r5  // r5 = r5 + r5 = 2 * p2
add r5 r5  // r5 = r5 + r5 = 4 * p2
add r6 r5  // r6 = r6 + r5 = { p2, p1, p0 }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
and r4 r1  // r4 = r4 & 1 = LSW[0]
add r4 r4  // r4 = r4 + r4 = 2 * LSW[0]
```

```
add r4 r4  // r4 = r4 + r4 = 4 * LSW[0]
add r4 r4  // r4 = r4 + r4 = 8 * LSW[0]
add r6 r4  // r6 = r6 + r4 = { LSW[0], p2, p1, p0 }

xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #2     // r7 = 4
add r4 r7  // r4 = r4 + r7 = 30 + 4 = 34
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 32 = 0x40 = &mem[64]
lb r5 r4   // r5 = mem[64] = p4
add r5 r5  // r5 = r5 + r5 = 2 * p4
add r5 r5  // r5 = r5 + r5 = 4 * p4
add r5 r5  // r5 = r5 + r5 = 8 * p4
add r5 r5  // r5 = r5 + r5 = 16 * p4
add r6 r5  // r6 = r6 + r5 = { p4, LSW[0], p2, p1, p0 }

xor r5 r5  // r5 = r5 ^ r5 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = LSW >> 1
and r4 r1  // r4 = r4 & 1 = LSW[1]
add r4 r4  // r4 = r4 + r4 = 2 * LSW[1]
add r4 r4  // r4 = r4 + r4 = 4 * LSW[1]
add r4 r4  // r4 = r4 + r4 = 8 * LSW[1]
add r4 r4  // r4 = r4 + r4 = 16 * LSW[1]
add r4 r4  // r4 = r4 + r4 = 32 * LSW[1]
add r6 r4  // r4 = r6 + r4 = { LSW[1], p4, LSW[0], p2, p1, p0 }

xor r4 r4  // r4 = r4 ^ r4 = 0
```

```
add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = LSW >> 2
and r4 r1  // r4 = r4 & 1 = LSW[2]
add r4 r4  // r4 = r4 + r4 = 2 * LSW[2]
add r4 r4  // r4 = r4 + r4 = 4 * LSW[2]
add r4 r4  // r4 = r4 + r4 = 8 * LSW[2]
add r4 r4  // r4 = r4 + r4 = 16 * LSW[2]
add r4 r4  // r4 = r4 + r4 = 32 * LSW[2]
add r4 r4  // r4 = r4 + r4 = 64 * LSW[2]
add r6 r4  // r4 = r6 + r4 = { LSW[2:1], p4, LSW[0], p2, p1, p0 }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = LSW >> 3
and r4 r1  // r4 = r4 & 1 = LSW[3]
add r4 r4  // r4 = r4 + r4 = 2 * LSW[3]
add r4 r4  // r4 = r4 + r4 = 4 * LSW[3]
add r4 r4  // r4 = r4 + r4 = 8 * LSW[3]
add r4 r4  // r4 = r4 + r4 = 16 * LSW[3]
add r4 r4  // r4 = r4 + r4 = 32 * LSW[3]
add r4 r4  // r4 = r4 + r4 = 64 * LSW[3]
add r4 r4  // r4 = r4 + r4 = 128 * LSW[3]
add r6 r4  // r4 = r6 + r4 = { LSW[3:1], p4, LSW[0], p2, p1, p0 } = LSW_out

// Storing LSW_out into mem[30 + count]
lut #9     // r7 = 0x1E = &mem[30]
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r0  // r4 = r4 + r0 = count
add r4 r7  // r4 = r4 + r7 = count + 0x1E = &mem[30 + count]
sb r6 r4   // mem[30 + count] = LSW_out
```

```
add r0 r1  // r0 = r0 + r1 = count + 1
add r0 r1  // r0 = r0 + r1 = count + 1 + 1 = count + 2

lut #9     // r7 = 30
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r0  // r4 = r4 + r0 = 0 + count = count
xor r4 r7  // r4 = r4 ^ r7 = count ^ 30
lut #8     // r7 = 0
beq r4 r7  // If count ^ 30 != 0, jump to LUT[0] (LOOP)

DONE:      // Label DONE
lut #63    // Ack
```

Program 2 Pseudocode

```
# Given the input msg_cor (MSW, LSW), we return msg_rec (MSW_rec, LSW_rec)

decode(MSW, LSW):
  s8 = ^MSW[7:1]                        // ^msg_cor[15:9]
  s4 = ^MSW[7:4] ^ ^LSW[7:5]    // ^msg_cor[15:12] ^ ^msg_cor[7:5]
  s2 = ^MSW[7:6] ^ ^MSW[3:2] ^ ^LSW[7:6] ^ LSW[3]
  // ^msg_cor[15:14] ^ ^msg_cor[11:10] ^ ^msg_cor[7:6] ^ msg_cor[3]
  s1 = MSW[7] ^ MSW[5] ^ MSW[3] ^ MSW[1] ^ LSW[7] ^ LSW[5] ^ LSW[3]
  // msg_cor[15] ^ msg_cor[13] ^ msg_cor[11] ^ msg_cor[9] ^ msg_cor[7] ^ msg_cor[5] ^
msg_cor[3]
  s0 = ^MSW[7:0] ^ ^LSW[7:1]    // ^msg_cor[15:1]
  par_p = { MSW[0],LSW[4],LSW[2],LSW[1] }  // { msg_cor[8], msg_cor[4], msg_cor[2], msg_cor[1] }
  par_s = { s8, s4, s2, s1 }
  par_mis = par_p ^ par_s
  shift = 1 << par_mis
  msg_sft = msg_cor ^ shift
  if msg_cor[0] == s0:
    if par_mis == 0:
      flag = 00000
    else:
      flag = 10000
  else:
      flag = 01000
  MSW_rec = { flag, MSW[7:5] }
  LSW_rec = { MSW[4:1], LSW[7:5], LSW[3] }
  // { flag, msg_sft[15:9], msg_sft[7:5], msg_sft[3] }
  return MSW_rec, LSW_rec
```

```
test_bench():
  for i in range(0, 30, 2):
    mem[i + 1], mem[i] = decode(mem[i + 31], mem[i + 30])
```

Program 2 Assembly Code

Assume mem[0] is at address 0x0, mem[1] at address 0x1, …, mem[30] at address 0x1E, mem[31] at address 0x1F, …
Assume code starts at address 0x000

LUT[0] = 0x001 (LOOP, 1)
LUT[1] = 0x1B7 (SHIFT_LOW, 439)
LUT[2] = 0x1C0 (SHIFT_HIGH, 448)
LUT[3] = 0x1CD (SHIFT_LOW_DONE, 461)
LUT[4] = 0x1D4 (SHIFT_HIGH_DONE, 468)
LUT[5] = 0x1D6 (SHIFT_DONE, 470)
LUT[6] = 0x1F8 (FLAG_IF_ELSE, 504)
LUT[7] = 0x1FD (FLAG_ELSE, 509)
LUT[8] = 0x200 (FLAG_DONE, 512)

LUTI[0] stores 2^0 = 1 = 0b0000 0001
LUTI[1] stores 2^1 = 2 = 0b0000 0010
LUTI[2] stores 2^2 = 4 = 0b0000 0100
LUTI[3] stores 2^3 = 8 = 0b0000 1000
LUTI[4] stores 2^4 = 16 = 0b0001 0000
LUTI[5] stores 2^5 = 32 = 0b0010 0000
LUTI[6] stores 2^6 = 64 = 0b0100 0000
LUTI[7] stores 2^7 = 128 = 0b1000 0000
LUTI[8] stores 0 = 0b0000 0000
LUTI[9] stores 30 = 0b0001 1110
LUTI[10] stores 7 = 0b0000 0111

```
xor r0 r0  // r0 = r0 ^ r0 = 0 (count = 0), address 0x000

LOOP:      // Assume label LOOP is at address 0x001
xor r1 r1  // r1 = r1 ^ r1 = 0
xor r2 r2  // r2 = r2 ^ r2 = 0
xor r3 r3  // r3 = r3 ^ r3 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r6 r6  // r6 = r6 ^ r6 = 0

lut #0     // r7 = 1
add r1 r7  // r1 = 1
lut #9     // r7 = 30
add r2 r7  // r2 = r2 + r7 = 0 + 30 = 30
add r2 r0  // r2 = r2 + r0 = 30 + count = &mem[30 + count]
lb r2 r2   // r2 = mem[30 + count] = LSW = msg_cor[7:0]

add r3 r1  // r3 = r3 + r1 = 0 + 1 = 1
lut #9     // r7 = 30
add r3 r7  // r3 = r3 + r7 = 1 + 30 = 31
add r3 r0  // r3 = r3 + r0 = 31 + count = &mem[31 + count]
lb r3 r3   // r3 = mem[31 + count] = MSW = msg_cor[15:8]

// Calculating s8
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = MSW >> 1
and r4 r1  // r4 = r4 & 1 = MSW[1]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = MSW[1]

add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
xor r4 r4  // r4 = r4 ^ r4 = 0
```

```
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = MSW >> 2
and r4 r1  // r4 = r4 & 1 = MSW[2]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[2:1]

add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = MSW >> 3
and r4 r1  // r4 = r4 & 1 = MSW[3]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[3:1]

add r5 r1  // r5 = r5 + r1 = 3 + 1 = 4
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 4 = MSW >> 4
and r4 r1  // r4 = r4 & 1 = MSW[4]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[4:1]

add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = MSW >> 5
and r4 r1  // r4 = r4 & 1 = MSW[5]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[5:1]

add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = MSW >> 6
and r4 r1  // r4 = r4 & 1 = MSW[6]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[6:1]
```

```
add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = MSW >> 7
and r4 r1  // r4 = r4 & 1 = MSW[7]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[7:1] = s8

// Storing s8 in mem[68]
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #3     // r7 = 8
add r4 r7  // r4 = r4 + r7 = 30 + 8 = 38
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 38 = 0x44 = &mem[68]
sb r6 r4   // mem[68] = r6 = s8

// Calculating s4
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r6 r6  // r6 = r6 ^ r6 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r5 r5  // r5 = r5 + r5 = 1 + 1 = 2
add r5 r5  // r5 = r5 + r5 = 2 + 2 = 4
add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = LSW >> 5
and r4 r1  // r4 = r4 & 1 = LSW[5]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = LSW[5]

add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
```

```
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = LSW >> 6
and r4 r1  // r4 = r4 & 1 = LSW[6]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[6:5]

add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = LSW >> 7
and r4 r1  // r4 = r4 & 1 = LSW[7]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[7:5]

xor r5 r5  // r5 = r5 ^ r5 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r5 r5  // r5 = r5 + r5 = 1 + 1 = 2
add r5 r5  // r5 = r5 + r5 = 2 + 2 = 4
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 4 = MSW >> 4
and r4 r1  // r4 = r4 & 1 = MSW[4]
xor r6 r4  // r6 = r6 ^ r4 = MSW[4] ^ ^LSW[7:5]

add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = MSW >> 5
and r4 r1  // r4 = r4 & 1 = MSW[5]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[5:4] ^ ^LSW[7:5]

add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
xor r4 r4  // r4 = r4 ^ r4 = 0
```

```
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = MSW >> 6
and r4 r1  // r4 = r4 & 1 = MSW[6]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[6:4] ^ ^LSW[7:5]

add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = MSW >> 7
and r4 r1  // r4 = r4 & 1 = MSW[7]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[7:4] ^ ^LSW[7:5] = s4

// Storing s4 in mem[64]
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #2     // r7 = 4
add r4 r7  // r4 = r4 + r7 = 30 + 4 = 34
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 34 = 0x40 = &mem[64]
sb r6 r4   // mem[64] = r6 = s4

// Calculating s2
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r6 r6  // r6 = r6 ^ r6 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = LSW >> 3
and r4 r1  // r4 = r4 & 1 = LSW[3]
```

```
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = LSW[3]

add r5 r5  // r5 = r5 + r5 = 3 + 3 = 6
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = LSW >> 6
and r4 r1  // r4 = r4 & 1 = LSW[6]
xor r6 r4  // r6 = r6 ^ r4 = LSW[6] ^ LSW[3]

add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = LSW >> 7
and r4 r1  // r4 = r4 & 1 = LSW[7]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[7:6] ^ LSW[3]

xor r5 r5  // r5 = r5 ^ r5 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r5 r5  // r5 = r5 + r5 = 1 + 1 = 2
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = MSW >> 2
and r4 r1  // r4 = r4 & 1 = MSW[2]
xor r6 r4  // r6 = r6 ^ r4 = MSW[2] ^ ^LSW[7:6] ^ LSW[3]

add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = MSW >> 3
and r4 r1  // r4 = r4 & 1 = MSW[3]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[3:2] ^ ^LSW[7:6] ^ LSW[3]
```

```
add r5 r5  // r5 = r5 + r5 = 3 + 3 = 6
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = MSW >> 6
and r4 r1  // r4 = r4 & 1 = MSW[6]
xor r6 r4  // r6 = r6 ^ r4 = MSW[6] ^ ^MSW[3:2] ^ ^LSW[7:6] ^ LSW[3]

add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = MSW >> 7
and r4 r1  // r4 = r4 & 1 = MSW[7]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[7:6] ^ ^MSW[3:2] ^ ^LSW[7:6] ^ LSW[3] = s2

// Storing s2 in mem[62]
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #1     // r7 = 2
add r4 r7  // r4 = r4 + r7 = 30 + 2 = 32
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 32 = 0x3E = &mem[62]
sb r6 r4   // mem[62] = r6 = s2

// Calculating s1
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r6 r6  // r6 = r6 ^ r6 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
add r5 r5  // r5 = r5 + r5 = 1 + 1 = 2
add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
```

```
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = LSW >> 3
and r4 r1  // r4 = r4 & 1 = LSW[3]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = LSW[3]

add r5 r1  // r5 = r5 + r1 = 3 + 1 = 4
add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = LSW >> 5
and r4 r1  // r4 = r4 & 1 = LSW[5]
xor r6 r4  // r6 = r6 ^ r4 = LSW[5] ^ LSW[3]

add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = LSW >> 7
and r4 r1  // r4 = r4 & 1 = LSW[7]
xor r6 r4  // r6 = r6 ^ r4 = LSW[7] ^ LSW[5] ^ LSW[3]

xor r5 r5  // r5 = r5 ^ r5 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = MSW >> 1
and r4 r1  // r4 = r4 & 1 = MSW[1]
xor r6 r4  // r6 = r6 ^ r4 = MSW[1] ^ LSW[7] ^ LSW[5] ^ LSW[3]

add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
```

```
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = MSW >> 3
and r4 r1  // r4 = r4 & 1 = MSW[3]
xor r6 r4  // r6 = r6 ^ r4 = MSW[3]^ MSW[1] ^ LSW[7] ^ LSW[5] ^ LSW[3]

add r5 r1  // r5 = r5 + r1 = 3 + 1 = 4
add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = MSW >> 5
and r4 r1  // r4 = r4 & 1 = MSW[5]
xor r6 r4  // r6 = r6 ^ r4 = MSW[5] ^ MSW[3]^ MSW[1] ^ LSW[7] ^ LSW[5] ^ LSW[3]

add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = MSW >> 7
and r4 r1  // r4 = r4 & 1 = MSW[7]
xor r6 r4  // r6 = r6 ^ r4 = MSW[7] ^ MSW[5] ^ MSW[3]^ MSW[1] ^ LSW[7] ^ LSW[5] ^ LSW[3] = s1

// Storing s1 in mem[61]
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
add r4 r1  // r4 = r4 + r1 = 30 + 1 = 31
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 31 = 0x3D = &mem[61]
sb r6 r4   // mem[61] = r6 = s1

// Calculating s0
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r6 r6  // r6 = r6 ^ r6 = 0
```

```
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = LSW >> 1
and r4 r1  // r4 = r4 & 1 = LSW[1]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = LSW[1]

add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = LSW >> 2
and r4 r1  // r4 = r4 & 1 = LSW[2]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[2:1]

add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = LSW >> 3
and r4 r1  // r4 = r4 & 1 = LSW[3]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[3:1]

add r5 r1  // r5 = r5 + r1 = 3 + 1 = 4
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 4 = LSW >> 4
and r4 r1  // r4 = r4 & 1 = LSW[4]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[4:1]

add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = LSW >> 5
```

```
and r4 r1  // r4 = r4 & 1 = LSW[5]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[5:1]

add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = LSW >> 6
and r4 r1  // r4 = r4 & 1 = LSW[6]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[6:1]

add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = LSW >> 7
and r4 r1  // r4 = r4 & 1 = LSW[7]
xor r6 r4  // r6 = r6 ^ r4 = ^LSW[7:1]

xor r5 r5  // r5 = r5 ^ r5 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
and r4 r1  // r4 = r4 & 1 = MSW[0]
xor r6 r4  // r6 = r6 ^ r4 = 0 + r4 = r4 = MSW[0] ^ ^LSW[7:1]

add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = MSW >> 1
and r4 r1  // r4 = r4 & 1 = MSW[1]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[1:0] ^ ^LSW[7:1]

add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
xor r4 r4  // r4 = r4 ^ r4 = 0
```

```
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = MSW >> 2
and r4 r1  // r4 = r4 & 1 = MSW[2]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[2:0] ^ ^LSW[7:1]

add r5 r1  // r5 = r5 + r1 = 2 + 1 = 3
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 3 = MSW >> 3
and r4 r1  // r4 = r4 & 1 = MSW[3]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[3:0] ^ ^LSW[7:1]

add r5 r1  // r5 = r5 + r1 = 3 + 1 = 4
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 4 = MSW >> 4
and r4 r1  // r4 = r4 & 1 = MSW[4]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[4:0] ^ ^LSW[7:1]

add r5 r1  // r5 = r5 + r1 = 4 + 1 = 5
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 5 = MSW >> 5
and r4 r1  // r4 = r4 & 1 = MSW[5]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[5:0] ^ ^LSW[7:1]

add r5 r1  // r5 = r5 + r1 = 5 + 1 = 6
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 6 = MSW >> 6
and r4 r1  // r4 = r4 & 1 = MSW[6]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[6:0] ^ ^LSW[7:1]
```

```
add r5 r1  // r5 = r5 + r1 = 6 + 1 = 7
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 7 = MSW >> 7
and r4 r1  // r4 = r4 & 1 = MSW[7]
xor r6 r4  // r6 = r6 ^ r4 = ^MSW[7:0] ^ ^LSW[7:1] = s0

// Storing s0 in mem[60]
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 30 = 0x3C = &mem[60]
sb r6 r4   // mem[60] = r6 = s0

// Constructing par_p
xor r5 r5  // r5 = r5 ^ r5 = 0
xor r6 r6  // r6 = r6 ^ r6 = 0
add r5 r1  // r5 = r5 + r1 = 0 + 1 = 1
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 1 = LSW >> 1
and r4 r1  // r4 = r4 & 1 = LSW[1]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = { LSW[1] }

add r5 r1  // r5 = r5 + r1 = 1 + 1 = 2
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 2 = LSW >> 2
and r4 r1  // r4 = r4 & 1 = LSW[2]
add r4 r4  // r4 = r4 + r4 = 2 * LSW[2]
```

```
add r6 r4  // r6 = r6 + r4 = { LSW[2], LSW[1] }

add r5 r5  // r5 = r5 + r5 = 2 + 2 = 4
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW
srl r4 r5  // r4 = r4 >> r5 = r4 >> 4 = LSW >> 4
and r4 r1  // r4 = r4 & 1 = LSW[4]
add r4 r4  // r4 = r4 + r4 = 2 * LSW[4]
add r4 r4  // r4 = r4 + r4 = 4 * LSW[4]
add r6 r4  // r6 = r6 + r4 = { LSW[4], LSW[2], LSW[1] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW
and r4 r1  // r4 = r4 & 1 = MSW[0]
add r4 r4  // r4 = r4 + r4 = 2 * MSW[0]
add r4 r4  // r4 = r4 + r4 = 4 * MSW[0]
add r4 r4  // r4 = r4 + r4 = 8 * MSW[0]
add r6 r4  // r6 = r6 + r4 = { MSW[0], LSW[4], LSW[2], LSW[1] } = par_p

// Storing par_p in mem[63]
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r1  // r4 = r4 + r1 = 0 + 1 = 1
add r4 r1  // r4 = r4 + r1 = 1 + 1 = 2
add r4 r1  // r4 = r4 + r1 = 2 + 1 = 3
add r4 r7  // r4 = r4 + r7 = 3 + 30 = 33
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 33 = 0x3F = &mem[63]
sb r6 r4   // mem[63] = r6 = par_p

// Constructing par_s
xor r6 r6  // r6 = r6 ^ r6 = 0
```

```
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
add r4 r1  // r4 = r4 + r1 = 30 + 1 = 31
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 31 = 0x3D = &mem[61]
lb r5 r4   // r5 = mem[61] = s1
add r6 r5  // r6 = r6 + r5 = { s1 }

xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #1     // r7 = 2
add r4 r7  // r4 = r4 + r7 = 30 + 2 = 32
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 32 = 0x3E = &mem[62]
lb r5 r4   // r5 = mem[62] = s2
add r5 r5  // r5 = r5 + r5 = 2 * s2
add r6 r5  // r6 = r6 + r5 = { s2, s1 }

xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #2     // r7 = 4
add r4 r7  // r4 = r4 + r7 = 30 + 4 = 34
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 34 = 0x40 = &mem[64]
lb r5 r4   // r5 = mem[64] = s4
add r5 r5  // r5 = r5 + r5 = 2 * s4
add r5 r5  // r5 = r5 + r5 = 4 * s4
add r6 r5  // r6 = r6 + r5 = { s4, s2, s1 }
```

```
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #3     // r7 = 8
add r4 r7  // r4 = r4 + r7 = 30 + 8 = 38
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 38 = 0x44 = &mem[68]
lb r5 r4   // r5 = mem[68] = s8
add r5 r5  // r5 = r5 + r5 = 2 * s8
add r5 r5  // r5 = r5 + r5 = 4 * s8
add r5 r5  // r5 = r5 + r5 = 8 * s8
add r6 r5  // r6 = r6 + r5 = { s8, s4, s2, s1 } = par_s

// Calculating par_mis
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r1  // r4 = r4 + r1 = 0 + 1 = 1
add r4 r1  // r4 = r4 + r1 = 1 + 1 = 2
add r4 r1  // r4 = r4 + r1 = 2 + 1 = 3
add r4 r7  // r4 = r4 + r7 = 3 + 30 = 33
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 33 = 0x3F = &mem[63]
lb r5 r4   // r5 = mem[63] = par_p

xor r6 r5  // r6 = r6 ^ r5 = par_s ^ par_p = par_mis

// Calculating shift (shift_high, shift_low)
xor r2 r2  // r2 = r2 ^ r2 = 0 = shift_count
xor r5 r5  // r5 = r5 ^ r5 = 0
add r5 r1  // r6 = r5 + r1 = 0 + 1 = 1

xor r3 r3  // r3 = r3 ^ r3 = 0
```

```
add r3 r6  // r3 = r3 + r6 = 0 + r6 = r6 = par_mis
srl r3 r1  // r3 = r3 >> 1 = par_mis >> 1
srl r3 r1  // r3 = r3 >> 2 = par_mis >> 2
srl r3 r1  // r3 = r3 >> 3 = par_mis >> 3
and r3 r1  // r3 = r3 & 1 = par_mis[3]
lut #1     // r7 = 2
beq r3 r7  // If par_mis[3] == 1, jump to LUT[2] (SHIFT_HIGH)

SHIFT_LOW:      // Assume label SHIFT_LOW is at address 0x1B7
xor r3 r3  // r3 = r3 ^ r3 = 0
add r3 r2  // r3 = r3 + r2 = 0 + r2 = r2 = shift_count
xor r3 r6  // r3 = r3 ^ r6 = shift_count ^ par_mis
add r5 r5  // r5 = r5 + r5 = 2 * r5 = r5 << 1 = 1 << 1
add r2 r1  // r2 = r2 + r1 = r2 + 1 = shift_count + 1 = shift_count++
beq r3 r1  // if shift_count ^ par_mis != 0, jump to LUT[1] (SHIFT_LOW)
lut #1     // r7 = 2
add r7 r1  // r7 = r7 + r1 = 2 + 1 = 3
beq r1 r7  // Jump to LUT[3] (SHIFT_LOW_DONE)

SHIFT_HIGH:     // Assume label SHIFT_HIGH is at address 0x1C0
xor r4 r4  // r4 = r4 ^ r4 = 0
lut #10    // r7 = LUTI[10] = 7
add r4 r6  // r4 = r4 + r6 = 0 + r6 = par_mis
and r4 r7  // r4 = r4 & r7 = r4 & 0111 = { 0, par_mis[2:0] }
xor r3 r3  // r3 = r3 ^ r3 = 0
add r3 r2  // r3 = r3 + r2 = 0 + r2 = r2 = shift_count
xor r3 r4  // r3 = r3 ^ r6 = shift_count ^ { 0, par_mis[2:0] }
add r5 r5  // r5 = r5 + r5 = 2 * r5 = r5 << 1 = 1 << 1
add r2 r1  // r2 = r2 + r1 = r2 + 1 = shift_count + 1 = shift_count++
lut #1     // r7 = 2
beq r3 r7  // if (shift_count ^ {0, par_mis[2:0]}) != 0, jump to LUT[2] (SHIFT_HIGH)
lut #2     // r7 = 4
```

```
beq r1 r7  // Jump to LUT[4] (SHIFT_HIGH_DONE)

SHIFT_LOW_DONE: // Assume label SHIFT_LOW_DONE is at address 0x1CD
srl r5 r1  // r5 = r5 >> r1 = r5 >> 1
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r5  // r4 = r4 + r5 = 0 + r5 = r5
xor r5 r5  // r5 = r5 ^ r5 = 0
lut #2     // r7 = 4
add r7 r1  // r7 = r7 + r1 = 4 + 1 = 5
beq r1 r7  // Jump to LUT[5] (SHIFT_DONE)

SHIFT_HIGH_DONE:// Assume label SHIFT_HIGH_DONE is at address 0x1D4
xor r4 r4  // r4 = r4 ^ r4 = 0
srl r5 r1  // r5 = r5 >> r1 = r5 >> 1

SHIFT_DONE:     // Assume label SHIFT_DONE is at address 0x1D6

xor r3 r3  // r3 = r3 ^ r3 = 0
add r3 r1  // r3 = r3 + r1 = 0 + 1 = 1
lut #9     // r7 = 30
add r3 r7  // r3 = r3 + r7 = 1 + 30 = 31
add r3 r0  // r3 = r3 + r0 = 31 + count = &mem[31 + count]
lb r3 r3   // r3 = mem[31 + count] = MSW = msg_cor[15:8]

xor r2 r2  // r2 = r2 ^ r2 = 0
lut #9     // r7 = 30
add r2 r7  // r2 = r2 + r7 = 0 + 30 = 30
add r2 r0  // r2 = r2 + r0 = 30 + count = &mem[30 + count]
lb r2 r2   // r2 = mem[30 + count] = LSW = msg_cor[7:0]

// Construct msg_sft (MSW_sft, LSW_sft)
xor r3 r5  // r3 = r3 ^ r5 = MSW ^ shift_high = MSW_sft
```

```
xor r5 r5  // r5 = r5 ^ r5 = 0
add r5 r2  // r5 = r5 + r2 = 0 + r2 = r2 = LSW
and r5 r1  // r5 = r5 & 1 = LSW[0] = msg_cor[0]
xor r2 r4  // r2 = r2 ^ r4 = LSW ^ shift_low = LSW_sft

xor r4 r4  // r4 = r4 ^ r4 = 0
lut #9     // r7 = 30
add r4 r7  // r4 = r4 + r7 = 0 + 30 = 30
lut #9     // r7 = 0x1E = &mem[30]
add r4 r7  // r4 = r4 + r7 = 0x1E + 30 = 0x3C = &mem[60]
lb r4 r4   // r4 = mem[60] = s0

// r2 = MSW_sft
// r3 = LSW_sft
// r4 = s0
// r5 = msg_cor[0]
// r6 = par_mis

// Construct flag
xor r4 r5  // r4 = r4 ^ r5 = s0 ^ msg_cor[0]
lut #10    // r7 = 7
beq r4 r7  // if s0 ^ msg_cor[0] != 0, jump to LUT[7] (FLAG_ELSE)

xor r4 r4  // r4 = r4 ^ r4 = 0
xor r4 r6  // r4 = r4 ^ r6 = 0 ^ par_mis
lut #2     // r7 = 4
add r7 r1  // r7 = r7 + r1 = 4 + 1 = 5
add r7 r1  // r7 = r7 + r1 = 5 + 1 = 6
beq r4 r7  // if 0 ^ par_mis != 0, jump to LUT[6] (FLAG_IF_ELSE)

xor r5 r5  // r5 = r5 ^ r5 = 0 = flag (0b00000)
```

```
lut #3     // r7 = 8
beq r1 r7  // Jump to LUT[8] (FLAG_DONE)

FLAG_IF_ELSE:   // Assume label FLAG_IF_ELSE is at address 0x1F8
lut #4     // r7 = 16 = 0b10000
xor r5 r5  // r5 = r5 ^ r5 = 0
add r5 r7  // r5 = r5 + r7 = 0 + r7 = flag (0b10000)
lut #3     // r7 = 8
beq r1 r7  // Jump to LUT[8] (FLAG_DONE)

FLAG_ELSE:      // Assume label FLAG_ELSE is at address 0x1FD
lut #3     // r7 = 8 = 0b01000
xor r5 r5  // r5 = r5 ^ r5 = 0
add r5 r7  // r6 = r5 + r7 = 0 + r7 = flag (0b01000)

FLAG_DONE:      // Assume label FLAG_DONE is at address 0x200

// Constructing LSW_rec
xor r6 r6  // r6 = r6 ^ r6 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 3
and r4 r1  // r4 = r4 & 1 = LSW_sft[3]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = { msg_sft[3] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 3
```

```
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 4
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 5
and r4 r1  // r4 = r4 & 1 = LSW_sft[5] = msg_sft[5]
add r4 r4  // r4 = r4 + r4 = 2 * msg_sft[5]
add r6 r4  // r6 = r6 + r4 = { msg_sft[5], msg_sft[3] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 3
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 4
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 5
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 6
and r4 r1  // r4 = r4 & 1 = LSW_sft[6] = msg_sft[6]
add r4 r4  // r4 = r4 + r4 = 2 * msg_sft[6]
add r4 r4  // r4 = r4 + r4 = 4 * msg_sft[6]
add r6 r4  // r6 = r6 + r4 = { msg_sft[6:5], msg_sft[3] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r2  // r4 = r4 + r2 = 0 + r2 = r2 = LSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 3
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 4
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 5
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 6
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = LSW_sft >> 7
and r4 r1  // r4 = r4 & 1 = LSW_sft[7] = msg_sft[7]
add r4 r4  // r4 = r4 + r4 = 2 * msg_sft[7]
add r4 r4  // r4 = r4 + r4 = 4 * msg_sft[7]
add r4 r4  // r4 = r4 + r4 = 8 * msg_sft[7]
```

```
add r6 r4  // r6 = r6 + r4 = r4 = { msg_sft[7:5], msg_sft[3] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 1
and r4 r1  // r4 = r4 & 1 = MSW_sft[1] = msg_sft[9]
add r4 r4  // r4 = r4 + r4 = 2 * msg_sft[9]
add r4 r4  // r4 = r4 + r4 = 4 * msg_sft[9]
add r4 r4  // r4 = r4 + r4 = 8 * msg_sft[9]
add r4 r4  // r4 = r4 + r4 = 16 * msg_sft[9]
add r6 r4  // r6 = r6 + r4 = { msg_sft[9], msg_sft[7:5], msg_sft[3] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 2
and r4 r1  // r4 = r4 & 1 = MSW_sft[2] = msg_sft[10]
add r4 r4  // r4 = r4 + r4 = 2 * msg_sft[10]
add r4 r4  // r4 = r4 + r4 = 4 * msg_sft[10]
add r4 r4  // r4 = r4 + r4 = 8 * msg_sft[10]
add r4 r4  // r4 = r4 + r4 = 16 * msg_sft[10]
add r4 r4  // r4 = r4 + r4 = 32 * msg_sft[10]
add r6 r4  // r6 = r6 + r4 = { msg_sft[10:9], msg_sft[7:5], msg_sft[3] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 3
and r4 r1  // r4 = r4 & 1 = MSW_sft[3] = msg_sft[11]
add r4 r4  // r4 = r4 + r4 = 2 * msg_sft[11]
add r4 r4  // r4 = r4 + r4 = 4 * msg_sft[11]
```

```
add r4 r4  // r4 = r4 + r4 = 8 * msg_sft[11]
add r4 r4  // r4 = r4 + r4 = 16 * msg_sft[11]
add r4 r4  // r4 = r4 + r4 = 32 * msg_sft[11]
add r4 r4  // r4 = r4 + r4 = 64 * msg_sft[11]
add r6 r4  // r6 = r6 + r4 = { msg_sft[11:9], msg_sft[7:5], msg_sft[3] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 3
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 4
and r4 r1  // r4 = r4 & 1 = MSW_sft[4] = msg_sft[12]
add r4 r4  // r4 = r4 + r4 = 2 * msg_sft[12]
add r4 r4  // r4 = r4 + r4 = 4 * msg_sft[12]
add r4 r4  // r4 = r4 + r4 = 8 * msg_sft[12]
add r4 r4  // r4 = r4 + r4 = 16 * msg_sft[12]
add r4 r4  // r4 = r4 + r4 = 32 * msg_sft[12]
add r4 r4  // r4 = r4 + r4 = 64 * msg_sft[12]
add r4 r4  // r4 = r4 + r4 = 128 * msg_sft[12]
add r6 r4  // r6 = r6 + r4 = { msg_sft[12:9], msg_sft[7:5], msg_sft[3] }

// Storing LSW_rec into mem[0 + count]
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r0  // r4 = r4 + r0 = count
sb r6 r4   // mem[0 + count] = LSW_rec

// Constructing MSW_rec
xor r6 r6  // r6 = r6 ^ r6 = 0
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 1
```

```
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 3
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 4
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 5
and r4 r1  // r4 = r4 & 1 = MSW_sft[5] = msg_sft[13]
add r6 r4  // r6 = r6 + r4 = 0 + r4 = r4 = { msg_sft[13] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 3
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 4
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 5
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 6
and r4 r1  // r4 = r4 & 1 = MSW_sft[6] = msg_sft[14]
add r4 r4  // r4 = r4 + r4 = 2 * msg_sft[14]
add r6 r4  // r6 = r6 + r4 = { msg_sft[14:13] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r3  // r4 = r4 + r3 = 0 + r3 = r3 = MSW_sft
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 3
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 4
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 5
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 6
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = MSW_sft >> 7
and r4 r1  // r4 = r4 & 1 = MSW_sft[7] = msg_sft[15]
add r4 r4  // r4 = r4 + r4 = 2 * msg_sft[15]
add r4 r4  // r4 = r4 + r4 = 4 * msg_sft[15]
add r6 r4  // r6 = r6 + r4 = { msg_sft[15:13] }
```

```
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r5 // r4 = r4 + r5 = 0 + r5 = r5 = flag
and r4 r1  // r4 = r4 & 1 = flag[0]
add r4 r4  // r4 = r4 + r4 = 2 * flag[0]
add r4 r4  // r4 = r4 + r4 = 4 * flag[0]
add r4 r4  // r4 = r4 + r4 = 8 * flag[0]
add r6 r4  // r6 = r6 + r4 = { flag[0], msg_sft[15:13] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r5 // r4 = r4 + r5 = 0 + r5 = r5 = flag
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = flag >> 1
and r4 r1  // r4 = r4 & 1 = flag[1]
add r4 r4  // r4 = r4 + r4 = 2 * flag[1]
add r4 r4  // r4 = r4 + r4 = 4 * flag[1]
add r4 r4  // r4 = r4 + r4 = 8 * flag[1]
add r4 r4  // r4 = r4 + r4 = 16 * flag[1]
add r6 r4  // r6 = r6 + r4 = { flag[1:0], msg_sft[15:13] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r5 // r4 = r4 + r5 = 0 + r5 = r5 = flag
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = flag >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = flag >> 2
and r4 r1  // r4 = r4 & 1 = flag[2]
add r4 r4  // r4 = r4 + r4 = 2 * flag[2]
add r4 r4  // r4 = r4 + r4 = 4 * flag[2]
add r4 r4  // r4 = r4 + r4 = 8 * flag[2]
add r4 r4  // r4 = r4 + r4 = 16 * flag[2]
add r4 r4  // r4 = r4 + r4 = 32 * flag[2]
add r6 r4  // r6 = r6 + r4 = { flag[2:0], msg_sft[15:13] }

xor r4 r4  // r4 = r4 ^ r4 = 0
```

```
add r4 r5 // r4 = r4 + r5 = 0 + r5 = r5 = flag
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = flag >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = flag >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = flag >> 3
and r4 r1  // r4 = r4 & 1 = flag[3]
add r4 r4  // r4 = r4 + r4 = 2 * flag[3]
add r4 r4  // r4 = r4 + r4 = 4 * flag[3]
add r4 r4  // r4 = r4 + r4 = 8 * flag[3]
add r4 r4  // r4 = r4 + r4 = 16 * flag[3]
add r4 r4  // r4 = r4 + r4 = 32 * flag[3]
add r4 r4  // r4 = r4 + r4 = 64 * flag[3]
add r6 r4  // r6 = r6 + r4 = { flag[3:0] msg_sft[15:13] }

xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r5 // r4 = r4 + r5 = 0 + r5 = r5 = flag
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = flag >> 1
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = flag >> 2
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = flag >> 3
srl r4 r1  // r4 = r4 >> r1 = r4 >> 1 = flag >> 4
and r4 r1  // r4 = r4 & 1 = flag[4]
add r4 r4  // r4 = r4 + r4 = 2 * flag[4]
add r4 r4  // r4 = r4 + r4 = 4 * flag[4]
add r4 r4  // r4 = r4 + r4 = 8 * flag[4]
add r4 r4  // r4 = r4 + r4 = 16 * flag[4]
add r4 r4  // r4 = r4 + r4 = 32 * flag[4]
add r4 r4  // r4 = r4 + r4 = 64 * flag[4]
add r4 r4  // r4 = r4 + r4 = 128 * flag[4]
add r6 r4  // r6 = r6 + r4 = { flag[4:0], msg_sft[15:13] } = MSW_rec

// Storing MSW_rec into mem[1 + count]
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r0  // r4 = r4 + r0 = count
```

```
add r4 r1  // r4 = r4 + r1 = count + 1
sb r6 r4   // mem[1 + count] = MSW_rec

add r0 r1  // r0 = r0 + r1 = count + 1
add r0 r1  // r0 = r0 + r1 = count + 1 + 1 = count + 2
lut #9     // r7 = 30
xor r4 r4  // r4 = r4 ^ r4 = 0
add r4 r0  // r4 = r4 + r0 = 0 + count = count
xor r4 r7  // r4 = r4 ^ r7 = count ^ 30
lut #8     // r7 = 0
beq r4 r7  // If count ^ 30 != 0, jump to label LUT[0] (LOOP)

DONE:      // Label DONE
lut #63    // Ack
```

Program 3 Pseudocode

```
pattern_search(message, pattern):
  pattern >>>= 3
  messageCopy = message
  messageCopy2 = message
  checkEquality = 0
  countWithinBytes = 0
  countBytes = 0
  countTotalOccurrences = 0
  found = 0
  for (i = 0; i < 4; i += 1):
    for (j = 0; j < 4; j += 1):
      checkEquality = (messageCopy2 & 31) ^ pattern
      if (checkEquality == 0):
         countWithinBytes += 1
         found = 1
      messageCopy2 >>>= 1
    messageCopy >>>= 8
    messageCopy2 = messageCopy
    if (found == 1) countBytes += 1
    found = 0
  messageCopy = message
  for (int k = 0; k < 28; k += 1):
    checkEquality = (messageCopy & 31) ^ pattern
    if (checkEquality == 0):
       countTotalOccurrences += 1
    messageCopy >>>= 1
  return countWithinBytes, countBytes, countTotalOccurrences
```

Program 3 Assembly Code

LUT[0] = SKIP = 34 (SKIP) 0x22
LUT[1] = SKIP2 = 45 (SKIP2) 0x2d
LUT[2] = SKIP3 = 57 (SKIP3) 0x39
LUT[3] = 74 (SKIP4) 0x4a
LUT[4] = LOOP1 = 25 (LOOP1) 0x19
LUT[5] = 105 (SKIP5) 0x69
LUT[6] = SKIP6 = 118 0x76
LUT[7] = SKIP7 = 132 0x84
LUT[8] = SKIP8 = 145 0x91
LUT[9] = 90 (LOOP2) 0x5a
LUT[10] = SKIP9 = 185 0xb9
LUT[11] = SKIP10 = 196 0xc4
LUT[12] = SKIP11 = 208 0xd0
LUT[13] = SKIP12 = 221 0xdd
LUT[14] = SKIP13 = 245 0xf5
LUT[15] = SKIP14 = 271 0x10f
LUT[16] = SKIP15 = 299 0x12b
LUT[17] = SKIP16 = 329 0x149
LUT[18] = LOOP3 = 172 0xac
LUT[19] = SKIP17 = 354 0x162
LUT[20] = SKIP18 = 365 0x16d
LUT[21] = SKIP19 = 377 0x179
LUT[22] = SKIP20 = 390 0x186

```
lut #0     // assume LUTI[0] stores 1
```

```
xor r1 r1
xor r2 r2
xor r3 r3
xor r4 r4
xor r5 r5
add r1 r7 // r1 = 1
add r2 r1 // r2 = 1
add r2 r2 // r2 = 2
add r3 r2 // r3 = 2
add r3 r3 // r3 = 4
add r4 r3 // r4 = 4
add r4 r4 // r4 = 8
add r5 r4 // r5 = 8
add r5 r5 // r5 = 16
add r5 r4 // r5 = 24
add r5 r3 // r5 = 28
add r5 r2 // r5 = 30
add r5 r1 // r5 = 31
add r2 r1 // r2 = 3
add r1 r5 // r1 = 32
lb r1 r1   // r1 = mem[32]
srl r1 r2 // r1 = mem[32], [7:3] (r1 = pattern)

xor r0 r0
xor r2 r2 // r2 = count
LOOP1

//inner for loop for part 1

xor r6 r6
add r6 r0
lb r6 r6 // r6 = mem[0]
```

```
and r6 r5
xor r6 r1
//DO NOT CHANGE: r1, r6, r5
xor r7 r7
beq r6 r7 //(assume LUT[0] stores the mem address for the SKIP label)
lut #0
add r2 r7
SKIP

xor r6 r6
add r6 r0
lb r6 r6 // r6 = mem[0]
lut #0
srl r6 r7
and r6 r5
xor r6 r1
lut #0
beq r6 r7  //(assume LUT[1] stores the mem address for the SKIP2 label)
lut #0
add r2 r7
SKIP2

xor r6 r6
add r6 r0
lb r6 r6 // r6 = mem[0]
lut #0
srl r6 r7
srl r6 r7
and r6 r5
xor r6 r1
lut #1     //(assume LUTI[1] stores 2)
beq r6 r7  //(assume LUT[2] stores the mem address for the SKIP3 label)
```

```
lut #0
add r2 r7
SKIP3

xor r6 r6
add r6 r0
lb r6 r6 // r6 = mem[0]
lut #0
srl r6 r7
srl r6 r7
srl r6 r7
and r6 r5
xor r6 r1
lut #0
xor r3 r3
add r3 r7
lut #1
add r3 r7
beq r6 r3  //(assume LUT[3] stores the mem address for the SKIP4 label)
lut #0
add r2 r7
SKIP4

lut #0
add r0 r7
xor r4 r4
add r4 r7
add r4 r5
xor r4 r0
lut #2     //(assume LUTI[2] stores 4)
beq r4 r7  //(assume LUT[4] stores the mem address for the LOOP1 label)
```

```
xor r0 r0
add r0 r5
lut #0
add r0 r7
add r0 r7
sb r2 r0
xor r0 r0
xor r2 r2 // r2 = count

LOOP2

//inner for loop for part 2

xor r6 r6
add r6 r0
lb r6 r6 // r6 = mem[0]
and r6 r5
xor r6 r1
//DO NOT CHANGE: r1, r6, r5
xor r3 r3
lut #0
add r3 r7
lut #2
add r3 r7
beq r6 r3 //(assume LUT[5] stores the mem address for the SKIP5 label)
lut #0
add r2 r7
lut #3    //(assume LUTI[3] stores 8)
beq r2 r7 //(assume LUT[8] stores the mem address for the SKIP8 label)
SKIP5

xor r6 r6
```

```
add r6 r0
lb r6 r6 // r6 = mem[0]
lut #0
srl r6 r7
and r6 r5
xor r6 r1
add r3 r7
beq r6 r3 //(assume LUT[6] stores the mem address for the SKIP6 label)
lut #0
add r2 r7
lut #3
beq r2 r7
SKIP6

xor r6 r6
add r6 r0
lb r6 r6 // r6 = mem[0]
lut #0
srl r6 r7
srl r6 r7
and r6 r5
xor r6 r1
add r3 r7
beq r6 r7 //(assume LUT[7] stores the mem address for the SKIP7 label)
lut #0
add r2 r7
lut #11
beq r2 r7
SKIP7

xor r6 r6
add r6 r0
```

```
lb r6 r6 // r6 = mem[0]
lut #0
srl r6 r7
srl r6 r7
srl r6 r7
and r6 r5
xor r6 r1
add r3 r7
beq r6 r3
lut #0
add r2 r7
SKIP8

lut #0
add r0 r7
xor r4 r4
add r4 r7
add r4 r5
xor r4 r0
xor r3 r3
lut #3
add r3 r7
lut #0
add r3 r7
beq r4 r3 //(assume LUT[9] stores the mem address for the LOOP2 label)

xor r0 r0
add r0 r5
lut #0
add r0 r7
add r0 r7
add r0 r7
```

```
sb r2 r0
xor r0 r0
xor r2 r2 // r2 = count

//part 3

add r1 r1
add r1 r1
add r1 r1
add r5 r5
add r5 r5
add r5 r5

LOOP3

xor r6 r6
add r6 r0
lb r6 r6
and r6 r5
xor r6 r1
xor r3 r3
lut #3
add r3 r7
lut #1
add r3 r7
beq r6 r3 //(assume LUT[10] stores the mem address for the SKIP9 label)
lut #0
add r2 r7
SKIP9

xor r6 r6
add r6 r0
```

```
lb r6 r6
add r6 r6
and r6 r5
xor r6 r1
lut #0
add r3 r7
beq r6 r3  //(assume LUT[11] stores the mem address for the SKIP10 label)
lut #0
add r2 r7
SKIP10

xor r6 r6
add r6 r0
lb r6 r6
add r6 r6
add r6 r6
and r6 r5
xor r6 r1
lut #0
add r3 r7
beq r6 r3  //(assume LUT[12] stores the mem address for the SKIP11 label)
lut #0
add r2 r7
SKIP11

xor r6 r6
add r6 r0
lb r6 r6
add r6 r6
add r6 r6
add r6 r6
and r6 r5
```

```
xor r6 r1
lut #0
add r3 r7
beq r6 r3 //(assume LUT[13] stores the mem address for the SKIP12 label)
lut #0
add r2 r7
SKIP12

xor r6 r6
add r6 r0
xor r4 r4
add r4 r6
lb r6 r6
add r6 r6
add r6 r6
add r6 r6
add r6 r6
lut #0
add r4 r7
lb r4 r4
srl r4 r7
srl r4 r7
srl r4 r7
srl r4 r7
add r6 r4
and r6 r5
xor r6 r1
lut #0
add r3 r7
beq r6 r3 //(assume LUT[14] stores the mem address for the SKIP13 label)
lut #0
add r2 r7
```

```
SKIP13

xor r6 r6
add r6 r0
xor r4 r4
add r4 r6
lb r6 r6
add r6 r6
add r6 r6
add r6 r6
add r6 r6
add r6 r6
lut #0
add r4 r7
lb r4 r4
srl r4 r7
srl r4 r7
srl r4 r7
srl r4 r7
add r4 r4
add r6 r4
and r6 r5
xor r6 r1
lut #0
add r3 r7
beq r6 r3 //(assume LUT[15] stores the mem address for the SKIP14 label)
lut #0
add r2 r7
SKIP14

xor r6 r6
add r6 r0
```

```
xor r4 r4
add r4 r6
lb r6 r6
add r6 r6
add r6 r6
add r6 r6
add r6 r6
add r6 r6
add r6 r6
lut #0
add r4 r7
lb r4 r4
srl r4 r7
srl r4 r7
srl r4 r7
srl r4 r7
add r4 r4
add r4 r4
add r6 r4
and r6 r5
xor r6 r1
lut #0
add r3 r7
beq r6 r3 (assume LUT[16] stores the mem address for the SKIP15 label)
lut #0
add r2 r7
SKIP15

xor r6 r6
add r6 r0
xor r4 r4
add r4 r6
```

```
lb r6 r6
add r6 r6
add r6 r6
add r6 r6
add r6 r6
add r6 r6
add r6 r6
add r6 r6
lut #0
add r4 r7
lb r4 r4
srl r4 r7
srl r4 r7
srl r4 r7
srl r4 r7
add r4 r4
add r4 r4
add r4 r4
add r6 r4
and r6 r5
xor r6 r1
lut #0
add r3 r7
beq r6 r3 (assume LUT[17] stores the mem address for the SKIP16 label)
lut #0
add r2 r7
SKIP16

lut #0
add r0 r7
xor r4 r4
lut #9
```

```
add r4 r7
lut #0
add r4 r7
xor r4 r0
add r3 r7
beq r4 r3  //(assume LUT[18] stores the mem address for the LOOP3 label)

xor r6 r6
add r6 r0
lb r6 r6
and r6 r5
xor r6 r1
xor r3 r3
lut #4
add r3 r7
lut #1
add r3 r7
lut #0
add r3 r7
beq r6 r3 //(assume LUT[19] stores the mem address for the SKIP17 label)
lut #0
add r2 r7
SKIP17

xor r6 r6
add r6 r0
lb r6 r6
add r6 r6
and r6 r5
xor r6 r1
lut #0
add r3 r7
```

```
beq r6 r3  //(assume LUT[20] stores the mem address for the SKIP18 label)
lut #0
add r2 r7
SKIP18

xor r6 r6
add r6 r0
lb r6 r6
add r6 r6
add r6 r6
and r6 r5
xor r6 r1
lut #0
add r3 r7
beq r6 r3  //(assume LUT[21] stores the mem address for the SKIP19 label)
lut #0
add r2 r7
SKIP19

xor r6 r6
add r6 r0
lb r6 r6
add r6 r6
add r6 r6
add r6 r6
and r6 r5
xor r6 r1
lut #0
add r3 r7
beq r6 r3 //(assume LUT[22] stores the mem address for the SKIP20 label)
lut #0
add r2 r7
```

```
SKIP20

xor r0 r0
lut #5
add r0 r7
lut #1
add r0 r7
lut #0
add r0 r7
sb r2 r0

DONE:      // Label DONE
lut #63    // Ack
```

8. Changelog

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

- Milestone 4
  - Architectural Overview
    - Added the Zero flag from the ALU to the Branch
  - Machine Specification
    - Updated information about lookup tables in Internal Operands
  - Individual Component Specification
    - Updated the Top Level Schematic
  - Program Implementation
    - Updated assembly code for Program 1, 2 and 3
- Milestone 3
  - Assembler
    - Section added as section number 6
  - Program Implementation
    - Section number changed from 6 to 7
  - Changelog
    - Section number changed from 7 to 8
- Milestone 2
  - Architectural Overview
    - Added label for instruction memory
    - Added an extra control signal LUTen to the register file
  - Programmer's Model [Lite]
    - Added 4.3 about using ALU for non-arithmetic instructions
  - Individual Component Specification
    - Section added as section number 5
  - Program Implementation
    - Section number changed from 5 to 6
- Milestone 1
  - Initial version