CSE 141L Milestone 3

Fangyu Zhu, A16991018; Chi Zhang, A16346955; Sara Enkhjargal, A17619485

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.

Fangyu Zhu
Chi Zhang
Sara Enkhjargal

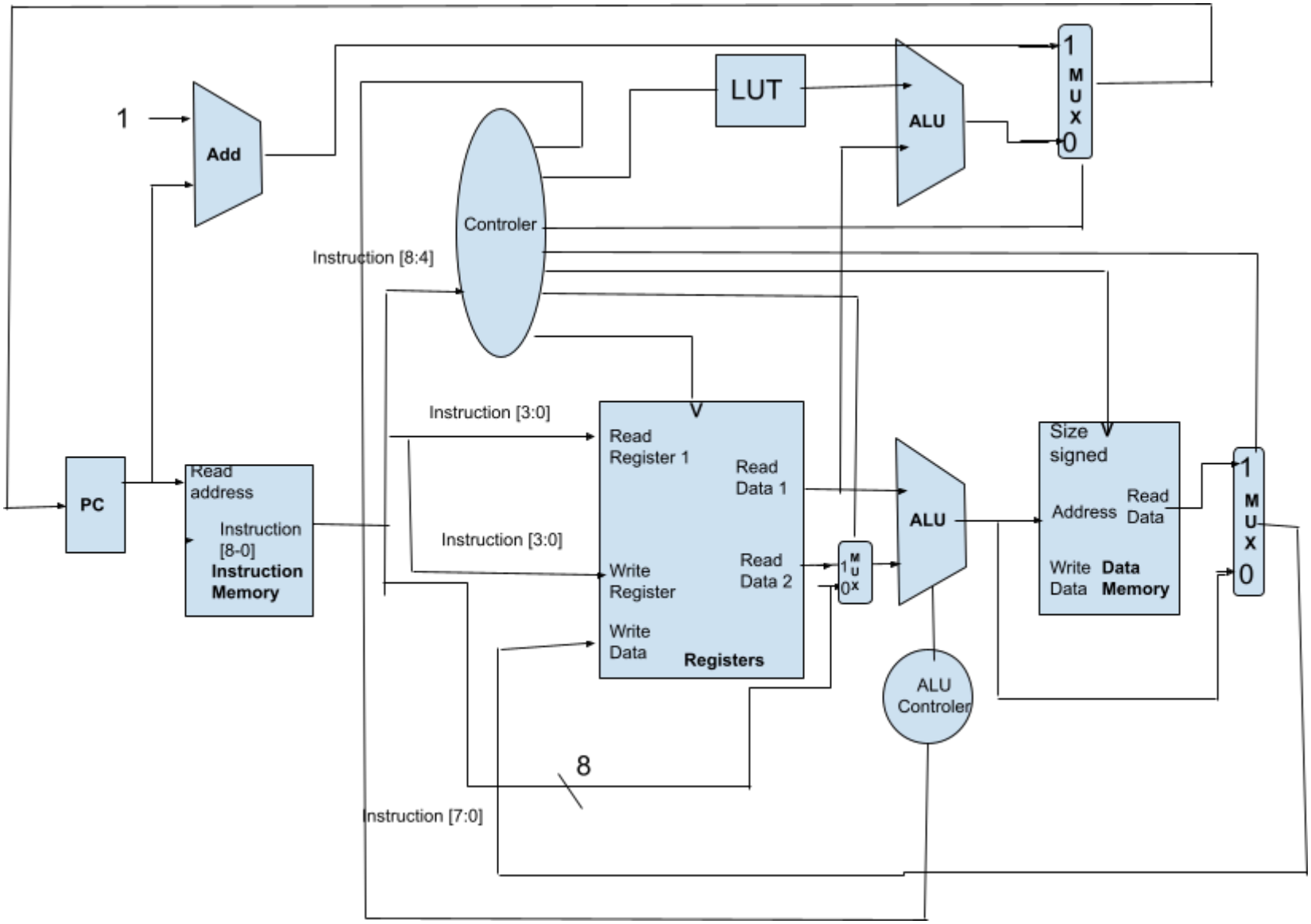
0. Team

Team members: Fangyu Zhu, Chi Zhang, and Sara Enkhjargal.

1. Introduction

Our architecture’s name is SESO (Start Early, Start Often). Our overarching philosophy was to make the algorithms for the 3 programs first. Depending on the algorithms, we find the instruction we need, and then design our instruction set. Based on the instruction set, we decided to classify our machine in the mixture of accumulator, register-register/load-store, register-memory way. Because of all the constraints, we keep optimizing the algorithm and the machine.

2. Architectural Overview



3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	1 bit type (1), 4 bits opcode, 4 bit operand register: ‘1_oooo_rrrr’ (R-type started with bit 1)	lsl, lsr, add, and, orr, eor, take, mov, ldr,str, cmp
B	1 bit type (1), 4 bits opcode, 4 bit operand register: ‘1_oooo_rrrr’ (B-type started with bit 1)	beq, bne, b
I	1 bit type (0), 8 bits 'i’ expressed by 0 or 1, ‘0_iiii_iiii’ (I-type started with bit 0)	imm ⇔ #255 (0_1111_1111)

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
imm	I	1 bit type (0), (I-type started with bit 0): ‘0_iiii_iiii’	Immediate like #1, #2…. (from #0 to #255) `imm #1`	The remaining 8 bits are all used to indicate an immediate (#0-#255).

lsl	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_0000_rrrr’	# Assume R0 has `0b0000_0011` (i here to be 3)
			# lsl r3, r0, #1 (r3 = 2 * i)
			`imm #1`
			`lsl R0`
			`mov R3`
				Because in the input memory of [0:29], there are in total 2 bytes, which are 8 bits for input MSW and another 8 bits for input LSW. Each time as for loop ‘i’ counts from 0 to 14, they would access the data memory mem[2*i] and mem[2*i+1] in order

			⇔
			`0_0000_0001`
			`1_0000_0000`
			`1_0111_0011`
			# after and instruction, R3 now holds
			`0b0000_0110`
				to load the two bytes (11-bit message in which MSW has the first 5 bits with fixed zeros). So here we might use the logical left shift in order to multiply i by 2 (use #1 if multiplied by 2, use #2 if multiplied by 4,...)

lsr	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): '1_0001_rrrr'	# Assume R7 has `0b0010_0100`
			# lsr r10, r7, #2
			`imm #2`
			`lsr R7`
			`mov R10`
			⇔
			`0_0000_0010`
			`1_0001_0111`
			`1_0111_1010`
			# after and instruction, R10 now holds
			`0b0000_1001`
				Here, we do a logical shift right, and use #1 if divided by 2, use #2 if we divide the current byte by 4,...

add	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): '1_0010_rrrr'	# Assume R0 has `0b0000_0100`
			# add r0, r0, #1
			`imm #1`
			`add R0`
			`mov R0`
			⇔
			`0_0000_0001`
			`1_0010_0000`
			`1_0111_0000`
			# after and instruction, R0 now holds
			`0b0000_0101`
				We can do this instruction by incrementing i and so repeat the for loop.

and	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_0011_rrrr’	# Assume R0 has `0b0000_0101`
			# 7 has `0b0000_0111`
			# and R1, R0, #7
			`imm #7`
			`and R0`
			`mov R1`
			⇔
			`0_0000_0111`
			`1_0011_0000`
			`1_0111_0001`
			# after and instruction, R1 now holds
			`0b0000_0101`
				During the programs, in order to get the encoded or decoded block’s specific positions’ bits, such as “msw_input_block[2:0]”, we may do bit masking in order to keep only the last 3 bits within the 8 bits of input MSW. So we would use #7 to mark the last 3 bits as only 1 and 1 outputs 1 while 0 and 1 or 1 and 0 or 0 and 0 would just output 0.

orr	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_0100_rrrr’	# Assume R10 has `0b1010_1010`
			# Assume R5 has `0b0000_0000` (p8)
			# orr r10, r10, r5
			`take R5`
			`orr R10`
			`mov R10`
			⇔
			`1_0110_0101`
			`1_0100_1010`
			`1_0111_1010`
			# after and instruction, R10 now holds
			`0b1010_1010`
				This could be used when we are trying to construct the encoded_block_msw, encoded_block_msw (output MSW or LSW), and so on. As we want to combine all the bits or partial bits together, we may need to use orr to either output 1 if we have 1 and 1, 0 and 1, or 1 and 0, whereas outputs 0 if 0 and 0. The values of p8, p4, p2, p1, and p0 would be 0 (`0b0000_0000`) or 1 (`0b0000_0001`).

eor	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_0101_rrrr’	# Assume R4 has `0b0000_0001`
			# Assume R5 has `0b0101_0000`
			# eor r5, r4, r5
			`take R5`
			`eor R4`
			`mov R5`
			⇔
			`1_0110_0101`
			`1_0101_0100`
				This could be used when we are trying to construct the encoded_block_msw, encoded_block_lsw (output MSW or LSW), and so on. As we want to combine all the bits or partial bits together, we may need to use eor to either output 1 if we have 1 and 0, 0 and 1, whereas outputs 0 if 0 and 0, 1 and 1.
				0001 1101

			`1_0111_0101` # after and instruction, R5 now holds `0b0101_0001`	

take	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_0110_rrrr’	# Assume R5 has `0b0000_0000`
			`take R5`
			⇔
			`1_0110_0101`
			# after take instruction, register of accumulator R15 now holds `0b0000_0000`
				Take the value of operand to register the accumulator R15.

mov	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_0111_rrrr’	# Assume register of accumulator R15 has `0b0000_1111`
			`mov R5`
			⇔
			`1_0111_0101`
			# after mov instruction, R5 now holds `0b0000_1111`
				Mov value of register of accumulator R15 to operand.

ldr	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_1000_rrrr’	# Assume data_mem[$R5] hold the value `0b0000_0110`
			`ldr R5`
			⇔
			`1_1000_0101`
			# after ldr instruction, register of accumulator R15 hold the value `0b0000_0110`
				Load value of data_mem[$operand] into register of accumulator R15.

str	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_1001_rrrr’	# Assume register of accumulator R15 has `0b0000_0110`
			`str R5`
			⇔
			`1_1001_0101`
			# after str instruction, data_mem[$R5] hold the value
				Store value of register of accumulator R15 into data_mem[$operand].

			0b0000_0110	
cmp	R	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_1010_rrrr’	# Assume register of accumulator R15 has 0b0000_0110 # Assume R5 has 0b0000_0110 # cmp R5, #15 # beq end_loop imm #15 cmp R5 beq end_loop ⇔ 1_1010_0101 # after cmp instruction, register of accumulator R15 hold the value 0b0000_0001	Compare both values in register of accumulator R15 and operand, if they are equal, output is 1, else is 0.
b	B	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_1011_rrrr’	b label ⇔ 1_1011_&[label] # after b instruction, branch to label	Branch to label.
beq	B	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_1100_rrrr’	# Assume register of accumulator R15 has 0b0000_0001 beq label ⇔ 1_1100_&[label] # after beq instruction, branch to label	When the register of accumulator R15 has value 1, branch to label.
bne	B	1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_1101_rrrr’	# Assume register of accumulator R15 has 0b0000_0000 bne label ⇔ 1_1101_&[label] # after bne instruction, branch to label	When the register of accumulator R15 has value 0, branch to label.

stop		1 bit type (1), 4 bits opcode, 4 bit operand register (R-type started with bit 1): ‘1_1110_0000”		

Internal Operands

There are 16 registers supported. R15 is the accumulator register.

Control Flow (branches)

b: Branch to label.
beq: When the register of accumulator R15 has value 1, branch to label.
bne: When the register of accumulator R15 has value 0, branch to label.

We support indirect ways to calculate target addresses. We will make a lookup table to pre-set the relative address of the target address. The branch distance range is -512, 511.

Addressing Modes

We support direct and indirect mode for memory addressing.
Example 1:

```
ldr R5
```

It is a direct mode, the instruction load value of data_mem[$R5] into register of accumulator R15.
Example 2:

```
imm #32
add R5
mov R6
ldr R6
```

It is an indirect mode, the instruction load value of data_mem[$R5+32] into the register of accumulator R15.

4. Programmer's Model [Lite]

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.

We have a for loop. Each time, we load one message with 2 parts MSW and LSW from our data memory to our register file. And then we do the computations based on the two inputs and store the results back from the register file to data memory. Begin by efficiently managing memory operations. Load required data into registers to minimize memory accesses during execution. Reuse registers whenever possible instead of constantly allocating new ones. After computation, we write results back to the correct memory locations. Minimize memory accesses within the loop as memory operations are typically slower than working with registers. In summary, effective programming for this machine demands a balance between memory management, register usage, and minimizing memory accesses to maximize computational efficiency and overall program performance.

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.

No, we cannot directly copy ARM or MIPS instructions into a design with a 9-bit instruction format. ARM and MIPS instructions are typically 32 bits long. Our format lacks the capacity to represent this information. We only have 9 bits for instructions, 1 bit for instruction type, 4 bits for opcode, and the last four bits for operand register rather than having 32-bit instruction for ARM ISA. Attempting to directly port complex ISAs like ARM or MIPS to our design is impractical due to the significant differences in instruction length and encoding.

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?"

Our ALU would not be used for non-arithmetic instructions. For our `imm` operation, it would only output the `ALU_OUT` by passing the last 8 bits with `Instruction[7:0]` without doing any computations. This immediate number would only later be written onto an accumulator. For other arithmetic operations like `lsl, lsr, add, and, orr, eor`, they would do their own arithmetic computations between two inputs `INPUT_REG` and `INPUT_ACC`. For operations like `take, mov, load,` and `str`, they would only need to use only one input, either `INPUT_REG` or `INPUT_ACC (like take` would be `$acc = $reg` while `move` would be `$reg = $acc`, and `load/str` would use the output `ALU_OUT` as `DataAddress` for `data_mem` to either read from (produce Mem_Out) or write with. In last, for ther opreation like `cmp`, it would also not do any ARM-like memory address pointer calculations or PC relative branch computations, it would only output `ALU_OUT` as 1 if two inputs `INPUT_REG` and `INPUT_ACC` are equal to each other and set `tojump_next` = 1'b1. If `INPUT_REG` and `INPUT_ACC` are not equal to each other, it would simply output `ALU_OUT` as 0 and set the flag `tojump_next` = 1'b0 without doing any computations.

5. Individual Component Specification

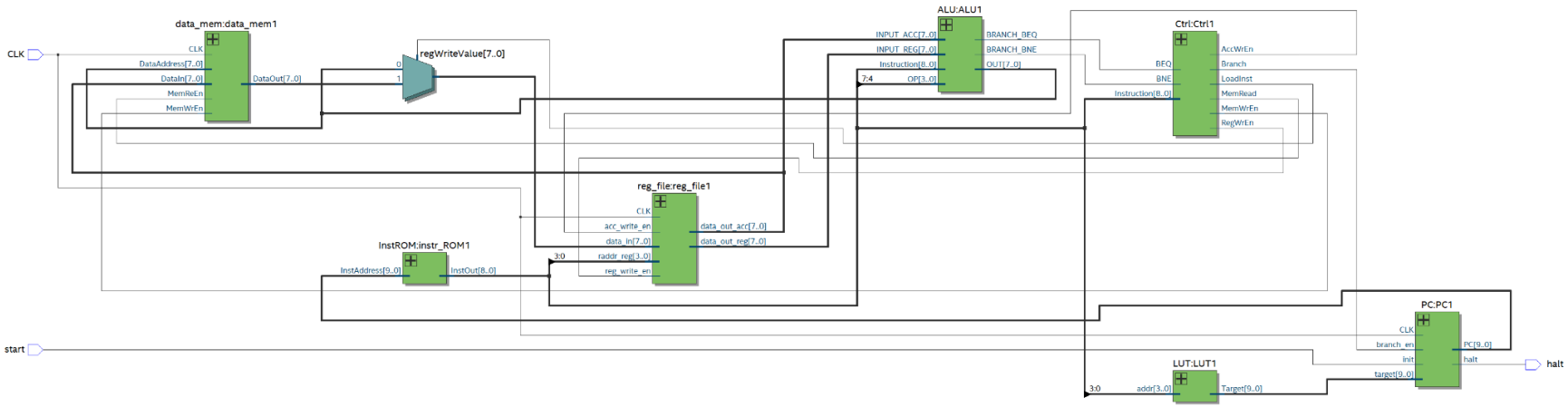
Top Level

Module file name: TopLevel.sv

Functionality Description

Top Level connects all separate parts, including program counter, control decoder, instruction memory, register file, ALU, and data memory of the architecture, and allows them to function as a whole. Input of Top Level includes a start indication and a clock, and output of Top Level is the halt indicating done flag.

Schematic



Program Counter

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

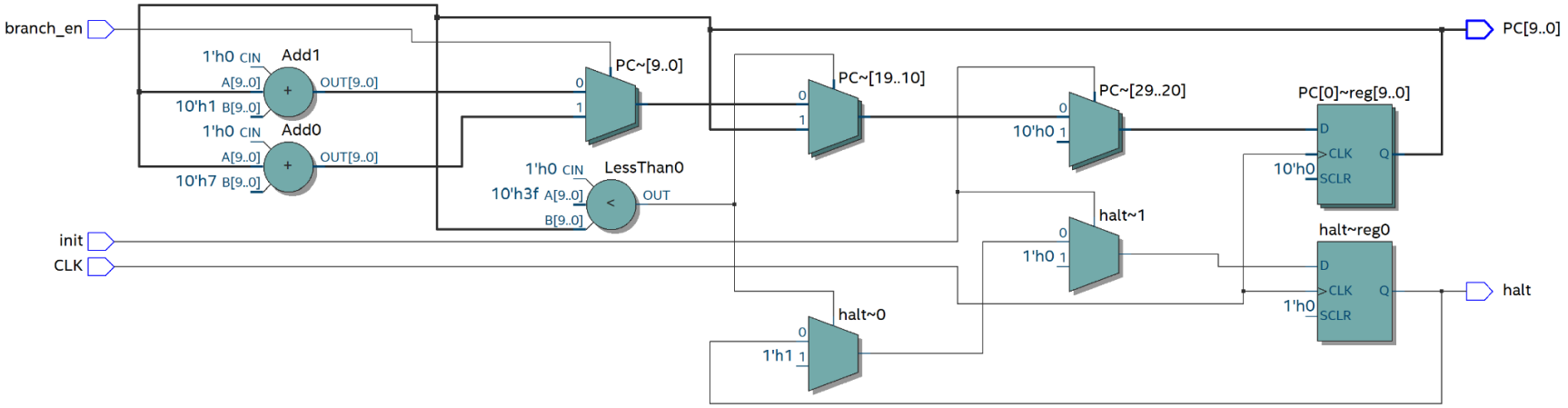
Functionality Description

Program Counter keeps track of the line of code running on the architecture as the program continues, and it allows branch instructions in the program. Input of program counter includes initialization indicator, branch enable and clock. Output of program counter includes halt done flag and updated program count.

(Optional) Testbench Description

TODO. Describe your testbench. How does it work? What test cases does it test?

Schematic



(Optional) Timing Diagram

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

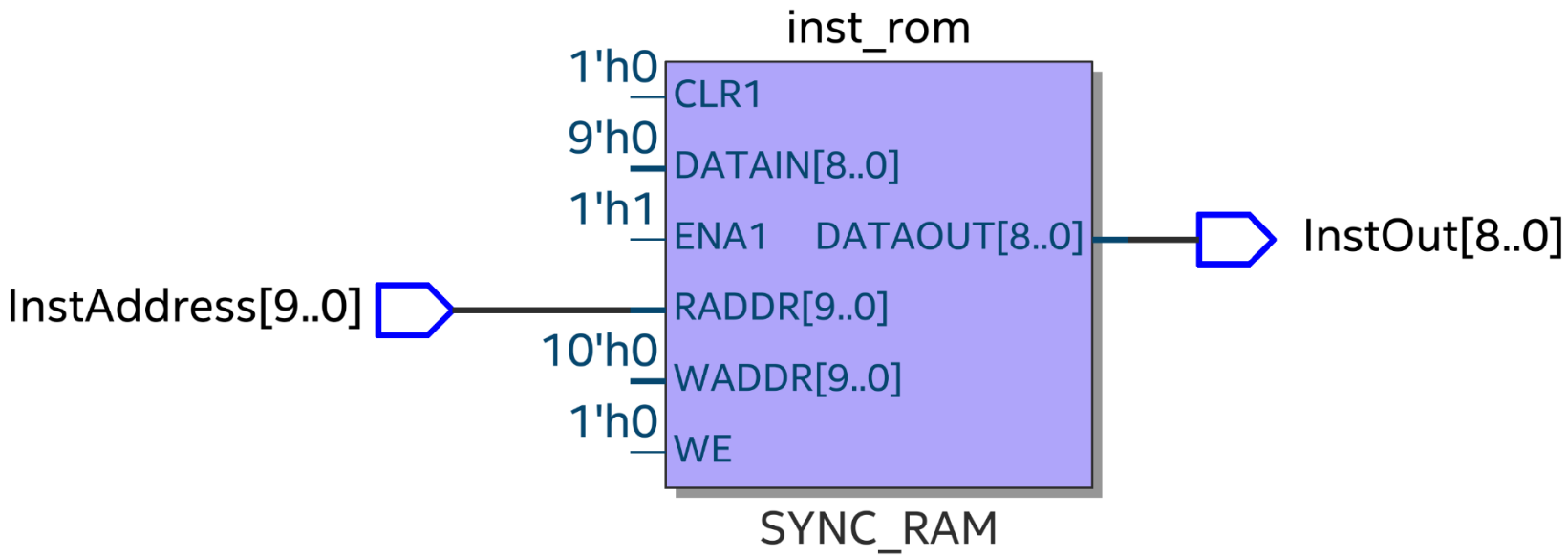
Instruction Memory

Module file name: inst_rom2.sv

Functionality Description

Instruction Memory loads the address that the program counter points to and finds the corresponding instruction based on the machine code. The input of instruction memory is the program counter value and the output is the current instruction.

Schematic



Control Decoder

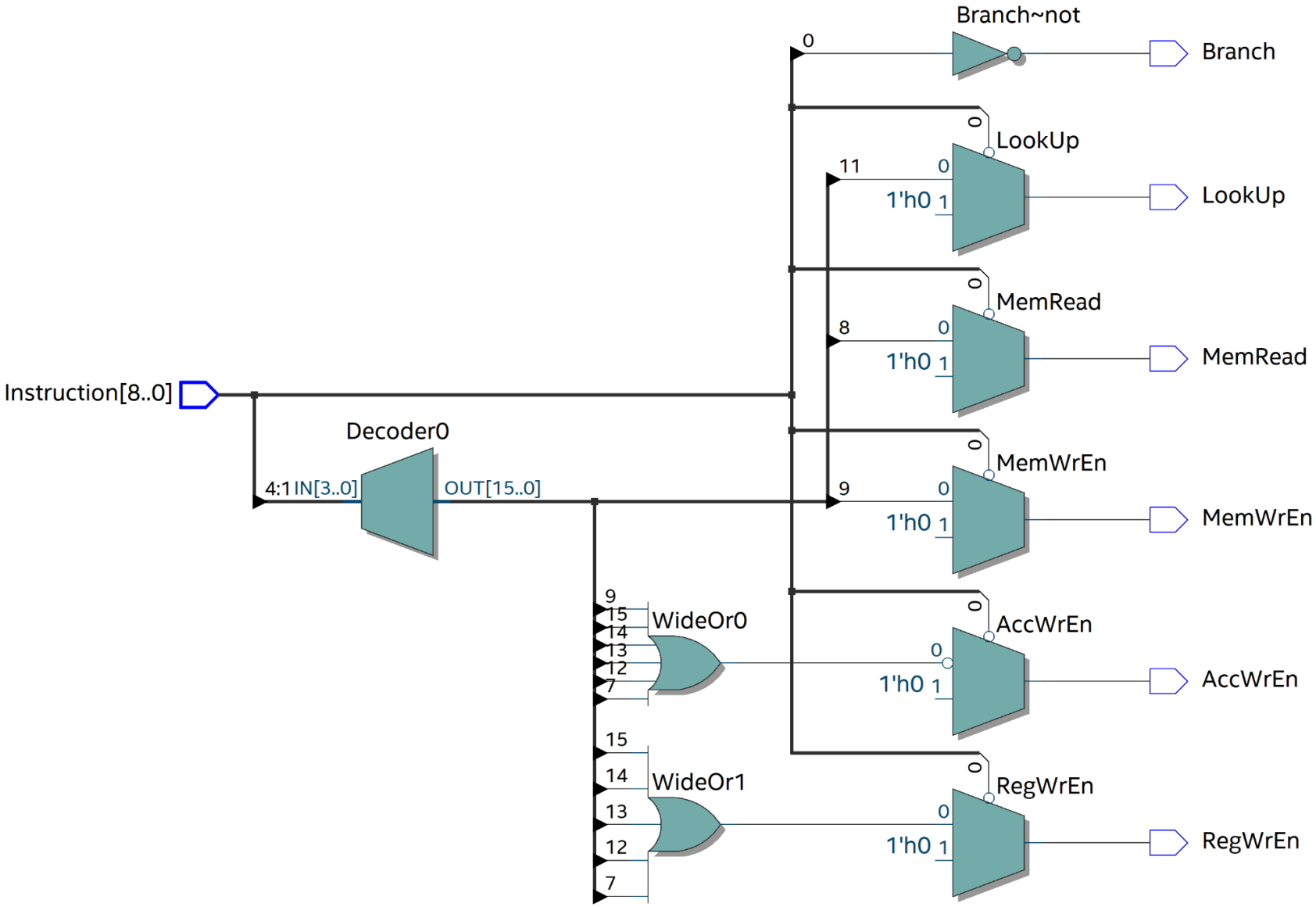
Module file name: Ctrl.sv

Functionality Description

Control Decoder inputs from instrROM, ALU flags, and then outputs to program_counter (fetch unit). For most cases of the operation, this module basically would help flag whether writing on an accumulator or a register. For example, for the `take` operation, it would be `$accumulator = $register`, so `AccWrEn == 1` and `RegWrEn == 0`. However, for the `mov` operation, it would be `$reg = $accumulator`, so `AccWrEn == 0` and `RegWrEn == 0`. These flag outputs would later tell reg_file.sv to write the `data_in` values either on the accumulator (r15) or on the specific register (r0 to r14). For branch operations (beq, and bne), we can mark the

BRANCH flag to be 1 when either BEQ==1 or BNE==1 justified from ALU's cmp operations satisfies each branch condition. For operation, it will automatically allow for relative jumpings without comparing.

Schematic


Branch~not
0
Branch
LookUp
11
0
1'h0
1
LookUp
MemRead
8
0
1'h0
1
MemRead
MemWrEn
9
0
1'h0
1
MemWrEn
Instruction[8..0]
Decoder0
4:1
IN[3..0]
OUT[15..0]
9
15
14
13
12
7
WideOr0
AccWrEn
0
1'h0
1
AccWrEn
15
14
13
12
7
WideOr1
RegWrEn
0
1'h0
1
RegWrEn

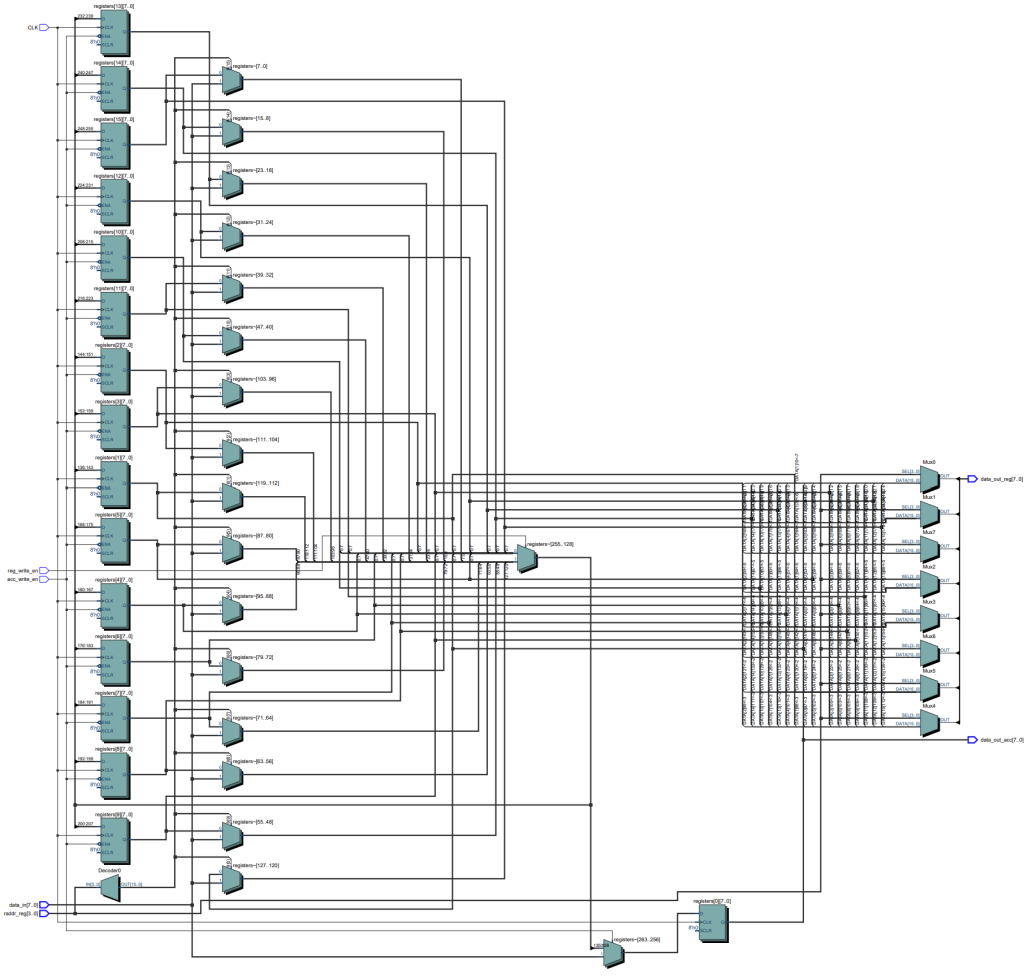
Register File

Module file name: reg_file.sv

Functionality Description

This module provides a register file with configurable width and depth. It allows for both reading and writing of data to and from the accumulator register (r15) and other registers (r0 to r14) in the file. The number 15 specifies the accumulator register to read from or write with, and the `raddr_reg` (register_index) input specifies which other registers to read from or write with. And data can be written to the registers based on control signals (`reg_write_en` and `acc_write_en`) getting from the Control Decoder and the input data (`data_in`). If it is a load operation and so when "`LoadInst`" flagged to be true, then `data_in` would be taken with `Mem_Out`. Otherwise, `data_in` would be the `ALU_out`.

Schematic


CLK
reg_write_en
acc_write_en
data_in[7..0]
raddr_reg[3..0]
Decoder0
registers[127..0]
registers[7..0]
registers[15..8]
registers[23..16]
registers[31..24]
registers[39..32]
registers[47..40]
registers[103..96]
registers[111..104]
registers[119..112]
registers[87..80]
registers[95..88]
registers[79..72]
registers[71..64]
registers[63..56]
registers[55..48]
registers[127..120]
registers[255..128]
registers[263..256]
Mux0
Mux1
Mux7
Mux2
Mux3
Mux6
Mux5
Mux4
data_out_reg[7..0]
data_out_acc[7..0]

ALU (Arithmetic Logic Unit)

Module file name: ALU.sv

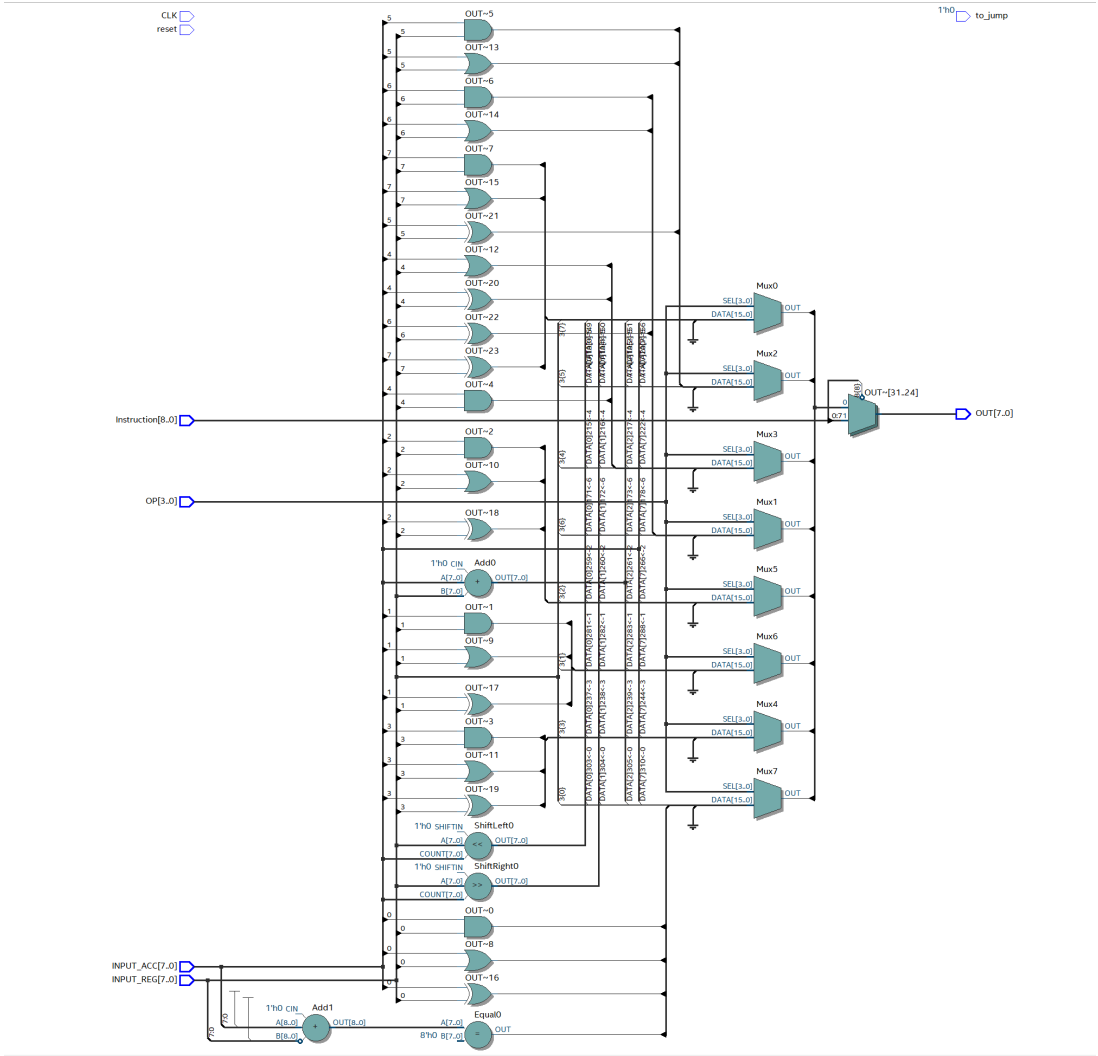
Functionality Description

This ALU module basically computes or calculates the value for most of the operations, such as adding or xoring the two inputs, `INPUT_REG` and `INPUT_ACC` getting from the reg_file.sv. We also take the immediate number for imm (1_iiii_iiii) operation such as #15 and write to our accumulator (r15 = #15), and then using the accumulator value `INPUT_ACC` to either add, xoring, orring with other registers `INPUT_REG`.

ALU Operations

1. Immediate imm (imm #15 means $accumulator = r15 = #15), and the ALU_OUT would be the last 8 bits in the instruction. So this data could be inputted to regWriteValue in reg_file.sv to write the immediate value into the accumulator r15.
2. Logical shift left (lsl) and logical shift right (lsr). Relevant instructions are imm #1 ($accumulator = r15 = #1), lsl r0 $accumulator = $r0 << #1). It works similarly for lsr operation.
3. Add $reg means $accumulator = $accumulator + $reg (if imm #1, add r0 ⇒ $accumulator = r15 = #1 + r0)
4. And $reg means $accumulator = $accumulator & $reg (if take r0, and r3 ⇒ $accumulator = r15 = r0 & r3)
5. Orr $reg means $accumulator = $accumulator | $reg (if take r0, orr r3 ⇒ $accumulator = r15 = r0 | r3)
6. Xor $reg means $accumulator = $accumulator ^ $reg (if take r0, xor r3 ⇒ $accumulator = r15 = r0 ^ r3)
7. Take $reg means $accumulator = $reg (if take r0 ⇒ $accumulator = r15 = r0)
8. Move $reg means $reg = $accumulator (if mov r0 ⇒ $r0 = $accumulator = r15)
9. Ldr $reg means $accumulator = MEM[$reg] (if ldr r2 ⇒ $accumulator = MEM[r2])
10. Str $reg means MEM[$reg] = $accumulator (if str r2 ⇒ MEM[r2] = $accumulator)
11. Cmp $reg means compare between INPUT_ACC and INPUT_REG, if they are equal, put $accumulator = 1 and BRANCH_BEQ =1; If they are not equal to each other, put $accumulator = 0 and BRANCH_BNE = 1.

Schematic


CLK
reset
1'h0
to_jump
Instruction[8..0]
OP[3..0]
INPUT_ACC[7..0]
INPUT_REG[7..0]
OUT[7..0]
OUT[31..24]
Mux0
Mux1
Mux2
Mux3
Mux4
Mux5
Mux6
Mux7
SEL[3..0]
DATA[15..0]
OUT
Add0
Add1
ShiftLeft0
ShiftRight0
Equal0
CIN
SHIFTIN
COUNT[7..0]
A[7..0]
B[7..0]
OUT[8..0]

(Optional) Timing Diagram

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

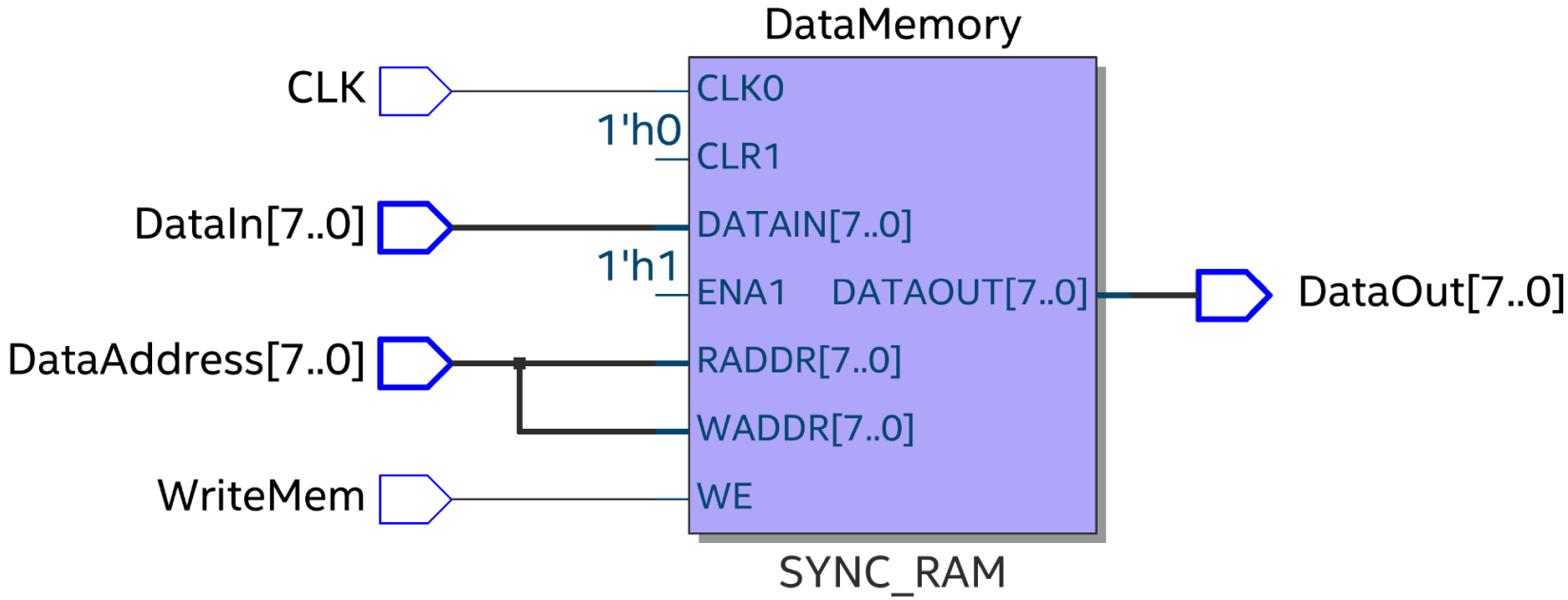
Data Memory

Module file name: data_mem.sv

Functionality Description

Using address pointer for both read and write data memory

Schematic



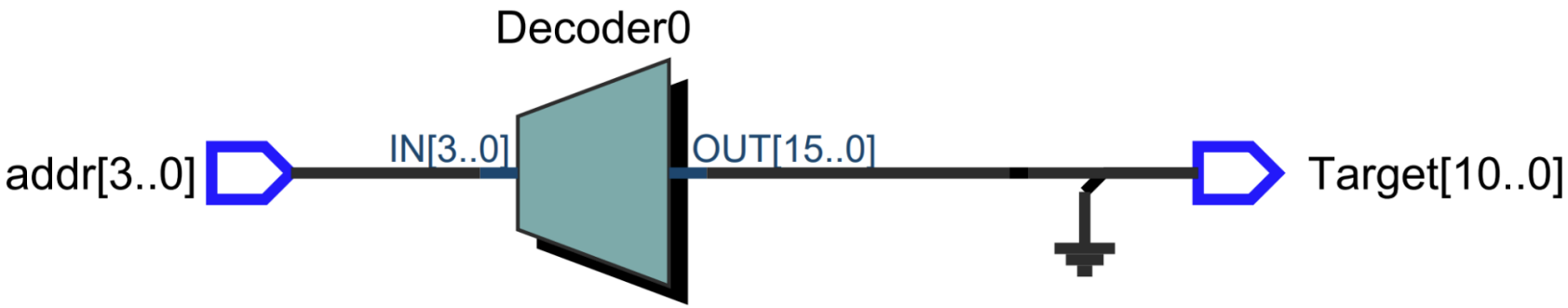
Lookup Tables

Module file name: LUT.sv

Functionality Description

Lookup table for PC target, leverage a 4-bit pointer to a 10-bit number

Schematic



Muxes (Multiplexers)

Module file name: MultiMux.sv

Functionality Description

ALUmux:

1: output value of regitster
0: output immediate

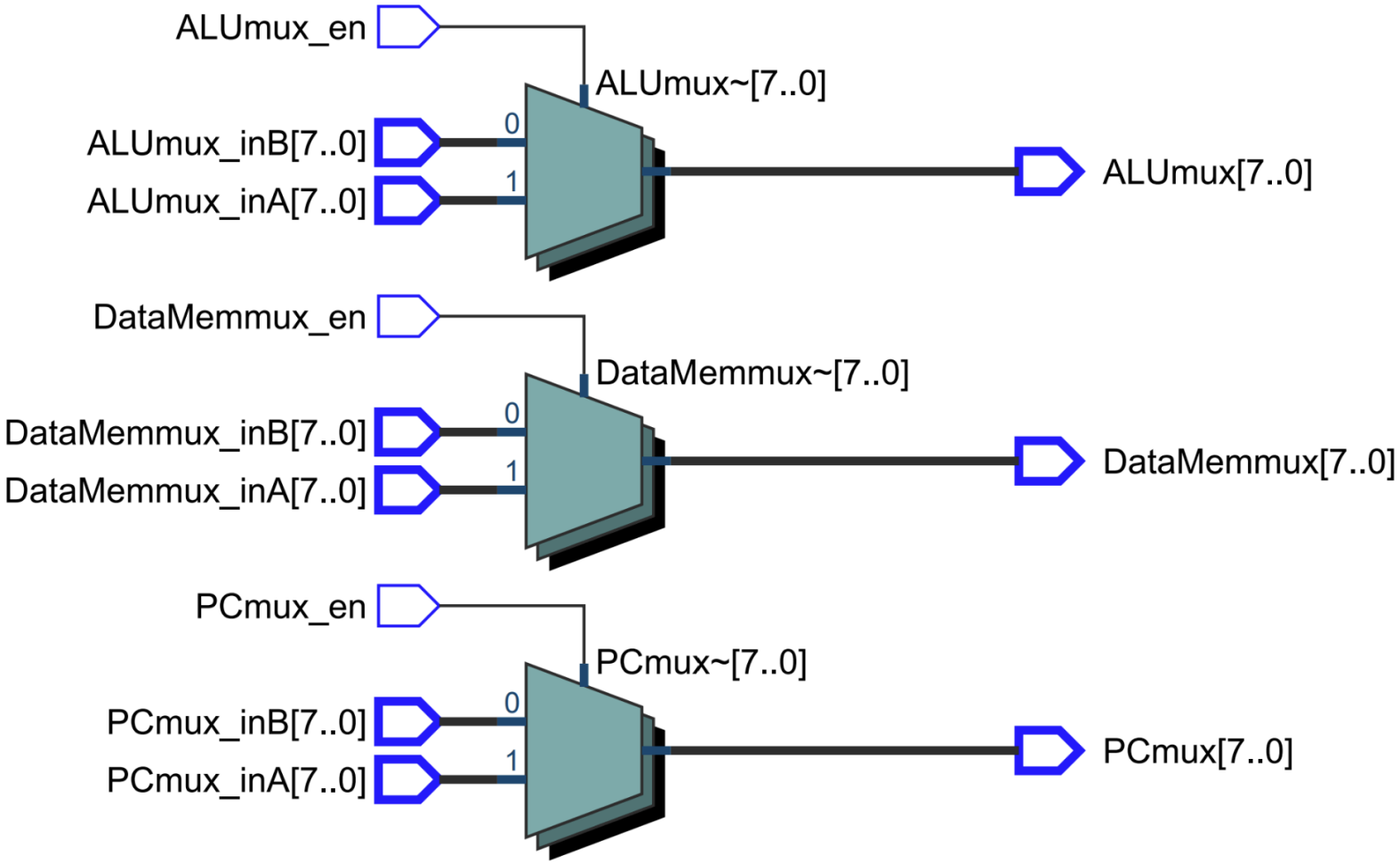
DataMemmux:

1: output value from data memory
0: output value from ALU

PCmux:

1: output next instruction’s address
0: output address of the instruction to jump to

Schematic


ALUmux_en
ALUmux~[7..0]
ALUmux_inB[7..0]
ALUmux_inA[7..0]
0
1
ALUmux[7..0]
DataMemmux_en
DataMemmux~[7..0]
DataMemmux_inB[7..0]
DataMemmux_inA[7..0]
0
1
DataMemmux[7..0]
PCmux_en
PCmux~[7..0]
PCmux_inB[7..0]
PCmux_inA[7..0]
0
1
PCmux[7..0]

Other Modules (if necessary)

Module file name: definitions.sv

Functionality Description

This module file mainly defines the parameters used in the ALU for most of the operations to be constants. So in the ALU file, we can directly use the kLSL, kADD and other constant operations under the case(OP) received from the range Instruction[7:4].

6. Program Implementation

Program 1 Pseudocode

```
void  encoder(mem[0:59]){
  for (i = 0; i < 15; i++) {
      # Load each 11-bit message block from data memory (mem[0:29])
      lsw_input_block = mem[i * 2];
      msw_input_block = mem[i * 2 + 1];

      # calculate parity bits
      p8 = (^msw_input_block[2:0])^(^lsw_input_block[7:4]);
      p4 = (^msw_input_block[2:0])^lsw_input_block[7]^(^lsw_input_block [3:1]);
      p2 = (^msw_input_block[2:1])^(^lsw_input_block[6:5])^(^lsw_input_block[3:2])^lsw_input_block[0];
      p1 = msw_input_block[2]^msw_input_block[0]^lsw_input_block[6]^(^lsw_input_block[4:3])^(^lsw_input_block[1:0]);
      p0 = (^msw_input_block[2:0])^(^lsw_input_block[7:0])^p8^p4^p2^p1);

      # Construct the 16-bit encoded version
      encoded_block_msw = {msw_input_block[2:0], lsw_input_block[7:4], p8};
      encoded_block_lsw = {lsw_input_block[3:1], p4, lsw_input_block[0], p2:p1, p0};
```

```
      # Store the encoded block in data memory (mem[30:59])
      mem[30 + (i * 2)] = encoded_block_lsw;
      mem[30 + (i * 2) + 1] = encoded_block_msw;
  }
}
```

Program 1 Assembly Code

```
# r0  = for loop index counter, i
# r1  = Load lsw_input_block = MEM[2*i]
# r2  = Load msw_input_block = MEM[2*i+1]
# r3  = value of 2*i
# r4  = 0000_000 b9^b11^b10
# r5  = value of p8
# r6  = value of p4
# r7  = value of p2
# r8  = value of p1
# r9  = value of p0
# r10 = output MSW
# r11 = output LSW
# r12 = 30 + 2*i for output LSW
# r13 = 0000_000 b8^b5^b6^b7, then r13 = 31 + 2*i for output MSW
# r14 = 0000_000 b9^b11
# r15 = $accumulator register


    # initialize loop counter
    imm 0
    mov r0   #r0 = 0

    # for loop index i from 0 to 14, stop at i reaches 15
    imm 15
    cmp r0
```

```
beq 0    #LUT[0] = 274

imm 1
lsl r0
mov r3   #r3 = 2 * r0 = 2*i

ldr r3
mov r1   #r1 = MEM[r3]

imm 1
add r3
mov r2
ldr r2
mov r2   #r2 = MEM[r3+1]


#calculate the value of p8
imm 7
and r2
mov r4   #r4 = 7 & r2 (0000_0 b11 b10 b9)

imm 2
lsr r4   #$accum = r4 >> 2
eor r4   #r4 = $accum ^ r4
mov r4   #r4 = 0000_0 b11 b10 b9^b11
imm 3
and r4
mov r4   #r4 = 3 && r4 (0000_00 b10 b9^b11)

imm 1
and r4
mov r14  #r14 = 1 & r4 (0000_000 b9^b11)
```

```
imm 1
lsr r4
eor r4   #$accum = $accum ^ $r4
mov r4   #r4 = 0000_00 b10 b9^b11^b10
imm 1
and r4
mov r4   #r4 = 0000_000 b9^b11^b10

imm 240
and r1
mov r5   #r5 = b8  b7  b6  b5_0000

imm 3
lsl r5
eor r5
mov r5   #r5 = b8^b5 b7 b6 b5_0000
imm 224
and r5
mov r5   #r5 = b8^b5 b7 b6 0_0000

imm 2
lsl r5
eor r5
mov r5   #r5 = b8^b5^b6 b7 b6 0_0000
imm 192
and r5
mov r5   #r5 = b8^b5^b6 b7 00_0000

imm 1
lsl r5
eor r5
```

```
mov r5   #r5 = b8^b5^b6^b7 b7 00_0000
imm 128
and r5
mov r5   #r5 = b8^b5^b6^b7 000_0000

imm 7
lsr r5
mov r13  #r13 = 0000_000 b8^b5^b6^b7

imm 7
lsr r5
eor r4
mov r5   #r5 = 0000_000 b8^b5^b6^b7^b9^b11^b10


#calculate the value of p4
imm 128
and r1
mov r6   #r6 = b8 000_0000
imm 7
lsr r6
eor r4
mov r6   #r6 = 0000_000 b8^b9^b11^b10

imm 8
and r1
mov r7   #r7 = 0000_b4 000
imm 3
lsr r7
eor r6
mov r6   #r6 = 0000_000 b8^b9^b11^b10^b4
```

```
imm 4
and r1
mov r7   #r7 = 0000_0 b3 00
imm 2
lsr r7
eor r6
mov r6   #r6 = 0000_000 b8^b9^b11^b10^b4^b3

imm 2
and r1
mov r7   #r7 = 0000_00 b2 0
imm 1
lsr r7
eor r6
mov r6   #r6 = 0000_000 b8^b9^b11^b10^b4^b3^b2


#calculate the value of p2
imm 6
and r2
mov r7   #r7 = 0000_0 b11 b10 0
imm 1
lsr r7
mov r7   #r7 = 00000_00 b11 b10
imm 1
lsr r7
eor r7
mov r7   #r7 = 0000_00 b11 b11^b10
imm 1
and r7
mov r7   #r7 = 0000_000 b11^b10
```

```
imm 64
and r1
mov r8   #r8 = 0 b7 00_0000
imm 6
lsr r8
eor r7
mov r7   #r7 = 0000_000 b11^b10^b7

imm 32
and r1
mov r8   #r8 = 00 b6 0_0000
imm 5
lsr r8
eor r7
mov r7   #r7 = 0000_000 b11^b10^b7^b6

imm 8
and r1
mov r8   #r8 = 0000_b4 000
imm 3
lsr r8
eor r7
mov r7   #r7 = 0000_000 b11^b10^b7^b6^b4

imm 4
and r1
mov r8   #r8 = 0000_0 b3 00
imm 2
lsr r8
eor r7
mov r7   #r7 = 0000_000 b11^b10^b7^b6^b4^b3
```

```
imm 1
and r1
mov r8   #r8 = 0000_000 b1
eor r7
mov r7   #r7 = 0000_000 b11^b10^b7^b6^b4^b3^b1


#calculate the value of p1
imm 64
and r1
mov r8   #r8 = 0 b7 00_0000
imm 6
lsr r8
eor r14
mov r8   #r8 = 0000_000 b9^b11^b7

imm 16
and r1
mov r9   #r9 = 000 b5_0000
imm 4
lsr r9
eor r8
mov r8   #r8 = 0000_000 b9^b11^b7^b5

imm 8
and r1
mov r9   #r9 = 0000_b4 000
imm 3
lsr r9
eor r8
mov r8   #r8 = 0000_000 b9^b11^b7^b5^b4
```

```
imm 2
and r1
mov r9   #r9 = 0000_00 b2 0
imm 1
lsr r9
eor r8
mov r8   #r8 = 0000_000 b9^b11^b7^b5^b4^b2

imm 1
and r1
mov r9   #r9 = 0000_000 b1
eor r8
mov r8   #r8 = 0000_000 b9^b11^b7^b5^b4^b2^b1


#calculate the value of p0
imm 15
and r1
mov r9   #r9 = 0000_b4 b3 b2 b1

imm 3
lsr r9
eor r9
mov r9   #r9 = 0000_b4 b3 b2 b4^b1
imm 7
and r9
mov r9   #r9 = 0000_0 b3 b2 b4^b1

imm 2
lsr r9
eor r9
mov r9   #r9 = 0000_0 b3 b2 b4^b1^b3
```

```
imm 3
and r9
mov r9   #r9 = 0000_00 b2 b4^b1^b3

imm 1
lsr r9
eor r9
mov r9   #r9 = 0000_00 b2 b4^b1^b3^b2
imm 1
and r9
mov r9   #r9 = 0000_000 b4^b1^b3^b2

take r13
eor r9
mov r9   #r9 = 0000_000 b4^b1^b3^b2^b8^b5^b6^b7

take r4  #$accum = r4 (0000_000 b9^b11^b10)
eor r9
mov r9   #r9 = 0000_000 b4^b1^b3^b2^b8^b5^b6^b7^b9^b11^b10


take r5
eor r9
mov r9   #r9 = 0000_000 b4^b1^b3^b2^b8^b5^b6^b7^b9^b11^b10^p8

take r6
eor r9
mov r9   #r9 = 0000_000 b4^b1^b3^b2^b8^b5^b6^b7^b9^b11^b10^p8^p4

take r7
eor r9
mov r9   #r9 = 0000_000 b4^b1^b3^b2^b8^b5^b6^b7^b9^b11^b10^p8^p4^p2
```

```
take r8
eor r9
mov r9   #r9 = 0000_000 b4^b1^b3^b2^b8^b5^b6^b7^b9^b11^b10^p8^p4^p2^p1


#construct the 16-bit encoded version
#construct the encoded_block_msw
imm 5
lsl r2
mov r10   #r10 = b11 b10 b9 0_0000

imm 3
lsr r1
mov r11   #r11 = 000 b8_b7 b6 b5 b4

imm 30
and r11
mov r11   #r11 = 000 b8_b7 b6 b5 0

take r10
orr r11   #$accum = 0 0 0 b8_b7 b6 b5 0 | b11 b10 b9 0 0 0 0 0
mov r11   #r11 =  = b11 b10 b9 b8_b7 b6 b5 0

take r5
orr r11   #$accum = b11 b10 b9 b8_b7 b6 b5 0 | 0 0 0 0 0 0 0 p8
mov r10   #r11 = b11 b10 b9 b8_b7 b6 b5 p8

#construct encoded_block_lsw
imm 4
lsl r1
mov r11   #r11 = b4  b3  b2  b1_0000
```

```
imm 224
and r11
mov r11   #r11 = b4  b3  b2  0_0000

imm 4
lsl r6
mov r12   #r12 =  000 p4_0000

take r11
orr r12
mov r11   #r11 =  b4  b3  b2  p4_0000

imm 3
lsl r1
mov r12   #r12 =  b5  b4  b3  b2_b1 000

imm 8
and r12
mov r12   #r12 =  0000_b1 000

take r11
orr r12
mov r11   #r11 =  b4  b3  b2  p4_b1 000

imm 2
lsl r7
mov r12   #r12 =  0000_0 p2 00
take r11
orr r12
mov r11   #r11 =  b4  b3  b2  p4_b1 p2 00
```

```
imm 1
lsl r8
mov r12   #r12 =  0000_00 p1 0
take r11
orr r12
mov r11   #r11 =  b4  b3  b2  p4_b1 p2 p1 0

take r9
orr r11
mov r11   #r11 =  b4  b3  b2  p4_b1 p2 p1 p0


# Store the encoded block in data memory (mem[30:59])
imm 30
add r3
mov r12   #r12 = $accum = 30 + r3
take r11  #$accum = $r11
str r12   #Mem[r12] = Mem[30 + r3] = $accumulator = r11

imm 31
add r3
mov r13   #r13 = 31 + r3
take r10
str r13   #Mem[r13] = Mem[31 + r3] = $accumulator = r10


#increment index i and repeat the for loop
imm 1
add r0
mov r0
b 1  #LUT[1] = -275
```

```
stop
```

Program 2 Pseudocode

First, calculate the XOR result for p0, p1, p2, p4, p8 using corresponding bits. When the result is 0, the corresponding p_r is correct; when the result is 1, the corresponding p_r is wrong.

ition	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
d data s	p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11
p1	✓		✓		✓		✓		✓		✓		✓		✓
p2		✓	✓			✓	✓			✓	✓			✓	✓
p4				✓	✓	✓	✓					✓	✓	✓	✓
e p8								✓	✓	✓	✓	✓	✓	✓	✓

This table is missing p0, which is the xor result of all bits d1-d11 and p1-p8.

Analysis on error (All the following assumes that there's only two types of errors - 1-bit error and 2-bits errors):
Case 1: 1-bit error

a. If only one of the p_ is wrong: the corresponding p_ has an error.
b. If p0, p1 and p2 are wrong: bit d1 has an error.
c. If p0, p1 and p4 are wrong: bit d2 has an error.

d. If p0, p2 and p4 are wrong: bit d3 has an error.
e. If p0, p1, p2 and p4 are wrong: bit d4 has an error.
f. If p0, p1 and p8 are wrong: bit d5 has an error.
g. If p0, p2 and p8 are wrong: bit d6 has an error.
h. If p0, p1, p2 and p8 are wrong: bit d7 has an error.
i. If p0, p4 and p8 are wrong: bit d8 has an error.
j. If p0, p1, p4 and p8 are wrong: bit d9 has an error.
k. If p0, p2, p4 and p8 are wrong: bit d10 has an error.
l. If p0, p1, p2, p4 and p8 are wrong: bit d11 has an error.

For all 1-bit error cases above (except a no need to fix), correct the error by simply reversing the error bit.

```
void  decoder(mem[0:59]){
  for (i = 0; i < 15; i++) {
      // Load each 11-bit message block from data memory (mem[30:59])
      encoded_block_lsw = mem[30 + (i * 2)];
      encoded_block_msw = mem[30 + (i * 2) + 1];

      // Calculate the results of p
      p8_r = ^encoded_block_msw[7:0];
      p4_r = (^encoded_block_msw[7:4])^(^encoded_block_lsw[7:4]);
      p2_r =
(^encoded_block_msw[7:6])^(^encoded_block_msw[3:2])^(^encoded_block_lsw[7:6])^(^encoded_block_lsw[3:2]);
      p1_r =
encoded_block_msw[7]^encoded_block_msw[5]^encoded_block_msw[3]^encoded_block_msw[1]^encoded_block_lsw[7]^encod
ed_block_lsw[5]^encoded_block_lsw[3]^encoded_block_lsw[1];
      p0_r = (^encoded_block_msw[7:0])^(^encoded_block_lsw[7:0]);

      // Determine errors based on the result
      result = {p0_r, p1_r, p2_r, p4_r, p8_r};

      //No error
      if (result == 0b00000) {
            output_msw = {0b00000,encoded_block_msw[7:5]};
            output_lsw = {encoded_block_msw[4:1],encoded_block_lsw[7:5],encoded_block_lsw[3]};
```

```
//2-bits error
} else if (p0_r == 0) {
      output_msw = {0b10000,encoded_block_msw[7:5]};
      output_lsw = {encoded_block_msw[4:1],encoded_block_lsw[7:5],encoded_block_lsw[3]};
//1-bit error
} else {
      output_msw = {0b01000,encoded_block_msw[7:5]};
      output_lsw = {encoded_block_msw[4:1],encoded_block_lsw[7:5],encoded_block_lsw[3]};

      if (p0_r,p1_r,p2_r == 1 && p4_r,p8_r == 0){
            output_lsw = output_lsw ^ 0b00000001;
      } else if (p0_r,p1_r,p4_r == 1 && p2_r,p8_r == 0) {
            output_lsw = output_lsw ^ 0b00000010;
      } else if (p0_r,p2_r,p4_r == 1 && p1_r,p8_r == 0) {
            output_lsw = output_lsw ^ 0b00000100;
      } else if (p0_r,p1_r,p2_r,p4_r == 1 && p8_r == 0) {
            output_lsw = output_lsw ^ 0b00001000;
      } else if (p0_r,p1_r,p8_r == 1 && p2_r,p4_r == 0) {
            output_lsw = output_lsw ^ 0b00010000;
      } else if (p0_r,p2_r,p8_r == 1 && p1_r,p4_r== 0) {
            output_lsw = output_lsw ^ 0b00100000;
      } else if (p0_r,p1_r,p2_r,p8_r == 1 && p4_r == 0) {
            output_lsw = output_lsw ^ 0b01000000;
      } else if (p0_r,p4_r,p8_r == 1 && p1_r,p2_r == 0) {
            output_lsw = output_lsw ^ 0b10000000;
      } else if (p0_r,p1_r,p4_r,p8_r == 1 && p2_r == 0) {
            output_msw = output_msw ^ 0b00000001;
      } else if (p0_r,p2_r,p4_r,p8_r == 1 && p1_r == 0) {
            output_msw = output_msw ^ 0b00000010;
      } else if (p0_r,p1_r,p2_r,p4_r,p8_r == 1) {
            output_msw = output_msw ^ 0b00000100;
      } else {
            output_msw[7] = 0b1;
      }
}
```

```
    // Store the decoded message
      mem[i * 2] = output_lsw;
      mem[(i * 2) + 1] = output_msw;
      }
}
```

Program 2 Assembly Code

Bitwise XOR method:

Actual Code:

```
      #r0 = for loop index counter
      #r1 = lsw storage
      #r2 = msw storage
      #r3 = temporary register
      #r4 = temporary register
      #r5 = p values storage
      #r6 = temporary register


      decoder:
      imm 0      # Initialize 2*i = 0 (loop index)
```

```
mov r0

decode_loop:
#Load encoded message from data memory
imm 30          # Load encoded_block_lsw
add r0
mov r1
ldr r1
mov r1
imm 31          # Load encoded_block_msw
add r0
mov r2
ldr r2
mov r2


#Check the correctness of p

#Calculate the correctness of p8
imm 4      #Extract bits [7:4]
lsr r2
mov r3
take r3    #[7]^[3] [6]^[2] [5]^[1] [4]^[0]
eor r2
mov r4
imm 15     #Remove first 4 bits
and r4
mov r4
imm 2      #Extract bits [7]^[3] [6]^[2]
lsr r4
mov r3
take r4    #[7]^[3]^[5]^[1] [6]^[2]^[4]^[0]
```

```
eor r3
mov r4
imm 3      #Keep last 2 bits
and r4
mov r4
imm 1      #Extract bit [7]^[3]^[5]^[1]
lsr r4
mov r3
take r4    #[7]^[3]^[5]^[1]^[6]^[2]^[4]^[0]
eor r3
mov r4
imm 1      #Keep the last bit
and r4
mov r4
take r4    #Store the result of p8 in r5
mov r5

#Calculate the correctness of p4
imm 4      #Get encoded_block_lsw[7:4]
lsr r1
mov r3
imm 4      #Get encoded_block_msw[7:4]
lsr r2
mov r4
take r4    #lsw[7]^msw[7] lsw[6]^msw[6] lsw[5]^msw[5] lsw[4]^msw[4]
eor r3
mov r4
imm 2      #Extract bits lsw[7]^msw[7] lsw[6]^msw[6]
lsr r4
mov r3
take r4    #lsw[7]^msw[7]^lsw[5]^msw[5] lsw[6]^msw[6]^lsw[4]^msw[4]
eor r3
```

```
    mov r4
    imm 3      #Keep the last 2 bits
    and r4
    mov r4
    imm 1      #Extract bit lsw[7]^msw[7]^lsw[5]^msw[5]
    lsr r4
    mov r3
    take r4    #lsw[7]^msw[7]^lsw[5]^msw[5]^lsw[6]^msw[6]^lsw[4]^msw[4]
    eor r3
    mov r4
    imm 1      #Keep the last bit
    and r4
    mov r4
    imm 1      #Shift p4 to index[1]
    lsl r4
    mov r4
    take r4    #Add p4 with p8
    add r5
    mov r5

    #Calculate the correctness of p2
    imm 51     #Get encoded_block_lsw[7:6],[3:2]
    and r1
    mov r3
    imm 51     #Get encoded_block_msw[7:6],[3:2]
    and r2
    mov r4
    take r4    #encoded_block_lsw[7:6]^encoded_block_msw[7:6]
encoded_block_lsw[3:2]^encoded_block_msw[3:2]
    eor r3
    mov r4
    imm 4      #Extract bits encoded_block_lsw[7:6]^encoded_block_msw[7:6]
```

```
      lsr r4
      mov r3
      take r4
#encoded_block_lsw[7:6]^encoded_block_msw[7:6]^encoded_block_lsw[3:2]^encoded_block_msw[3:2]
      eor r3
      mov r4
      imm 3      #Keep the last 2 bits
      and r4
      mov r4
      imm 1      #Extract bit
encoded_block_lsw[7]^encoded_block_msw[7]^encoded_block_lsw[3]^encoded_block_msw[3]
      lsr r4
      mov r3
      take r4         #encoded_block_lsw[7]^[6]^[3]^[2]^encoded_block_msw[7]^[6]^[3]^[2]
      eor r3
      mov r4
      imm 1     #Keep the last bit
      and r4
      mov r4
      imm 2      #Shift p2 to index[2]
      lsl r4
      mov r4
      take r4         #Add p2 with p4, p8
      add r5
      mov r5


      #Calculate the correctness of p1
      imm 170         #Get encoded_block_lsw[7,5,3,1]
      and r1
      mov r3
      imm 170         #Get encoded_block_msw[7,5,3,1]
```

```
and r2
mov r4
take r4         #lsw[7]^msw[7] lsw[5]^msw[5] lsw[3]^msw[3] lsw[1]^msw[1]
eor r3
mov r4
imm 4      #Extract bits lsw[7]^msw[7] lsw[5]^msw[5]
lsr r4
mov r3
take r4         #lsw[7]^msw[7]^lsw[3]^msw[3] lsw[5]^msw[5]^lsw[1]^msw[1]
eor r3
mov r4
imm 15          #Keep the last 4 bits
and r4
mov r4
imm 2      #Extract bit lsw[7]^msw[7]^lsw[3]^msw[3]
lsr r4
mov r3
take r4         #lsw[7]^msw[7]^lsw[3]^msw[3]^lsw[5]^msw[5]^lsw[1]^msw[1]
eor r3
mov r4
imm 2      #Keep the bit at index[1]
and r4
mov r4
imm 2      #Shift p1 to index[3]
lsl r4
mov r4
take r4         #Add p1 to p2, p4, p8
add r5
mov r5

#Calculate the correctness of p0
take r1         #Get encoded_block_lsw[7:0]
```

```
      mov r3
      take r2          #Get encoded_block_msw[7:0]
      mov r4
      take r4     #lsw[7]^msw[7] lsw[6]^msw[6] ... lsw[1]^msw[1] lsw[0]^msw[0]
      eor r3
      mov r4
      imm 4      #Extract bits lsw[7]^msw[7] lsw[6]^msw[6] lsw[5]^msw[5] lsw[4]^msw[4]
      lsr r4
      mov r3
      take r4          #lsw[7]^[3]^msw[7]^[3] lsw[6]^[2]^msw[6]^[2] lsw[5]^[1]^msw[5]^[1]
lsw[4]^[0]^msw[4]^[0]
      eor r3
      mov r4
      imm 15           #Keep the last 4 bits
      and r4
      mov r4
      imm 2      #Extract lsw[7]^[3]^msw[7]^[3] lsw[6]^[2]^msw[6]^[2]
      lsr r4
      mov r3
      take r4          #lsw[7]^[5]^[3]^[1]^msw[7]^[5]^[3]^[1]
lsw[6]^[4]^[2]^[0]^msw[6]^[4]^[2]^[0]
      eor r3
      mov r4
      imm 3      #Keep the last 2 bits
      and r4
      mov r4
      imm 1      #Extract lsw[7]^[5]^[3]^[1]^msw[7]^[5]^[3]^[1]
      lsr r4
      mov r3
      take r4          #(^lsw[7:0])^(^msw[7:0])
      eor r3
      mov r4
```

```
imm 1      #Keep the last bit
and r4
mov r4
imm 4      #Shift p0 to index[4]
lsl r4
mov r4
take r4          #Add p0 to p1, p2, p4, p8
add r5
mov r5


#Store decoded message into r1, r2
imm 3      #Shift b1 to index[0]
lsr r1
mov r3
imm 1      #Keep only b1
and r3
mov r3
imm 4      #Shift b2-b4 to index[3:1]
lsr r1
mov r4
imm 14           #Keep only b2-b4
and r4
mov r4
take r4          #Put b1-b4 together
orr r3
mov r3
imm 3      #Shift b5-b8 to the left
lsl r2
mov r4
imm 240          #Keep b5-b8 only
and r4
```

```
mov r4
take r4          #Store decoded lsw b1-b8 in r1
orr r3
mov r1
imm 5       #Keep b9-b11 only for msw
lsr r2
mov r2


#0 error case
imm 0       #Determine if the results of all p's are 0
cmp r5
beq 3

#1 error case
imm 4       #Shift p0 to index[0]
lsr r5
mov r3
imm 1       #Determine if p0 = 1
cmp r3
beq 4

#2 errors case
imm 128          #Change F1F0 to 10
add r2
mov r2
b 5

one_error:
imm 64           #Change F1F0 to 01
add r2
mov r2
```

```
#Check the error bit
imm 28          #Check b1
cmp r5
beq 2       #Fix b1

b 7
imm 26          #Check b2
cmp r5
beq 2       #Fix b2

b 7
imm 22          #Check b3
cmp r5
beq 2       #Fix b3

b 7
imm 30          #Check b4
cmp r5
beq 2       #Fix b4

b 7
imm 25          #Check b5
cmp r5
beq 2       #Fix b5

b 7
imm 21          #Check b6
cmp r5
beq 2       #Fix b6

b 7
```

```
imm 29          #Check b7
cmp r5
beq 2      #Fix b7

b 7
imm 19          #Check b8
cmp r5
beq 2      #Fix b8

b 7
imm 27          #Check b9
cmp r5
beq 2      #Fix b9

b 7
imm 23          #Check b10
cmp r5
beq 2     #Fix b10

b 7
imm 31          #Check b11
cmp r5
beq 2      #Fix b11

b1_error:
imm 1      #Reverse b1
eor r1
mov r1
b 13       #Go down 4 lines

b2_error:
imm 2      #Reverse b2
```

```
eor r1
mov r1
b 13       #Go down 4 lines

b3_error:
imm 4      #Reverse b3
eor r1
mov r1
b 13       #Go down 4 lines

b4_error:
imm 8      #Reverse b4
eor r1
mov r1
b 13       #Go down 4 lines

b5_error:
imm 16          #Reverse b5
eor r1
mov r1
b 13       #Go down 4 lines

b6_error:
imm 32          #Reverse b6
eor r1
mov r1
b 13       #Go down 4 lines

b7_error:
imm 64          #Reverse b7
eor r1
mov r1
```

```
b 13       #Go down 4 lines

b8_error:
imm 128          #Reverse b8
eor r1
mov r1
b 13       #Go down 4 lines

b9_error:
imm 1      #Reverse b9
eor r2
mov r2
b 13       #Go down 4 lines

b10_error:
imm 2      #Reverse b10
eor r2
mov r2
b 13       #Go down 4 lines

b11_error:
imm 4      #Reverse b11
eor r2
mov r2

end_decode:
take r1          #Store lsw
str r0
imm 1      #Store msw
add r0
mov r6
take r2
```

```
str r6

imm 2      #Increment data_mem pointer
add r0
mov r0
imm 30          #Determine if finished
cmp r0
bne 6

stop
```

Program 3 Pseudocode

```
// input data_mem[0:32]
// output data_mem[33:35]
void patternSearch(data_mem[0:35][0:7]){
     // cnt_a: number of occurrences of the given 5-bit pattern in any byte
     // cnt_b: number of bytes within which the pattern occurs
     // cnt_c: number of times it occurs anywhere in the string
     int cnt_a = 0, cnt_b = 0, cnt_c = 0;
     // check data_mem[0:31]
     for (int i = 0; i < 32; i++){
          // flag: if data_mem[i] contains pattern, flag = 1; else , flag = 0
          boolean flag = 0;
          for (int j = 0; j < 4; j++){
               // check pattern in each address
               if( data_mem[i][j:j+4] == data_mem[32][3:7] ){
                    cnt_a++;
                    flag = 1;
               }
          }
```

```
            if(flag = 1){
                cnt_b++;
            }
        }
        // check pattern cross addresses
        cnt_c = cnt_a;
        for(int i = 0; i < 31; i++){
            if(data[i][4:7] == data[32][3:6] && data[i+1][0] == data[32][7])
                cnt_c++;
            if(data[i][5:7] == data[32][3:5] && data[i+1][0:1] == data[32][6:7])
                cnt_c++;
            if(data[i][6:7] == data[32][3:4] && data[i+1][0:2] == data[32][5:7])
                cnt_c++;
            if(data[i][7] == data[32][3] && data[i+1][0:3] == data[32][4:7])
                cnt_c++;
        }
        data_mem[33]=cnt_a;
        data_mem[34]=cnt_b;
        data_mem[35]=cnt_c;
}
```

Program 3 Assembly Code

```
# r0  = cnt_a: number of occurrences of the given 5-bit pattern in any byte
# r1  = cnt_b: number of bytes within which the pattern occurs
# r2  = cnt_c: number of times it occurs anywhere in the string
# r3  = temp
# r4  = Outer loop counter i
# r5  = flag of the pattern occurs in the current byte
# r6  = Inner loop counter j
# r7  = data_mem[i]
# r8  = search pattern
```

```
# r9  = data_mem[i+1]
# r10 = temp
# r11 = temp
# r12 = temp
# r13 = temp
# r14 = temp
# r15 = accumulator register


imm 0
mov r0     #r0  = 0
imm 0
mov r1     #r1  = 0
imm 0
mov r2     #r2  = 0
imm 32
mov r3     #r3  = 32
ldr r3     #r15 = data_mem[32]
mov r3     #r3  = data_mem[32]
imm 248
and r3     #r15 = data_mem[32] * b11111000
mov r8     #r8  = search pattern
imm 0
mov r3     #r3  = 0
imm 0
mov r4     #r4  = 0

# loop_outer:
imm 0
mov r5     #r5  = 0
imm 0
mov r6     #r6  = 0   j = 0
```

```
# loop_inner:
ldr r3     #r15 = data_mem[0]
lsl r6     #
mov r7     #r7 = data_mem[0] left shift j
imm 248
and r7     #r15 = data_mem[0] * b11111000
cmp r8
#bne not_matched
bne 11

imm 1
add r0
mov r0
imm 1
add r2
mov r2
imm 1
mov r5

# not_matched:
imm 1
add r6
mov r6     #r6++   j++
imm 4
cmp r6
#bne loop_inner
bne 12

imm 0
cmp r5
#beq skip_cnt_b     # check flag
```

```
beq 13
imm 1
add r1
mov r1

# skip_cnt_b:
ldr r3     #r15 = data_mem[i]
mov r7     #r7 = data_mem[i]
imm 1
add r3
mov r3     #r3++
ldr r3
mov r9     #r9 = data_mem[i+1]
imm 1
add r4
mov r4     #r4++
imm 32
cmp r4
#beq skip_cross:
beq 14

# cross_step_a:
imm 1
mov r10    #r10 = 1
take r7
lsr r10
mov r10    #r10 = data_mem[i] right shift 1
imm 120
and r10
mov r10    #r10 = r0 * b01111000
imm 7
mov r11
```

```
take r9
lsl r11    #r15 = data_mem[i+1] left shift 7
orr r10    #r15 = r15 + r10
cmp r8     #compare with r8
#bne cross_step_b
bne 13
imm 1
add r2
mov r2

# cross_step_b:
imm 2
mov r10    #r10 = 2
take r7
lsr r10
mov r10    #r10 = data_mem[i] right shift 2
imm 56
and r10
mov r10    #r10 = r0 * b00111000
imm 6
mov r11
take r9
lsl r11    #r15 = data_mem[i+1] left shift 6
orr r10    #r15 = r15 + r10
cmp r8     #compare with r8
#bne cross_step_c
bne 13
imm 1
add r2
mov r2

# cross_step_c:
```

```
imm 3
mov r10    #r10 = 3
take r7
lsr r10
mov r10    #r10 = data_mem[i] right shift 3
imm 24
and r10
mov r10    #r10 = r0 * b00011000
imm 5
mov r11
take r9
lsl r11    #r15 = data_mem[i+1] left shift 5
orr r10    #r15 = r15 + r10
cmp r8     #compare with r8
#bne cross_step_d
bne 13
imm 1
add r2
mov r2

# cross_step_d:
imm 4
mov r10    #r10 = 4
take r7
lsr r10
mov r10    #r10 = data_mem[i] right shift 4
imm 8
and r10
mov r10    #r10 = r0 * b00001000
imm 4
mov r11
take r9
```

```
lsl r11    #r15 = data_mem[i+1] left shift 4
orr r10    #r15 = r15 + r10
cmp r8     #compare with r8
#bne loop_outer
bne 13
imm 1
add r2
mov r2
#b loop_outer
b 15

#skip_cross:
imm 33
mov r14
take r0
str r14
imm 34
mov r14
take r1
str r14
imm 35
mov r14
take r2
str r14
stop
```

7. Assembler

File Name: sesoassembler.py

Code:

```
import sys
from bitstring import Bits
opcodes = {
    'lsl': 0, 'lsr': 1, 'add': 2, 'and': 3, 'orr': 4, 'eor': 5,
    'take': 6, 'mov': 7, 'ldr': 8, 'str': 9,  'cmp': 10, 'b': 11,
    'beq': 12, 'bne': 13, 'stop': 14
}
registers = {
    'r0': 0, 'r1': 1, 'r2': 2, 'r3': 3, 'r4': 4, 'r5': 5,
    'r6': 6, 'r7': 7, 'r8': 8,  'r9': 9, 'r10': 10, 'r11': 11,
    'r12': 12, 'r13': 13, 'r14': 14, 'r15': 15
}
LUT = {
    '0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5,
    '6': 6, '7': 7, '8': 8, '9': 9, '10': 10, '11': 11,
    '12': 12, '13': 13, '14': 14, '15': 15
}
if __name__ == "__main__":
    args = sys.argv
    if len(args) != 3:
        sys.exit(0);
    with open(args[1], 'r') as programFile, open(args[2], 'w') as machineCodeFile:
        instr = ''
        for i in programFile:
            i = i.split('#', 1)[0]
            if ':' in i:
```

```
    instr = i.split(':', 1)[1]
else:
    instr = i
words = instr.split()
if len(words) == 0:
    pass
elif len(words) == 1:
    opcode = opcodes[words[0]]
    machineCodeFile.write(format(1, 'b').zfill(1))
    machineCodeFile.write(format(opcode, 'b').zfill(4))
    machineCodeFile.write(format(0, 'b').zfill(4))
    machineCodeFile.write('\n')
elif len(words) == 2:
    if words[0] in ['b','bne','beq']:
        opcode = opcodes[words[0]]
        register = LUT[words[1]]
        machineCodeFile.write(format(1, 'b').zfill(1))
        machineCodeFile.write(format(opcode, 'b').zfill(4))
        machineCodeFile.write(format(register, 'b').zfill(4))
        machineCodeFile.write('\n')
    else:
        if words[0] == 'imm':
            machineCodeFile.write(format(0, 'b').zfill(1))
            imm_num = int(words[1])
            machineCodeFile.write(format(imm_num, 'b').zfill(8))
            machineCodeFile.write('\n')
        else:
            opcode = opcodes[words[0]]
            register = registers[words[1]]
            machineCodeFile.write(format(1, 'b').zfill(1))
            machineCodeFile.write(format(opcode, 'b').zfill(4))
            machineCodeFile.write(format(register, 'b').zfill(4))
```

```
                machineCodeFile.write('\n')
        else:
            exit(0)
```

How to Run:
pip install bitstring
python sesoassembler.py program1.txt machine_code_1.txt
python sesoassembler.py program2.txt machine_code_2.txt
python sesoassembler.py program3.txt machine_code_3.txt

8. Machine Code

Program 1 Machine Code

000000000
101110000
000001111
110100000
111000000
000000001
100000000
101110011
110000011
101110001
000000001
100100011
101110010
110000010
101110010
000000111

100110010
101110100
000000010
100010100
101010100
101110100
000000011
100110100
101110100
000000001
100110100
101111110
000000001
100010100
101010100
101110100
000000001
100110100
101110100
011110000
100110001
101110101
000000011
100000101
101010101
101110101
011100000
100110101
101110101
000000010
100000101
101010101

101110101
011000000
100110101
101110101
000000001
100000101
101010101
101110101
010000000
100110101
101110101
000000111
100010101
101111101
000000111
100010101
101010100
101110101
010000000
100110001
101110110
000000111
100010110
101010100
101110110
000001000
100110001
101110111
000000011
100010111
101010110
101110110

000000100
100110001
101110111
000000010
100010111
101010110
101110110
000000010
100110001
101110111
000000001
100010111
101010110
101110110
000000110
100110010
101110111
000000001
100010111
101110111
000000001
100010111
101010111
101110111
000000001
100110111
101110111
001000000
100110001
101111000
000000110
100011000

101010111
101110111
000100000
100110001
101111000
000000101
100011000
101010111
101110111
000001000
100110001
101111000
000000011
100011000
101010111
101110111
000000100
100110001
101111000
000000010
100011000
101010111
101110111
000000001
100110001
101111000
101010111
101110111
001000000
100110001
101111000
000000110

100011000
101011110
101111000
000010000
100110001
101111001
000000100
100011001
101011000
101111000
000001000
100110001
101111001
000000011
100011001
101011000
101111000
000000010
100110001
101111001
000000001
100011001
101011000
101111000
000000001
100110001
101111001
101011000
101111000
000001111
100110001
101111001

000000011
100011001
101011001
101111001
000000111
100111001
101111001
000000010
100011001
101011001
101111001
000000011
100111001
101111001
000000001
100011001
101011001
101111001
000000001
100111001
101111001
101101101
101011001
101111001
101100100
101011001
101111001
101100101
101011001
101111001
101100110
101011001

101111001
101100111
101011001
101111001
101101000
101011001
101111001
000000101
100000010
101111010
000000011
100010001
101111011
000011110
100111011
101111011
101101010
101001011
101111011
101100101
101001011
101111010
000000100
100000001
101111011
011100000
100111011
101111011
000000100
100000110
101111100
101101011

101001100
101111011
000000011
100000001
101111100
000001000
100111100
101111100
101101011
101001100
101111011
000000010
100000111
101111100
101101011
101001100
101111011
000000001
100001000
101111100
101101011
101001100
101111011
101101001
101001011
101111011
000011110
100100011
101111100
101101011
110011100
000011111

100100011
101111101
101101010
110011101
000000001
100100000
101110000
110110001
111100000

Program 2 Machine Code

000000000
101110000
000011110
100100000
101110001
110000001
101110001
000011111
100100000
101110010
110000010
101110010
000000100
100010010
101110011
101100011
101010010
101110100

000001111
100110100
101110100
000000010
100010100
101110011
101100100
101010011
101110100
000000011
100110100
101110100
000000001
100010100
101110011
101100100
101010011
101110100
000000001
100110100
101110100
101100100
101110101
000000100
100010001
101110011
000000100
100010010
101110100
101100100
101010011
101110100

000000010
100010100
101110011
101100100
101010011
101110100
000000011
100110100
101110100
000000001
100010100
101110011
101100100
101010011
101110100
000000001
100110100
101110100
000000001
100000100
101110100
101100100
100100101
101110101
000110011
100110001
101110011
000110011
100110010
101110100
101100100
101010011

101110100
000000100
100010100
101110011
101100100
101010011
101110100
000000011
100110100
101110100
000000001
100010100
101110011
101100100
101010011
101110100
000000001
100110100
101110100
000000010
100000100
101110100
101100100
100100101
101110101
010101010
100110001
101110011
010101010
100110010
101110100
101100100

101010011
101110100
000000100
100010100
101110011
101100100
101010011
101110100
000001111
100110100
101110100
000000010
100010100
101110011
101100100
101010011
101110100
000000010
100110100
101110100
000000010
100000100
101110100
101100100
100100101
101110101
101100001
101110011
101100010
101110100
101100100
101010011

101110100
000000100
100010100
101110011
101100100
101010011
101110100
000001111
100110100
101110100
000000010
100010100
101110011
101100100
101010011
101110100
000000011
100110100
101110100
000000001
100010100
101110011
101100100
101010011
101110100
000000001
100110100
101110100
000000100
100000100
101110100
101100100

100100101
101110101
000000011
100010001
101110011
000000001
100110011
101110011
000000100
100010001
101110100
000001110
100110100
101110100
101100100
101000011
101110011
000000011
100000010
101110100
011110000
100110100
101110100
101100100
101000011
101110001
000000101
100010010
101110010
000000000
110100101
111000011

000000100
100010101
101110011
000000001
110100011
111000100
010000000
100100010
101110010
110110101
001000000
100100010
101110010
000011100
110100101
111000010
110110111
000011010
110100101
111000010
110110111
000010110
110100101
111000010
110110111
000011110
110100101
111000010
110110111
000011001
110100101
111000010

110110111
000010101
110100101
111000010
110110111
000011101
110100101
111000010
110110111
000010011
110100101
111000010
110110111
000011011
110100101
111000010
110110111
000010111
110100101
111000010
110110111
000011111
110100101
111000010
000000001
101010001
101110001
110111101
000000010
101010001
101110001
110111101

000000100
101010001
101110001
110111101
000001000
101010001
101110001
110111101
000010000
101010001
101110001
110111101
000100000
101010001
101110001
110111101
001000000
101010001
101110001
110111101
010000000
101010001
101110001
110111101
000000001
101010010
101110010
110111101
000000010
101010010
101110010
110111101

000000100
101010010
101110010
101100001
110010000
000000001
100100000
101110110
101100010
110010110
000000010
100100000
101110000
000011110
110100000
111010110
111100000

Program 3 Machine Code

000000000
101110000
000000000
101110001
000000000
101110010
000100000
101110011
110000011
101110011

011111000
100110011
101111000
000000000
101110011
000000000
101110100
000000000
101110101
000000000
101110110
110000011
100000110
101110111
011111000
100110111
110101000
111011011
000000001
100100000
101110000
000000001
100100010
101110010
000000001
101110101
000000001
100100110
101110110
000000100
110100110
111011100

000000000
110100101
111001101
000000001
100100001
101110001
110000011
101110111
000000001
100100011
101110011
110000011
101111001
000000001
100100100
101110100
000100000
110100100
111001110
000000001
101111010
101100111
100011010
101111010
001111000
100111010
101111010
000000111
101111011
101101001
100001011
101001010

110101000
111011101
000000001
100100010
101110010
000000010
101111010
101100111
100011010
101111010
000111000
100111010
101111010
000000110
101111011
101101001
100001011
101001010
110101000
111011101
000000001
100100010
101110010
000000011
101111010
101100111
100011010
101111010
000011000
100111010
101111010
000000101

101111011
101101001
100001011
101001010
110101000
111011101
000000001
100100010
101110010
000000100
101111010
101100111
100011010
101111010
000001000
100111010
101111010
000000100
101111011
101101001
100001011
101001010
110101000
111011101
000000001
100100010
101110010
110111111
000100001
101111110
101100000
110011110

000100010
101111110
101100001
110011110
000100011
101111110
101100010
110011110
111100000

9. Changelog

Milestone 3

a. Added assembler
b. Updated programs
c. Added machine codes

Milestone 2

d. Added Individual Component Specification Part
e. Removed sign-extend in the architecture overview
f. Changed branch (b, beq, bne) to b-type

Milestone 1

g. Initial version