MiniMA Architecture

0. Team

Members: Thanh Phan, Thu Mai

1. Introduction

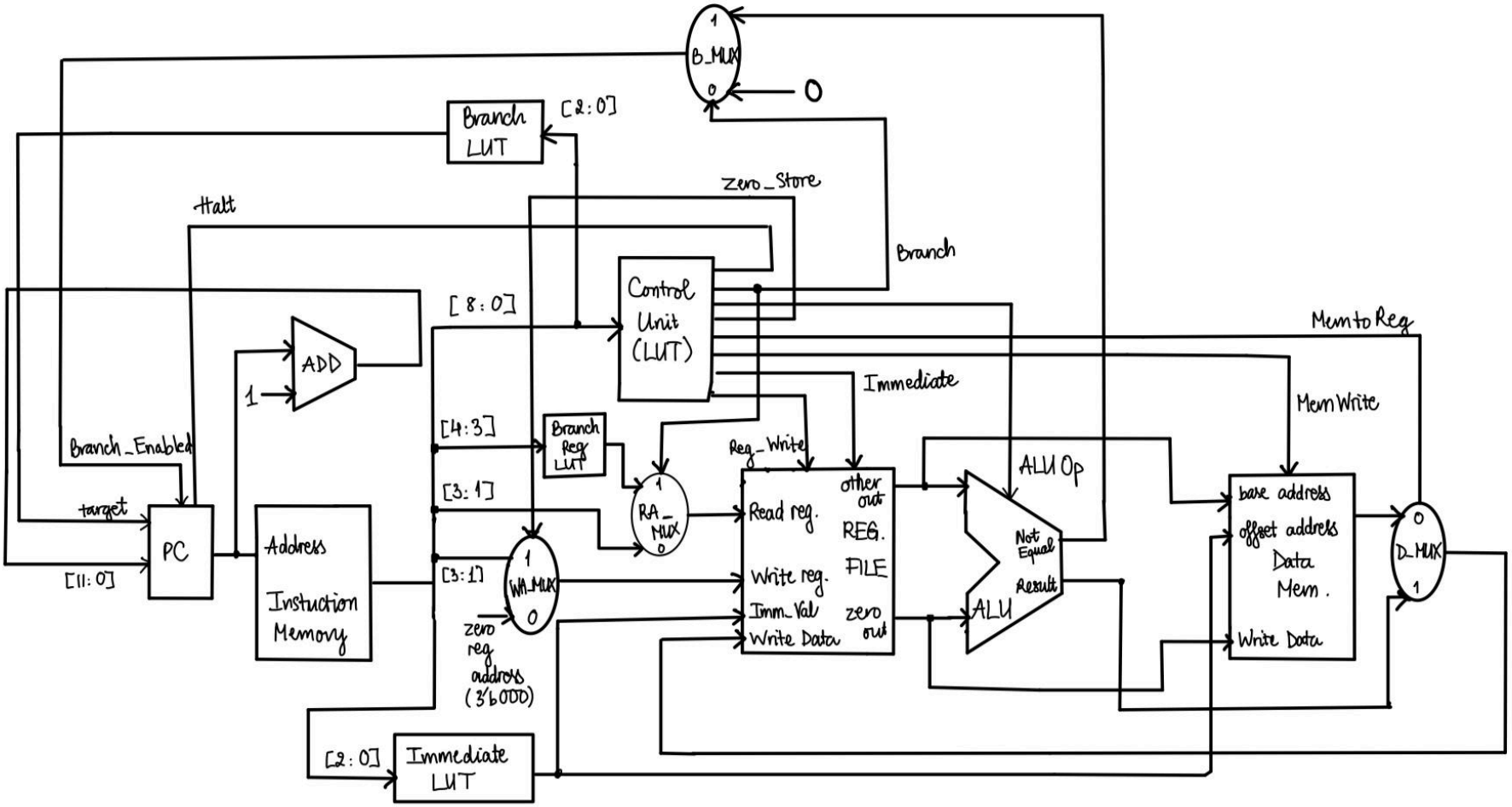
Name: MiniMA (Minimal Machine Architecture)

Overall philosophy: MiniMA is a design of ISA that narrow down number of bits using, 9-bit architecture, with purpose to prioritize the simplicity in memory but ensure the efficiency

Goals: The architecture should be easy to implement while support sufficient number of operations, ensure the performance given limited amount of memory, testing using the supported three programs

Classify the machine: Combination of load-store and accumulator machine

2. Architectural Overview



3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	2 bit type, 3 bits opcode, 3 bit operand register, 1 bit storage register	add, sub, xor, and, or, lsl, lsr, mov
Mem	2 bit type, 3 bit register source, 3 bit offset, 1 bit operation type	sb, lb
B	2 bit type, 2 bit operand register, 2 bit operand register, 3 bit target address	bne
IH	2 bit type 3 bit operand register, 3 bit offset, 1 bit operation type	imm, done

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
and = bitwise and	R	2 bit type (00) 3 bits opcode (000) 3 bit operand register (XXX) 1 bit storage register (X)	# Assume R0 has `0b0001_0001` # Assume R2 has `0b1001_0000` `and R2, R0, R2` ⇔ `00_000_010_1` # after and instruction, R2 now holds `0b0001_0000`	When performing operations, one of the address pointers always reads from register 0. If the 1-bit storage operand = 0, we will store it to R0. If the 1-bit storage operand = 1, we will store it to the other operand register.
or = bitwise or	R	2 bit type (00) 3 bits opcode (001) 3 bit operand register (XXX) 1 bit storage register (X)	# Assume R0 has `0b0001_0001` # Assume R2 has `0b1001_0000` `or R0, R0, R2` ⇔ `00_001_001_0` # after or instruction, R0 now holds `0b1001_0001`	When performing operations, one of the address pointers always reads from register 0. If the 1-bit storage operand = 0, we will store it to R0. If the 1-bit storage operand = 1, we will store it to the other operand register.
add = add two register	R	2 bit type (00) 3 bits opcode (010) 3 bit operand register (XXX) 1 bit storage register (X)	# Assume R0 has `0b0001_0001` # Assume R4 has `0b1001_0000` `add R0, R0, R4` ⇔ `00_010_011_0` # after add instruction, R0 now holds `0b1011_0001`	When performing operations, one of the address pointers always reads from register 0. If the 1-bit storage operand = 0, we will store it to R0. If the 1-bit storage operand = 1, we will store it to the other operand register.
sub = subtract two registe	R	2 bit type (00) 3 bits opcode (011) 3 bit operand register (XXX) 1 bit storage register (X)	# Assume R0 has `0b0001_0001` # Assume R3 has `0b0001_0000` `sub R0, R0, R3` ⇔	When performing operations, one of the address pointers always reads from register 0.

rs			`00_011_010_0` # after sub instruction, R0 now holds `0b0000_0001`	If the 1-bit storage operand = 0, we will store it to R0. If the 1-bit storage operand = 1, we will store it to the other operand register.
xor = bitwise xor	R	2 bit type (00) 3 bits opcode (100) 3 bit operand register (XXX) 1 bit storage register (X)	# Assume R0 has `0b0001_0001` # Assume R5 has `0b1001_0000` `add R5, R0, R5` ⇔ `00_100_100_1` # after xor instruction, R5 now holds `0b1000_0001`	When performing operations, one of the address pointers always reads from register 0. If the 1-bit storage operand = 0, we will store it to R0. If the 1-bit storage operand = 1, we will store it to the other operand register.
lsl = logical shift left	R	2 bit type (00) 3 bits opcode (101) 3 bit operand register (XXX) 1 bit storage register (X)	# Assume R2 has `0b0001_0001` `lsl R1, R2` ⇔ `00_101_010_1` # after lsl instruction, R1 now holds `0b0010_0010`	When performing operations, one of the address pointers always reads from register 0. If the 1-bit storage operand = 0, we will store it to R0. If the 1-bit storage operand = 1, we will store it to the other operand register. Only shift to the left 1 bit at a time, and do not need the second operand since we will always read from register 0
lsr = logical right shift	R	2 bit type (00) 3 bits opcode (110) 3 bit operand register (XXX) 1 bit storage register (X)	# Assume R0 has `0b0001_0001` `lsr R1, R0` ⇔ `00_110_000_1` # after lsr instruction, R1 now holds `0b0000_1000`	When performing operations, one of the address pointers always reads from register 0. If the 1-bit storage operand = 0, we will store it to R0. If the 1-bit storage operand = 1, we will store it to the other operand register.

				Only shift to the right 1 bit at a time, and do not need the second operand since we will always read from register 0
mov = move	R	2 bit type (00) 3 bits opcode (111) 3 bit operand register (XXX) 1 bit storage register (X)	# Assume R0 has `0b0001_0001` `mov R1, R0` ⇔ `00_111_000_1` # after mov instruction, R1 now holds `0b0001_0001`	When performing operations, one of the address pointers always reads from register 0. If the 1-bit storage operand = 0, we will store it to R0. If the 1-bit storage operand = 1, we will store it to the other operand register.
lb = load word	Mem	2 bit type (01) 3 bit offset (XXX) 3 bit register source (XXX) 1 bit operation type (0)	# Assume R0 has `0b0001_0001` # Assume data_mem[R5+5] has `0b0011_0001` `lb R0, 5[R5]` ⇔ `01_101_101_0` # after lb instruction, R0 now holds `0b0011_0001`	Always store the loaded data from memory to R0 since we are always reading from R0 in the R instructions.
sb = store word	Mem	2 bit type (01) 3 bit offset (XXX) 3 bit register source (XXX) 1 bit operation type (1)	# Assume R0 has `0b0001_0001` # Assume data_mem[R5+1] has `0b0011_0001` `sb R0, 1[R5]` ⇔ `01_001_101_1` # after sb instruction, data_mem[R5+1] now holds `0b0001_0001`	The intended value to store back to the memory is always at R0 before the sb instruction.
bne =	B	2 bit type (10)	# Assume R7 has `0b0001_0001`	We will dedicate R0,R5,R6,R7 to be

branch if not equal		2 bit operand register (XX) 2 bit operand register (XX) 3 bit target address (XXX)	# Assume R8 has `0b1001_0000` `bne R7, R8, 'target'` ⇔ `10_10_11_xxx` # after bne instruction, if R7 != R8, program counter set to 'target'	available for 'bne' instruction thus we only need 2 bits for this instruction. We also will use a branch LUT, thus we will get the actual target label. A branch flag will be set to 1 if R7 == R8 else nothing
imm = populate immediate to register	IH	2 bit type (11) 3 bit offset (XXX) 3 bit operand register (XXX) 1 bit operation type (0)	# Assume R0 has `0b0001_0001` `imm R0, 1`⇔ `11_001_000_0` # after imm instruction, R0 now holds `0b0000_0001`	The bit offset is the index bit in the Immediate LUT storing the actual value
halt	IH	Unique sequence: 11_111111_1	`halt` ⇔ `11_111111_1`	The HALT flag will be set high which signals PC to set Done flag high

Internal Operands

How many registers are supported?
We are supporting 8 registers in total: (R1-R7) and R0.

Is there anything special about any of the registers (e.g. constant, accumulator), or all of them general purpose?
R1-R7 serves as the general purpose registers and R0 whose functionality is adapted from the accumulator.

Control Flow (branches)

What types of branches are supported?

bne - branch if not equal: if the two provided register are not equal to each other, branch to the specified target according to the Branch LUT

How are the target addresses calculated?
The targets are being stored in the Branch LUT

What is the maximum branch distance supported?
For now, we have not decided yet. Currently in our Branch LUT we are supporting 5 temporary placeholders for different 5 Targets branches.

Addressing Modes

What memory addressing modes are supported, e.g. direct, indirect? How are addresses calculated? Give examples.
To access the memory, we are using the displacement, such as “lb R0, 5[R5]” from the base address of R5 + 5 to jump.

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.

Since our design adapts both the load-store and accumulator machine, the programmer can use this design more flexibly in contrast with the restriction on the machine code bit length which is 9-bit. Our goal for this design is the ability to allow the programmer to retrieve data from the data memory in a more efficient way in complement with the needed calculation, thus the desired approach is to be loading and writing to memory in between calculation steps. Additional note on our design, based on certain instructions the number of allowed registers to use can vary. For example, when the programming uses the ‘lb' , the data should always be stored to the zero register, this is similar to the functionality of accumulator. There are some variations on the usage of register which we can find more information in our Operation section.

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 copy all the instructions/operation from MIPS or ARM ISA because they are not compatible with our design due to the limit on the length of machine code which does not give enough room to fully use the instruction set from either MIPS or ARM. To overcome this, we adapt some of the necessary instructions for our design and change the bit rule for each instruction to fit to our restrain 9-bit machine code.

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?
Yes, there is a default state NOOP which will always set the output to be the initial value of zero register. For the memory address pointer calculations, our current design takes in the result NOOP result and perform the addition with the offset immediate retirieved from the Immediate_LUT in the Data_Mem module.

5. Individual Component Specification

a. Top Level

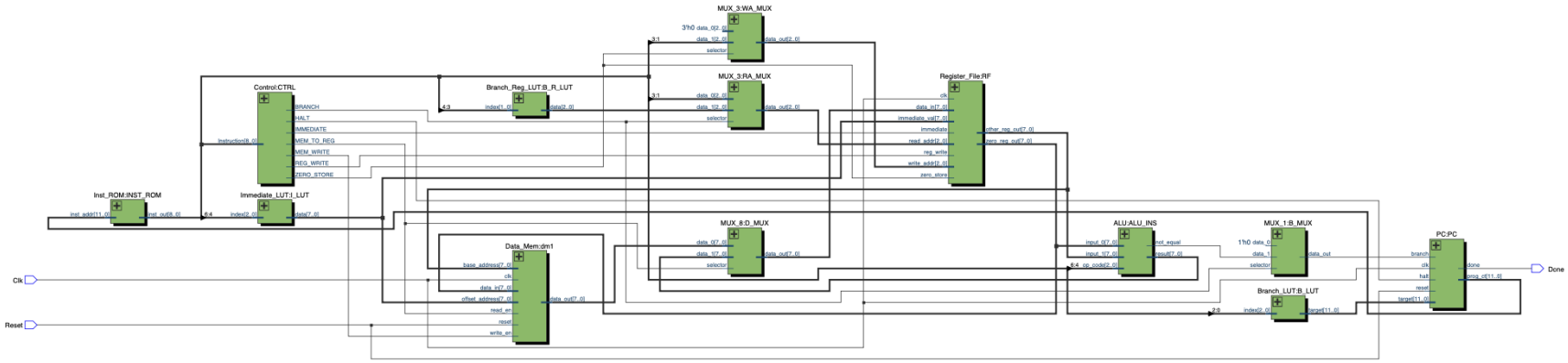
Module file name:

top_level.sv

Functionality Description

Connect all the components of this architecture design

Schematic



b. Program Counter

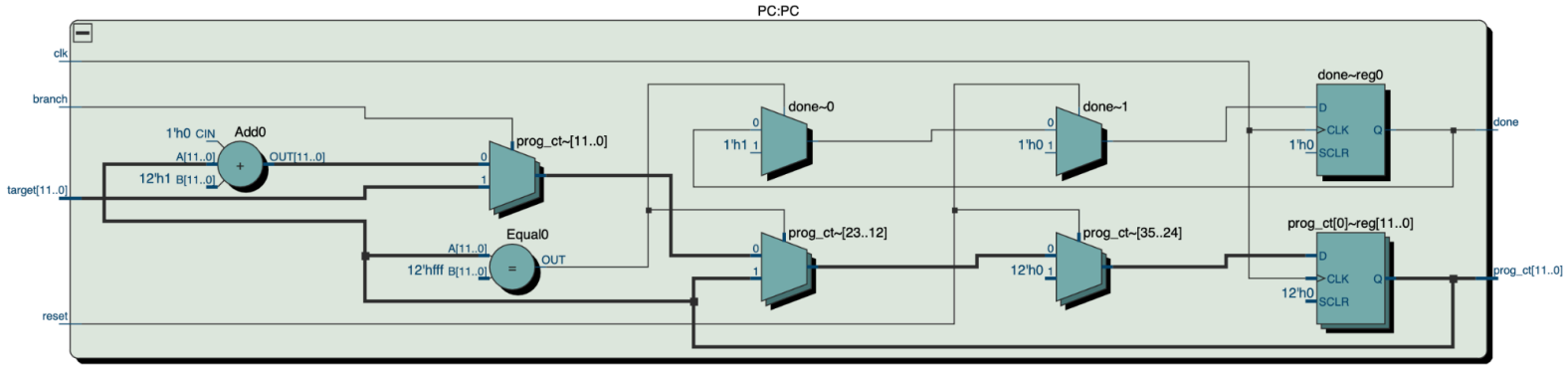
Module file name:

PC.sv

Functionality Description:

- When Reset, set the program counter back to 0
- Increase the program counter by 1 for each cycle
- When branching, this set the program counter to the target address which is retrieved from the Branch_LUT

Schematic



c. Instruction Memory

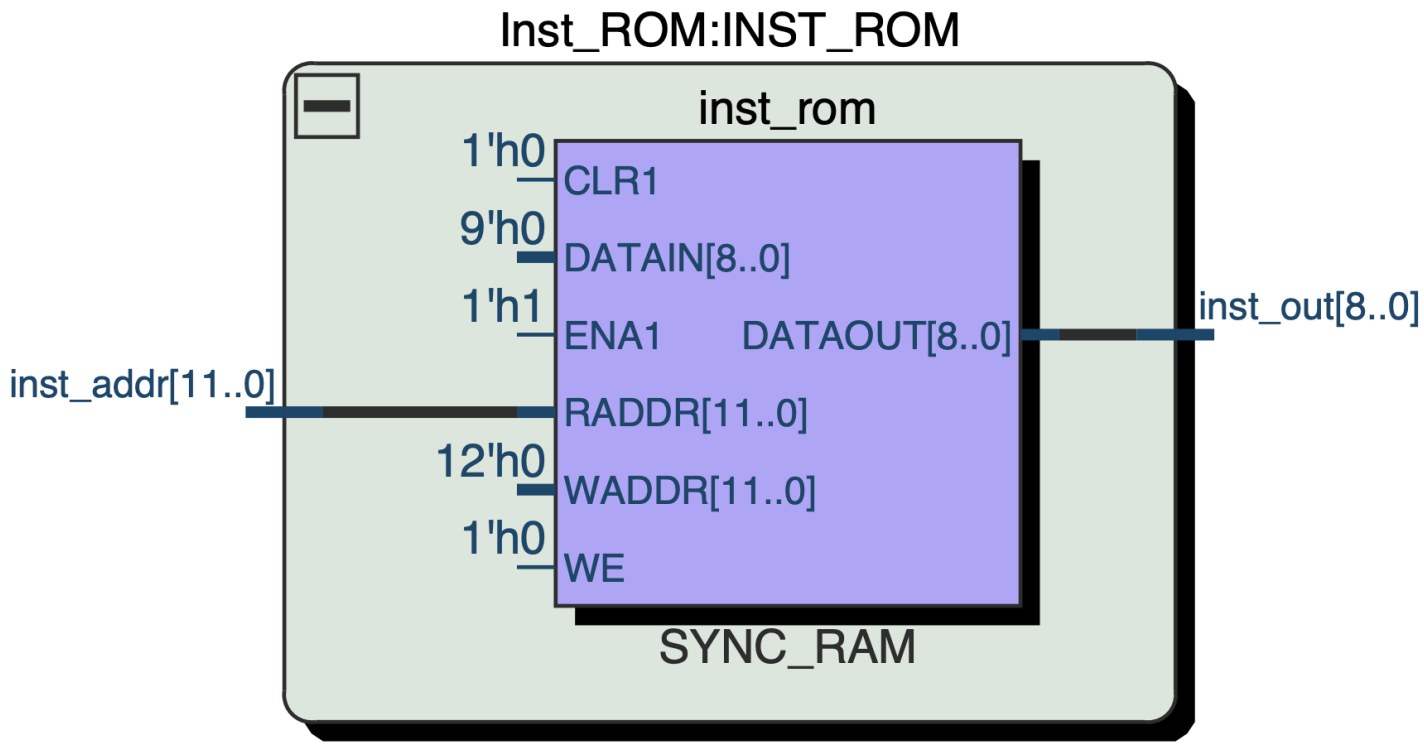
Module file name:

Inst_ROM.sv

Functionality Description:

Retrieve the encoded machine code to determine which instruction should be executed

Schematic


Inst_ROM:INST_ROM
inst_rom
1'h0
CLR1
9'h0
DATAIN[8..0]
1'h1
ENA1
DATAOUT[8..0]
inst_out[8..0]
inst_addr[11..0]
RADDR[11..0]
12'h0
WADDR[11..0]
1'h0
WE
SYNC_RAM

d. Control Decoder

Module file name:

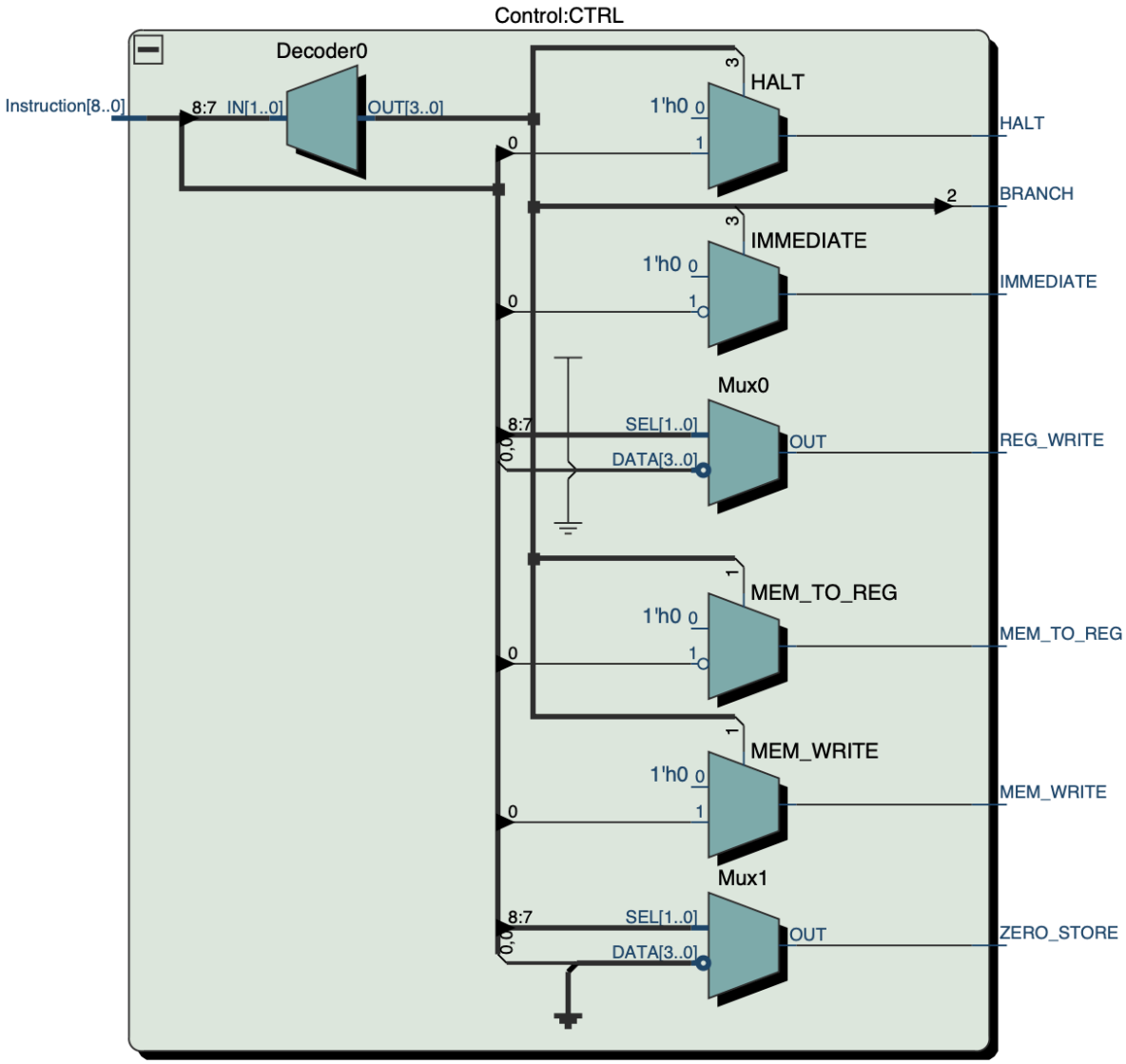
Control.sv

Functionality Description

Decode the given instruction from Inst_ROM:

- The first two bit of the machine code gives us the type of instruction
- This module also decides based on the last bit of the machine code to give out the desired register for storing

Schematic


Control:CTRL
Decoder0
Instruction[8..0]
8:7
IN[1..0]
OUT[3..0]
HALT
1'h0
0
1
3
HALT
2
BRANCH
IMMEDIATE
IMMEDIATE
Mux0
SEL[1..0]
DATA[3..0]
OUT
REG_WRITE
0,0
MEM_TO_REG
MEM_TO_REG
MEM_WRITE
MEM_WRITE
Mux1
ZERO_STORE

e. Register File

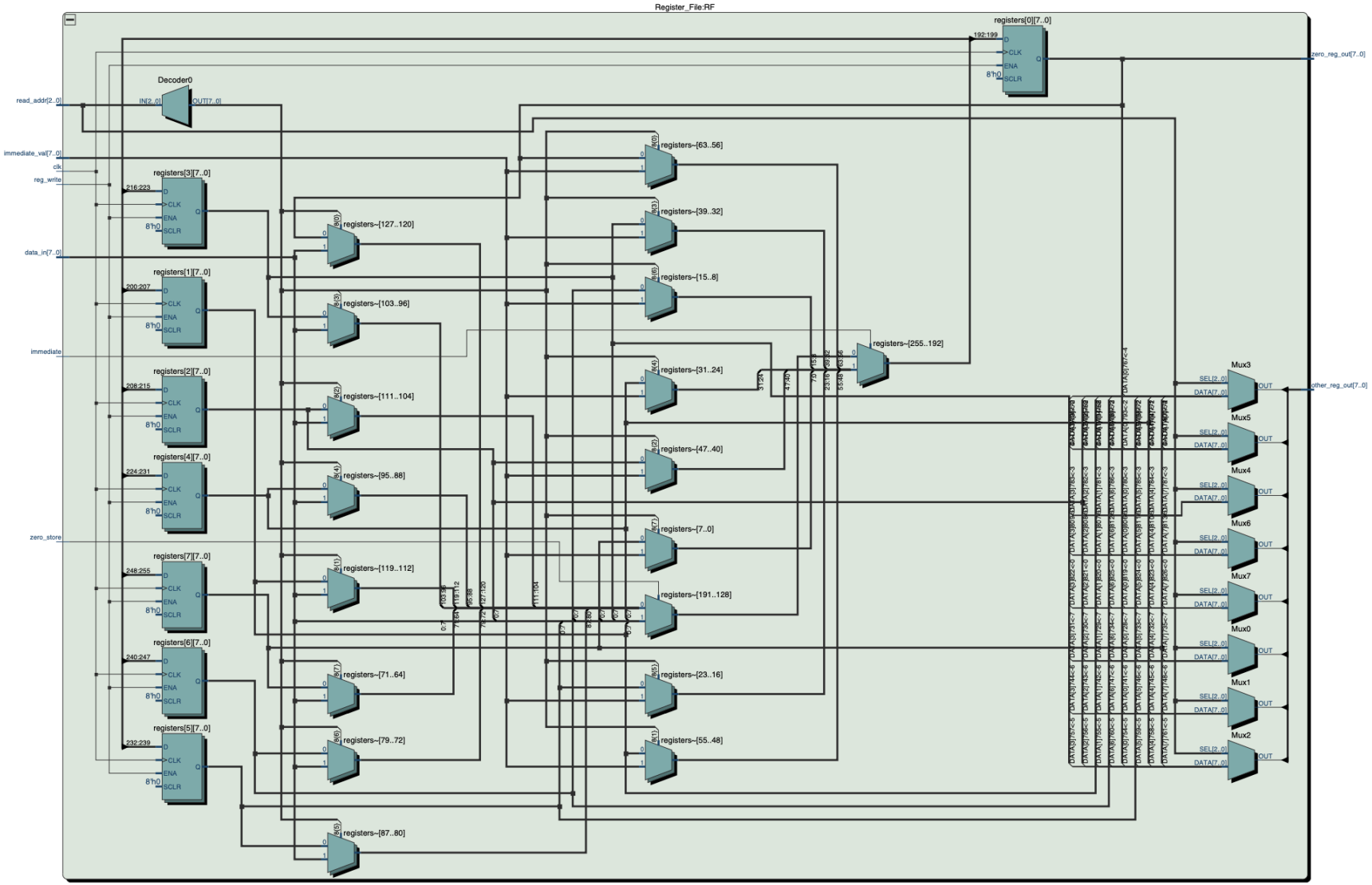
Module file name:

Register_File.sv

Functionality Description

This module includes a total 8 registers: zero-register serves as the accumulator and other 7 general purpose registers. Based on the given address, this module always reads at zero-register and based on the given address, it also outputs the desired register's value.

Schematic


Register_File:RF
Decoder0
IN[2..0]
OUT[7..0]
read_addr[2..0]
immediate_val[7..0]
clk
reg_write
data_in[7..0]
immediate
zero_store
registers[0][7..0]
192:199
zero_reg_out[7..0]
registers[3][7..0]
216:223
registers[1][7..0]
200:207
registers[2][7..0]
208:215
registers[4][7..0]
224:231
registers[7][7..0]
248:255
registers[6][7..0]
240:247
registers[5][7..0]
232:239
registers~[127..120]
registers~[103..96]
registers~[111..104]
registers~[95..88]
registers~[119..112]
registers~[71..64]
registers~[79..72]
registers~[87..80]
registers~[63..56]
registers~[39..32]
registers~[15..8]
registers~[31..24]
registers~[47..40]
registers~[7..0]
registers~[191..128]
registers~[23..16]
registers~[55..48]
registers~[255..192]
Mux3
Mux5
Mux4
Mux6
Mux7
Mux0
Mux1
Mux2
SEL[2..0]
DATA[7..0]
OUT
other_reg_out[7..0]

f. ALU (Arithmetic Logic Unit)

Module file name:

ALU.sv

Functionality Description

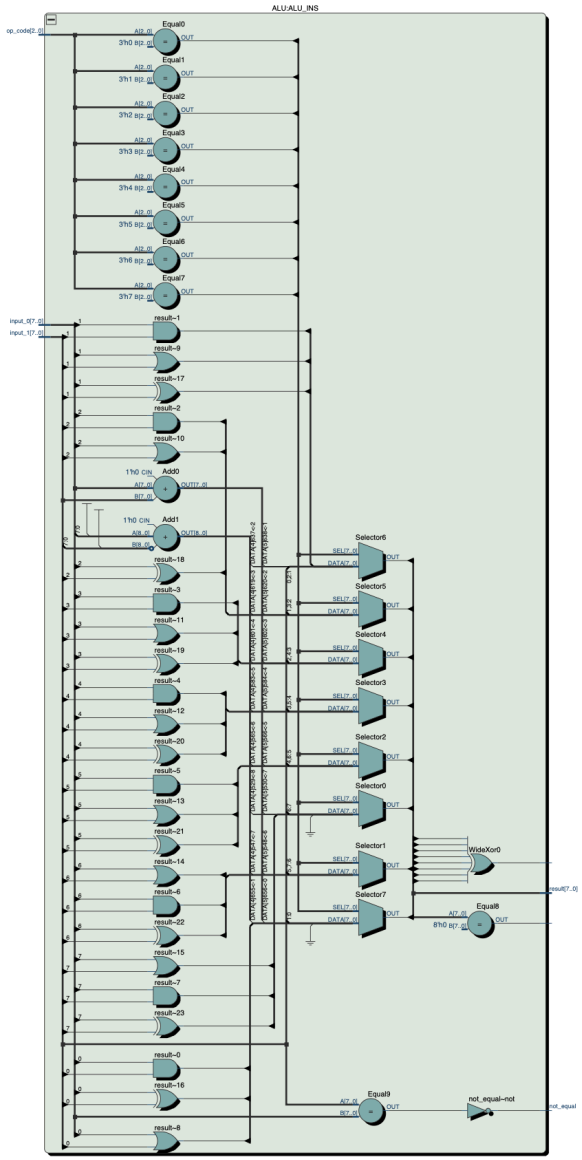
This module takes in 2 inputs to perform arithmetic operations and outputs 1 result.
The zero flag is high if the result is zero, otherwise not.

ALU Operations

We are demonstrating these operations:

- AND: bitwise and
- OR: bitwise or
- XOR: reduction xor
- ADD: addition
- SUB: subtraction
- LSR: logical shift right
- LSL: logical shift left

Schematic


ALU:ALU_INS
op_code[2..0]
input_0[7..0]
input_1[7..0]
Equal0
Equal1
Equal2
Equal3
Equal4
Equal5
Equal6
Equal7
Add0
Add1
Selector6
Selector5
Selector4
Selector3
Selector2
Selector0
Selector1
Selector7
WideXor0
Equal8
Equal9
not_equal~not
result[7..0]
not_equal

g. Data Memory

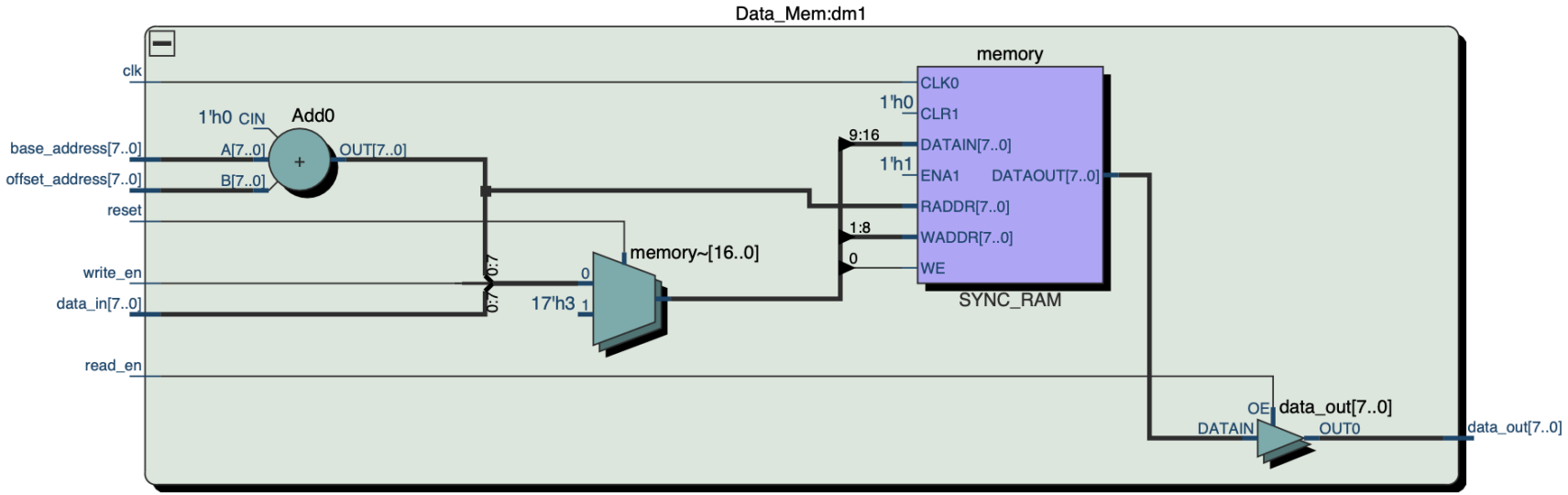
Module file name:

Data_Mem.sv

Functionality Description

This module stores the data information to be performed based on the task. In addition, this module is also where the desired output is stored.

Schematic



h. Immediate Lookup Table

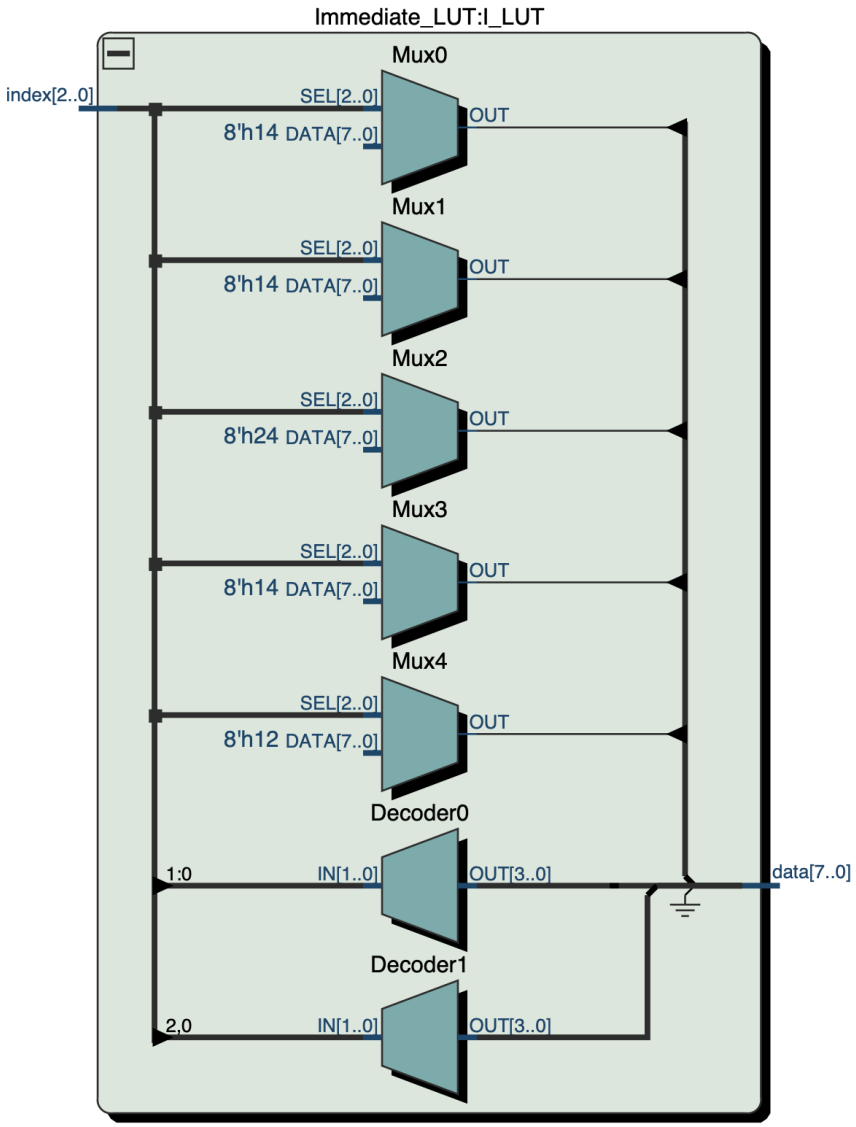
Module file name:

Immediate_LUT.sv

Functionality Description

This module is where we access to retrieve an immediate using the index address that being extracted from our 0-bit machine code from Inst_ROM.

Schematic


Immediate_LUT:I_LUT
Mux0
index[2..0]
SEL[2..0]
8'h14 DATA[7..0]
OUT
Mux1
SEL[2..0]
8'h14 DATA[7..0]
OUT
Mux2
SEL[2..0]
8'h24 DATA[7..0]
OUT
Mux3
SEL[2..0]
8'h14 DATA[7..0]
OUT
Mux4
SEL[2..0]
8'h12 DATA[7..0]
OUT
Decoder0
1:0
IN[1..0]
OUT[3..0]
data[7..0]
Decoder1
2,0
IN[1..0]
OUT[3..0]

i. Branch Lookup Table

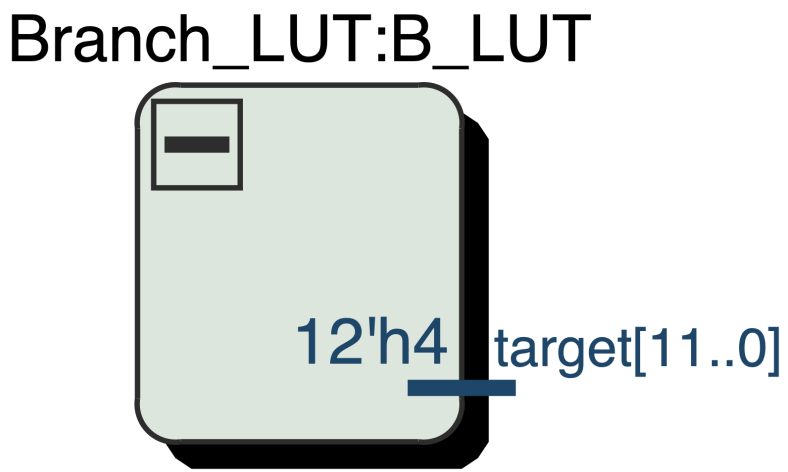
Module file name:

Branch_LUT.sv

Functionality Description

This module is where we access to retrieve a branch target address using the for the program counter when branching is needed.

Schematic



j. MUX_1

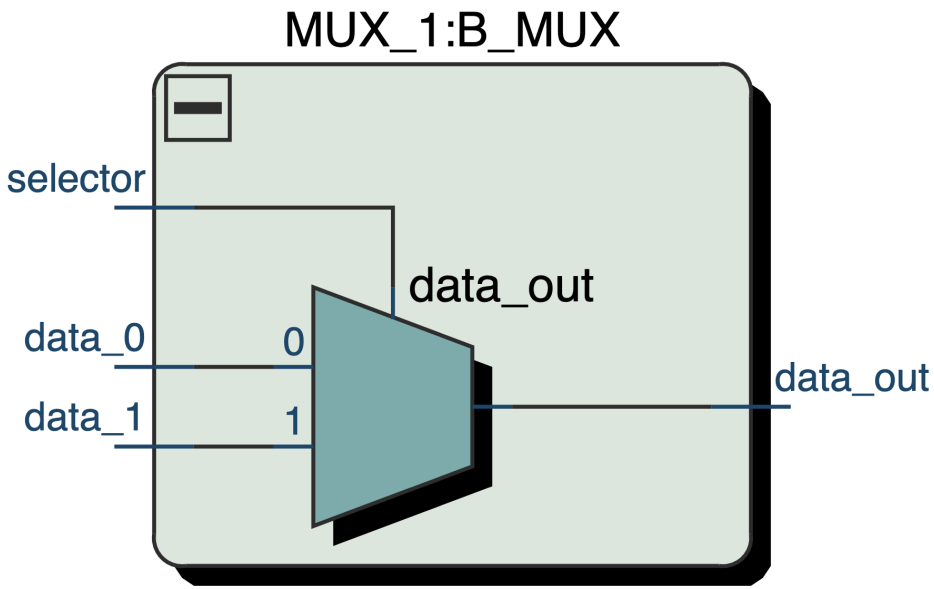
Module file name:

MUX_1.sv

Functionality Description

This module uses a binary selector to output whether a branch target address or the program counter that is incremented by 1. The selector for this MUX is the Branch flag being decoded by the Control module.

Schematic



k. Data Write MUX

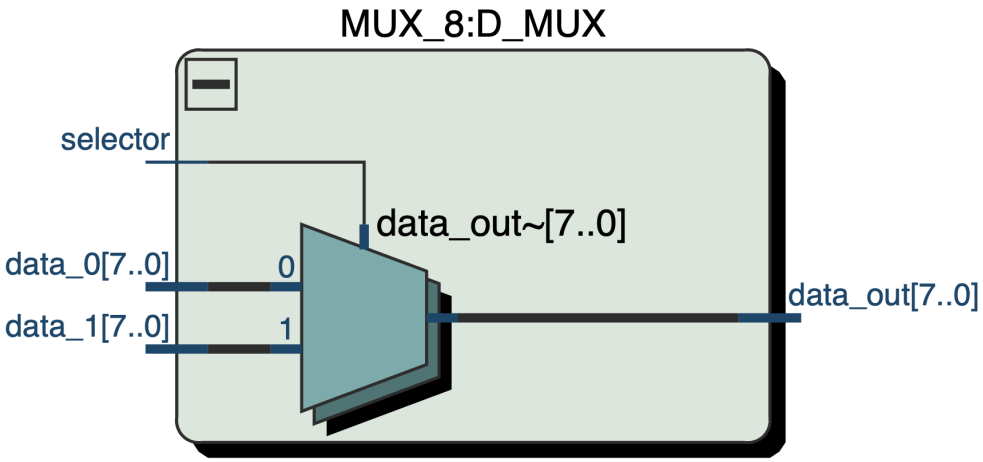
Module file name:

MUX_8.sv

Functionality Description

This module uses a binary selector to output whether the data loaded from the Data Memory module or the result from the ALU module. The selector for this MUX is the Mem_To_Reg flag being decoded by the Control module.

Schematic



I. Read Address MUX

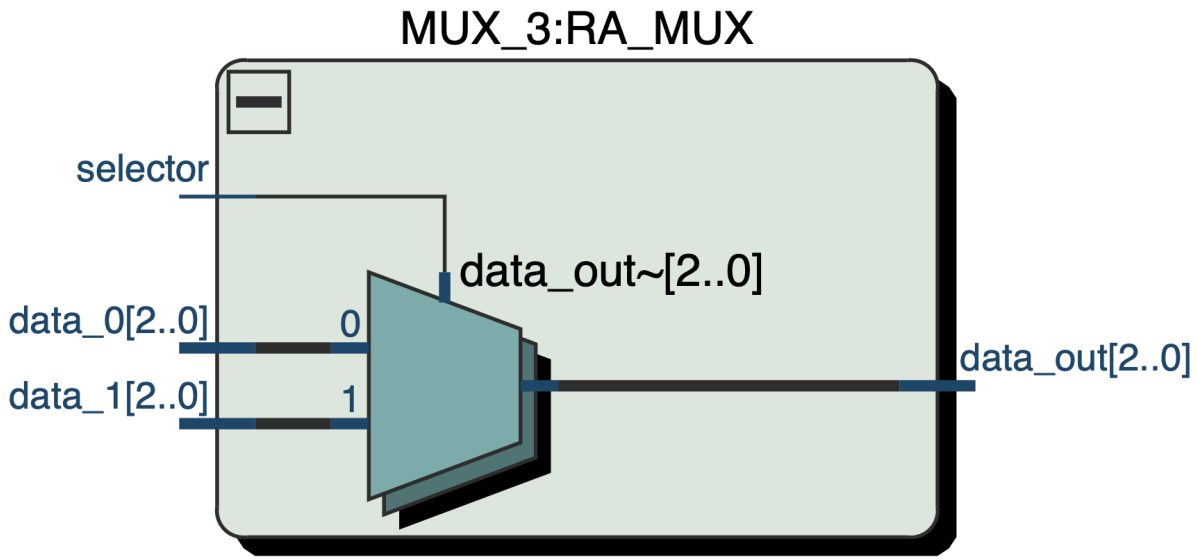
Module file name:

MUX_3.sv

Functionality Description

This MUX is used to determine the write_addr in Register File which takes two 8-bit inputs.

Schematic



m.Write Address MUX

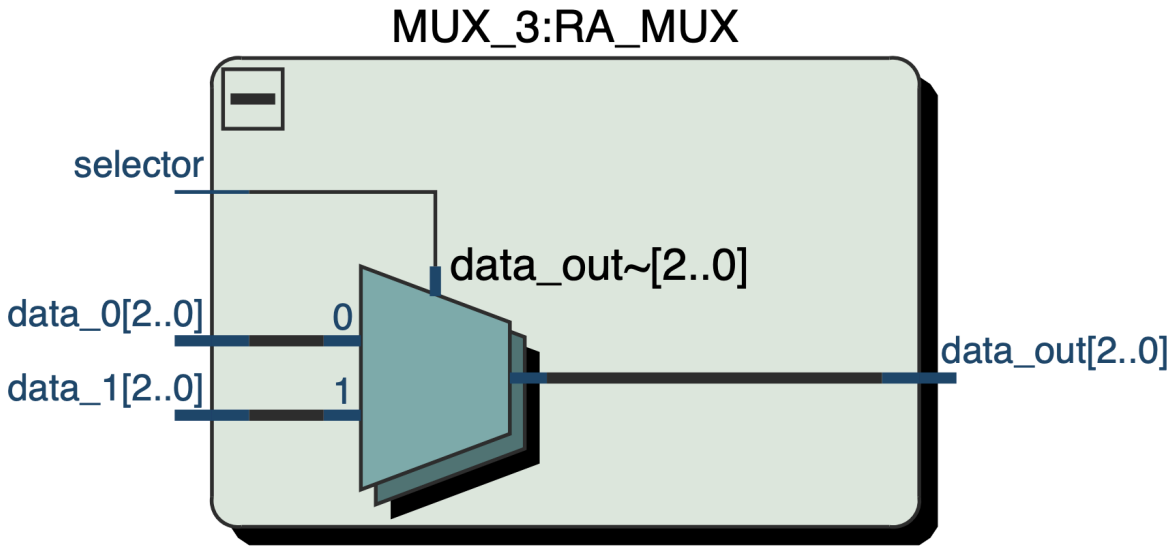
Module file name:

MUX_3.sv

Functionality Description

This module uses a binary selector to output whether the address of one of the general purpose registers or the address of the zero register, to decide where to store the result of the ALU. The selector for this MUX is the Zero_Store flag being decoded by the Control module.

Schematic



6. Program Implementation

Given a series of fifteen 11-bit message blocks in data mem[0:29], generate the corresponding 16-bit encoded versions and store these in data mem[30:59].

Solution scripting:

```
IMM r1 29
loop:
   IMM r6 128       // Load data mem input index
   LB r0 [r6]
   MOV r2 r0
   LB r0 1[r2]      // Load the first half 8-bit given message from data mem: 0_0_0_0_0_b11_b10_b9
   MOV r4 r0        // Store the first half 8-bit for temporary
   LB r0 [r2]       // Load the second half 8-bit given message from data mem: b8_b7_b6_b5_b4_b3_b2_b1
   MOV r5 r0        // Store the second half 8-bit fOR tempORary
   IMM r0 1         // Load immediate 1 for increment
   ADD r2 r0 r2     // Increment the data mem index
   ADD r0 r0 r2
   SB r0 [r6]       // Store back data mem input index to memory for next iteration
   MOV r0 r4        // Calculating p8
   LSR r0 r0        // Right shift to get b11, r6 now holds: 0_0_0_0_0_0_b11_b10
   LSR r0 r0        // Right shift to get b11, r6 now holds: 0_0_0_0_0_0_0_b11
   MOV r2 r0
   MOV r0 r4        // Right shift to get b10, r6 now holds: 0_0_0_0_0_0_b11_b10
   LSR r0 r0
   LSL r0 r0        // Left shift to get b10, r6 now holds: b10_0_0_0_0_0_0_0
   LSL r0 r0
   LSL r0 r0
   LSL r0 r0
   LSL r0 r0
```

```
LSL r0 r0
LSL r0 r0
LSR r0 r0       // Right shift 7 times to get 0_0_0_0_0_0_0_b10
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0    // XOR r2 and r0, r2 now holds: b11^b10
MOV r0 r4       // Left shift r4 to get: b11_b10_b9_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
MOV r3 r0       // Store b11_b10_b9_0_0_0_0_0 to r3, r3 now holds: b11_b10_b9_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSR r0 r0       // Right shift 7 time to get: 0_0_0_0_0_0_0_b9
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0    // XOR r2 and r0, r2 now holds: b11^b10^b9
MOV r0 r5
LSR r0 r0       // Right shift r5 4 times to get: 0_0_0_0_b8_b7_b6_b5
LSR r0 r0
```

```
LSR r0 r0
LSR r0 r0
LSL r0 r0        // Left shift r0 once to get: 0_0_0_b8_b7_b6_b5_0
OR r3 r3 r0      // OR with r3 to get: b11_b10_b9_b8_b7_b6_b5_0
LSR r0 r0        // Continue shifting, to get r0: 0_0_0_0_0_0_0_b8
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b9^b8
MOV r0 r5        // Left shift r5 once to get: b7_b6_b5_b4_b3_b2_b1_0
LSL r0 r0
LSR r0 r0        // Right shift r0 7 times to get: 0_0_0_0_0_0_0_b7
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b9^b8^b7
MOV r0 r5        // Left shift r5 twice to get: b6_b5_b4_b3_b2_b1_0_0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift r6 7 times to get: 0_0_0_0_0_0_0_b6
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b9^b8^b7^b6
```

```
MOV r0 r5        // Left shift r5 3 times to get: b5_b4_b3_b2_b1_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift r6 7 times to get: 0_0_0_0_0_0_0_b5
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r0 r2 r0     // XOR r2 and r0, r0 now holds: b11^b10^b9^b8^b7^b6^b5 which is p8
MOV r7 r0        // Store p8 to r7
OR r0 r3 r0      // OR r0 with r3 to get: b11_b10_b9_b8_b7_b6_b5_p8
SB r0 2[r1]      // Store the first half output to memory
MOV r0 r7        // Store r7 to r0 to do 'XOR' operation
XOR r7 r7 r0     // XOR r0, r7 now holds: ^(b11:b5)^p8
MOV r0 r4        // Calculating p4
LSR r0 r0        // Right shift to get b11, r6 now holds: 0_0_0_0_0_0_0_b11
LSR r0 r0
MOV r2 r0        // Store r0 to r2 for later parity bit p4
MOV r0 r4        // Right shift to get b10, r6 now holds: 0_0_0_0_0_0_b11_b10
LSR r0 r0
LSL r0 r0        // Left shift to get b10, r6 now holds: b10_0_0_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
```

```
LSR r0 r0       // Rightt shift r6 now holds: 0_0_0_0_0_0_0_b10
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0    // XOR r2 and r0, r2 now holds: b11^b10
MOV r0 r4       // Left shift r4 7 times to get: b9_0_0_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0    // XOR r2 and r0, r2 now holds: b11^b10^b9
MOV r0 r5       // Right shift r5 7 times to get: 0_0_0_0_0_0_0_b8
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
```

```
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b9^b8
MOV r0 r5        // Left shift r5 4 times to get: b4_b3_b2_b1_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift 7 times to get: 0_0_0_0_0_0_0_b4
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b9^b8^b4
XOR r7 r7 r0     // XOR r7 and r0, r7 now holds: ^(b11:b4)^p8
MOV r0 r5        // Left shift 5 times to get: b3_b2_b1_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift 7 times to get: 0_0_0_0_0_0_0_b3
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
```

```
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b9^b8^b4^b3
XOR r7 r7 r0     // XOR r7 and r0, r7 now holds: ^(b11:b3)^p8
MOV r0 r5        // Left shift 6 times to get: b2_b1_0_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift 7 times to get: 0_0_0_0_0_0_0_b2
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r7 r7 r0     // XOR r7 and r0, r7 now holds: ^(b11:b2)^p8
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b9^b8^b4^b3^b2 which is p4
MOV r0 r2        // XOR r7 and r0, r7 now holds: ^(b11:b2)^p8^p4
XOR r7 r7 r0
MOV r0 r5        // Right shift r5 once to get: 0_b8_b7_b6_b5_b4_b3_b2
LSR r0 r0
LSL r0 r0        // Left shift r6 5 times to get: b4_b3_b2_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
MOV r3 r0        // StORe r0 to r3, r3 now holds: b4_b3_b2_0_0_0_0_0
MOV r0 r2        // Left shift r2 4 times to get: 0_0_0_p4_0_0_0_0
LSL r0 r0
```

```
LSL r0 r0
LSL r0 r0
LSL r0 r0
OR r3 r3 r0      // OR r0 with r3 to get: b4_b3_b2_p4_0_0_0_0
MOV r0 r4        // Calculating p2
LSR r0 r0        // Right shift to get b11, r6 now holds: 0_0_0_0_0_0_b11_b10
LSR r0 r0        // Right shift to get b11, r6 now holds: 0_0_0_0_0_0_0_b11
MOV r2 r0        // Store r0 to r2
MOV r0 r4        // Right shift to get b10, r6 now holds: b10_0_0_0_0_0_0_0
LSR r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift 7 times
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10
MOV r0 r5        // Left shift r5 once to get: b7_b6_b5_b4_b3_b2_b1_0
LSL r0 r0
LSR r0 r0        // Right shift r6 7 times to get: 0_0_0_0_0_0_0_b7
LSR r0 r0
LSR r0 r0
```

```
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b7
MOV r0 r5        // Left shift r5 twice to get: b6_b5_b4_b3_b2_b1_0_0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift r6 7 times to get: 0_0_0_0_0_0_0_b6
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b7^b6
MOV r0 r5        // Left shift 4 times to get: b4_b3_b2_b1_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift 7 times to get: 0_0_0_0_0_0_0_b4
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b7^b6^b4
MOV r0 r5        // Left shift 5 times to get: b3_b2_b1_0_0_0_0_0
```

```
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift 7 times to get: 0_0_0_0_0_0_0_b3
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b7^b6^b4^b3
MOV r0 r5        // Left shift 7 times to get: b1_0_0_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift 4 times to get: 0_0_0_0_b1_0_0_0
LSR r0 r0
LSR r0 r0
LSR r0 r0
OR r3 r3 r0      // OR r0 with r3 to get: b4_b3_b2_p4_b1_0_0_0
LSR r0 r0        // Continue left shifting r6 to get: 0_0_0_0_0_0_0_b1
LSR r0 r0
LSR r0 r0
XOR r7 r7 r0     // XOR r7 and r0, r7 now holds: ^(b11:b1)^p8^p4
```

```
XOR r0 r2 r0     // XOR r2 and r0, r2 now holds: b11^b10^b7^b6^b4^b3^b1 which is p2
XOR r7 r7 r0     // XOR r7 and r0, r7 now holds: ^(b11:b1)^p8^p4^p2
LSL r0 r0        // Left shift r2 2 times to get: 0_0_0_0_0_p2_0_0
LSL r0 r0
OR r3 r3 r0      // OR r0 with r3 to get: b4_b3_b2_p4_b1_p2_0_0
MOV r0 r4        // Calculating p1
LSR r0 r0        // Right shift to get b11, r6 now holds: 0_0_0_0_0_0_b11_b10
LSR r0 r0        // Right shift to get b11, r6 now holds: 0_0_0_0_0_0_0_b11
MOV r2 r0        // Store r0 to r2
MOV r0 r4        // Left shift r4 7 times to get: b9_0_0_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift 7 time to get: 0_0_0_0_0_0_0_b9
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b9
MOV r0 r5        // Left shift r5 once to get: b7_b6_b5_b4_b3_b2_b1_0
LSL r0 r0
LSR r0 r0        // Right shift r6 7 times to get: 0_0_0_0_0_0_0_b7
LSR r0 r0
LSR r0 r0
```

```
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b9^b7
MOV r0 r5        // Left shift r5 3 times to get: b5_b4_b3_b2_b1_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift r6 7 times to get: 0_0_0_0_0_0_0_b5
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b9^b7^b5
MOV r0 r5        // Left shift 4 times to get: b4_b3_b2_b1_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0        // Right shift 7 times to get: 0_0_0_0_0_0_0_b4
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0     // XOR r2 and r0, r2 now holds: b11^b9^b7^b5^b4
```

```
MOV r0 r5       // Left shift 6 times to get: b2_b1_0_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0       // Right shift 7 times to get: 0_0_0_0_0_0_0_b2
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
XOR r2 r2 r0    // XOR r2 and r0, r2 now holds: b11^b9^b7^b5^b4^b2
MOV r0 r5       // Left shift 7 times to get: b1_0_0_0_0_0_0_0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSL r0 r0
LSR r0 r0       // Right shift 7 times to get: 0_0_0_0_0_0_0_b1
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
LSR r0 r0
```

```
   XOR r0 r2 r0     // XOR r2 and r0, r2 now holds: b11^b9^b7^b5^b4^b2^b1 which is p1
   XOR r7 r7 r0     // XOR r7 and r0, r7 now holds: ^(b11:b1)^p8^p4^p2^p1 which is p0
   LSL r0 r0        // Left shift r0 once to get: 0_0_0_0_0_0_p1_0
   OR r3 r3 r0      // OR r0 with r3 to get: b4_b3_b2_p4_b1_p2_p1_0
   MOV r0 r7        // Store r7 to r0 to do 'OR' operation
   OR r0 r3 r0      // OR r0 with r3 to get: b4_b3_b2_p4_b1_p2_p1_p0
   SB r0 1[r1]      // Store the second half of output to memory
   IMM r0 1         // Load immediate 1 for increment
   ADD r1 r1 r0     // Increment memory index for next iteration
   ADD r1 r1 r0
   IMM r6 59        // Load immediate 59
   MOV r0 r1        // Check if done iteration, if index of data_mem for output is not equal 59, then continue looping
   BNE r0 r6 loop
HALT
```