CSE 141L Milestone 3

Zuo Yang, A16631720; Zhengyu Huang, A16358704

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.

Zuo Yang
Zhengyu Huang

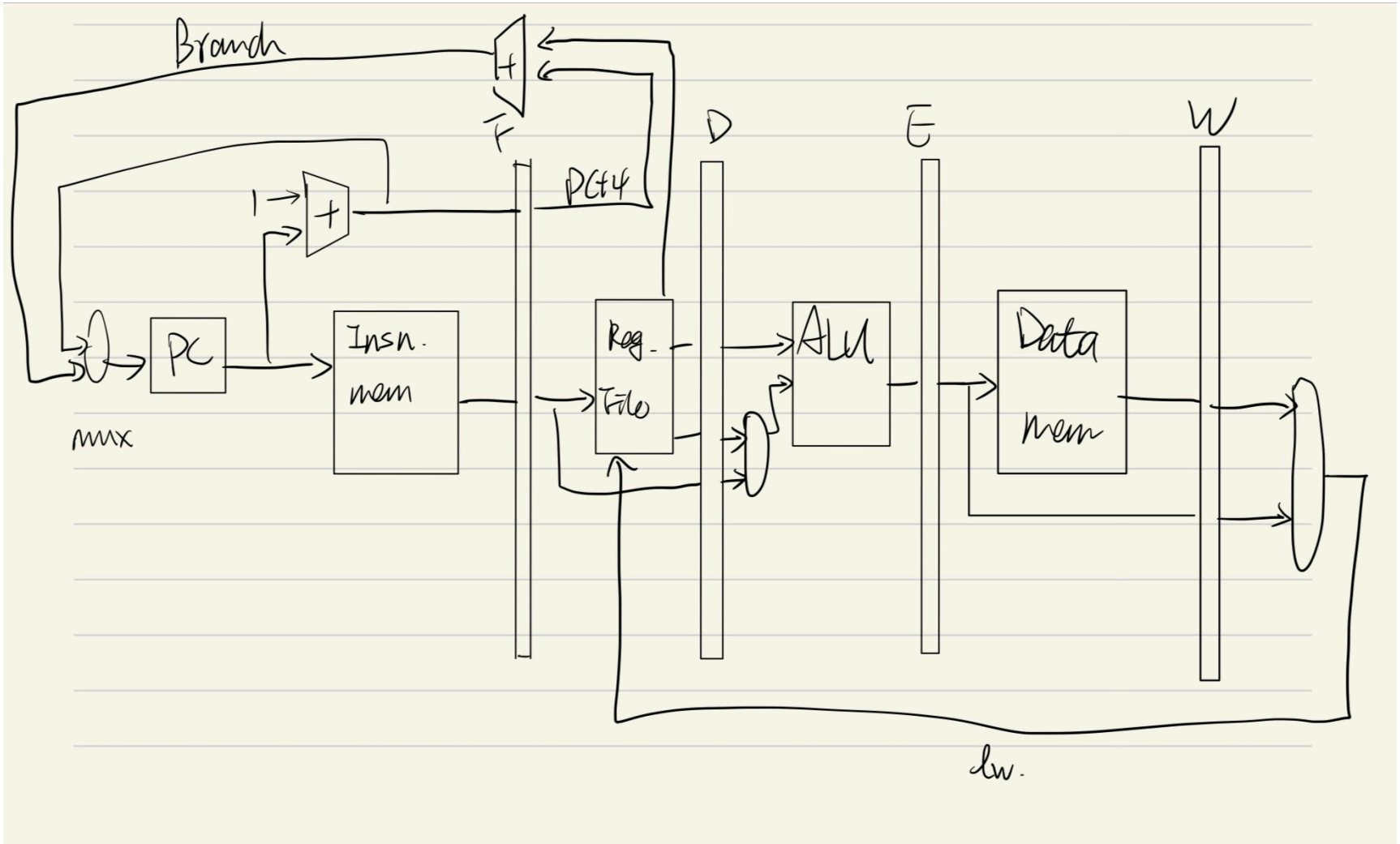
0. Team

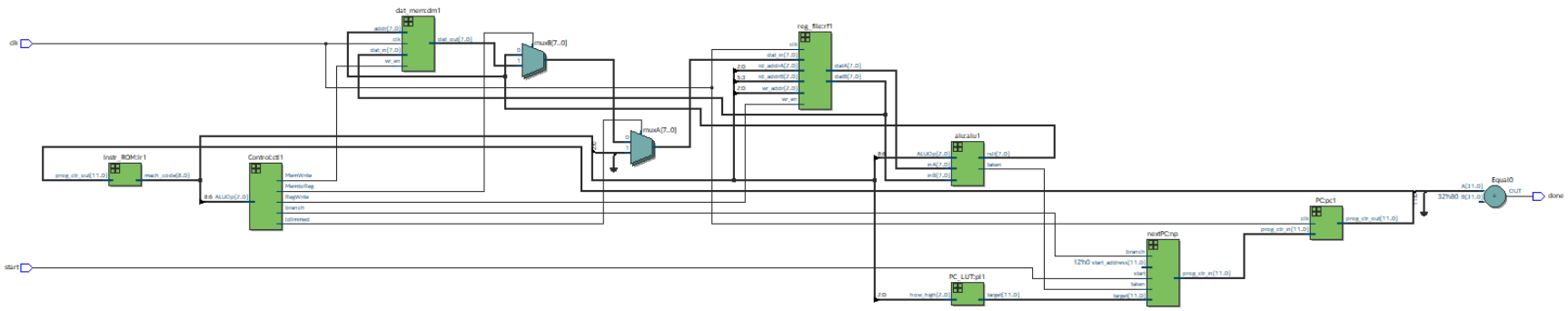
Zuo Yang, Zhengyu Huang

1. Introduction

Our architecture has a name of Low Bandwidth Deducer (LBD). The overall philosophy is to be clear and simple. The goal we have is to consume resources as little as possible. Our machine is a load-store machine. We load the desired operand from data mem into the reg file and perform bitwise operations and store the result back into data mem. Our goals include design of bitwise operations, load and store operations, branching operations, etc.

2. Architectural Overview




clk
start
dst_memrcdm1
muxB[7..0]
muxA[7..0]
reg_filecrf1
instr_ROMclr1
Controlctl1
alucalu1
nextPCcnp
PC_LUTpl1
PCpc1
Equal0
done

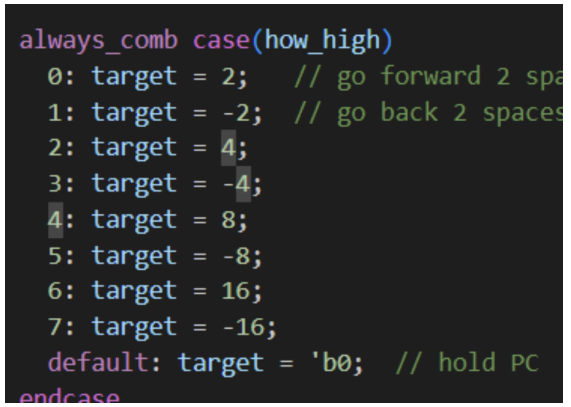
3. Machine Specification

Instruction formats

TYPE	FORMAT	CORRESPONDING INSTRUCTIONS
2 source regs insns	3 bits opcode, 3 bits destination and source register, 3 bits source register	xor, beq, lw, sw, pos
1 source insn (3 bits reg)	3 bits opcode, 3 bits destination and source register, 3 bits intermediate	lrt
1 source isns (2 bits reg)	3 bits opcode, 2 bits destination register, 4 bits intermediate	ld

Operations

NAME	TYPE	BIT BREAKDOWN	EXAMPLE	NOTES
ld = load an intermediate to 3 LSB of the register	1 source regisns (2bits reg)	3 bits opcode (000), 3 bits source register(XXX), 3 bits intermediate (XXX)	ld R3 7⇔ `000_011_111` # after and instruction, R3 now holds `0b0000_0111`	Loading a 3 bits intermediate to the LSB of a register. Intermediate field can only take 0-7
rst = arithmetic right shift 1	1 source regs insns (3 bits reg)	3 bits opcode (001), 3 bits destination register (XXX)	# Assume R0 has `0b1010_0000` `lrt R0 101 ⇔ 001_000_5` # after and instruction, R0 now holds `0b0100_0000`	Right shift by 1
add = arithmetic add	2 source regs insns	3 bits opcode (010), 3 bits destination and source register (XXX), 3 bits source register(XXX)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` `add R0 R1 ⇔ 010_000_001` # after and instruction, R0 now holds `0b1010_0001`	
pos = positive	2 source regs	3 bits opcode (011), 3 bits destination and source register (XXX), 3 bits	# Assume R0 has `0b1001_0000` # Assume R1 has `0b0000_0001`	If R1 > 0, go to the memory location at R0 Pos compares if data in R1 is positive, if so, take the branch.

number	insns	source register(XXX)	`pos R0 R1 ⇔ 011_000_001` # after and instruction, the current address is saved in LUT, and PC is now `0b1001_0000`	
xor = bitwise xor	2 source regs insns	3 bits opcode (100), 3 bits destination and source register (XXX), 3 bits source register(XXX)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1001_0000` `xor R0 R1 ⇔ 100_000_001` # after and instruction, R0 now holds `0b1000_0001`	The data of the first reg will be replaced If want to have “~r2”, can use the following: ld r1, 4b1111 lrt r1, 4 ld r1, 4b1111 xor r2, r1
Jmp = jump to if equals to 0	1 source reg insns	3 bits opcode (101), 3 bits source register (XXX), 3 bits jump code(XXX)	# Assume R0 has 0b0000_0000 # Assume jmp code is 6 = 0b110 # Assume pc = 0b1000_0000 beq R0 R1 ⇔ 101_000_110 # after and instruction, PC now equals to 0b1001_0000	
lw = load word	2 source regs insns	3 bits opcode (110), 3 bits destination register (XXX), 3 bits register for memory origin (XXX)	# Assume R1 has `0b1011_0011` # Assume memory 0xb3 has `0b1111_1111` `lw R0 R1 ⇔ 110_000_001`	0b10110011=0xb3

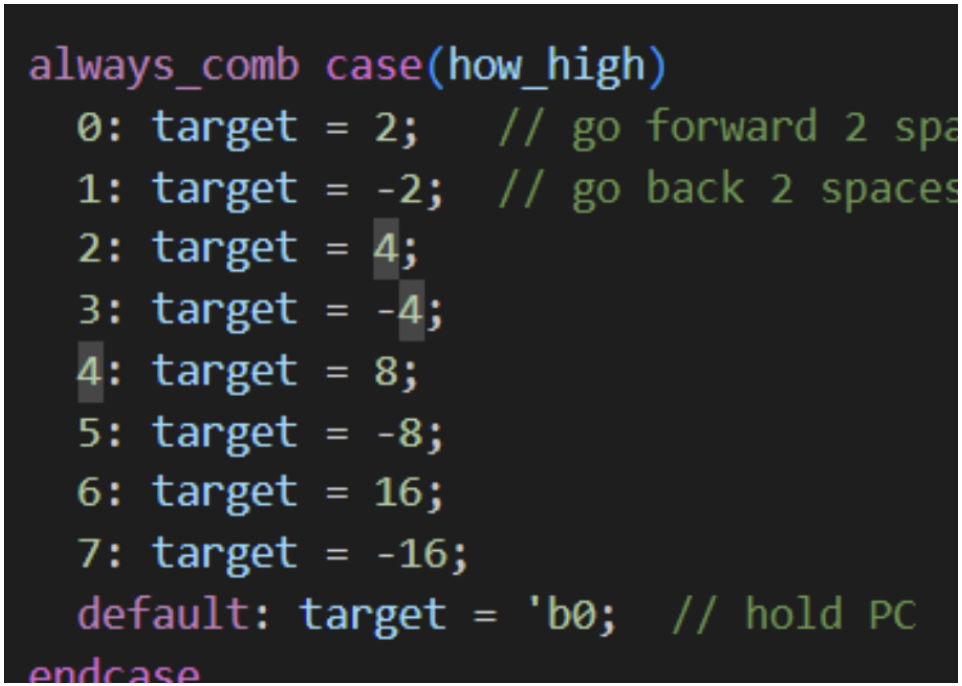
			# after and instruction, R0 now holds `0b1111_1111`	
sw = store word	2 source regs insns	3 bits opcode (111), 3 bits source register (XXX), 3 bits register for memory destination(XXX)	# Assume R0 has `0b0001_0001` # Assume R1 has `0b1011_0011` `sw R0 R1 ⇔ 111_000_001` # after and instruction, memory 0xb3 now holds `0b0001_0001`	

Internal Operands

5 general purpose registers storing 00000000 each. R5 stores constant 11111111. R6 stores constant 00000001. R7 stores constant 01000000. 1 program counter.

Control Flow (branches)

We will use jump and no-operation codes to control the flow. If register input of jmp has a value 0, modify pc according to the value from the look-up table by jump code input (see below). Use no–operations insns in between jumps gaps, (e.g. lrt R0 0).



```
always_comb case(how_high)
  0: target = 2;   // go forward 2 spa
  1: target = -2;  // go back 2 spaces
  2: target = 4;
  3: target = -4;
  4: target = 8;
  5: target = -8;
  6: target = 16;
  7: target = -16;
  default: target = 'b0;  // hold PC
endcase
```

Addressing Modes

Direct. Addr = reg. Only 256 addresses supported
E.g. to access mem@ 0xde
ld R0, 0xd
lrt R0, 4
ld R0, 0xe
lw R1, R0

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.

Our machine requires the programmer to load everything from data memory to registers before executing programs, because our machine is a load-store machine and it uses absolute addressing. Load and store uses direct addressing. To perform a for loop, first store the value of index in one of the gp registers. Then use beq to store the absolute address of the for loop instruction in another gp register. After performing the instructions inside the for loop, increment the index, and load the address of for loop back to pc to complete the jump. The loop will terminate once the index is equal to the last value. For conditionals, to compare to values, first make one side negative. Then add the new value with the other one. Finally we use pos to check if the result is positive or negative.

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. there are a lot of bit length differences in the instructions for the bit budget purpose. We limit some of the insns like ld only accessible to a few registers.
We borrowed some of the ideas of instructions in MIPS ISA in our instructions design. Instructions like add, lrt, sw, lw, and xor. But our beq is different because it only compares the value to 0. Pos is our unique instruction.

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?

No

5. Individual Component Specification

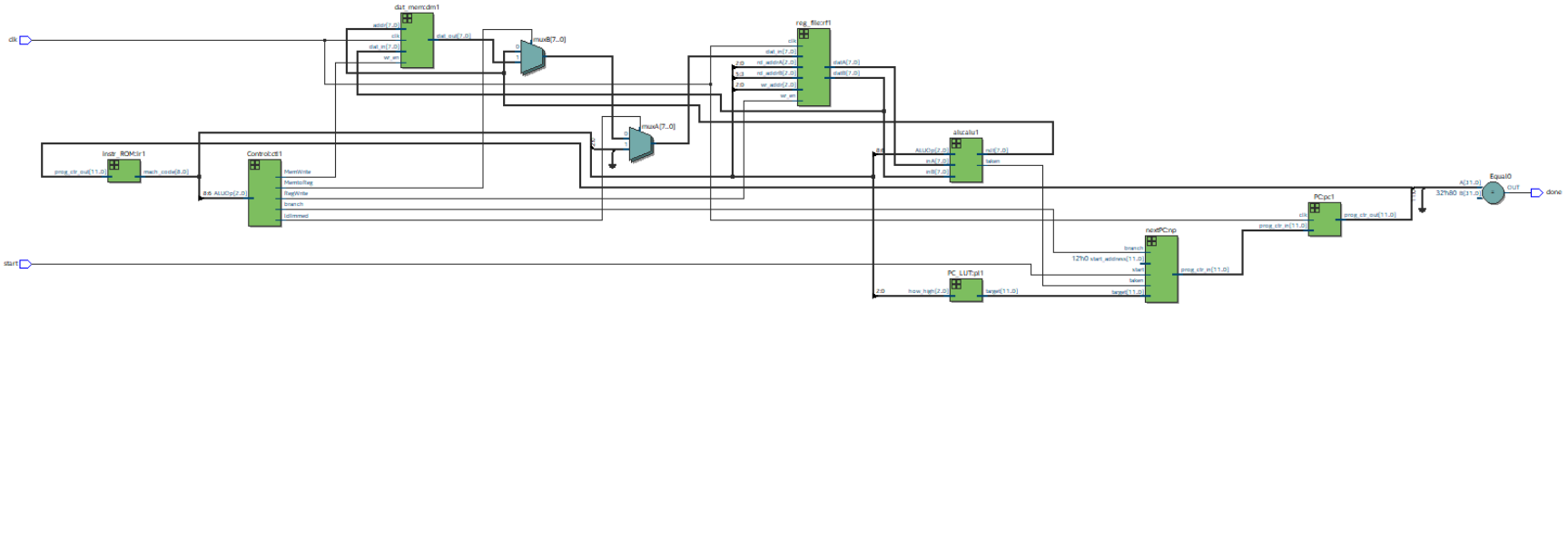
Top Level

Module file name: Top_level.sv

Functionality Description

TODO. Write a brief description of the functionality of this module.

Schematic


clk
start
dat_memrdm1
muxB[7..0]
muxA[7..0]
reg_filerf1
instr_ROMirl
Controlctl1
alucallu1
PCpc1
nextPCnp
PC_LUTplt1
Equal0
done

Program Counter

Module file name: PC.sv
Module testbench file name: testbenches/PC/PC_testbench.sv

Functionality Description;

When clk rises, prog_ctr_out is updated with the value of prog_ctr_in.

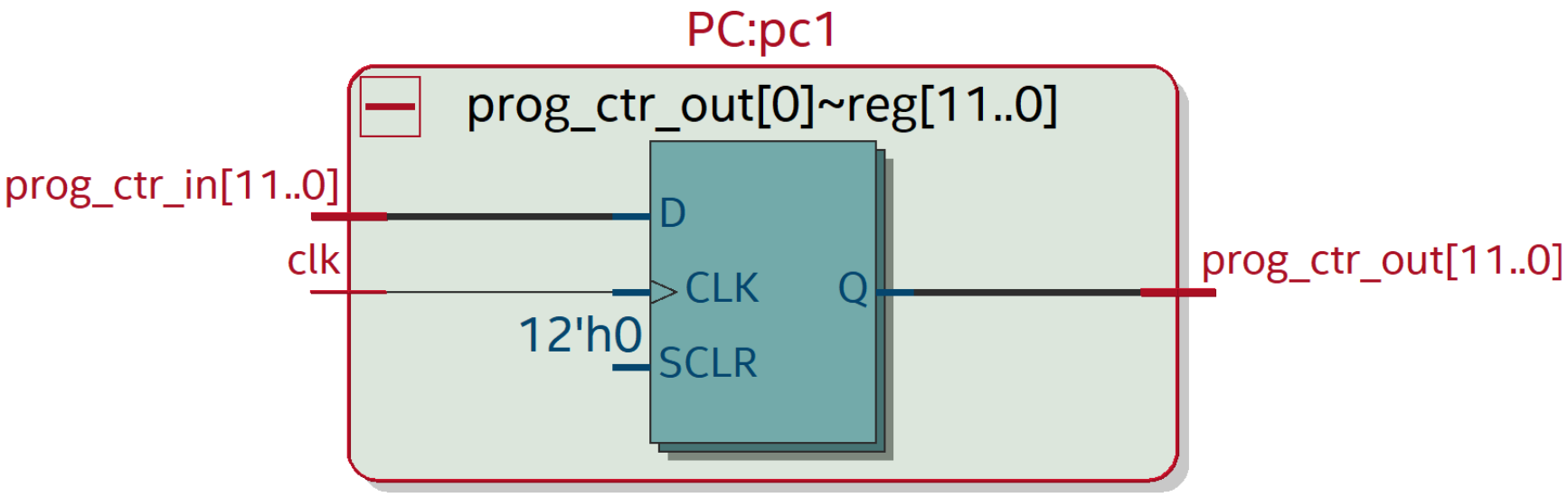
(Optional) Testbench Description

The test bench will instantiate a PC and a nextPC. Both will be run to test if the behavior is correct under different corresponding conditions.

Cases:

1. Initialize
2. Normal incrementing
3. Only *branch* signal
4. Only *taken* signal
5. *jump* 16 and -2
6. *start* at 128

Schematic



(Optional) Timing Diagram

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

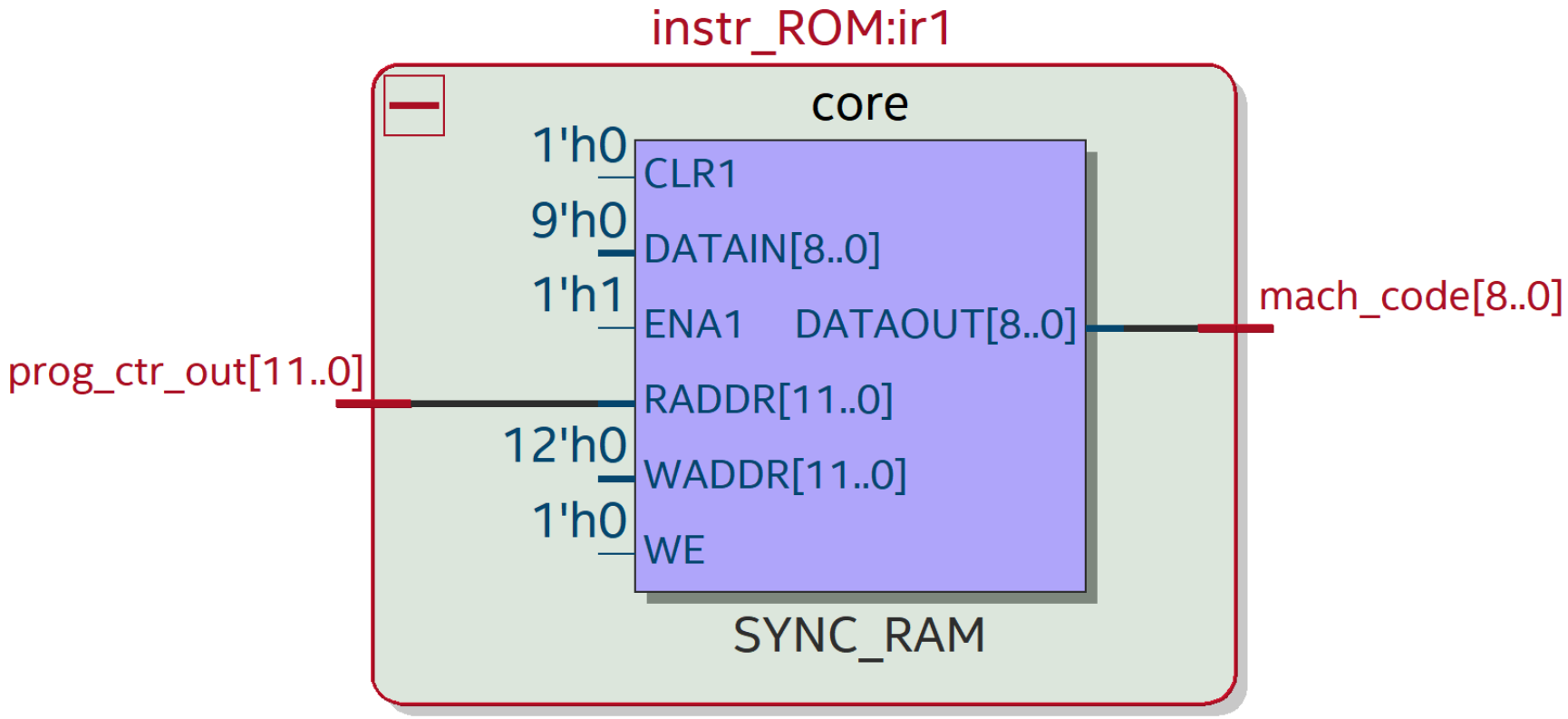
Instruction Memory

Module file name: instr_ROM.sv

Functionality Description

Read the contents of mach_code.txt, output the line of machine code pointed by prog_ctr_out.

Schematic



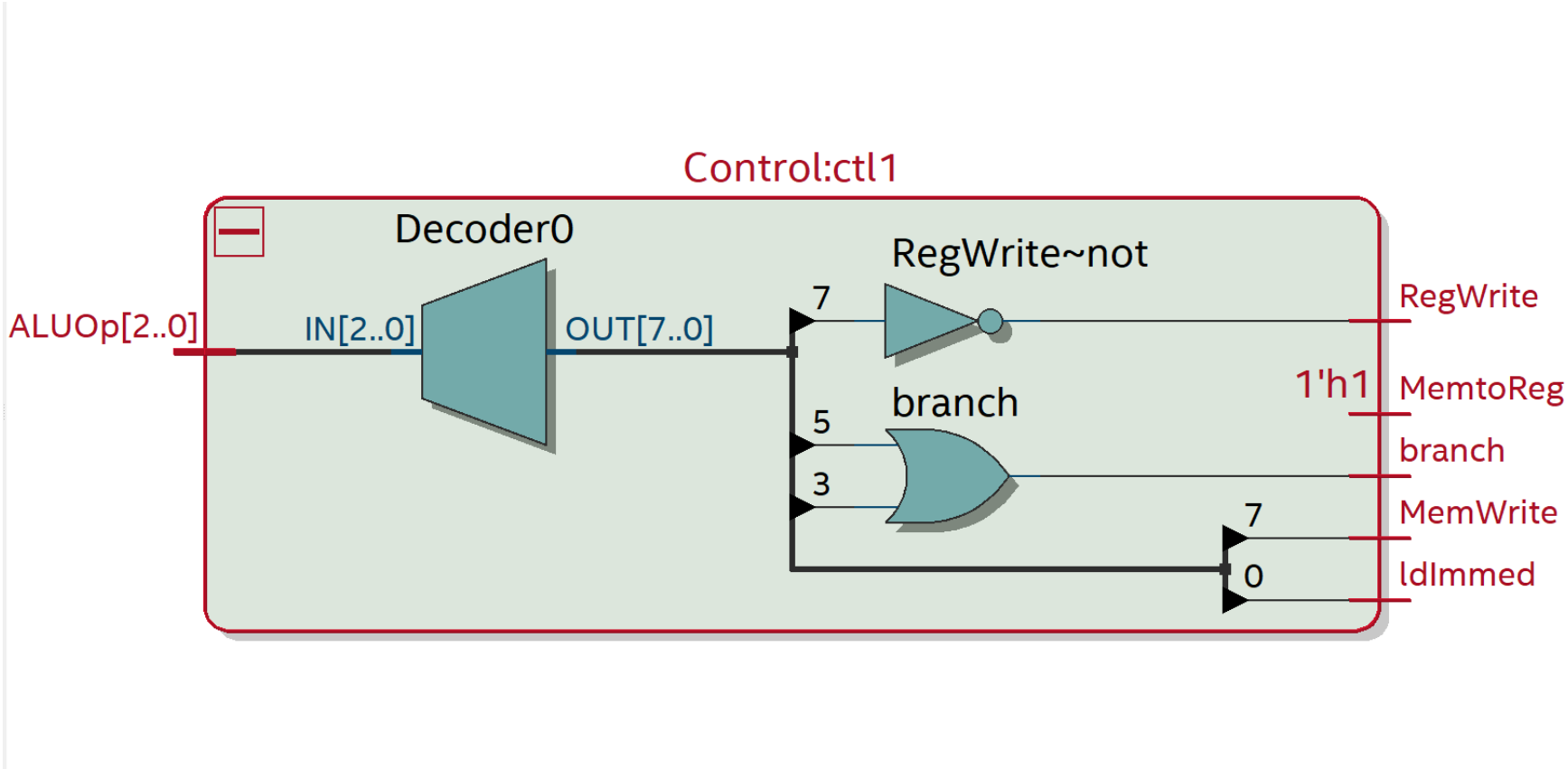
Control Decoder

Module file name: Control.sv

Functionality Description

Read the 3-bit opcode and decide the output of each control signal.

Schematic



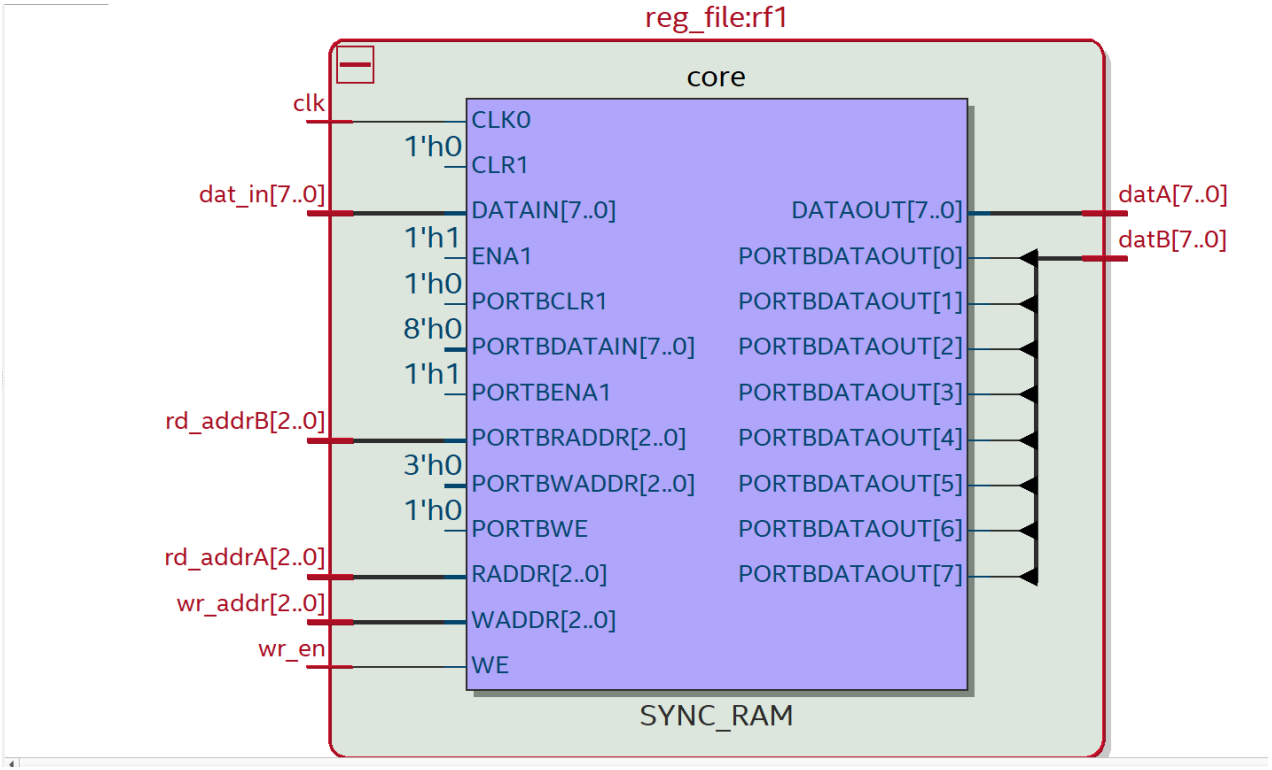
Register File

Module file name: reg_file.sv

Functionality Description

Write the value of dat_in to wr_addr when wr_in is 1. Convert rd_addrA and rd_addrB to its data in the reg_file.

Schematic



ALU (Arithmetic Logic Unit)

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

Functionality Description

Do the calculations based on opcode. If it is a branch instruction and should be taken, taken is 1.

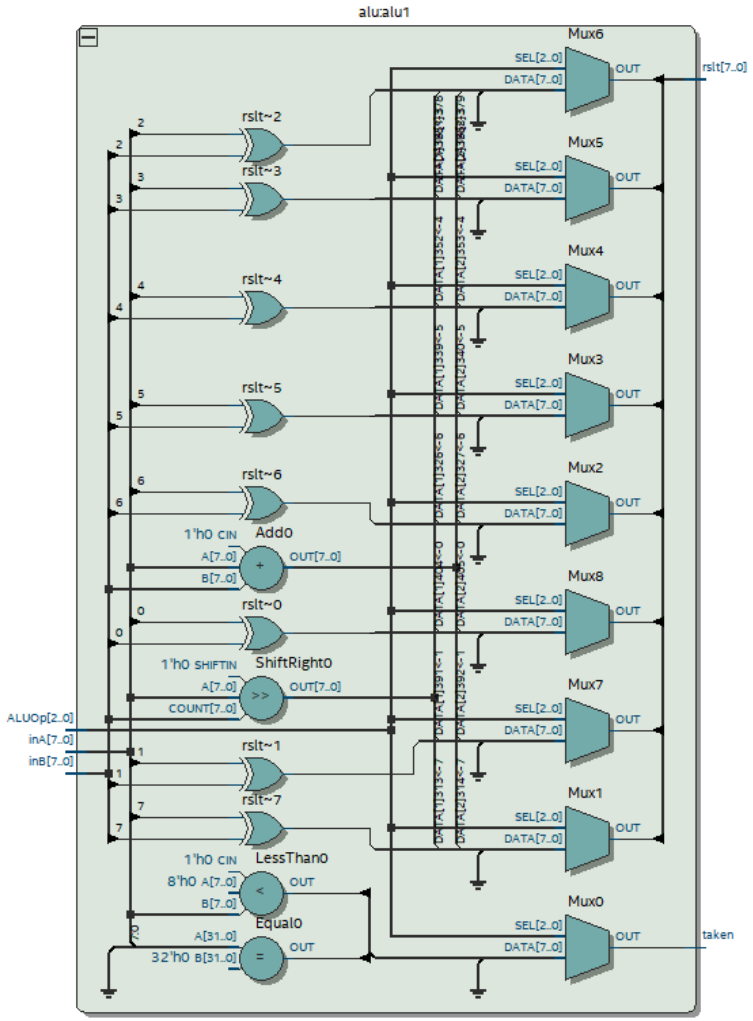
(Optional) Testbench Description

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

ALU Operations

001: left shift
010: add
011: if positive number, taken is 1.
100: bitwise xor.
101: if equal to 0, taken is 1.

Schematic


alu:alu1
Mux6
Mux5
Mux4
Mux3
Mux2
Mux8
Mux7
Mux1
Mux0
SEL[2..0]
DATA[7..0]
OUT
rslt[7..0]
taken
rslt~2
rslt~3
rslt~4
rslt~5
rslt~6
rslt~0
rslt~1
rslt~7
1'h0 CIN
Add0
A[7..0]
B[7..0]
OUT[7..0]
1'h0 SHIFTIN
ShiftRight0
COUNT[7..0]
LessThan0
8'h0 A[7..0]
Equal0
A[31..0]
32'h0 B[31..0]
ALUOp[2..0]
inA[7..0]
inB[7..0]

(Optional) Timing Diagram

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

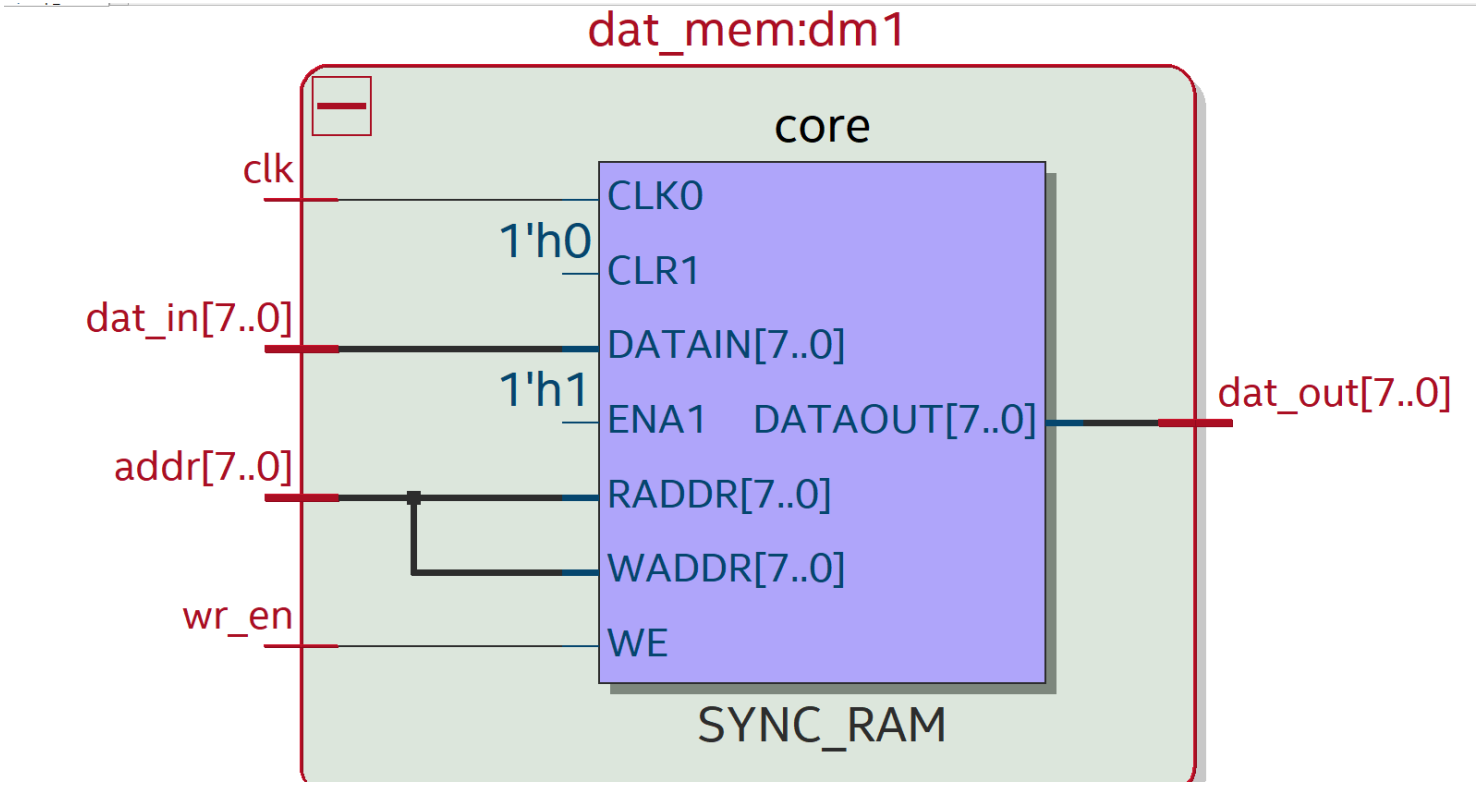
Data Memory

Module file name: dat_mem.sv

Functionality Description

Write the value of dat_in to addr when wr_in is 1. Output the value pointed by addr.

Schematic



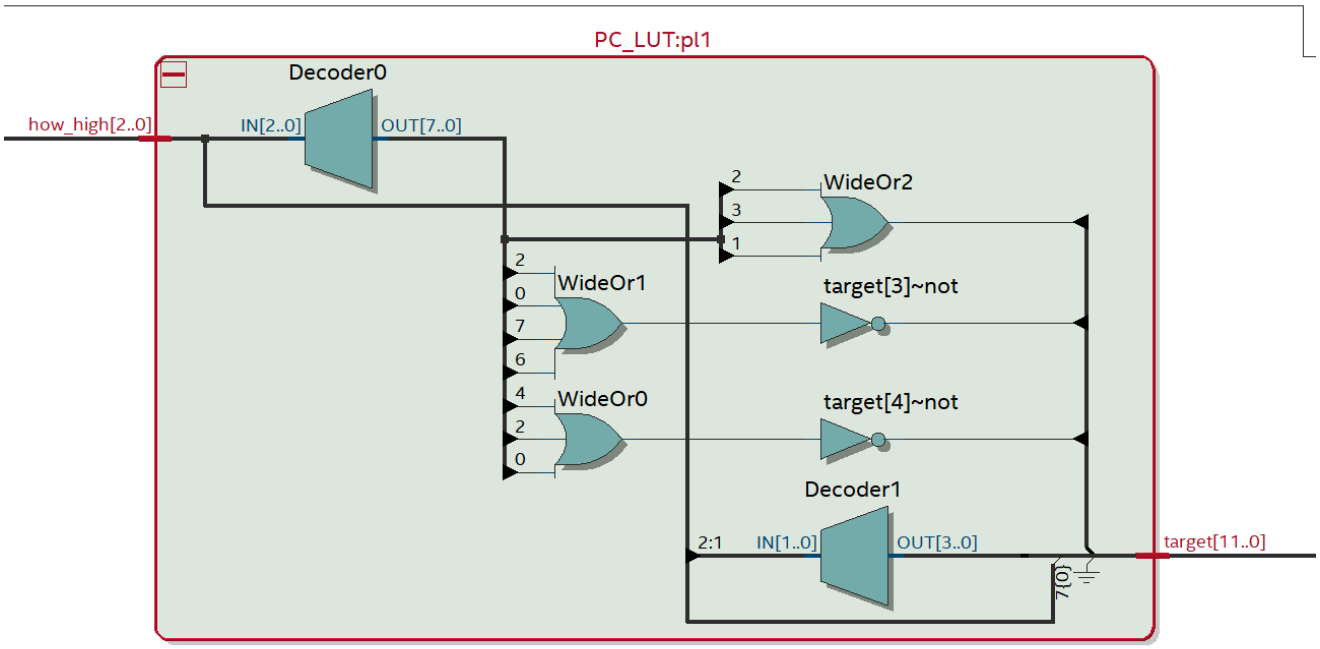
Lookup Tables

Module file name: PC_LUT.sv

Functionality Description

Output target value based on how_high.

Schematic



Muxes (Multiplexers)

No separate file
Module file name: TODO

Functionality Description

TODO. Write a brief description of the functionality of this module.

Schematic

TODO. Show us your schematic for your mux(es).

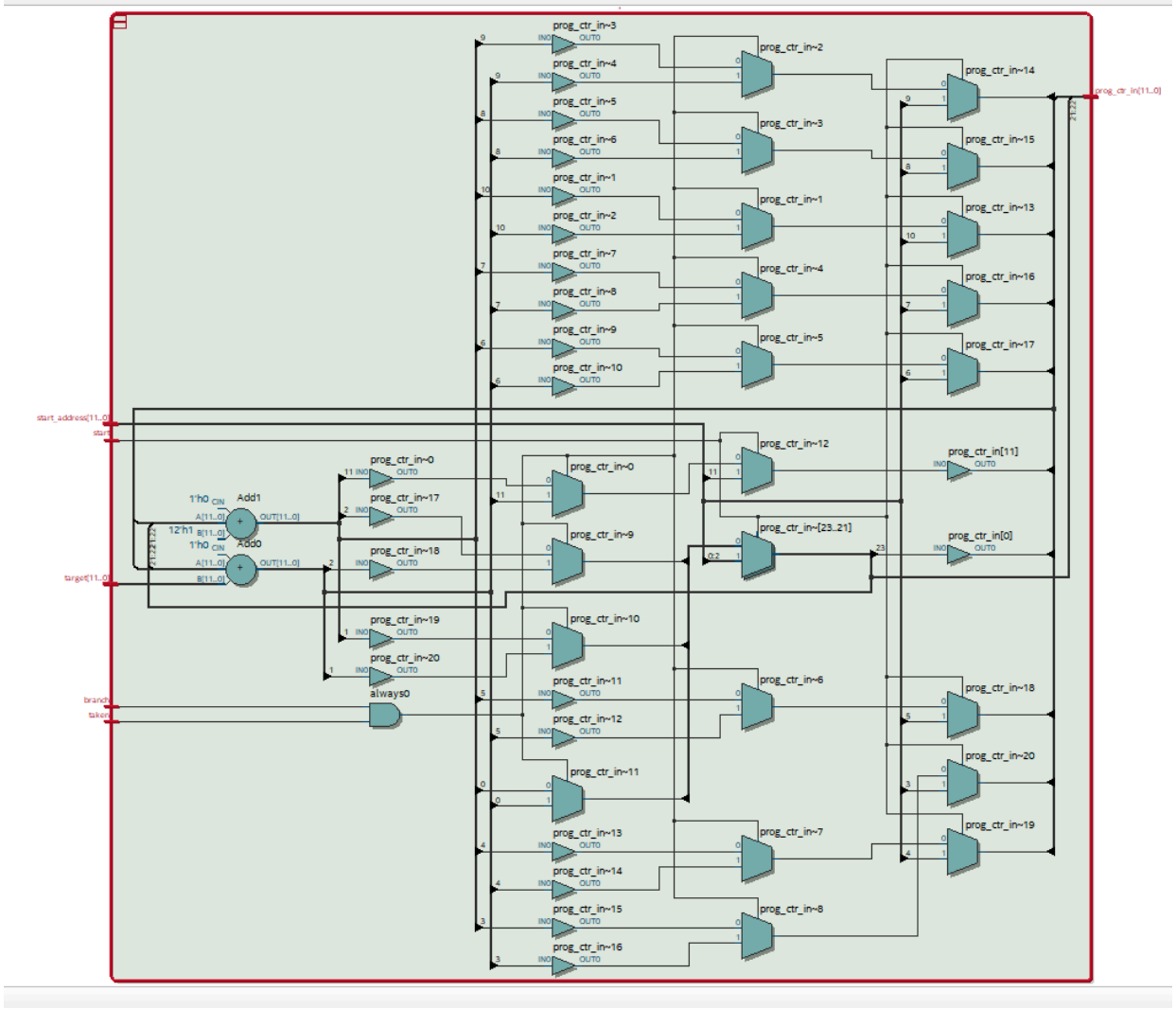
Other Modules (if necessary)

Module file name: nextPC.sv

Functionality Description

If start is 1, reset prog_ctr_out to start address. If branch and taken are both 1, jump to target. Else plus 1.

Schematic


start_address[11..0]
start
target[11..0]
branch
taken
1'h0 CIN
Add1
A[11..0]
12'h1 B[11..0]
1'h0 CIN
Add0
A[11..0]
B[11..0]
OUT[11..0]
always0
prog_ctr_in~0
prog_ctr_in~1
prog_ctr_in~2
prog_ctr_in~3
prog_ctr_in~4
prog_ctr_in~5
prog_ctr_in~6
prog_ctr_in~7
prog_ctr_in~8
prog_ctr_in~9
prog_ctr_in~10
prog_ctr_in~11
prog_ctr_in~12
prog_ctr_in~13
prog_ctr_in~14
prog_ctr_in~15
prog_ctr_in~16
prog_ctr_in~17
prog_ctr_in~18
prog_ctr_in~19
prog_ctr_in~20
prog_ctr_in~[23..21]
prog_ctr_in[11]
prog_ctr_in[0]
prog_ctr_in[11..0]
IN0
OUT0

6. Program Implementation

Assembler: code/assembler/assembler.py

Program 1 Pseudocode

```
def hammingDistance(n):

    #x = n1 ^ n2
    #setBits = 0

    #while (x > 0) :
        #setBits += x & 1
        #x >>= 1

    #return setBits
    max = 0
    min = 16
    for i in range(len(n)-1):
        for j in range(i+1, len(n)):
            x = n[i] ^ n[j]
            setBits = 0

            while x > 0:
                setBits += x & 1
                x >>= 1

            if setBits>max:
                max = setBits
```

```
            elif setBits<min:
                min = setBits
    return max, min
if __name__=='__main__':
    n = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    print(hammingDistance(n))
```

Program 1 Assembly Code

code/assember/prog1.txt

Program 1 Machine Code

code/assember/prog1.sv

Program 2 Pseudocode

```
def arithmaticDifference(n):
    max = 0
    min = 16
    for i in range(len(n)):
        for j in range(i+1, len(n)):
            setBits = n[i] + (~n[j] + 1)
            if setBits<0:
                setBits = -setBits
            if setBits>max:
                max = setBits
            elif setBits<min:
                min = setBits
```

```
    return max, min

if __name__=='__main__':
    n = [0, -2, 4, -1]
    print(arithmaticDifference(n))
```

Program 2 Assembly Code

code/assember/prog2.txt

Program 2 Machine Code

code/assember/prog2.sv

Program 3 Pseudocode

```
import copy


def multiplication(n):
    max = 0
    min = 16
    result = []
    for i in range(len(n)):
        for j in range(i+1, len(n)):
            setBits = 0
            multiplicant1 = copy.deepcopy(n[i])
            multiplicant2 = copy.deepcopy(n[j])
            multiplicant2Copy = copy.deepcopy(n[j])
```

```
            result.append(setBits)
    return result

if __name__=='__main__':
    n = [1, 5, 1, 2, 3]
    print(multiplication(n))
```

Program 3 Assembly Code

code/assember/prog3.txt

Program 3 Machine Code

code/assember/prog3.sv

7. Changelog

- Milestone 3
    - Change lrt to rst, which means arithmetic right shift by 1
    - R5 stores constant -1.
    - R6 stores constant 1.
    - R7 stores constant 64.
    - Ld → 3 opcode, 3 dst/src reg, 3 immediate
- Milestone 2
    - Introduction
        - edited to change from a load/store architecture to accumulator architecture.
    - Operations:

        - Name change lsf → lrt
        - beq→ jmp
    - Control flow:
        - Branch to hard-coded jump.
    - TODO: add bullet points as necessary
- Milestone 1
    - Initial version

1) Your assembler

2) If you had not previously completed your Assembly scripts for the 3 programs then you should add that now. Does not need to be debugged yet, that will be your final goal.

3) Your machine code corresponding to each programs assembly script.