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: [1,4] ; ex. 00 = 1 | 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 will be used as select line in a MUX to choose program specific jumps | beqR0 R0, brk  e.g. brk = 15  101 can be the select for pc+=15 | 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 |

##### 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 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:

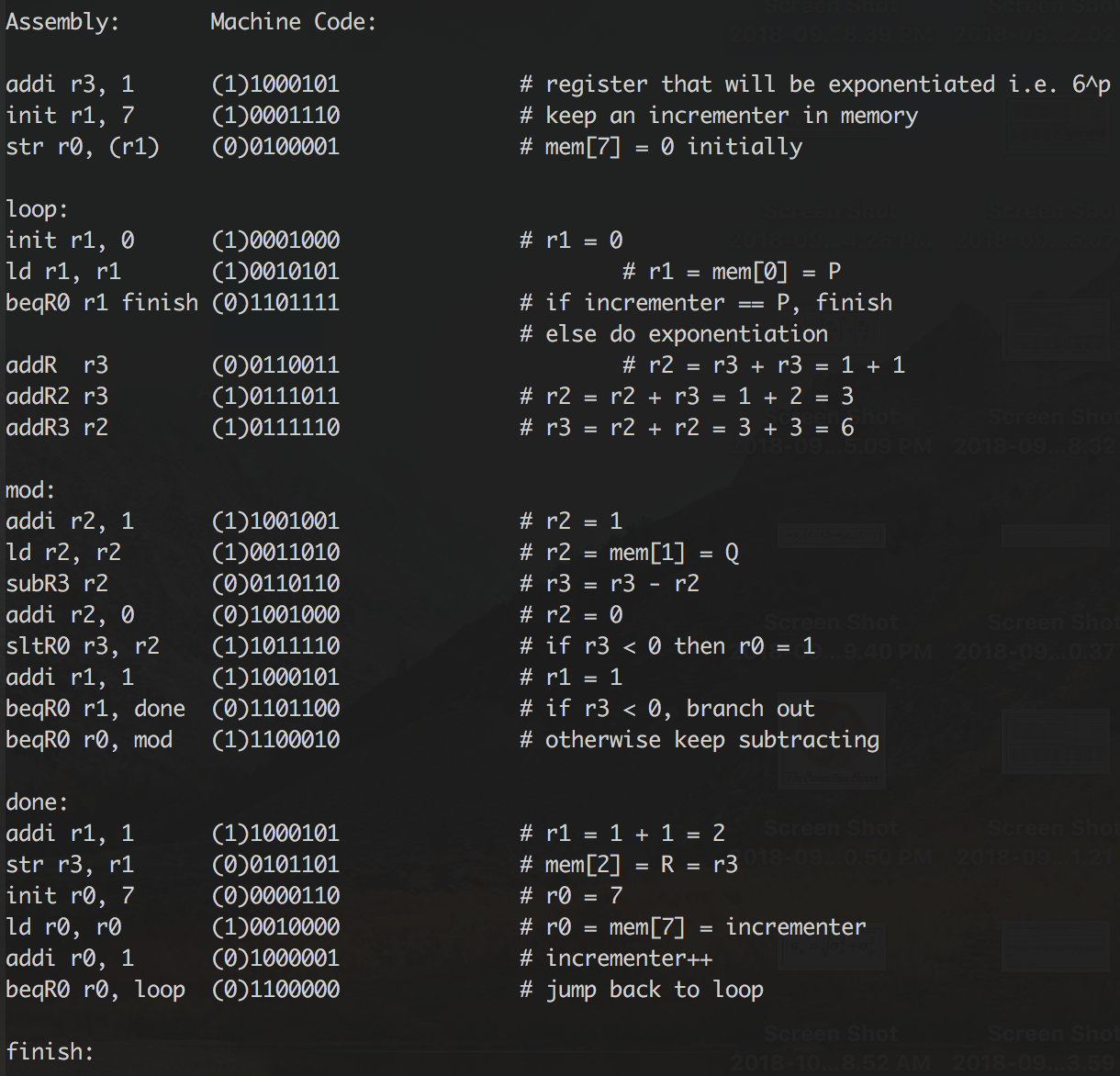
We worked together in the library to brainstorm ideas for the ISA. We then collaborated for program 1, doing revisions. Similar for program 2. We met the first progress milestone and second, as well. Working time was spent half together half remote but with relaying of ideas and checkpoints.

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

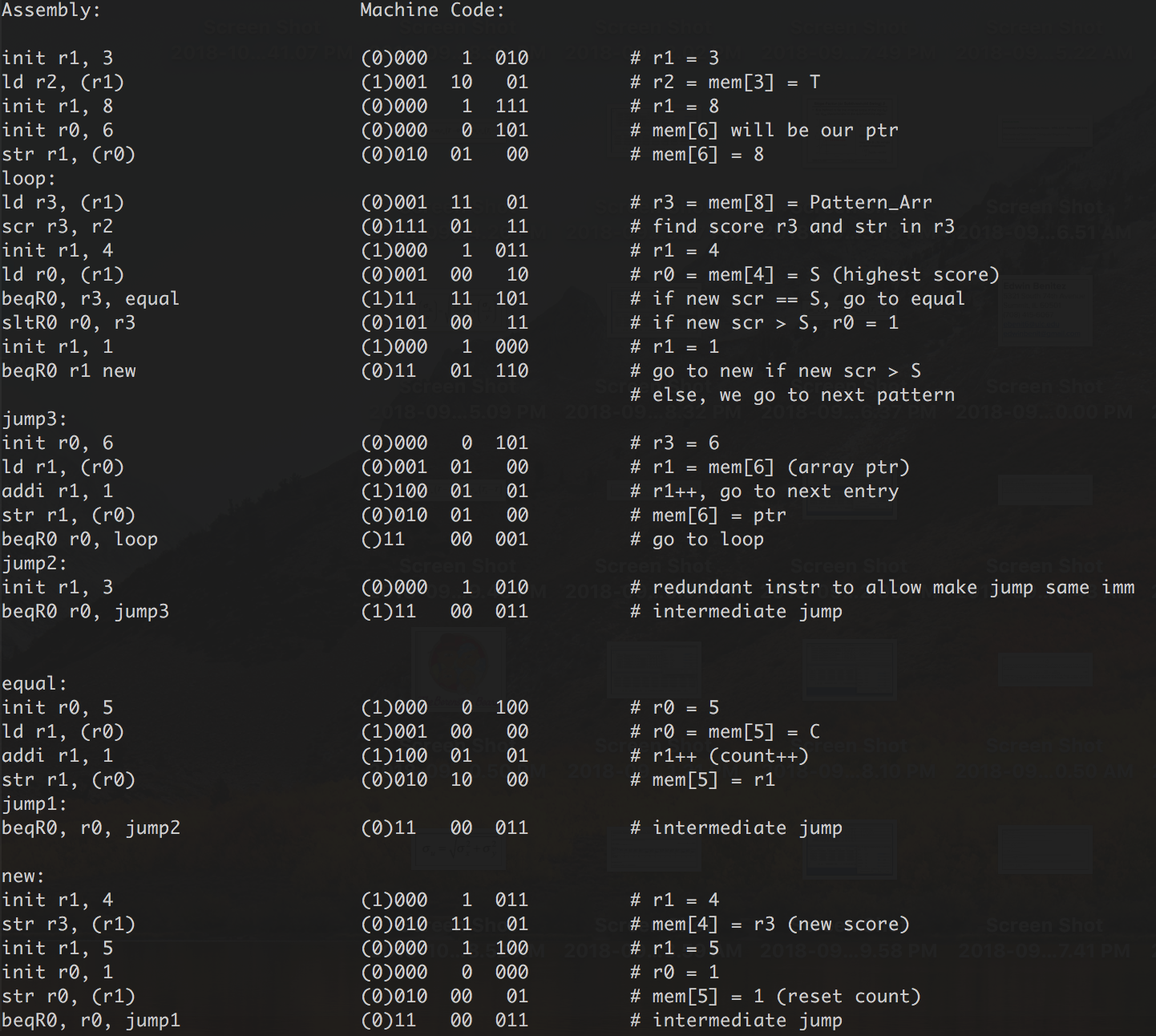
Since we know exactly what the program needs in terms of unique instructions we can have utilize unused instructions. We can also try to use more hardware implementations of instructions to lower dynamic instruction count. For example, we can condense the add instructions to just an exponentiator.

### Part C. Software Package

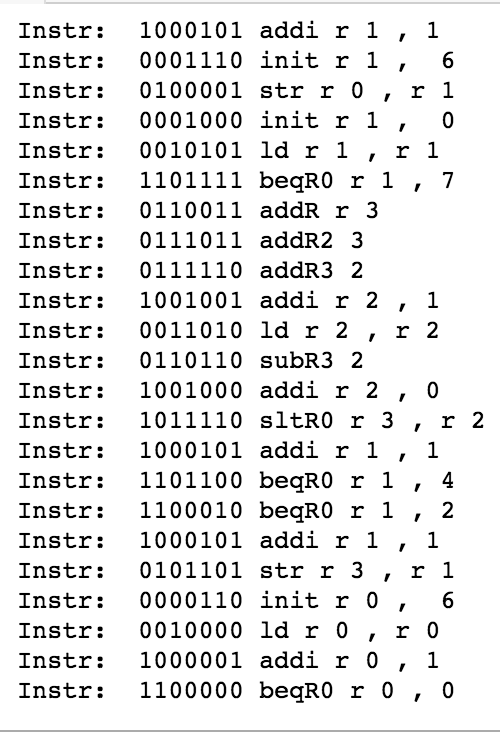
**Program 1 Algorithm & Machine Code**



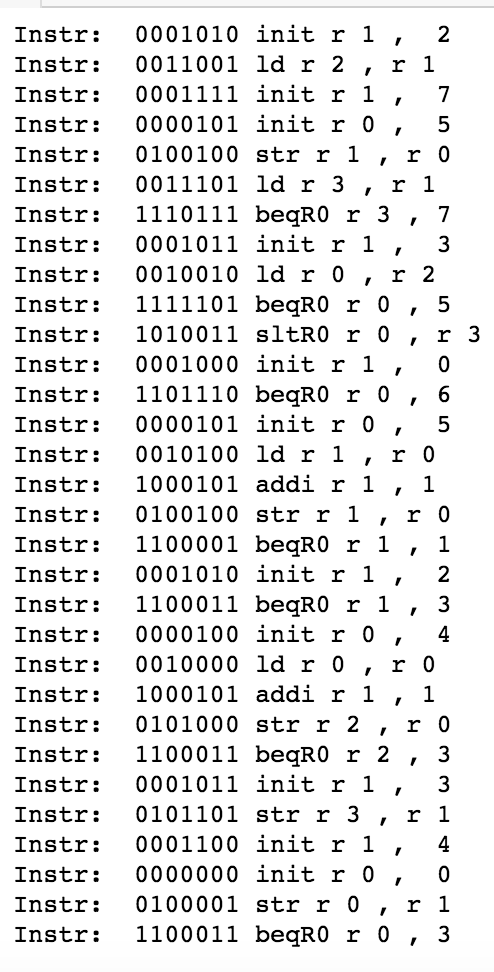
**Program 2 Algorithm & Machine Code**



**Python Disassembler Output for Program 1:**

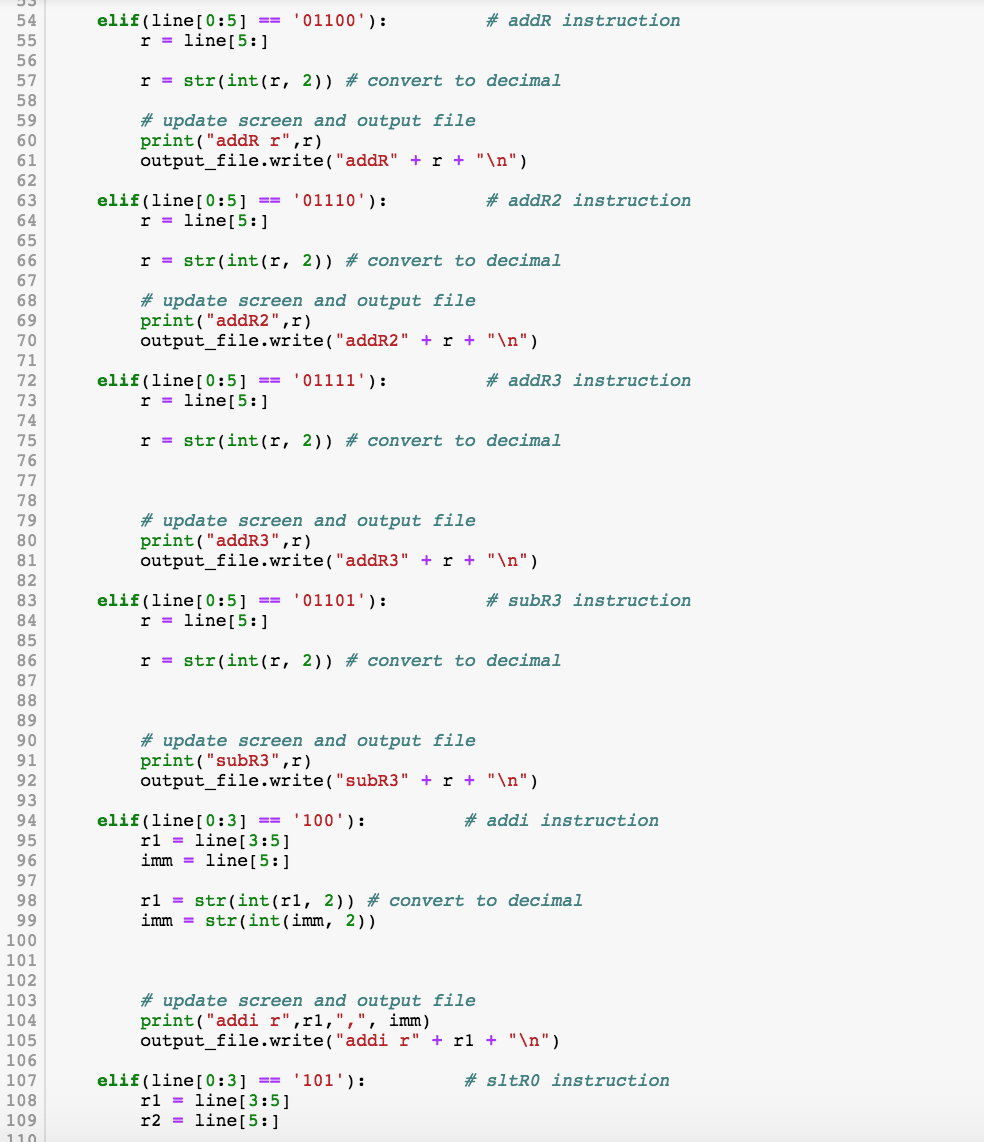
****

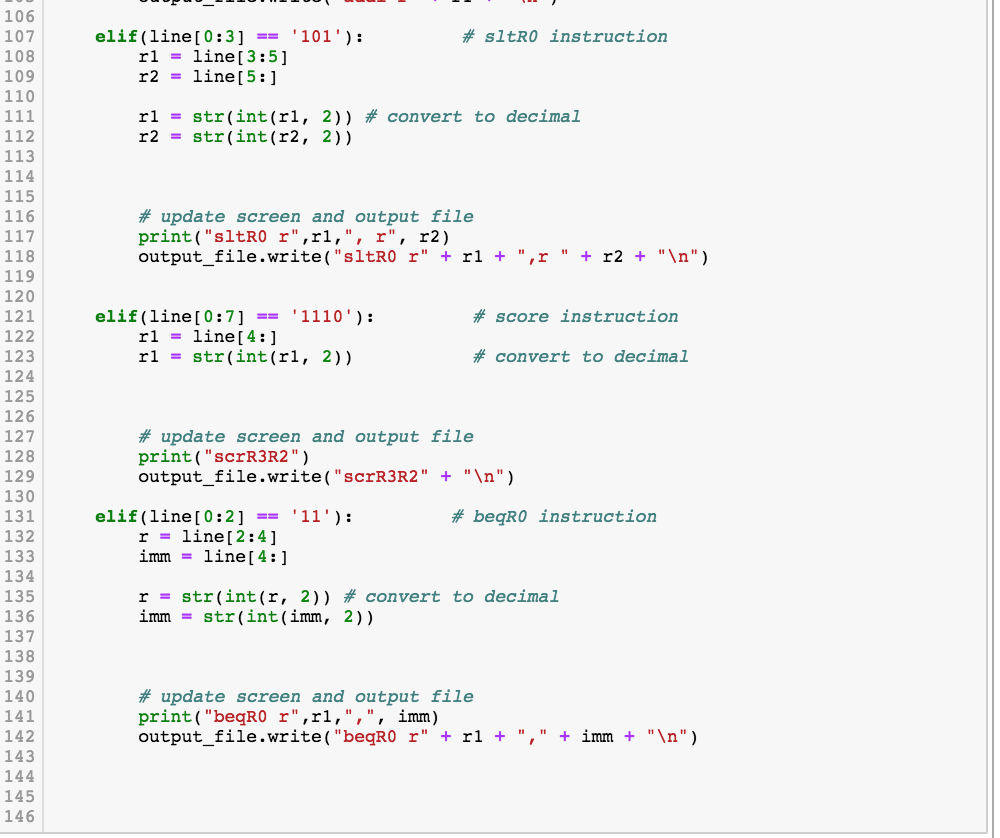
**Python Disassembler Output for Program 2:**

****

### Python Code for Disassembler

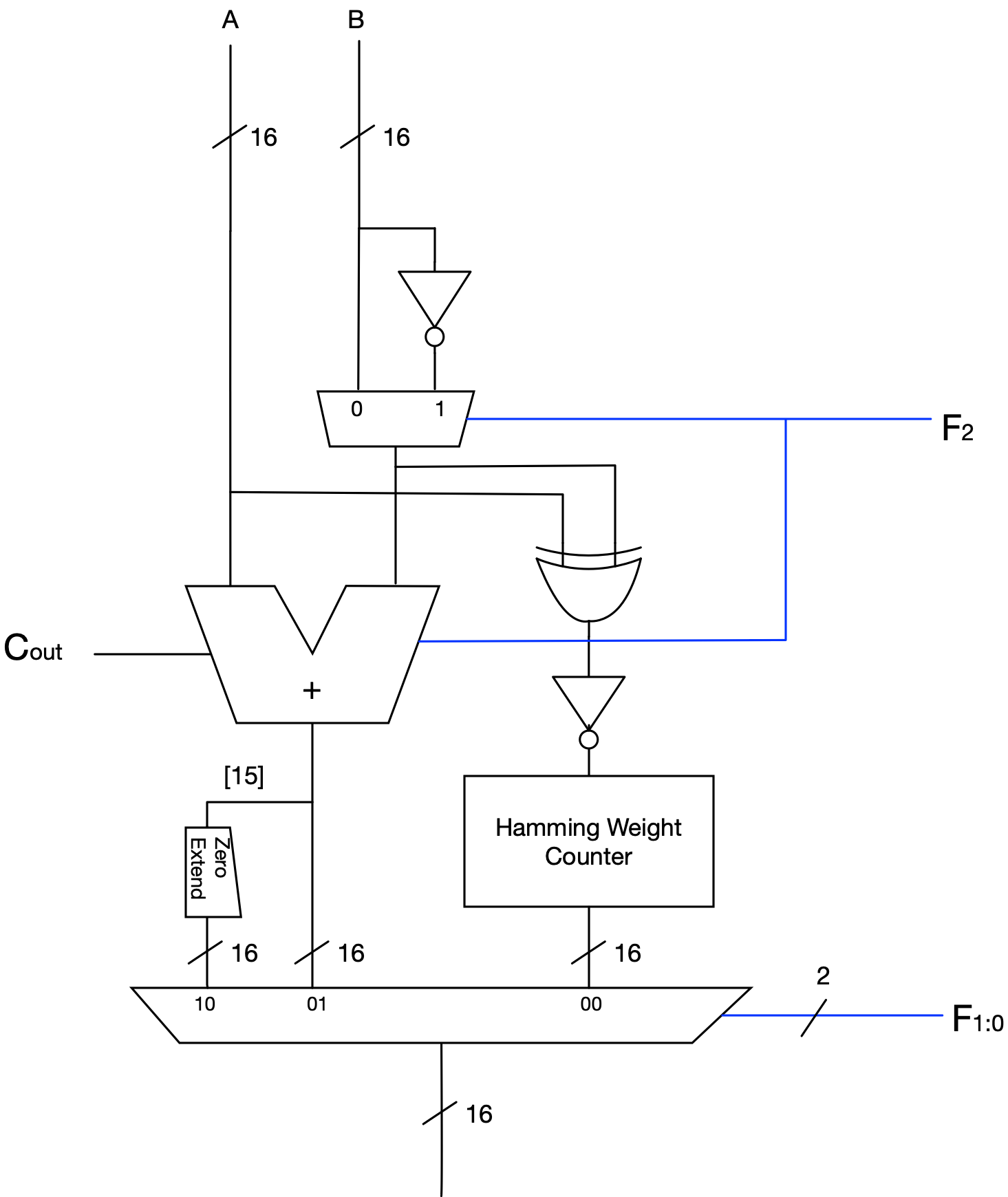
### 



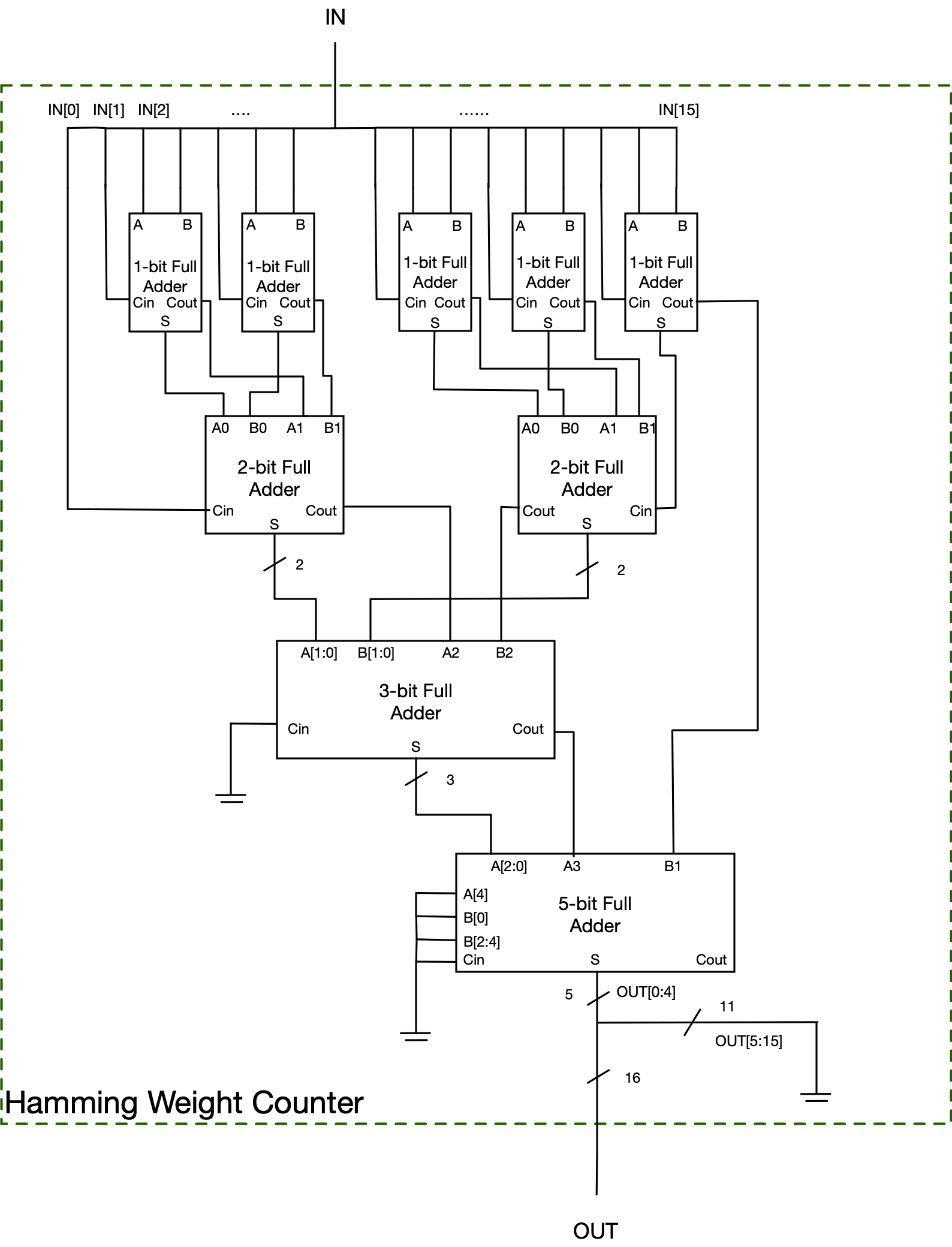


### Part D. Hardware Implementation

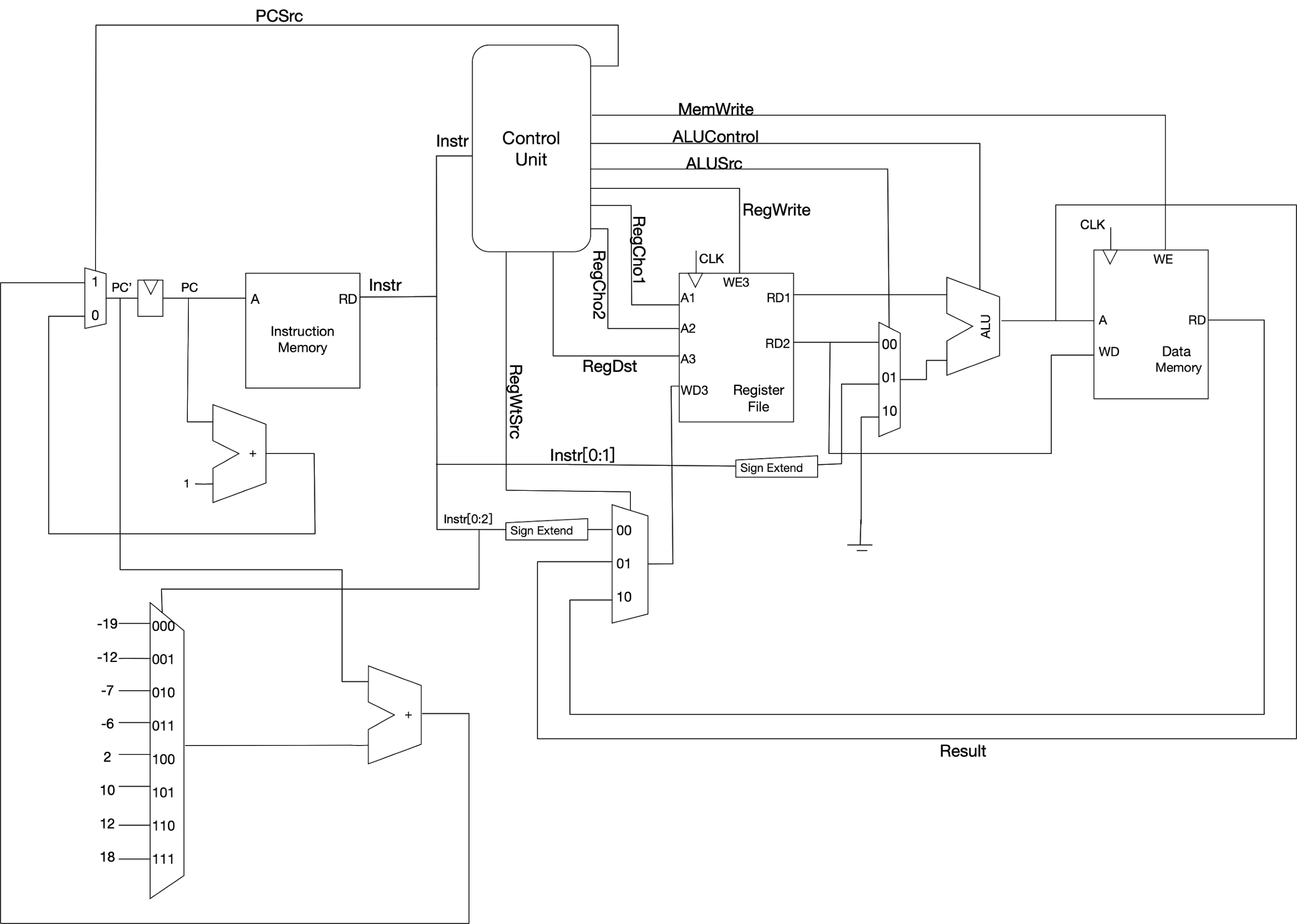
##### 1. ALU schematic



**Hamming Weight Counter**



##### 2. CPU Datapath



##### 

##### 3. Control Logic

**Truth Table for control unit:**

