ECE 366 Project 3, Group 4

F2 (FASTv2)

ISA

Part A. ISA Introduction

`FAST (stands for Fast Assembly Super Turbo) ISA`

Philosophy is to minimize use of loops except for looping through entries and have some instructions carry implicit details. A key component we focused on what capturing the counting of bits operation for program 2 in a single instruction that does not utilize loops. We also wanted to support more than 8 instructions by using unique bit encoding that would be particular to what the programs needed (non-generality).

1. Instruction list

Instruction	PC	Coding	Functionality	Example	
init Rx, imm	PC++	000 x iii	Rx = imm Rx ∈ {R0, R1} imm: [0,7]	init R0,4	000 0 100
ld Rx, Ry	PC++	001 xx yy	Rx = Mem[Ry]	ld R0, R1	001 00 01
str Rx, Ry	PC++	010 xx yy	Mem[Ry] = Rx Rx ∈ {R0, R1, R2, R3} Rx ∈ {R0, R1, R2, R3}	st R0, R1	010 00 01
addR Rx	PC++	01100 xx	R2 = Rx + Rx Rx ∈ {R0, R1, R2, R3}	addR R0	01100 00
addR2 Rx	PC++	01110 xx	R2 = R2 + Rx Rx ∈ {R0, R1, R2, R3}	addR2 R1	01110 01
addR3 Rx	PC++	01111 xx	R3 = Rx + Rx Rx ∈ {R0, R1, R2, R3}	addR3 R2	01111 10
subR3 Rx	PC++	01101 xx	R3 = R3 - Rx Rx ∈ {R0, R1, R2, R3}	subR3 R0	01101 00
addi Rx imm	PC++	100 xx ii	Rx = Rx + imm Rx ∈ {R0, R1, R2, R3} imm: [0,3]	addi R0, 2	100 00 01
sltR0 Rx,Ry	PC++	101 xx yy	R0 =1 if Rx < Ry Rx ∈ {R0, R1, R2, R3} Rx ∈ {R0, R1, R2, R3}	sltR0 R0,R1	101 00 11

beqR0 Rx imm	if Rx==RO:
	PC == MUX(imm)
	else:
	PC++
		11 xx iii	Rx ∈ {R0, R1, R3}
			Imm number used to select jumps in a mux
				beqR0 R0, brk	11 00 101

scrR3R2	PC++	1110 111	R3 = the match score of R3 and R2. This function is done using logic circuit.	scrR3R2	1110 111
haltPC	PC stable	1110 000	PC will not change	haltPC	1110 000

2. Register Design

Register Name	Number
R0	00
R1	01
R2	10
R3	11

3. Control Flow

We reduced the number of immediate values for branch instructions to 8 for all of Program 1 and Program 2 to use a MUX to select specific jumps. Our 3 bit immediate number in the beqR0 instruction are used as the select in the MUX since there are 2^3 different PC distances. Since our instruction memory will not change, we were able to allow 3 registers {R0, R1, R0} as possible comparators with R0 increasing the versatility of branching which helped offset having just four registers. In this manner beqR0 was also serving as a "jump" instruction when the registers compared were both R0. The furthest distances we supported were -19 and 18.

Example:

```
Instruction:
beqR0, R3, equal    # Label is +11 bytes from this instruction
                    # All of the jumps are connected to a MUX
                    # In this case we chose 101 as the select for 11
Machine Code:
(0)1111101          # First 4 bits are for beqR0 and register R3 (11)
                    # Last 3 bits (101) are selecting 11 in the MUX

Instruction:
beqR0, R0, loop     # Label is -12 bytes from this instruction
                    # We chose 001 as the select for -12 in MUX design
Machine Code:
(1)1100001
```

4. Memory Model

4.1 Data Memory

- 16-bit double-byte addressable
- 128 memory units in total
- using 7-bit address.

Address	Memory
000 0000	Mem[0]
000 0001	Mem[1]
...	...
111 1111	Mem[127]

4.2 Instruction Memory

- 8-bit byte addressable, PC is initialized at 0
- 64 memory units in total
- using 6-bit address.

Address	Memory
00 0000	Mem[0]
00 0001	Mem[1]
...	...
11 1111	Mem[63]

Example:

Instruction:	
ld r1, (r0)	# r0 = 6 so r1 = mem[6]
Machine Code:	
(0)0010100	# 001 is for ld instruction
	# 01 is for r1(destination) and 00 for r0(address)
Instruction:	
str r1, (r0)	# r0 = 6 so mem[6] = contents of r1
Machine Code:	
(0)0100100	# 010 is for str instruction
	# 01 for r1 and 00 for r0(address)

Part B. Answers to Questions

1. Comparing to the sample of “My_straightforward_ISA”, what are the unique features of your ISA? Explain why your ISA is better.

This ISA features a single instruction for minimizing the time complexity of the bit counting calculation from quadratic to linear time. For example, without using combinational logic there would have to be a loop for each pattern array entry and another loop to check each bit in those entries. This rapidly becomes very slow. Our solution will compute each computation in a constant amount of time for each pattern array entry. Also, we improved branch instruction to be able to jump to specific locations throughout the program. Using a MUX and 3 bit immediate number encoding we were able to jump long distances from current PC. Using uniqueness of bits we were able to support many add instructions that were instrumental in computed exponentiation.

2. In what ways did you optimize for the two goals? If you optimized for anything additional, what and how?
We optimized for lower dynamic instruction count by limiting loops, especially in program 2. Tradeoff for this was slightly increased complexity in hardware when counting bits.

3. What would you have done differently if you had 1 more bit for instructions? How about 1 fewer bit?
If we had 1 more bit for instruction I think I would utilize that for immediate numbers and try to simplify the instruction set and hardware. If we had 1 fewer bit we would have to condense our add instruction to an exponentiator and increase hardware complexity by implementing this. I would also look to see what immediate numbers could possible be used in a MUX and selected based on what is needed for the program.

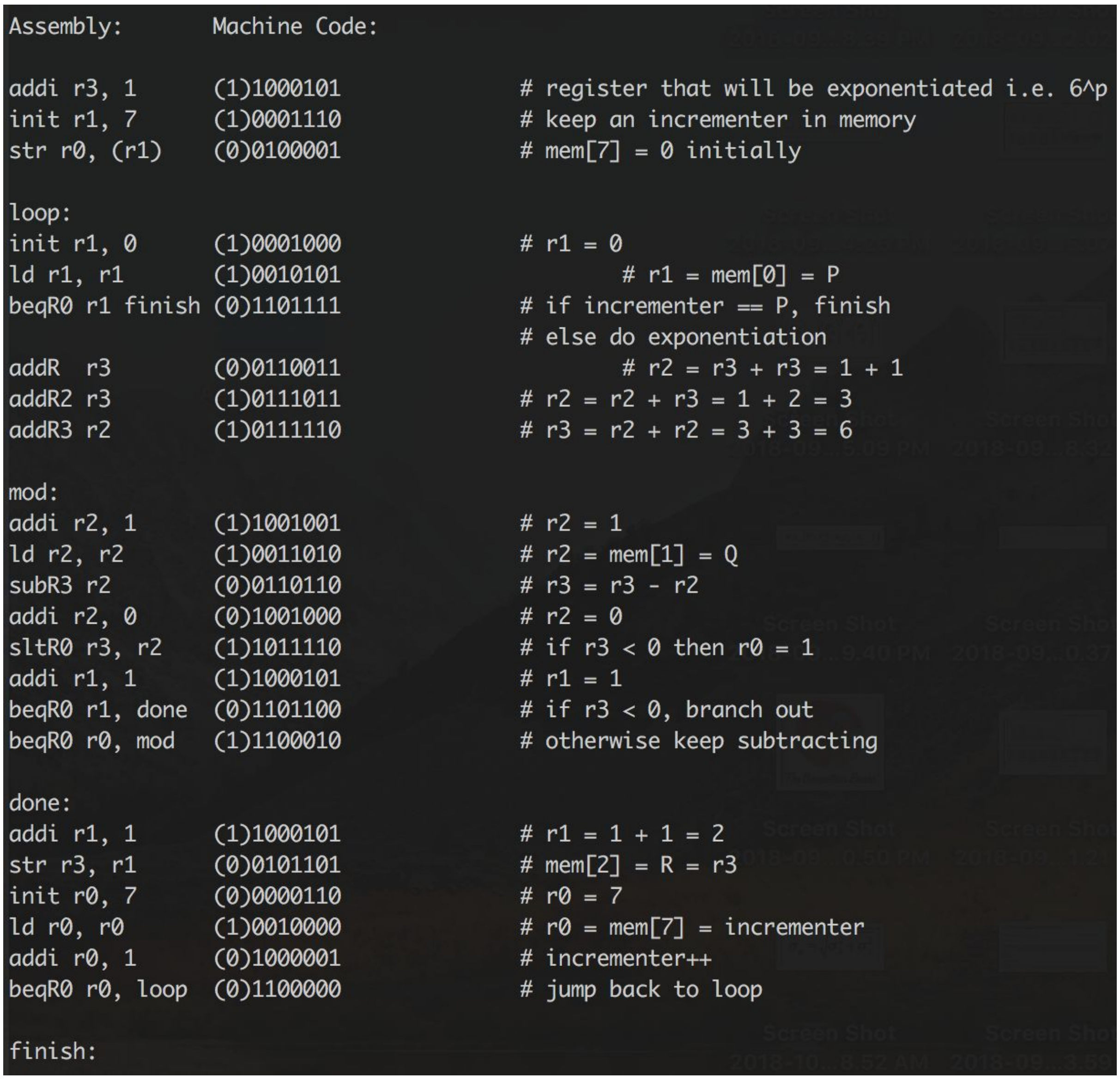
4. How did your team work together to accomplish this project? (Role of each team member, progress milestones, time spent individually and together?)
Role of team members:
We worked together in the library to brainstorm ideas for the ISA. We then collaborated for program 1, doing revisions. Similar for program 2. We met the first progress milestone and second, as well. Working time was spent half together half remote but with relaying of ideas and checkpoints.

5. If you had a chance to restart this project afresh with 3 weeks’ time, how would your team have done differently?

Since we know exactly what the program needs in terms of unique instructions we can have utilize unused instructions. We can also try to use more hardware implementations of instructions to lower dynamic instruction count. For example, we can condense the add instructions to just an exponentiator.

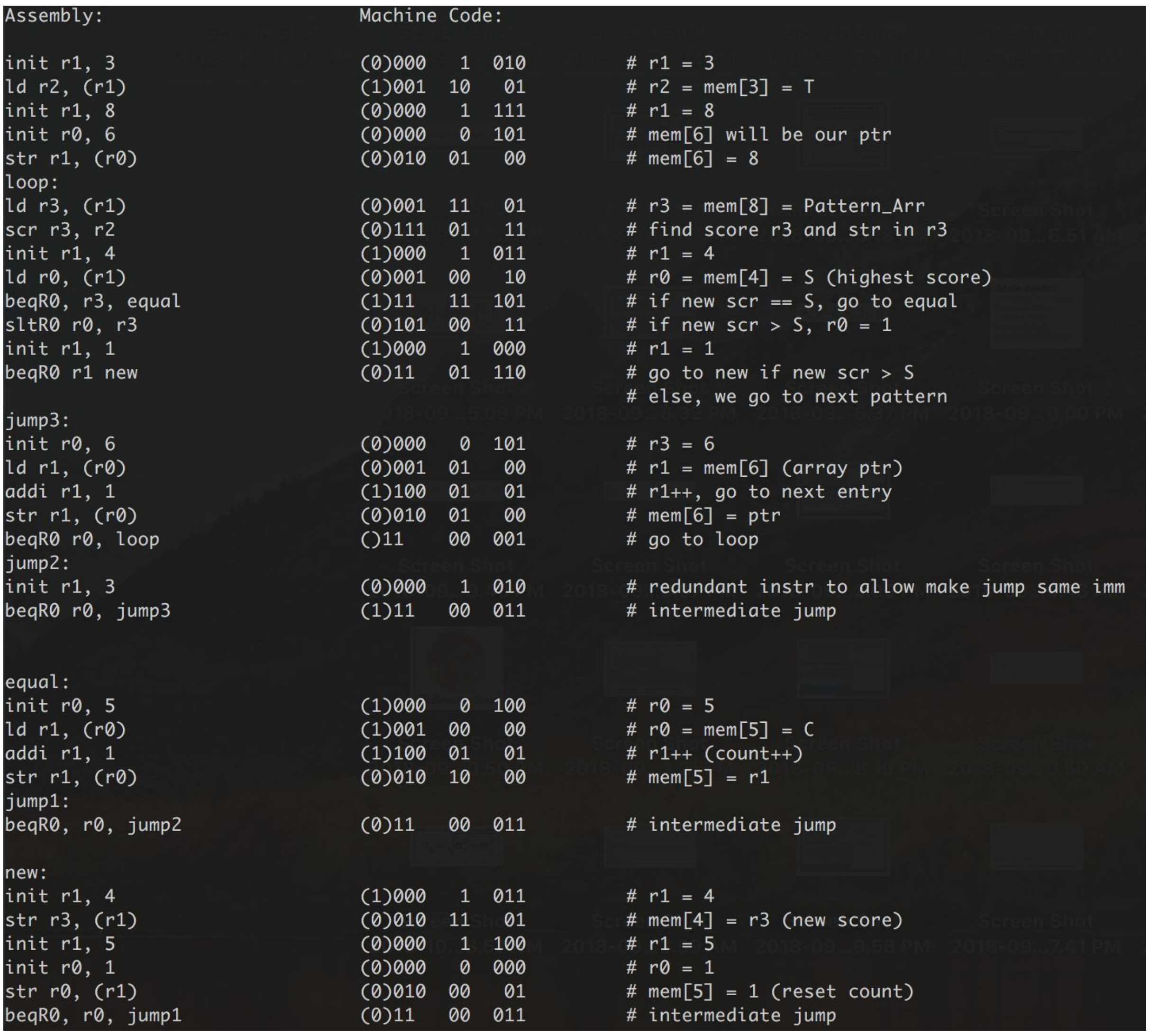
Part C. Software Package

Program 1 Algorithm & Machine Code



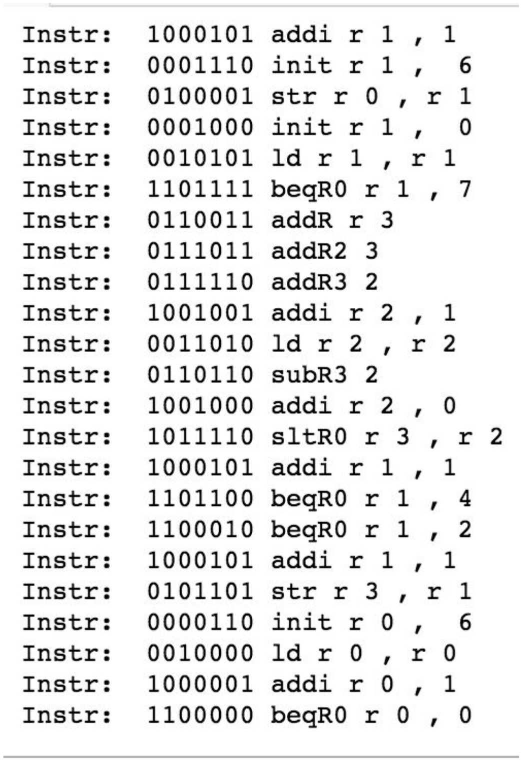
```
Assembly:       Machine Code:

addi r3, 1      (1)1000101          # register that will be exponentiated i.e. 6^p
init r1, 7      (1)0001110          # keep an incrementer in memory
str r0, (r1)    (0)0100001          # mem[7] = 0 initially

loop:
init r1, 0      (1)0001000          # r1 = 0
ld r1, r1       (1)0010101                  # r1 = mem[0] = P
beqR0 r1 finish (0)1101111          # if incrementer == P, finish
                                    # else do exponentiation
addR  r3        (0)0110011                  # r2 = r3 + r3 = 1 + 1
addR2 r3        (1)0111011          # r2 = r2 + r3 = 1 + 2 = 3
addR3 r2        (1)0111110          # r3 = r2 + r2 = 3 + 3 = 6

mod:
addi r2, 1      (1)1001001          # r2 = 1
ld r2, r2       (1)0011010          # r2 = mem[1] = Q
subR3 r2        (0)0110110          # r3 = r3 - r2
addi r2, 0      (0)1001000          # r2 = 0
sltR0 r3, r2    (1)1011110          # if r3 < 0 then r0 = 1
addi r1, 1      (1)1000101          # r1 = 1
beqR0 r1, done  (0)1101100          # if r3 < 0, branch out
beqR0 r0, mod   (1)1100010          # otherwise keep subtracting

done:
addi r1, 1      (1)1000101          # r1 = 1 + 1 = 2
str r3, r1      (0)0101101          # mem[2] = R = r3
init r0, 7      (0)0000110          # r0 = 7
ld r0, r0       (1)0010000          # r0 = mem[7] = incrementer
addi r0, 1      (0)1000001          # incrementer++
beqR0 r0, loop  (0)1100000          # jump back to loop

finish:
```

Program 2 Algorithm & Machine Code



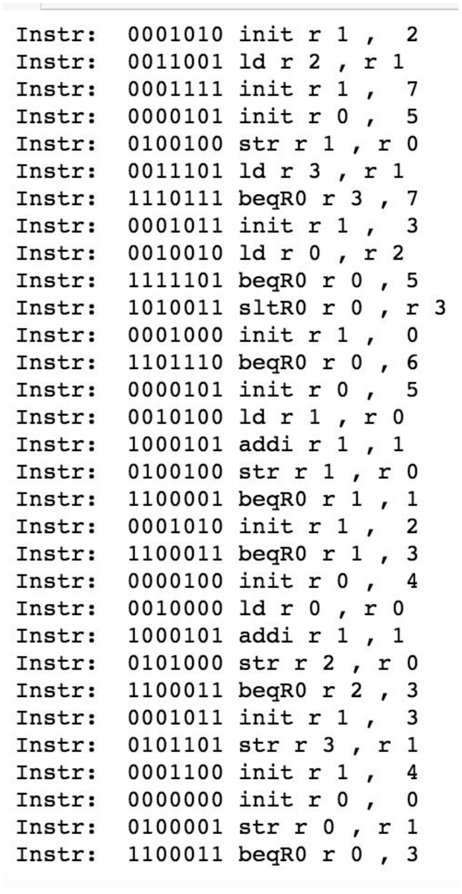
```
Assembly:                       Machine Code:

init r1, 3                      (0)000   1  010        # r1 = 3
ld r2, (r1)                     (1)001  10   01        # r2 = mem[3] = T
init r1, 8                      (0)000   1  111        # r1 = 8
init r0, 6                      (0)000   0  101        # mem[6] will be our ptr
str r1, (r0)                    (0)010  01   00        # mem[6] = 8
loop:
ld r3, (r1)                     (0)001  11   01        # r3 = mem[8] = Pattern_Arr
scr r3, r2                      (0)111  01   11        # find score r3 and str in r3
init r1, 4                      (1)000   1  011        # r1 = 4
ld r0, (r1)                     (0)001  00   10        # r0 = mem[4] = S (highest score)
beqR0, r3, equal                (1)11   11  101        # if new scr == S, go to equal
sltR0 r0, r3                    (0)101  00   11        # if new scr > S, r0 = 1
init r1, 1                      (1)000   1  000        # r1 = 1
beqR0 r1 new                    (0)11   01  110        # go to new if new scr > S
                                                       # else, we go to next pattern
jump3:
init r0, 6                      (0)000   0  101        # r3 = 6
ld r1, (r0)                     (0)001  01   00        # r1 = mem[6] (array ptr)
addi r1, 1                      (1)100  01   01        # r1++, go to next entry
str r1, (r0)                    (0)010  01   00        # mem[6] = ptr
beqR0 r0, loop                  ()11    00  001        # go to loop
jump2:
init r1, 3                      (0)000   1  010        # redundant instr to allow make jump same imm
beqR0 r0, jump3                 (1)11   00  011        # intermediate jump


equal:
init r0, 5                      (1)000   0  100        # r0 = 5
ld r1, (r0)                     (1)001  00   00        # r0 = mem[5] = C
addi r1, 1                      (1)100  01   01        # r1++ (count++)
str r1, (r0)                    (0)010  10   00        # mem[5] = r1
jump1:
beqR0, r0, jump2                (0)11   00  011        # intermediate jump

new:
init r1, 4                      (1)000   1  011        # r1 = 4
str r3, (r1)                    (0)010  11   01        # mem[4] = r3 (new score)
init r1, 5                      (0)000   1  100        # r1 = 5
init r0, 1                      (0)000   0  000        # r0 = 1
str r0, (r1)                    (0)010  00   01        # mem[5] = 1 (reset count)
beqR0, r0, jump1                (0)11   00  011        # intermediate jump
```

Python Disassembler Output for Program 1:



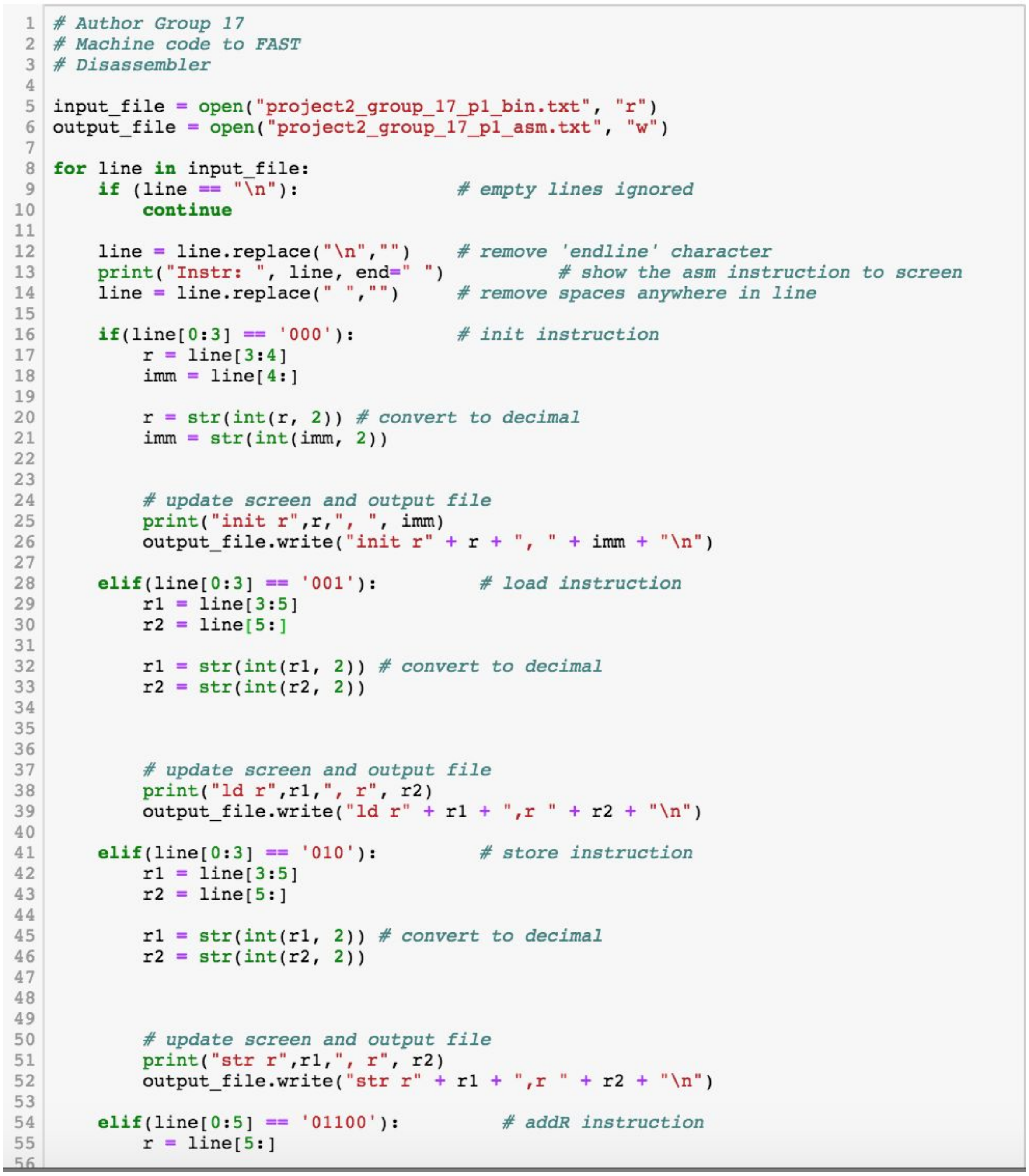
```
Instr:  1000101 addi r 1 , 1
Instr:  0001110 init r 1 ,  6
Instr:  0100001 str r 0 , r 1
Instr:  0001000 init r 1 ,  0
Instr:  0010101 ld r 1 , r 1
Instr:  1101111 beqR0 r 1 , 7
Instr:  0110011 addR r 3
Instr:  0111011 addR2 3
Instr:  0111110 addR3 2
Instr:  1001001 addi r 2 , 1
Instr:  0011010 ld r 2 , r 2
Instr:  0110110 subR3 2
Instr:  1001000 addi r 2 , 0
Instr:  1011110 sltR0 r 3 , r 2
Instr:  1000101 addi r 1 , 1
Instr:  1101100 beqR0 r 1 , 4
Instr:  1100010 beqR0 r 1 , 2
Instr:  1000101 addi r 1 , 1
Instr:  0101101 str r 3 , r 1
Instr:  0000110 init r 0 ,  6
Instr:  0010000 ld r 0 , r 0
Instr:  1000001 addi r 0 , 1
Instr:  1100000 beqR0 r 0 , 0
```

Python Disassembler Output for Program 2:

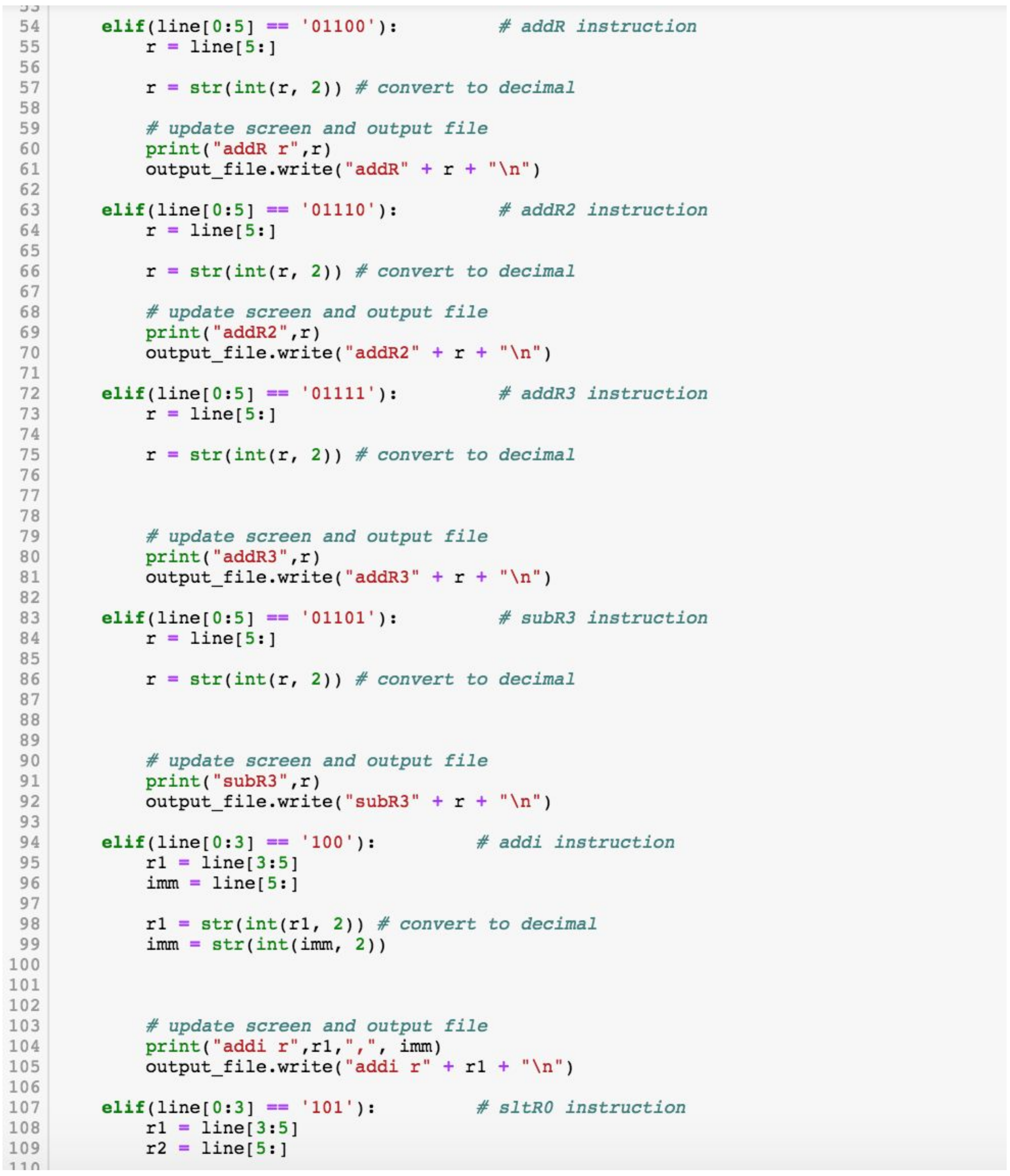


```
Instr:  0001010 init r 1 ,  2
Instr:  0011001 ld r 2 , r 1
Instr:  0001111 init r 1 ,  7
Instr:  0000101 init r 0 ,  5
Instr:  0100100 str r 1 , r 0
Instr:  0011101 ld r 3 , r 1
Instr:  1110111 beqR0 r 3 , 7
Instr:  0001011 init r 1 ,  3
Instr:  0010010 ld r 0 , r 2
Instr:  1111101 beqR0 r 0 , 5
Instr:  1010011 sltR0 r 0 , r 3
Instr:  0001000 init r 1 ,  0
Instr:  1101110 beqR0 r 0 , 6
Instr:  0000101 init r 0 ,  5
Instr:  0010100 ld r 1 , r 0
Instr:  1000101 addi r 1 , 1
Instr:  0100100 str r 1 , r 0
Instr:  1100001 beqR0 r 1 , 1
Instr:  0001010 init r 1 ,  2
Instr:  1100011 beqR0 r 1 , 3
Instr:  0000100 init r 0 ,  4
Instr:  0010000 ld r 0 , r 0
Instr:  1000101 addi r 1 , 1
Instr:  0101000 str r 2 , r 0
Instr:  1100011 beqR0 r 2 , 3
Instr:  0001011 init r 1 ,  3
Instr:  0101101 str r 3 , r 1
Instr:  0001100 init r 1 ,  4
Instr:  0000000 init r 0 ,  0
Instr:  0100001 str r 0 , r 1
Instr:  1100011 beqR0 r 0 , 3
```

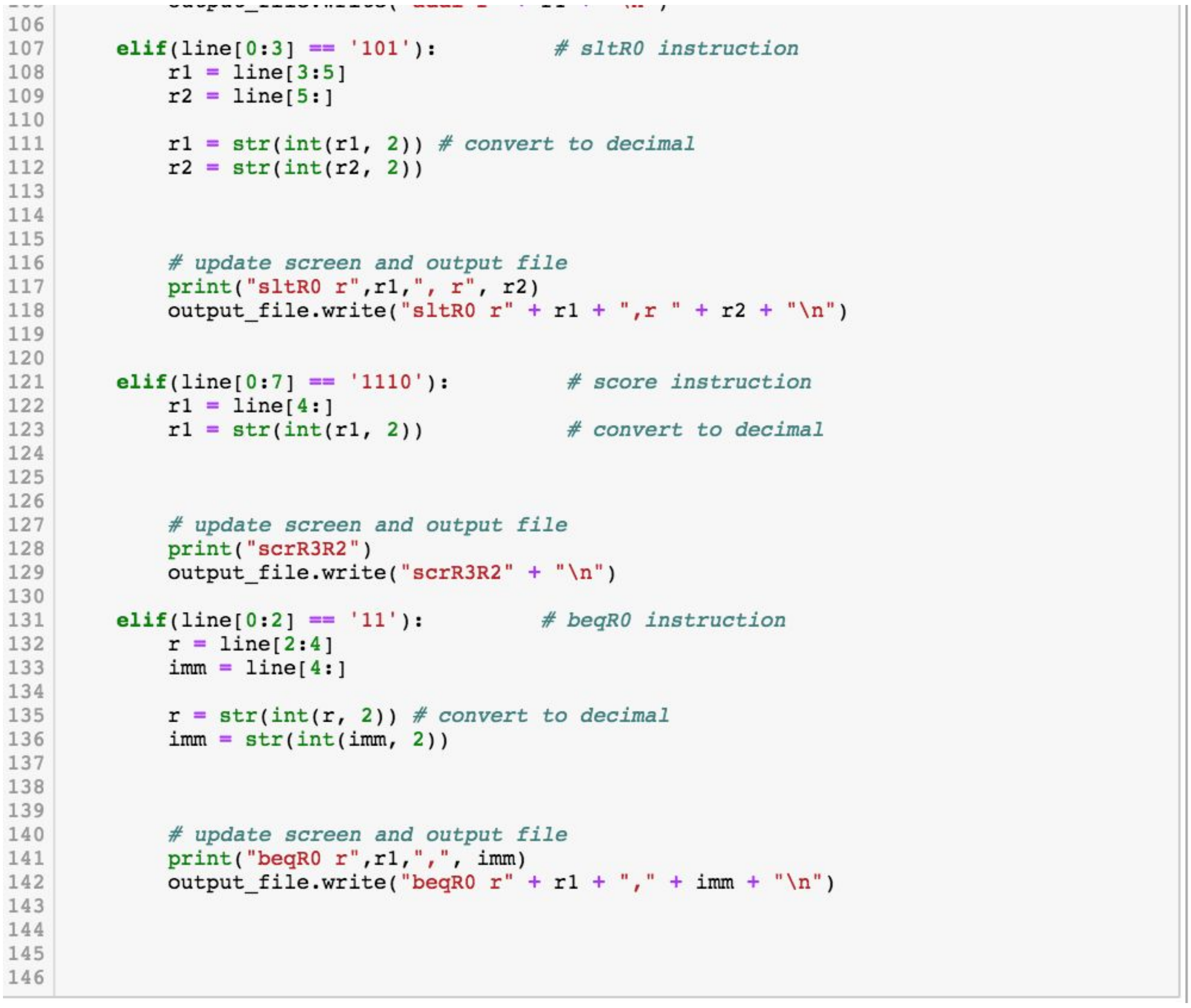
Python Code for Disassembler



```python
# Author Group 17
# Machine code to FAST
# Disassembler

input_file = open("project2_group_17_p1_bin.txt", "r")
output_file = open("project2_group_17_p1_asm.txt", "w")

for line in input_file:
    if (line == "\n"):                # empty lines ignored
        continue

    line = line.replace("\n","")      # remove 'endline' character
    print("Instr: ", line, end=" ")            # show the asm instruction to screen
    line = line.replace(" ","")       # remove spaces anywhere in line

    if(line[0:3] == '000'):           # init instruction
        r = line[3:4]
        imm = line[4:]

        r = str(int(r, 2)) # convert to decimal
        imm = str(int(imm, 2))


        # update screen and output file
        print("init r",r,", ", imm)
        output_file.write("init r" + r + ", " + imm + "\n")

    elif(line[0:3] == '001'):          # load instruction
        r1 = line[3:5]
        r2 = line[5:]

        r1 = str(int(r1, 2)) # convert to decimal
        r2 = str(int(r2, 2))



        # update screen and output file
        print("ld r",r1,", r", r2)
        output_file.write("ld r" + r1 + ",r " + r2 + "\n")

    elif(line[0:3] == '010'):          # store instruction
        r1 = line[3:5]
        r2 = line[5:]

        r1 = str(int(r1, 2)) # convert to decimal
        r2 = str(int(r2, 2))



        # update screen and output file
        print("str r",r1,", r", r2)
        output_file.write("str r" + r1 + ",r " + r2 + "\n")

    elif(line[0:5] == '01100'):          # addR instruction
        r = line[5:]
```



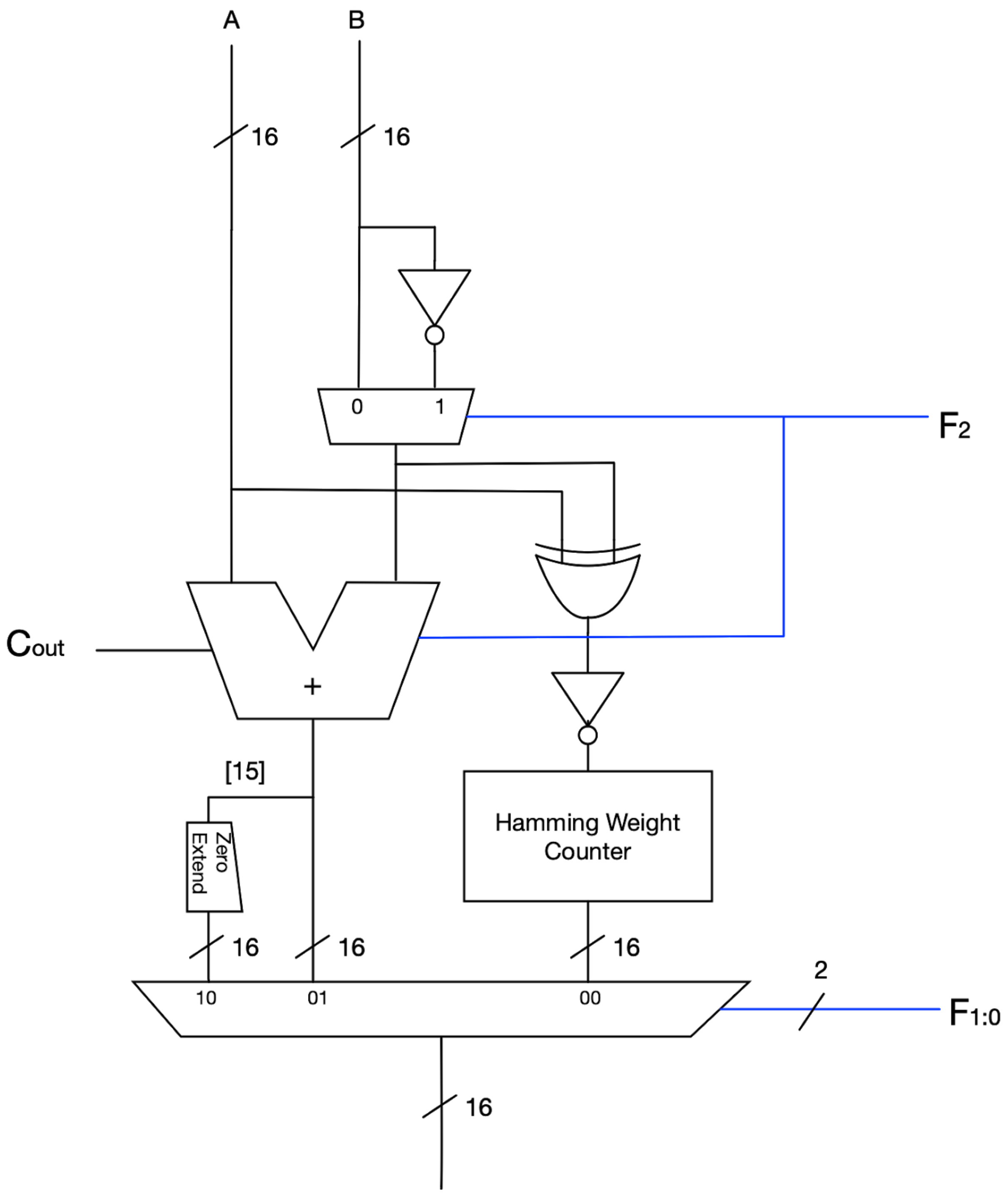
```python
    elif(line[0:5] == '01100'):         # addR instruction
        r = line[5:]

        r = str(int(r, 2)) # convert to decimal

        # update screen and output file
        print("addR r",r)
        output_file.write("addR" + r + "\n")

    elif(line[0:5] == '01110'):         # addR2 instruction
        r = line[5:]

        r = str(int(r, 2)) # convert to decimal

        # update screen and output file
        print("addR2",r)
        output_file.write("addR2" + r + "\n")

    elif(line[0:5] == '01111'):         # addR3 instruction
        r = line[5:]

        r = str(int(r, 2)) # convert to decimal



        # update screen and output file
        print("addR3",r)
        output_file.write("addR3" + r + "\n")

    elif(line[0:5] == '01101'):         # subR3 instruction
        r = line[5:]

        r = str(int(r, 2)) # convert to decimal



        # update screen and output file
        print("subR3",r)
        output_file.write("subR3" + r + "\n")

    elif(line[0:3] == '100'):         # addi instruction
        r1 = line[3:5]
        imm = line[5:]

        r1 = str(int(r1, 2)) # convert to decimal
        imm = str(int(imm, 2))



        # update screen and output file
        print("addi r",r1,",", imm)
        output_file.write("addi r" + r1 + "\n")

    elif(line[0:3] == '101'):         # sltR0 instruction
        r1 = line[3:5]
        r2 = line[5:]
```



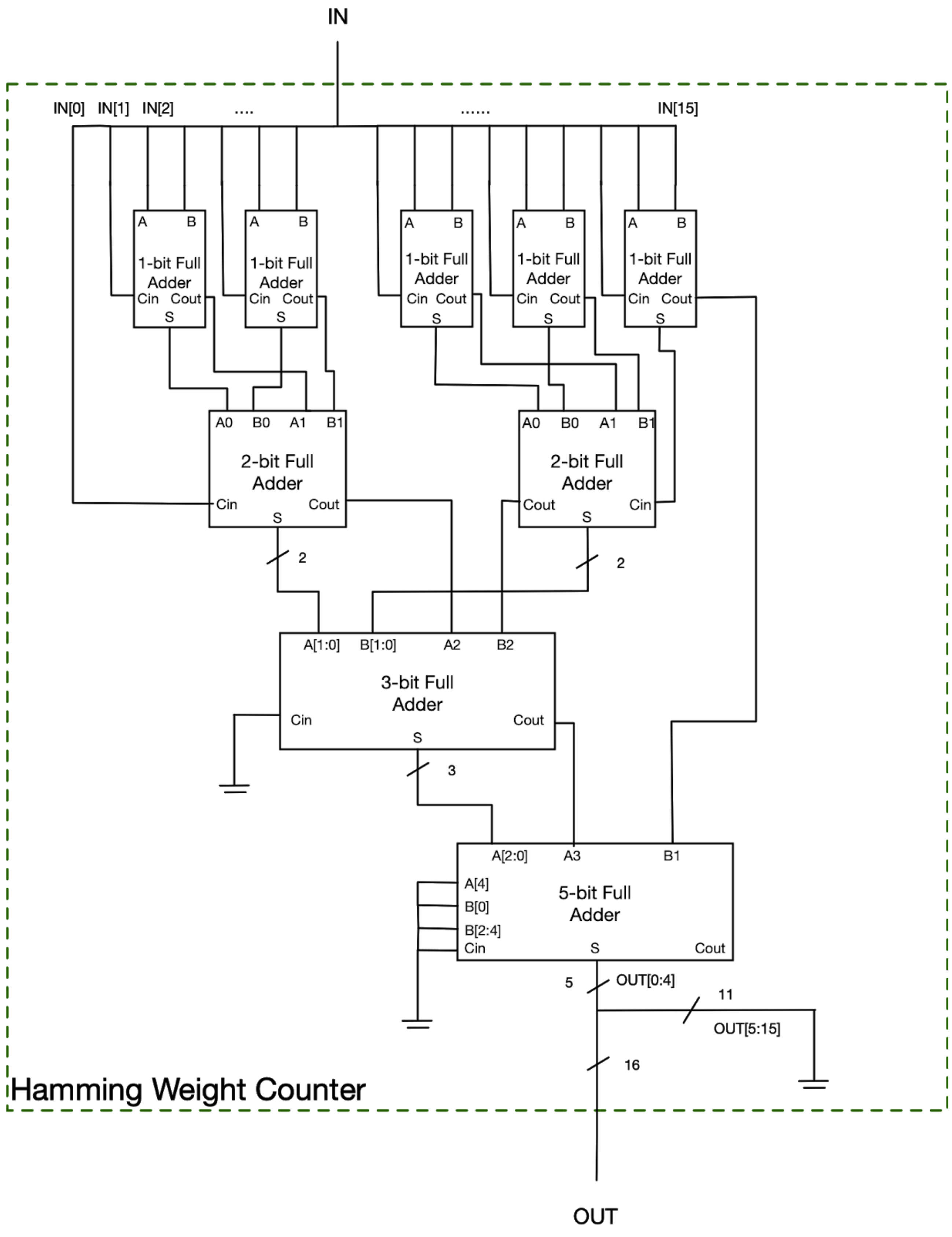
```
    elif(line[0:3] == '101'):          # sltR0 instruction
        r1 = line[3:5]
        r2 = line[5:]

        r1 = str(int(r1, 2)) # convert to decimal
        r2 = str(int(r2, 2))



        # update screen and output file
        print("sltR0 r",r1,", r", r2)
        output_file.write("sltR0 r" + r1 + ",r " + r2 + "\n")


    elif(line[0:7] == '1110'):          # score instruction
        r1 = line[4:]
        r1 = str(int(r1, 2))            # convert to decimal



        # update screen and output file
        print("scrR3R2")
        output_file.write("scrR3R2" + "\n")

    elif(line[0:2] == '11'):          # beqR0 instruction
        r = line[2:4]
        imm = line[4:]

        r = str(int(r, 2)) # convert to decimal
        imm = str(int(imm, 2))



        # update screen and output file
        print("beqR0 r",r1,",", imm)
        output_file.write("beqR0 r" + r1 + "," + imm + "\n")
```

Part D. Hardware Implementation

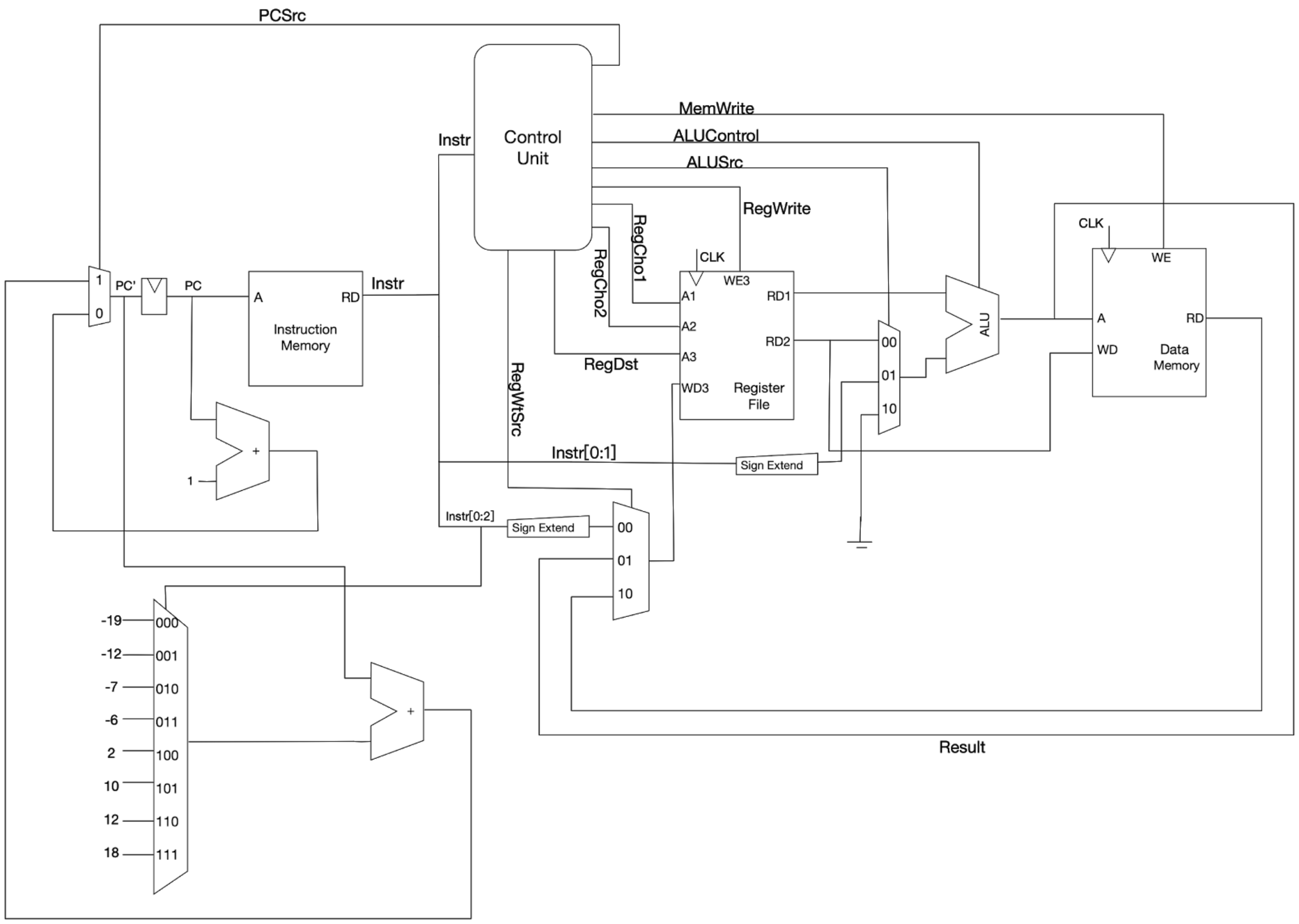
1. ALU schematic



Hamming Weight Counter

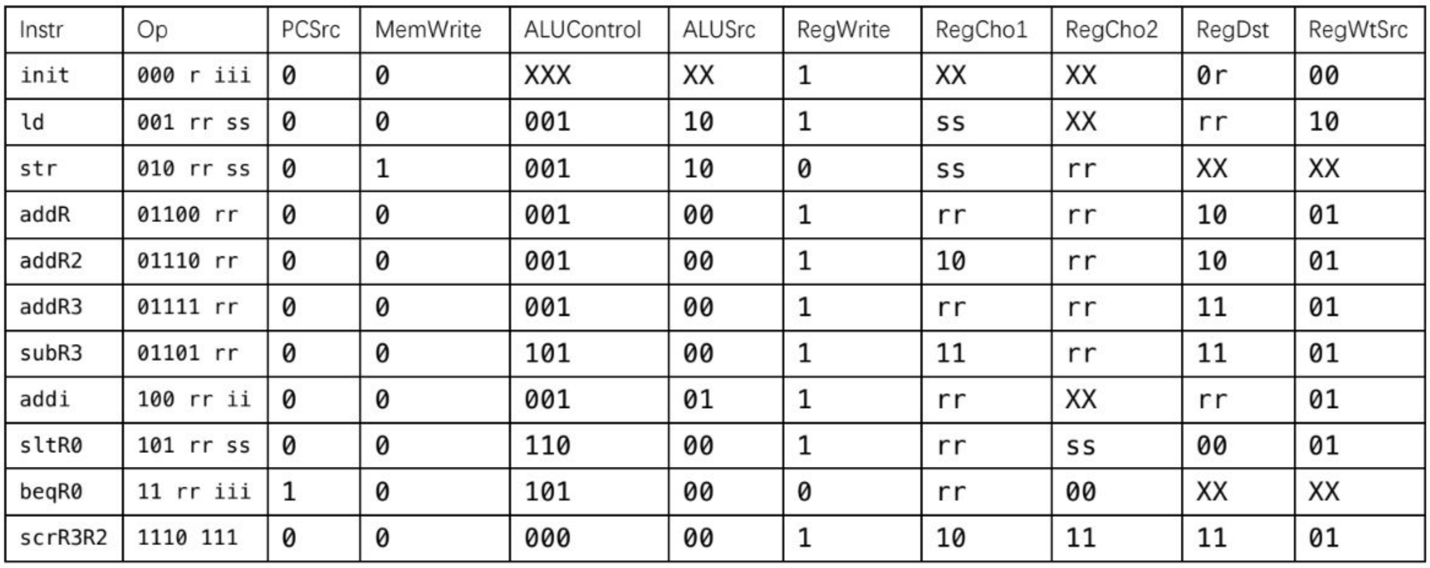


2. CPU Datapath



3. Control Logic

Truth Table for control unit:



Instr	Op	PCSrc	MemWrite	ALUControl	ALUSrc	RegWrite	RegCho1	RegCho2	RegDst	RegWtSrc
init	000 r iii	0	0	XXX	XX	1	XX	XX	0r	00
ld	001 rr ss	0	0	001	10	1	ss	XX	rr	10
str	010 rr ss	0	1	001	10	0	ss	rr	XX	XX
addR	01100 rr	0	0	001	00	1	rr	rr	10	01
addR2	01110 rr	0	0	001	00	1	10	rr	10	01
addR3	01111 rr	0	0	001	00	1	rr	rr	11	01
subR3	01101 rr	0	0	101	00	1	11	rr	11	01
addi	100 rr ii	0	0	001	01	1	rr	XX	rr	01
sltR0	101 rr ss	0	0	110	00	1	rr	ss	00	01
beqR0	11 rr iii	1	0	101	00	0	rr	00	XX	XX
scrR3R2	1110 111	0	0	000	00	1	10	11	11	01