# CSE 141L Milestone 3

Amy Koh, A16865992; Thanh-Nha Tran A16493988; Campbell Singhasemanon, A16017354

## 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.

Campbell Singhasemanon
Thanh-Nha Tran
Amy Koh

# 0. Team

List the names of all members of your team. Note: in this report, whenever an instruction/question is prepended with "TODO", please delete the instruction/question in your submission.

**Amy Koh**
**Campbell Singhasemanon**
**Thanh-Nha Tran**

# 1. Introduction

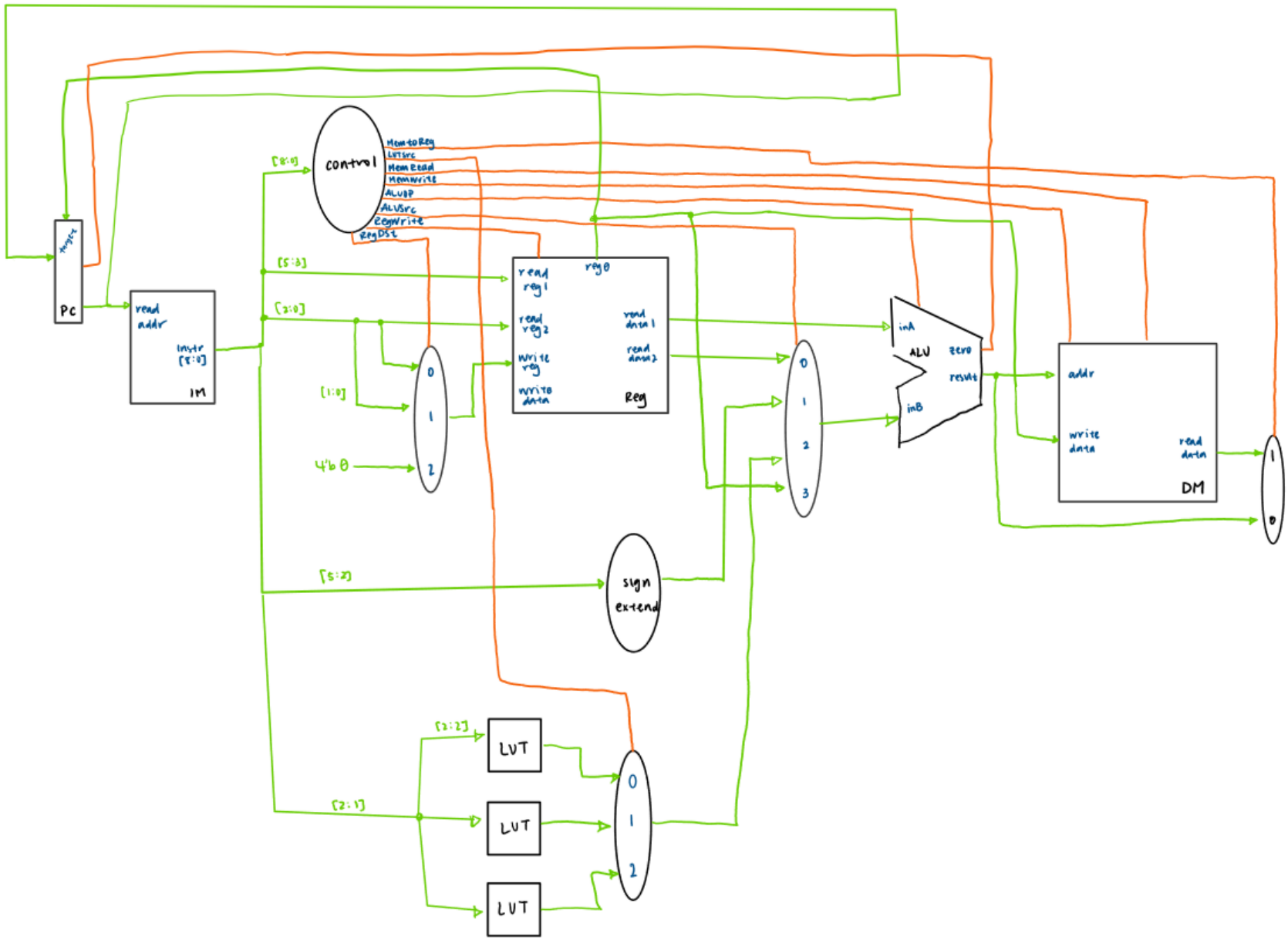
Name your architecture. What is your overall philosophy? What specific goals did you strive to achieve? Can you classify your machine in any of the standard ways (e.g., stack machine, accumulator, register-register/load-store, register-memory)? If so, which? If not, devise a name for your class of machine. Word limit: 200 words.

The name of our architecture is Small Instruction Register Processor (SIRP). We wanted to create an architect that would effectively solve the 3 given problems using the smallest amount of types of instruction as possible. We started by informally writing out pseudocode for the 3 problems and using MIPS assembly language as a basis to figure out what operations we would need. We narrowed it down to the operations we really needed and devised our architecture from there. We wanted to prioritize the use of registers over loading and storing from memory. Our team's preference to use registers to store information and help us manipulate information has caused us to create a register-memory machine. Most of our operations are done using the register, however we still utilize load/store operations in order to access and write to date memory. We also use a bit of accumulator logic. A lot of our operations automatically use R0 as its destination register. However, our "or" operation is special. It uses R0 both as a destination register as well as using R0 as one of its source registers, so it uses accumulator logic. This special case occurs because we wanted to include the "or" operation, but ran out of specialized op codes to include it as a Double instruction. As such "or" is treated as a Single instruction. (Double and Single instructions are further defined in section 3. Machine Specification).

## 2. Architectural Overview

This must be in picture form. What are the major building blocks you expect your processor to be made up of? You must have data memory in your architecture. (Example of MIPS: https://www.researchgate.net/figure/The-MIPS-architecture_fig1_251924531)

Out architecture is very similar to MIPS's architecture except that our microprocessor is single-cycle so it doesn't use data registers. Also our program counter increments by 1 byte instead of 4, so we add 1 to our program counter after each instruction.


control
MemtoReg
LUTsrc
MemRead
MemWrite
ALUOP
ALUSrc
RegWrite
RegDst
[8:0]
[5:3]
[2:0]
[1:0]
4'b 0
[5:2]
[2:2]
[2:1]
PC
read addr
instr [8:0]
IM
read reg1
read reg2
write reg
write data
reg0
read data1
read data2
Reg
sign extend
LUT
LUT
LUT
inA
ALU
inB
zero
result
addr
write data
read data
DM

# 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 |
|---|---|---|
| D | 3 bit opcode, 3 bit source reg address, 3 bit source reg address | cpy, beq, add, xor, and |
| S | 3 bit opcode, 3 bit reg address, 3 bit function | Load, store, or, ls(left/right), addi, andi |
| I | 3 bit opcode, 4 bit immediate, 2 bit destination reg address | set |

## 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 |
|---|---|---|---|---|
| cpy = copy registe r | D | 3 bit opcode (000), 3 bit source register (XXX), 3 bit destination register (XXX) | # Assume R0 has `0b0001_0001`<br><br>`cpy R0, R1` ⇔ `000_000_001`<br><br># after cpy instruction, R1 now holds `0b0001_0001` | Copies the contents of one register into another.<br><br>This operation is intended to compensate for not being able to choose a destination register. |
| store | S | 3 bit opcode (001), 3 bit source register (XXX), 3 bit function (001) | #Assume R0 holds `0b0001_0011`<br>And R1 has `0b0000_1010`<br><br>`store R1` ⇔ `001_001_001`<br><br># after load instruction mem address [10] now holds `0b0001_0011` | Stores the contents of $0 into memory at the index specified by the contents of the source register. |
| load | S | 3 bit opcode (001), 3 bit source register (XXX), 3 bit function (000) | #Assume mem address [10] holds `0b0001_0011`<br>And R1 has `0b0000_1010`<br><br>`load R1` ⇔ `001_001_000`<br><br># after load instruction R1 now holds `0b0001_0011` | Uses the contents of the register specified in the instruction to index into datamem and load the target byte into $0. |
| Or = logical or | S | 3 bit opcode (001), 3 bit source register (XXX), 3 bit function (010) | #Assume R2 holds `0b0001_0011`<br>And R0 has `0b0000_1010`<br><br>`or R2` ⇔ `001_010_010`<br><br># after or instruction R0 now holds `0b0001_1011` | Result is stored in R0 |
| andi = and | S | 3 bit opcode (001), 3 bit source register (XXX), 3 bit | #Assume R2 holds `0b0001_0011`<br>And R0 has `0b0000_1010` | The result is stored in R0. We can and by two values: binary 1 or binary 3. |

| | | | | |
|---|---|---|---|---|
| immediate | | function (X11) | `andi R2, 1` ⇔ `001_010_010`<br><br># after or instruction R0 now holds `0b0000_0001` | |
| lsl = logical shift left | S | 3 bit opcode (010), 3 bit source register (XXX), 3 bit function (X00) | #Assume R2 holds `0b0001_0011`<br><br>`lsl R2, 4` ⇔ `010_010_010`<br><br>#after lsl instruction, R0 now holds `0b0011_0000` | The byte can be shifted by 4 bits or 1 bit, result is stored in R0 |
| lsr = logical shift right | S | 3 bit opcode (010), 3 bit source register (XXX), 3 bit function (X10) | #Assume R2 holds `0b0001_0011`<br><br>`lsr R2, 4` ⇔ `010_010_011`<br><br>#after lsr instruction, R0 now holds `0b0000_0001` | The byte can be shifted by 4 bits or 1 bit only, result is stored in R0 |
| addi = add immediate | S | 3 bit opcode (010), 3 bit source register (XXX), 3 bit function (XX1) | #Assume R2 holds `0b0001_0011`<br><br>`addi R2, 15` ⇔ `010_010_110`<br><br>#after addi instruction, R0 now holds `0b0010_0010` | The value in the register can be incremented by 1, 5, 15, or -1 only |
| beq | D | 3 bit opcode (011), 3 bit register address (XXX), 3 bit register address (XXX) | #Assume R1 holds `0b0000_0011`, R2 holds `0b0000_0011`, R0 holds `0b0001_0000`<br><br>`beq R1, R2` ⇔ `011_001_010`<br><br>#after beq instruction, the program counter is now `0b0001_0000` | The branch address is stored in R0 |

| | | | | |
|---|---|---|---|---|
| add | D | 3 bit opcode (100), 3 bit register address (XXX), 3 bit register address (XXX) | #Assume R1 holds `0b0000_0010`, R2 holds `0b0000_0001`.<br><br>`add R1, R2` ⇔ `0000_00011`<br><br>After add R0 holds `0b0000_0011` | The value of R1 + R2 is stored in R0. |
| xor | D | 3 bit opcode (101), 3 bit register address (XXX), 3 bit register address (XXX) | #Assume R1 holds `0b0000_0011`, R2 holds `0b0000_1001`<br><br>`xor R1, R2` ⇔ `101_001_010`<br><br>#after xor instruction, R0 is now `0b0000_1010` | Result is stored in R0 |
| and | D | 3 bit opcode (110), 3 bit register address (XXX), 3 bit register address (XXX) | #Assume R1 holds `0b0000_0011`, R2 holds `0b0000_1001`<br><br>`and R1, R2` ⇔ `110_001_010`<br><br>#after xor instruction, R0 is now `0b0000_0001` | Result is stored in R0 |
| set = set LSB | I | 3 bit opcode, 4 bit immediate(#XXXX), 2 bit destination reg address (XX) | #Assume R2 holds `0b0100_0111`,<br><br>`set 1000, R2` ⇔ `111_1000_10`<br><br>#after xor instruction, R2 is now `0b0100_1000` | Sets the 4 LSB of a register to the specified 4'b immediate value. It is meant to be used alongside our 4-bit left shift to set the value of a whole register. |

## Internal Operands

How many registers are supported? Is there anything special about any of the registers (e.g. constant, accumulator), or all of them general purpose?

We have 8 registers. Some of our registers interact with our instructions in unique ways. Most of the instructions use $0 as the destination register implicity. Our branch equal instruction also uses the contents of $0 as the relative address. Aside from this, there is nothing particularly special about the registers. None of them hold constant values.

## Control Flow (branches)

What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported? How do you accommodate large jumps?

We only support relative branching. The target address is specified by the programmer and should be stored in register 0. Then the programmer can use the “beq” instruction to decide if they would like to branch to another line. If beq is 1, then it will branch to PC + target (from r0). Otherwise, it will keep going (PC+1). If the programmer always wants to branch, then they still have to use the beq instruction. They can just use arbitrary registers to ensure it will always branch (ex: r1=r1). Branches distances can range from [-128,127]. Long jumps must consist of branch statements chained together at maximum distance apart.

## Addressing Modes

What memory addressing modes are supported, e.g. direct, indirect? How are addresses calculated? Give examples.

We use direct addressing. The address must be loaded into a register and the register specified in a load or store instruction. Addresses are interpreted as 8’b unsigned binary numbers. Our memory contains 256 entries.

Ex. set 1000 $3
load $3

In the above example, We first set the 4 LSB of $3 to 1000 so it now contains ‘00001000’. This is equal to 8 in binary, so datamem[8] is loaded into $0.

# 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.

A programmer using our machine must base their program around the register 0. Many of our operations used r0 as an implicit destination register. For example, our *add* instruction only allows for two registers to be specified: a source and secondary source. The output of source + secondary source will automatically be put into register 0, overwriting register 0 contents. Contents can be moved in and out of r0 with ease with the use of *cpy*. *Cpy* will move the contents into other registers as storage that the programmer uses to perform further calculations.This design choice was created to maximize the amount of registers we could use while still keeping our instruction format relatively easy to use. Our machine prioritizes the use of registers to limit the amount of load and store operations that need to be done. Load and store operations still exist in our machine. However, they too are dependent on the 0 register. *Load* still take a register containing an address to memory as a parameter, read from memory, and load those contents into r0. On the other hand, *store* will store the contents of whatever is in r0 into the memory address in the given register.

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 can not copy the instructions from MIPS. In our machine, we are limited to a 9 bit instruction, whereas MIPS has a 32 bit instruction. This drastically changed how we went about our current design. We reduced the number of registers in our machine from 16 to 8, and got rid of the need to specify a destination register. This reduced the number of bits we needed to dedicate to specifying registers. We also got rid of strict formatting types to allow for more flexibility in our instruction design.

4.3 Will your ALU be used for non-arithmetic instructions (e.g., MIPS or ARM-like memory address pointer calculations, PC relative branch computations, etc.)? If so, how does that complicate your design?
Our ALU will not be used for non-arithmetic instructions. We do not have instructions for memory address pointer calculations and use absolute branching instead of relative. As such our ALU is a simple design.

# 5. Individual Component Specification

## Top Level

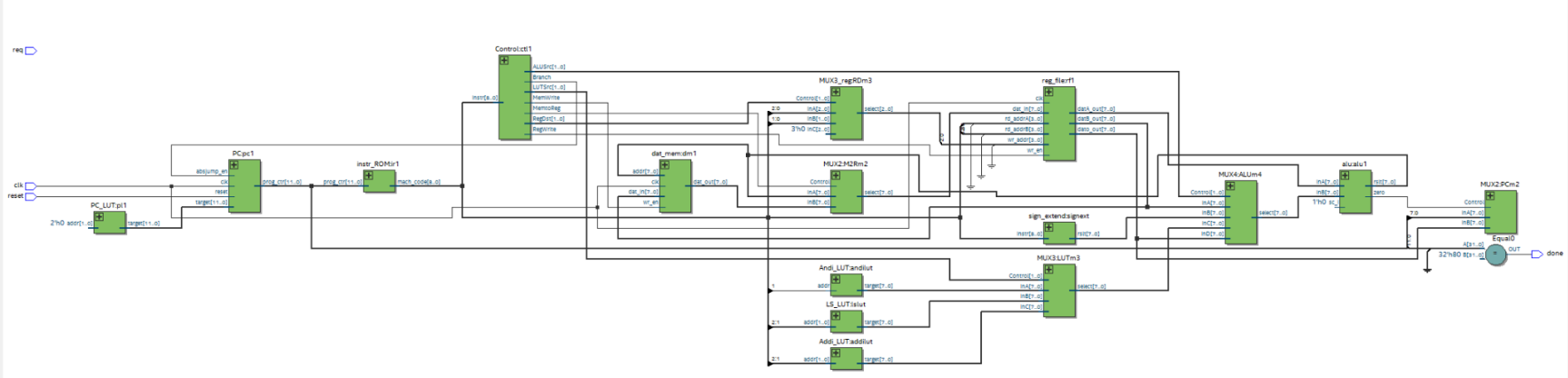
Module file name: top_level.sv

### Functionality Description

This module connects all the other modules together with wires.

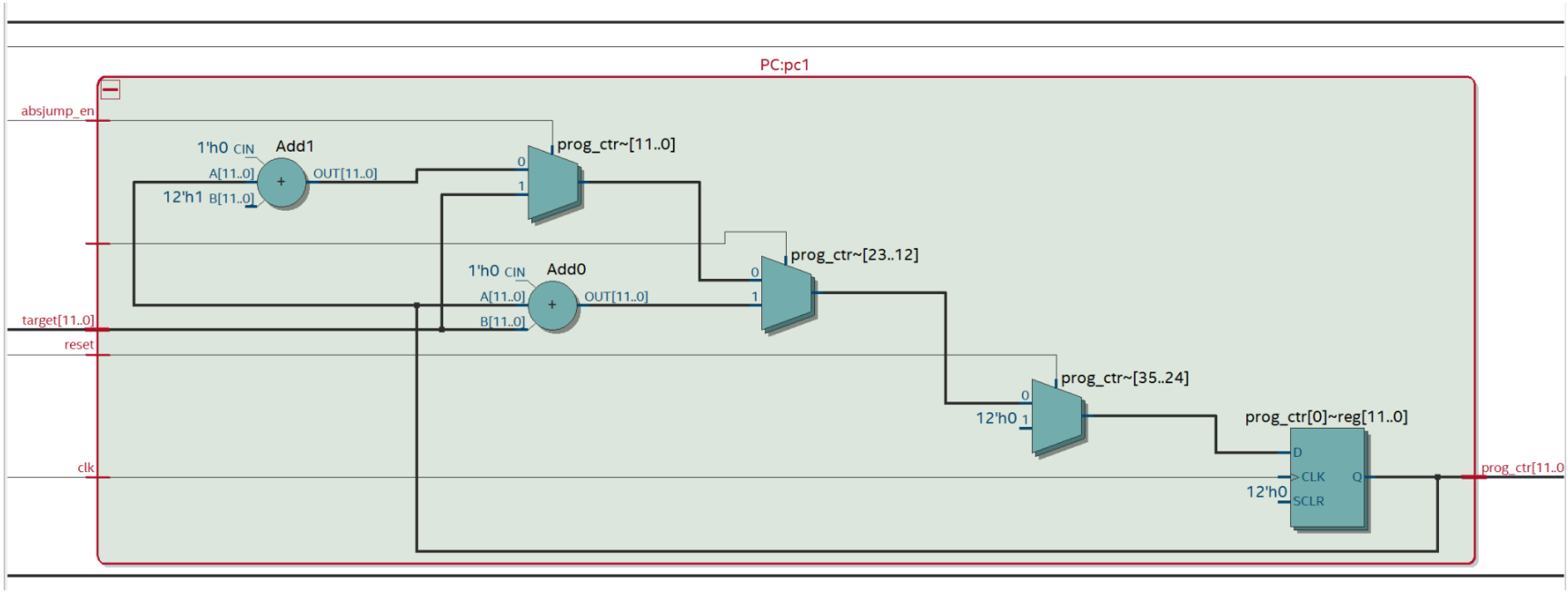
### Schematic

Show us your schematic for the top level.




## Program Counter

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




## Functionality Description

The PC module increments our program counter by 1 if we are moving to the next instruction. If we are jumping, then it will jump to the correct address.

## (Optional) Testbench Description

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

## Schematic

TODO Show us your schematic for the fetch unit.

### (Optional) Timing Diagram

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

## Instruction Memory

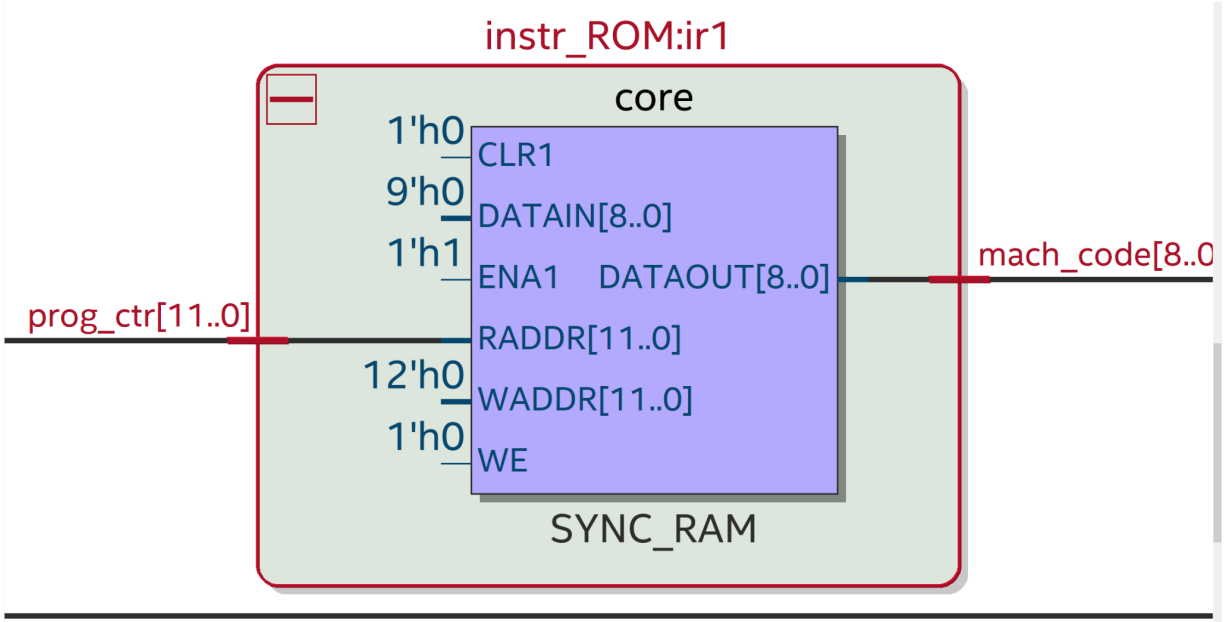
Module file name: instr_ROM.sv

### Functionality Description

This is our instruction module that reads in our instructions from “mach_code.txt”. It will store the instructions in an array. We will use this array to access our instructions one at a time.

### Schematic

Show us your schematic for the fetch unit.



# Control Decoder

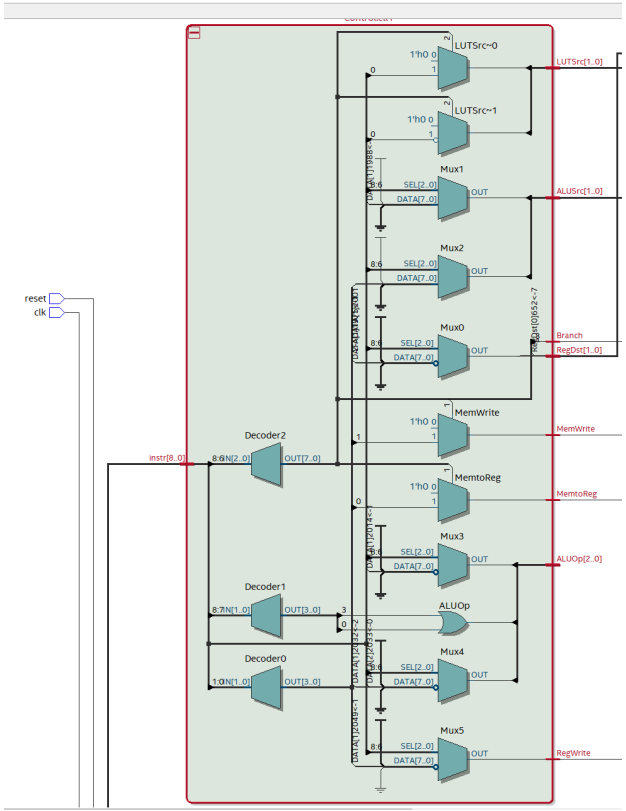
Module file name: Control.sv

## Functionality Description

This module is our control unit. It used the instruction to figure out what control operations needed to be turned on (using our op codes). We have the following controls that need to be initialized: RegDst, Branch, MemWrite, ALUSrc, RegWrite, MemtoReg, ALUOp, LUTSrc.

## Schematic

Show us your schematic for the control decoder.




# Register File

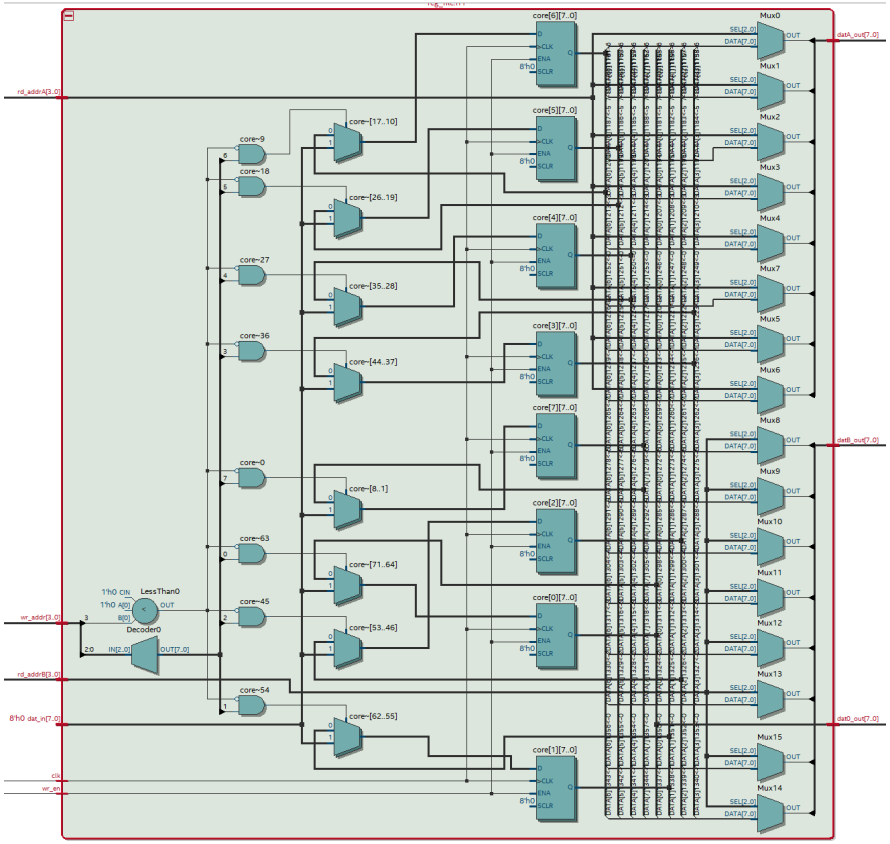
Module file name: reg_file.sv

Functionality Description

This is our register module. It can read 3 registers: 2 can be specified and 1 is the zero register. It will output the data within the register. It can also write to a register if RegWrite is turned on.

## Schematic

Show us your schematic for the register file.




# ALU (Arithmetic Logic Unit)

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

## Functionality Description

This is our ALU, which handles all our computation work.

(Optional) Testbench Description
Describe your testbench. How does it work? What test cases does it test?

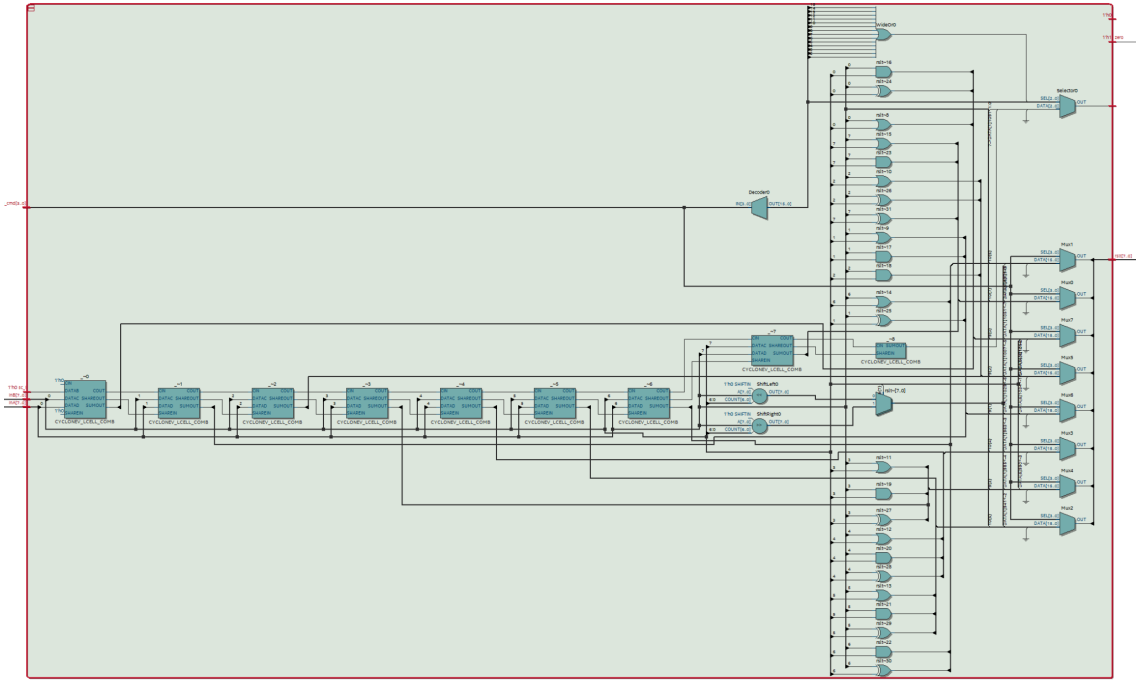
## ALU Operations

What ALU operations will you be demonstrating? What instructions are they relevant to?
We have the following operations:

- Add - used for add and addi instructions
- logical shift - used for logical shift left and right instructions
- bitwise or - used for or instructions
- bitwise and - used for and and andi instructions
- bitwise xor - used for xor instructions
- beq - used for beq instructions
- set - used for set instructions
- pass - used for load, store, and cpy instructions

## Schematic

Show us your schematic for the register file.



## (Optional) Timing Diagram

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

# Data Memory

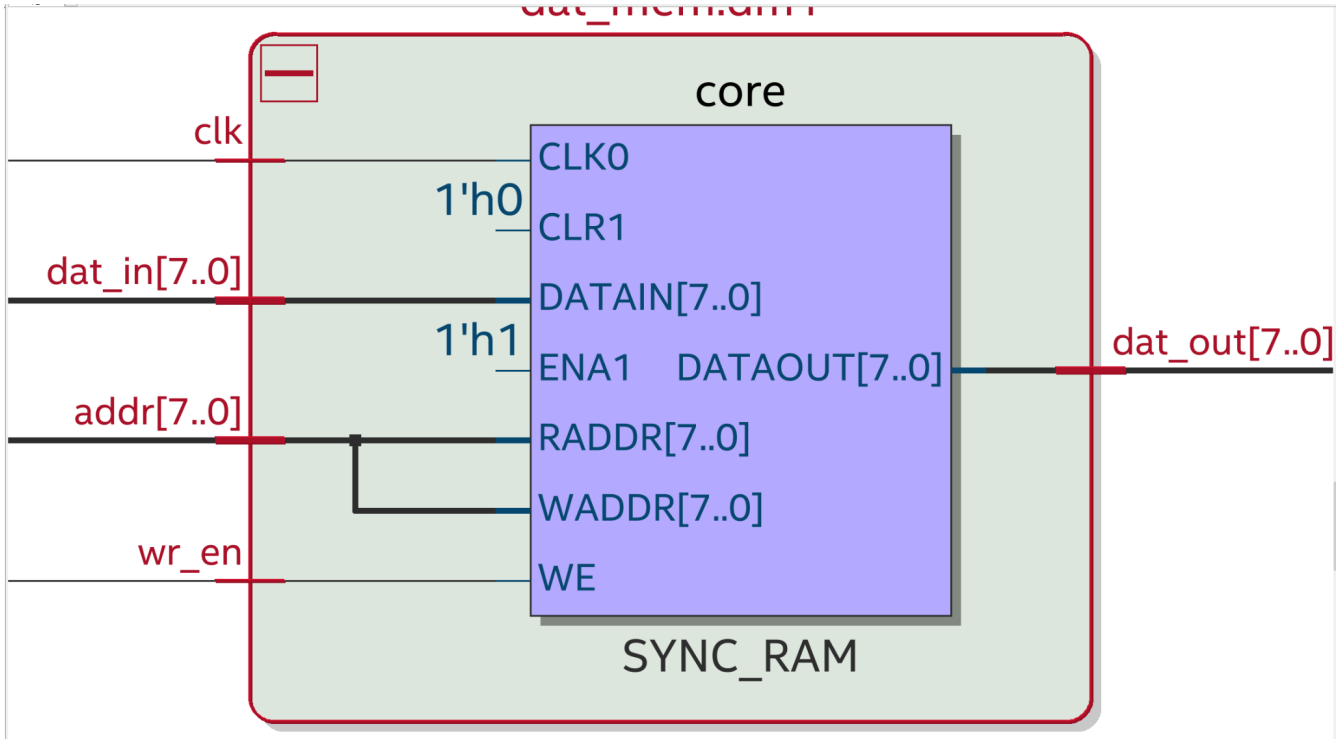
Module file name: dat_mem.sv

## Functionality Description

This module stores data into memory or accesses data from memory to be written into a register.

### Schematic

Show us your schematic for the data memory.




## Lookup Tables

Module file name: Addi_LUT.sv, LS_LUT.sv, Andi_LUT.sv

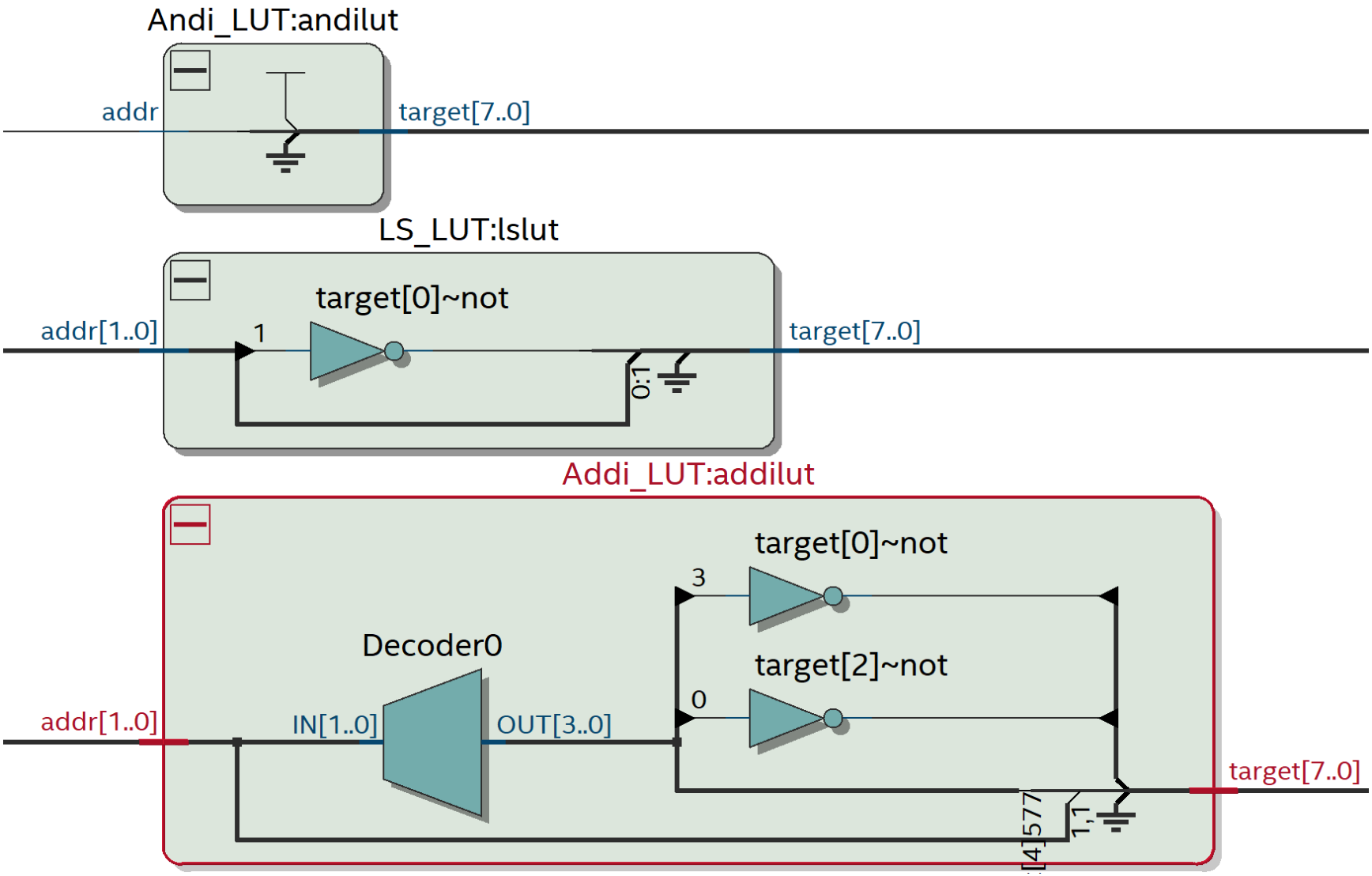
### Functionality Description

These 3 three look up tables correspond to the addi, logical shift, and andi operations.
Addi has a lookup table to determine if we are adding by 1, 5, 15, or 30.
Logical shift has a lookup table to determine if we are shifting left by 1, shifting left by 4, shifting right by 1, or shifting right by 4.

Andi has a lookup table to determine if we are anding by binary 1 or binary 3.

## Schematic

Show us your schematic(s) for the lookup table(s).


Andi_LUT:andilut
addr
target[7..0]
LS_LUT:lslut
target[0]~not
addr[1..0]
1
0:1
target[7..0]
Addi_LUT:addilut
target[0]~not
3
Decoder0
target[2]~not
0
addr[1..0]
IN[1..0]
OUT[3..0]
target[7..0]
1,1

# Muxes (Multiplexers)

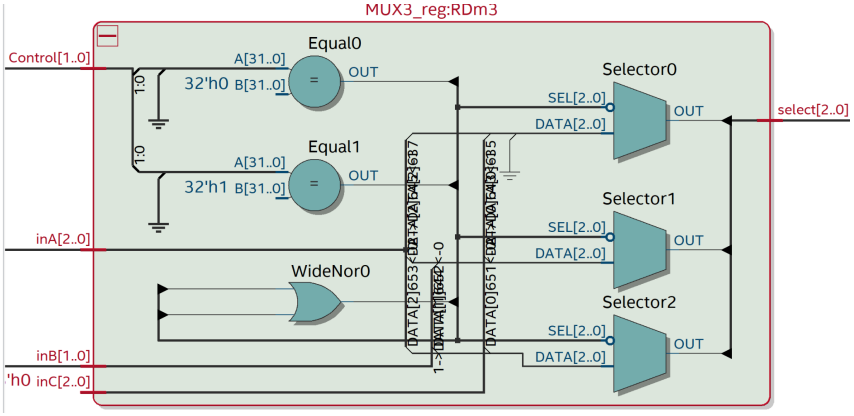
Module file name: MUX2.sv, MUX3.sv, MUX3_reg, MUX4.sv

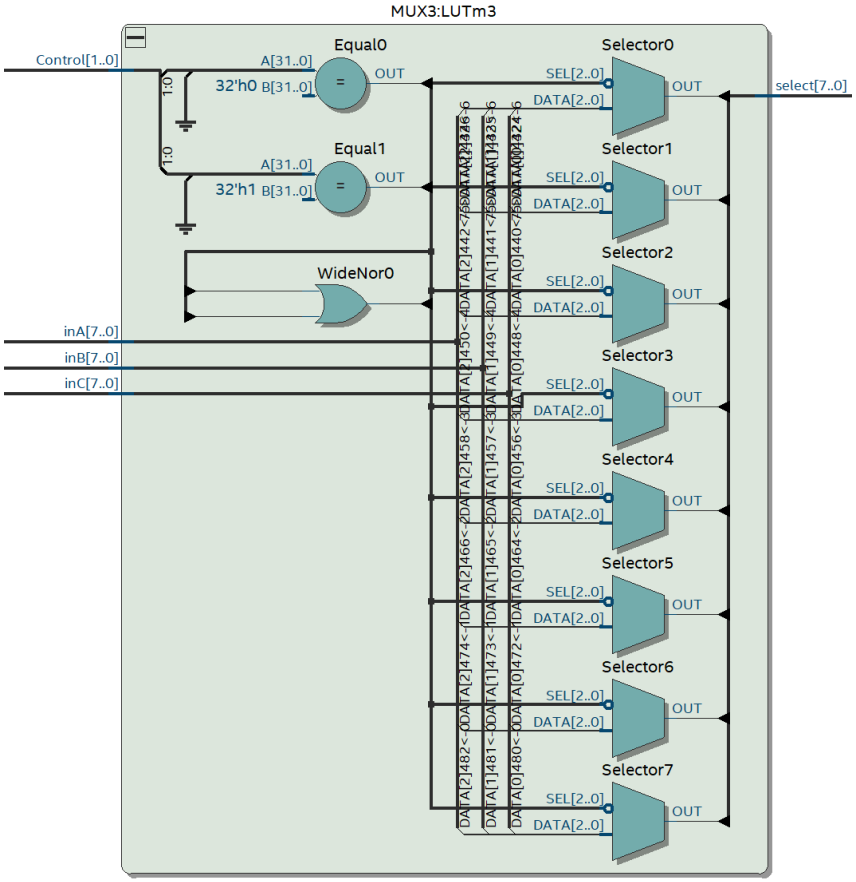
## Functionality Description

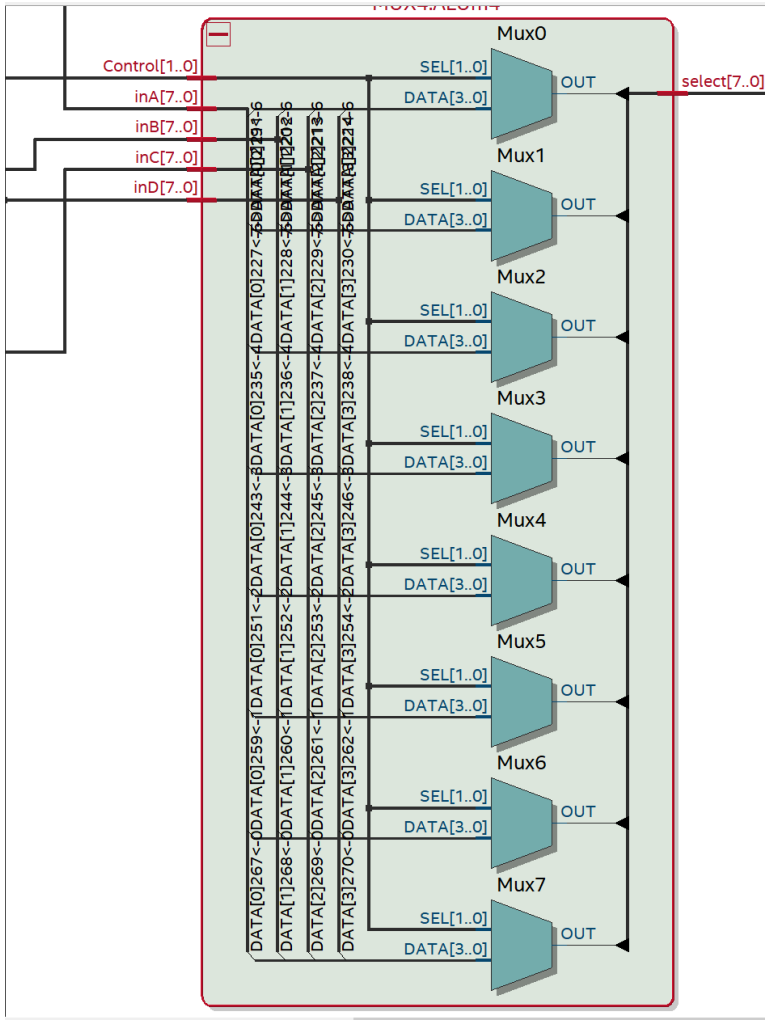
We have 4 different types of muxes to help us decide which output we need to input for certain modules.
MUX2 toggles between two options. We use this for PC logic and memory to register logic.
MUX3 toggles between three options. We use it to decide which of the 3 lookup table outputs we need.
MUX3_reg also toggles between three options, but accounts for the inputs being different lengths. Whereas MUX3 is expecting data to be 8 bits. We use this for register logic.
MUX4 toggles between four options. We use this ALU logic.

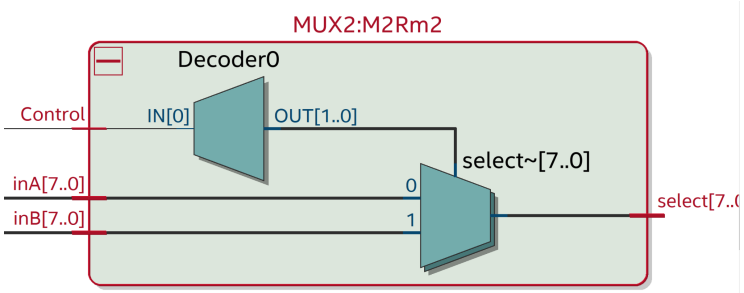
## Schematic

Show us your schematic for your mux(es).




MUX3:LUTm3
Control[1..0]
Equal0
A[31..0]
32'h0 B[31..0]
OUT
Equal1
A[31..0]
32'h1 B[31..0]
OUT
WideNor0
inA[7..0]
inB[7..0]
inC[7..0]
Selector0
Selector1
Selector2
Selector3
Selector4
Selector5
Selector6
Selector7
SEL[2..0]
DATA[2..0]
OUT
select[7..0]


Control[1..0]
inA[7..0]
inB[7..0]
inC[7..0]
inD[7..0]
Mux0
Mux1
Mux2
Mux3
Mux4
Mux5
Mux6
Mux7
SEL[1..0]
DATA[3..0]
OUT
select[7..0]




## Other Modules (if necessary)

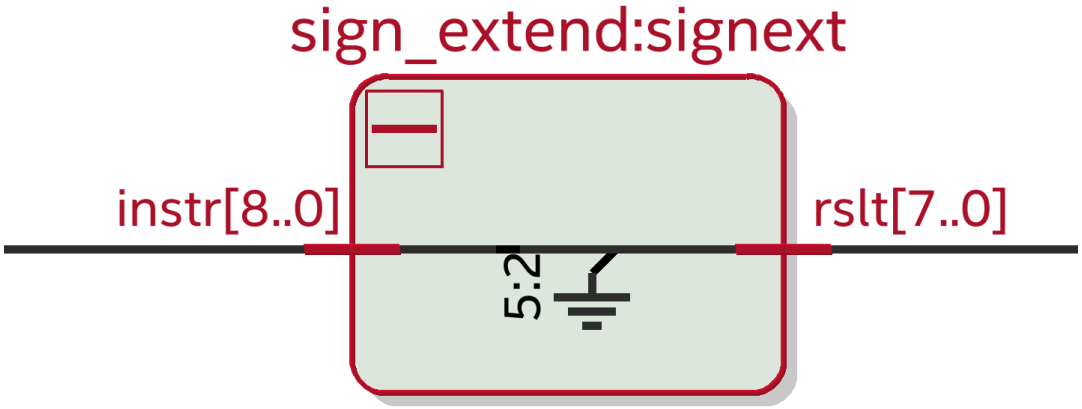
Module file name: sign_extend.sv

### Functionality Description

This module sign extends a 4 bit input to become 8 bits. This is used for our set operation.

### Schematic

Show us your schematic for your module.



# 6. Program Implementation

An example Pseudocode and Assembly Code has been filled out for you. When you submit, **please delete the example along with this paragraph.**

## Example Pseudocode

```
# function that performs division
mul_inverse(operand):
  divisor = operand
  dividend = 1
  result = 0
  counter = 0
  while counter != 16:
    if dividend > divisor:
      dividend -= divisor
      result = (result << 1) || 1
    else:
      result = (result << 1)
    dividend <<= 1
    counter += 1
  return result
```

## Example Assembly Code

```
# Do not try to understand this code. It is bogus code, but a good example of what to submit.

# loading divisor
load R0, %0010        # 0010 = location of the divisor in memory
load R1, %0100        # 0100 = location of the dividend in memory

add R0, R1, R2        # R0 + R1 => R2 adding the divisor and the dividend together
```

```
...
# more assembly code
...

# note that this may be several pages long. The teaching staff will not be verifying
correctness of your assembly code for Milestone 1.
```

## Program 1 Pseudocode

```
transmitter(byte[] memdata[0:59]){
        for (int i = 0; i < 15: i++) {                          //0   1   2   3   4    5    6   7
                int message1 = memdata[i*2+1];                  //0   0   0   0   0   b11 b10 b9
                int message0 = memdata[i*2];                    //b8 b7 b6 b5 b4 b3   b2   b1
                int parity8 = ^(message1(5, 7), message0(0, 3)); //range is inclusive
                int parity4 = ^(message0(4, 6), message0(0), message1(5, 7)); //b2, 3, 4, 8, 9, 10, 11
                int parity2 = ^(message0(4, 5), message0(7), message0(1, 2), message1(5, 6)); //b1, 3, 4, 6, 7, 10, 11
                int parity1 = ^(message0(6, 7), message0(3, 4), message0(1), message1(7), message1(5)); //b1, 2, 4, 5, 7, 9, 11
                int parity0 = ^(message1(5, 7), message0, parity1, parity2, parity4, parity8);

                int ret31 = message1(5,7) + message0(0, 3) + parity8; //we really mean concat not add
                int ret30 = message0(4, 6) + parity4 + message0(7) + parity2 + parity1 + parity0;

                memdata[i*2+31] = ret31;
                memdata[i*2+30] = ret30;
```

# Program 1 Assembly Code

```
addi $0, #15
addi $0, #15            // # iterations -> 30
cpy $0, $8              //r8 = 30
set #0001, $0
lsl $0, #4
set #1100, $0
or $1                   //set branch destination 268 or 1 0000 1100 -> end of loop/program
begin: beq $8, $7       //$7 = 0
// loop contents
load $2                 //$2 = mem[0]
addi $8, #1             //increment counter by 1
cpy $0, $8
load $3                 //$3 = mem[1]
addi $8, #1
cpy $0, $8              //increment counter by 1 again for next loop


// parity8 -> $14
lsr $2, #4              //$0 = message0(0,3)
xor $3, $0              //$0 = 4 bits xor result
cpy $0, $4              //$4 = 4 bits xor result
lsr $4, #1              //$0 = 2 bits to xor
lsr $0, #1
cpy $0, $1              //$1 = 2 bits to xor
andi $4, #3             //$0 = 2 bits to xor
xor $0, $1              //$0 = 2 bits xor result
cpy $0, $4              //$4 = 2 bit xor result
lsr $4, 1               //$0 = 1 bit to xor
cpy $0, $2              //$2 = 1 bit to xor
andi $4, #1             //$0 = 1 bit to xor
```

```
xor $0, $2                  //$0 = p8 final result
cpy $0, $4                  //$4 = p8
set #0100, $1               //$1 = 00000100
lsl $1, #4                  //$0 = 01000000
cpy $0, $1                  //$1 = 01000000
set 0001, $1                //$1 = &mem[65] = 0100 0001
cpy $4, $0                  //$0 = $4
store $1                    //mem[65] = p8


// parity4 ->$13
lsl $2, #4                  //remove message0(0, 3)
lsr $0, #4                  //message0(4,6) by removing message0(7)
lsr $0, #1                  //$0 = message0(4,6)
cpy $0, $6                  //$6 = message0(4,6)
lsr $3, #4                  //lsr $0, #1
lsr $0, #1
lsr $0, #1
lsr $0, #1                  //shift right by 7
xor $6, $0                  //$0 = message0(4,6) ^ message(0) 3 bit xor result
xor $0, $2                  //message1 ^ (message0(4,6) ^ message(0))= 0 0 0 0 0 c1 c2 c3
cpy $0, $6
lsr $6, #2                  // isolates c1
cpy $0, $1
andi $6, #3                 //isolates c2 c3
xor $0, $1                  //(c1, c2)^(c3) = 0 0 0 0 0 0 C1 C2
cpy $0, $6
lsr $6, #1                  //isolate C1
cpy $0, $1
andi $6, #1                 //isolate C2
xor $1, $0          // xored
cpy $0, $5
```

```
set 0100, $1                    //$1 = 00000100
lsl $1, #4                      //$0 = 01000000
cpy $0, $1                      //$1 = 01000000
set 0010, $1                    //$1 = &mem[66] = 0100 0010
cpy $5, $0                      //$0 = $5
store $1                        //mem[66] = p4


// parity2 ->$12
lsl $2, #4
lsr $0, #4
lsr $0, #1
lsr $0, #1
cpy $0, $4                  //$4 = message0(4,5)
lsl  $2, #4
lsl $0, #1
lsl $0, #1
lsr $0, #4
lsr $0, #1
lsr $0, #1
lsr $0, #1              //message0(7)
xor $0, $4
cpy $0, $4              //$r4 = message0(4,5) ^ message0(7)
lsl $2, #1
lsr $0,  #4
lsr $0, #1
lsr $0, #1              //$0 = message0(1,2)
xor $0, $4
cpy $0, $4              //$4 = message0(4,5) ^ message0(7) ^ message0(1,2)
lsl $3, #4
lsl $0, #1
```

```
lsr $0, #4
lsr $0, #1
lsr $0, #1          //$4 = message1(5,6)
xor $0, $4
cpy $0, $4          //$r4 = message0(4,5) ^ message0(7) ^ message0(1,2) ^ message1(5,6) = 0 0 0 0 0 0 c1 c2
lsr $4, #1          //isolates c1
cpy $0, $5          //$r5 = c1
andi $4, #1         //isolate c2
xor $0, $5
cpy $0, $4          //$4 = parity2

set 0100, $1            //$1 = 00000100
lsl $1, #4              //$0 = 01000000
cpy $0, $1              //$1 = 01000000
set #0011, $1           //$1 = &mem[66] = 0100 0011
cpy $4, $0              //$0 = $4
store $1                //mem[67] = p2


// parity1 -> $11
andi $2, 3          //$0 = message0(6,7)
cpy $0, $6          //$6 = message0(6,7)
lsl $2, #1
lsl $0, #1
lsl $0, #1
lsr $0, #4
lsr $0, #1
lsr $0, #1          //$0=message0(3,4)
xor $6, $0          //message0(3,4) ^ message(6,7) = 0 0 0 0 0 0 c1 c2
cpy $0, $6
lsr $6, #1          //isolate c1
cpy $0, $1
```

```
andi $0, #1             //isolate c2
xor $0, $1
cpy $0, $1
lsl $2, #1
lsr $0, #4              //message0(1)
lsr $0, #1
lsr $0, #1
lsr $0, #1
xor $1, $0
cpy $0, $1
andi $3, #1             //message1(7)
xor $0, $1
cpy $0, $1
lsr $3, #1              //message1(5)
lsr $0, #1
xor $0, $1
cpy $0, $4
set #0100, $1                   //$1 = 00000100
lsl $1, #4                      //$0 = 01000000
cpy $0, $1                      //$1 = 01000000
set #0100, $1                   //$1 = &mem[68] = 0100 1000
cpy $4, $0                      //$0 = $3
store $1                        //mem[68] = p1


//parity0 -> $10
set #0100, $1                   //$1 = 00000100
lsl $1, #4                      //$0 = 01000000
cpy $0, $1                      //$1 = 01000000
set #0001, $1                   //$1 = &mem[65] = 0100 0001
load $1
cpy $1, $4                      //$4 = parity8
```

```
set #0100, $1          //$1 = 00000100
lsl $1, #4             //$0 = 01000000
cpy $0, $1             //$1 = 01000000
set #0010, $1          //$1 = &mem[66] = 0100 0001
load $1
cpy $1, $5             //$5 = parity4

xor $4, $5
cpy $0, $4             //$4 = p8^p4


set #0100, $1          //$1 = 00000100
lsl $1, #4             //$0 = 01000000
cpy $0, $1             //$1 = 01000000
set #0011, $1          //$1 = &mem[67] = 0100 0011
load $1
cpy $1, $5             //$5 = p2

xor $4, $5
cpy $0, $4             //$4 = p8^p4^p2

set #0100, $1          //$1 = 00000100
lsl $1, #4             //$0 = 01000000
cpy $0, $1             //$1 = 01000000
set #0100, $1          //$1 = &mem[68] = 0100 0100
load $1
cpy $1, $5             //$5 = parity1

xor $4, $5
cpy $0, $4             //$4 = p8^p4^p2^p1 = p0
```

```
xor $2, $3          //xor message0 and message1
cpy $0, $5
lsr $0, #4          //upper 4 bits
lsl $0, #4
cpy $0, $6
lsl $5, #4
lsr $0, #4          //lower 4 bits
xor $0, $6          //4 bits left  0000xxxx
cpy $0, $5
lsl $5, #4
lsl $0, #1
lsl $0, #1
lsr $0, #4
lsr $0, #1
lsr $0, #1
cpy $0, $6           //upper 2 bits
lsr $5, #4
lsr $0 #1
lsr $0, #1          //lower 2 bits
xor $0, $6          //2 bits lefts  000000xx
cpy $0, $5
lsr $0, #1
cpy $0, $5          //upper bit
lsl $6, #4
lsl $0, #1
lsl $0, #1
lsr $0, #4
lsr $0, #1
lsr $0, #1          //lower bit
xor $0, $6          //1 bit left
```

```
xor $0, $4                  //p0



//ret31  = msw
lsl $3, #1
cpy $0, $5                  //$5 = b11 b10 b9 b8 b7 b6 b5
set #0100, $1                       //$1 = 00000100
lsl $1, #4                          //$0 = 01000000
cpy $0, $1                          //$1 = 01000000
set #0001, $1                       //$1 = &mem[65] = 0100 0001
load $1
or $1
cpy $0, $5                          //$5 = ret31

//ret30  = lsw
lsr $2, #1        //get rid of b1
lsl $0, #4
lsl $0, #1
cpy $0, $5        //$6 = message0(2,4)

set 0100, $1                        //$1 = 00000100
lsl $1, #4                          //$0 = 01000000
cpy $0, $1                          //$1 = 01000000
set #0010, $1                       //$1 = &mem[66] = 0100 0010
load $1                             //$1 = p4

lsl $1, #4        //get P4 in position 5
cpy $5, $0
or $1
cpy $0, $5        //$5 = message0(2,4) + P4
lsr $2, #4
```

```
lsr $0, #1
lsr $0, #1
lsl $0, #4          //get b1 in position 4
cpy $0, $1
cpy$6, $0
or $6
cpy $0, $6      //$6 = message0(2,4) + P4 + b1

set #0100, $1                   //$1 = 00000100
lsl $1, #4                      //$0 = 01000000
cpy $0, $1                      //$1 = 01000000
set 0011, $1                    //$1 = &mem[67] = 0100 0011
load $1                         //$1 = p2

lsl $0, #1
lsl $0, #1          //get P2 in position
cpy $0, $1
cpy $6, $0
or $1
cpy $0, $6       //$6 = message0(2,4) + P4 + b1 + P2
cpy $4, $0
or $6
cpy $0, $6    //$6 = message0(2,4) + P4 + b1 + P2 + P1 + P0


Cpy $8, $1
addi $1, #30
Cpy $5, $0
Store, $1

Addi $1, #1
Cpy $4, $0
```

```
Store, $1

addi $8, #-1
copy $0, $8     //decrement counter

Set 0000, $0
Lsl $0, #4
Set 0101, $0            //set $0 to address 5
Beq $1, $1
end:
```

# Program 2 Pseudocode

```
receiver(byte[] memdata[0:59]) {
        for (int i = 0; i < 15: i++) {                          //  0     1   2   3   4   5    6   7
                int message1 = memdata[i*2+31];                 //b11 b10 b9 b8 b7 b6 b5 p8
                int message0 = memdata[i*2 + 30];               //b4   b3   b2 p4 b1 p2 p1 p0

                int parity8 = message1(7);
                int s_parity8 = ^(message1(0, 6)); //range is inclusive
                int parity4 = message0(3); //b2, 3, 4, 8, 9, 10, 11
                int s_parity4 = ^(message0(0, 2), message1(0, 3));
                int parity2 = message0(5); //b1, 3, 4, 6, 7, 10, 11
                int s_parity2 = ^(message0(4), message0(0, 1), message1(4, 5), message1(0, 1));
                int parity1 = message0(6);
                int s_parity1 = ^(message0(4), message0(2), message0(0), message1(6), message1(4), message1(2), message1(0));
                int parity0 = message0(0);
                int s_parity0 = ^(message0(1, 7), message1(0,7);

                int error_pointer;
                error_pointer[0] = parity8 ^ s_parity8;
                error_pointer[1] = parity4 ^ s_parity4;
                error_pointer[2] = parity2 ^ s_parity2;
                error_pointer[3] = parity1 ^ s_parity1;

                2’b flag = 0;
                if (parity0)
                        flag = 01
                else if(error_pointer == 4’b0)
                        flag = 00
                else
                        flag = 1X
```

```
int index = error_pointer % 8
if(error_pointer > 7){
        Message1[error_pointer] = !message1[error_pointer];
} else {
        Message0[error_pointer] = !message0[error_pointer];
}


int ret[31] = flag[1] + flag[0] + 0 + 0+ 0 + message1(0,2);
int ret[30] = message1(3, 7) + message0(0, 2) + message0(4);
```

## Program 2 Assembly Code

```
set #0100, $0
lsl $0, #4
set #0000, $0           // $0 = 65
cpy $0, $6              // $6 = 65
addi $3, #30            // $0 = 30 = max index + 1
store $6                // datamem[65] = max index

// Loop:
set #0100, $0
lsl $0, #4
set 0001, $0            // $0 = 65
cpy $0, $6              // $6 = 65
load $6                 // load max index into $0
beq $0, $6, end

set #0100, $0
lsl $0, #4
```

```
set #0001, $0          // $0 = 65
cpy $0, $6             // $6 = 65
addi $3, #30           // $0 = 30 = max index + 1
store $6               // datamem[65] = max index

addi $7, #30
load $7                // $0 = message0
cpy $0, $5             // $5 = message0
addi $7, #1
cpy $0, $7             // increment current index
addi $7, #30
load $0                // $0 = message1
cpy $0, $6             // $6 = message1


lsl $6, #4
lsr $0, #4
lsr $0, #1       // $0 = message1(4,6)
cpy $0, $1       // $1 = message1(4,6)
lsr $6, #4       // $0 = message1(0,3)
xor $0 $1       // $0 = 4 bit xor res

cpy $0, $1       // $1 = 4 bit xor res
lsr $0, #1
lsr $0, #1       // $0 = 2 bit to xor
cpy $0, $2       // $2 = 2 bit to xor
andi $1, #3      // $0 = 2 bit to xor
xor $0, $2       // $0 = 2 bit xor result
cpy $0, $1       // $1 = 2 bit xor result
lsr $0, #1       // $0 = 1 bit to xor
cpy $0, $1       // $2 = 1 bit to xor
andi $1, #1      // $0 = 1 bit to xor
```

```
xor $0, $2      // $0 = s_p8
lsl $0, #1      // $0 = 000000 s_p8 0
cpy $0, $4      // $4 = 000000 s_p8 0

lsr $5, #4
lsr $0, #1      // $0 = m0(0,2)
cpy $0, $1      // $1 = m0(0,2)
lsr $6, #4      // $0 = m1(0,3)
xor $0, $1      // $0 = 4 bit xor result

cpy $0, $1      // $1 = 4 bit xor res
lsr $0, #1
lsr $0, #1      // $0 = 2 bit to xor
cpy $0, $2      // $2 = 2 bit to xor
andi $1, #3     // $0 = 2 bit to xor
xor $0, $2      // $0 = 2 bit xor result
cpy $0, $1      // $1 = 2 bit xor result
lsr $0, #1      // $0 = 1 bit to xor
cpy $0, $1      // $2 = 1 bit to xor
andi $1, #1     // $0 = 1 bit to xor
xor $0, $2      // $0 = s_p8
add $0, $4      // $0 = 000000 s_p8 s_p4
lsl $0, #1      // $0 = 00000 s_p8 s_p4 0
cpy $0, $4      // $4 = 00000 s_p8 s_p4 0


lsl $5, #4
lsl $0, #1
lsl $0, #1      // $0 = message0(0,1)
cpy $0, $1      // $1 = message0(0,1)
lsl $5, #1
lsl $0, #1
```

```
lsl $0, #1
andi $0, #1     // $0 = message0(4)
xor $0, $1      // $0 = 2 bit xor result from message0
cpy $0, $1      // $1 = 2 bit xor result from message0

lsr $6, #4
lsr $0, #1
lsr $0, #1      // $0 = message1(0,1)
xor $0, $1      // $0 = 2 bit xor result missing m1(4,5)
cpy $0, $1      // $1 = 2 bit xor result missing m1(4,5)

lsl $6, #4
lsr $0, #4
lsr $0, #1
lsr $0, #1      // $0 = message1(4,5)
xor $0, $1      // 2 bit xor result
cpy $0, $1      // $1 = 2 bit xor result
lsr $0, #1      // 1 bit to xor
cpy $0, $2      // $2 = 1 bit to xor
andi $1, #1     // $0 = 1 bit to xor
xor $0, $2      // $0 = s_p2
add $0, $4      // $0 = 00000 s_p8 s_p4 s_p2
lsl $0, $1      // $0 = 0000 s_p8 s_p4 s_p2 0
cpy $0, $4      // $4 = 0000 s_p8 s_p4 s_p2 0
andi $5, #1     // $0 = message0(0)
cpy $0, $1      // $1 = message0(0)
lsr $5, #1
lsr $0, #1
andi $0, #1     // $0 = message0(2)
xor $0, $1      // $0 = ^(m0(0), m0(2))
cpy $0, $1      // $1 = ^(m0(0), m0(2))
lsr $5, #4
```

```
andi $0, #1     // $0 = message0(4)
xor $0, $1      // $0 =  ^(m0(0), m0(2), m0(4))
cpy $0, $1      // $1 =  ^(m0(0), m0(2), m0(4))
andi $6, #1     // $0 = message1(0)
xor $0, $1      // $0 =  ^(m0(0), m0(2), m0(4), m1(0))
cpy $0, $1      // $1 =  ^(m0(0), m0(2), m0(4), m1(0))
lsr $6, #1
lsr $0, #1
andi $0, #1     // $0 = m1(2)
xor $0, $1      // $0 = ^(m0(0), m0(2), m0(4), m1(0), m1(2))
cpy $0, $1      // $1 = ^(m0(0), m0(2), m0(4), m1(0), m1(2))
lsr $6, #4
andi $0, #1     // $0 = m1(4)
xor $0, $1      // $0 = ^(m0(0), m0(2), m0(4), m1(0), m1(2), m1(4))
cpy $0, $1      // $1 = ^(m0(0), m0(2), m0(4), m1(0), m1(2), m1(4))
lsr $6, #4
lsr $0, #1
lsr $0, #1
andi $0, #1     // $0 = m1(6)
xor $0, $1      // $0 = s_p1
add $0, $4      // $0 = 0000 s_p8 s_p4 s_p2 s_p1
cpy $0, $4      // $4 = 0000 s_p8 s_p4 s_p2 s_p1
andi $6, #1     // $0 = p8
lsl $0, #1
cpy $0, $3      // $3 = 000000 p8 0

store $3
cpy $4, $0
store $2

// End:
```

OLD

```
//parity8 -> $14

//s_parity8 -> $14
srl $2, #4                  //$0 = message0(0,3)
cpy $0, $4.                 //$4 = message0(0,3)
xor $5, $4                  //$5 = 4 bits xor result
srl $6, $5, 2               //$6 = 2 bits to xor
andi $5, $5, 3              //$5 = 2 bits to xor
xor $5, $5, $6              //$5 = 2 bits xor result
srl $6, $5, 1               //$6 = 1 bit to xor
andi $5, $5, 1              //$5 = 1 bit to xor
```

```
xor $14, $5, $6                 //$14 = p8 final result


//s_parity4 ->$13
Sll $6, $2, 4
Srl $6, $6, 5                   //message0(0, 2)
Slr $7, $3, 7                   //message0(0, 3)
Xor $6, $6, $7                  //message0(0,2) ^ message(0, 3) = 0 0 0 0 c1 c2 c3 c4
Slr $7, $6, 2                   // isolates c1 c2
Andi $6, $6, 3                  //isolates c3 c4
Xor $6, $6, $7                  //(c1, c2)^(c3, c4) = 0 0 0 0 0 0 C1 C2
Slr $7, $6, 1                   //isolate C1
Andi $6, $6, 1                  //isolate C2
Xor $13, $6, $7                 // xored


//s_parity2 ->$12
sll $4, $2, 4
srl $4, $4, 6                 //message0(4)
sll $5, $2, 6
srl $5, $5, 7             //message0(0,1)
xor $4, $4, $5            //message0(4) ^ message0(0,1)
sll $5, $2, 1
srl $5, $5, 6             //message0(4,5)
xor $4,$4, $5             //message0(4) ^ message0(0,1) ^ message0(4,5)
srl $5, $3, 1             //message1(0,1)
xor $4, $4, $5            //message0(4) ^ message0(0,1) ^ message0(4,5) ^ message1(0,1) = 0 0 0 0 0 0 c1 c2
Srl $5, $4, 1             //isolates c1
Andi $4, $4, 1            //isolate c2
Xor $12, $4, $5
```

```
//s_parity1 -> $11
Andi $6, $2, 3          //message0(4)
Sll $7, $2, 3
Srl $7, $7, 6           //message0(2)
Xor $6, $6, $7          //message0(4) ^ message(2) = 0 0 0 0 0 0 c1 c2
Srl $7, $6, 1           //isolate c1
Andi $6, $6, 1          //isolate c2
Xor $6, $6, $7
Sll $7, $2, 1
Srl $7, $7, 7           //message1(6)
Xor $6, $6, $7
Andi $7, $3, 1          //message1(4)
Xor $6, $6, $7
Srl $7, $3, 2           //message1(2)
Xor $11, $6, $7


//s_parity0 -> $10
xor $10, $11, $12       //p0 = p1^p2
xor $10, $10, $13       //xor p4
xor $10, $10, $14       //xor p8


xor $4, $2, $3          //xor message0 and message1
srl $5, $4, 4           //upper 4 bits
sll $5, $5, 4
sll $6, $4, 4           //lower 4 bits
xor $4, $5, $6          //4 bits left  0000xxxx
sll $5, $4, 6
srl $5, $5, 6           //lower 2 bits
srl $6, $4, 2           //upper 2 bits
xor $4, $5, $6          //2 bits lefts
```

```
sll $5, $4, 7
srl $5, $5, 7          //lower bit
srl $6, $4, 1          //upper bit
xor $4, $5, $6         //1 bit left

xor $10, $10, $4

//error pointer -> $9    //^(parity8^s_parity8,parity4^s_parity4,parity2^s_parity2, parity1, s_parity1)

//get parity8

andi $4, $3, 1  //$4 = 0000000 p8
Sll $4, $4, 3     // $4 = 0000 p8 000

//get parity4
andi $5, $2, 16        //$5 = 000 p4 0000
srl $5, $5, 2          //$5 = 00000 p4 00
or $4, $4, $5          //$4 = 0000 p8 p4 00

//get parity2
andi $5, $2, 4         //$5 = 00000 p2 00
srl $5, $5, 1          //$5 = 000000 p2 0
or $4, $4, $5          //$4 = 0000 p8 p4 p2 0


//get parity1
andi $5, $2, 2         //$5 = 000000 p1 0
srl $5, $5, 1          //$5 = 0000000 p1
or $4, $4, $5          //$4 = 0000 p8 p4 p2 p1

xor $11, $13, $6       //$11 = s_parity1 ^ parity1
```

```
xor $9, $14,$13        //"parity8"^"parity4"
xor $9, $9, $12          //^"parity2"
xor $9, $9, $11          //^"parity1"



//flag setting
Beq $10, $0, elseif              //if parity0 ==0, branch to else if
Addi $r1, $0, 1                  //set flag to 01

Elseif:
Bne error, $0, else              //if error != 0, branch to else
Addi $r1, $0, $0                 /set flag to 00

Else
Addi $r1, $0, 3                  //set flag to 11

//ret1 = $r4 = msw

Sll $4, $3, 4
Srl $r6, $2, 4
Add $4, $4, $6
Sll $4, $4, 3
Add $4, $4, $14                  //flag[1] + flag[0] + 0 + 0+ 0 + message1(0,2);

//ret0 = $r5 = lsw
srl $5, $2, 1
sll $5, $5, 5     //message0(3,7)
sll $6, $13, 4
or $5, $5, $6   //message0(0,2)
srl $6, $2, 7
sll $6, $6, 4
```

```
or $5, $5, $6    //message0(3,7) + message(0,2)
sll $6, $12, 2
or $5, $5, $6    //message0(4)
sll $6, $11, 1
or $5, $5, $6    //message0(2,4) + P4 + b1 + P2 + P1
or $5, $5, $10 //message1(3, 7) + message0(0, 2) + message0(4);

//store final values in datamem[0:29]
sw $5, 0($15)
sw $4, 1($15)

addi $15, $15, #-1

jump Begin:
End:
```

# Program 3 Pseudocode

```
pattern_search(byte[] memdata[0:59]){
        byte pattern_byte = memdata[32]
        5'b pattern = pattern_byte[7:3]
        int count_a = 0;
        int count_b = 0;
        int count_c = 0;
        for (int i = 0, i <31, i ++){
                Byte b1 = memdata[i];
                Byte b2 = memdata[i+1];
                bool matched = false;
                for(int j=0; j<8; j++){                    //scan through each bit in the byte
                        Int Tmp1, Tmp2;
                        if(j < 4){
                                Tmp1 = b1(j, j+5);
                                Tmp2 = tmp1 ^ pattern_byte;
                                if(!tmp2) {
                                        matched = true;
                                        count_a++;
                                        count_c++;
                                }
                        }
                        Else{
                                Tmp1 = b1(j, 8) + b2(0, j-5);
                                Tmp2 = tmp1 ^ pattern_byte;
                                if(!tmp2) {
                                        matched = true;
                                        count_c++;
                                }
                        }
```

```
            }
            if(matched == true){
                        count_b++;
                }
        }
```

## Program 3 Assembly Code

```
//mem[65] = i (outer loop counter)
//mem[66] = j (inner loop counter)
//mem[67] = count_a          → store counts in memory
//mem[68] = count_b
//mem[69] = count_c
//$4 = b1
//$5 = b2
//$3 = match

// Begin: set 0100, $1          //mem[65] = i (outer loop counter)
lsl $1, #4
set 0001, $0
load $0
cpy $0, $1              //$1 = i
// set $0 to 31
set #0001, $2
lsl $2, #4
set #1111, $2
beq $1, $2      //set $0 to end address
```

```
set #0100, $1          //mem[65] = i (outer loop counter)
lsl $1, #4
set 0001, $0
cpy $0, $2             //$2 = 64 save to address to store later
load $0
cpy $0, $1             //$1 = $0 = i

// getting mem[i]
load $1                //contents of mem[i] into $0
cpy $0, $4             //$4 = mem[i]

// getting mem[i+1]
addi $1, #1            //i++
load $1
cpy $0, $5             //$5 = mem[i+1]
cpy $1, $0
store $2               //store incremented i back in mem[65]

set 0000, $3           //make sure matched = false
set 0000, $6

// use $6 as midpoint passed, ie byte boundaries breached
// Inner: set #0100, $1        //mem[66] = j, check j<8
lsl $1, #4
set #0010, $0
cpy $0, $2                     //$2 = &j
load $0
addi $0, #1
store $2               //increment j and store
cpy $0, $1             //$1 = j (after incrementing)
set #0000, $2          //set $2 to 8
lsl $2, #4
```

```
set #1000, $2
beq $1, $2              //set $0 to InnerEnd address


lsl $4, #1              //shift b1 left by 1
set #0100, $2           //$2=4
// set $0 to Concat address
beq $1, $2              //if j=4
/ set $0 to NoConcat address
beq $1, $1
// Concat:              //b1 = $4, b2 = $5
lsr $5, #4              //shift b2 right 4
or $4, $5
cpy $0, $4              //b1 = b1 | b2
set #0001, $6

// NoConcat:
lsl $4, #1
lsl $0, #1
lsl $0, #1              //$0 = $4 shifted left by 3
xor $0, patternbits?

cpy $0, $7              //$7 = xor result
set #0000, $8
beq $7, $8              //set $0 to Match, if xor result == 0 match found
beq $7, $7              //set $0 to InnerEnd. no match found, branch to end of inner loop

// if j<4 increment count a, regardless increment count c and set matched = true
// Match:
// load count c, increment, store
set #0100, $1           //mem[69] = count_c
lsl $1, #4
```

```
set #0101, $0
cpy $0, $1          //$1 = &mem[69]
load $0             //$0 = count_c
addi $0, #1         //count_c++
store $1            //store count_c back in memory
set #0001, $3       //match = 1 = true

set #0000, $7
lsl $7, #4
set #0001, $0
beq $0, $6          //check if midpoint was passed, if no branch to FirstHalf
beq $1, $1          //set $0 to
// FirstHalf:
set #0100, $1       //mem[67] = count_a
lsl $1, #4
set #0011, $0
cpy $0, $1          //$1 = &mem[67]
load $0             //$0 = count_a
addi $0, #1         //count_a++
store $1

// beq
// InnerEnd: Label

Set #0000, $7
Lsl $7, #4
Set #0001, $0
beq $0, $3          //check if match == 1, if yes branch to IncB
Beq $1, $1          //branch to LoopEnd

set #0100, $1       //mem[68] = count_b     IncB Label:
Lsl $1, #4
```

```
Set #0100, $0
Cpy $0, $1          //$1 = &mem[68]
Load $0             //$0 = count_b
Addi $0, #1         //count_b++
Store $1


// LoopEnd:
// set $0 to begin address
Beq $0, $0          //← b begin
// end label
```

# 7. Program Machine Code

Assembler code to convert assembly code into machine code is turned in as a separate file.

## Program 1

010000101
010000101
000000111
111000101
010001100
010100000
111110001
001001010
001010000
010111001
000000111
001011000
010111001
000000111
010010110
101011000
000000100
010100010
010000010
000000001
001100111
101000001
000000100
010100010
000000010
001100011
101000010
000000100
111010001
010001100

000000001
111000101
000100000
001001001
010010100
010000110
010000010
000000110
010011110
010000010
010000010
010000010
101110000
101000010
000000110
010110010
010110010
000000001
001110111
101000001
000000110
001110011
000000001
001110011
101001000
000000101
111010001
010001100
000000001
111001001
000101000
001001001
010010100
010000110
010000010

010000010
000000100
010010100
010000000
010000000
010000110
010000010
010000010
010000010
101000100
000000100
010010000
010000110
010000010
010000010
101000100
000000100
010011100
010000000
010000110
010000010
010000010
101000100
000000100
010100010
000000101
001100011
101000101
000000100
111010001
010001100
000000001
111001101
000100000
001001001

001010111
000000110
010010000
010000000
010000000
010000110
010000010
010000010
101110000
000000110
010110010
000000001
001000011
101000001
000000001
010010000
010000110
010000010
010000010
010000010
101001000
000000001
001011011
101000001
000000001
010011010
010000010
101000001
000000100
111010001
010001100
000000001
111010001
000100000
001001001

111010001
010001100
000000001
111000101
001001000
000001100
111010001
010001100
000000001
111001001
001001000
000001101
101100101
000000100
111010001
010001100
000000001
111001101
001001000
000001101
101100101
000000100
111010001
010001100
000000001
111010001
001001000
000001101
101100101
000000100
101010011
000000101
010000110
010000100
000000110

010101100
010000110
101000110
000000101
010101100
010000000
010000000
010000110
010000010
010000010
000000110
010101110
010000010
010000010
101000110
000000101
010000010
000000101
010110100
010000000
010000000
010000110
010000010
010000010
101000110
101000100
010011000
000000101
111010001
010001100
000000001
111000101
001001000
001001010
000000101

010010010
010000100
010000000
000000101
111010001
010001100
000000001
111001001
001001000
010001100
000101000
001001010
000000101
010010110
010000010
010000010
010000100
000000001
000110000
001110010
000000110
111010001
010001100
000000001
111001101
001001000
010000000
010000000
000000001
000110000
001001010
000000110
000100000
001110010
000000110

000111001
010001111
000101000
001001001
010001001
000100000
001001001
000000111
111000000
010000100
111010100
011001001

# Program 2

111010000
010000100
111000000
000000110
010011111
001110001
111010000
010000100
111000100
000000110
001110000
011000110
111010000
010000100
111000100
000000110
010011111
001110001
010111111

001111000
000000101
010111001
000000111
010111111
001000000
000000110
010110100
010000110
010000010
000000001
010110110
101000001
000000001
010000010
010000010
000000010
001001111
101000010
000000001
010000010
000000001
001001011
101000010
010000000
000000100
010101110
010000010
000000001
010110110
101000001
000000001
010000010
010000010
000000010

001001111
101000010
000000001
010000010
000000001
001001011
101000010
100000100
010000000
000000100
010101100
010000000
010000000
000000001
010101000
010000000
010000000
001000011
101000001
000000001
010110110
010000010
010000010
101000001
000000001
010110100
010000110
010000010
010000010
101000001
000000001
010000010
000000010
001001011
101000010

100000100
010000000
000000100
001101011
000000001
010101010
010000010
001000011
101000001
000000001
010101110
001000011
101000001
000000001
001110011
101000001
000000001
010110010
010000010
001000011
101000001
000000001
010110110
001000011
101000001
000000001
010110110
010000010
010000010
001000011
101000001
100000100
000000100
001110011
010000000

000000011
001011001
000100000
001010001

# Program 3

010001100
111000100
001000000
000000001
111000110
010010100
111111110
011001010
111010001
010001100
111000100
000000010
001000000
000000001
001001000
000000100
010001001
001001000
000000101
000001000
001010001
111000011
111000010
010001100

111001000
000000010
001000000
010000001
001010001
000000001
111000010
010010100
111100010
011001010
010100000
111010010
011001010
011001001
010101110
001100010
000000100
111000110
010100000
010000000
010000000
101000000
000000111
111000001
011111001
011111111
111010001
010001100
111010100
000000001
001000000
010000001
001001001
111000111
111000011

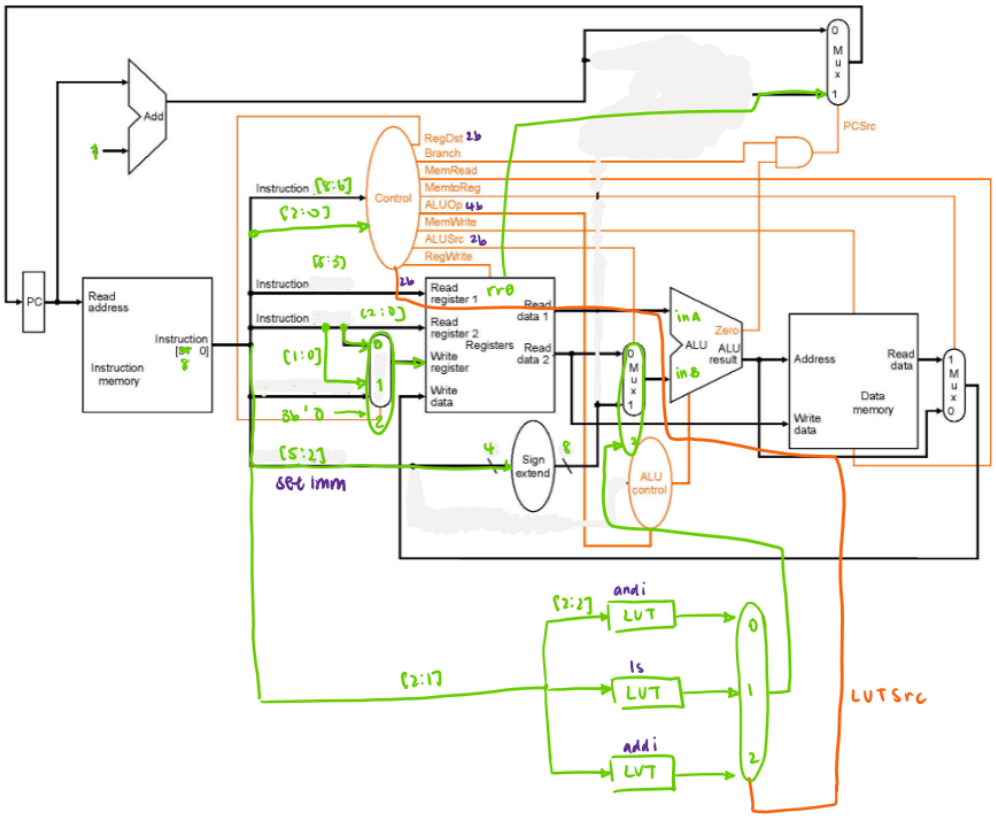
010111100
111000100
011000110
011001001
111010001
010001100
111001100
000000001
001000000
010000001
001001001
111000011
010111100
111000100
011000011
011001001
111010001
010001100
111010000
000000001
001000000
010000001
001001001

# 8. Changelog

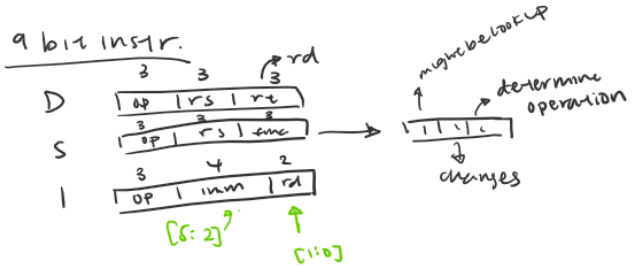
- Milestone 3
  - New section 7 for machine code of each program
  - Programmer's Model
    - We changed our branch implementation back to relative, as we realized relative worked better for what we wanted to do.
  - Architectural Overview
    - Updated diagram to fix branch logic and added computer generated version
- Milestone 2
  - Introduction
    - Added a little bit of how we are using accumulator logic.
  - Architectural Overview
    - Updated diagram to include wires and detailed control logic.
  - Machine Specification
    - Changed function bits for logical shift and addi to be more consistent with operations that also use the last 3 or 2 bits as function bits
  - Programmer's Model
    - We changed our branching implementation to support absolute branching instead of relative.
- Milestone 1
  - Initial draft

| function | b1 | b2 | b3 | b4 | b5 | b6 | b7 | b8 | b9 |
|---|---|---|---|---|---|---|---|---|---|
| cpy | 0 | 0 | 0 | rd1 | rd2 | rd3 | rs1 | rs2 | rs3 |
| Load (absolute address of memory -> r0) | 0 | 0 | 1 | r1 | r2 | r3 | ? | 0 | 0 |
| Store (rs is always r0) | 0 | 0 | 1 | r1 | r2 | r3 | ? | 0 | 1 |
| or (rs \| r0 = r0 | 0 | 0 | 1 | rs1 | rs2 | rs3 | ? | 1 | 0 |
| andi | 0 | 0 | 1 | rs1 | rs2 | rs3 | 0 = 1<br>1 = 3 | 1 | 1 |
| ls (logical shift) | 0 | 1 | 0 | rs1 | rs2 | rs3 | 0 = shift 1<br>1 = shift 4 | 0<br>0 = shift left<br>1 = shift right | 0 |
| addi | 0 | 1 | 0 | rs1 | rs2 | rs3 | 00 = add 1<br>01 = add 5<br>10 = add 15<br>11 = add 30 | | 1 |
| beq (r0 always | 0 | 1 | 1 | rs1 | rs2 | rs3 | rt1 | rt2 | rt3 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| holds branch address) | | | | | | | | | |
| add (rs + rt = r0) rd is r0 | 1 | 0 | 0 | rs | rs | rs | rt | rt | rt |
| xor (rs + rs = r0) | 1 | 0 | 1 | rs | rs | rs | rt | rt | rt |
| and (rs + rt = r0) | 1 | 1 | 0 | rs | rs | rs | rt | rt | rt |
| set(4 LSB) | 1 | 1 | 1 | i1 | i2 | i3 | i4 | r1 | r2 |

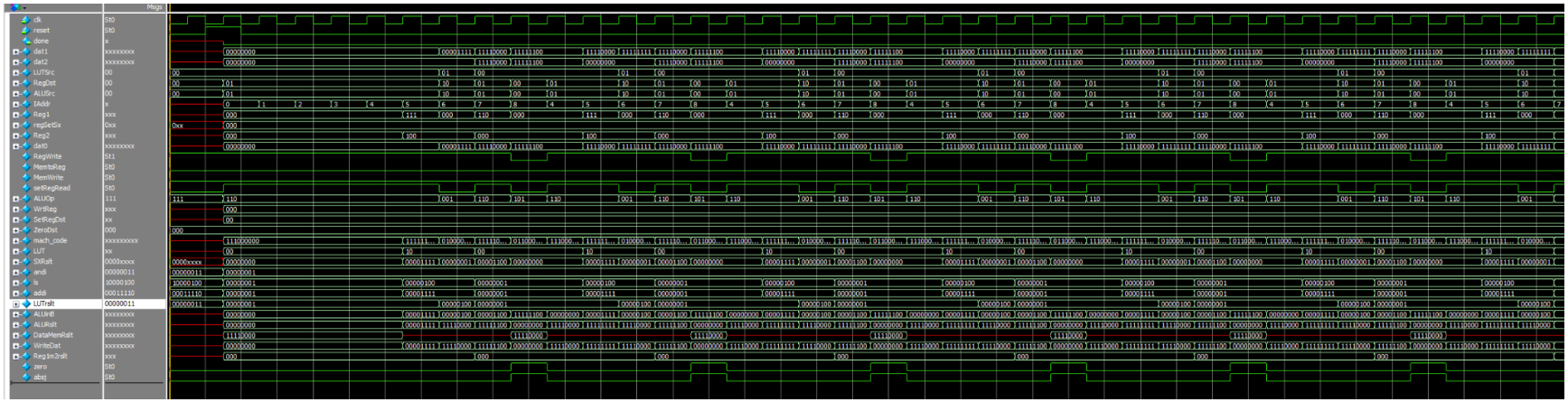



9 bit instr.




RegDst = 0 → last 3 b
= 1 → last 2 b → sign extend?
= 2 → r0

LUTSrc = 0 → andi ( 0=1 , 1=3)
= 1 → ls ( 0x = shift 1
1x = shift 4
x0 = shift left
x1 = shift right)
= 2 → addi ( 00 = 1
01 = 5
10 = 15
11 = 30 )


Msgs
clk St0
reset St0
done x
dat1 xxxxxxxx
dat2 xxxxxxxx
LUTSrc 00
RegDst 00
ALUSrc 00
IAddr x
Reg1 xxx
regSetSx 0xx
Reg2 xxx
dat0 xxxxxxxx
RegWrite St1
MemtoReg St0
MemWrite St0
setRegRead St0
ALUOp 111
WrtReg xxx
SetRegDst xx
ZeroDst 000
mach_code xxxxxxxxx
LUT xx
SXRslt 0000xxxx
andi 00000011
la 10000100
addi 00011110
LUTrslt 00000011
ALUrrR xxxxxxxx
ALURslt xxxxxxxx
DataMemRslt xxxxxxxx
WriteDat xxxxxxxx
RegIn2rslt xxx
zero St0
abej St0

SE 141L > reg_file.sv

```
// cache memory/register file
// default address pointer width = 4, for 16 registers
//change: width = 3, for 8 registers
module reg_file #(parameter pw=2)(
  input reset,
  input[7:0] dat_in,
  input      clk,
  input      wr_en,           // write enable
  input[pw:0] wr_addr,        // write address pointer
              rd_addrA,       // read address pointers
              rd_addrB,
  output logic[7:0] datA_out, // read data
                    datB_out,
                    dat0_out);

  logic[7:0] core[2**pw];    // 2-dim array  8 wide  16 deep

// reads are combinational
  assign datA_out = core[rd_addrA];
  assign datB_out = core[rd_addrB];
  assign dat0_out = core[0];

// writes are sequential (clocked)
  always_ff @(posedge clk)
  if(reset & !wr_en) begin
    core[0] <= 8'b00000000;
    core[1] <= 8'b00000000;
    core[2] <= 8'b00000000;
    core[3] <= 8'b00000000;
    core[4] <= 8'b00000000;
    core[5] <= 8'b00000000;
    core[6] <= 8'b00000000;
    core[7] <= 8'b00000000;
  end
    if(wr_en)                  // anything but stores or no ops
      core[wr_addr] <= dat_in;

endmodule
/*
      xxxx_xxxx
      xxxx_xxxx
      xxxx_xxxx
      xxxx_xxxx
      xxxx_xxxx
      xxxx_xxxx
      xxxx_xxxx
      xxxx_xxxx
      xxxx_xxxx
      xxxx_xxxx
```

...E 141L/reg_file.sv -- Unsuccessful Compile

```
vlog -work work -sv -stats=none {C:/Users/singh/Documents/school-work-testing/CS
E 141L/reg_file.sv}
Model Technology ModelSim - Intel FPGA Edition vlog 2020.1 Compiler 2020.02 Feb
28 2020
-- Compiling module reg_file
** Error: (vlog-13069) C:/Users/singh/Documents/school-work-testing/CSE 141L/reg
_file.sv(36): near "[": syntax error, unexpected '['.
** Error: C:/Users/singh/Documents/school-work-testing/CSE 141L/reg_file.sv(36):
 (vlog-13205) Syntax error found in the scope following 'core'. Is there a missi
ng '::'?
```

Close

```
// program counter
// supports both relative and absolute jumps
// use either or both, as desired
module PC #(parameter D=12)(
  input reset,          // synchronous reset
        clk,
  reljump_en,             // rel. jump enable
        absjump_en,       // abs. jump enable
  input       [7:0] target, // how far/where to jump
  output logic[D-1:0] prog_ctr
);

  always_ff @(posedge clk)
    if(reset)
    prog_ctr <= 'b000000000000;
  else if(reljump_en)
    prog_ctr <= prog_ctr - (128 * target[7]) + target[6:0];
    else if(absjump_en)
    prog_ctr <= target;
  else
    prog_ctr <= prog_ctr + 'b1;

endmodule
```

111000000
010000100
111110000
011000000
111000000
111000000

111000000
111000000
111000000