Table of Contents

A) ISA Intro

1. Introduction

- **Name of Architecture:** JHT
- **Overall Philosophy:**

  The philosophy behind this architecture is to have an efficient and user-friendly ISA Design.

- **Goals:**
  - Achieve Modular Exponentiation Program for any 16-bit P and Q
  - Achieve Best Match Count Program for any best-matching score

2. Instruction List

Instruction	Format	Opcode	Example
Add	Add Rx, Ry	000 xx yy	Rx = Rx + Ry
Sub	Sub Rx, Ry	001 xx yy	Rx = Rx - Ry
load	Load Rx, (Ry)	010 xx yy	Rx <- M[Ry]
Store	Store Rx, (Ry)	011 xx yy	M[Ry] <- Rx
Jump	Jump Rx	100 00 xx	PC -= Rx
bezR1	bezR1 Rx	101 00 xx	If R1==0, PC += Rx
SltR1	SltR1 Rx, Ry	110 xx yy	if Rx < Ry, R1 = 1
Init	Init Rx, imm	111 0 xx i	Rx = imm
ShiftL	ShiftL	111 10 00	R2 <<
ShiftR	ShiftR	111 10 01	R2 >>
AndR3	AndR3	111 11 01	R3 AND 1
XorR2R1	XorR2R1	111 11 00	R2 = R2 XOR R1

3. Register Design

Number of registers: 4 (R0 to R3)

R1 is for testing purposes when using bezR1 or sltR1.

4. Control Flow (Branches)

- **Branches supported:**
  - Jump: only goes backward, based on value in register Rx
  - bezR1: branch if R1 == 0, only goes forward, based on value in register Rx

- **Target address calculus:** PC = PC ± Rx

- **Maximum branch distance:**
  - For Jump: Rx (bin) = value of Rx (only goes backward)
  - For bezR1: Rx (bin) = value of Rx (only goes forward)

- **Examples:**

Assembly Instruction	Machine Code	Function
Jump R2	1100 0010	PC = PC – R2
bezR1 R0	0101 0000	If R1==0: PC += R0 Else: PC++

5. Data addressing modes

What addressing modes are supported for data memory? How are the addresses calculated? Give examples of your assembly load / store instructions and their corresponding machine code.

- **Addressing Modes:** M[Rx] with Rx being a register with a 16-bit value

- **Address Calculation:** M[0] is the 16-bit word in address 0 of the data memory

- **Examples:**

Assembly Instruction	Machine Code	Function
Load R0, (R1)	0010 0001	R0 <- M[R1]
Store R2, (R3)	1011 1011	M[R3] <- R2

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 uses one less instruction bit and fits in more instructions. Our ISA has a parity bit. Our ISA can jump to any location using load register.

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

A lot of our instructions are optimized to achieve the two goals (ex: ShiftL, XorR2R1, AndR3…)

3. What would you have done differently if you had 1 more bit for instructions? How about 1 fewer bit?

- We would use more instructions with registers (ex: AND Rx, Rx, Ry).
- We would need to heavily optimize our instructions for the goals and we would have less instructions.

4. How did your team work together to accomplish this project?

We spent countless hours meeting up and solving problems together. Each one of us completed an important part of the project.

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

We would have optimized the Project 1 assembly code to simplify the translation of our machine code for our ISA.

C) Software Package

1. Algorithms (in assembly code) of the two programs

Program 1 (ME):

```
init R0, 1            //R0 = 1
init R1, 1            //R1 = 1
init R3, 0            //R3 = 0
load R2, (R3)         //R2 = M[R3=0] = p
```

```
    add R0, R0          //R0 = R0 + R0 = 2
    add R0, R1          //R0 = R0 + R1 = 3
    add R0, R0          //R0 = R0 + R0 = 6
loop:
    init R1, 0          //R1 = 0
    sltR1 R3, R2        //if R3 < R2, R1 = 1
    bezR1 R0            //R1 = 0, jump to location in R0 (mod)
    init R1, 1          //R1 = 1
    add R3, R1          //R3 = R3 + R1 = i + 1
    add R0, R0          //R0 = R0 + R0 = 12
    init R1, 1          //R1 = 1
    add R1, R1          //R1 = 2
    add R1, R1          //R1 = 4
    store R0, (R1)      //R0 = 12 -> M[R1=4]
    load  R1, (R1)      //R1 = 12
    add R0, R1          //R0 = R0 + R1 = 24
    add R0, R0          //R0 = R0 + R0 = 36
    jump R2             //PC = PC – 14

//2nd Part: Mod Calculus
mod:
    init R3, 0          //R3 = 0 in which we will store R0 = 6^p
    store R0, (R3)      //Store the value of 6^p in M[0]
    init R1, 1          //Set R1 = 1
    add R3, R1          //increment R3 by 1 to save 17 w/o overwriting 6^p
    init R2, 1          //Set R2 = 1
    ShiftL              //Shift R2 left until it gets to 16 (1 -> 10)
    ShiftL              //shift from 10 -> 100
    ShiftL              //Shift from 100 -> 1000
    ShiftL              //Shift from 1000 -> 10000
    Add R2, R1          //Then R2 = R2 + R1 = 17 (10000 + 1 =10001)
    Store R2, (R3)      //save 17 (R2) in memory
shifting:               //loop the shifting
    ShiftL              //Shift R2 left until R2 > 6^p
    sltR1 R0, R2        //If R0 (6^p) < R2, R1 = 1 else R1 = 0
```

```
        bezR1 R3                 //if R1 = 1
        sub R3, R1               //move R3 to 6^p
        load R0, (R3)            //R0=6^p
        add R3, R1               //move R3 to read 17 from mem
        load R3, (R3)            //load 17 into R3 from R3
sub17:          //this is where we loop a subtraction of the shifted 17...for example
100010000000 - 17 repeatedly until under 6^p
        Sub R2, R3               //While (R2 > 6^p): R2 = R2 - 17
        sltR1 R0, R2             //10001..<6^p, R1=1
        bezR1 R3                 //since R1=1 go back 4 to sub17 loop position
        sub R0, R2               //Result = 6^p - R2 = 6^p mod 17
        store R0, (R3)           //store result into memory and we are done
```

Program 2 (BMC):

```
        Init R2, 1               //R2 = 1
        ShiftL                   //R2 = 2
        ShiftL                   //R2 = 4
        ShiftL                   //R2 = 8
        ShiftL                   //R2 = 16 comparison bits
        Add R3, R2               //R3 = 16
        Init R2, 1               //R2 = 1
        Store R3, (R2)           //M[1] = 16 compare bits
        Add R2, R2               //R2=2
        Init R1, 0               //R1=0
        Store R1, (R2)           //M[2]=0
        /*
        R0=M[4] is the store dest of done
        Note that each loop location has been preset in data memory locations so it would
        be declared in an array just like this project's comparing array
        R0 = M[5] is the store dest of one
        R0 = M[6] is the store dest of L2
        R0 = M[7] is the store dest of best2
        R0 = M[8] is the store dest of best
        R0 = M[9] is the store dest of finish
        R0 = M[10] is the store dest of exit
```

```
    R0 = M[11] is the store counter for amount of numbers
    */
L1:
    Init R1, 1              //R1=1
    Init R2, 1              //R2=1
    Add R1, R1              //R1=2
    Add R1, R2              //R1=3
    Load R0, (R1)           //R0=Array from M[3]
    Add R1, R2              //R1=4
    Load R1, (R1)           //R1=Counter for 1 bits M[4]
    Init R2, 1              //R2=1
    Add R3, R2              //R3 = R3 + R2
    Init R2, 0              //R2 = 0
    Store R2, (R3)          //M[R3] <- R2
    Load R2, (R0)           //R2 <- M[R0]
    XORR2R1                 //
L2:                         //loop to XOR bits
    Init R1, 0              //R1=0
    XORR2R1                 //conditional to see if we should quit looping based
    upon incrementally anding xored value with 1 and shifting right
    init r3, 0              //R3=0
    SltR1 R3, R2            //as long as xored valued is not zero then do not quit
    this loop
    BezR1, R0               //jump/quit to done
    Add R3, R2              //R3 now equals xored value as well
    Init R2, 1              //R2=1
    ShiftL                  //shift xor
    Add R2, R1              //R2 = 3 or m[3]
    Store R3, (R2)          //current xor gets stored in m[3]
    ANDR3                   //AND xor lsb with 1
    Init R2, 1              //R2 = 1
    ShiftL                  //R2 = 2
    ADD R2, R1              //R2 = 3
    Load R2, (R2)           //R2 = XOR
    ShiftR                  //XOR/2 or get rid of LSB in XOR
```

```
    Init R1, 1          //R1 = 1
    SltR1 R3, R1        //check if AND value is equal to 1
    Init R2, 1          //R2 = 1
    ShiftL              //R2 = 2
    ShiftL              //R2 = 4
    Add R2, R1          //R2 = 5
    Load R0, (R2)       //R0 = memory location of where to jump
    BezR1 R0            //if the ANDR3 was a 1AND1 then we will jump to
    'one' func
    Init R1, 1          //R1 = 1
    Add R2, R1          //memory location of L2 loop
    Load R0, (R2)       //load L2 location for upcoming jump
    ShiftR              //memory is now looking at XOR
    Load R2, (R2)       //load XOR for L2 initial conditionals
    Jump R0             //jump back to beginning of this loop
one:
    Init R1, 1          //R1 = 1
    Sub R2, R1          //R2 = M[4]
    ShiftR              //R2 = M[2]
    Load R1, (R2)       //load number of ones in counter
    Init R3, 1          //R3 = 1
    Add R1, R3          //add one to counter since we found a 1 from XOR
    value in previous loop
    Store R1, (R2)      //store this into counter value in memory
    Add R2, R3          //move memory location to XOR location
    init R0, 1          //R0 = 1
    Add R0, R2          //R0 = M[4]
    Load R2, (R2)       //R2 = XOR value
    Init R3, 1          //R3 = 1
    Add R0, R3          //R0 = M[5]
    Add R0, R3          //R0 = M[6]
    Load R1, (R3)       //load location of 'L2' or previous loop
    Jump R1             //jump to L2
done:
    Init R0, 1          //R0 = 1
```

```
Load R3, (R0)      //R3 = 16 bits of 1111111111111111
Init R2, 1         //R2 = 1
ShiftL             //R2 = 2
Load R0, (R2)      //R0 = 1's counted from loop operations Xor/AND
Sub R3, R0         //111... - # of 1's
Init R2, 0         //R2 = 0
Load R1, (R2)      //R1 = M[0] = top score
Sub R1, R3         //check to see if R3 total is equal to top score
Init R0, 1         //R0 = 1
Init R2, 1         //R2 = 1
ShiftR             //R2 = 2
ShiftR             //R2 = 4
Add R2, R0         //R2 = 5
Add R2, R0         //R2 = 6
Add R2, R0         //R2 = 7
load R0, (R2)      //R0 is now location of 'best2' since top and xor were
equal we will add 1 to best matching counter
BezR1, R0          //branch to location of 'best2' based upon condition
stated above
Init R0, 1         //R0 = 1
Add R0, R2         //R0 = location of 'best' in memory M[8]
Init R2, 0         //R2 = 0
load R2, (R2)      //R2 = M[0] = top score
sltR1, R3, R2      //check if current xor has more matching bits than
the previous best score
load R2, (R0)      //R2 = location 'best' loop now
BezR1, R2          //branch if xor matching bits is greater than the
previous best score
init R2, 1         //reset R2 to 1 if above condition didn't branch
add R0, R2         //R0 = 9
Load R0, (R0)      //R0 = M[9] = location of finish
Jump R0            //jump to finish
best:
Init R0, 0         //R0 = 0
Store R3, (R0)     //store new top score R3 in M[0]
Init R0, 1         //R0 = 1
```

```
        Add R0, R0              //R0 = 2
        Init R1, 0              //R1 = 0
        Store R1, (R0)          //M[2] total number of best matching #'s reset to 0
best2:
        init R0, 1              //R0 = 1
        Add R0, R0              //R0 = 2
        load R0, (R0)           //R0 = M[2]
        Init R1, 1              //R1 = 1
        Add R0, R1              //R0 = R0 + R1
finish:
        init R2, 1              //R2 = 1
        ShiftL                  //R2 = 2
        init R0, 1              //R0 = 1
        Add R0, R2              //R0 = 3
        load R2, (R0)           //R2 = M[3] loads array values into R2
        /*
        shift left up until 16 bits has passed and we now use a new value in the array for
        the next iterations of checking for best scores/matches
        */
        ShiftL
        ShiftL
        ShiftL
        ShiftL
        ShiftL
        Store R2, (R0)          //store value so we can use it once we loop back
        Init R2, 1              //R2 = 1
        Sub R0, R2
        Sub R0, R2
        Sub R0, R2
        Sub R0, R2
        Sub R0, R2              //R0 = 11
        load R1, (R0)           //counter is loaded from how many numbers are in
        array
        Add R1, R2              //counter++
        Sub R0, R2              //R0 = 10
```

```
        Init R2, 0          //R2 = 0
        Add R2, R0          //R2 = R0 = 10
        ShiftL              //R2 = 20
        Init R0, 1          //R1 = 1
        Add R2, R0          //R2 = 21
        Add R2, R0          //R2 = 22
        Add R2, R0          //R2 = 23
        Add R2, R0          //R2 = 24
        Add R2, R0          //R2 = 25
        ShiftL              //R2 = 50
        ShiftL              //R2 = 100
        SltR1 R1, R2        //if R1<100
        Init R0, 1          //R0 = 1
        Init R2, 1          //R2 = 1
        Add R2, R0          //R2 = 2
        Add R2, R0          //R2 = 3
        Add R2, R2          //R2 = 6
        ShiftL              //R2 = 12
        Sub R2, R0          //R2 = 11
        Sub R2, R0          //R2 = 10
        load R0, (R2)       //R0 = M[10] is the location of exit
        bezR1, R0           //branch to exit
        Init R0, 1          //initiate R0 back to 1 if we don't exit
        Sub R2, R0
        Sub R2, R0
        Sub R2, R0
        Sub R2, R0          //R2 = 6 address of L2 in memory
        Load R0, (R2)       //R0 = M[6] is the location of L2
        Jump R0             //jump to L2
exit:
        Init R0, 1          //R0 = 1
        Add R0, R0          //R0 = 2
        Load R0, (R0)       //R0 = M[2]
```

2. Machine code for each of the programs

Program 1 (ME):

01110001
11110011
11110110
00101011
00000000
10000001
00000000
01110010
11101110
01010000
11110011
10001101
00000000
11110011
00000101
00000101
10110001
10100101
10000001
00000000
01000010
11110110
00110011
11110011
10001101
11110101
01111000
01111000
01111000
01111000
00001001
10111011
01111000
11100010
01010011
00011101
10100011
10001101
10101111
00011011
11100010
01010011
00010010
00110011

Program 2 (BMC):

11110101
01111000
01111000
01111000
01111000
10001110
11110101
10111110
11110101
00001010
01110010
00110110
11110011
11110101
00000101
00000110
00100001
00000110
10100101
01110101
10001110
01110100
10111011
00101000
11111100
01110010
11111100
11110110
11101110
01010000
10001110
11110101
01111000
00001001
10111110
01111101
11110101
01111000
00001001
10101010
11111001
11110011
11101101
11110101
01111000
01111000
00001001

00100010
01010000
11110101
00001001
00100010
01111001
10101010
11000000
11110011
10011001
11111001
10100110
01110111
10000111
00110110
10001011
01110001
10000010
10101010
01110111
00000011
00000011
00100111
10000001
01110001
10101100
11110101
01111000
00100010
10011100
01110100
10100110
00010111
01110001
11110101
01111001
01111001
10001000
10001000
10001000
00100010
01010000
01110001
10000010
01110100
10101010
11101110
00101000
01010110

11110101
10000010
10100000
11000000
11110000
00111100
01110001
00000000
01110010
10110100
01110001
00000000
10100000
11110011
10000001
11110101
01111000
01110001
10000010
00101000
01111000
01111000
01111000
01111000
01111000
10111000
11110101
00010010
00010010
00010010
00010010
00010010
00100100
00000110
00010010
01110100
10000010
01111000
01110001
10001000
10001000
10001000
10001000
10001000
01111000
01111000
01100110
01110001
11110101

```
10001000
10001000
00001010
01111000
00011000
00011000
00100010
11010100
01110001
00011000
00011000
00011000
00011000
00100010
11000000
01110001
00000000
10100000
```

3. Output of your Python disassembler for each program

Program 1 (ME):

Disassembling Machine Code from Program 1...

```
Init R0, 1        // R0 = 1
Init R1, 1        // R1 = 1
Init R3, 0        // R3 = 0
Load R2, (R3)         // R2 <- M[R3]
Add R0, R0        // R0 = R0 + R0
Add R0, R1        // R0 = R0 + R1
Add R0, R0        // R0 = R0 + R0
Init R1, 0        // R1 = 0
SltR1 R3, R2        // if R3 < R2, R1 = 1
bezR1 R0        // if R1 == 0: PC = PC + R0 | else: PC++
Init R1, 1        // R1 = 1
Add R3, R1        // R3 = R3 + R1
Add R0, R0        // R0 = R0 + R0
Init R1, 1        // R1 = 1
Add R1, R1        // R1 = R1 + R1
Add R1, R1        // R1 = R1 + R1
Store R0, (R1)         // M[R1] <- R0
Load R1, (R1)         // R1 <- M[R1]
Add R0, R1        // R0 = R0 + R1
Add R0, R0        // R0 = R0 + R0
Jump R2        // PC = PC - R2
Init R3, 0        // R3 = 0
Store R0, (R3)         // M[R3] <- R0
```

```
Init R1, 1        // R1 = 1
Add R3, R1        // R3 = R3 + R1
Init R2, 1        // R2 = 1
ShiftL            // R2 <<
ShiftL            // R2 <<
ShiftL            // R2 <<
ShiftL            // R2 <<
Add R2, R1        // R2 = R2 + R1
Store R2, (R3)    // M[R3] <- R2
ShiftL            // R2 <<
SltR1 R0, R2      // if R0 < R2, R1 = 1
bezR1 R3          // if R1 == 0: PC = PC + R3 | else: PC++
Sub R3, R1        // R3 = R3 - R1
Load R0, (R3)     // R0 <- M[R3]
Add R3, R1        // R3 = R3 + R1
Load R3, (R3)     // R3 <- M[R3]
Sub R2, R3        // R2 = R2 - R3
SltR1 R0, R2      // if R0 < R2, R1 = 1
bezR1 R3          // if R1 == 0: PC = PC + R3 | else: PC++
Sub R0, R2        // R0 = R0 - R2
Store R0, (R3)    // M[R3] <- R0
```

<u>Program 2 (BMC):</u>

Disassembling Machine Code from Program 2...

```
Init R2, 1        // R2 = 1
ShiftL            // R2 <<
ShiftL            // R2 <<
ShiftL            // R2 <<
ShiftL            // R2 <<
Add R3, R2        // R3 = R3 + R2
Init R2, 1        // R2 = 1
Store R3, (R2)    // M[R2] <- R3
Init R2, 1        // R2 = 1
Add R2, R2        // R2 = R2 + R2
Init R1, 0        // R1 = 0
Store R1, (R2)    // M[R2] <- R1
Init R1, 1        // R1 = 1
Init R2, 1        // R2 = 1
Add R1, R1        // R1 = R1 + R1
Add R1, R2        // R1 = R1 + R2
Load R0, (R1)     // R0 <- M[R1]
Add R1, R2        // R1 = R1 + R2
Load R1, (R1)     // R1 <- M[R1]
Init R2, 1        // R2 = 1
Add R3, R2        // R3 = R3 + R2
Init R2, 0        // R2 = 0
Store R2, (R3)    // M[R3] <- R2
```

```
Load R2, (R0)       // R2 <- M[R0]
XorR2R1             // R2 = R2 XOR R1
Init R1, 0          // R1 = 0
XorR2R1             // R2 = R2 XOR R1
Init R3, 0          // R3 = 0
SltR1 R3, R2        // if R3 < R2, R1 = 1
bezR1 R0            // if R1 == 0: PC = PC + R0 | else: PC++
Add R3, R2          // R3 = R3 + R2
Init R2, 1          // R2 = 1
ShiftL              // R2 <<
Add R2, R1          // R2 = R2 + R1
Store R3, (R2)      // M[R2] <- R3
AndR3               // R3 AND 1
Init R2, 1          // R2 = 1
ShiftL              // R2 <<
Add R2, R1          // R2 = R2 + R1
Load R2, (R2)       // R2 <- M[R2]
ShiftR              // R2 >>
Init R1, 1          // R1 = 1
SltR1 R3, R1        // if R3 < R1, R1 = 1
Init R2, 1          // R2 = 1
ShiftL              // R2 <<
ShiftL              // R2 <<
Add R2, R1          // R2 = R2 + R1
Load R0, (R2)       // R0 <- M[R2]
bezR1 R0            // if R1 == 0: PC = PC + R0 | else: PC++
Init R2, 1          // R2 = 1
Add R2, R1          // R2 = R2 + R1
Load R0, (R2)       // R0 <- M[R2]
ShiftR              // R2 >>
Load R2, (R2)       // R2 <- M[R2]
Jump R0             // PC = PC - R0
Init R1, 1          // R1 = 1
Sub R2, R1          // R2 = R2 - R1
ShiftR              // R2 >>
Load R1, (R2)       // R1 <- M[R2]
Init R3, 1          // R3 = 1
Add R1, R3          // R1 = R1 + R3
Store R1, (R2)      // M[R2] <- R1
Add R2, R3          // R2 = R2 + R3
Init R0, 1          // R0 = 1
Add R0, R2          // R0 = R0 + R2
Load R2, (R2)       // R2 <- M[R2]
Init R3, 1          // R3 = 1
Add R0, R3          // R0 = R0 + R3
Add R0, R3          // R0 = R0 + R3
Load R1, (R3)       // R1 <- M[R3]
Add R0, R1          // R0 = R0 + R1
Init R0, 1          // R0 = 1
```

```
Load R3, (R0)      // R3 <- M[R0]
Init R2, 1         // R2 = 1
ShiftL             // R2 <<
Load R0, (R2)      // R0 <- M[R2]
Sub R3, R0         // R3 = R3 - R0
Init R2, 0         // R2 = 0
Load R1, (R2)      // R1 <- M[R2]
Sub R1, R3         // R1 = R1 - R3
Init R0, 1         // R0 = 1
Init R2, 1         // R2 = 1
ShiftR             // R2 >>
ShiftR             // R2 >>
Add R2, R0         // R2 = R2 + R0
Add R2, R0         // R2 = R2 + R0
Add R2, R0         // R2 = R2 + R0
Load R0, (R2)      // R0 <- M[R2]
bezR1 R0           // if R1 == 0: PC = PC + R0 | else: PC++
Init R0, 1         // R0 = 1
Add R0, R2         // R0 = R0 + R2
Init R2, 0         // R2 = 0
Load R2, (R2)      // R2 <- M[R2]
SltR1 R3, R2       // if R3 < R2, R1 = 1
Load R2, (R0)      // R2 <- M[R0]
Init R2, 1         // R2 = 1
Add R0, R2         // R0 = R0 + R2
Load R0, (R0)      // R0 <- M[R0]
Jump R0            // PC = PC - R0
Init R0, 0         // R0 = 0
Store R3, (R0)     // M[R0] <- R3
Init R0, 1         // R0 = 1
Add R0, R0         // R0 = R0 + R0
Init R1, 0         // R1 = 0
Store R1, (R0)     // M[R0] <- R1
Init R0, 1         // R0 = 1
Add R0, R0         // R0 = R0 + R0
Load R0, (R0)      // R0 <- M[R0]
Init R1, 1         // R1 = 1
Add R0, R1         // R0 = R0 + R1
Init R2, 1         // R2 = 1
ShiftL             // R2 <<
Init R0, 1         // R0 = 1
Add R0, R2         // R0 = R0 + R2
Load R2, (R0)      // R2 <- M[R0]
ShiftL             // R2 <<
ShiftL             // R2 <<
ShiftL             // R2 <<
ShiftL             // R2 <<
ShiftL             // R2 <<
Store R2, (R0)     // M[R0] <- R2
```

```
Init R2, 1        // R2 = 1
Sub R0, R2        // R0 = R0 - R2
Sub R0, R2        // R0 = R0 - R2
Sub R0, R2        // R0 = R0 - R2
Sub R0, R2        // R0 = R0 - R2
Sub R0, R2        // R0 = R0 - R2
Load R1, (R0)     // R1 <- M[R0]
Add R1, R2        // R1 = R1 + R2
Sub R0, R2        // R0 = R0 - R2
Init R2, 0        // R2 = 0
Add R0, R2        // R0 = R0 + R2
ShiftL            // R2 <<
Init R0, 1        // R0 = 1
Add R2, R0        // R2 = R2 + R0
Add R2, R0        // R2 = R2 + R0
Add R2, R0        // R2 = R2 + R0
Add R2, R0        // R2 = R2 + R0
Add R2, R0        // R2 = R2 + R0
ShiftL            // R2 <<
ShiftL            // R2 <<
SltR1 R1, R2      // if R1 < R2, R1 = 1
Init R0, 1        // R0 = 1
Init R2, 1        // R2 = 1
Add R2, R0        // R2 = R2 + R0
Add R2, R0        // R2 = R2 + R0
Add R2, R2        // R2 = R2 + R2
ShiftL            // R2 <<
Sub R2, R0        // R2 = R2 - R0
Sub R2, R0        // R2 = R2 - R0
Load R0, (R2)     // R0 <- M[R2]
Init R0, 1        // R0 = 1
Sub R2, R0        // R2 = R2 - R0
Sub R2, R0        // R2 = R2 - R0
Sub R2, R0        // R2 = R2 - R0
Sub R2, R0        // R2 = R2 - R0
Load R0, (R2)     // R0 <- M[R2]
Jump R0           // PC = PC - R0
Init R0, 1        // R0 = 1
Add R0, R0        // R0 = R0 + R0
Load R0, (R0)     // R0 <- M[R0]
```

4. Python code for your ISA's disassembler

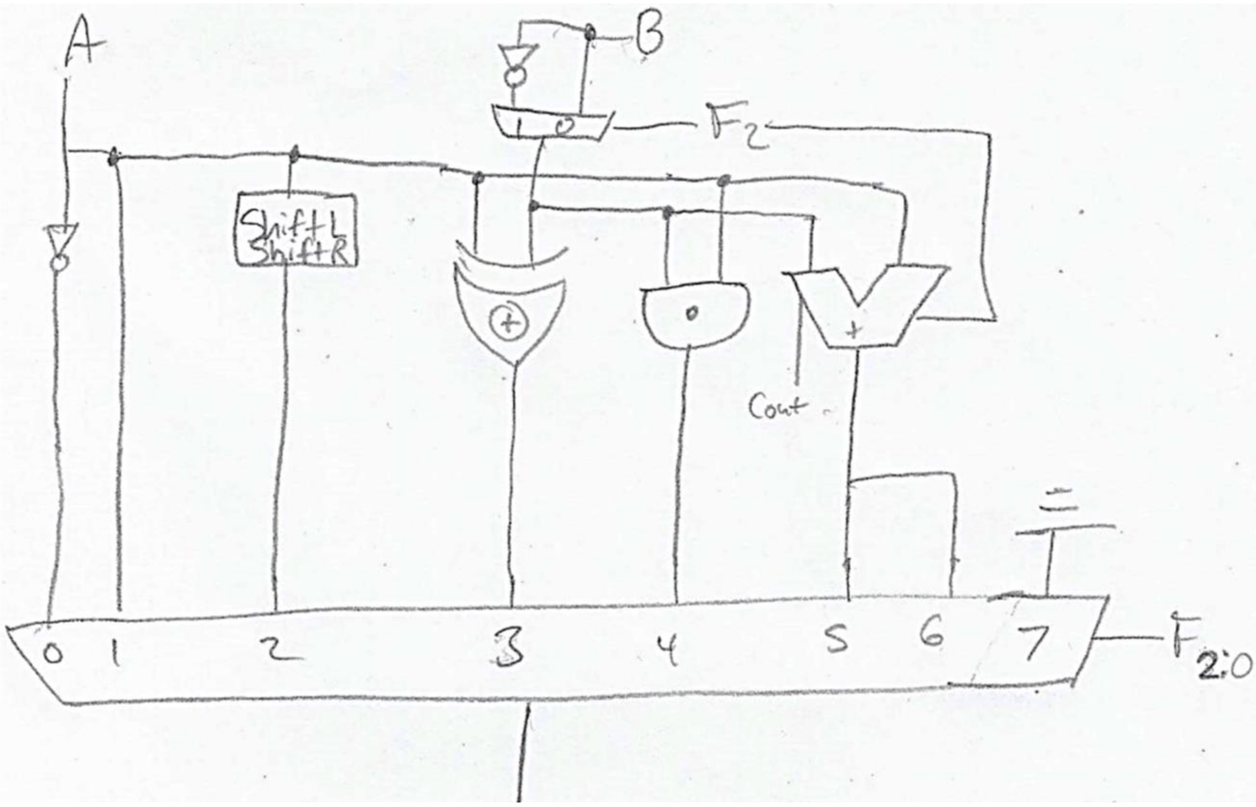
```
input_file = open("project2_group_4_p2_mc.txt","r")
output_file = open("project2_group_4_p2_asm.txt","w")

output_file.write("Disassembling Machine Code from Program 2...\n \n")

for line in input_file:
    if (line[1:8] == "1111100"):
        output_file.write("XorR2R1              // R2 = R2 XOR R1 \n")
```

```
    elif (line[1:8] == "1111101"):
        output_file.write("AndR3                    // R3 AND 1 \n")

    elif (line[1:8] == "1111001"):
        output_file.write("ShiftR                   // R2 >> \n")

    elif (line[1:8] == "1111000"):
        output_file.write("ShiftL                   // R2 << \n")

    elif (line[1:5] == "1110"):
        register = str(int(line[5:7],2))
        immediate = str(int(line[7],2))
        output_file.write("Init R" + register + ", " + immediate)
        output_file.write("          // R" + register + " = " + immediate +
"\n")

    elif (line[1:4] == "110"):
        register1 = str(int(line[4:6],2))
        register2 = str(int(line[6:8],2))
        output_file.write("SltR1 R" + register1 + ", R" + register2)
        output_file.write("          // if R" + register1 + " < R" +
register2 + ", R1 = 1 \n")

    elif (line[1:4] == "011"):
        register1 = str(int(line[4:6],2))
        register2 = str(int(line[6:8],2))
        output_file.write("Store R" + register1 + ", (R" + register2 + ")")
        output_file.write("          // M[R" + register2 + "] <- R" +
register1 + "\n")

    elif (line[1:4] == "010"):
        register1 = str(int(line[4:6],2))
        register2 = str(int(line[6:8],2))
        output_file.write("Load R" + register1 + ", (R" + register2 + ")")
        output_file.write("          // R" + register1 + " <- M[R" +
register2 + "] \n")

    elif (line[1:4] == "001"):
        register1 = str(int(line[4:6],2))
        register2 = str(int(line[6:8],2))
        output_file.write("Sub R" + register1 + ", R" + register2)
        output_file.write("          // R" + register1 + " = R" + register1
+ " - R" + register2 + "\n")

    elif (line[1:4] == "000"):
        register1 = str(int(line[4:6], 2))
        register2 = str(int(line[6:8], 2))
        output_file.write("Add R" + register1 + ", R" + register2)
        output_file.write("          // R" + register1 + " = R" + register1
+ " + R" + register2 + "\n")

    elif (line[1:6] == "10000"):
        register = str(int(line[6:8], 2))
        output_file.write("Jump R" + register)
        output_file.write("          // PC = PC - R" + register + "\n")

    elif (line[1:6] == "10100"):
        register = str(int(line[6:8], 2))
        output_file.write("bezR1 R" + register)
```

```
        output_file.write("          // if R1 == 0: PC = PC + R" + register
+ " | else: PC++ \n")
```

D) Hardware Implantations

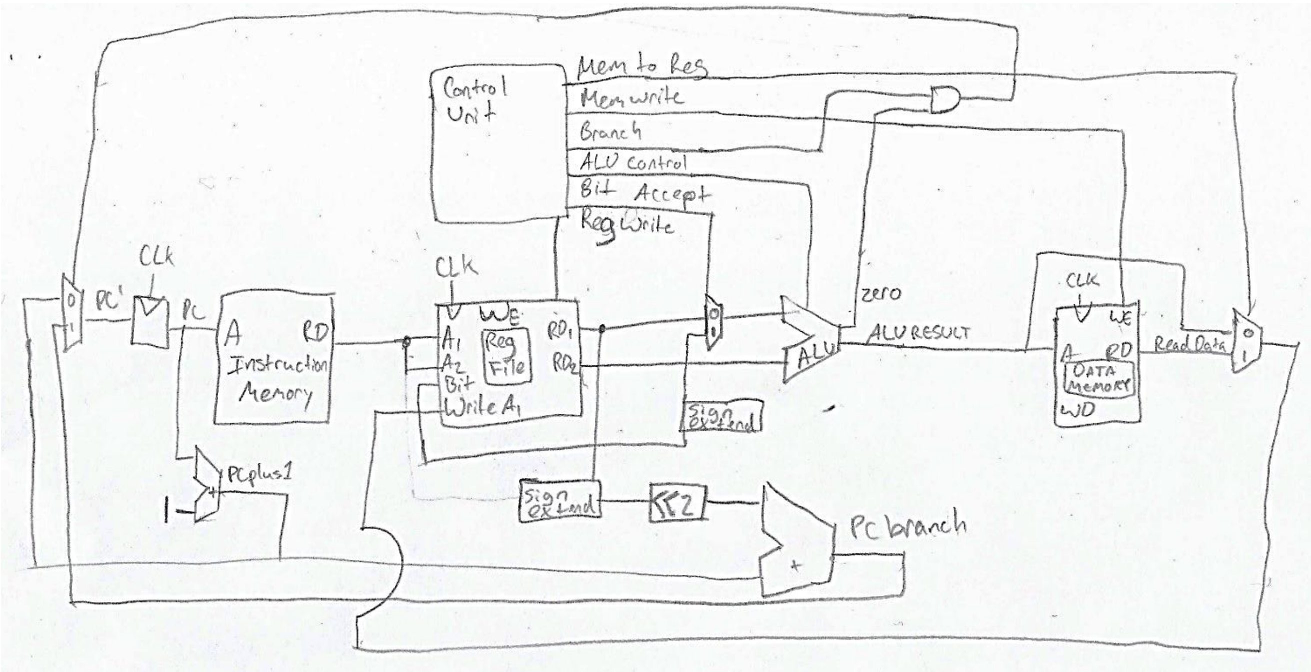
1. ALU schematic



If $F_{2:0}$ picks:

- 000 we get "Init A,0"
- 001 we get "Init A,1"
- 010 we get "ShiftL" or "ShiftR"
- 011 we get "XorR2R1"
- 100 we get "AndR3"
- 101 we get "Sub" or "Add"
- 110 we get "sltR1"
- 111 is grounded so it cannot be picked from the MUX

2. CPU Datapath



3. Control logic

Instr	Op	RegWrite	BitAccept	ALUControl	Branch	MemWrite	MemtoReg	ALUOp
Add	000	1	0	0	0	0	0	10
Sub	001	1	0	0	0	0	0	10
Load	010	1	0	1	0	0	1	00
Store	011	0	0	1	0	1	0	00
Jump	10000	0	0	0	1	0	0	01
bezR1	10100	0	0	0	1	0	0	01
SltR1	110	0	0	0	0	0	0	10
Init	1110	1	1	0	0	0	0	10
ShiftL	1111000	1	0	0	0	0	0	10
ShiftR	1111001	1	0	0	0	0	0	10
AndR3	1111101	1	0	0	0	0	0	10
XorR2R1	1111100	1	0	0	0	0	0	10