# CSE 141L Final Report

Asher Av, A16047586; Anoop Gunawardhena , A15474948; Rick Truong, A15483726

## 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.

Asher Av
Anoop Gunawardhena
Rick Truong

# 0. Team

- Asher Av
- Anoop Gunawardhena
- Rick Truong

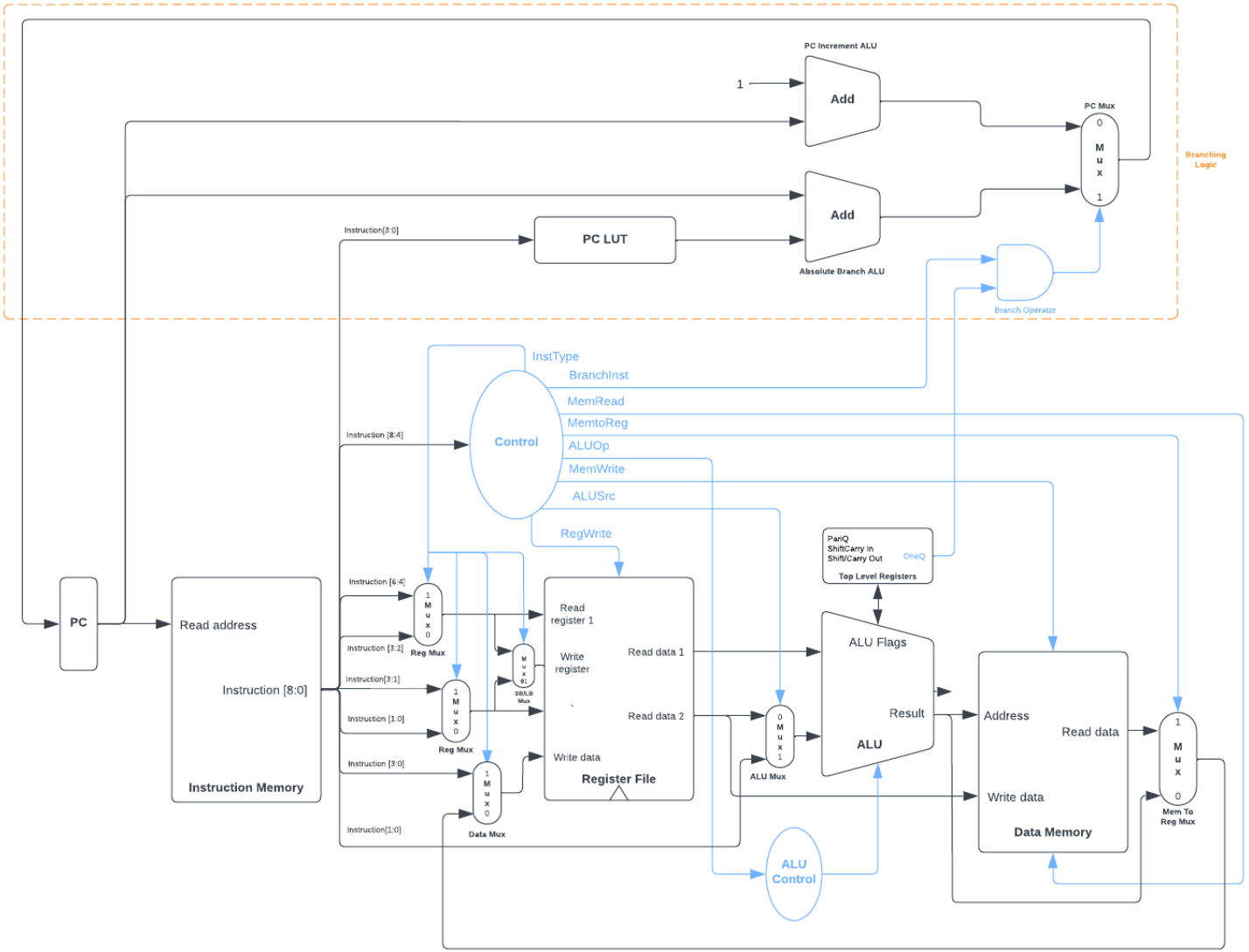
# 1. Introduction

Our architecture is called the streamlined-store architecture, it is a general-purpose register oriented architecture that uses a small amount of registers to cover a compact instruction set in a manner that strikes a balance between program complexity and architectural simplicity.

The guiding theme when making design choices for our infrastructure was to maximize efficiency through simplicity, how could we get the most out of a limited amount of registers and instruction bits? This kind of thinking led to us incorporating strategies like bit masking,hardwiring to maximize the amount of instructions and registers that could be specified by any 9-bit instruction. We were also selective in our instruction set, including only instructions that had seemed necessary for implementing the three programs.

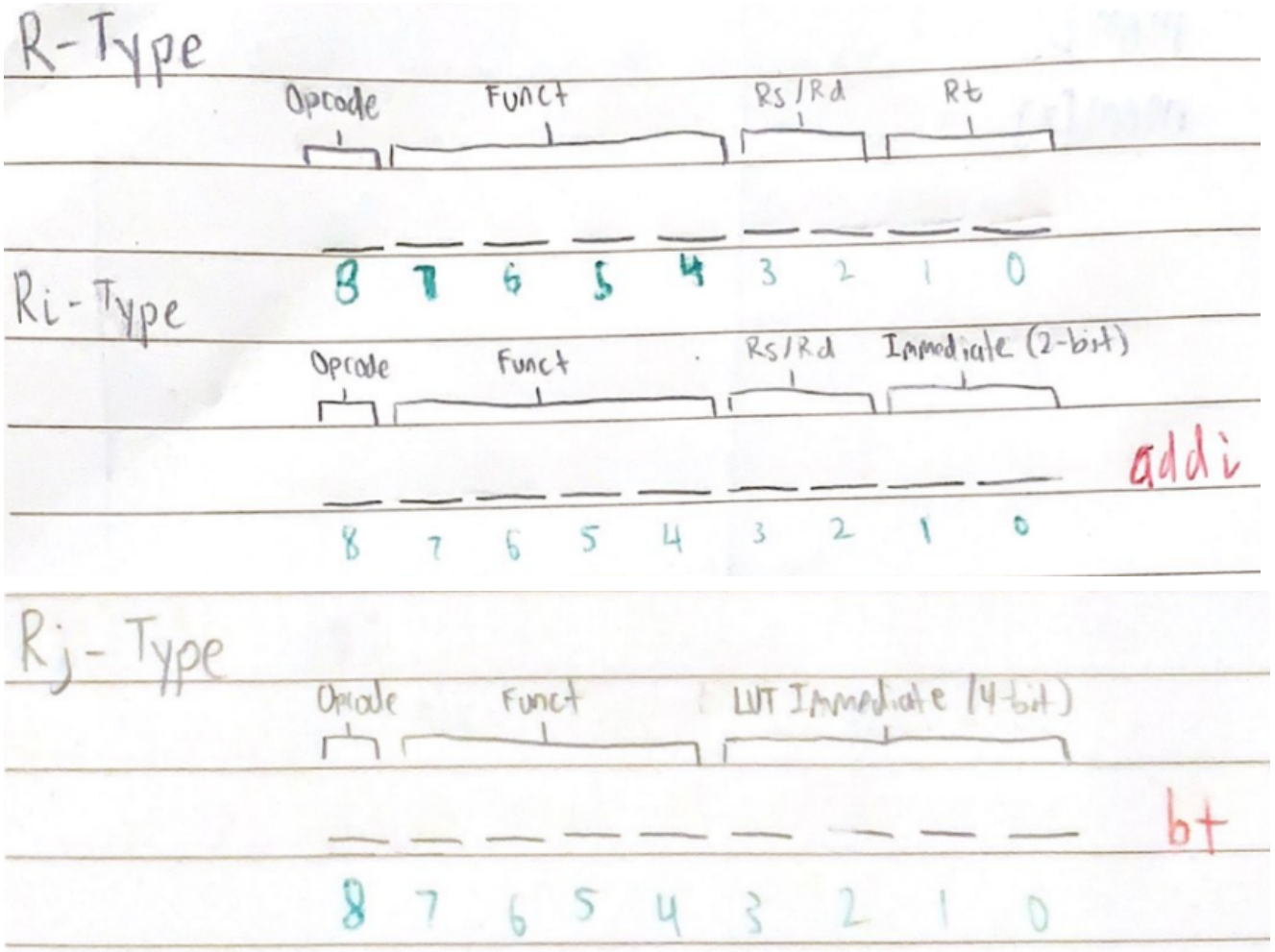
The standard from which our architecture borrows the most from is the load-store architecture because it is the most common general-purpose architecture, it's quite easy to understand and also its flexibility reduces the would-be complexity of writing programs for it. For all the aforementioned reasons and particularly the last is why we chose to base our architecture on the load-store architecture.

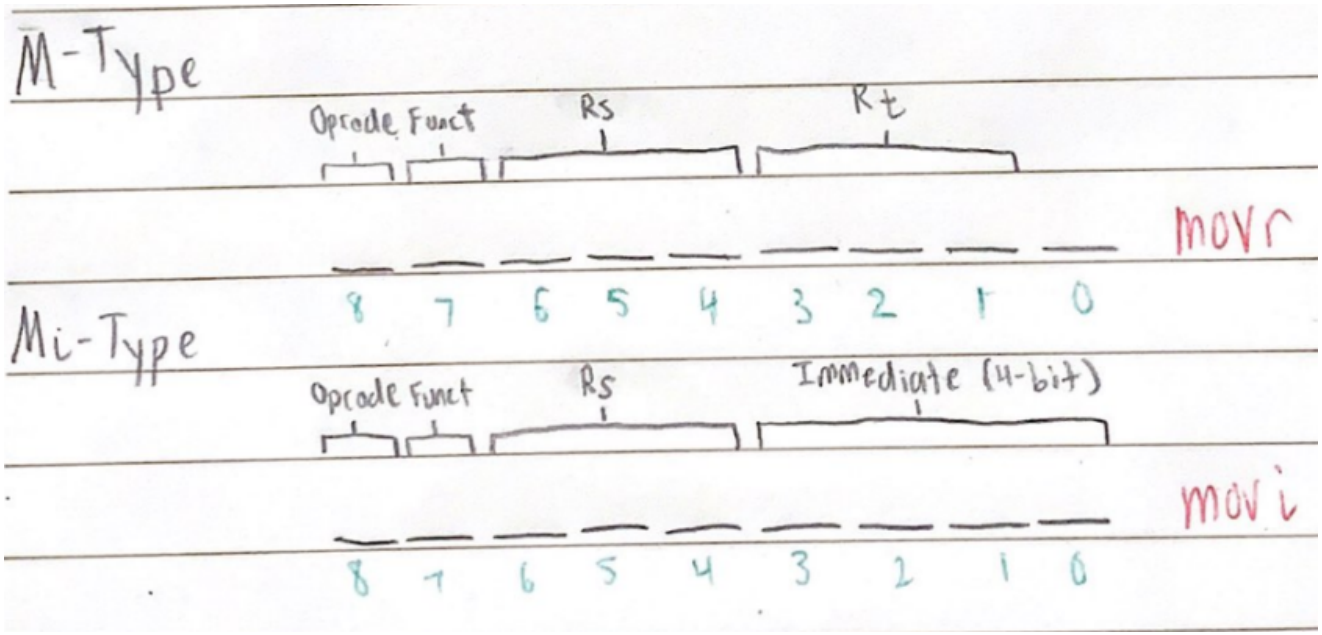
# 2. Architectural Overview



# 3. Machine Specification

Instruction formats






| TYPE | Subtype | FORMAT | CORRESPONDING INSTRUCTIONS |
|---|---|---|---|
| R | | 1 bit opcode, 4 bits funct, 2 bit operand register, 2 bit operand register | add, sub, lb, sb, nor, xor, and, or, sll, slr, eq, lt, rxor |
| R | i | 1 bit opcode, 4 bits funct, 2 bit operand register, 2 bit immediate | addi |
| R | j | 1 bit opcode, 4 bits funct, 4 bit immediate | bt, bne |
| M | | 1 bit opcode, 1 bit funct, 3 bit operand register, 3 bit operand register, 1 bit free | movr |
| M | i | 1 bit opcode, 1 bit funct, 3 bit operand register, 4 bit immediate | movi |

# Operations

| NAME | TYPE | BIT BREAKDOWN | EXAMPLE | NOTES |
|---|---|---|---|---|
| Add | R | 1 bit op code (0),<br>4 bit funct code (0000),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = R[rs] + R[rt]<br><br>; Assume $1 has 0b0001_0001<br>; Assume $2 has 0b0001_0010<br>add $1, $2 ⇔ `0_0000_01_10`<br>; After instruction $1 now holds `0b0001_0011` | 8 implied register bits<br>First operand has implied 0<br>Second operand has implied 1 |
| Sub | R | 1 bit op code (0),<br>4 bit funct code (0001),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = R[rs] - R[rt]<br><br># Assume $1 has 0b0001_0001<br># Assume $2 has 0b0001_0010<br>Sub $2, $1 ⇔ `0_0001_01_10`<br># After instruction $2 now holds `0b0001_0001` | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |
| Addi | Ri | 1 bit op code (0),<br>4 bit funct code (0010),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = R[rs] + Immediate<br><br>; Assume $2 has 0b0001_0001<br>addi $2, #1 ⇔ `0_0010_01_01`<br>; After instruction $2 now holds `0b0001_0010` | Second operand has 2-bit signed decimal range: [-2, 1] |
| Load Byte | R | 1 bit op code (0),<br>4 bit funct code (0011),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = M[R[rt]]<br><br>;Assume $2 has `0b00110110`<br>;Assume #1 has `0b00110111` which is an address that holds `0b11011011` | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |

| | | | | |
|---|---|---|---|---|
| | | | lb $1, $2 ⇔ `0_0011_01_10`<br>;After instruction $1 now holds `0b11011011` | |
| Store Byte | R | 1 bit op code (0),<br>4 bit funct code (0100),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | M[R[rs]] = R[rt]<br><br>;Assume that $rt contains address `0b00000010`, and $rs contains `0b11110000`.<br><br>sb $1, $2 ⇔ `0_0100_01_10`<br><br>;The memory address `0b00000010`, contains `0b11110000`. | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |
| Branch on True | Rj | 1 bit op code (0),<br>4 bit funct code (0101),<br>4 bit immediate | If (OneQ) PC = LUT Immediate<br><br>;Assume Loop corresponds to LUT[0]=2<br><br>bt @Loop ⇔ `0_0101_00_00`<br><br>; PC is now `0b00000010` | The branch condition is checked implicitly against OneQ(as to be performed in another instruction). .<br>The immediate corresponds to an index in the Look Up Table which references an absolute address to jump to. |
| Bitwise Nor | R | 1 bit op code (0),<br>4 bit funct code (0111),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = ~(R[rs] \| R[rt])<br><br>;Assume that $1 contains `0b11110000` and $2 contains `0b00001111`<br><br>nor $1, $2 ⇔ `0_0111_01_10` | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |

| | | | | |
|---|---|---|---|---|
| | | | ;$1 will now only contain `0b00000000` | |
| Bitwise Xor | R | 1 bit op code (0),<br>4 bit funct code (1000),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = R[rs] ^ R[rt]<br><br>;Assume that $1 contains `0b11110000` and $2 contains `0b00001111`<br><br>xor $1, $2 ⇔ `0_1000_01_10`<br><br>;$1 will now contain `0b11111111` | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |
| Bitwise And | R | 1 bit op code (0),<br>4 bit funct code (1001),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = R[rs] &&R[rt]<br><br>;Assume that $1 contains `0b11110000` and $2 contains `0b00001111`<br><br>and $1, $2 ⇔ `0_1001_01_10`<br><br>;$1 will now only contain `0b00000000` | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |
| Bitwise Or | R | 1 bit op code (0),<br>4 bit funct code (1010),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = R[rs] \|\| R[rt]<br><br>;Assume that $1 contains `0b11110000` and $2 contains `0b00001111`<br><br>or $1, $2 ⇔ `0_1010_01_10`<br><br>#$1 will now contain `0b11111111` | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |

| | | | | |
|---|---|---|---|---|
| Logical Shift Left | R | 1 bit op code (0),<br>4 bit funct code (1011),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = R[rs] << R[rt]<br><br>;Assume that $1 contains the data `0b11110000` and that $2 contains the value 4<br><br>sll $1, $2 ⇔ `0_1010_01_10`<br><br>;$1 will now contain `0b00000000` | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |
| Logical right shift | R | 1 bit op code (0),<br>4 bit funct code (1100),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = R[rs] >> R[rt]<br><br>;Assume that $1 contains the data `0b11110000` and that $2 contains the value 4<br><br>slr $1, $2 ⇔ `0_1100_01_10`<br><br>;$1 will now contain `0b00001111` | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |
| Equal | R | 1 bit op code (0),<br>4 bit funct code (1101),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = R[rs] == R[rt]<br><br>;Assume that both $1 and $2 contain the same value<br><br>eq $1, $2<br><br>;After the instruction $1 will hold `0b00000001 or 1`. However if the values were unequal $1 would contain 0,`0b00000000`. | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |
| Less than | R | 1 bit op code (0),<br>4 bit funct code (1110),<br>2 bit operand/dest(XX), | R[rs] = R[rs] < R[rt]<br><br>;Assume that $1 contains | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers |

| | | | | |
|---|---|---|---|---|
| | | 2 bit operand register (XX) | 0b00000011 and $2 contains 0b00000111<br><br>lt $1, $2 ⇔ 0_1110_01_10<br><br>;After the instruction $1 will contain 1, 0b00000001, if $1 contained a larger number than 7, then $1 would contain 0,0b00000000 | $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |
| Reduction Xor | R | 1 bit op code (0),<br>4 bit funct code (1111),<br>2 bit operand/dest(XX),<br>2 bit operand register (XX) | R[rs] = ^ R[rt]<br><br>;Assume that before the instruction $2 contains the code 0b10000001<br><br>rxor $1,$2 ⇔ 0_1111_01_xx<br><br>;$1 now contains 0b00000000 | 8 implied register bits<br>First operand has implied 0 so the first register operand must be from registers $0~$3<br>Second operand has implied 1 so the first register operand must be from registers $4~$7 |
| Move (Register) | M | 1 bit op code (1),<br>1 bit funct code (0),<br>3 bit operand/dest(XXX),<br>3 bit operand register(XXX) | R[rs] = R[rt]<br><br>;Assume that before the instruction $2 contains the value 0b00101101<br><br>movr $1, $2 ⇔ 1_0_001_0010<br><br>;After the instruction $1 now contains 0b00101101 | |
| Move (Immediate) | Mi | 1 bit op code (1),<br>1 bit funct code (1),<br>3 bit operand/dest(XXX),<br>4 bit immediate (XXXX) | R[rs] = Immediate<br><br>; Assume that $1 has 0b00000000<br><br>movi $1, #-8 ⇔ 1_1_001_1111 | (4-bit signed decimal range: [-8, 7]) |

| | | | | |
|---|---|---|---|---|
| | | | ;After instruction $1 now contains<br>`0b00001111` | |

## Internal Operands

- $Program\ counter\ (PC)\ CPU\ Register$
- $Register\ file\ space = 2^3 = 8\ registers\ (1 - byte)$
  - $8\ General - purpose\ registers$
    - $\$0,\ \$1,\ \$2,\ \$3,\ \$4, \$5, \$6,\ \$7$

# Control Flow (branches)

## Supported Branch Types

- Branch on True $(bt)$

## Address Calculation

- Absolute addressing with register direct addressing mode
- Address calculation in Verilog
  - $If\ (OneQ)\ PC\ = LUT\ immediate$

## Maximum Branch Distance Supported

$Maximum\ branch\ distance\ supported$: $2^{12}$ $instruction\ entries$

# Addressing Modes

This architecture relies solely on direct register-based addressing, this means that the addresses are pre calculated and stored in registers which are used as operands in instructions that require data retrieval or transfer. We selected this method because it is very flexible and allow up to a large comprehensive(8 bits) amount of addresses that can be reached.An example is presented below

```
movi $3, 54   #moves the address 54 into the register R[$3]
lw $1, $3     #loads the byte stored at memory location 54 into the register R[$1]
```

# 4. Programmer's Model [Lite]

In order to operate our streamlined-store machine, our programmer should familiarize himself/herself with the general machine specifications. Our programmer should familiarize themselves with the two different data buses which our machine data path operates on: 9-bit instructions, present only from instruction memory outputs, and 8-bit (1-byte) data buses, present in outputs in the register file, ALU, and data memory block. The width of our machine's register memory elements, ALU computation elements, and data memory elements are 8-bits.

In addition, our programmer should acquaint themselves with the following nuances in our R-Type instruction formats. The first register operand (denoted as Rs/Rd from bits [3:2]) may only reference registers $0 - $3. The second register operand (denoted as Rt from bits [1:0]) may only reference registers $4 - $7. In contrast, in our machine's I-Type instructions, all register operands may reference all register addresses in our register file.

There's no consensus on whether a programmer should prioritize loading in the necessary values from memory into as many registers as possible, then perform calculations, or not. We recommend to programmers to designate one register to hold values to point to memory and load and store values from-and-to memory using that register.

We cannot copy instructions from MIPS or ARM ISA without modification. This is because, in contrast, we are limited to 9 registers and each of our registers holds 8-bits(1 byte) of information.
To overcome this, each of our fields were reduced significantly or otherwise omitted. Additionally we implied certain bits in our register fields. Generally our first register field also behaves as the destination register.

Our ALU will be used for non-arithmetic instructions which includes logical instructions(nor, xor, and, or, sll, slr, rxor), data transfer instructions(lb, sb, movr), and conditional instructions(bt, eq, lt ). Essentially the ALU will be used for every instruction except movi.This decision has not complicated our design thus far. Even if the ALU does not do any meaningful computation, we simply set the result directly from the second operand or inputB.

# 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. Muxes

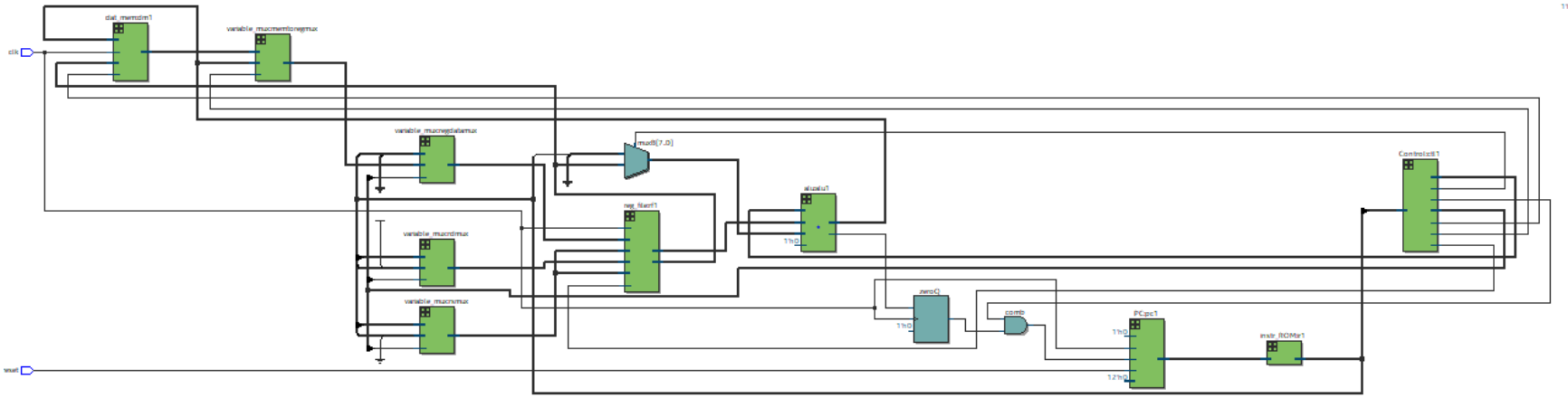
## Top Level

Module file name: top_level.sv

### Functionality Description

Top Level contains all other modules.

## Schematic



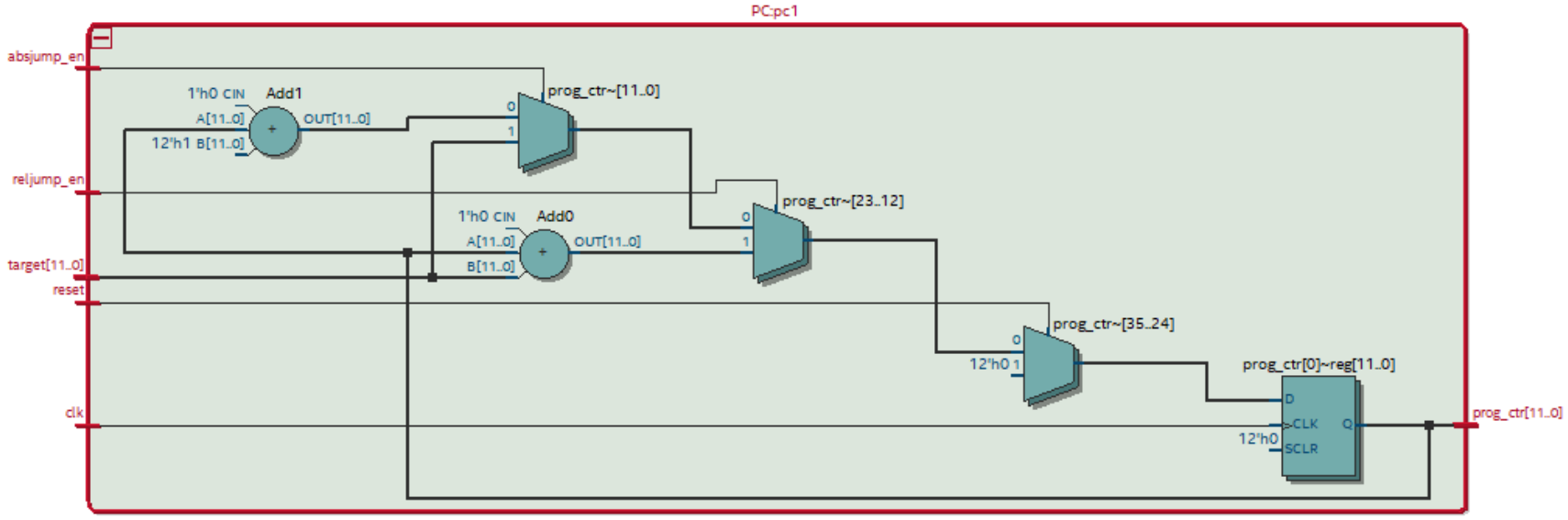
# Program Counter

Module file name: PC.sv

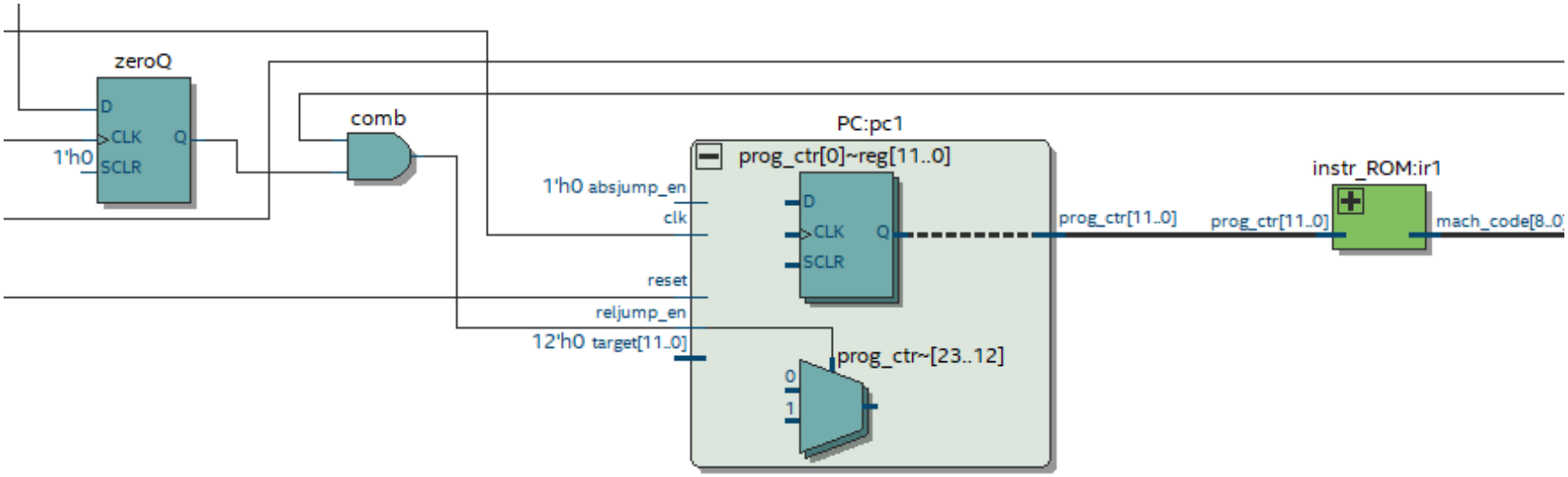
## Functionality Description

Program Counter points to the current instruction while supporting both relative and absolute jumps.

## Schematic




To also encapsulate the greater functionality of the fetch cycle here below is a schematic of the PC and its relevant paths, zero Q and the combinator represent the logical components that allow us to replace the functionality of a PC LUT.



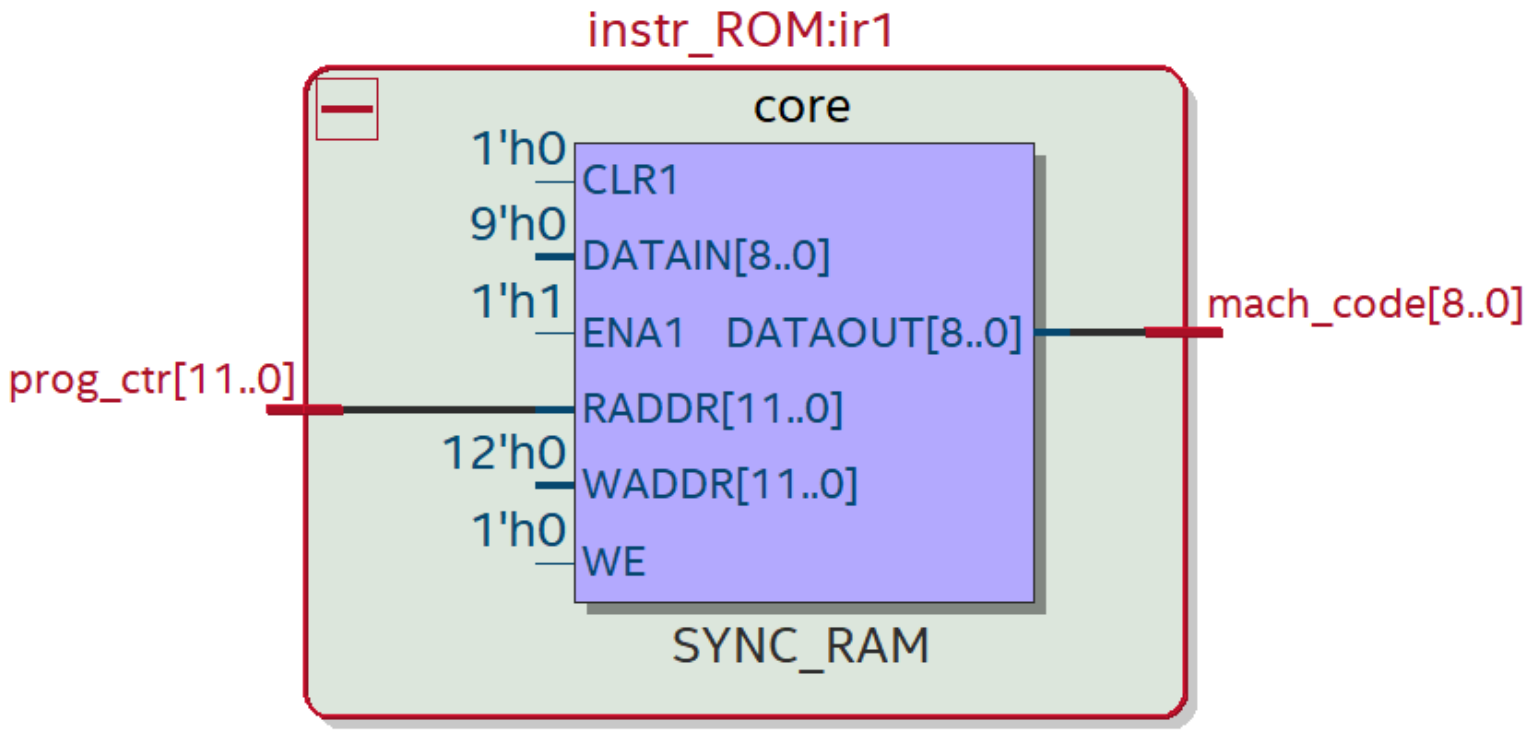

# Instruction Memory

Module file name: instr_ROM.sv

## Functionality Description

Instruction Memory takes as input a program counter address pointer and outputs machine code.

Schematic



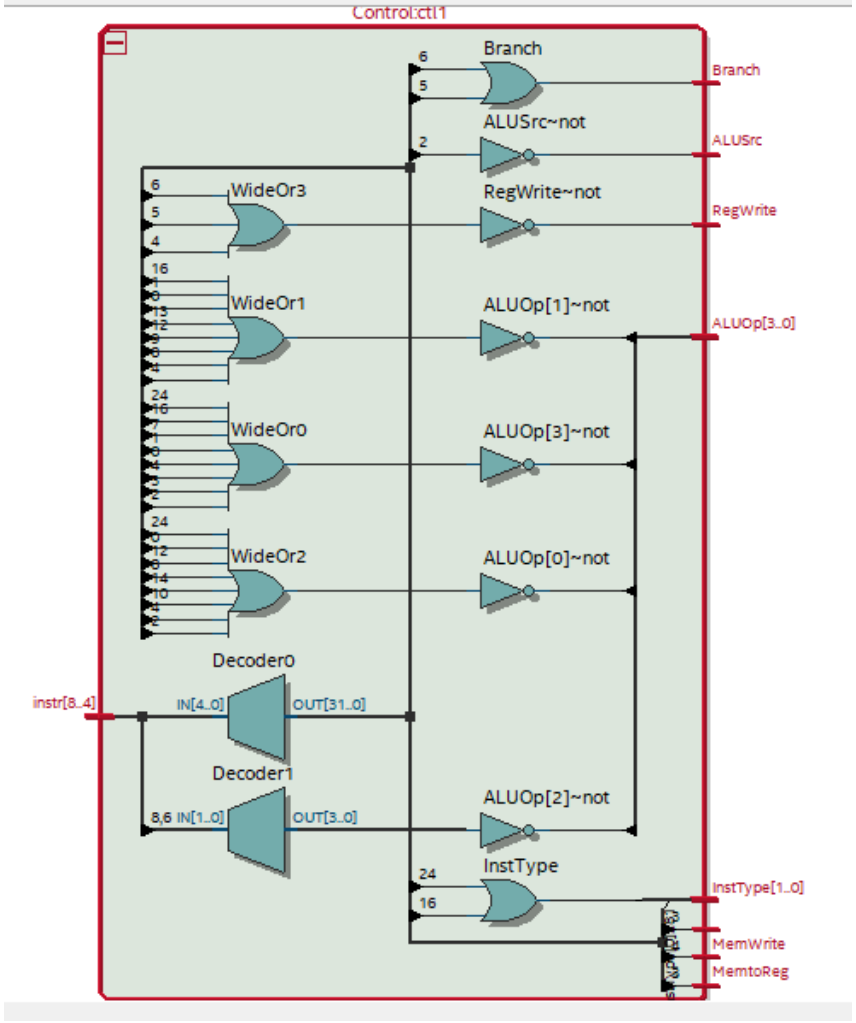

## Control Decoder

Module file name: Control.sv

### Functionality Description

Control Decoder takes as input an instruction opcode which decodes to an ALU operation.

## Schematic


Control:ctl1
Branch
Branch
ALUSrc~not
ALUSrc
WideOr3
RegWrite~not
RegWrite
WideOr1
ALUOp[1]~not
ALUOp[3..0]
WideOr0
ALUOp[3]~not
WideOr2
ALUOp[0]~not
Decoder0
instr[8..4]
IN[4..0]
OUT[31..0]
Decoder1
8,6 IN[1..0]
OUT[3..0]
ALUOp[2]~not
InstType
InstType[1..0]
MemWrite
MemtoReg

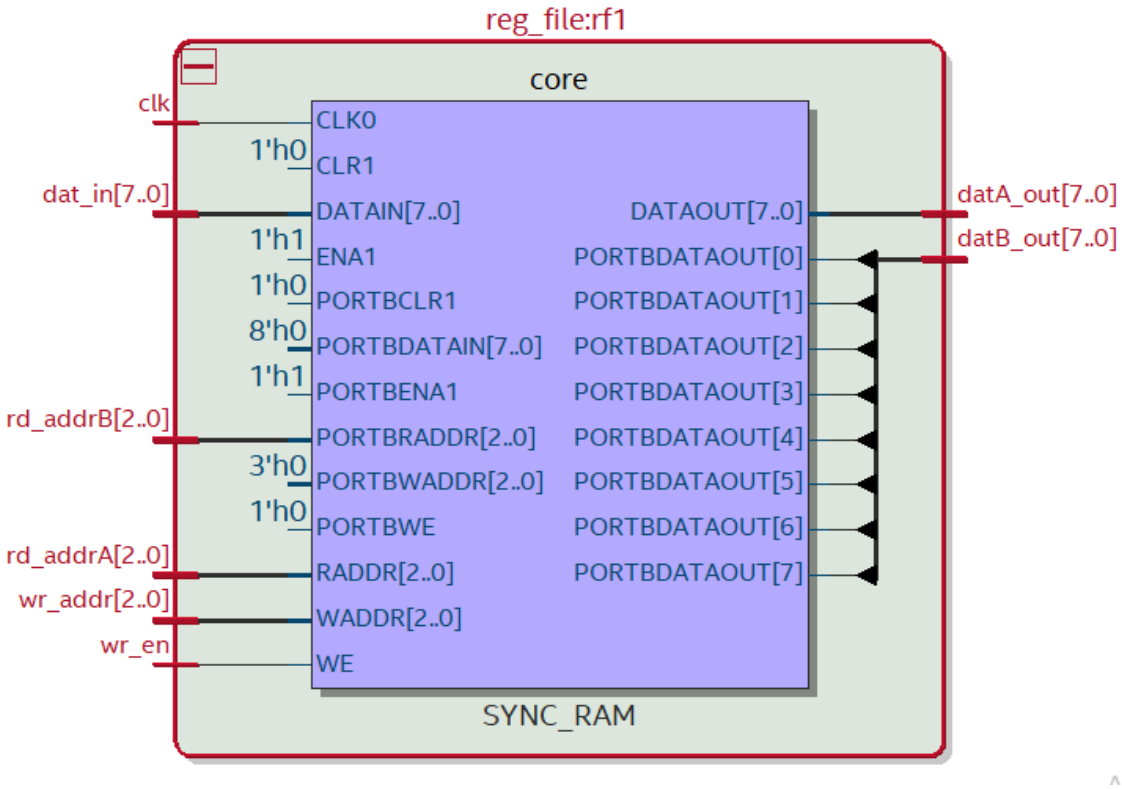
# Register File

Module file name: reg_file.sv

## Functionality Description

Register File takes as input Read register 1, Read register 2, a write register pointer, whether write is enabled, the write data, and outputs to Read data 1 and Read data 2.

## Schematic



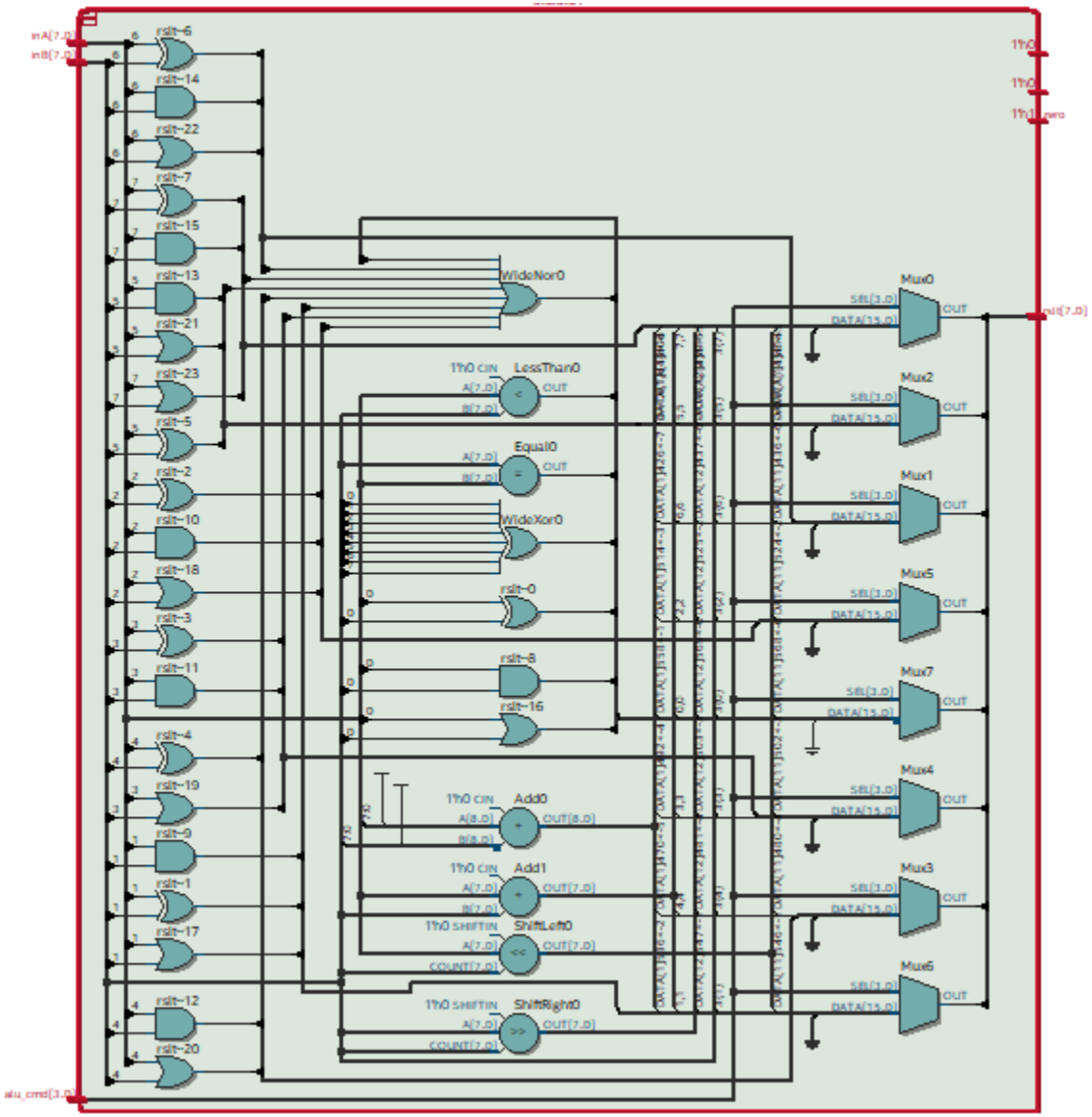
# ALU (Arithmetic Logic Unit)

Module file name: alu.sv

## Functionality Description

ALU outputs the computation of one or more operands.

## Schematic


rslt~6
rslt~14
rslt~22
rslt~7
rslt~15
rslt~13
rslt~21
rslt~23
rslt~5
rslt~2
rslt~10
rslt~18
rslt~3
rslt~11
rslt~4
rslt~19
rslt~9
rslt~1
rslt~17
rslt~12
rslt~20
WideNor0
LessThan0
Equal0
WideXor0
rslt~0
rslt~8
rslt~16
Add0
Add1
ShiftLeft0
ShiftRight0
Mux0
Mux2
Mux1
Mux5
Mux7
Mux4
Mux3
Mux6
OUT
SEL[3..0]
DATA[15..0]

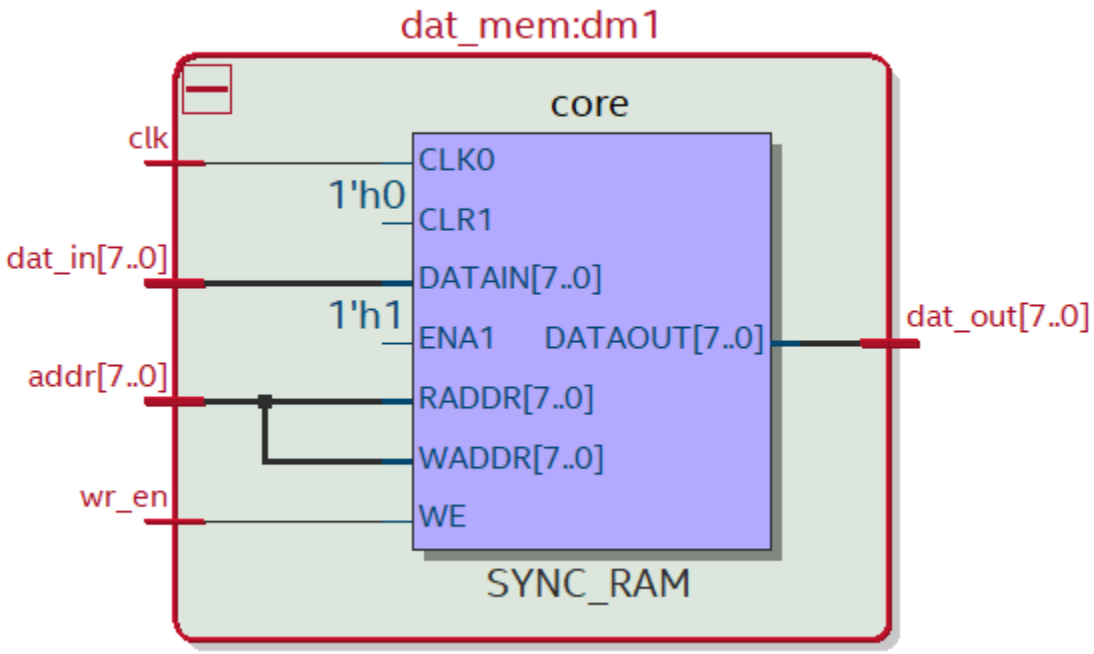
# Data Memory

Module file name: dat_mem.sv

## Functionality Description

Data Memory is used to read and write from memory.

## Schematic



# Muxes (Multiplexers)

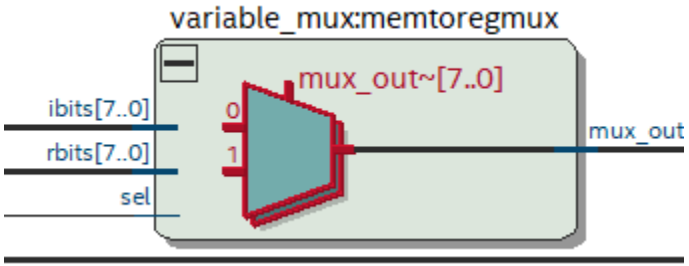
Module file name: muxrcompact.sv

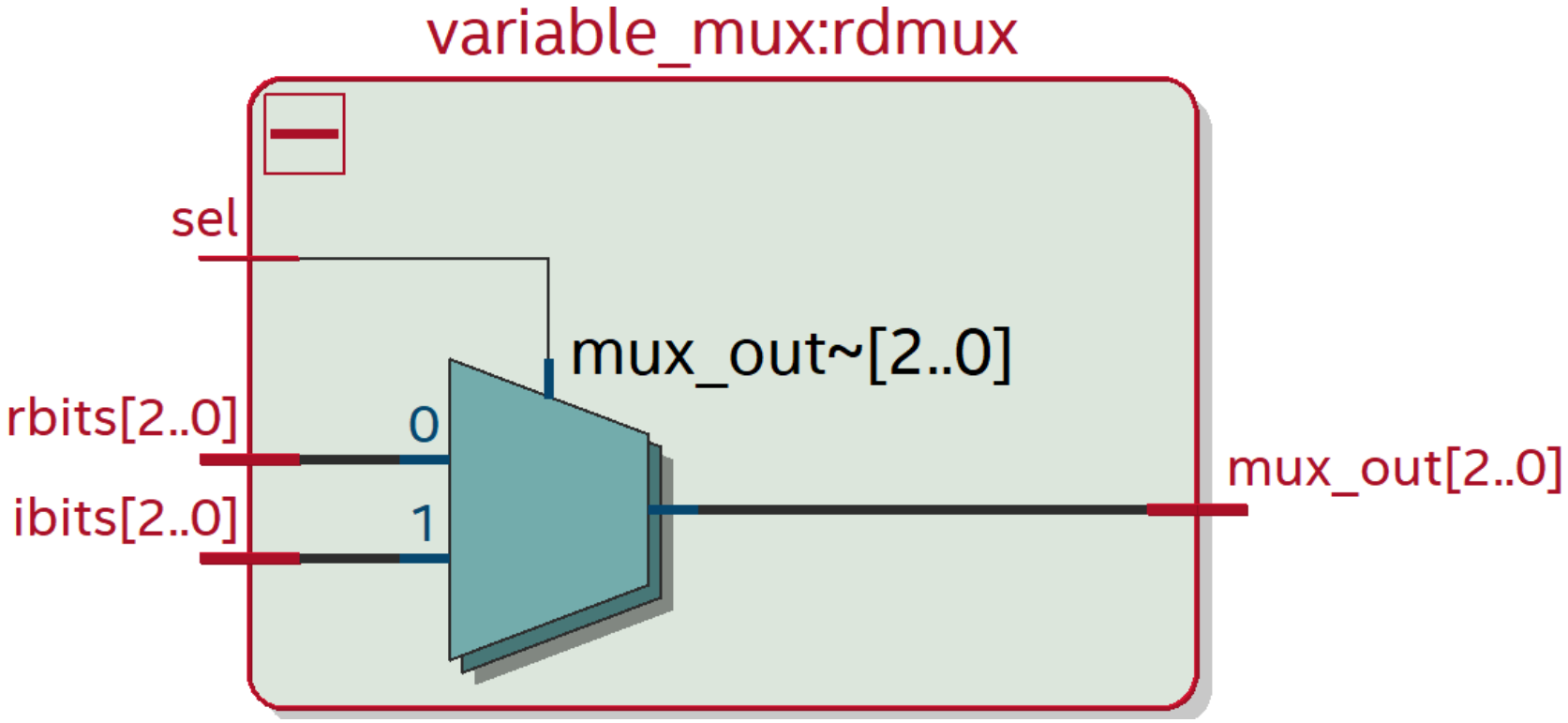
## Functionality Description

Muxes select between input signals and output the selected signal.

## Schematic

We use muxes of variable size for instance: 7-bit and 3-bit






## 6. Program Implementation

*Program implementations included in source code files.*

# 7. Changelog

- Milestone 4
    - Machine Specification
        - Operations
            - Updated BT description and example
            - Removed BNE as it is no longer a supported instruction
        - Control Flow
            - Supported Branch types
                - Updated beq to bt
            - Address calculation
                - Uses absolute addressing instead of PC-relative
            - Maximum branch distance supported
                - Updated maximum branch distance calculation
    - Programmer's Model [Lite]
        - Updated references of beq and bne to just bt
- Milestone 3
    - Machine Specification
        - Operations
            - Renamed beq to bt
            - Comments are denoted with " ; " instead of " # "
    - Assembler
        - Added assembler section
        - Provided example of input to and output from assembler.py program
            - Added command line usage example
            - Showed input line and output line example
- Milestone 2
    - Introduction
        - Removed prompt.
    - Architectural Overview
        - Our Architectural Overview has changed since Milestone I in the following ways:

- A Register Mux (Reg Mux) has been added into our datapath during our instruction decode stage to determine the value of Read register 1 for our Register File from an instruction
- A Register Mux (Reg Mux) has been added into our datapath during our instruction decode stage to determine the value of Read register 2 for our Register File from an instruction
- A Data Mux has been added into our datapath during our instruction decode stage to determine the value of Write data for our Register File from an instruction
- An ALU Mux has been added into our datapath during the end of our instruction decode stage to determine the value of Data in 2 for our ALU from an instruction
- Our Program Counter Lookup Table (PC LUT) has been omitted from our instruction fetch stage as branching is handled directly though relative addressing with register direct addressing

    - Machine Specification
        - Instruction Formats
            - Our Instruction Formats have been updated in the following ways:
                - We've decided to include an Ri-Type instruction format to handle our addi instruction
                - We've decided to rename our formerly known I-Type instruction formats to M-Type instructions formats
                - We've decided to distinguish between Move Register (movr) and Move Immediate (movi) instructions by including an Mi-Type instruction format
        - Operations
            - Removed prompt and example operation.
        - Internal Operands
        - Control Flow
        - Addressing Modes
    - Programmer's Model [Lite]
        - Our Programmer's Model[Lite] has been changed since Milestone I in the following ways:
            - We described whether our ALU will be used for non-arithmetic instructions and whether this would complicate our design.
- Milestone 1
    - Initial version

# 8. Assembler

- An example of input to and output from your assembler.
    - Assembler.py command line usage
        - python assembler.py INPUT_FILE OUTPUT_FILE
            - Ex: assembler.py “Dummy Programs\dummy.s” “Dummy Programs\dummy_mach_code.txt”
        - Command line example: (NOTE: print statements only for this demonstration, are removed for actual program)

```
PS C:\Users\Rick Truong\OneDrive\Desktop\Quartus Projects\CSE 141L Project - ARA> python .\assembler.py "Dummy Programs\dummy_program.s" "Dummy Programs\dummy_mach_code.txt"
unaltered line is movi $0, #0

uncommented line is movi $0, #0
1
MOVI
['0', '0']
unaltered line is lb $0, $4

uncommented line is lb $0, $4
2
LB
['0', '4']
unaltered line is movr $1, $4

uncommented line is movr $1, $4
3
MOVR
['1', '4']
unaltered line is addi $0, #1

uncommented line is addi $0, #1
4
ADDI
['0', '1']
unaltered line is lb $0, $5

uncommented line is lb $0, $5
5
LB
['0', '5']
unaltered line is add $1, $5

uncommented line is add $1, $5
6
ADD
['1', '5']
unaltered line is addi $0, #1

uncommented line is addi $0, #1
7
ADDI
['0', '1']
unaltered line is movr $6, $1

uncommented line is movr $6, $1
8
MOVR
['6', '1']
unaltered line is sb $0, $6
uncommented line is sb $0, $6
9
SB
['0', '6']
```

- Line-by-Line Example
  - Input line from input file(1st arg)
    - `movi    $0, #0                    ; Initialize i = 0 ($0);`
  - Output line from output(2nd arg) file
    - 110000000