CSE 141L Milestone 1

Brian Mendez A17211975; Phiroze Duggal A17215613; Jinshi He, A17005195

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.

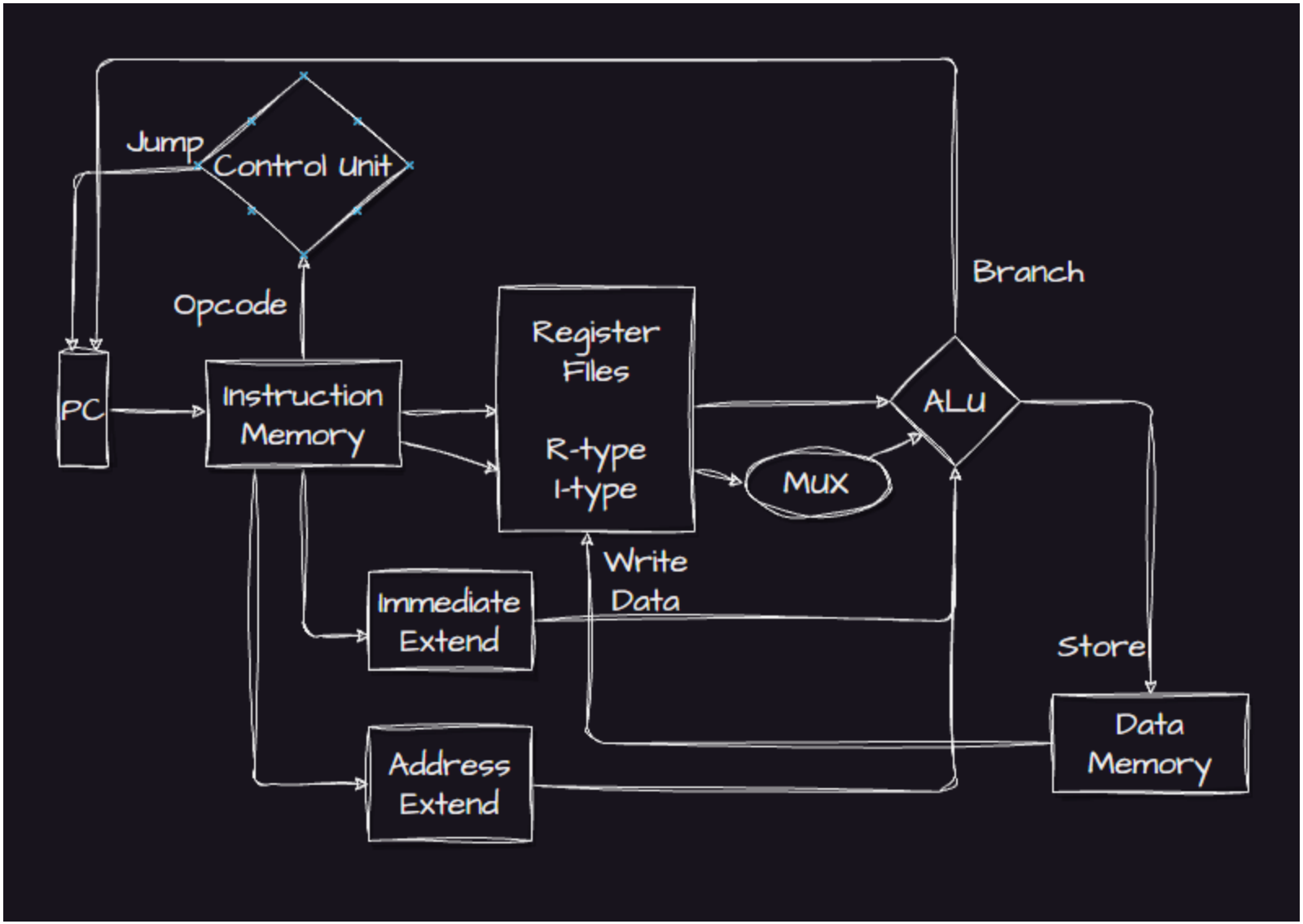
Brian Mendez
Jinshi He
Phiroze Duggal

1. Introduction

Name: FEGprocessor
Philosophy: The overall goal for the ISA is to be able to recover information sent to it using the additional bits/information so that it is able to accurately restore that information in case of transmission corruption.
Specific Goals: Some of our design goals include ensuring the ISA machine contains the necessary mathematical operations for the correct logical handling required for FEC. The instructions for the design should allow us to rely solely on the use of registers in order to handle the data rather than on any memory accesses as well. Processing of information and conducting of operations should be able to be done simultaneously, and our machine ought to contain several branches relating to the various cases of handling the data to perform FEC.
Machine Type: register-register/load-store

2. Architectural Overview

PC:
Data Memory:
Instruction Memory:
Register File: 16 registers maybe 4 bits each
ALU: (controls arithmetic and logical computation)
Control Unit: (determine what each instruction does and controls the flow of data between components
MUX:


Jump
Control Unit
Opcode
Branch
PC
Instruction Memory
Register Files
R-type
I-type
ALU
MUX
Write Data
Immediate Extend
Store
Address Extend
Data Memory

3. Machine Specification

Instruction formats

Two example rows have been filled for you. When you submit, do not include the example types. Add rows as necessary. In your submission, please delete this paragraph.

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
R	AND and XOR: Opcode (3 bits): Defines the specific operation. Source Register 1 (3 bits) Source Register 2 (3 bits) SHIFT: Opcode (3 bits): Defines the specific operation. Source Register (3 bits): Specifies one of 8 registers. Left or Right: (1 bit) if 0 shift left 1 If 1 shift right 1 Immediate (2 bits): amount 4	AND - Bitwise AND XOR - Bitwise XOR * SHIFT - shift to right or left by a distance of the immediate bits
I	LDR and STR: Opcode (3 bits): Defines the specific operation. Address Register (3 bits): Specifies one of 8 registers. Data Register (3 bits): Specifies the register that stores the data mem MOV/ADD:	Loading and Storing: LDR - Load from memory * STR - Store to memory * Loop Counters: MOV - Move an immediate value to a register * ADD - add an immediate value to a register

	Opcode (3 bits): Defines the specific operation. Source Register (3 bits): Waste/offset: (1 bit): 0 then mov if 1 then add => MOV R0 #1 immediate Immediate (2 bits):	
J	CMP: Opcode: 3 bits Source Register 1 (3 bits) Source Register 2 (3 bits) BEQ: Opcode: 3 bits Target address: 6 bits	CMP - compare two registers BNE - Branch if equal

Operations

An example row has been filled for you. When you submit, do not include the example type. In the name column, be sure to also add the definition of what the example actually does. For example, "lsl = logical shift left" would be an appropriate value to put in the name column. In the bit breakdown column, add in parenthesis what specific values the bits should be in order. X indicates that it will be specified by the programmer's instruction itself (i.e. specifying registers). In the example column, give an example of an “assembly language” instruction in your machine, then translate it into machine code. Add rows as necessary. In your submission, please delete this paragraph.

NAME	**TYPE**	**BIT BREAKDOWN**	**EXAMPLE**	**NOTES**

AND= logical and	R	3 bits opcode (000), 3 bit source register 1 (XXX), 3 bit source register 2 (XXX) 100_XXX_XXX	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` `and R0, R1` ⇔ `100_000_001`	Performs a bitwise AND between R0 and R1 and stores the result in R2.
XOR = logical XOR	R	3 bits opcode (000), 3 bit source register 1 (XXX), 3 bit source register 2 (XXX) 011_XXX_XXX	#Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` XOR R2, R0, R1 Machine Code: 011_000_001	Performs a bitwise XOR between R0 and R1 and stores the result in R2.
SHIFT = shift left or right logical	R	3 bits opcode (000), 3 bit source register 1 (XXX), 1 bit direction for left or right, 2 bit shift value (XX) 011_XX_X_XX	#Assume R0 contains the binary value: 0b0001_0101 #Assume R1 contains the value: 2. SHIFT R1 L 4 Machine Code: 011_00_01_0	Left shifts value in R1 by 4 and stores the result in R1. This means we'll be shifting the value in R0 to the left by 2 positions.
MOV = immediate move	I	3 bits opcode (000), 3 bit source register 1 (XXX), 1 bit control for mov or add, 2 bit immediate value (XX) 010_XXX_X_XX	#Assume R1 contains the binary value: 0b1001_0010 MOV R1 1 Machine Code: 100_001_0_01	MOV immediate value 1 to R1
ADD = immediate bitwise add	I	3 bits opcode (000), 3 bit source register 1 (XXX), 1 bit control for mov or add, 2 bit	#Assume R0 contains the binary value: 0b1001_0010 #Assume R1 contains the value: 3.	ADD immediate value 1 to R1 which is R1 =+ 1

		immediate value 010_XXX_X_XX	ADD R1 1 Machine Code: 100_001_1_01	
LDR = load from memory	I	3 bits opcode (000), 3 bit source register (XXX), 3 bit address register (XXX) 000_XX_XXX	#Assume R0 is the destination register LDR R0, [R1] Machine Code: 000_000_001	Loads value from memory address R1 into R0.
STR = store to memory	I	3 bits opcode (000), 3 bit source register (XXX), 3 bit address register (XXX) 001_XXX_XXX	#Assume R0 and R1 have values in their addresses STR R0, [R1] Machine Code: 000_000_001	Stores value from memory address 6 into R0
CMP- compare registers	J	Opcode: 3 bits Source Register 1 (3 bits) Source Register 2 (3 bits) 110_XXX_XXX	CMP R0, R1 Machine Code: 1_101_00_01_00101	No direct output in registers, as it's a jump instruction. The program counter would move to address #5 if R0 and R1 are not equal.
BNE - Branch if equal	J	Opcode: 3 bits Target address: 6 bits 111_XXXXX	BNE R0 label Machine Code: 110_00_01_00111	No direct output in registers, as it's a jump instruction. The program counter would move to address #7 if R0 and R1 are equal.

Internal Operands

8 registers. Considering the instruction formats, having a 4-bit field for registers would allow you to address up to 16 registers.

Control Flow (branches)

Conditional and unconditional branches are supported. The target addresses are usually specified using labels, and the assembler calculates the relative offset from the current instruction. The maximum branch distance is 5 bits. If a branch target is outside the maximum range, one possible workaround is to use a combination of loading the target address into a register and then using an indirect branch.

Lookup Table

Addressing Modes

Direct address is supported, and the address is calculated by using a base register and an offset.
Ex: LDR X0, [X1, #8]: Load from the address in X1 plus 8

4. Programmer's Model [Lite]

TODO. 4.1 How should a programmer think about how your machine operates? Provide a description of the general strategy a programmer should use to write programs with your machine. For example, one could say that the programmer should prioritize loading in the necessary values from memory into as many registers as possible, then perform calculations. Another approach could be loading and writing to memory in between every calculation step. Word limit: 200 words.
In general, we will be prioritizing loading in the necessary values from memory into as many registers as possible. This method is done in order to avoid going back and forth between retrieving and sending data back into the memory while the machine is going about its calculations. In other words, the data would be provided in a more handy way. Then, depending on the operation type, the data would be fed into the individual parts of the machine according to its necessary amount of information. This would be gathered from the registers themselves rather than a backup memory source.

TODO. 4.2 Can we copy the instructions/operation from MIPS or ARM ISA? If no, explain why not? How did you overcome this or how do you deal with this in your current design? Word limit: 100 words.
No, instructions cannot be directly copied from MIPS or ARM ISA, as each architecture has its unique instruction set, encoding, and operational semantics. While there may be similarities in concepts and some instructions, the binary encoding and specific behavior

can differ. To overcome this, we must understand the specific characteristics of our architecture and translate or rewrite the code accordingly. Utilizing the appropriate instructions, addressing modes, and control structures ensures that the code aligns with the architecture's design and operates correctly.

On the other hand, instructions can be copied from MIPS and ARM with modifications to satisfy our ISA for programs 1 between 3.

5. Program Implementation

Program 1 Pseudocode

```
forward_correction_coder(data_mem):
     for i from 0 to 14:
          MSW_input = data_mem[2*i + 1]
          LSW_input = data_mem[2*i]

          b11_b5 = MSW_input << 5 | (LSW_input & 11110000) >> 3
          b4_b1 = (LSW_input & 00001111) << 4

          p8 = (b11_b5 >> 1) ^ (b11_b5 >> 2) ^ (b11_b5 >> 3) ^ (b11_b5 >> 4) ^ (b11_b5 >> 5) ^
     (b11_b5 >> 6) ^ (b11_b5 >> 7)
          p8 = (p8 & 00000001) << 7

          p4 = b11_b5 ^ (b11_b5 << 1) ^ (b11_b5 << 2) ^ (b11_b5 << 3) ^ b4_b1 ^ (b4_b1 << 1) ^
     (b4_b1 << 2)
          p4 = p4 & 10000000

          p2 = b11_b5 ^ (b11_b5 << 1) ^ (b11_b5 << 4) ^ (b11_b5 << 5) ^ b4_b1 ^ (b4_b1 << 1) ^
     (b4_b1 << 3)
          p2 = p2 & 10000000
```

```
     p1 = b11_b5 ^ (b11_b5 << 2) ^ (b11_b5 << 4) ^ (b11_b5 << 6) ^ b4_b1 ^ (b4_b1 << 2) ^
(b4_b1 << 3)
     p1 = p1 & 10000000

     p0 = p8 ^ b4_b1 ^ (b4_b1 << 1) ^ (b4_b1 << 2) ^ (b4_b1 << 3) ^ p8 ^ p4 ^ p2 ^ p1
     p0 = p0 & 10000000

     MSW_output = b11_b5 | p8
     LSW_output = (b4_b1 ^ 11100000) | (p4 >> 3) | ((b4_b1 ^ 00010000) >> 1) | (p2 >> 5) |
(p1 >> 6) | (p0 >> 7)

     data_mem[2*i + 31] = MSW_output
     data_mem[2*i + 30] = LSW_output
end for
```

Program 1 Assembly Code

```
MOV r0 0
MOV r1 4
SHIFT r1 L 4
STR r0 r1

LDR r1 r0
ADD r0 1
LDR r2 r0

XOR r3 r1
XOR r4 r2

// p8
```

```
SHIFT r3 R 4
XOR r3 r4
MOV r4 0
XOR r4 r3
SHIFT r4 R 2
XOR r3 r4
MOV r4 0
XOR r4 r3
SHIFT r4 R 1
XOR r3 r4

LDR r3 p8  // store p8



// p2
MOV r3 0
MOV r4 0
XOR r3 r1
XOR r4 r2
MOV r5 0

SHIFT r4 R 1
XOR r3 r4
XOR r5 r3

MOV r3 0
MOV r4 0
XOR r3 r1
XOR r4 r2

XOR r3 r4
```

```
SHIFT r3 R 2
XOR r5 r3

MOV r4 0
XOR r4 r2

SHIFT r4 R 3
XOR r5 r4

MOV r4 0
XOR r4 r2

SHIFT r4 R 4
SHIFT r4 R 1
XOR r5 r4

MOV r4 0
XOR r4 r2

SHIFT r4 R 4
SHIFT r4 R 2
XOR r5 r4

LDR r5 p2

// p1
```

Program 2 Pseudocode

```
forward_correction_decoder(data_mem):
      for i from 0 to 14:
```

```
        MSW_input = data_mem[2*i + 31]
        LSW_input = data_mem[2*i + 30]

        b11_b5 = MSW_input & 11111110
        b4_b1 = (LSW_input & 11100000) | ((LSW_input & 00001000) << 1)

        p8_received = MSW_input << 7
        p4_received = (LSW_input & 00010000) << 3
        p2_received = (LSW_input & 00000100) << 5
        p1_received = (LSW_input & 00000010) << 6
        p0_received = (LSW_input & 00000001) << 7

        p8_cal = (b11_b5 >> 1) ^ (b11_b5 >> 2) ^ (b11_b5 >> 3) ^ (b11_b5 >> 4) ^ (b11_b5 >>
    5) ^ (b11_b5 >> 6) ^ (b11_b5 >> 7)
        p8_cal = (p8_cal & 00000001) << 7

        p4_cal = b11_b5 ^ (b11_b5 << 1) ^ (b11_b5 << 2) ^ (b11_b5 << 3) ^ b4_b1 ^ (b4_b1 <<
    1) ^ (b4_b1 << 2)
        p4_cal = p4_cal & 10000000

        p2_cal = b11_b5 ^ (b11_b5 << 1) ^ (b11_b5 << 4) ^ (b11_b5 << 5) ^ b4_b1 ^ (b4_b1 <<
    1) ^ (b4_b1 << 3)
        p2_cal = p2_cal & 10000000


        p1_cal = b11_b5 ^ (b11_b5 << 2) ^ (b11_b5 << 4) ^ (b11_b5 << 6) ^ b4_b1 ^ (b4_b1 <<
2) ^
    (b4_b1 << 3)
        p1_cal = p1_cal & 10000000

        p0_cal = p8_cal ^ b4_b1 ^ (b4_b1 << 1) ^ (b4_b1 << 2) ^ (b4_b1 << 3) ^ p8_cal ^
    p4_cal ^ p2_cal ^ p1_cal
```

```
p0_cal = p0_cal & 10000000


error_check = 0;
if (p8_cal != p8_received) {
     error_check += 1;
}
if (p4_cal != p4_received) {
     error_check += 1;
}
if (p2_cal != p2_received) {
     error_check += 1;
}
if (p1_cal != p1_received) {
     error_check += 1;
}
if (p0_cal != p0_received) {
     error_check += 1;
}

F1F0 = 0
if (error_check > 0) {
     F1F0 = 1

     if (p0_cal != p0_received) {
          F1F0 = 3
          break
     }

     if (p1_cal != p1_received and p2_cal != p2_received and p4_cal == p4_received
and p8_cal == p8_received) {
          d1 = d1 ^ 1
```

```
      }
      if (p1_cal != p1_received and p2_cal == p2_received and p4_cal != p4_received
and p8_cal == p8_received) {
            d2 = d2 ^ 1
      }
      if (p1_cal == p1_received and p2_cal != p2_received and p4_cal != p4_received
and p8_cal == p8_received) {
            d3 = d3 ^ 1
      }
      if (p1_cal != p1_received and p2_cal != p2_received and p4_cal != p4_received
and p8_cal == p8_received) {
            d4 = d4 ^ 1
      }
      if (p1_cal != p1_received and p2_cal == p2_received and p4_cal == p4_received
and p8_cal != p8_received) {
            d5 = d5 ^ 1
      }
      if (p1_cal == p1_received and p2_cal != p2_received and p4_cal == p4_received
and p8_cal != p8_received) {
            d6 = d6 ^ 1
      }
      if (p1_cal != p1_received and p2_cal != p2_received and p4_cal == p4_received
and p8_cal != p8_received) {
            d7 = d7 ^ 1
      }
      if (p1_cal == p1_received and p2_cal == p2_received and p4_cal != p4_received
and p8_cal != p8_received) {
            d8 = d8 ^ 1
      }
      if (p1_cal != p1_received and p2_cal == p2_received and p4_cal != p4_received
and p8_cal != p8_received) {
            d9 = d9 ^ 1
```

```
            }
            if (p1_cal == p1_received and p2_cal != p2_received and p4_cal != p4_received
      and p8_cal != p8_received) {
                  d10 = d10 ^ 1
            }
            if (p1_cal != p1_received and p2_cal != p2_received and p4_cal != p4_received
      and p8_cal != p8_received) {
                  d11 = d11 ^ 1
            }
      }

      MSW_output = (F1F0 << 6) | (b11_b5 >> 5)
      LSW_output = (b11_b5 << 3) | (b4_b1 >> 4)

      data_mem[2*i + 1] = MSW_output
      data_mem[2*i] = LSW_output
end for
```

Program 2 Assembly Code

```
MOV r0 0          // initialize loop counter i

SHIFT r0 L 1
ADD r0 3
ADD r0 3
ADD r0 3
ADD r0 3
ADD r0 3
ADD r0 3
ADD r0 3
ADD r0 3
ADD r0 3
```

```
ADD r0 3                // r0 = 2*i + 30

LDR r1 r0               // r1 = data_mem[2*i + 30] LSW
ADD r0 1
LDR r2 r0               // r2 = data_mem[2*i + 31] MSB

MOV r3 0                // Store all the parity bits as p8_p4_p2_p1_p0_000

//p0
MOV r4 00000001
AND r4 r1
XOR r3 r4               // 0000000p0
SHIFT r3 L 3            // 0000p0_000

//p1
MOV r4 00000010             // Mask all bits in r1 except the second bit (p1)
AND r4 r1
SHIFT r4 L 3            // 000p10_000
XOR r3 r4               // 000p1_p0_000 stored in r3

//p2
MOV r4 00000100
AND r4 r1               // Mask all bits in r1 except the second bit (p2)
SHIFT r1 R 1            // Shift to the right to align with p2 in r3
XOR r3 r1               // Store p2 in r3 00_p2_p1_p0_000

//p4
MOV r4 00010000             // Mask all bits in r1 except the fifth bit (p4)
AND r4 r1
SHIFT r1 L 3            // Shift to the right to align with p4 in r3
XOR r3 r1               // Store p4 in r3 0p4_p2_p1_p0_000
```

```
//p8
MOV r4 00000001          // Mask all bits in r2 except the first bit (p8)
AND r4 r2
SHIFT r2 R 1         // Shift to the right to align with p8 in r3
XOR r3 r4            // Store p8 in r3 p8_p4_p2_p1_p0_000

MOV r4 0
MOV r5 0

// Extract d1-d4 from r1
MOV r6 11100000          // Mask d4 d3 d2 0 0 0 0 0
AND r6 r1
XOR r4 r6            // Store d2-d4 in r4
MOV r6 00001000          // get d1 and add to r4
AND r6 r1
SHIFT r6 L 1
XOR r4 r6            // store d1:d4 in r4 d4_d3_d2_d1_0000

// Extract d5-d11 from r2
MOV r6 11111110          // mask p8 to have d5:d11
AND r6 r2
XOR r5 r6            // Store d5:d11 in r5 d11_d10_d9_d8_d7_d6_d5_0

MOV r6 0             // Calculate parity bit and store in r6  to get p8_p4_p2_p1_p0_000 calc

//p8_cal
MOV r7 10000000          // d11
AND r7 r5
XOR r6 r7
MOV r7 01000000          // d10
AND r7 r5
SHIFT r7 L 1
```

```
XOR r6 r7
MOV r7 00100000          // d9
AND r7 r5
SHIFT r7 L 2
XOR r6 r7
MOV r7 00010000          // d8
AND r7 r5
SHIFT r7 L 3
XOR r6 r7
MOV r7 00001000          // d7
AND r7 r5
SHIFT r7 L 3
SHIFT r7 L 1
XOR r6 r7
MOV r7 00000100          // d6
AND r7 r5
SHIFT r7 L 3
SHIFT r7 L 2
XOR r6 r7
MOV r7 00000010          // d5
AND r7 r5
SHIFT r7 L 3
SHIFT r7 L 3
XOR r6 r7          // p8cal_0000000

//p4_cal
MOV r7 10000000          // d11
AND r7 r5
SHIFT R7 R 1
XOR r6 r7
MOV r7 01000000          // d10
AND r7 r5
```

```
XOR r6 r7
MOV r7 00100000          // d9
AND r7 r5
SHIFT r7 L 1
XOR r6 r7
MOV r7 00010000          // d8
AND r7 r5
SHIFT r7 L 2
XOR r6 r7
MOV r7 10000000          // d4
AND r7 r4
SHIFT r7 R 1
XOR r6 r7
MOV r7 01000000          // d3
AND r7 r4
XOR r6 r7
MOV r7 00100000          // d2
AND r7 r4
SHIFT r7 L 1
XOR r6 r7          // p8cal_p4cal_000000

//p2_cal
MOV r7 10000000          // d11
AND r7 r5
SHIFT r7 R 2
XOR r6 r7
MOV r7 01000000          // d10
AND r7 r5
SHIFT r7 R 1
XOR r6 r7
MOV r7 00001000          // d7
AND r7 r5
```

```
SHIFT r7 L 2
XOR r6 r7
MOV r7 00000100             // d6
AND r7 r5
SHIFT r7 L 3
XOR r6 r7
MOV r7 10000000             // d4
AND r7 r4
SHIFT r7 R 2
XOR r6 r7
MOV r7 01000000             // d3
AND r7 r4
SHIFT r7 R 1
XOR r6 r7
MOV r7 00010000             // d1
AND r7 r4
SHIFT r7 L 1
XOR r6 r7               // p8cal_p4cal_p2cal_00000

//p1_cal
MOV r7 10000000             // d11
AND r7 r5
SHIFT r7 R 3
XOR r6 r7
MOV r7 00100000             // d9
AND r7 r5
SHIFT r7 R 1
XOR r6 r7
MOV r7 00001000             // d7
AND r7 r5
SHIFT r7 L 1
XOR r6 r7
```

```
MOV r7 00000010          // d5
AND r7 r5
SHIFT r7 L 3
XOR r6 r7
MOV r7 10000000          // d4
AND r7 r4
SHIFT r7 R 3
XOR r6 r7
MOV r7 00100000          // d2
AND r7 r4
SHIFT r7 R 1
XOR r6 r7
MOV r7 00010000          // d1
AND r7 r4
XOR r6 r7          // p8cal_p4cal_p2cal_p1cal_0000

//p0_cal           // XOR d1:d11 p8 p4 p2 p1
MOV r7 10000000          // d11
AND r7 r5
SHIFT r7 R 3
SHIFT r7 R 1
XOR r6 r7
MOV r7 01000000          // d10
AND r7 r5
SHIFT r7 R 3
XOR r6 r7
MOV r7 00100000          // d9
AND r7 r5
SHIFT r7 R 2
XOR r6 r7
MOV r7 00010000          // d8
AND r7 r5
```

```
SHIFT r7 R 1
XOR r6 r7
MOV r7 00001000          // d7
AND r7 r5
XOR r6 r7
MOV r7 00000100          // d6
AND r7 r5
SHIFT r7 L 1
XOR r6 r7
MOV r7 00000010          // d5
AND r7 r5
SHIFT r7 L 2
XOR r6 r7
MOV r7 10000000          // d4
AND r7 r4
SHIFT r7 R 3
SHIFT r7 R 1
XOR r6 r7
MOV r7 01000000          // d3
AND r7 r4
SHIFT r7 R 3
XOR r6 r7
MOV r7 00100000          // d2
AND r7 r4
SHIFT r7 R 2
XOR r6 r7
MOV r7 00010000          // d1
AND r7 r4
SHIFT r7 R 1
XOR r6 r7
MOV r7 10000000          // p8
AND r7 r3
```

```
SHIFT r7 R 3
SHIFT r7 R 1
XOR r6 r7
MOV r7 01000000                 // p4
AND r7 r3
SHIFT r7 R 3
XOR r6 r7
MOV r7 00100000                 // p2
AND r7 r3
SHIFT r7 R 2
XOR r6 r7
MOV r7 00010000                 // p1
AND r7 r3
SHIFT r7 R 1
XOR r6 r7                   // store calc parity p8cal_p4cal_p2cal_p1cal_p0cal_000

//free registers are now  r1,r2,r7
//Use R1 & R2 to store p_rec & p_cal then compare
//CMP p0
MOV r1 00001000
AND r1 r3
MOV r2 00001000
AND r2 r6
CMP r1 r2
// if not equal jump to label SEC
// if equal proceed to the next line

//CMP p8
MOV r1 10000000
AND r1 r3
MOV r2 10000000
AND r2 r6
```

```
CMP r1 r2
// if not equal jump to label DED

//CMP p4
MOV r1 01000000
AND r1 r3
MOV r2 01000000
AND r2 r6
CMP r1 r2
// if not equal jump to label DED

//CMP p2
MOV r1 00100000
AND r1 r3
MOV r2 00100000
AND r2 r6
CMP r1 r2
// if not equal jump to label DED

//CMP p1
MOV r1 00010000
AND r1 r3
MOV r2 00010000
AND r2 r6
CMP r1 r2
// if not equal jump to label DED
// if equal that means we made it to the end no errors found label NED

SEC(single error correction):
MOV r1 11110000
AND r1 r6
XOR r1 r3
```

```
MOV r2 11110000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
MOV r1 10000000
XOR r5 r1          // flip d11 then jump to end

MOV r1 01110000
AND r1 r6
XOR r1 r3
MOV r2 01110000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
MOV r1 01000000
XOR r5 r1          // flip d10 then jump to end

MOV r1 10110000
AND r1 r6
XOR r1 r3
MOV r2 10110000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
MOV r1 00100000
XOR r5 r1          // flip d9 then jump to end

MOV r1 00110000
AND r1 r6
XOR r1 r3
MOV r2 00110000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
MOV r1 00010000
XOR r5 r1          // flip d8 then jump to end

MOV r1 11010000
AND r1 r6
XOR r1 r3
```

```
MOV r2 11010000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
MOV r1 00001000
XOR r5 r1          // flip d7 then jump to end

MOV r1 01010000
AND r1 r6
XOR r1 r3
MOV r2 01010000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
MOV r1 00000100
XOR r5 r1          // flip d6 then jump to end

MOV r1 10010000
AND r1 r6
XOR r1 r3
MOV r2 10010000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
MOV r1 00000010
XOR r5 r1          // flip d5 then jump to end

MOV r1 10000000
AND r1 r6
XOR r1 r3
MOV r2 10000000
CMP r1 r2          // jump to the end because no change to p8

MOV r1 11100000
AND r1 r6
XOR r1 r3
MOV r2 11100000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
```

```
MOV r1 10000000
XOR r4 r1          // flip d4 then jump to end

MOV r1 01100000
AND r1 r6
XOR r1 r3
MOV r2 01100000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
MOV r1 01000000
XOR r4 r1          // flip d3 then jump to end

MOV r1 10100000
AND r1 r6
XOR r1 r3
MOV r2 10100000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
MOV r1 00100000
XOR r4 r1          // flip d2 then jump to end

MOV r1 00100000
AND r1 r6
XOR r1 r3
MOV r2 00100000
CMP r1 r2          // jump to the end because no change to p4

MOV r1 11000000
AND r1 r6
XOR r1 r3
MOV r2 11000000
CMP r1 r2          // if they equal proceed to next line if not jump 2 lines
MOV r1 00010000
XOR r4 r1          // flip d1 then jump to end
```

```
MOV r1 01000000
AND r1 r6
XOR r1 r3
MOV r2 01000000
CMP r1 r2              // jump to the end because no change to p2

MOV r1 10000000
AND r1 r6
XOR r1 r3
MOV r2 10000000
CMP r1 r2              // jump to the end because no change to p1


MOV r1 01000000
MOV r2 11100000
XOR r2 r5
SHIFT r2 R 3
SHIFT r2 R 2
XOR r1 r2              // final format r1 = F1 F0 0 0 0 D11 D10 D9
MOV r2 00011110
AND r2 r5
SHIFT r2 L 3
MOV r3 11110000
SHIFT r3 R 3
SHIFT r3 R 1
XOR r2 r3              // final format r2 = D8 D7 D6 D5 D4 D3 D2 D1

//register r1-r3 & r6, r7 are free
//r4 has d4_d3_d2_d1_0000
//r5 has d11_d10_d9_d8_d7_d6_d5_0
DED:
```

```
MOV r1 1x000000
MOV r2 11100000
XOR r2 r5
SHIFT r2 R 3
SHIFT r2 R 2
XOR r1 r2          // final format r1 = F1 F0 0 0 0 D11 D10 D9

MOV r2 00011110
AND r2 r5
SHIFT r2 L 3
MOV r3 11110000
SHIFT r3 R 3
SHIFT r3 R 1
XOR r2 r3          // final format r2 = D8 D7 D6 D5 D4 D3 D2 D1

NED:
MOV r1 00000000
MOV r2 11100000
XOR r2 r5
SHIFT r2 R 3
SHIFT r2 R 2
XOR r1 r2          // final format r1 = F1 F0 0 0 0 D11 D10 D9

MOV r2 00011110
AND r2 r5
SHIFT r2 L 3
MOV r3 11110000
SHIFT r3 R 3
SHIFT r3 R 1
XOR r2 r3          // final format r2 = D8 D7 D6 D5 D4 D3 D2 D1

ADD r0 1
```

BEQ

Program 3 Pseudocode

```
pattern_search(data_mem):
      occurrence_in_bytes = 0;
      num_of_bytes = 0;
      num_of_times = 0;

      pattern = (data_mem[32] >> 3) & 11111; #extracting [7:3] bits

      for i from 0 to 31:
            Occurrence_in_single_byte = 0;
            byte_value = data_mem[i];
            for j from 0 to 3:
                  if (byte_value >> j) & 11111 == pattern
                        occurrence_in_bytes += 1;
                        occurrence_in_single_byte += 1;
            num_of_bytes++
            if occurrence_in_single_byte == 0
                  num_of_bytes—-;


      for i from 0 to 251:
            bytes_index = i // 8;
            bit_position = i % 8;
            if bit_position < 3:
                  skip to the next for loop
            else:
                  pattern_to_be_compared = ((data_mem[byte_index] & (11111111 >> bit_position))
<< bit_position) & ((data_mem[byte_index + 1] >> (5 - (8 - bit_position)))
```

```
                pattern_to_be_compared = (pattern_to_be_compared & 11111000) >> 3
                if (pattern_to_be_compared == pattern)
                     num_of_times += 1
     num_of_times += occurrence_in_bytes

     data_mem[33] = occurrence_in_bytes;
     data_mem[34] = num_of_bytes;
     data_mem[35] = num_of_times;

return data_mem;
```

Program 3 Assembly Code

```
; Assuming:
; - R0 is our loop counter for the first loop (i)
; - R1 is our loop counter for the second loop (j)
; - R2 points to the start of data_mem
; - R3 will store our pattern
; - R4 will store the current byte_value
; - R5 will store occurrence_in_bytes
; - R6 will store occurrence_in_single_byte
; - R7 will store num_of_bytes
; - R8 will store num_of_times
; - R9 will store bytes_index
; - R10 will store bit_position
; - R11 will store pattern_to_be_compared
; - R12 is used as temporary storage

; Initialize
MOV R5, #0
```

```
MOV R6, #0
MOV R7, #0
MOV R8, #0

; Extract pattern from data_mem[32]
LDR R3, [R2, #32]
LSR R3, #3
AND R3, #0b11111

; First loop (i from 0 to 31)
MOV R0, #0
loop1_start:
CMP R0, #31
BGT loop1_end

; Load byte_value
LDR R4, [R2, R0]

; Second loop (j from 0 to 3)
MOV R1, #0
MOV R6, #0
loop2_start:
CMP R1, #3
BGT loop2_end

LSR R12, R4, R1
AND R12, #0b11111
CMP R12, R3
BNE no_increment

ADD R5, R5, #1
ADD R6, R6, #1
```

```
no_increment:
ADD R1, R1, #1
B loop2_start

loop2_end:
CMP R6, #0
BEQ no_byte_increment

ADD R7, R7, #1

no_byte_increment:
ADD R0, R0, #1
B loop1_start

loop1_end:

; ... Continue with the rest of the logic for the second main loop

; At the end, store results back to data_mem
STR R5, [R2, #33]
STR R7, [R2, #34]
STR R8, [R2, #35]

; Return (depends on calling convention, here's a simple RET)
RET
```

CSE 141L Milestone 2 Add-on Template

Instructions

We are providing you with the template that should be copied and pasted into your Milestone 1 Report. The updated report, alongside your Verilog code, will be your submission for Milestone 2.

Steps

1. Change the title of your report from "CSE 141L Milestone 1" to "CSE 141L Milestone 2".
2. Please make appropriate changes to your Milestone 1 content based off of feedback from the teaching staff. We will release feedback and changes on Gradescope by Tuesday, May 3, 2022. Make sure to update your Changelog with what you have changed.
3. Add your answer to the question "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). Label this question 4.3.
4. Change the Program Implementation number from 5 to 6.

(more steps on the next page)

5. Copy, paste, and complete the section templates we provide in the following pages to your report. Omit sections that you don't intend on using from your report (e.g. lookup table if you do not have one). We want you to test two modules: the Program Counter (to see if you can perform jumps/branches) and the ALU (to verify you have implementations for your specified instructions). Your report should consist of, *in order*:
    0. Team
    1. Introduction
    2. Architectural Overview
    3. Machine Specification
    4. Programmer's Model
    5. Individual Component Specification
        a. Top Level
        b. Program Counter
        c. Instruction Memory
        d. Control Decoder
        e. Register File
        f. ALU
        g. Data Memory
        h. Lookup Tables
        i. Muxes
        j. (if necessary) other modules - *name sections as appropriate*
    6. Program Implementation
    7. Changelog

Individual Component Specification

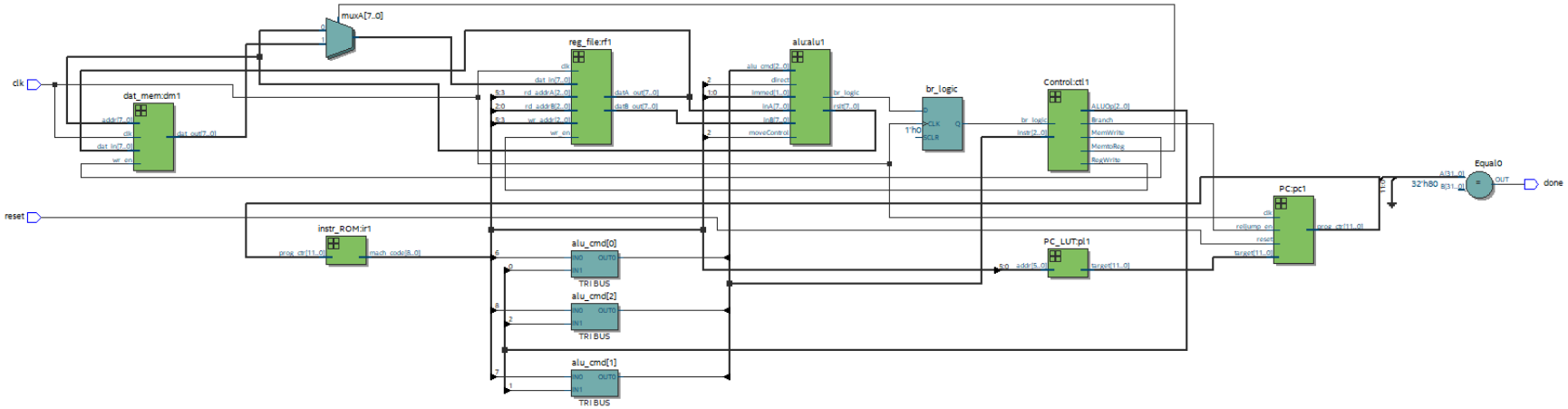
Top Level

Module file name: top_level.sv

Functionality Description

The top-level module serves as the central hub that integrates all other modules within the processor. It is responsible for connecting ports to their respective inputs and ensuring proper communication between different components. It orchestrates the overall operation of the processor, managing the flow of data and control signals between various modules.

Schematic



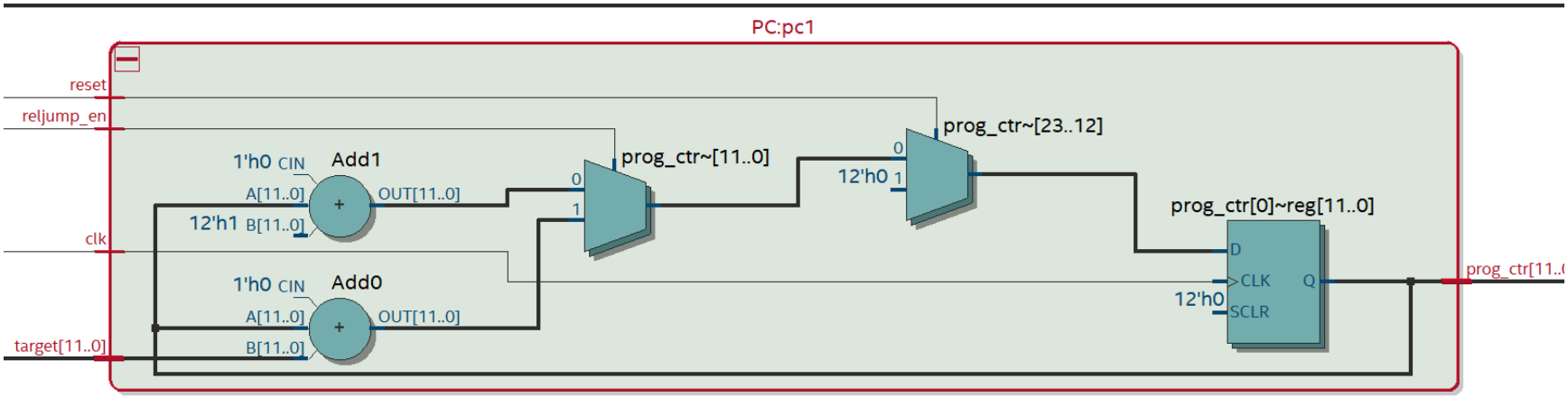
Program Counter

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

Functionality Description

The Program Counter (PC) module is responsible for managing the execution flow of the program. It holds the address of the current instruction being executed and updates to the next instruction or jumps to a specific location based on control signals or lookup tables. This ensures the sequential or conditional execution of instructions.

Schematic



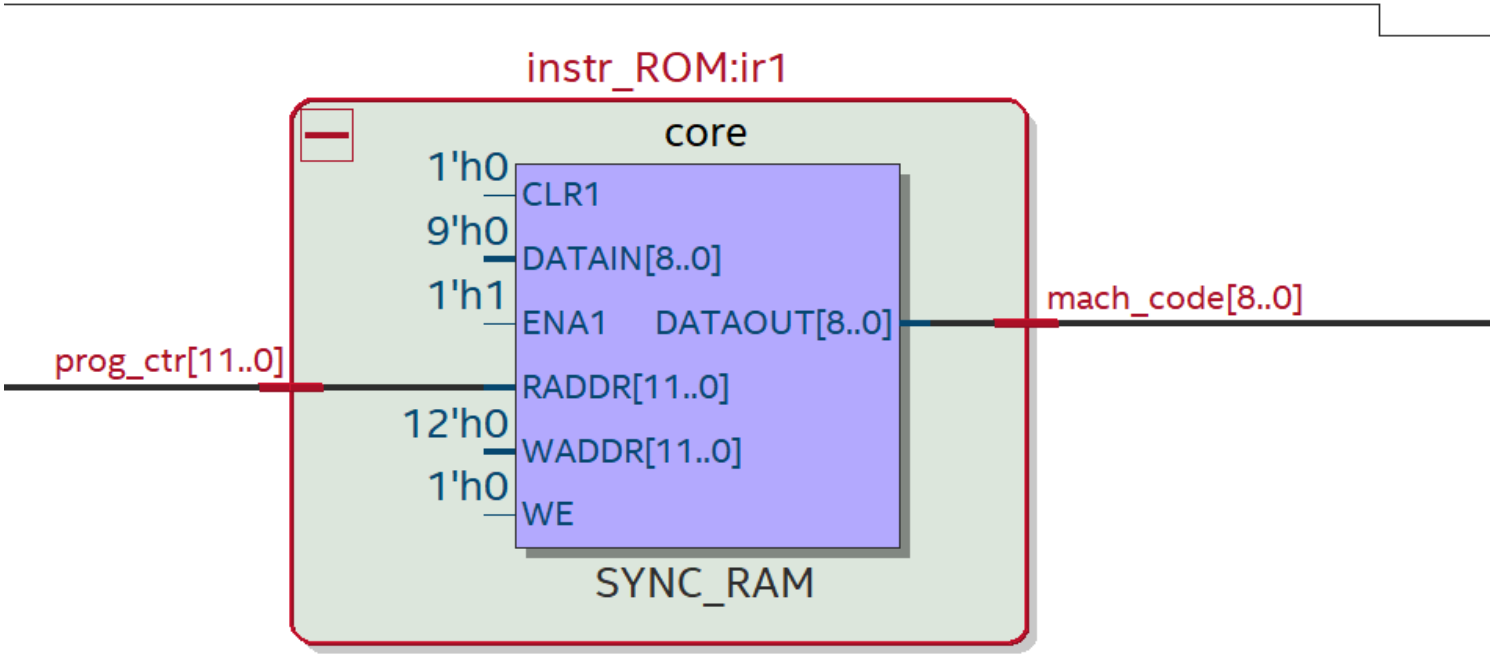
Instruction Memory

Module file name: instr_ROM.sv

Functionality Description

The Instruction Memory module serves as a lookup table that maps the program counter to the corresponding machine code. It stores the instructions to be executed and provides them to the processor based on the current value of the program counter.

Schematic



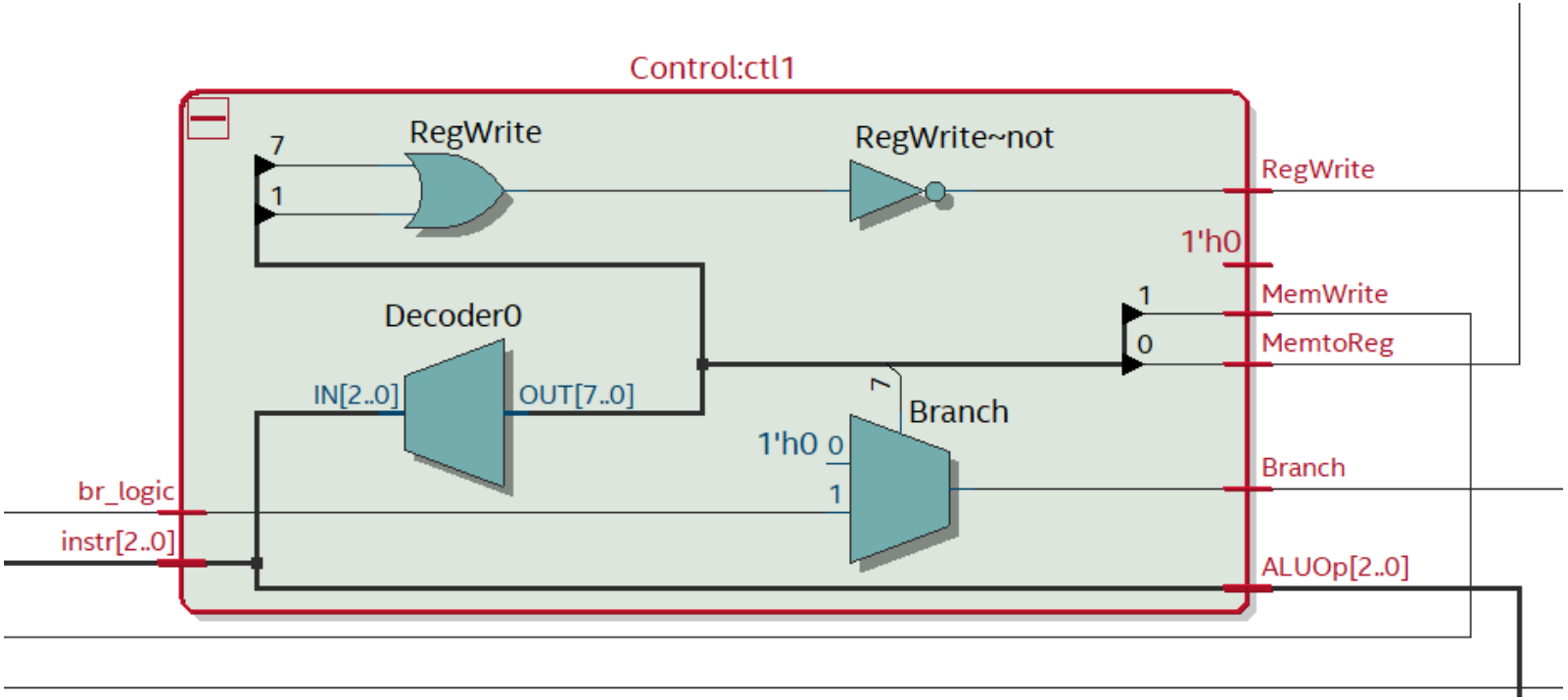
Control Decoder

Module file name: Control.sv

Functionality Description

The Control Decoder module interprets the opcode of the current instruction and generates the necessary control signals to execute it. These signals enable or disable various functions within the processor, guiding the execution of the instruction according to its semantics.

Schematic



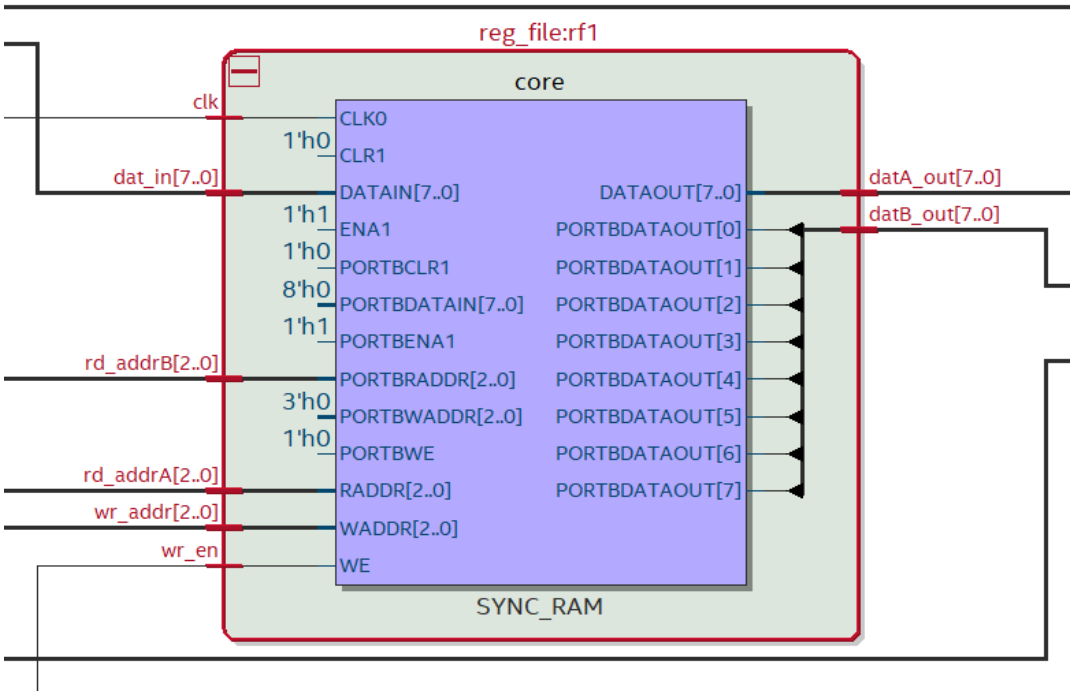
Register File

Module file name: reg_file.sv

Functionality Description

The Register File module handles the reading and writing of processor registers. It provides access to the data stored in the registers, allowing for quick retrieval and modification of values used in computations.

Schematic



ALU (Arithmetic Logic Unit)

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

Functionality Description

The Arithmetic Logic Unit (ALU) is responsible for performing arithmetic and logical operations such as addition, subtraction, AND, XOR, etc. It takes operands from the register file or immediate values and performs the specified operation, returning the result to the appropriate destination.

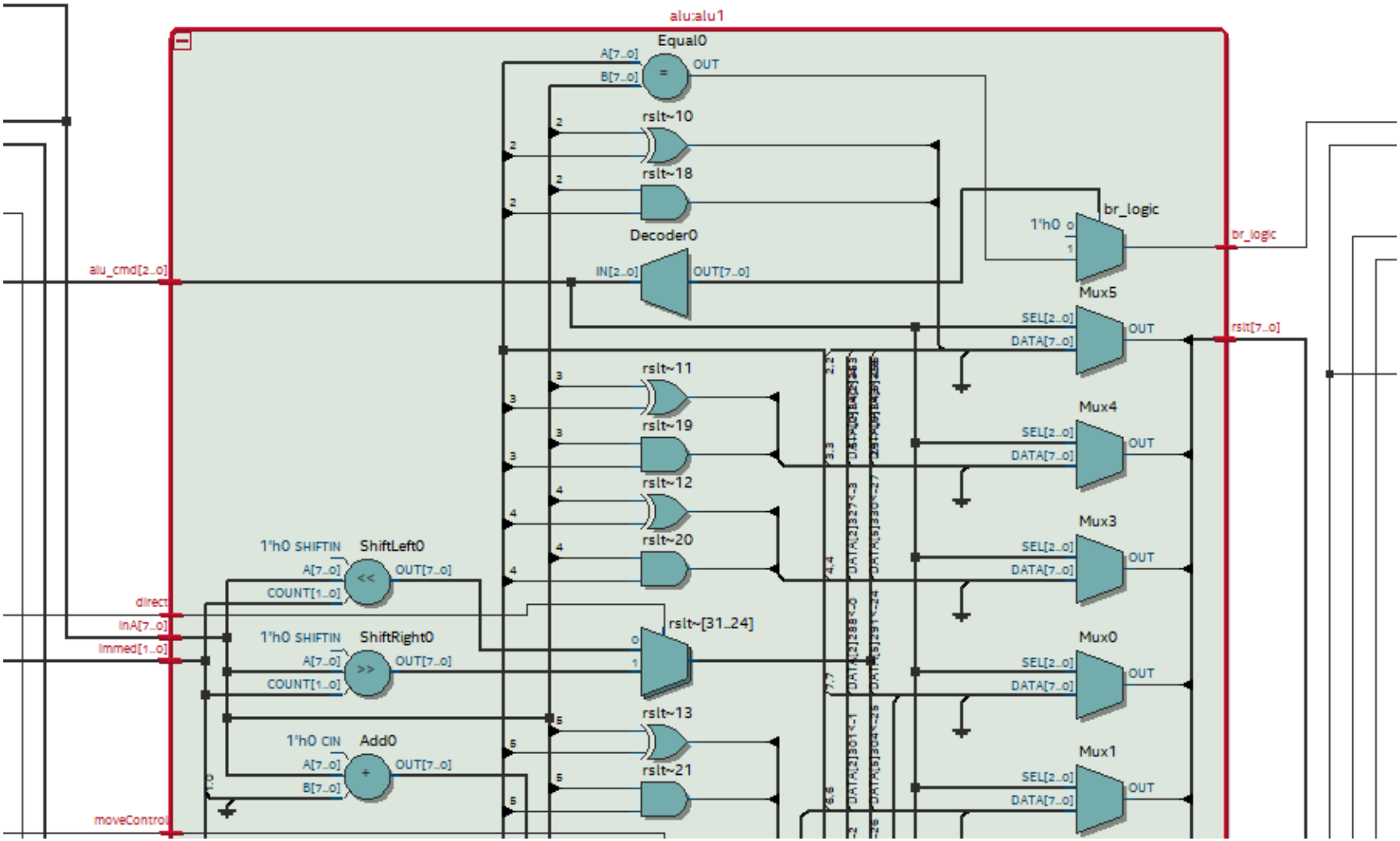
ALU Operations

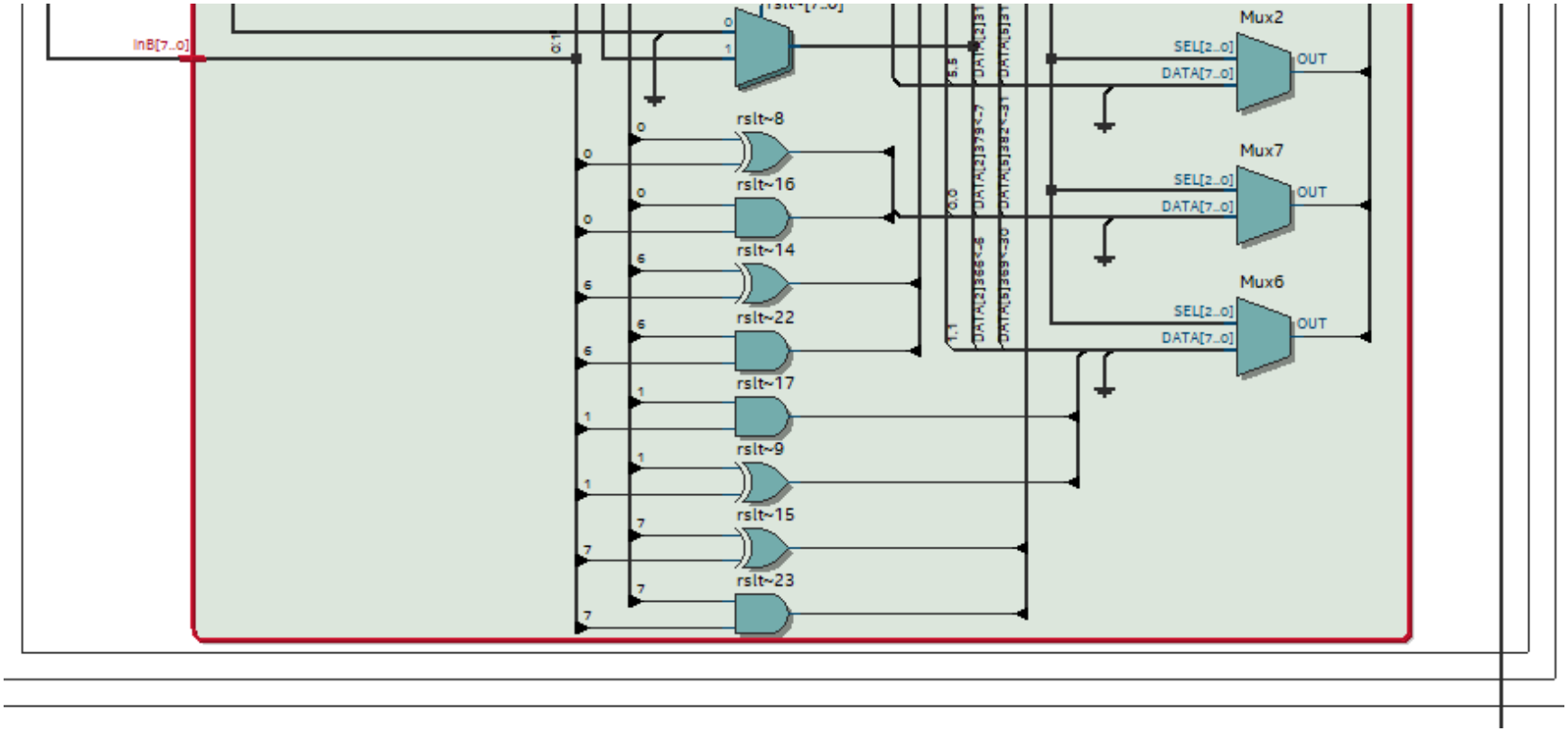
TODO. What ALU operations will you be demonstrating? What instructions are they relevant to?
AND - Bitwise AND
XOR - Bitwise XOR
SHIFT - Bitwise shift left or right

Loading and Storing:
LDR - Load from memory
STR - Store to memory
MOV & ADD - Move an immediate value to the source register or add an immediate value to the source register

Loop Counters:
CMP - compare registers
BEQ - Branch if equal

Schematic


alu:alu1
Equal0
A[7..0]
B[7..0]
=
OUT
rslt~10
rslt~18
Decoder0
IN[2..0]
OUT[7..0]
alu_cmd[2..0]
br_logic
1'h0
br_logic
Mux5
SEL[2..0]
DATA[7..0]
OUT
rslt[7..0]
rslt~11
rslt~19
rslt~12
rslt~20
Mux4
Mux3
Mux0
Mux1
1'h0 SHIFTIN
ShiftLeft0
<<
COUNT[1..0]
direct
inA[7..0]
immed[1..0]
1'h0 SHIFTIN
ShiftRight0
>>
rslt~[31..24]
rslt~13
rslt~21
1'h0 CIN
Add0
+
moveControl


InB[7..0]
rslt~8
rslt~16
rslt~14
rslt~22
rslt~17
rslt~9
rslt~15
rslt~23
Mux2
Mux7
Mux6
SEL[2..0]
DATA[7..0]
OUT

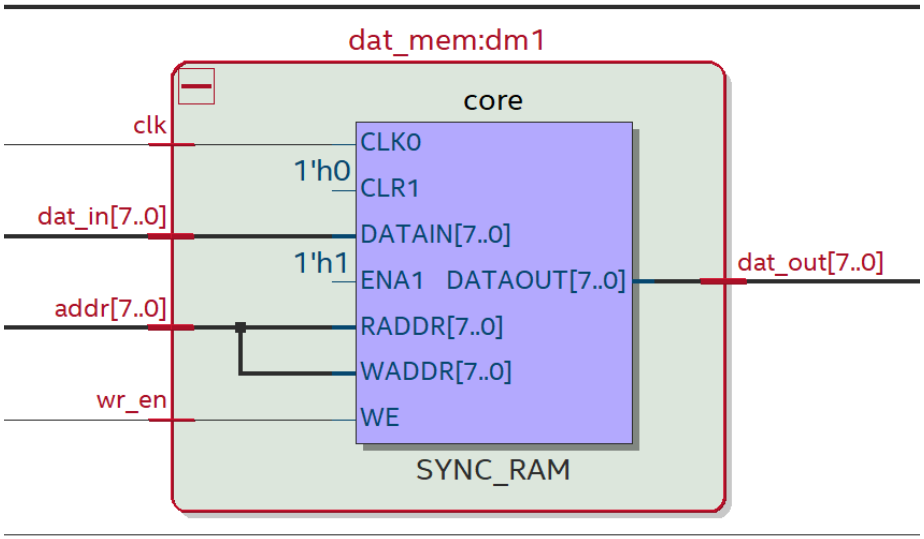
Data Memory

Module file name: dat_mem.sv

Functionality Description

The Data Memory module provides access to the data memory of the system. It allows for reading from and writing to memory locations, enabling the processor to interact with stored data.

Schematic



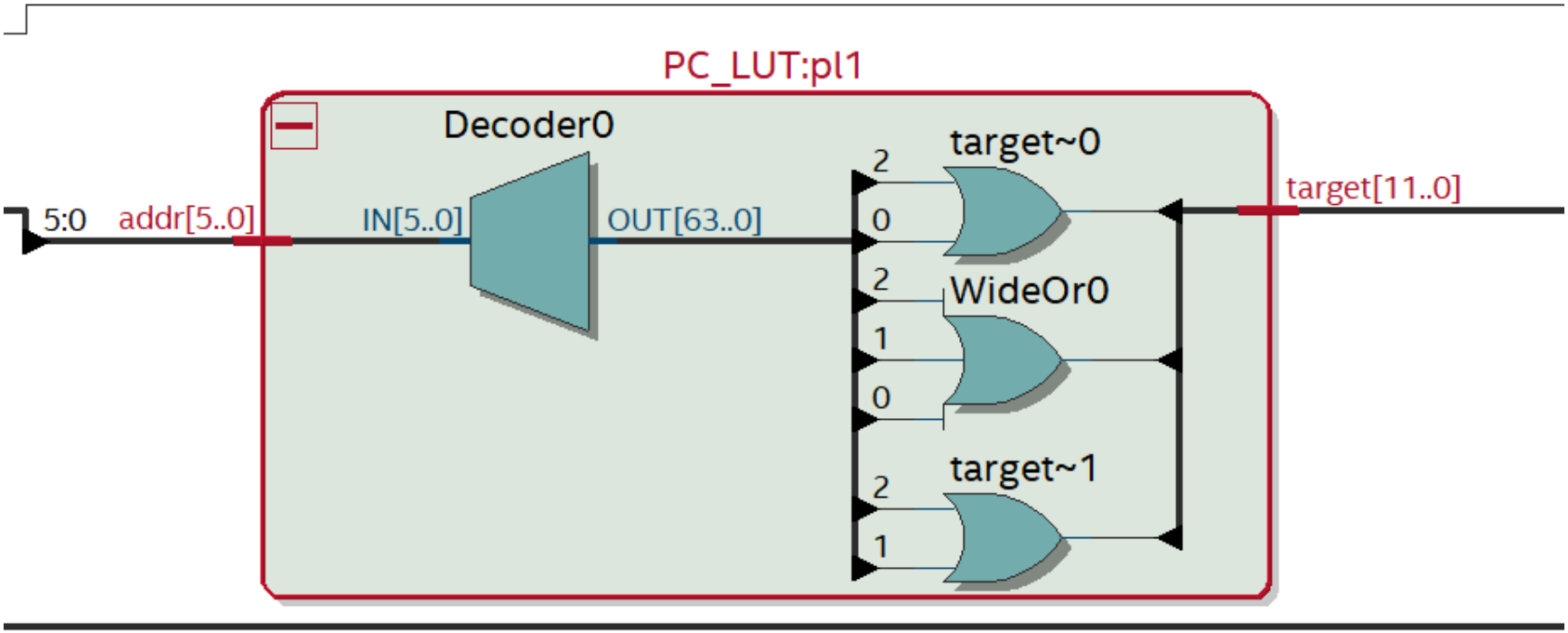
Lookup Tables

Module file name: PC_LUT.sv

Functionality Description

The Lookup Tables module works in conjunction with the Program Counter module to facilitate large jumps within the program. It stores predefined jump addresses and provides them to the Program Counter when needed, allowing for non-sequential execution flow.

Schematic



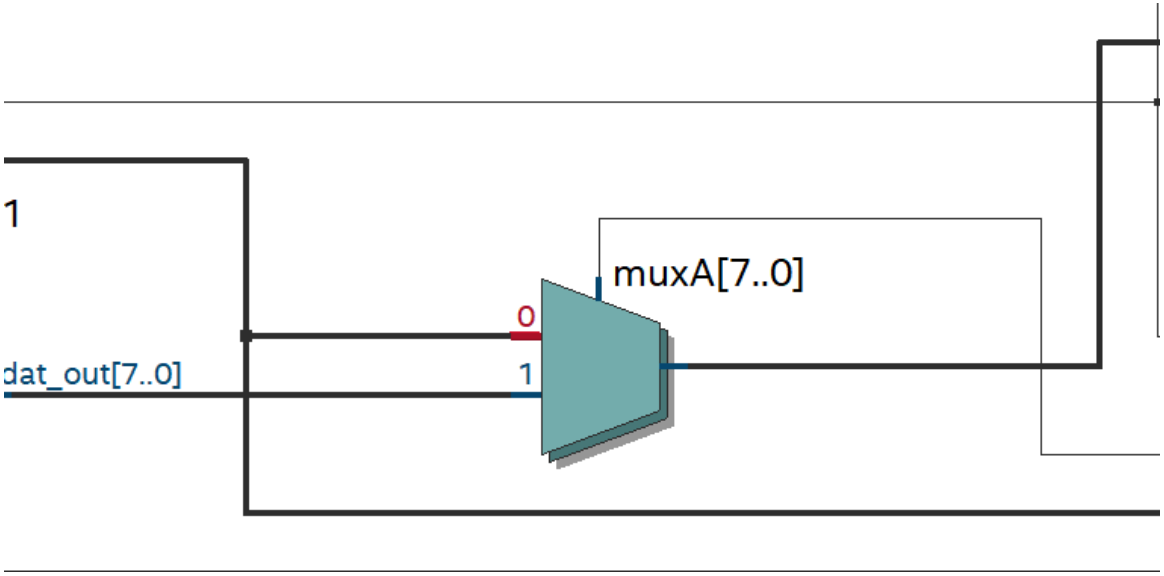
Muxes (Multiplexers)

Module file name: muxA

Functionality Description

The Multiplexers (Muxes) module is used to select specific values as output based on control signals. It takes multiple inputs and, based on the control signal, chooses one of them as the output. This is essential for routing data and control signals to the appropriate destinations within the processor. In our case muxA is used to determine whether the register file receives the results from the ALU or the data output from the data memory.

Schematic



7. Changelog

- Milestone 2
  - Introduction
    - edited to change from a load/store architecture to accumulator architecture.
  - Machine Specification
    - remove 1 bit for opcode, add 1 bit for destination register, and update remaining bits to satisfy our operations.
    - remove instruction set and operation breakdown to support 3 bit opcode for the rest of our operations
    - Case function for our LDR, STR, and MOV function using a selection cmd (3 bit)
    - Switched to 8 registers instead of 16
  - Control Flow
    - add lookup table to overcome large jump
  - Programmer's Model [Lite]
    - add the comment that copying from MIPS and ARM with modifications is allowed.
  - Program Implementation
    - Instantiated all our modules and implemented our ALU to support 8 operations.
- Milestone 1
  - Initial version

Milestone 3

```
def convert(inFile, outFile1):
	assembly_file = open(inFile, 'r')
	machine_file = open(outFile1, 'w')
	assembly = list(assembly_file.read().split('\n'))

	#keep track of index and file line number
	lineNum = 0;
	labelsNum = 0;

	#dictionaries to ease conversion of opcodes/operands to binary (PREVIOUS
	# OPCODES GIVEN)
	# opcodes = {'ADR' : '000', 'ADI' : '001', 'STR' : '010', 'LDR' : '011',
	# 'MOV' : '100', 'CMP' : '101', 'BR' : '110', 'HT' : '111'}

	opcodes = {'LDR': '000', 'STR': '001', 'MOV': '010', 'XOR': '011',
	'AND': '100', 'SHIFT' : '101', 'CMP' : '110', 'BEQ': '111', 'ADD': '0000'}

	registers = {'r0' : '000', 'r1' : '001', 'r2' : '010', 'r3' : '011',
	'r4' : '100', 'r5' : '101', 'r6' : '110', 'r7' : '111'}

	distance = {'p1for1': '000000', 'p1for2': '000111'}

	#reads through file to convert instructions to machine code
	for line in assembly:
		output = ""
		instr = line.split(); #split to get instruction and different operands\
		operation = opcodes[instr[0]]
		#make sure it is an instruction, skip over labels
		if instr[0] in opcodes:
```

```
# AND and XOR
# Example of AND and XOR
# AND R0 R1
# instr[0] = 'AND' instr[1]=R0 instr[2]=R1
# 100_000_001
if operation == '100' or operation == '011':
      # R-type instruction
      output += operation
      output += registers[instr[1]]
      output += registers[instr[2]]
# SHIFT
# Example of SHIFT
# SHIFT r0 R immediate
# instr[0] = 'SHIFT' instr[1]=R0 instr[2]=1 instr[3]=immediate
elif operation == '101':
      # R-type instruction
      output += operation
      output += registers[instr[1]]
      if instr[2] == "R":
            output += '1'
      else:
            output += '0'
      output += bin(int(instr[3]))[2:].zfill(2)
# LDR and STR
# Example of LDR and STR
# LDR r0 r1
# instr[0] = 'LDR' instr[1]=R0 instr[2]=R1
elif operation == '000' or operation == '001':
      # I type instruction
      output += operation
      output += registers[instr[1]]
      output += registers[instr[2]]
```

```
# MOV/AND
# Example of MOV//AND
# MOV R0 control immediate / AND R0 control immediate
# instr[0] = 'MOV' instr[1]=control instr[2]=R0 instr[3]=immediate
elif operation == '010' or operation == '0000':
      # I type instruction
      output += '010'
      output += registers[instr[1]]
      if operation == '010':
            output += '0'
      else:
            output += '1'
      output += bin(int(instr[2]))[2:].zfill(2)
# CMP
# Example of CMP
# CMP R0 R1
# instr[0] = 'CMP' instr[1]=R0 instr[2]=R1
elif operation == '110':
      # J type instruction
      output += operation
      output += registers[instr[1]]
      output += registers[instr[2]]
# BEQ
# Example of BEQ
# BEQ target
# instr[0] = 'BEQ' instr[1]=target
else:
      output += operation
      output += distance[instr[1]]
#write binary to machine code output file
machine_file.write(str(output) + '\t// ' + line + '\n')
```

```
        assembly_file.close()
        machine_file.close()

convert("assembly.txt", "machine.txt")
```

Changelog Milestone 3

- Milestone 3
    - Machine Specification
        - Redesigned our instruction set to satisfy our requirements
            - Changed LDR and STR to take two (3 bit) source registers.
            - Removed OR and ADD set under R-type
            - Added a control bit to distinguish between ADD and MOV source register for immediate values (I-type).
            - Added a CMD and BEQ branch type to control our looping and if statement functions
        - Redesigned our SHIFT function
            - Added direct bit to distinguish between left and right bits
        - Adjusted our operation and machine code designs to simulate our changes.
    - Control Flow
        - Add a mux
    - Program Implementation
        - Add a mux module to control the data out vs alu results.
        - Redesigned the top level design to remove inputs and wires that were not being used.
        - Fixed major design conflicts and instantiations with the top level.

- Milestone 2
    - Introduction
        - edited to change from a load/store architecture to accumulator architecture.
    - Machine Specification
        - remove 1 bit for opcode, add 1 bit for destination register, and update remaining bits to satisfy our operations.
        - remove instruction set and operation breakdown to support 3 bit opcode for the rest of our operations

        - Case function for our LDR, STR, and MOV function using a selection cmd (3 bit)
        - Switched to 8 registers instead of 16
    - Control Flow
        - add lookup table to overcome large jump
    - Programmer's Model [Lite]
        - add the comment that copying from MIPS and ARM with modifications is allowed.
    - Program Implementation
        - Instantiated all our modules and implemented our ALU to support 8 operations.
- Milestone 1
    - Initial version