Project 2

ECE 366: MW 4:30 pm

Sanjida Choudhury, Michael Dritlein, Branimir Maximov

Professor: Wenjing Rao

Part A: ISA Intro

1. Introduction:

Our architecture is the Limited Instruction Set. Our philosophy was to utilize multiple sizings of op-codes to allow for more instructions. We also created more specific and niche instructions to ensure both programs can function correctly. Our goal was to create instructions such that each instruction will have a distinct bit pattern. We also strived to use only instructions that are necessary. This was done by optimizing the instruction list and an iterative process of improving our implementation of the programs. This iterative process helped us to understand which instructions were necessary to keep when completing the programs.

2. Instruction List:

Instr	Format	Op	Example	
ADD	100 xx, yy Rx Ry	100	ADD r3, r2	r3 = r3 + r2
LWD	001 xx, yy Rx Ry	001	LWD r0, [r1]	r0 = M[r1]
SWD	011 xx, yy Rx Ry	011	SWD r1, [r2]	M[r2] = r1
INIT	101 xx, yy Rx Ry	101	INIT r0, 1	r0 = 1
ADDI	111 xx, yy Rx Ry	111	ADDI r0, 2	r0 = r0 + 2
SLE	110 xx, yy	110	SLE r1, r2	If r1 < r2

	Rx Ry			r3 = 1 Else r3 = 0
JIF	010 xxxx imm	010	JIF -7	If r3 = 1 => PC = PC + imm Else PC = PC + 1
XOR	0001 x, yy Rx Ry	0001	XOR r1, r2	r1 ^= r2
SLER	0000 xx, y Rx Ry (Ry cannot be r1) (Rx must be r0)	0000	SLER r1, r0	If r1 < r0 r3 = 1 Else r3 = 0
SUBR0	00000 yy Ry	00000	SUBR0 r2	r0 = r0 - r2
ADDN	1111100	1111100	ADDN	r3 = r3 - 1
CNTR0	0001000	0001000	CNTR0	Count the number of bits with a value of '1' in r0
HLT	0001111	0001111	HLT	Stop the program

3. Register Design:

There are 4 registers supported. The registers are mostly multi-purpose except r3 is used as the register to store the result of SLE and SLER. r3 is also the register that is checked to determine if a jump occurs or not (JIF). Some registers cannot be used for certain instructions or are the only register that can be used as one of the operands.

4. Control Flow (Branches):

The only branching instruction is JIF. This checks if the value in r3 is 1, then branches if so. If the value is anything else, then nothing is done. There is no additional calculation for this instruction, an immediate is sent to an adder that sums the immediate with PC. This supports a jump of up to 7 or -7.

Example: JIF 5

Machine Code: 0100101

This example jumps 5 instructions ahead if r3 = 1.

5. **Data memory addressing Modes:**

Load word and store word are supported for data memory. There is no additional calculation, a register is used to load to memory.

Example: LWD r2, [r3]

Machine Code: 0011011

This loads the contents of register r2 into the memory location corresponding to the value in r3.

Example: SWD r0, [r1]

Machine Code: 0110001

This stores the contents of r0 into the memory location corresponding to the value in r1.

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.**

   Our ISA has an unique jump functionality where the register r3 is checked to determine to jump or not. This allows for a greater immediate value in the JIF instruction. We have a unique CNTR0 instruction that counts the number of '1's in a 16 bit value. We have a SLE (set less than or equal to) than can be used on immediates and a SLER that can be used on registers as well. We have a ADDN instruction that subtracts 1 from r3 specifically. Our SUBR0 instruction specifically subtracts an immediate from r0.

   Our ISA is better because it uses less bits and allows for parity checking within 8 bits. To use less bits we make our instructions more specific and optimized for the two programs.

2. **In what ways did you optimize for the two goals? If you optimized for anything additional, what and how?**

   We created an instruction CNTR0 that is only used to count the number of '1's in r0. This reduces the number of instructions needed in program 2. We also created a ADDN instruction to subtract 1 from r3. We created a SUBR0 instruction that subtracts only from r0 and a SLER instruction that checks if a register is less than or equal to r0. Our JIF only takes an immediate and utilizes the value currently in r3 to determine to jump or not. This allows for a greater immediate value. By making these instructions more specific, we can keep bit patterns unique

while adding the necessary instructions to complete the two programs.

3. **What would you have done differently if you had 1 more bit for instructions? How about 1 fewer bit?**
   If we had one more bit for instructions, we would make more generic instructions and increase immediate values. This would allow for less instructions since we would need less specific instructions. If we had one fewer bit, we would make even more specific instructions and take care of more functionality within the hardware.

4. **How did your team work together to accomplish this project? (Role of each team member, progress milestones, time spent individually and together?)**
   How we accomplished this project together was we made a Slack group chat to communicate, and met in the CS Lounge to work on the project together. We worked on the project in total about 20 hours.

5. **If you had a chance to restart this project afresh with 3 weeks' time, how would your team have done differently?**
   If we had the chance to restart this project afresh with three weeks time, we would have made the program more efficient than the previous. We would use a Hamming distance algorithm for program 2 to lower dynamic instruction count. We would also experiment with new and differently formatted instructions to attempt optimization of our current programs.

Part C: Software Package

1. **Algorithms of the two programs. Make sure your assembly format is either obvious or well described, and that the code is well commented.**

Program 1:

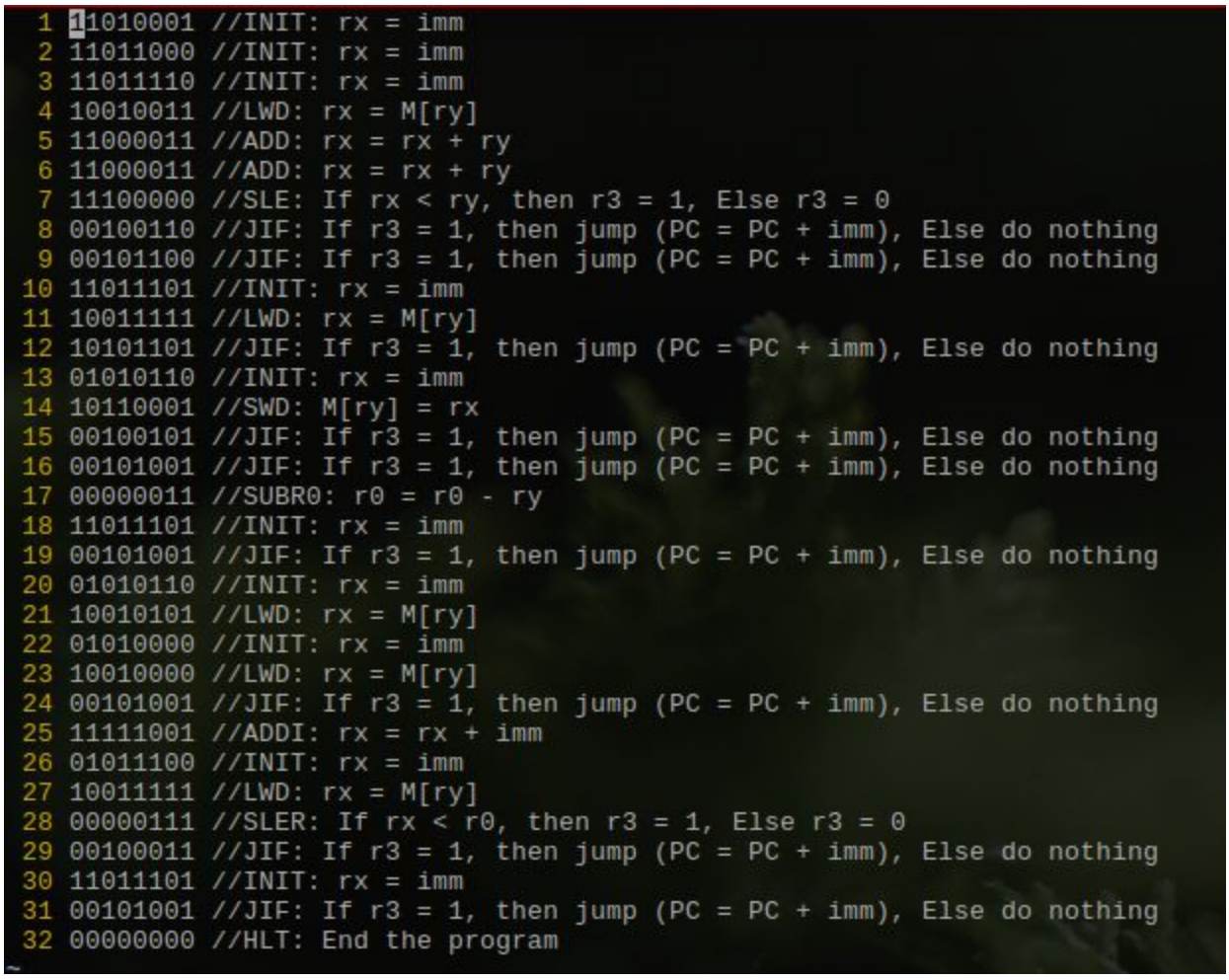
We compute 6^P mod Q by first adding 3 twice and then computing the modulus. The result of the modulus is carried over into the next calculation so that we have
((6^P mod Q) * 6^P) mod Q
And this is continued for each power until the desired power is reached.

Program 2:

We XOR the target pattern with each element in the pattern array. Then, the number of ones in the result is counted through hardware with the specialized instruction CNTR0. Next, the number of ones counted is checked against the current best match. If the number of ones is greater, a new best match is stored in memory, if less we skip to the next loop. If the number of ones is equal to the best match, we add one to the best match count.

2. **Machine Code for each of the Programs.**

Program 1 Machine Code



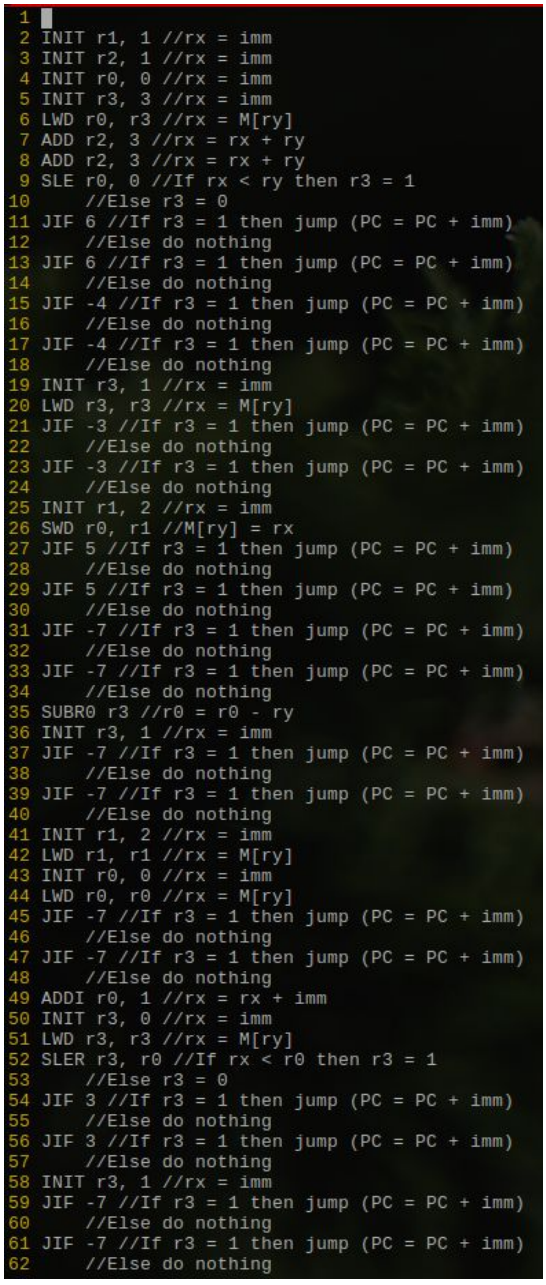
```
 1 11010001 //INIT: rx = imm
 2 11011000 //INIT: rx = imm
 3 11011110 //INIT: rx = imm
 4 10010011 //LWD: rx = M[ry]
 5 11000011 //ADD: rx = rx + ry
 6 11000011 //ADD: rx = rx + ry
 7 11100000 //SLE: If rx < ry, then r3 = 1, Else r3 = 0
 8 00100110 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
 9 00101100 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
10 11011101 //INIT: rx = imm
11 10011111 //LWD: rx = M[ry]
12 10101101 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
13 01010110 //INIT: rx = imm
14 10110001 //SWD: M[ry] = rx
15 00100101 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
16 00101001 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
17 00000011 //SUBR0: r0 = r0 - ry
18 11011101 //INIT: rx = imm
19 00101001 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
20 01010110 //INIT: rx = imm
21 10010101 //LWD: rx = M[ry]
22 01010000 //INIT: rx = imm
23 10010000 //LWD: rx = M[ry]
24 00101001 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
25 11111001 //ADDI: rx = rx + imm
26 01011100 //INIT: rx = imm
27 10011111 //LWD: rx = M[ry]
28 00000111 //SLER: If rx < r0, then r3 = 1, Else r3 = 0
29 00100011 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
30 11011101 //INIT: rx = imm
31 00101001 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
32 00000000 //HLT: End the program
```

Program 2 Machine Code

```
 1 11010001 //INIT: rx = imm
 2 11010100 //INIT: rx = imm
 3 10110001 //SWD: M[ry] = rx
 4 01010011 //INIT: rx = imm
 5 01110010 //ADDI: rx = rx + imm
 6 01010101 //INIT: rx = imm
 7 10110001 //SWD: M[ry] = rx
 8 01010011 //INIT: rx = imm
 9 01110001 //ADDI: rx = rx + imm
10 01010110 //INIT: rx = imm
11 10110001 //SWD: M[ry] = rx
12 11010001 //INIT: rx = imm
13 01010110 //INIT: rx = imm
14 10011001 //LWD: rx = M[ry]
15 01111011 //ADDI: rx = rx + imm
16 10110001 //SWD: M[ry] = rx
17 01010000 //INIT: rx = imm
18 01010101 //INIT: rx = imm
19 111100101 //ADDI: rx = rx + imm
20 11110101 //ADDI: rx = rx + imm
21 01100110100 //SLE: If rx < ry, then r3 = 1, Else r3 = 0
22 10101101 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
23 111101100 //ADDI: rx = rx + imm
24 01011010 //INIT: rx = imm
25 10011010 //LWD: rx = M[ry]
26 01111011 //ADDI: rx = rx + imm
27 10110010 //SWD: M[ry] = rx
28 11010001 //INIT: rx = imm
29 00011000 //LWD: rx = M[ry]
30 01010011 //INIT: rx = imm
31 10011100 //LWD: rx = M[ry]
32 00010010 //LWD: rx = M[ry]
33 10001011 //XOR: rx = rx XOR ry
34 00001000 //CNTR0: Count the number of 1's in r0
35 11010010 //INIT: rx = imm
36 00101010 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
37 00011000 //LWD: rx = M[ry]
38 10111110 //SWD: M[ry] = rx
39 11010010 //INIT: rx = imm
40 00011000 //LWD: rx = M[ry]
41 11111001 //ADDI: rx = rx + imm
42 00010010 //LWD: rx = M[ry]
43 00101001 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
44 11101100 //SLE: If rx < ry, then r3 = 1, Else r3 = 0
45 10100100 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
46 10111110 //SWD: M[ry] = rx
47 11011101 //INIT: rx = imm
48 10100111 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
49 01111100 //ADDN: r3 = r3 - 1
50 00101001 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
51 11100011 //SLE: If rx < ry, then r3 = 1, Else r3 = 0
52 00100011 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
53 110100100 //INIT: rx = imm
54 00011000 //LWD: rx = M[ry]
55 00100110 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
56 11111010 //ADDI: rx = rx + imm
57 00101001 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
58 00011110 //LWD: rx = M[ry]
59 01111101 //ADDI: rx = rx + imm
60 10111110 //SWD: M[ry] = rx
61 11010001 //INIT: rx = imm
62 00011000 //LWD: rx = M[ry]
63 00101010 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
64 11111001 //ADDI: rx = rx + imm
65 11010010 //INIT: rx = imm
66 10011100 //LWD: rx = M[ry]
67 11111111 //ADDI: rx = rx + imm
68 10011111 //LWD: rx = M[ry]
69 10000101 //SLER: If rx < r0, then r3 = 1, Else r3 = 0
70 00101001 //JIF: If r3 = 1, then jump (PC = PC + imm), Else do nothing
71 00000000 //HLT: End the program
```

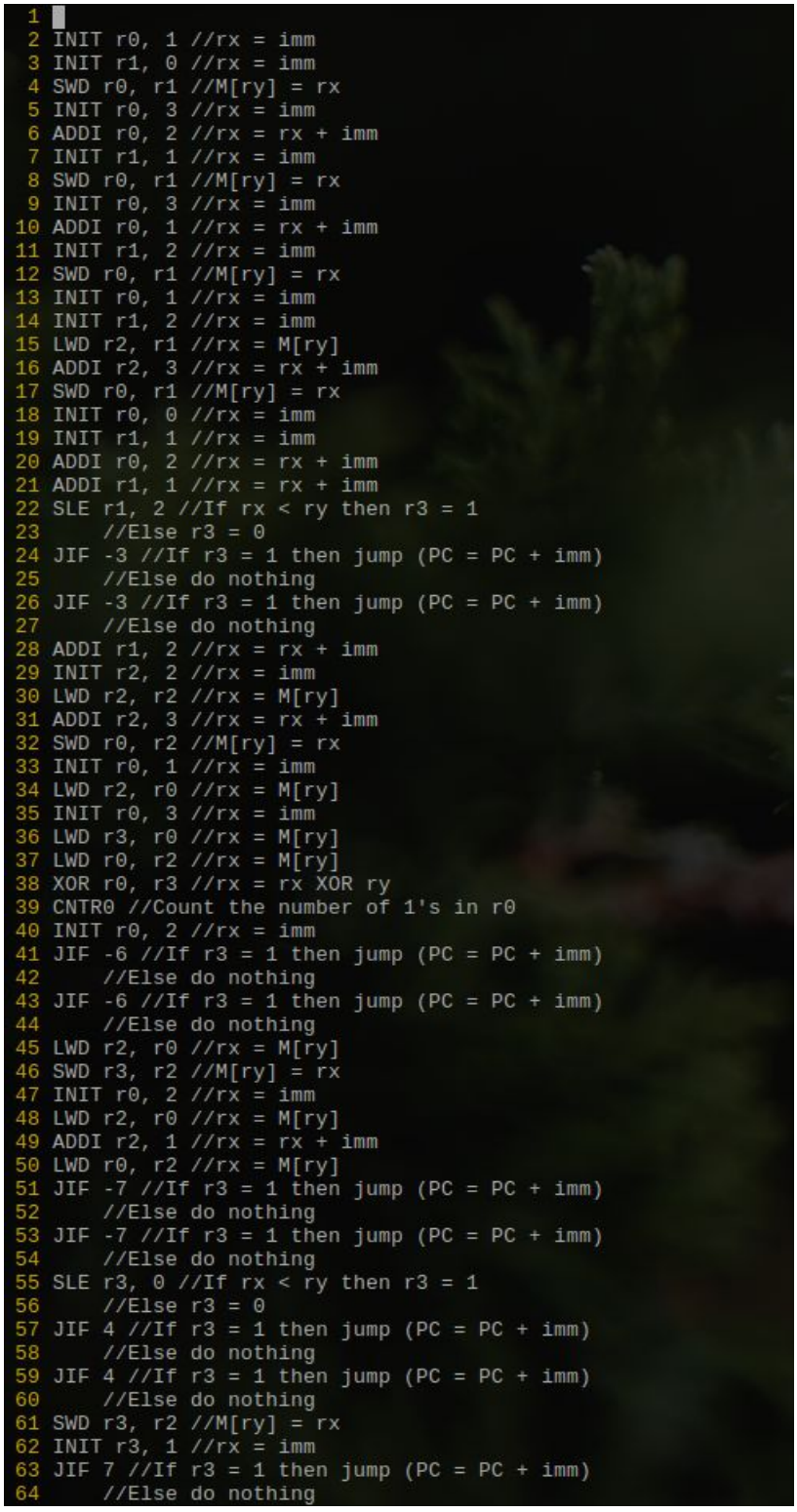
3. **Output of your Python disassembler for each program. This should be a line-by-line explanation of the machine code, what is done by each line of code.**

Program 1 Disassembler Output



```
 1
 2 INIT r1, 1 //rx = imm
 3 INIT r2, 1 //rx = imm
 4 INIT r0, 0 //rx = imm
 5 INIT r3, 3 //rx = imm
 6 LWD r0, r3 //rx = M[ry]
 7 ADD r2, 3 //rx = rx + ry
 8 ADD r2, 3 //rx = rx + ry
 9 SLE r0, 0 //If rx < ry then r3 = 1
10     //Else r3 = 0
11 JIF 6 //If r3 = 1 then jump (PC = PC + imm)
12     //Else do nothing
13 JIF 6 //If r3 = 1 then jump (PC = PC + imm)
14     //Else do nothing
15 JIF -4 //If r3 = 1 then jump (PC = PC + imm)
16     //Else do nothing
17 JIF -4 //If r3 = 1 then jump (PC = PC + imm)
18     //Else do nothing
19 INIT r3, 1 //rx = imm
20 LWD r3, r3 //rx = M[ry]
21 JIF -3 //If r3 = 1 then jump (PC = PC + imm)
22     //Else do nothing
23 JIF -3 //If r3 = 1 then jump (PC = PC + imm)
24     //Else do nothing
25 INIT r1, 2 //rx = imm
26 SWD r0, r1 //M[ry] = rx
27 JIF 5 //If r3 = 1 then jump (PC = PC + imm)
28     //Else do nothing
29 JIF 5 //If r3 = 1 then jump (PC = PC + imm)
30     //Else do nothing
31 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
32     //Else do nothing
33 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
34     //Else do nothing
35 SUBR0 r3 //r0 = r0 - ry
36 INIT r3, 1 //rx = imm
37 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
38     //Else do nothing
39 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
40     //Else do nothing
41 INIT r1, 2 //rx = imm
42 LWD r1, r1 //rx = M[ry]
43 INIT r0, 0 //rx = imm
44 LWD r0, r0 //rx = M[ry]
45 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
46     //Else do nothing
47 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
48     //Else do nothing
49 ADDI r0, 1 //rx = rx + imm
50 INIT r3, 0 //rx = imm
51 LWD r3, r3 //rx = M[ry]
52 SLER r3, r0 //If rx < r0 then r3 = 1
53     //Else r3 = 0
54 JIF 3 //If r3 = 1 then jump (PC = PC + imm)
55     //Else do nothing
56 JIF 3 //If r3 = 1 then jump (PC = PC + imm)
57     //Else do nothing
58 INIT r3, 1 //rx = imm
59 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
60     //Else do nothing
61 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
62     //Else do nothing
```

Program 2 Disassembler Output



```
1
2  INIT r0, 1 //rx = imm
3  INIT r1, 0 //rx = imm
4  SWD r0, r1 //M[ry] = rx
5  INIT r0, 3 //rx = imm
6  ADDI r0, 2 //rx = rx + imm
7  INIT r1, 1 //rx = imm
8  SWD r0, r1 //M[ry] = rx
9  INIT r0, 3 //rx = imm
10 ADDI r0, 1 //rx = rx + imm
11 INIT r1, 2 //rx = imm
12 SWD r0, r1 //M[ry] = rx
13 INIT r0, 1 //rx = imm
14 INIT r1, 2 //rx = imm
15 LWD r2, r1 //rx = M[ry]
16 ADDI r2, 3 //rx = rx + imm
17 SWD r0, r1 //M[ry] = rx
18 INIT r0, 0 //rx = imm
19 INIT r1, 1 //rx = imm
20 ADDI r0, 2 //rx = rx + imm
21 ADDI r1, 1 //rx = rx + imm
22 SLE r1, 2 //If rx < ry then r3 = 1
23     //Else r3 = 0
24 JIF -3 //If r3 = 1 then jump (PC = PC + imm)
25     //Else do nothing
26 JIF -3 //If r3 = 1 then jump (PC = PC + imm)
27     //Else do nothing
28 ADDI r1, 2 //rx = rx + imm
29 INIT r2, 2 //rx = imm
30 LWD r2, r2 //rx = M[ry]
31 ADDI r2, 3 //rx = rx + imm
32 SWD r0, r2 //M[ry] = rx
33 INIT r0, 1 //rx = imm
34 LWD r2, r0 //rx = M[ry]
35 INIT r0, 3 //rx = imm
36 LWD r3, r0 //rx = M[ry]
37 LWD r0, r2 //rx = M[ry]
38 XOR r0, r3 //rx = rx XOR ry
39 CNTR0 //Count the number of 1's in r0
40 INIT r0, 2 //rx = imm
41 JIF -6 //If r3 = 1 then jump (PC = PC + imm)
42     //Else do nothing
43 JIF -6 //If r3 = 1 then jump (PC = PC + imm)
44     //Else do nothing
45 LWD r2, r0 //rx = M[ry]
46 SWD r3, r2 //M[ry] = rx
47 INIT r0, 2 //rx = imm
48 LWD r2, r0 //rx = M[ry]
49 ADDI r2, 1 //rx = rx + imm
50 LWD r0, r2 //rx = M[ry]
51 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
52     //Else do nothing
53 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
54     //Else do nothing
55 SLE r3, 0 //If rx < ry then r3 = 1
56     //Else r3 = 0
57 JIF 4 //If r3 = 1 then jump (PC = PC + imm)
58     //Else do nothing
59 JIF 4 //If r3 = 1 then jump (PC = PC + imm)
60     //Else do nothing
61 SWD r3, r2 //M[ry] = rx
62 INIT r3, 1 //rx = imm
63 JIF 7 //If r3 = 1 then jump (PC = PC + imm)
64     //Else do nothing
```

```
 65 JIF 7 //If r3 = 1 then jump (PC = PC + imm)
 66     //Else do nothing
 67 ADDN//r3 = r3 - 1
 68 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
 69     //Else do nothing
 70 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
 71     //Else do nothing
 72 SLE r0, 3 //If rx < ry then r3 = 1
 73     //Else r3 = 0
 74 JIF 3 //If r3 = 1 then jump (PC = PC + imm)
 75     //Else do nothing
 76 JIF 3 //If r3 = 1 then jump (PC = PC + imm)
 77     //Else do nothing
 78 INIT r0, 2 //rx = imm
 79 LWD r2, r0 //rx = M[ry]
 80 JIF 6 //If r3 = 1 then jump (PC = PC + imm)
 81     //Else do nothing
 82 JIF 6 //If r3 = 1 then jump (PC = PC + imm)
 83     //Else do nothing
 84 ADDI r2, 2 //rx = rx + imm
 85 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
 86     //Else do nothing
 87 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
 88     //Else do nothing
 89 LWD r3, r2 //rx = M[ry]
 90 ADDI r3, 1 //rx = rx + imm
 91 SWD r3, r2 //M[ry] = rx
 92 INIT r0, 1 //rx = imm
 93 LWD r2, r0 //rx = M[ry]
 94 JIF -6 //If r3 = 1 then jump (PC = PC + imm)
 95     //Else do nothing
 96 JIF -6 //If r3 = 1 then jump (PC = PC + imm)
 97     //Else do nothing
 98 ADDI r2, 1 //rx = rx + imm
 99 INIT r0, 2 //rx = imm
100 LWD r3, r0 //rx = M[ry]
101 ADDI r3, 3 //rx = rx + imm
102 LWD r3, r3 //rx = M[ry]
103 SLER r2, r1 //If rx < r0 then r3 = 1
104     //Else r3 = 0
105 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
106     //Else do nothing
107 JIF -7 //If r3 = 1 then jump (PC = PC + imm)
108     //Else do nothing
```

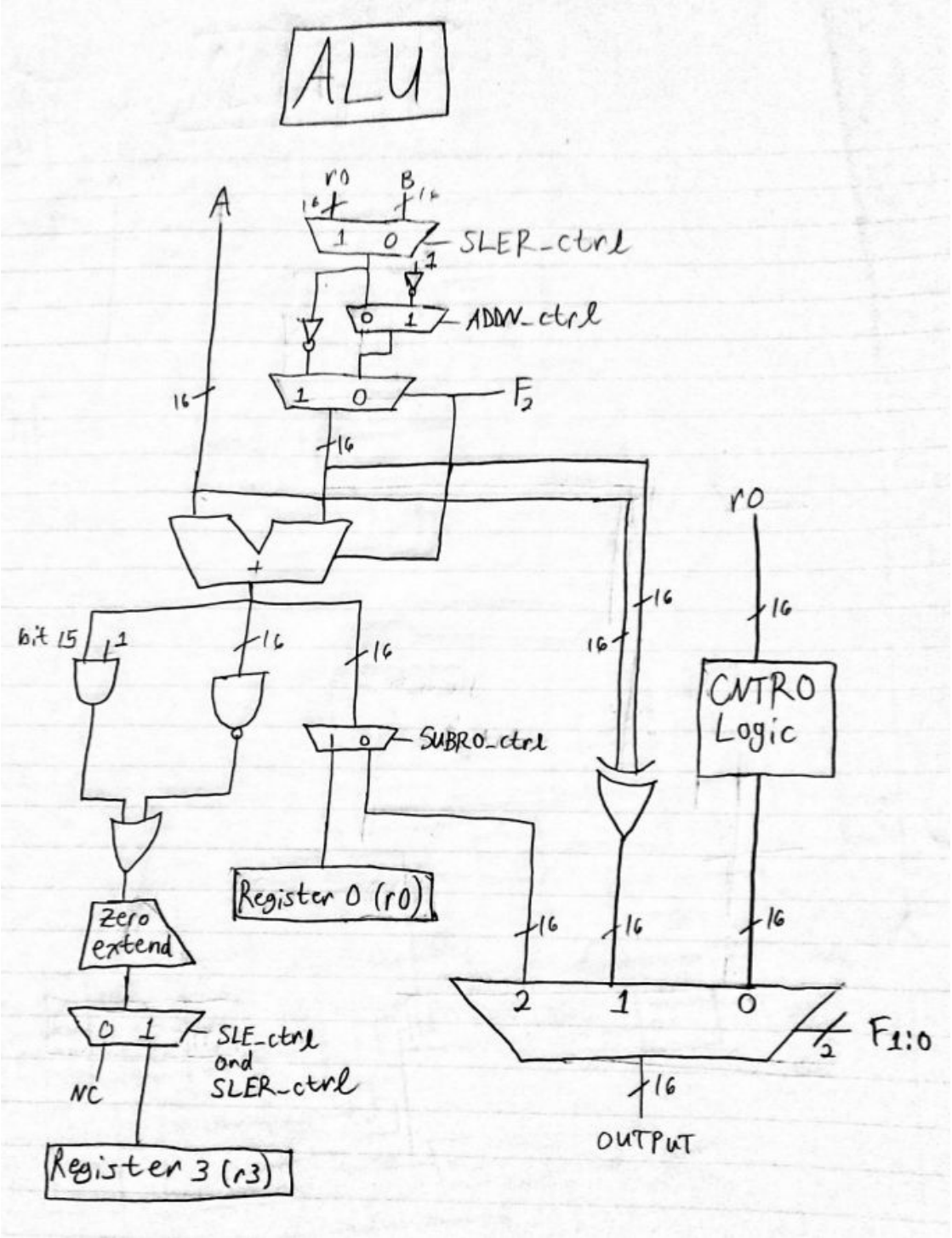
4. Python Code for your ISA's disassembler.

```
#!/usr/bin/env python

input_file = open("LIS_machine_code_program1.txt", "r")
output_file = open("program1_disassembled.lis", "w")

output = "\n"

for line in input_file:
    print output
    output_file.write(output)

    wrongOp = False
    if (line == "\n"):
        continue

    line = line.replace("\n", "")
    print "Machine code: ", line

    op_bin = line[1:8]
    if (op_bin == "0000000"):
        op = "HLT"
        output = op + " //Stop program\n"
        continue

    elif (op_bin == "1111100"):
        op = "ADDN"
        output = op + "//r3 = r3 - 1\n"
        continue

    elif (op_bin == "0001000"):
        op = "CNTR0"
        output = op + " //Count the number of 1's in r0\n"
        continue

    op_bin = line[1:6]

    if (op_bin == "00000"):
        op = "SUBR0"
        ry = str(int(line[6:8], 2))
        if (ry == '1'):
            wrongOp = True
        if (wrongOp == False):
            output = op + " r" + ry + " //r0 = r0 - ry\n"
            continue

    op_bin = line[1:5]

    if (op_bin == "0001"):
        op = "XOR"
        rx = str(int(line[5:6], 2))
        ry = str(int(line[6:8], 2))
        output = op + " r" + rx + ", r" + ry + " //rx = rx XOR ry\n"
        continue

    elif (op_bin == "0000"):
        op = "SLER"
        rx = str(int(line[5:7], 2))
        ry = str(int(line[7], 2))
        output = op + " r" + rx + ", r" + ry + " //If rx < r0 then r3 = 1\n\t//Else r3 = 0\n"
        continue
```

```
61
62      op_bin = line[1:4]
63
64      if (op_bin == "100"):
65          op = "ADD"
66          rx = str(int(line[4:6], 2))
67          ry = str(int(line[6:8], 2))
68          output = op + " r" + rx + ", " + ry + " //rx = rx + ry\n"
69          continue
70
71      elif (op_bin == "111"):
72          op = "ADDI"
73          rx = str(int(line[4:6], 2))
74          const = str(int(line[6:8], 2))
75          output = op + " r" + rx + ", " + const + " //rx = rx + imm\n"
76          continue
77
78      elif (op_bin == "001"):
79          op = "LWD"
80          rx = str(int(line[4:6], 2))
81          ry = str(int(line[6:8], 2))
82          output = op + " r" + rx + ", r" + ry + " //rx = M[ry]\n"
83          continue
84
85      elif (op_bin == "011"):
86          op = "SWD"
87          rx = str(int(line[4:6], 2))
88          ry = str(int(line[6:8], 2))
89          output = op + " r" + rx + ", r" + ry + " //M[ry] = rx\n"
90          continue
91
92      elif (op_bin == "110"):
93          op = "SLE"
94          rx = str(int(line[4:6], 2))
95          ry = str(int(line[6:8], 2))
96          output = op + " r" + rx + ", " + ry + " //If rx < ry then r3 = 1\n\t//Else r3 = 0\n"
97          continue
98
99      elif (op_bin == "101"):
100         op = "INIT"
101         rx = str(int(line[4:6], 2))
102         const = str(int(line[6:8], 2))
103         output = op + " r" + rx + ", " + const + " //rx = imm\n"
104         continue
105
106     elif (op_bin == "010"):
107         op = "JIF"
108         sign = line[4]
109         const = int(line[5:8], 2)
110         if (sign == '1'):
111             const = -(0b111 - int(const) + 1)
112
113         const = str(const)
114
115         output = op + " " + const + " //If r3 = 1 then jump (PC = PC + imm)\n\t//Else do nothing\n"
116
117     print output
118     output_file.write(output)
```

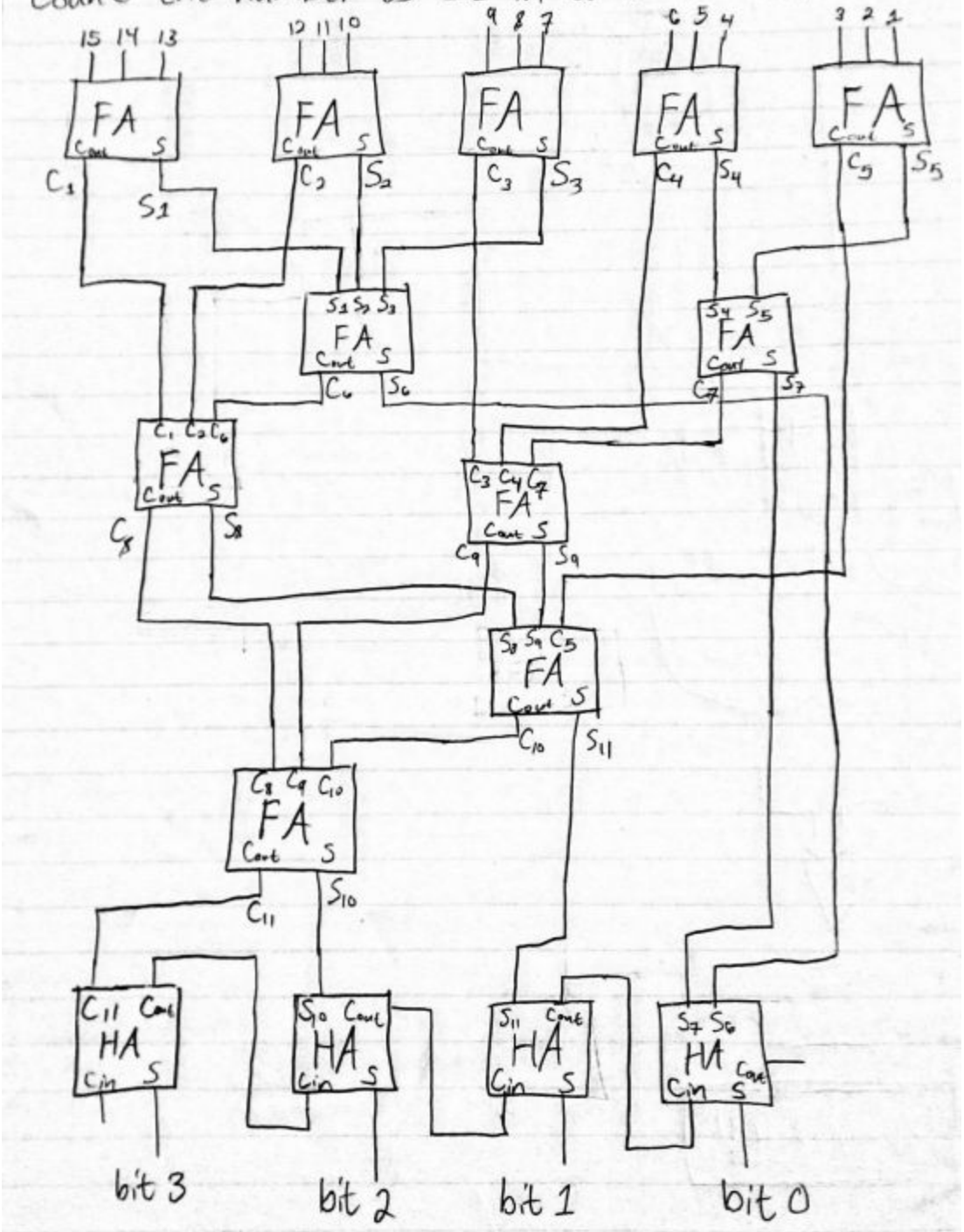
Part D: Hardware Implementation

1. **ALU Schematic. A hierarchical sketch of your Arithmetic Logic Unit which implements whatever computation that your ISA instructions use.**

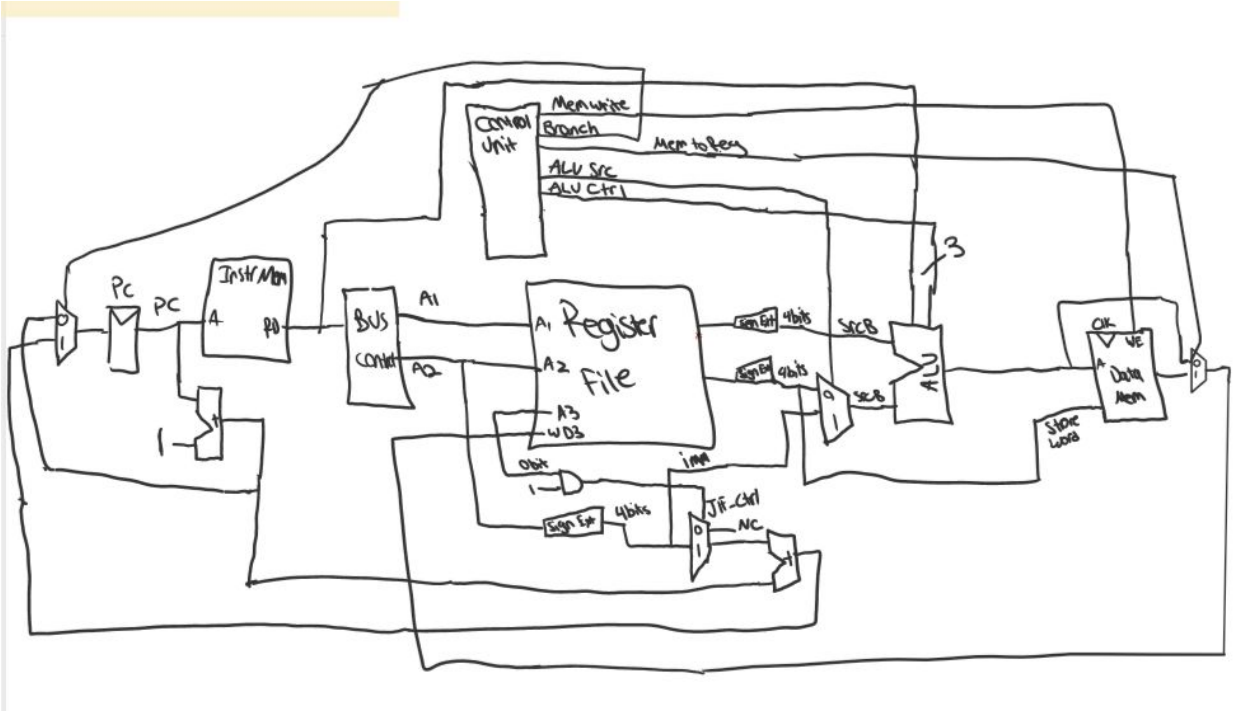


CNTR0

Count the number of 1's in a 16-bit half-word



2. **CPU Datapath Design. A schematic including your register file, ALU, PC logic, and memory components.**

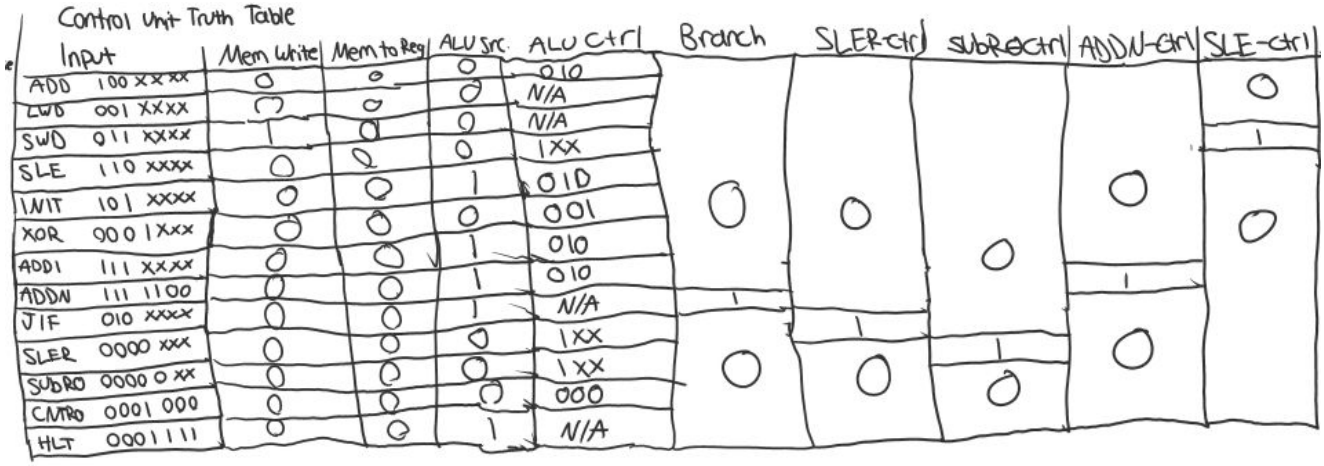


Bus Control Truth Table

#1 = first occurance
#2 = second occurance

	Instr bits	A1	A2
SLE	110 xx xx	XX #1	XX #2
INIT	101 XXXX	XX #1	XX #2
XOR	0001 XXX	X #1	XX #2
ADDI	111 XXXX	XX #1	XX #2
ADD	100 XXXX	XX #1	XX #2
LWD	001 XXXX	XX #1	XX #2
SWD	011 XXXX	XX #1	XX #2
ADDN	1111100	11	00
JIF	010 XXXX	0	XXXX #1
SLER	0000 XXX	XX #1	X #2
SUBR0	00000 XX	0	XX
CNTR0	0001000	00	0
HLT	000 1111	11	11

3. **Control Logic Design. Decoder truth-table indicating how each control signal is generated from an instruction.**



Control Unit Truth Table

Input	Mem Write	Mem to Reg	ALU Src	ALU Ctrl	Branch	SLER-ctrl	SUBRO-ctrl	ADDN-ctrl	SLE-ctrl
ADD 100 XXXX	0	0	0	010					
LWD 001 XXXX	0	1	0	N/A					
SWD 011 XXXX	1	0	0	N/A					
SLE 110 XXXX	0	0	0	1XX					1
INIT 101 XXXX	0	0	1	010					
XOR 000 1XXX	0	0	0	001	0	0	0	0	0
ADDI 111 XXXX	0	0	1	010					
ADDN 111 1100	0	0	1	010				1	
JIF 010 XXXX	0	0	1	N/A	1				
SLER 0000 XXX	0	0	0	1XX		1			
SUBRO 0000 0 XX	0	0	0	1XX	0	0	1	0	
CMPRO 0001 000	0	0	0	000			0		
HLT 0001111	0	0	1	N/A					