CSE 141L Milestone 4

Vincent Ren; A17566012

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.

Vincent Ren

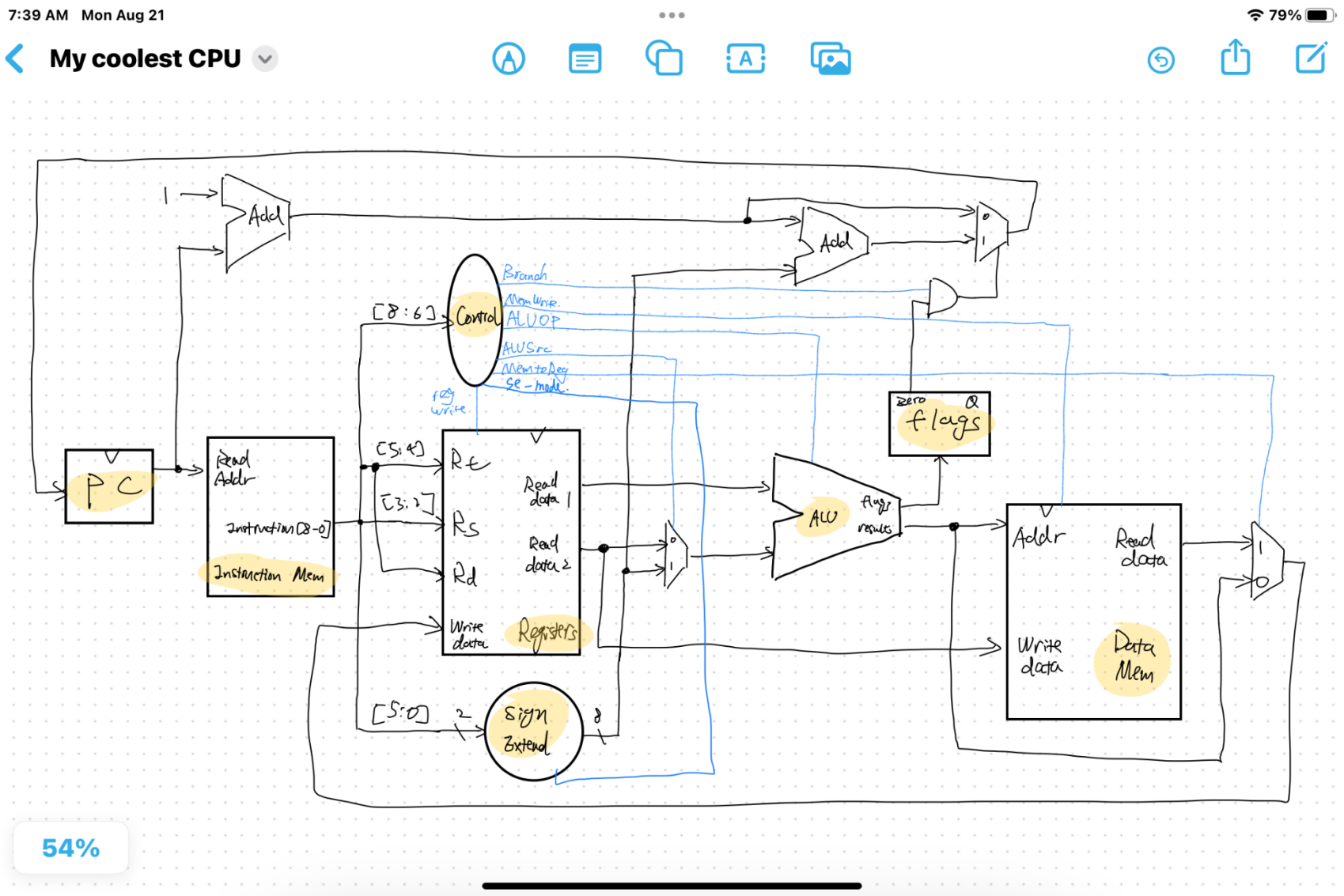
0. Team

Vincent Ren, A17566012

1. Introduction

My architecture is named MCC (My coolest CPU). The overall philosophy is designing an efficient microprocessor that is able to run grading programs and get full marks. My program is in reg-reg/load-store way, like the MIPS example.

2. Architectural Overview



3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	1bit immed bit; 2bits opcode; 2bits Rs; 2bits Rt; 2bits offset/NOT USE	LDR, STR, ADR, XOR
I	1bit immed; 2bits opcode; 2bits Rs, 4bits immed4	MOV, LSH, ADI
B	1bit immed; 2bits opcode; 6bits offset6	BRC

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
LDR = Load register	R	1 immed bit (0), 2bits opcode(00), 2bits Rs(XX), 2bits Rt(XX), 2bits NO USE(XX)	#Assume R1 has 0x30 LDR R0, R1 #Now R0 = Mem[48]	Result is saved in Rs (Rd == Rs) Rt's value should be valid memory address.
STR = Store to mem	R	1 immed bit (0), 2 bits opcode(01), 2bits Rs(XX), 2bits Rt(XX), 2bits NO USE(XX)	#Assume R0 has #30 #Assume R1 has 0xFF STR R1, R0 #now Mem[30] = 0xFF	Data mem is one byte wide. In the example, R1 is passed as Rs and R0 is passed as Rt.

ADR = Addition reg	R	1 immed bit (0), 2 bits opcode(10), 2bits Rs(XX), 2bits Rt(XX), 2bit NOT USE(X)	#Assume R0 has 0x01 #Assume R1 has 0x03 ADDR R0, R1 #Now R0 = 0x04	Result is saved in Rs (i.e. Rd = Rs) Overflow might happen
XOR = exclusive OR	R	1 immed bit (0), 2 bits opcode(11), 2bits Rs(XX), 2bits Rt(XX), 2bits NOT USE (XX)	#Assume R0 has 0b0001_1110 #Assume R1 has 0b0001_0011 XOR R0, R1 #Now R0 = 0b0000_1101	Result is saved in Rs (i.e. Rd = Rs)
MOV =move	I	1 immed bit (1), 2bits opcode(00), 2bits Rs(XX), 4bits immed4(XXXX)	#Assume R0 was initially 0 MOV R0, #5 #Now R0 = 0x05	Immed4 is 2's comp, range [-7, 7]; Immed4 is reserved for getting parity value;
LSH = Logic shift	I	1 immed bit(1), 2bits opcode(01), 2bits Rs(XX), 4bits shift4(XXXX)	#Assume R0 has 8'b0011_1111 LS R0, #3 #Now R0 has 8'b1111_1000 LS R0, #-4 #Now R0 has 8'b0000_1111	Result is saved in Rs (i.e. Rd = Rs) This is logic shift (bits will not be rotated) Shift4 is a value indicating number of shift positions desired. If shift4 is positive, Rs will be left shifted shift4 positions; if shift4 is negative, Rs will be shifted abs(shift4) positions
ADI = Addition Immediat e	I	1 immed bit (1), 2bits opcode (10), 2bits Rs(XX), 4bits immed4(XXXX)	#Assume R0 has 0x03 ADDI R0, #-5 #Now R0 = 0xFE	Result is saved in Rs (i.e. Rd = Rs) Overflow might happen. Immed4 range [-8, 7]

BRC = branch	B	3bits opcode(111), 6bits offset6(XXXXXX)	#Assume we have the following code 0x01: BR MYBRANCH … MYBRANCH: 0x11: #SOME CODE	BR will be determined by Z flag (zero). If Z flag is 1 (meaning ALU result is 0), BR will be executed; otherwise, BR will not be executed. Offset6 will be automatically calculated by the following formula: Label PC - current PC

Internal Operands

There are only 4 registers supported. All of them are general purpose registers. However, write register will always be equal to Rs, meaning that we cannot do R0 = R1 + R2 but R0 = R0 + R1

Control Flow (branches)

Only regular branch is provided. Branch will be depends on zeroQ value. If zeroQ == 1, branch will be triggered; otherwise will not. Branch is using lookup table. By passing a immed6 value to branch, loopkup table will return a absolute address to pc, and pc will update to that address in next cycle if absj == 1;

Addressing Modes

The mem address will be preserved as a constant in my program. To touch specific address, we can save the index of that address to a register. Then we can use LDR/STR to load and store data. Therefore, except the base address of Mem, indirect addressing mode plays the major role here.
e.g. Assume we know that the base address of Mem is 0x10. If we want to access the data in index 6 (starting from 0). We can do the following:

```
MOVI R0, base_addr
ADDI R0, #3
ADDI R0, #3
LDR R0, R0          # load data in the address saved in R0 to R0
```

4. Programmer's Model [Lite]

4.1

Since accessing memory is costly, programming should read/write necessary large amount of data in one time into somewhere not too expensive. Then programmer can gradually read/write data from that place with low cost. That's the idea of buffer.
Another thing I found during my processor design is that we have to trade off. In this project, instruction only have 9bits wide. So we need to really carefully allocate bits to different things such as opcode, registers, immediate values. If we give larger bits for registers and less bits for opcode, there will be only few operation options. Though that might work, but for single move we might need to divide it into several moves, resulting CPI increment. Respectively, less bits for registers means less space to hold multi-value at the same time. So we might have to store some of them into memory, resulting worse execution time.

4.2

Apparently no. MIPS and ARM have instruction size of 32 bits, but we only have 9bits. Some advanced operation would not work in our scenario. Therefore, we need to save the essentials into our design. Then for advanced operation we can divide it into several base operations. One approach I used in my design is that for R type instruction, the destination register would always be equal to the source register (i.e. Rd = Rs). Under that picture, we cannot do A = B + C but A = A + B only.

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?" to your Programmer Model's section (Section 4)

ALU will involve memory address pointer calculations, but will not participate in PC relative branch computations.
For memory address (LDR/STR), Rs and immediate value will be calculated as memory address by simply adding them up.
For PC relative branch, my processor will rely on PC_LUT, which hard coded necessary jump distance. When doing relative jumps (BR), immed6 will be passed as jump label, and PC_LUT will decode that (using case) to achieve specific relative jumps.

5. Individual Component Specification

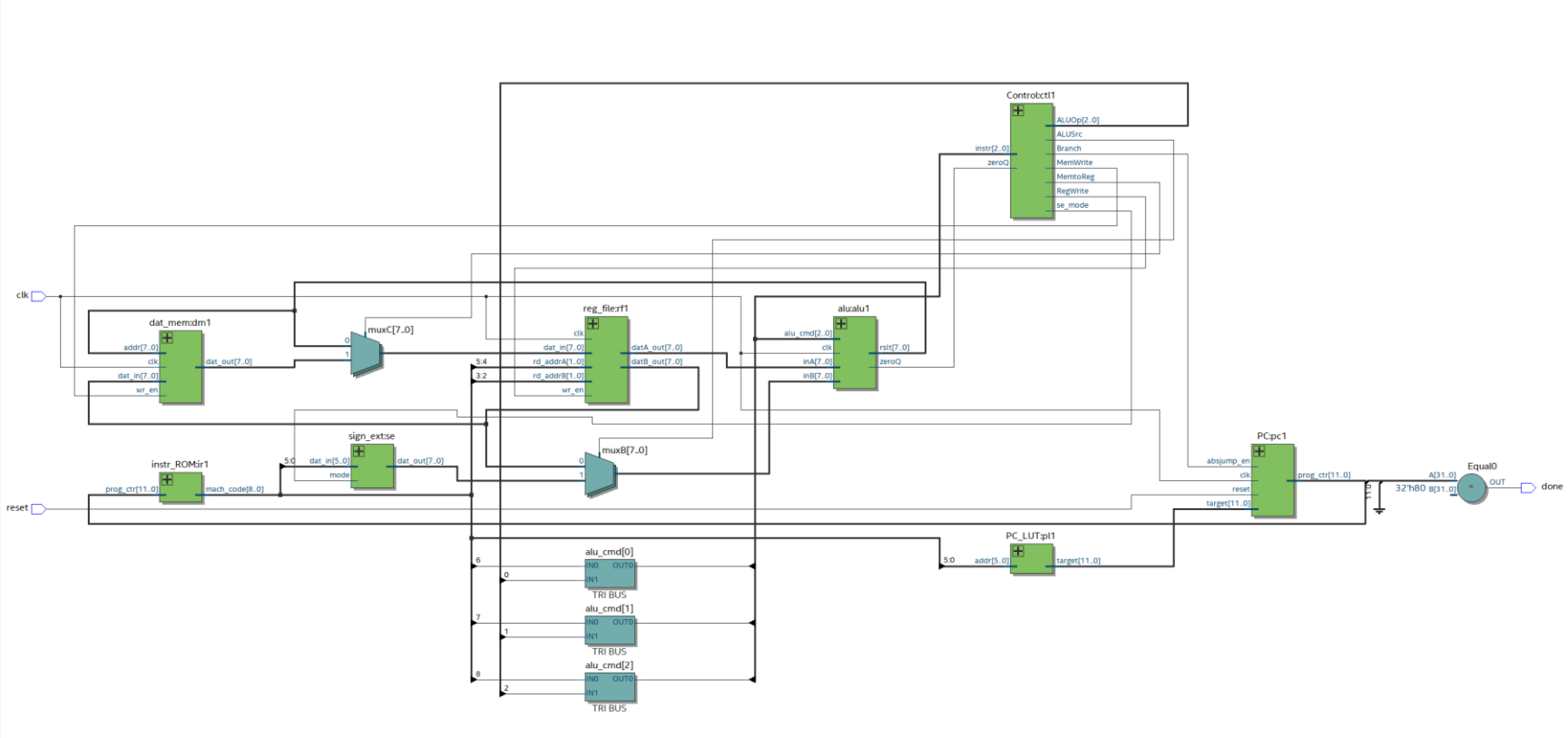
Top Level

Module file name: top_level.sv

Functionality Description

Integrate all components together.

Schematic



Program Counter

Module file name: PC.sv
Module testbench file name: milestone2_quicktest_tb.sv

Functionality Description

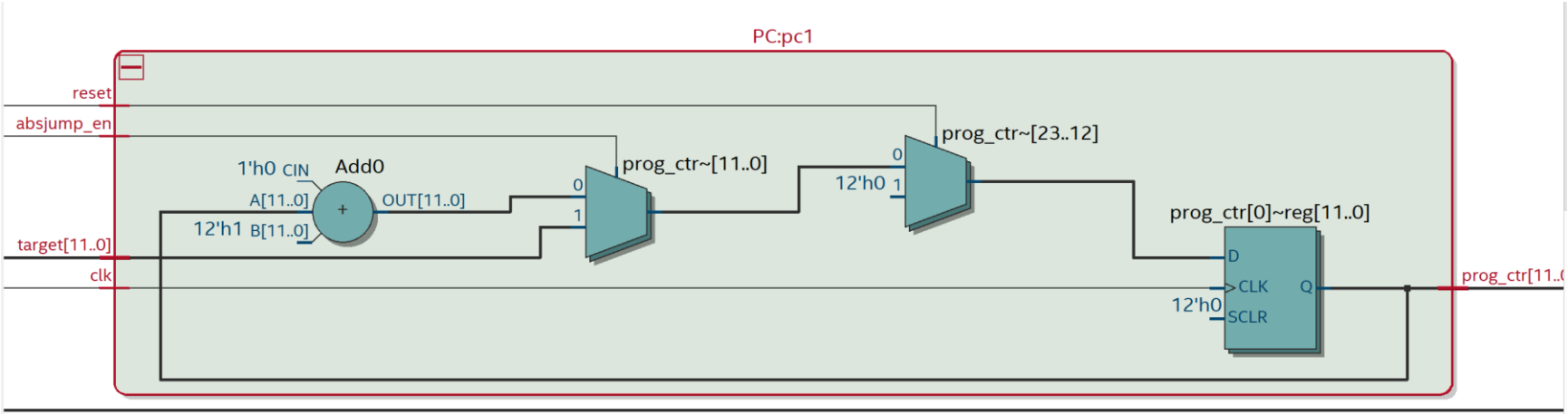
Control instruction flow of the processor. By default increment by 1 at pose edge.

Support branch jump using absolute destinations.

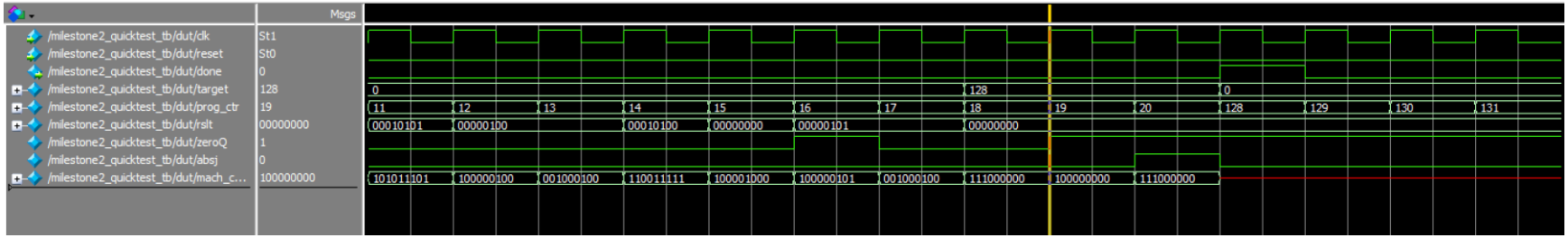
(Optional) Testbench Description

It tests if pc is increment by 1 at every cycle.
Also test branch function. Test if alu result is not equal to 0 (zeroQ is low), branch would not be triggered. Test if alu result is 0 (zeroQ is high), branch should be triggered.
Also test done flag. Test when pc = 128 (assume so far program ends at 128), done flag would be high.

Schematic



(Optional) Timing Diagram



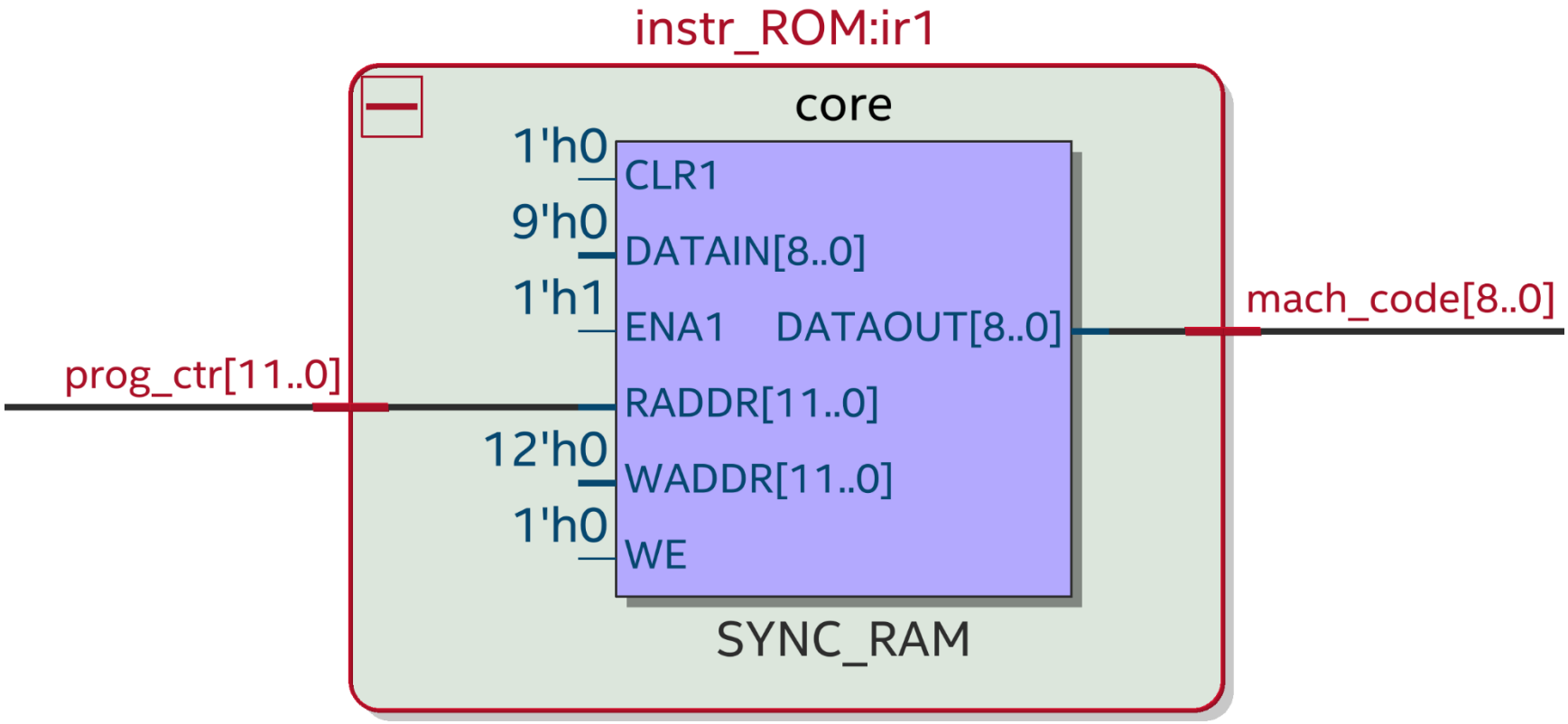
Instruction Memory

Module file name: instr_ROM.sv

Functionality Description

Hardcoded assembly code into mach_code.txt
instr_ROM will load mach_code.txt into its core memory and output machine code by given PC

Schematic



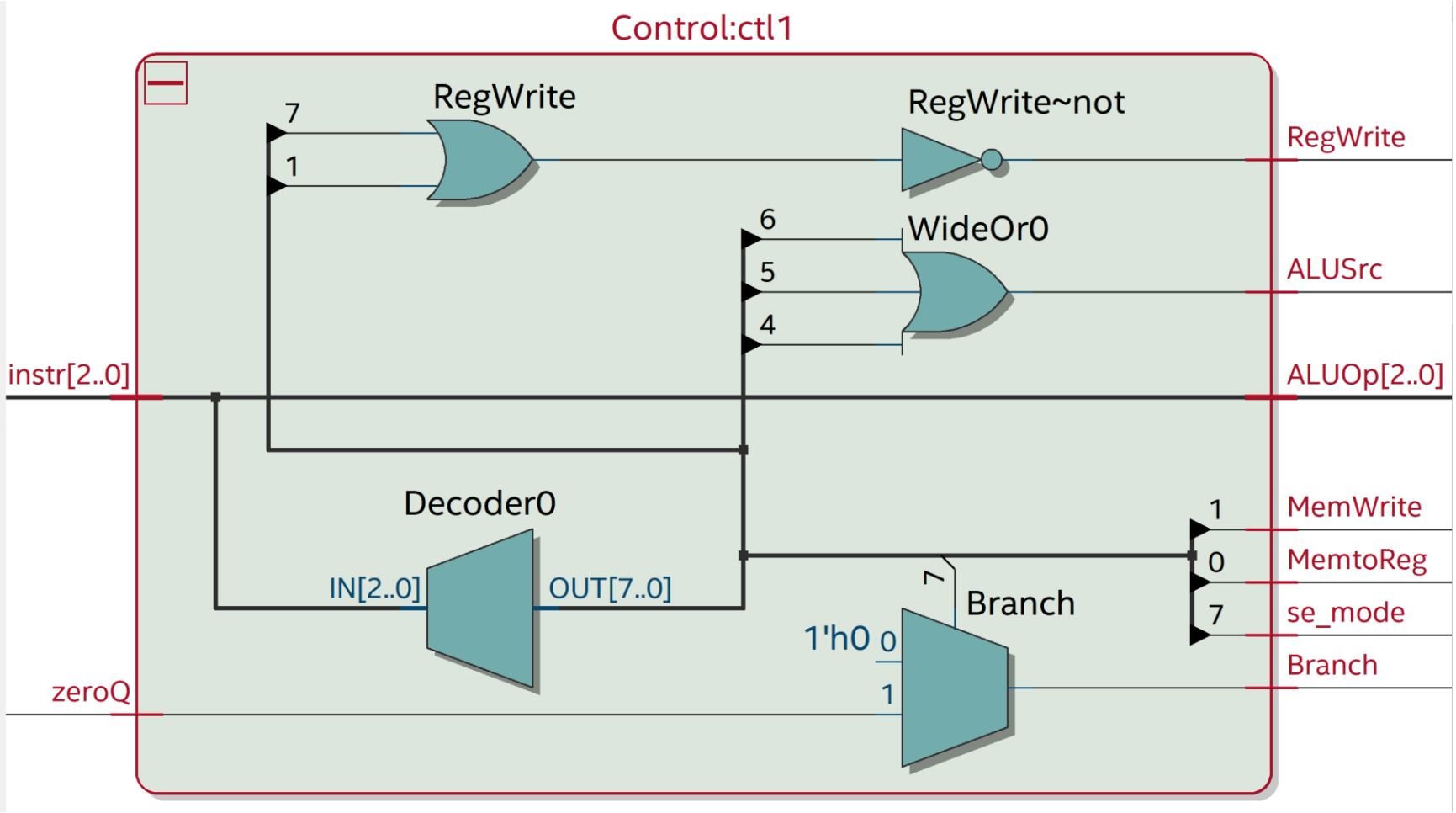
Control Decoder

Module file name: Control.sv

Functionality Description

Take first three bits of mach_code and assert a few signals properly

Schematic



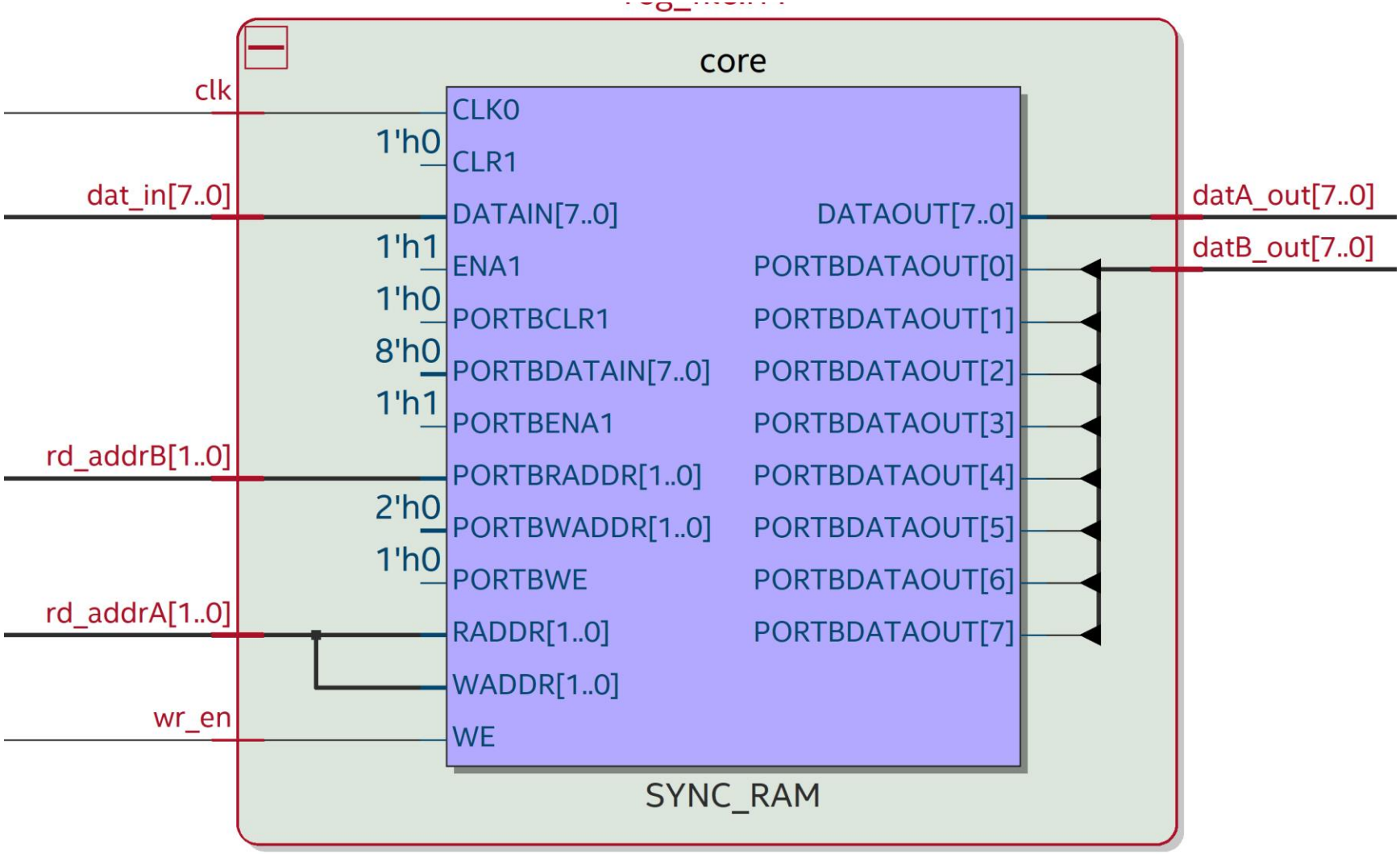
Register File

Module file name: reg_file.sv

Functionality Description

Reg_file memory. Provide fast cache memory.
Write reg will always be equal to Rs.
Take mach_code[5:2] and spilt it half to control two read address A(Rs) and B(Rt)

Schematic


core
clk
dat_in[7..0]
rd_addrB[1..0]
rd_addrA[1..0]
wr_en
1'h0
1'h1
1'h0
8'h0
1'h1
2'h0
1'h0
CLK0
CLR1
DATAIN[7..0]
ENA1
PORTBCLR1
PORTBDATAIN[7..0]
PORTBENA1
PORTBRADDR[1..0]
PORTBWADDR[1..0]
PORTBWE
RADDR[1..0]
WADDR[1..0]
WE
DATAOUT[7..0]
PORTBDATAOUT[0]
PORTBDATAOUT[1]
PORTBDATAOUT[2]
PORTBDATAOUT[3]
PORTBDATAOUT[4]
PORTBDATAOUT[5]
PORTBDATAOUT[6]
PORTBDATAOUT[7]
datA_out[7..0]
datB_out[7..0]
SYNC_RAM

ALU (Arithmetic Logic Unit)

Module file name: alu.sv
Module testbench file name: milestone2_quicktest_tb.sv

Functionality Description

Provide arithmetic function. For further reference, please refer to 3. Machine Specification.

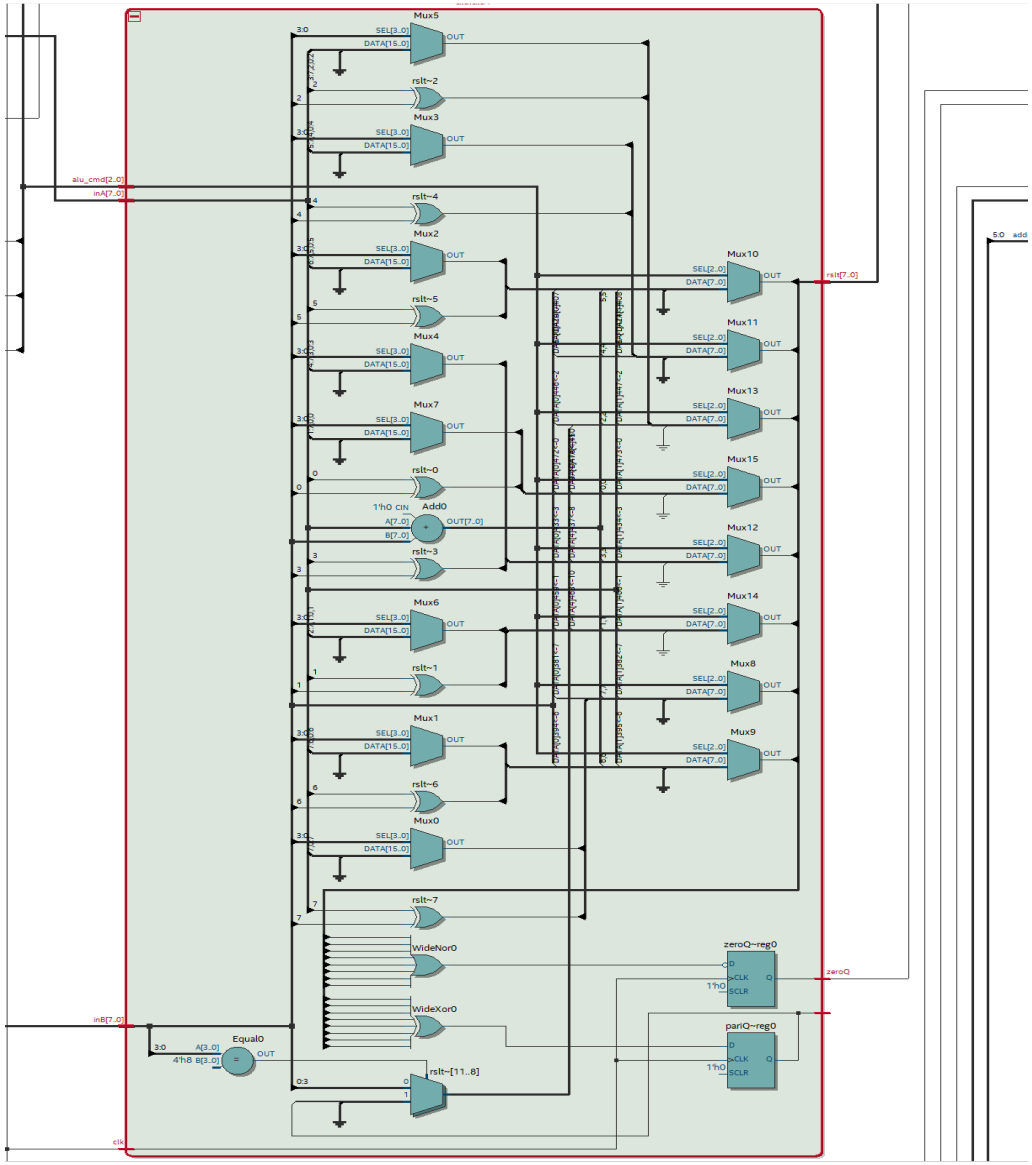
(Optional) Testbench Description

Test several general cases of each alu function.
Test LDR and STR
Test ADDR
Test XOR
Test MOV to assign decimal value and parity flag.
Test LS for both positive number (left shift) and negative number (right shift)
Test ADDI

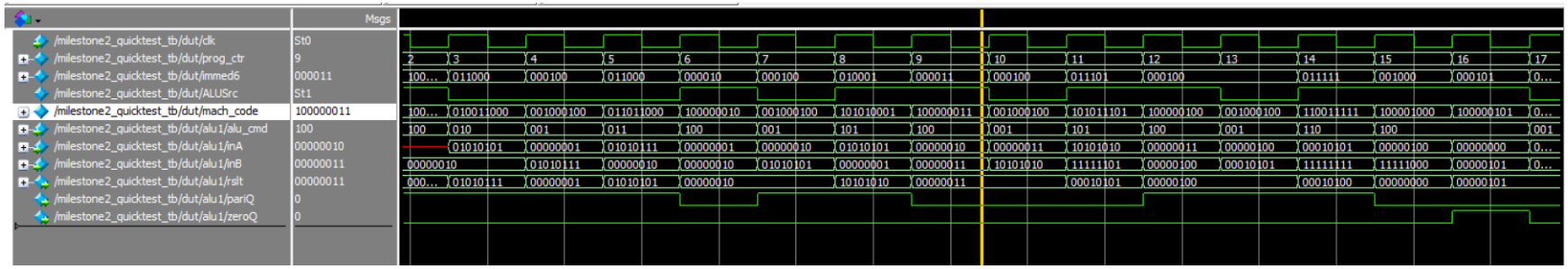
ALU Operations

TODO. What ALU operations will you be demonstrating? What instructions are they relevant to?
LDR: load from memory to reg
STR: store from reg to memory
ADDR: add two registers Rs and Rt, save sum to Rs
XOR: bitwise XOR two registers Rs and Rt, save result to Rs
MOV: assign decimal value [-7, 7] to reg or for special case assign parity flag to reg using reserved value -8
LS: logic shift bits of reg. Immed4 from [-7, -1] and [1, 7]. If positive, shift left; if negative shift right. Value 0 and -8 makes no sense here.
ADDI: add reg and immed4 value and save the result to reg itself. Immed4 can be positive and negative (support subtraction)

Schematic


Mux5
Mux3
Mux2
Mux4
Mux7
Mux6
Mux1
Mux0
Mux10
Mux11
Mux13
Mux15
Mux12
Mux14
Mux8
Mux9
rslt~0
rslt~1
rslt~2
rslt~3
rslt~4
rslt~5
rslt~6
rslt~7
rslt~[11..8]
Add0
WideNor0
WideXor0
Equal0
zeroQ~reg0
pariQ~reg0
aluCmd[2..0]
inA[7..0]
inB[7..0]
clk
rslt[7..0]
zeroQ

(Optional) Timing Diagram



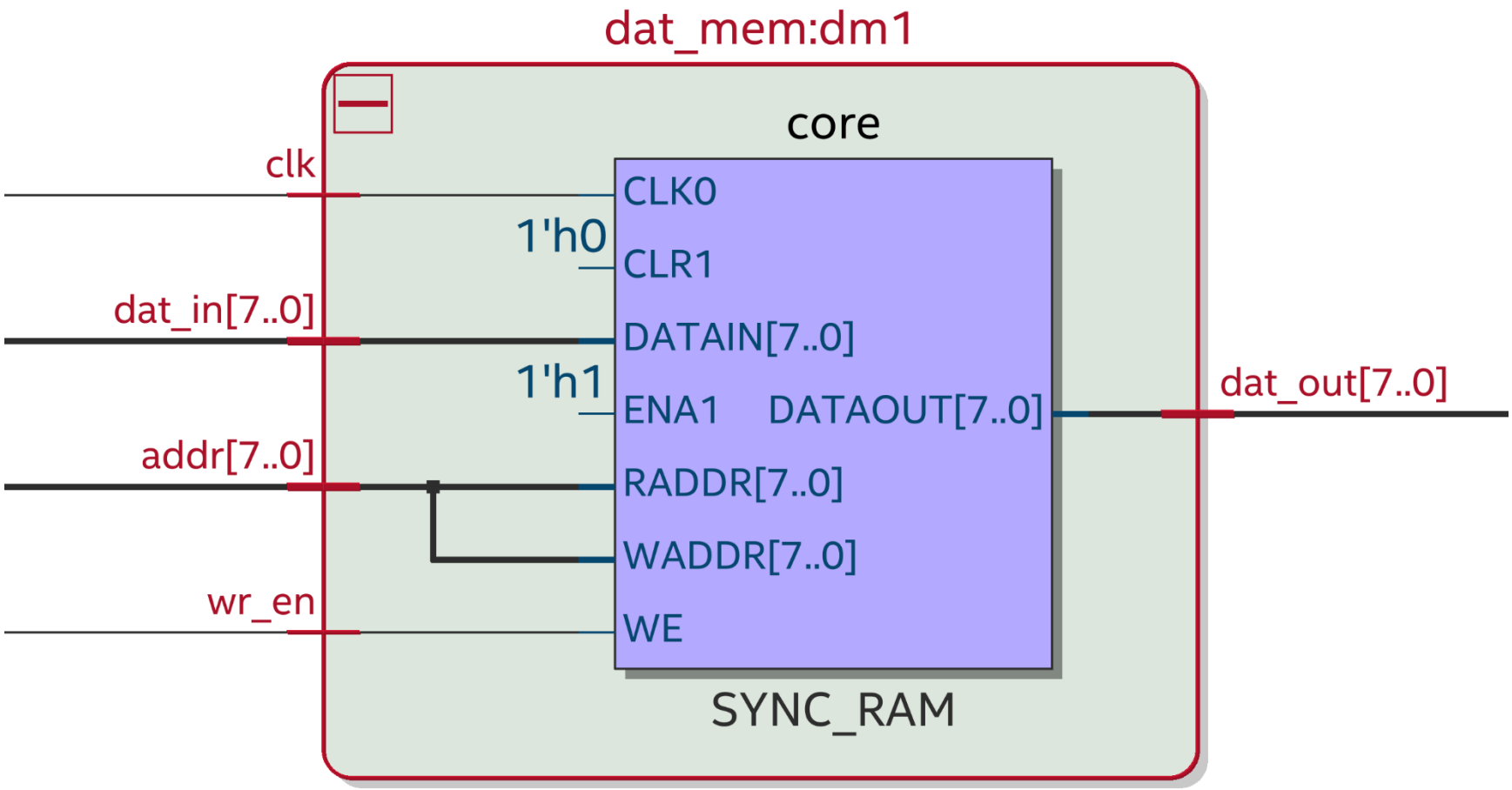
Data Memory

Module file name: dat_mem.sv

Functionality Description

Save data. Both provided data and processed data. Some might be used for frame to process function call

Schematic



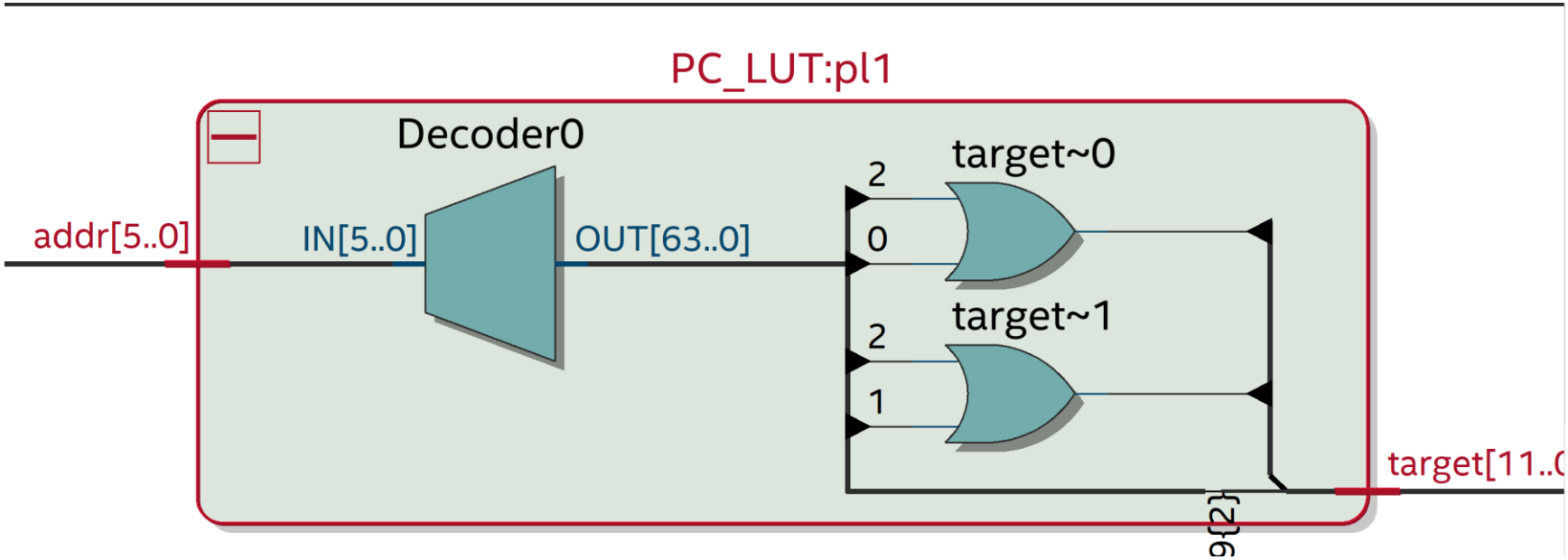
Lookup Tables

Module file name: PC_LUT.sv

Functionality Description

Lookup table for PC. Given a number, return a absolute address.
Function like labels in other machine. Here we pre-decode “label” into corresponding absolute address.

Schematic (Will be more implemented in milestone 3)



Muxes (Multiplexers)

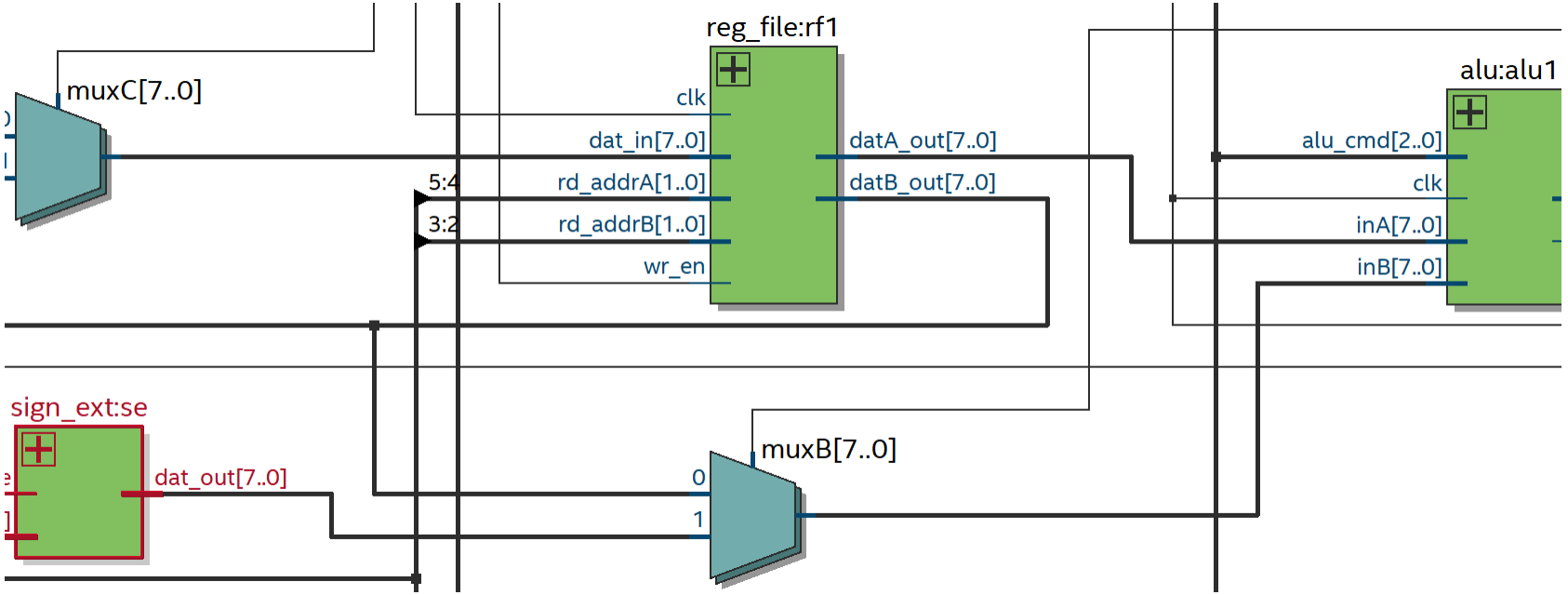
Module file name: No Name. Use *condition? result1 : result2* to achieve.

Functionality Description

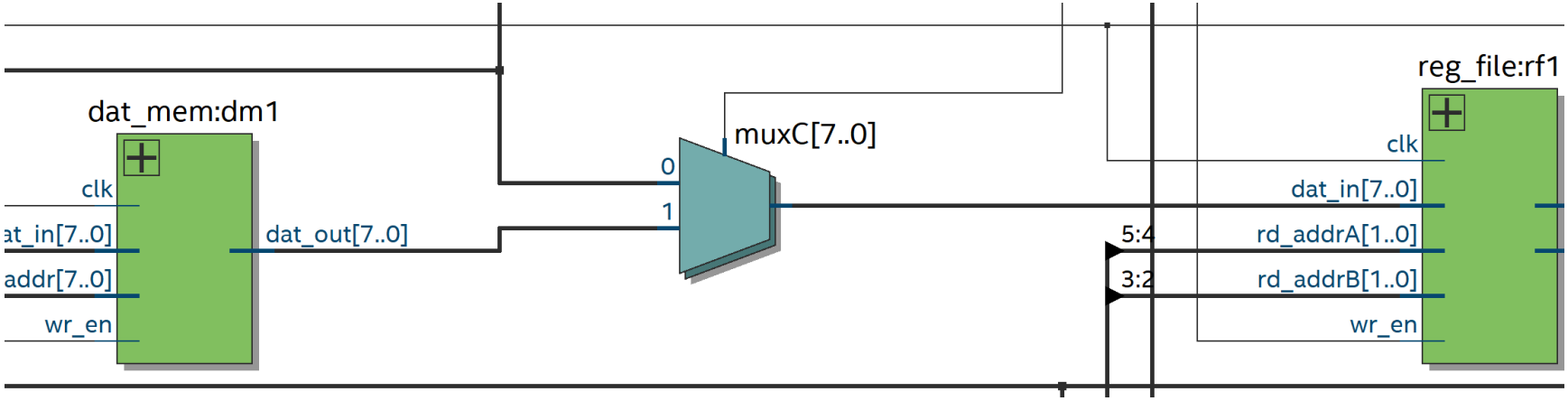
Select between two inputs, assign one of them to output.

Schematic

MUXB:



MUXC:



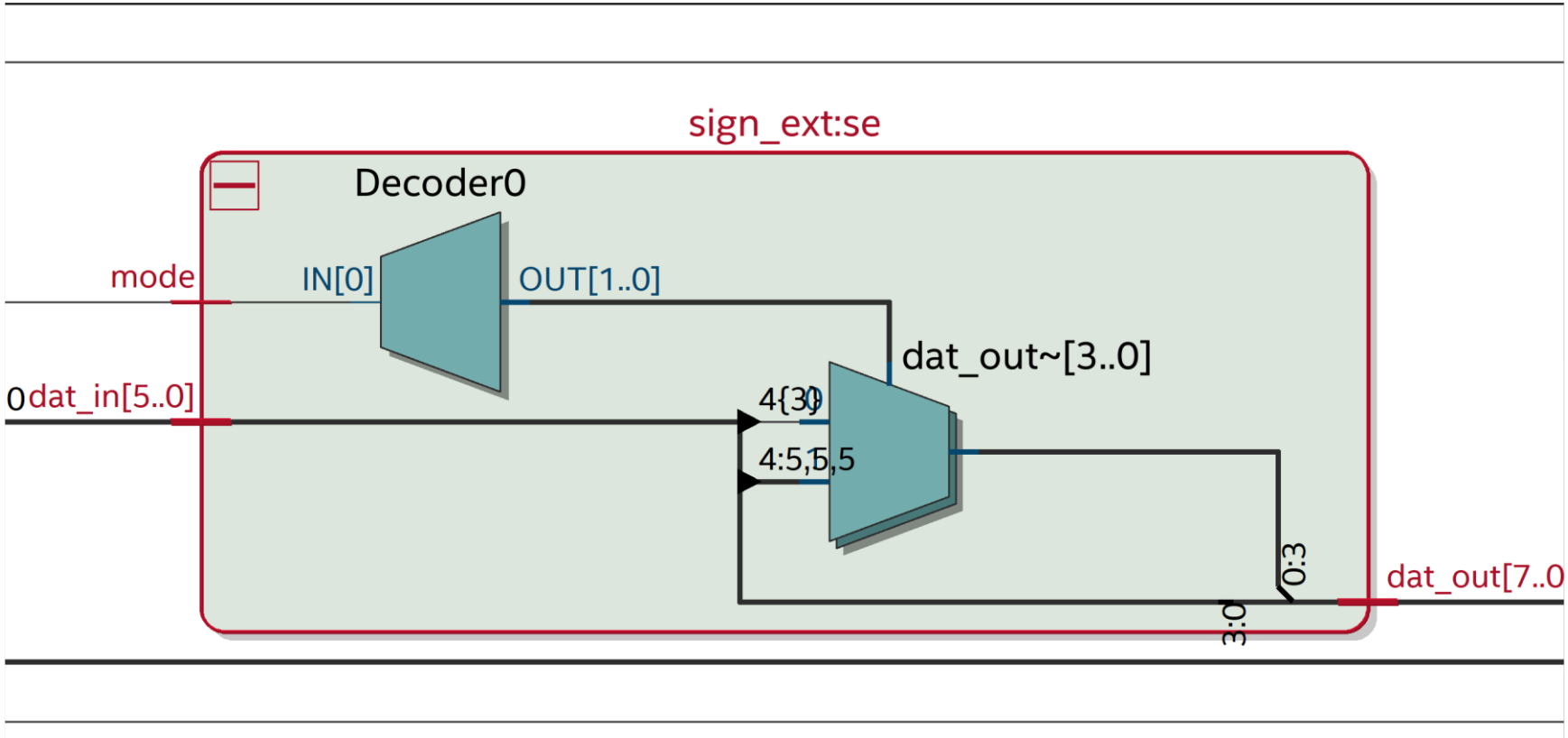
Sign Extension

Module file name: sign_ext.sv

Functionality Description

Sign extent immed6 to proper values
For MOV, LS, ADDI, since they only support immed4, sign_ext will extend bits based on mach_code[3]
For BR, since it support immed6, sign_ext will extend bits based on mach_code[5]

Schematic



6. Program Implementation

Example Pseudocode

```
// just initial R0, R1 to 1 and 2 respectively
// Add them up and saved the updated result to R2

R0 = 1;
R1 = 2;
R2 = R0 + R1;
```

Example Assembly Code

```
# just initial R0, R1 to 1 and 2 respectively
# Add them up and saved the updated result to R2

MOV R0, 1        // R0 = 1;
MOV R1, 2        // R1 = 2;
ADR R0, R1       // R0 = R0 + R1;
MOV R2, 0        // clear R2 to hold the sum
ADR R2, R0       // R2 = R1 (i.e. R2 = R0 + R1)
```

C++ Assembler

```
#include <bitset>
#include <fstream>
```

```cpp
#include <iostream>

using namespace std;

int main() {
  ifstream input(
      "input.txt"); // Replace "input.txt" with the actual input file name
  ofstream output("mach_code.txt");

  if (!input.is_open()) {
    cerr << "Error opening input file." << std::endl;
    return 1;
  }

  if (!output.is_open()) {
    cerr << "Error opening output file." << std::endl;
    return 1;
  }

  string line;
  while (getline(input, line)) {
    string op = line.substr(0, 3);
    string mach = "";

    if (op != "BRC") {
      string op1 = line.substr(5, 1);
      string op2 = line.substr(8, 2);
      int num1 = atoi(op1.c_str());
      bitset<2> BR1(num1);
      string b1 = BR1.to_string();
      if (op == "LDR") {
        mach += "000";
```

```
        int num2 = atoi(op2.substr(1).c_str());
        bitset<2> BR2(num2);
        string b2 = BR2.to_string();
        mach += b1;
        mach += b2;
        mach += "00";
      } else if (op == "STR") {
        mach += "001";
        int num2 = atoi(op2.substr(1).c_str());
        bitset<2> BR2(num2);
        string b2 = BR2.to_string();
        mach += b2;
        mach += b1;
        mach += "00";
      } else if (op == "XOR") {
        mach += "011";
        int num2 = atoi(op2.substr(1).c_str());
        bitset<2> BR2(num2);
        string b2 = BR2.to_string();
        mach += b1;
        mach += b2;
        mach += "00";
      } else if (op == "ADR") {
        mach += "010";
        int num2 = atoi(op2.substr(1).c_str());
        bitset<2> BR2(num2);
        string b2 = BR2.to_string();
        mach += b1;
        mach += b2;
        mach += "00";
      } else if (op == "MOV") {
        mach += "100";
```

```
      int num2 = atoi(op2.c_str());
      bitset<4> BR2(num2);
      string b2 = BR2.to_string();
      mach += b1;
      mach += b2;
    } else if (op == "LSH") {
      mach += "101";
      int num2 = atoi(op2.c_str());
      bitset<4> BR2(num2);
      string b2 = BR2.to_string();
      mach += b1;
      mach += b2;
    } else if (op == "ADI") {
      mach += "110";
      int num2 = atoi(op2.c_str());
      bitset<4> BR2(num2);
      string b2 = BR2.to_string();
      mach += b1;
      mach += b2;
    }
  } else if (op == "BRC") {
    mach += "111";
    string op1 = line.substr(4, 3);
    int num1 = atoi(op1.c_str());
    bitset<6> BR1(num1);
    string b1 = BR1.to_string();
    mach += b1;
  } else {
    cerr << "Instruction wrong" << endl;
    return 1;
  }
  output << mach << endl;
```

```
  }

  input.close();
  output.close();
  return 0;
}
```

Program 1 Pseudocode

```
For (int I = 0; I < 30; I = I + 2) {
X = mem[I + 1];
X <<= 5;                // X=KJI00000
Y = mem[I];
Y >>= 4;                // Y=0000HGFE
Y <<= 1;
X ^= Y;                 // X=X XOR Y
                        // X=KJI0HGFE
X += Q;                 // Q is the redundant XOR or ALU-result. So here it's ^(KJIHGFE0)
                        // X=KJIHGFEp_8
mem[I + 31] = X;        // first half done!
X = Mem[I];
X >>= 1;
X <<= 5;                // X=DCB00000
A = mem[I+1];
A <<= 5;                // A=KJI00000
B = mem[I];
B >>= 7;                // B=0000000H
A ^= B;                 // A=KJI0000H    Same. Odd/Even matter!
B = mem[I];
```

```
B <<= 4;
B >>= 5;
B <<= 1;              // B=0000DCB0
A ^= B;               // A=KJI0DCBH
A = Q;                // A=00000000p_4
A <<= 4;              // A=000P_40000
X ^= A;               // X=DCBp_40000
A = mem[I];
A <<= 7;
A >>= 4;              // A=0000A000
X ^= A;               // X=DCBp_4A000
A = mem[I];
A >>= 1;
A <<= 6;              // A=KJ000000
B = mem[I];
B <<= 1;
B >>= 6;              // B=000000GF
A ^= B;               // A=KJ0000GF
B = mem[I];
B <<= 4;
B >>= 6;
B <<= 2;              // B=0000DC00
A ^= B;               // A=KJ00DCGF
B = mem[I];
B <<= 7;
B >>= 2;              // B=00A00000
A ^= B;               // A=KJA0DCGF
A = Q;                // A=0000000p_2
A <<= 2;              // A=00000p_200
X ^= A;               // X=DCBp_4Ap_200
A = mem[I+1];
A >>= 2;
```

```
A <<= 7;              // A=K0000000
B = mem[I+1];
B <<= 7;
B >>= 1;              // B=0I000000
A ^= B;               // A=KI000000
B = mem[I];
B <<= 1;
B >>= 7;
B <<= 5;              // B=00G00000
A ^= B;               // A=KIG00000
B = mem[I];
B <<= 3;
B >>= 6;
B <<= 3;              // B=000DE000
A ^= B;               // A=KIGDE000
B = mem[I];
B <<= 6;
B >>= 5;              // B=00000BC0
A ^= B;               // A=KIGDEBC0
A = Q;                // A=0000000p_1
A <<= 1;              // A=000000p_10
X ^= A;               // X=DCBp_4Ap_2p_10
A = mem[I+31];        // A=KJIHGFEp_8
A ^= X;               // A=(D^k)(C^J)(B^I)(p_4^H)(A^G)(p_2^F)(p_1^E)p_8
                      // Q=p_0
B = Q;
X += Q;               // X=DCBp_4Ap_2p_1p_0
Mem[I+30] = X;        // done!
}
```

Program 1 Assembly Code

```
MOV R1, 0        // I = 0
MOV R0, 0        // Branch 1
ADR R0, R1       // X = I
ADI R0, -8
ADI R0, -8
ADI R0, -8
ADI R0, -6       // I - 30 == 0?
BRC 0            // jump to end (4095)
MOV R0, 0        // X = R0, I = R1
ADR R0, R1
ADI R0, 1             // X = I + 1
LDR R0, R0       // X = MEM[I + 1];
LSH R0, 5             // X <<= 5;
MOV R2, 0
ADR R2, R1       // A = I
LDR R2, R2       // A = MEM[I];
LSH R2, -4       // A >>= 4;
LSH R2, 1             // A <<= 1;
XOR R0, R2       // X ^= A;
MOV R2, -8       // get Q, A = Q
ADR R0, R2       // X += A(Q)
MOV R2, 0
ADR R2, R1       // A = I
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 3             // A = I + 31
STR R0, R2       // MEM[I +31] = X, done first hafl
MOV R0, 0
```

```
ADR R0, R1
LDR R0, R0       // X = MEM[I]
LSH R0, -1       // X >>= 1
LSH R0, 5             // X <<= 5
MOV R2, 0        // A = R2, B = R3
ADR R2, R1       // A = I
ADI R2, 1             // A = I + 1
LDR R2, R2       // A = MEM[I + 1]
LSH R2, 5             // A <<= 5
MOV R3, 0
ADR R3, R1
LDR R3, R3       // B = MEM[I]
LSH R3, -7       // B >>= 7
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
LDR R3, R3       // B = MEM[I]
LSH R3, 4             // B <<= 4
LSH R3, -5       // B >>= 5
LSH R3, 1             // B <<= 1
XOR R2, R3       // A ^= B
MOV R2, -8       // A = Q
LSH R2, 4             // A <<= 4
XOR R0, R2       // X ^= A
MOV R2, 0
ADR R2, R1
LDR R2, R2       // A = MEM[I]
LSH R2, 7             // A <<= 7
LSH R2, -4       // A >>= 4
XOR R0, R2       // X ^= A
MOV R2, 0
ADR R2, R1
```

```
ADI R2, 1
LDR R2, R2       // A = MEM[I + 1]
LSH R2, -1       // A >>= 1
LSH R2, 6             // A <<= 6
MOV R3, 0
ADR R3, R1
LDR R3, R3       // B = MEM[I]
LSH R3, 1             // B <<= 1
LSH R3, -6       // B >>= 6
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
LDR R3, R3       // B = MEM[I]
LSH R3, 4             // B <<= 4
LSH R3, -6       // B >>= 6
LSH R3, 2             // B <<= 2
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
LDR R3, R3       // B = MEM[I]
LSH R3, 7             // B <<= 7
LSH R3, -2       // B >>= 2
XOR R2, R3       // A ^= B
MOV R2, -8       // A = Q
LSH R2, 2             // A <<= 2
XOR R0, R2       // X ^= A
MOV R2, 0
ADR R2, R1
ADI R2, 1
LDR R2, R2       // A = MEM[I + 1]
LSH R2, -2       // A >>= 2
LSH R2, 7             // A <<= 7
```

```
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
LSH R3, 7             // B <<= 7
LSH R3, -1       // B >>= 1
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
LDR R3, R3       // B = MEM[I];
LSH R3, 1             // B <<= 1
LSH R3, -7       // B >>= 7
LSH R3, 5             // B <<= 5
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
LDR R3, R3       // B = MEM[I]
LSH R3, 3             // B <<= 3
LSH R3, -6       // B >>= 6
LSH R3, 3             // B <<= 3
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
LDR R3, R3       // B = MEM[I]
LSH R3, 6             // B <<= 6
LSH R3, -5       // B >>= 5
XOR R2, R3       // A ^= B
MOV R2, -8       // A = Q
LSH R2, 1             // A <<= 1
XOR R0, R2       // X ^= A
MOV R2, 0
ADR R2, R1
```

```
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 3
LDR R2, R2       // A = MEM[I + 31]
XOR R2, R0       // A ^= X
MOV R3, -8       // B = Q
ADR X0, R3       // A += B(Q)
MOV R2, 0
ADR R2, R1
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 2
STR R0, R2       // MEM[I + 30] = X
ADI R1, 2             // I = I + 2
MOV R0, 0        // Make branch
BRC 1            // branch to loop (1)
```

Program 2 Pseudocode

// X = R0, I = R1, A = R2, B = R3

```
For (int I = 30; I < 60; I = I + 2) {
     X = 0;          // find which bit is flipped
     // check p8
     B = MEM[I + 1]
     B <<= 7;
     B >>= 7;              // B = 0000000P8
```

```
A = MEM[I + 1];
A >>= 1;              // A = 0, B11, B10, …, B5
A = Q;                // A = PARITY
A ^= B;
IF (A != 0) X += 8;
// check p4
A = MEM[I];
A >>= 5;              // A = 00000432
B = MEM[I + 1];
B >>= 4;
B <<= 4;              // B = 11, 10, 9, 8, 0000
A ^= B;               // A = 11 10  9  8  0  4  3  2
A = Q;                // A = P4
B = MEM[I];
B <<= 3;
B >>= 7;              // B = P4
A ^= B;
IF (A != 0) X += 4;
// check p2
A = MEM[I + 1];
A >>= 6;              // A = 0 0 0 0 0 0 11 10
B = MEM[I + 1];
B >>= 2;
B <<= 6;              // B = 7 6 0 0 0 0  0  0
A ^= B;               // A = 7 6 0 0 0 0 11 10
B = MEM[I];
B >>= 6;
B <<= 4;              // B = 0 0 0 0 4 3  0  0
A ^= B;               // A = 7 6 0 0 4 3 11 10
B = MEM[I];
B <<= 4;
B >>= 7;
```

```
B <<= 4;              // B = 0 0 0 1 0 0  0  0
A ^= B;               // A = 7 6 0 1 4 3  11 10
A = Q;                // A = P2
B = MEM[I];
B <<= 5;
B >>= 7;              // B = P2
A ^= B;
IF (A != 0) X += 2;
// check p1
A = MEM[I + 1];
A >>= 7;              // A = 0 0 0 0 0 0 0 11
B = MEM[I + 1];
B >>= 5;
B <<= 7;              // B = 9 0 0 0 0 0 0  0
A ^= B;               // A = 9 0 0 0 0 0 0 11
B = MEM[I + 1];
B <<= 4;
B >>= 7;
B <<= 6;              // B = 0 7 0 0 0 0 0  0
A ^= B ;              // A = 9 7 0 0 0 0 0 11
B = MEM[I + 1];
B <<= 6;
B >>= 7;
B <<= 5;              // B = 0 0 5 0 0 0 0  0
A ^= B;               // A = 9 7 5 0 0 0 0 11
B = MEM[I];
B >>= 7;
B <<= 4;              // B = 0 0 0 4 0 0 0 0
A ^= B;               // A = 9 7 5 4 0 0 0 11
B = MEM[I];
B <<= 2;
B >>= 7;
```

```
B <<= 3;              // B = 0 0 0 0 2 0 0 0
A ^= B ;              // A = 9 7 5 4 2 0 0 11
B = MEM[I];
B <<= 4;
B >>= 7;
B <<= 2;              // B = 0 0 0 0 0 1 0  0
A ^= B;               // A = 9 7 5 4 2 1 0 11
A = Q;                // A = P1
B = MEM[I];
B <<= 6;
B >>= 7;              // B = P1
A ^= B;
If( A != 0) X += 1;
// x tells which bit is wrong. 11 cases need to flip data bit

// check p0
A = MEM[I];
B = MEN[I + 1];
B >>= 1;
A ^= B;
A = Q;                // get parity
B = MEM[I + 1];
B <<= 7;
B >>= 7;              // get given parity
A ^= B;               // if parity is correct, result = 0; else 1
IF (A != 0) {         // SEC
      A = X;
      IF ((A - 15) == 0) {
            A = 1;
            A <<= 7;         // A = 11 0 0 0 0 0 0 0
            B = MEM[I + 1];
            A ^= B           // A = 11' 10 9 8 7 6 5 P8
```

```
      MEM[I + 1] = A;  // store back. Correct data
}
ELSE IF (A == 14) {
      A = 1
      A <<= 6;
      B = MEM[I + 1];
      A ^= B;
      MEM[I + 1] = A;
}
ELSE IF (A == 13) {
      A = 1
      A <<= 5;
      B = MEM[I + 1];
      A ^= B;
      MEM[I + 1] = A;
}
ELSE IF (A == 12) {
      A = 1
      A <<= 4;
      B = MEM[I + 1];
      A ^= B;
      MEM[I + 1] = A;
}
ELSE IF (A == 11) {
      A = 1
      A <<= 3;
      B = MEM[I + 1];
      A ^= B;
      MEM[I + 1] = A;
}
ELSE IF (A == 10) {
      A = 1
```

```
      A <<= 2;
      B = MEM[I + 1];
      A ^= B;
      MEM[I + 1] = A;
}
ELSE IF (A == 9) {
      A = 1
      A <<= 1;
      B = MEM[I + 1];
      A ^= B;
      MEM[I + 1] = A;
}
ELSE IF (A == 7) {
      A = 1
      A <<= 7;          // A = 4 0 0 0 0 0 0 0
      B = MEM[I];
      A ^= B;           // A = 4' 3 2 P4 1 P2 P1 P0
      MEM[I] = A;
}
ELSE IF (A == 6) {
      A = 1
      A <<= 6;
      B = MEM[I];
      A ^= B;
      MEM[I] = A;
}
ELSE IF (A == 5) {
      A = 1
      A <<= 5;
      B = MEM[I];
      A ^= B;
      MEM[I] = A;
```

```
        }
        ELSE IF (A == 3) {
                A = 1
                A <<= 3;
                B = MEM[I];
                A ^= B;
                MEM[I] = A;
        }
        // single error correction done here. Load upper half and set flag
        A = 1;
        A <<= 6;                // f1f0 = 01
        B = MEM[I + 1];
        B >>= 5;
        A ^= B;                 // A = 0 1 0 0 0 11 10 9
        MEM[I - 29] = A;
}
ELSE {      // DED
        A = X;
        IF (A == 0){     // MESSAGE CLEAN
                A = MEM[I + 1];
                A >>= 5;
                MEM[I - 29] = A;
        }
        ELSE {
                A = 2;           // A = 0 0 0 0 0 0 1 0
                A <<= 6;         // F1F0 = 10
                B = MEM[I + 1];
                B >>= 5;
                A ^= B;          // A = 1 0 0 0 0 11 10 9
                MEM[I - 29] = A;
        }
}
```

```
    // load lower half
    A = MEM[I + 1];
    A >>= 1;
    A <<= 4;                    // A = 8 7 6 5 0 0 0 0
    B = MEM[I];
    B >>= 5;
    B <<= 1;
    A ^= B;                     // A = 8 7 6 5 4 3 2 0
    B = MEM[I];
    B <<= 4;
    B >>= 7;
    A ^= B;                     // A = 8 7 6 5 4 3 2 1
    MEM[I - 30] = A;
}
```

Program 2 Assembly Code

```
MOV R1, 7
ADI R1, 7
ADI R1, 7
ADI R1, 7
ADI R1, 2              // I = 30;
MOV R2, 0        // dest 21
ADR R2, R1       // A = I
ADI R2, -7
ADI R2, -7
ADI R2, -7
ADI R2, -7
ADI R2, -7
ADI R2, -7
ADI R2, -7
ADI R2, -7
```

```
ADI R2, -4       // A - 60 == 0?
BRC 0            // I = 60, END
MOV R0, 0        // X = 0
MOV R3, 0        // chekc p8
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
LSH R3, 7             // B <<= 7
LSH R3, -7       // B >>= 7
MOV R2, 0
ADR R2, R1
ADI R2, 1
LDR R2, R2       // A = MEM[I + 1]
LSH R2, -1       // A >>= 1
MOV R2, -8       // A = Q
XOR R2, R3       // A ^= B
BRC 2            // A == 0  ; A != 0 exe following
ADI R0, 7
ADI R0, 1             // X += 8
LDR R2, R1       // 2 dest. chek p4
LSH R2, -5       // A >>= 5
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
LSH R3, -4       // B >>= 4
LSH R3, 4             // B <<= 4
XOR R2, R3       // A ^= B
MOV R2, -8       // A = Q
LDR R3, R1       // B = MEM[I]
LSH R3, 3             // B <<= 3
LSH R3, -7       // B >>= 7
```

```
XOR R2, R3      // A ^= B
BRC 3           // A == 0; A != 0 exe following
ADI R0, 4            // X += 4
MOV R2, 0       // 3 dest. check p2
ADR R2, R1
ADI R2, 1
LDR R2, R2      // A = MEM[I + 1]
LSH R2, -6      // A >>= 6
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3      // B = MEM[I + 1]
LSH R3, -2      // B >>= 2
LSH R3, 6            // B <<= 6
XOR R2, R3      // A ^= B
LDR R3, R1      // B = MEM[I];
LSH R3, -6      // B >>= 6
LSH R3, 4            // B <<= 4
XOR R2, R3      // A ^= B
LDR R3, R1      // B = MEM[I]
LSH R3, 4            // B <<= 4
LSH R3, -7      // B >>= 7
LSH R3, 4            // B <<= 4
XOR R2, R3      // A ^= B
MOV R2, -8      // A = Q
LDR R3, R1      // B = MEM[I]
LSH R3, 5            // B <<= 5;
LSH R3, -7      // B >>= 7
XOR R2, R3      // A ^= B
BRC 4           // A == 0; A != 0 exe following
ADI R0, 2
MOV R2, 0       // 4 dest. check p1
```

```
ADR R2, R1
ADI R2, 1
LDR R2, R2       // A = MEM[I + 1]
LSH R2, -7       // A >>= 7
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I+ 1]
LSH R3, -5       // B >>= 5
LSH R3, 7             // B <<= 7
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
LSH R3, 4             // B <<= 4
LSH R3, -7       // B >>= 7
LSH R3, 6             // B <<= 6
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
LSH R3, 6             // B <<= 6
LSH R3, -7       // B >>= 7
LSH R3, 5             // B <<= 5
XOR R2, R3       // A ^= B
LDR R3, R1       // B = MEM[I]
LSH R3, -7       // B >>= 7
LSH R3, 4             // B <<= 4
XOR R2, R3       // A ^= B
LDR R3, R1       // B = MEM[I]
```

```
LSH R3, 2             // B <<= 2
LSH R3, -7       // B >>= 7
LSH R3, 3             // B <<= 3
XOR R2, R3       // A ^= B
LDR R3, R1       // B = MEM[I]
LSH R3, 4             // B <<= 4
LSH R3, -7       // B >>= 7
LSH R3, 2             // B <<= 2
XOR R2, R3       // A ^= B
MOV R2, -8       // A = Q
LDR R3, R1       // B = MEM[I]
LSH R3, 6             // B <<= 6
LSH R3, -7       // B >>= 7
XOR R2, R3
BRC 5            // A == 0; A !=0 exe following
ADI R0, 1             // X += 1
LDR R2, R1       // 5 dest. check p0        A = MEM[I]
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
LSH R3, -1       // B >>= 1
XOR R2, R3       // A ^= B
MOV R2, -8       // A = Q
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
LSH R3, 7             // B <<= 7
LSH R3, -7       // B >>= 7
XOR R2, R3       // A ^= B
BRC 6            // A == 0,  go to DED
```

```
MOV R2, 0
ADR R2, R0      // A = X
ADI R2, -8
ADI R2, -7      // A - 15 == 0?
BRC 8
MOV R2, 0
ADR R2, R0      // A = X
ADI R2, -7
ADI R2, -7      // A - 14 == 0?
BRC 9
MOV R2, 0
ADR R2, R0      // A = X
ADI R2, -7
ADI R2, -6      // A - 13 == 0?
BRC 10
MOV R2, 0
ADR R2, R0      // A = X
ADI R2, -7
ADI R2, -5      // A - 12 == 0?
BRC 11
MOV R2, 0
ADR R2, R0      // A = X
ADI R2, -6
ADI R2, -5      // A - 11 == 0?
BRC 12
MOV R2, 0
ADR R2, R0      // A = X
ADI R2, -5
ADI R2, -5      // A - 10 == 0?
BRC 13
MOV R2, 0
ADR R2, R0      // A = X
```

```
ADI R2, -4
ADI R2, -5       // A - 9 == 0?
BRC 14
MOV R2, 0
ADR R2, R0       // A = X
ADI R2, -7       // A - 7 == 0?
BRC 15
MOV R2, 0
ADR R2, R0       // A = X
ADI R2, -6       // A - 6 == 0?
BRC 16
MOV R2, 0
ADR R2, R0       // A = X
ADI R2, -5       // A - 5 == 0?
BRC 17
MOV R2, 0
ADR R2, R0       // A = X
ADI R2, -3       // A - 3 == 0?
BRC 18
MOV R2, 0
BRC 7            // if no case match. parity error. just exit
MOV R2, 1        // 8 dest. A == 15
LSH R2, 7             // A <<= 7
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
ADI R3, 1             // B = I + 1
STR R2, R3       // MEM[I + 1] = A
```

```
MOV R2, 0        // for BRC purpose
BRC 7            // correction done, exit to 7
MOV R2, 1        // 9 dest
LSH R2, 6              // A <<= 6
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
ADI R3, 1              // B = I + 1
STR R2, R3       // MEM[I + 1] = A
MOV R2, 0
BRC 7            // exit to 7
MOV R2, 1        // 10 dest
LSH R2, 5
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3
XOR R2, R3
MOV R3, 0
ADR R3, R1
ADI R3, 1
STR R2, R3
MOV R2, 0
BRC 7
MOV R2, 1        // 11 dest
LSH R2, 4
MOV R3, 0
ADR R3, R1
```

```
ADI R3, 1
LDR R3, R3
XOR R2, R3
MOV R3, 0
ADR R3, R1
ADI R3, 1
STR R2, R3
MOV R2, 0
BRC 7
MOV R2, 1        // 12 dest
LSH R2, 3
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3
XOR R2, R3
MOV R3, 0
ADR R3, R1
ADI R3, 1
STR R2, R3
MOV R2, 0
BRC 7
MOV R2, 1        // 13 dest
LSH R2, 2
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3
XOR R2, R3
MOV R3, 0
ADR R3, R1
ADI R3, 1
```

```
STR R2, R3
MOV R2, 0
BRC 7
MOV R2, 1        // 14 dest
LSH R2, 1
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3
XOR R2, R3
MOV R3, 0
ADR R3, R1
ADI R3, 1
STR R2, R3
MOV R2, 0
BRC 7
MOV R2, 1        // 15 dest
LSH R2, 7             // A <<= 7
LDR R3, R1       // B = MEM[I]
XOR R2, R3       // A ^= B
STR R2, R1       // MEM[I] = A
MOV R2, 0
BRC 7
MOV R2, 1        // 16 dest
LSH R2, 6
LDR R3, R1
XOR R2, R3
STR R2, R1
MOV R2, 0
BRC 7
MOV R2, 1        // 17 dest
LSH R2, 5
```

```
LDR R3, R1
XOR R2, R3
STR R2, R1
MOV R2, 0
BRC 7
MOV R2, 1        // 18 dest
LSH R2, 3
LDR R3, R1
XOR R2, R3
STR R2, R1
MOV R2, 0
BRC 7
MOV R2, 1        // 7 dest. corret bit exit here. load upper hafl here
LSH R2, 6             // A <<= 6
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
LSH R3, -5       // B >>= 5
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
ADI R3, -8
ADI R3, -8
ADI R3, -8
ADI R3, -5       // B = I - 29
STR R2, R3       // MEM[I - 29] = A
MOV R2, 0
BRC 19                // go to load lower half
MOV R2, 0        // 6 dest. DED goes here
ADR R2, R0       // A = X
BRC 20                // A == 0, message clean
```

```
MOV R2, 2        // A != 0, DED        A = 2
LSH R2, 6             // A <<= 6
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3       // B = MEM[I + 1]
LSH R3, -5       // B >>= 5
XOR R2, R3       // A ^= B;
MOV R3, 0
ADR R3, R1
ADI R3, -8
ADI R3, -8
ADI R3, -8
ADI R3, -5       // B = I - 29
STR R2, R3       // MEM[I - 29] = A
MOV R2, 0
BRC 19                // go to load lower half
MOV R2, 0        // 20 dest
ADR R2, R1
ADI R2, 1
LDR R2, R2       // A = MEM[I + 1]
LSH R2, -5       // A >>= 5
MOV R3, 0
ADR R3, R1
ADI R3, -8
ADI R3, -8
ADI R3, -8
ADI R3, -5       // B = I - 29
STR R2, R3       // MEM[I - 29] = A.    Then directly go to load lower half
MOV R2, 0        // 19 dest. load lower half
ADR R2, R1
ADI R2, 1
```

```
LDR R2, R2       // A = MEM[I + 1]
LSH R2, -1       // A >>= 1
LSH R2, 4             // A <<= 4
LDR R3, R1       // B = MEM[I]
LSH R3, -5       // B >>= 5
LSH R3, 1             // B <<= 1
XOR R2, R3       // A ^= B
LDR R3, R1       // B = MEM[I]
LSH R3, 4             // B <<= 4
LSH R3, -7       // B >>= 7
XOR R2, R3       // A ^= B
MOV R3, 0
ADR R3, R1
ADI R3, -8
ADI R3, -8
ADI R3, -8
ADI R3, -6       // B = I - 30
STR R2, R3       // MEM[I - 30] = A
ADI R1, 2             // I = I + 2
MOV R2, 0
BRC 21                // go back to loop
```

Program 3 Pseudocode

```
MEM[33] = 0;
MEM[34] = 0;
```

```
MEM[35] = 0;
X = MEM[32];
X >>= 3;
For (INT I = 0; I < 31; ++i) {
      B = 0;     // used for count if this byte include pattern
      A = MEM[I];
      A >>= 3;
      A ^= X;
      IF (A == 0) {    // IF [7:3] match pattern
            B++;
      }
      A = MEM[I];
      A <<= 1;
      A >>= 3;
      A ^= X;
      IF (A == 0) {    // IF [6:2] match pattern
            B++;
      }
      A = MEM[I];
      A <<= 2;
      A >>= 3;
      A ^= X;
      IF (A == 0) {    // IF [5:1] match pattern
            B++;
      }
      A = MEM[I];
      A <<= 3;
      A >>= 3;
      A ^= X;
      IF (A == 0) {    // if [4:0] match pattern
            B++
      }
```

```
IF (B != 0) {    // this byte contains pattern
      MEM[33] += B;
      MEM[34] ++;
      MEM[35] += B;
}
// processing cross byte boundary case
// [3:0] + [7]
A = MEM[I];
A <<= 4;
A >>= 3;
B = MEM[I + 1];
B >>= 7;
A ^= B;
A ^= X;
IF (A == 0) MEM[35]++;
// [2:0] + [7:6]
A = MEM[I];
A <<= 5;
A >>= 3;
B = MEM[I + 1];
B >>= 6;
A ^= B;
A ^= X;
IF (A == 0) MEM[35]++;
// [1:0] + [7:5]
A = MEM[I];
A <<= 6;
A >>= 3;
B = MEM[I + 1];
B >>= 5;
A ^= B;
A ^= X;
```

```
      IF (A == 0) MEM[35]++;
      //[0] + [7:4]
      A = MEM[I];
      A <<= 7;
      A >>= 3;
      B = MEM[I + 1];
      B >>= 4;
      A ^= B;
      A ^= X;
      IF (A == 0) MEM[35]++;
}

// now just need to process mem[31]
      B = 0;
      A = MEM[I];      // I = 31 now
      A >>= 3;
      A ^= X;
      IF (A == 0) {    // IF [7:3] match pattern
            B++;
      }
      A = MEM[I];
      A <<= 1;
      A >>= 3;
      A ^= X;
      IF (A == 0) {    // IF [6:2] match pattern
            B++;
      }
      A = MEM[I];
      A <<= 2;
      A >>= 3;
      A ^= X;
      IF (A == 0) {    // IF [5:1] match pattern
```

```
        B++;
    }
    A = MEM[I];
    A <<= 3;
    A >>= 3;
    A ^= X;
    IF (A == 0) {    // if [4:0] match pattern
        B++
    }
    IF (B != 0) {    // this byte contains pattern
        MEM[33] += B;
        MEM[34] ++;
        MEM[35] += B;
    }
```

Program 3 Assembly Code

```
MOV R0, 0  // X = 0. For mem[33:35] initialization
MOV R1, 7
ADI R1, 7
ADI R1, 7
ADI R1, 7
ADI R1, 5
STR R0, R1 // MEM[33] = 0
ADI R1, 1
STR R0, R1 // MEM[34] = 0
ADI R1, 1
STR R0, R1 // MEM[35] = 0
```

```
ADI R1, -3
LDR R1, R1
ADR R0, R1 // X = MEM[32]
LSH R0, -3 // X >>= 3
MOV R1, 0
MOV R2, 0  // 22 dest
ADR R2, R1
ADI R2, -7
ADI R2, -7
ADI R2, -7
ADI R2, -7
ADI R2, -3 // I - 31 == 0?
BRC 23
MOV R3, 0  // B = 0;
LDR R2, R1 // A = MEM[I]
LSH R2, -3 // A >>= 3
XOR R2, R0 // A ^= X
BRC 24          // go to B++
BRC 25          // skip B++
ADI R3, 1       // 24 dest. B++
LDR R2, R1 // 25 dest. A = MEM[I]
LSH R2, 1       // A <<= 1
LSH R2, -3 // A >>= 3
XOR R2, R0 // A ^= 0
BRC 26
BRC 27
ADI R3, 1       // 26 dest
LDR R2, R1 // 27 dest
LSH R2, 2
LSH R2, -3
XOR R2, R0
BRC 28
```

```
BRC 29
ADI R3, 1        // 28 dest
LDR R2, R1 // 29 dest.
LSH R2, 3
LSH R2, -3
XOR R2, R0
BRC 30
BRC 31
ADI R3, 1        // 30 dest
ADI R3, 0        // 31 dest
BRC 32           // If B == 0
MOV R0, 0  // tempory clean X for storing issue. Store back later
MOV R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 5
LDR R0, R2 // X = MEM[33]
ADR R0, R3 // X += B
STR R0, R2 // MEM[33] += B
ADI R2, 1
LDR R0, R2 // X = MEM[34]
ADI R0, 1        // X ++
STR R0, R2 // MEM[34]++
ADI R2, 1
LDR R0, R2 // X = MEM[35]
ADR R0, R3
STR R0, R2 // MEM[35] += B
ADI R2, -3 // A = 32
LDR R0, R2 // X = MEM[32]
LSH R0, -3 // X >>= 3
LDR R2, R1 // 32 dest. A = MEM[I]
```

```
LSH R2, 4        // A <<= 4
LSH R2, -3 // A >>= 3
MOV R3, 0
ADR R3, R1
ADI R3, 1        // B = I + 1
LDR R3, R3 // B = MEM[I + 1]
LSH R3, -7 // B >>= 7
XOR R2, R3 // A ^= B
XOR R2, R0 // A ^= X
BRC 33           // A == 0
BRC 34
MOV R2, 7  // 33 dest
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 7        // A = 35
LDR R3, R2 // B = MEM[35]
ADI R3, 1        // B++
STR R3, R2 // MEM[35] = B = MEM[35]++
LDR R2, R1 // 34 dest.
LSH R2, 5
LSH R2, -3
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3
LSH R3, -6
XOR R2, R3
XOR R2, R0
BRC 35
BRC 36
MOV R2, 7  // 35 dest
```

```
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 7
LDR R3, R2
ADI R3, 1
STR R3, R2
LDR R2, R1 // 36 dest.
LSH R2, 6
LSH R2, -3
MOV R3, 0
ADR R3, R1
ADI R3, 1
LDR R3, R3
LSH R3, -5
XOR R2, R3
XOR R2, R0
BRC 37
BRC 38
MOV R2, 7  // 37 dest
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 7
LDR R3, R2
ADI R3, 1
STR R3, R2
LDR R2, R1 // 38 dest.
LSH R2, 7
LSH R2, -3
MOV R3, 0
ADR R3, R1
```

```
ADI R3, 1
LDR R3, R3
LSH R3, -4
XOR R2, R3
XOR R2, R0
BRC 39
BRC 40
MOV R2, 7  // 39 dest
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 7
LDR R3, R2
ADI R3, 1
STR R3, R2
ADI R1, 1        // 40 dest. I++
MOV R2, 0
BRC 22           // go back to loop
MOV R3, 0  // 23 dest. Single process mem[31]. B = 0
LDR R2, R1 // A = MEM[I]
LSH R2, -3 // A >>= 3
XOR R2, R0 // A ^= X
BRC 41           // go to B++
BRC 42           // skip B++
ADI R3, 1        // 41 dest. B++
LDR R2, R1 // 42 dest. A = MEM[I]
LSH R2, 1        // A <<= 1
LSH R2, -3 // A >>= 3
XOR R2, R0 // A ^= 0
BRC 43
BRC 44
ADI R3, 1        // 43 dest
```

```
LDR R2, R1 // 44 dest
LSH R2, 2
LSH R2, -3
XOR R2, R0
BRC 45
BRC 46
ADI R3, 1        // 45 dest
LDR R2, R1 // 46 dest.
LSH R2, 3
LSH R2, -3
XOR R2, R0
BRC 47
BRC 48
ADI R3, 1        // 47 dest
ADI R3, 0        // 48 dest
BRC 0      // If B == 0. Just finish the program
MOV R0, 0  // tempory clean X for storing issue. Store back later
MOV R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 7
ADI R2, 5
LDR R0, R2 // X = MEM[33]
ADR R0, R3 // X += B
STR R0, R2 // MEM[33] += B
ADI R2, 1
LDR R0, R2 // X = MEM[34]
ADI R0, 1        // X ++
STR R0, R2 // MEM[34]++
ADI R2, 1
LDR R0, R2 // X = MEM[35]
ADR R0, R3
```

```
STR R0, R2 // MEM[35] += B
MOV R0, 0
BRC 0      // finish program either way
```

7. Changelog

- Milestone 4
  - None
- Milestone 3
  - Machine specification
    - Format operation name to 3 characters for Assembly_translate convenience.
    - Remove instruction table (image).
    - Fix update on instruction table that's left from milestone 2
  - Program Implementation
    - Update example C code and Assembly code to updated version
    - Add C++ assembly code
    - Update program 1 C code and Assembly code
    - Update program 2 C code and Assembly code
    - Update program 3 C code and Assembly code
- Milestone 2
  - Machine Specification
    - Modify instructions table in more efficient way. Remove few instructions like SUBI that is not necessary to the program.
    - Modify instruction table that LDR and STR no longer support offset2
    - Update branch method from relative jump to absolute jump
    - Update elaboration of each instruction due to modification
  - Architecture overview
    - Updated signal connections to fit changes in the modification of instructions table.
    - New component: Sign-Extent. Extend immed bits to proper value. Controlled by Controller by signal se_mode
  - Individual Component Specification

        - Initial version
    - Instruction
        - Update some instruction content.
    - Programmer’s Model [Lite]
        - Add 4.3 initial version
- Milestone 1
    - Initial version