# MILESTONE ONE

**0. Team members:**

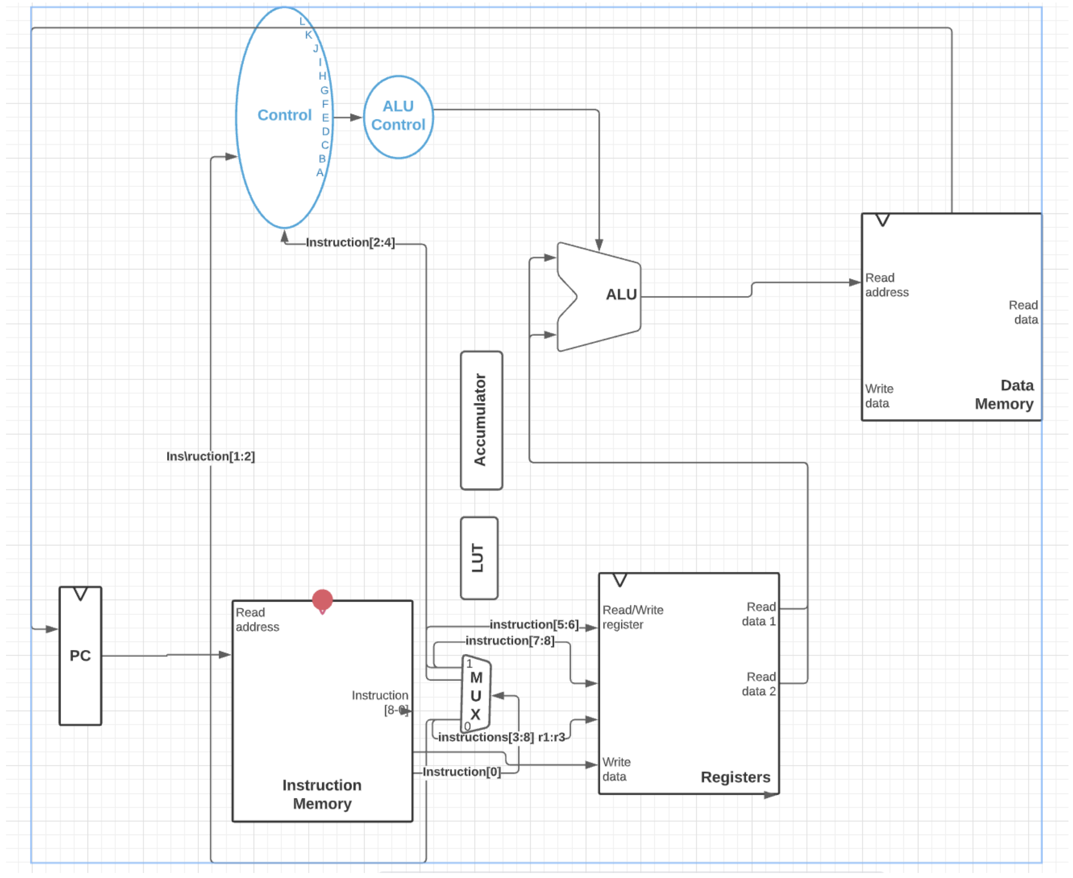
1. Lucy Hu
2. Ayushi Sharma
3. Nathan Nakamura

**1. Introduction:**

1. This should include the name of your architecture (have fun with this J), overall philosophy, specific goals strived for and achieved.
    a. The name of our architecture is ALN which is the name initials for our group. Our philosophy is to design the best possible ISA given the task requirements provided to us. We strive to design our ISA so that it supports 9 bits wide fixed-length instructions using a load-store design.
2. Can you classify your machine in any of the classical ways (e.g., stack machine, accumulator, register, load-store)? If so, which? If not, devise a name for your class of machine.
    a. We classify our machine as a combination of Load Store architecture.

**2. Architectural Overview:**

1. This must be in picture form.



2. What are the major building blocks you expect your processor to be made up of?
    a. We expect our processor to be made up of ALU, data memory, instruction memory, registers, program counter (PC), Mux, LUTs,

**3. Machine Specification:**

1. Instruction formats.
    a. List all formats and an example of each. (ARM has R, I, and B type instructions, for example.)
        - A type
            - 0-2 opcode
            - 3-5 output register/ input register
            - 6-8 - input register
            - Example:  010 001 010
                - AND r1, r2
        - I type
            - 0-2 opcode
            - 3-4 output register/input register
            - 5-8  immediate value
            - Example: 001 01 0001
                - Addi r1, 1
        - J type
            - 0-2 is opcode
            - 3-4 bit to instruction
            - 5-8 address/ inputs
            - Example: 110 01 0011
                - Jump to 3 lines after
2. Operations.
    a. List all instructions supported and their opcodes/formats.
        i. Opcode (3 bits)
            - Add r1, r2,    (000)
            - Addi r1, #        (001)
            - AND r1, r2,   (010)
            - XOR r1,r2,       (011)
            - Store r1, r2/MEM (100)
            - Load r1, r2/MEM (101)
            -
        ii. Opcode ( 4 bits) only when opcode = 111 do shifts
            - >> r1, number (1110)
            - << r2,  number (1111)
        iii. Opcode( 5 bits)  defining branches is 110 opcode
            - Lt ,r1, r2          (11000)
            - Neq, r1, r2       (11001)
            - Eq, r1, r2          (11010)
            - Jump, address  (11011)

3. Internal operands.
    a. How many registers are supported?
        i. 9 registers supported
    b. Is there anything special about any of the registers, or all of them general purpose?
        i. 8 will be general purpose registers
        ii. One register will have the value 28

4. Control flow (branches).
    a. What types of branches are supported?
        i. Absolute branch (like *jump*ing to a specific address)
            1. Equal, not equals, less than. We can specify an address using these comparators to cause the jump to a specific address directly like in the case of Mem[1000].
        ii. Relative branches (like PC + 4, PC+ 8, etc.)
            1. Equal, not equals, less than. We can specify a number from the PC too to move the program counter to another location.
    b. How are the target addresses calculated?
        i. Mainly for relative branch
            1. Using the program counter + 1 when there is no branch instructions. When there is a branch instruction, if the branch is not taken, it goes to PC + 1, if the branch is taken, it goes to PC+2
        ii. For absolute branch:
            1. Jump would be scaled to 5 so Jump 1 is equivalent to actually jumping 5 instructions instead
    c. What is the maximum branch distance supported?
        i. Upper bound for the relative branch
            1. This decision is still a work in progress as our team is thinking about what will work best.

5. Addressing modes.
    a. What memory addressing modes are supported, e.g. direct, indirect?
        i. Direct addressing:
            1. Using this we can do something like mem[10]
        ii. Indirect addressing:
            1. Using this is helpful when we have to access a memory location whose address is too big to fit the 9 bits requirement. We could load a register with a big number in order to point to an address with a big number. And we could set that register equal to a big number by doing a bunch of calculations so that the 9 bits requirement is maintained.
    b. How are addresses calculated?
        i. We don't have anything special for calculating addresses
    c. Give examples.

i. Mem[100] grabs the 100th byte in memory

**4. Programmer's Model:**

1. How should a programmer think about how your machine operates?
    a. This machine is a load-store design that allows for 12 different instructions and 4 different registers.
2. Give an example of an "assembly language" instruction in your machine, then translate it into machine code
    a. Add r1, r3
    b. 000 001 011

**5. Program Implementation:**

1. For each of the 3 programs, give assembly instructions that will implement the program correctly. Make sure your assembly format is either very obvious or well described, and that the code is (very) well commented. If you also want to include machine code, the effort will not be wasted, since you will need it later. We shall not correct/grade the machine code. State any assumptions you make.

**Program 1:**

*Some notes:*
*store all register bits in memory 195 to 205, rightmost value is the bit*
*r0 = i for loop*
*R1 = mem[1st input]*
*R2 = mem[2nd input]*
*R3 = p8*
*R4 = p4*
*R5 = P2*
*R6 = P1*
*R7 = p16*

*Actual program code:*

Loop:
   //r0 or the 'i' starts from 0
   Load r1 mem[r0]              //input MSW                    101 001 ___ (TODO: figure out how to do memory addresses here) it should just be r0, bc r0 is the address for i
   Addi r0, #4            //increment i                    001 000 010
   Load r2 mem[r0]              //input LSW                   101 010 ___ (TODO: memory address)

   Add r3, r1    //set r3 = r1                               000 011 001

```
   Store r3, mem[197] //mem[197] = b9                    100 011 __ (TODO: can we do addi work
with -1 somehow ??? )

   >> r3, 1                                    111 0 11 001
   Store r3, mem[196] //mem[196] = b10                   100 011 __ (TODO)

   >> r3, 1                                                    111 0 11 001
   Store r3, mem[195] //mem[195] = b11 (rightmost bit is b11, ignore rest)    100 011 __ (TODO:
memory address)


   Add r3, r2    //set r3 = r2                          000 011 010
   Store r3, mem[205] //mem[205] = b1                    100 011 __ (TODO)

   >> r3, 1                                    111 0 11 001
   Store r3, mem[204] //mem[204] = b2                    100 011 __ (TODO)

   >> r3, 1                                    111 0 11 001
   Store r3, mem[203] //mem[203] = b3                    100 011 __ (TODO)

   >> r3, 1                                    111 0 11 001
   Store r3, mem[202] //mem[202] = b4                    100 011 __ (TODO)

   >> r3, 1                                    111 0 11 001
   Store r3, mem[201] //mem[201] = b5                    100 011 __ (TODO)

   >> r3, 1                                    111 0 11 001
   Store r3, mem[200] //mem[200] = b6                    100 011 __ (TODO)

   >> r3, 1                                    111 0 11 001
   Store r3, mem[199] //mem[199] = b7                    100 011 __ (TODO)

   >> r3, 1                                    111 0 11 001
   Store r3, mem[198] //mem[198] = b8                    100 011 __ (TODO)


   // b11, b10, b9, b8 b7,... are all memory addresses


   //to make r4 = 195
   And r4, 0                                              010 100 000
   Addi r4 #64                                             001 100 011
   Addi r4 #64                                             001 100 011
   Addi r4 #64                                             001 100 011
   Addi r4 #2                                              001 100 111
   Addi r4 #1       //r4 is now 195                              001 100 001
   Load r1, mem[r4] //mem[195] or just that r4 holds address 195           101 001 100
   Addi r4 1                                               001 100 001
   Load r2, mem[r4]  //196                                       101 010 100
   XOR r2, r1                                               011 010 001
   Addi r4 1                                               001 100 001
   Load r1, mem[r4]  //197                                       101 001 100
   XOR  r1, r2                                              011 001 010
   Addi r4 1                                               001 100 001
   Load r2, mem[r4]   //198                                      101 010 100
   XOR r2, r1                                               011 010 001
   Store r2, mem[r4]  // storing the value of b8 for the next parity       100 010 100
   Addi r4 1                                               001 100 001
   Load r1, mem[r4] //199                                        101 001 100
   XOR r1, r2                                               011 001 010
   Addi r4 1                                               001 100 001
   Load r2, mem[r4] //200                                        101 010 100
   XOR r2, r1                                               011 010 100
   Addi r4 1                                               001 100 001
   Load r3, mem[r4] //201                                        101 011 100
   XOR r3, r2   // b5 is p8  and will be register 3                     011 011 010


   And r5 0                                                000 101 000
   Addi r5 #64                                              001 101 011
   Addi r5 #64                                              001 101 011
   Addi r5 #64                                              001 101 011
   Addi r5 #2                                              001 101 111
   Addi r5 #1       //r5 is now 195                              001 101 001
   Addi r5 2                                               001 101 111
   Addi r5 1                                               001 101 001
   Load r1, mem[r5]   //198                                      101 001 101
   Addi r5 5                                               001 101 101
   Load r2, mem[r5]  //202                                       101 010 101
   XOR r2, r1                                               011 010 001
   Addi r5 1                                               001 101 001
   Load r1, mem[r5]   //203                                      101 001 101
   XOR r1, r2                                               011 001 010
   Addi r5 1                                               001 101 001
   Load r4, mem[r5]  //204                                       101 100 101
   XOR r4, r1  // b2 is p4 and will be register 4                       011 100 001
```

```
And r7 0                                            010 111 000
Addi r7 #64                                          001 111 011
Addi r7 #64                                          001 111 011
Addi r7 #64                                          001 111 011
Addi r7 #2                                          001 111 111
Addi r7 #1      //R7 is now 195                            001 111 001
// may need to get the values again starting from loop label to end of storing b4.
Load r1, mem[r7]    //195                                 101 001 111
Addi r7 1                                           001 111 001
Load r2, mem[r7]   //196                                  101 010 111
XOR r2, r1                                           011 010 001
Addi r7 1                                           001 111 001
Addi r7 2                                           001 111 111
Load r1, mem[r7] //199                                    101 001 111
XOR r1, r2                                           011 001 010
Addi r7 1                                           001 111 001
Load r2, mem[r7]   //200                                  101 010 111
XOR r2, r1                                           011 010 111
Addi r7 2                                           001 111 111
Load r1, mem[r7]   //202                                  101 001 111
XOR r1, r2                                           011 001 010
Addi r7 1                                           001 111 001
Load r2, mem[r7]  //203                                   101 010 111
XOR r2, r1                                           011 010 001
Addi r7 2                                           001 111 111
Load r5, mem[r7]   //205                                  101 101 111
XOR r5, r2  // b1 is p2  and will be register 5                011 101 010


And r7 0                                            010 111 000
Addi r7 #64                                          001 111 011
Addi r7 #64                                          001 111 011
Addi r7 #64                                          001 111 011
Addi r7 #2                                          001 111 111
Addi r7 #1      //R7 is now 195                            001 111 001
// may need to get the values again  starting from loop label to end of storing b4.
Load r1, mem[r7]   //195                                  101 001 111
Addi r7 2                                           010 111 111
Load r2, mem[r7]   //197                                  101 010 111
XOR r2, r1                                           011 010 001
Addi r7 2                                           001 111 111
Load r1, mem[r7]  //199                                   101 001 111
XOR r1, r2                                           011 001 010
Addi r7 1                                           001 111 001
```

```
Load r2, mem[r7]   //201                              101 010 111
XOR r2, r1                                        011 010 001
Addi r7 1                                        001 111 001
Load r1, mem[r7]  //202                               101 001 111
XOR r1, r2                                        011 001 010
Addi r7 2                                        001 111 111
Load r2, mem[r7]   //204                              101 010 111
XOR r2, r1                                        011 010 001
Addi r7 1                                        001 111 001
Load r6, mem[r7]   //205                              101 110 111
XOR r6, r2  // b1 is p1  and will be register 6                011 110 010


And r7 0                                         010 111 000
Addi r7 #64                                       001 111 011
Addi r7 #64                                       001 111 011
Addi r7 #64                                       001 111 011
Addi r7 #2                                        001 111 111
Addi r7 #1       //R7 is now 195                          001 111 001
// may need to get the values again starting from loop label to end of storing b4.
Load r1, mem[r7]  //195                               101 001 111
Addi r7 1                                        001 111 001
Load r2, mem[r7]   //196                              101 010 111
XOR r2, r1                                        011 010 001
Addi r7 1                                        001 111 001
Load r1, mem[r7]    //197                             101 001 010
XOR r1, r2                                        011 001 010
Addi r7 1                                        001 111 001
Load r2, mem[r7]  //198                               101 010 111
XOR r2, r1                                        011 010 001
Addi r7 1                                        001 111 001
Load r1, mem[r7]   //199                              101 001 111
XOR r1, r2                                        011 001 010
Addi r7 1                                        001 111 001
Load r2, mem[r7]   //200                              101 010 111
XOR r2, r1                                        011 010 001
Addi r7 1                                        001 111 001
Load r1, mem[r7]   //201                              101 001 111
XOR r1, r2                                        011 001 010
Addi r7 1                                        001 111 001
Load r2, mem[r7]   //202                              101 010 111
XOR r2, r1                                        011 010 001
Addi r7 1                                        001 111 001
Load r1, mem[r7]   //203                              101 001 111
```

```
XOR r1, r2                                              011 001 010
Addi r7 1                                               001 111 001
Load r2, mem[r7]   //204                                    101 010 111
XOR r2, r1                                              011 010 001
Addi r7 1                                               001 111 001
Load r1, mem[r7]   //205                                    101 001 111
XOR r1, r2                                              011 001 010
Load r2, r3     // r3 is p8                                   101 010 011
XOR r2, r1                                              011 010 001
Load r1, r4    // r4 is p4                                    101 001 100
XOR r1, r2                                              011 001 010
Load r2, r5     //p2 is r5                                    101 010 101
XOR r2, r1                                              011 010 001
Load r7, r6    //r6 is p1 (based on prev. vals)                     101 111 110
XOR r7, r2   // p1 = p16  and will be register 7                      011 111 010

// may need to get the values again  starting from loop label to end of storing b4.

And r3, 0 //to be used for and later on                          010 011 000
Addi r3, 1                                              001 011 001

Load r1, mem[195]                                           101 001 ___ (TODO: mmry address)
AND r1, r3                                              010 001 011
<< r1, 1       shift left 1                                  111 1 01 001
And r7 0                                                010 111 000
Addi r7 #64                                              001 111 011
Addi r7 #64                                              001 111 011
Addi r7 #64                                              001 111 011
Addi r7 #2                                              001 111 111
Addi r7 #1       //R7 is now 195                              001 111 001
Addi r7 1                                               001 111 001
Load r2 mem[r7]   //196                                     101 010 111
AND r2, r3                                              010 010 011
Add r1, r2                                              000 001 010
<< r1, 1       shift left 1                                  111 1 01 001
Addi r7 1                                               001 111 001
Load r2 mem[r7]  //197                                      101 010 111
AND r2, r3                                              010 010 011
Add r1, r2                                              000 001 010
<< r1, 1       shift left 1                                  111 1 01 001
Addi r7 1                                               001 111 001
Load r2 mem[r7]  //198                                      101 010 111
AND r2, r3                                              010 010 011
Add r1, r2                                              000 001 010
```

```
<< r1, 1       shift left 1                          111 1 01 001
Addi r7 1                                           001 111 001
Load r2 mem[r7]  //199                                 101 010 111
AND r2, r3                                           010 010 011
Add r1, r2                                          000 001 010
<< r1, 1       shift left 1                          111 1 01 001
Addi r7 1                                           001 111 001
Load r2 mem[r7]  //200                                 101 010 111
AND r2, r3                                           010 010 011
Add r1, r2                                          000 001 010
<< r1, 1        shift left 1                         111 1 01 001
Addi r7 1                                           001 111 001
Load r2 mem[r7]   //201                                101 010 001
AND r2, r3                                           010 010 011
Add r1, r2                                          000 001 010
<< r1, 1       shift left  1                         111 1 01 001
Load r2 r3  //p8 is r3                               101 010 011
AND r2, r3                                           010 010 011
Add r1, r2   // b11 is the output [0]                     000 001 010

Store r1, mem[ i + 30]                                TODO: Lucy, why adding 30 here?//

Addi r7 1                                           001 111 001
Load r1 mem[r7]  //202                                 101 001 111
AND r1, r3                                           010 001 011
<< r1, 1       shift left 1                          111 1 01 001
Addi r7 1                                           001 111 001
Load r2 mem[r7]  //203                                 101 010 111
AND r2, r3                                           010 010 011
Add r1, r2                                          000 001 010
<< r1, 1       shift left 1                          111 1 01 001
Addi r7 1                                           001 111 001
Load r2 mem[r7]  //204                                 101 010 111
AND r2, r3                                           010 010 011
Add r1, r2                                          000 010 011
<< r1, 1       shift left 1                          111 1 01 001
Load r2 r4   //p4 is r4                              101 010 100
AND r2, r3                                           010 010 011
Add r1, r2                                          000 001 010
<< r1, 1       shift left 1                          111 1 01 001
Addi r7 1                                           001 111 001
Load r2 mem[r7]  //205                                 101 010 111
AND r2, r3                                           010 010 011
Add r1, r2                                          000 001 010
```

```
   << r1, 1       shift left 1                               111 1 01 001
   Load r2 r5  //p2 is r5                                   101 010 111
   AND r2, r3                                              010 010 011
   Add r1, r2                                              000 001 010
   << r1, 1        shift left 1                              111 1 01 001
   Load r2 r7   //r7 is p1                                  101 010 111
   AND r2, r3                                              010 010 011
   Add r1, r2                                              000 001 010
   << r1, 1       shift left  1                              111 1 01 001
   Load r2 r7   //r7 is p16                                  101 010 111
   AND r2, r3                                              010 010 011
   Add r1, r2   // b4  is the output [1]                           000 001 010

   Store r1, mem [ i + 31]                              TODO: Lucy, why adding 31 here?

   Ne r0 r8 (r8 predefined to be 28)                         110 01 00 __ (TODO: how to
represent this general purpose reg in 2 bits here?)
   Jump loop  // if r0 != r8 then loop again                      110 11 __ (TODO: add to jump's
LUT)
```

**Program 2:**

```
Addi r1 011          // Memory index variable r1 = 64 (LUT)
Addi r4 000          // Store parity (in decimal) r4 = 0 (LUT)
Addi r7 011          // r7 += 64 (64)
Addi r7 100          // r7 += 32 (96)
Addi r7 101          // r7 += 16 (112)
Addi r7 110          // r7 += 8 (120)
Addi r7 111          // r7 += 2 (122)
Add r8 r7            // r8 += r7
Addi r8 001          // r8 += 1
Addi r6 011          // r6 += 64 (64)
Addi r6 101          // r6 += 16 (80)
Addi r6 110          // r6 += 8 (88)
Addi r6 010          // r6 += 4 (92)

Loop_Condition:
   Load r1 r7        // Load mem[123] into r1
   Addi r6 001       // r6 += 1 (93)
   Neq r1 r6         // if r1 != 93 TODO FIX
```

```
    Jump Done            // False: we have finished all 15 decodings
    Jump Loop_Content          // True: decode next message

Loop_Content:
    Load r2 r1           // r2 set to memory at location r1
    Addi r1 001          // r1++, grab the next byte in data
    Load r3 r1           // r3 set to memory at location r1
    Addi r1 011          // r1++
    Store r2 r7          // Store r2 into mem[122]
    Store r3 r8          // Store r3 into mem[123]
    Jump Calc_p1        // Jump to calculate the first parity bit

Calc_p1:
    Addi r0 011          // r0 += 64
    Addi r0 011          // r0 += 64 (128)
    Addi r0 100          // r0 += 32 (160)
    Addi r0 110          // r0 += 8 (168)
    And r2 r0            // And LSW with decimal = 168
    Addi r0 111          // r0 += 2 (170)
    And r3 r0            // And MSW with decimal = 170
    Xor r2 r3            // Xor results of r2 and r3 and store into r2
    Lt r2 00             // if r2 < 1
    Addi r4 1            // False: r4 += 1
    Jump Calc_p2        // True: Move onto calculate parity bit 2

Calc_p2:
    Addi r0 100          // r0 += 32 (200)
    Addi r0 010          // r0 += 4 (204)
    Load r2 r7           // r2 = mem[122]
    And r2 r0            // And LSW with decimal = 204
    Load r3 r8           // r3 = mem[123]
    And r3 r0            // And MSW with decimal = 204
    Xor r2 r3            // Xor results of r2 and r3 and store into r2
    Lt r2 00             // if r0 < 1
    Addi r4 111          // False: r4 += 2
    Jump Calc_p2        // True: Move onto calculate parity bit 4

Calc_p4:
    Addi r0 100          // r0 += 32 (236)
    Addi r0 010          // r0 += 4 (240)
    Load r2 r7           // r2 = mem[122]
    And r2 r0            // And LSW with decimal = 240
    Load r3 r8           // r3 = mem[123]
    And r3 r0            // And MSW with decimal = 240
    Xor r2 r3             // Xor results of r2 and r3 and store into r2
    Lt r2 00              // if r0 < 1
    Addi r4 010           // False: r4 += 4
    Jump Calc_p8         // True: Move onto calculate parity bit 8

Calc_p8:
    Addi r0 110           // r0 += 8 (248)
    Addi r0 010           // r0 += 4 (252)
    Addi r0 111           // r0 += 2 (254)
    Addi r0 001           // r0 += 1 (255)
    Load r2 r7            // r2 = mem[122]
    Load r5 r7            // r5 = mem[122]
    And r2 r0             // And LSW with decimal = 255
    Load r3 r8            // r3 = mem[123]
    And r0 0              // Set r0 to 0
    And r3 r0             // And MSW with decimal = 0
    Xor r2 r3             // Xor results of r2 and r3 and store into r2
    Lt r2 00              // if r0 < 1
    Addi r4 110           // False: r4 += 8
    Jump Calc_p16         // True: Move onto calculate parity bit 8

Calc_p16:
    Addi r0 011           // r0 += 64 (64)
    Addi r0 011           // r0 += 64 (128)
    Addi r0 011           // r0 += 64 (192)
    Addi r0 100           // r0 += 32 (224)
    Addi r0 101           // r0 += 16 (240)
    Addi r0 110           // r0 += 8 (248)
    Addi r0 010           // r0 += 4 (252)
    Addi r0 111           // r0 += 2 (254)
    Addi r0 001           // r0 += 1 (255)
    And r2 r0             // And LSW with decimal = ??
    And r3 r0             // And MSW with decimal = ??
    Xor r2 r3             // Xor results of r2 and r3 and store into r2
    Lt r2 00              // if r2 < 1
    Addi r4 101           // False: r4 += 16
    Jump Comp_parity              // True: Compare calculated parity to actual parity

Comp_parity:
    And r0 000            // Set r0 to 0
    Addi r0 101           // r0 += 16 (16)
    And r7 000            // Set r7 to 0
    Add r7 r4             // r7 = r4
    And r7 r0             // Get calculated global parity p16
    And r0 000            // Set r0 to 0
    Addi r0 001           // r0 += 1 (1)
    And r5 r0             // Get message global parity p16
    Neq r5 r7             // if r1 != r0
    Jump Done             // False: r1 = r0 and two bit error
    Eq r7 011             // True: one bit correction; check if p16 == 1
    Addi r4 0             // False: p16 != 1, do nothing

// TODO I DONT KNOW HOW TO DO (SOLUTION: And with 11101111)
    And r4                // r4 -= 16
    Lt r4 10              // if r4 < 8
    Xor r0 r3 r4          // False: r4 >= 8, Xor r3 by r4 parity
    Xor r0 r2 r4          // True: r4 < 8, Xor r2 by r4 parity
    Jump Loop_Condition        // Check if there are more messages to decode, do nothing

Done:
```

**Program 3: (any branch format, the next line is for false, the line after is for when it is true)**
**Assuming we can also add to memory by 1.**
**Can't yet calculate the addresses so for now putting in labels to jump to**

```
Load r0 mem[160]  ## loads in pattern at mem160                        101 000
>>    r0  5   ## getting the 5 bit pattern of mem[160]                 11101 00
Addi r2, 0  - >this represents i                                       001 010 000
Addi r3, 0 -> this represents j                                        001 011 000
Addi r4, 1 -> this represents z                                        001 100 000
Addi r5, 0  -> this represents the # of time pattern shows up          001 101 001
Addi r6, 0  ->  this represent the # of bytes that the pattern shows up     001 110 000
Loop1:
        Lt, r2, 32       ## if i < 32                                  11000 10
        Jump Done    ## when this is false                             11001
        Load r1 r2    ##  when true, loads the i byte of memory        101 001 010
        Jump loop2: ## go to the next for loop                         11001
End1:
        Addi r2 1      ##  i ++                                        001 010 001
        Addi r6 1 ## add 1 to the memory of total bytes of patten      001 110 001
        Jump Loop1                                                     11001
Loop2:
        Lt r3 8         ## if j < 8                                    11000 11 10
        Jump End1       ## when this is false                          11001
        Eq r0, r1      ## if  r0 == r1                                   11011 00 01
        Jump End 2       ## when this is false                           11001
        Jump loop 3                                                      11001
End2:
        Addi r3 1     ## j  ++                                           001 011 001
        Jump Loop2                                                       11001
Loop3:
        Lt r4, 5       ## if z< 5                                        11000
        Jump End3           ## when this is false                        11001
        Neq r1 r0   ## if r1 ! = r0                                      11010 01 00
        Jump  If       ## when this is false                             11001
        Jump End2                                                        11001
If:
        Addi r4 1  ## z ++                                               001 100 001
End3:
        Addi r5 ,1   add 1 to the memory for total count of pattern      001 101 001
        Jump End2:                                                       11001
Done:
        Store r5 MEM[ 192]                                               100 101
        Store  r6 MEM[193]                                               100 110
```

| Opcode | Instructions | input1 / output | Input2 | LUT |
|---|---|---|---|---|
| 000 | Add | r1 | r2 | |
| 001 | Addi | r1 | # | Yes (8-16) |
| 010 | AND | r1 | r2 | |
| 011 | XOR | r1 | r2 | |
| 100 | Store | r1 (3 bit) | r2 (3 bits) register have address of memory | |
| 101 | Load | r1 (3 bits) | r2 (3 bits) registers have address of memory | |
| 110+ (00) | Lt | r1 (2 bits) | #(2 bits) | Yes (4) |
| 110+ (01) | Neq | r1 (2 bits) | r2 (2 bits) | |

| | | | | |
|---|---|---|---|---|
| 110+ (10) | Eq | r1 (2 bits) | r2 (2 bits) | |
| 110 + (11) | Jump | # (4 bits) | X | Yes (16) |
| 111+ (0) | >> | r1(2 bits) | # (3 bit) | |
| 111 + (1) | << | r1 (2 bits) | # (3 bit) | |
| All 1111111 | Ack, means machine is done | | | |

| LUT for Addi | The immediate value |
|---|---|
| 000 | 0 |
| 001 | 1 |
| 010 | 4 |
| 011 | 64 |
| 100 | 32 |
| 101 | 16 |
| 110 | 8 |
| 111 | 2 |

| LUT for lt (2 bits) | immediate /register value |
|---|---|
| 00 | 1 |
| 01 | 5 |
| 10 | 8 |
| 11 | 32 |

| LUT for Jump (4 bits) | immediate / register value |
|---|---|
| 0000 | LoopCondition |
| 0001 | LoopContent |
| 0010 | CalcP1 |
| 0011 | CalcP2 |
| 0100 | CalcP4 |
| 0101 | CalcP8 |
| 0110 | CalcP16 |
| 0111 | Compparity |
| 1000 | DoneProgram2 |
| 1001 | Loop1 |
| 1010 | End1 |
| 1011 | Loop2 |
| 1100 | End2 |
| 1101 | Loop3 |
| 1110 | End3 |
| 1111 | If |
| ??? | Done Program3 |

## MILESTONE TWO

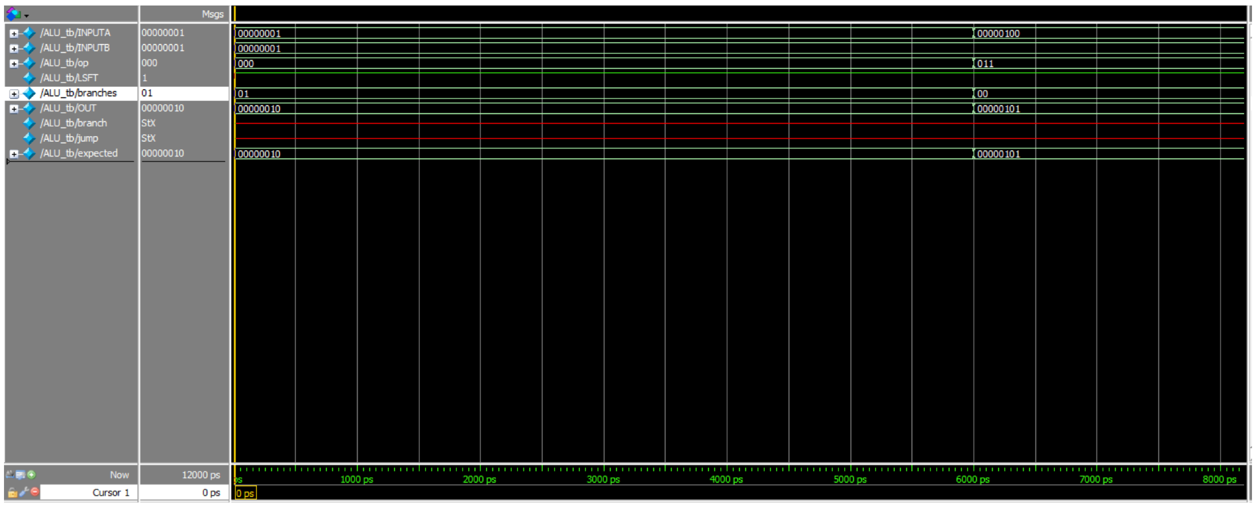
**ALU.sv testbench**
ALU operations/ Instructions we will be testing

- Add r1, r2
- Addi r1, r2

- And r1, r2
- Xor r1, r2
- Lt r1, r2
- Neq r1, r2
- Eq r1, r2
- >> r1, r2
- << r1, r2

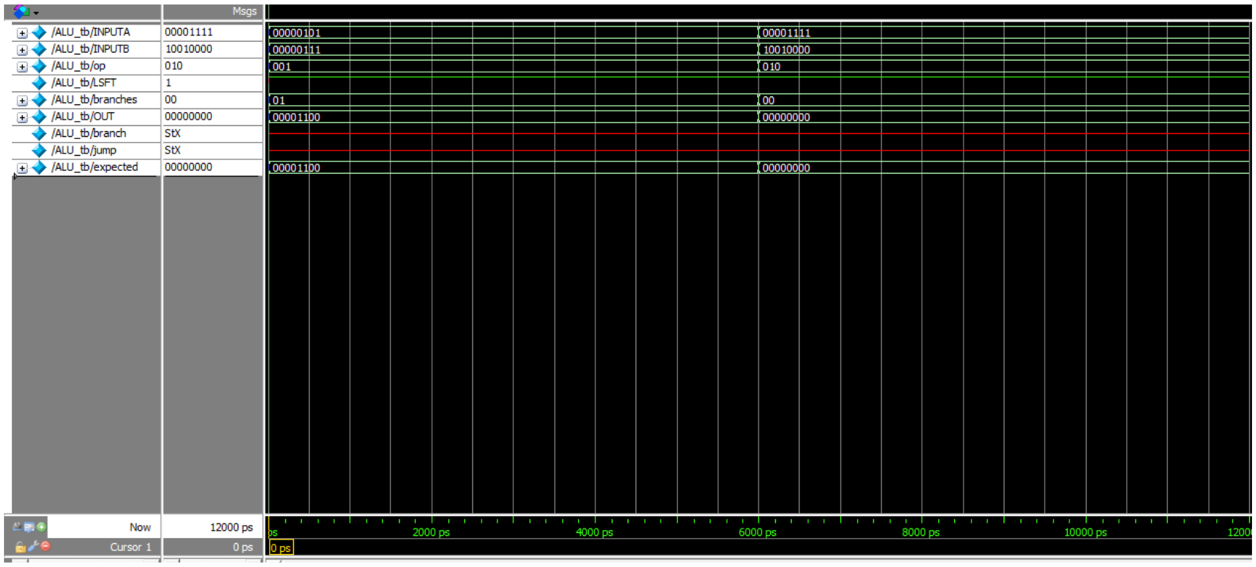
ADD for the first test case is working fine. We decided to have ADD be an opcode of 000 and this is one of the instructions that does not need LFST or branches so those are irrelevant. I take in two registers and add them.
XOR for the second case is also working fine. This instruction has an opcode of 011 and also does not use branch or LFST. This also takes in two registers as inputs.

```
VSIM 7> run -all
#              1000 YAY!! inputs = 01 01, opcode = 000, LFST = 1, branches = 01, answer = 00000010
#              7000 YAY!! inputs = 04 01, opcode = 011, LFST = 1, branches = 00, answer = 00000101
```

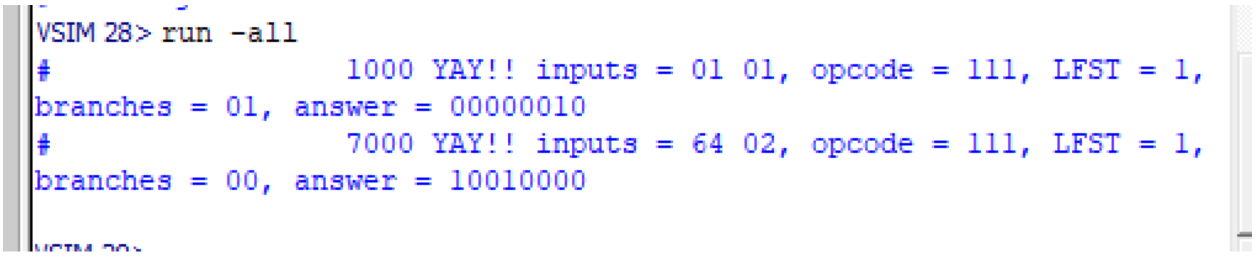


ADDI was the first test case and works fine. We decided to have ADDI be an opcode of 001, and this is one of the instructions that also does not need LSFT and branches. There are two registers one is for the output and input  and the other is a register filled with immediate values to add. We currently do not have LUT implemented so for now it has the same functionality as ADD.
AND was the second test case and also works fine. It has an opcode of 010 and it does not need LSFT and branches IIt takes in two registers as input.

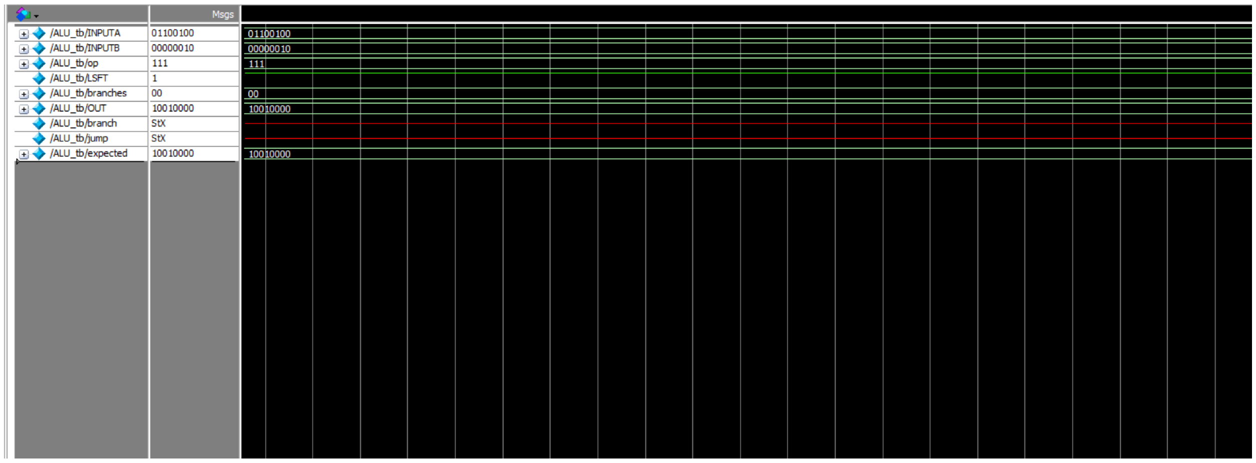
```
I 12> run -all
                1000 YAY!! inputs = 05 07, opcode = 001, LFST = 1, branches = 01, answer = 00001100
                7000 YAY!! inputs = 0f 90, opcode = 010, LFST = 1, branches = 00, answer = 00000000
```



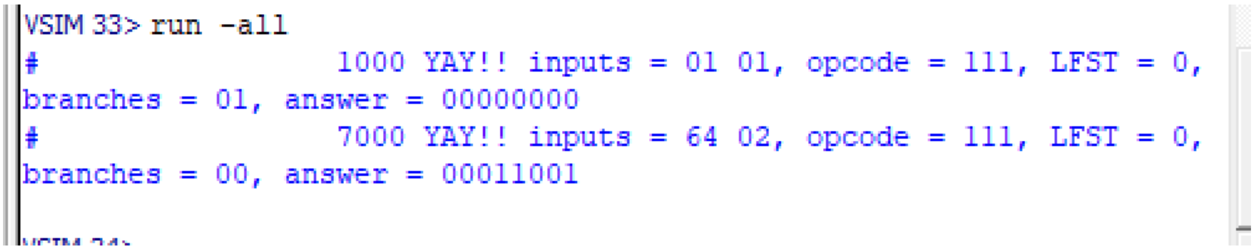
Left shift for both the first and second case and it worked as expected. This also takes in LFST where if it is 1, then you left shift and the opcode for shifts is 111.  So here we tested if it shifts left based on InputB.



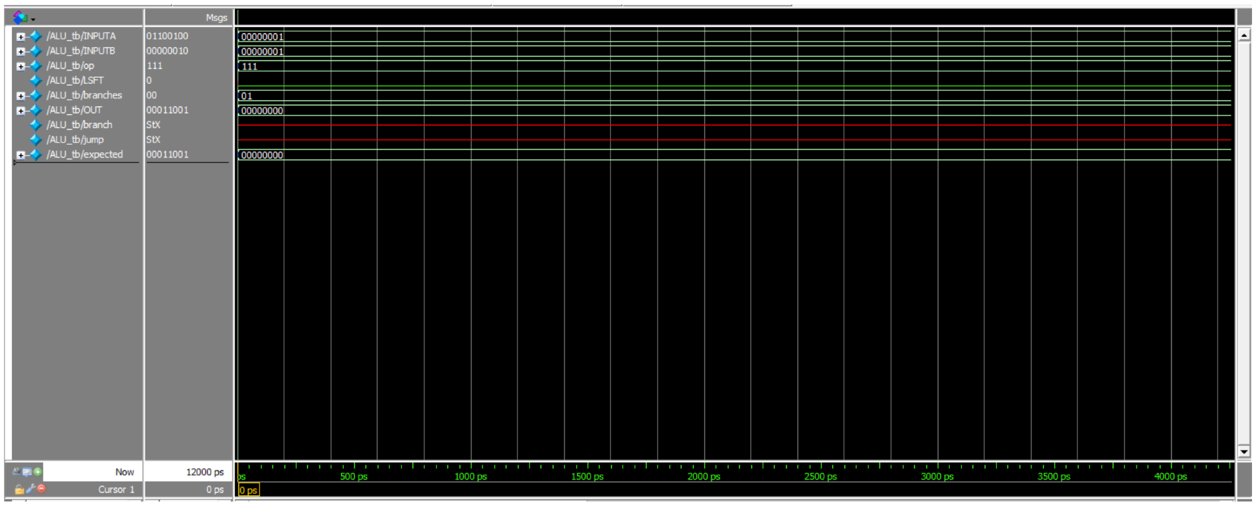
```
VSIM 28> run -all
#                    1000 YAY!! inputs = 01 01, opcode = 111, LFST = 1,
branches = 01, answer = 00000010
#                    7000 YAY!! inputs = 64 02, opcode = 111, LFST = 1,
branches = 00, answer = 10010000
```



Right shift for both the first and second case worked as expected This is still an opcode of 111 and this time to get a right shift, LSFT needs be 0. And then the Input A will shift right based on Input B.

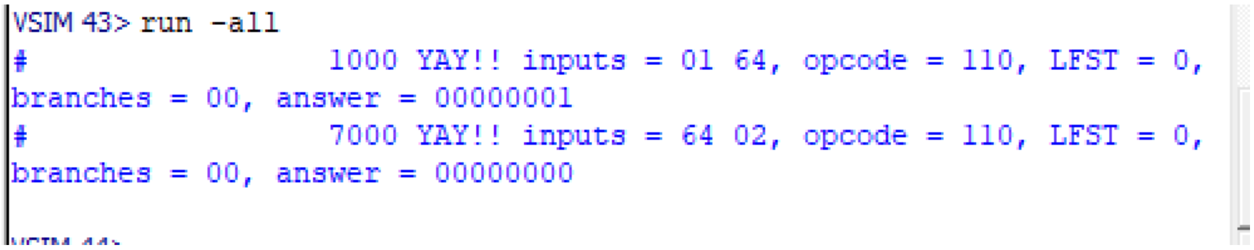


```
VSIM 33> run -all
#                    1000 YAY!! inputs = 01 01, opcode = 111, LFST = 0,
branches = 01, answer = 00000000
#                    7000 YAY!! inputs = 64 02, opcode = 111, LFST = 0,
branches = 00, answer = 00011001
```

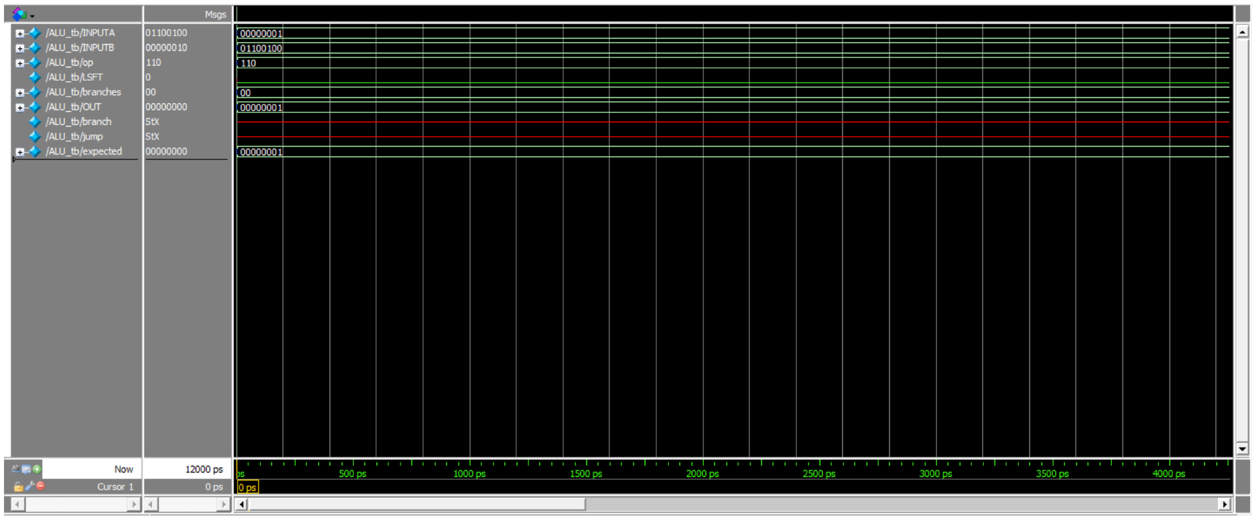


```
                        Msgs
/ALU_tb/INPUTA          01100100    00000001
/ALU_tb/INPUTB          00000010    00000001
/ALU_tb/op              111         111
/ALU_tb/LSFT            0
/ALU_tb/branches        00          01
/ALU_tb/OUT             00011001    00000000
/ALU_tb/branch          StX
/ALU_tb/jump            StX
/ALU_tb/expected        00011001    00000000

Now       12000 ps
Cursor 1  0 ps
```

Less than branch is working as expected The opcode is 110 and here we decided that for branches flag, we need to have it at 00 to test for less than with the two inputs. We tested two different cases where inputA is greater than inputB and when inputB is greater than Input A.

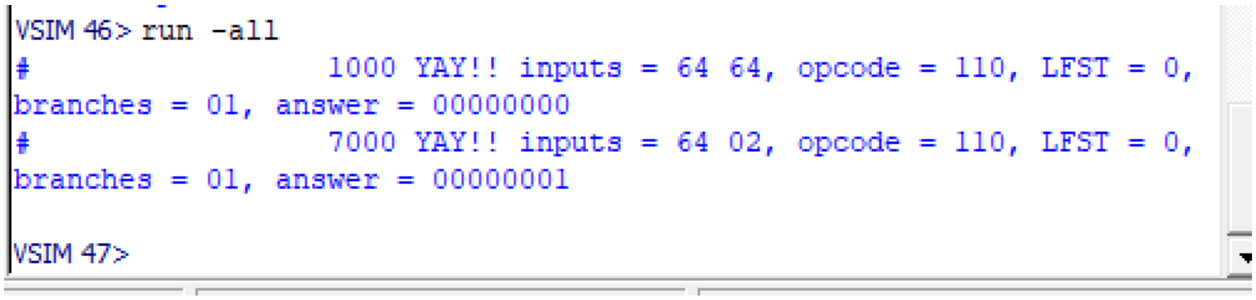


```
VSIM 43> run -all
#                    1000 YAY!! inputs = 01 64, opcode = 110, LFST = 0,
branches = 00, answer = 00000001
#                    7000 YAY!! inputs = 64 02, opcode = 110, LFST = 0,
branches = 00, answer = 00000000
```

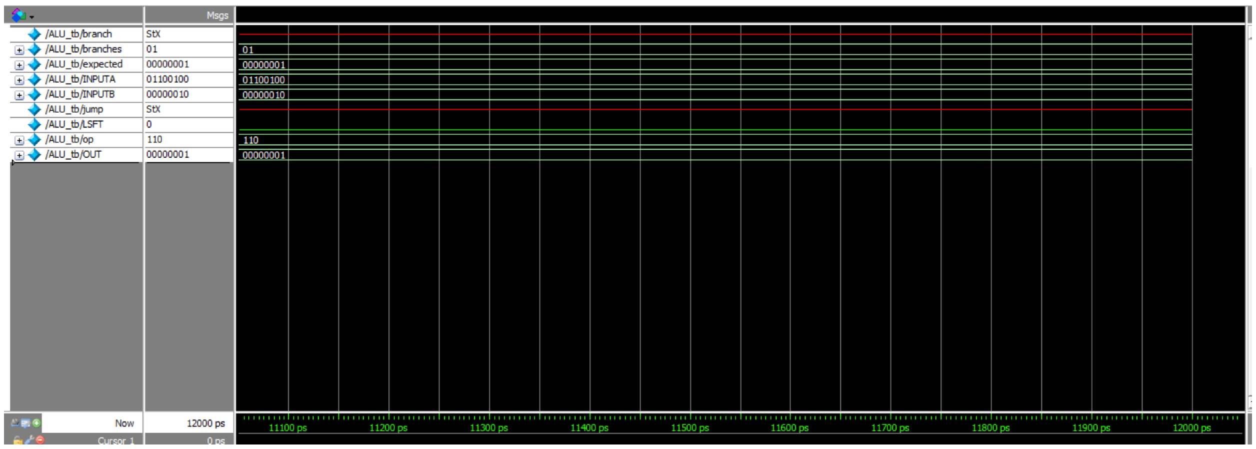


```
/ALU_tb/INPUTA     01100100   00000001
/ALU_tb/INPUTB     00000010   01100100
/ALU_tb/op         110        110
/ALU_tb/LSFT       0
/ALU_tb/branches   00         00
/ALU_tb/OUT        00000000   00000001
/ALU_tb/branch     StX
/ALU_tb/jump       StX
/ALU_tb/expected   00000000   00000001
Now       12000 ps
Cursor 1  0 ps
```

Not equal is working as expected. The opcode is 110 and here we also decided that for the branch flag, we need to have it at 00 to test out not equal with two inputs. We tested two different inputs for this case. The first one (as seen below) is making sure that it returns 0 when the two inputs are equal to each other and that is the case we get. Similarly, the second inputs are testing for the opposite case where the two inputs are not equal to each other and in that case we received 1 as the output just as it was expected.

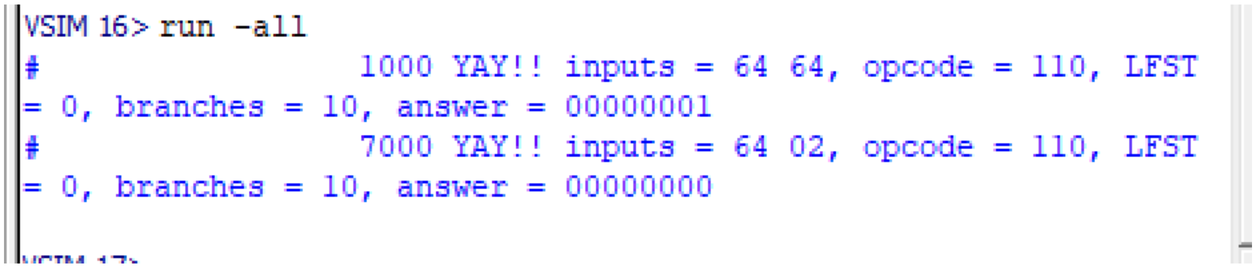


```
VSIM 46> run -all
#                    1000 YAY!! inputs = 64 64, opcode = 110, LFST = 0,
branches = 01, answer = 00000000
#                    7000 YAY!! inputs = 64 02, opcode = 110, LFST = 0,
branches = 01, answer = 00000001

VSIM 47>
```

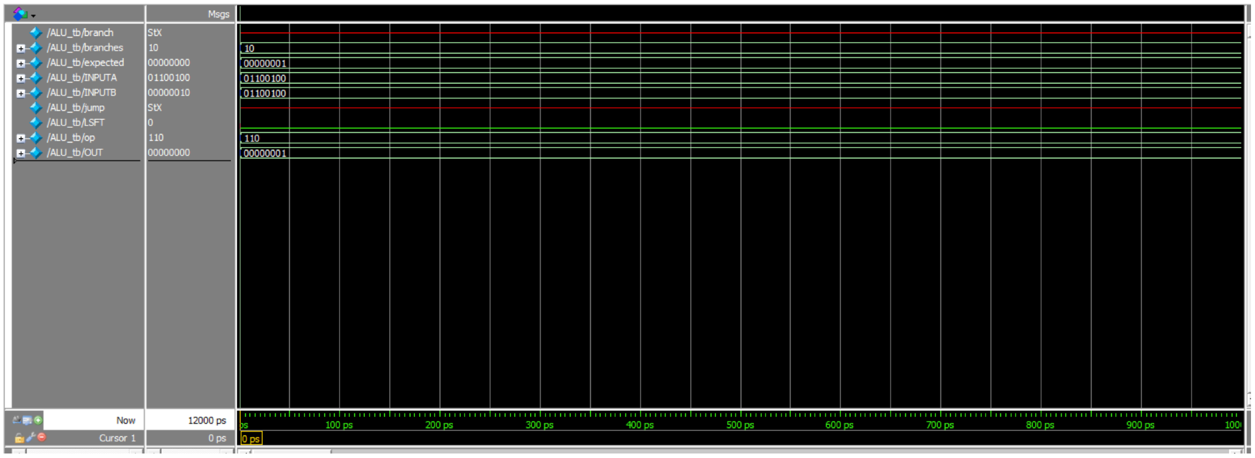


```
/ALU_tb/branch     StX
/ALU_tb/branches   01         01
/ALU_tb/expected   00000001   00000001
/ALU_tb/INPUTA     01100100   01100100
/ALU_tb/INPUTB     00000010   00000010
/ALU_tb/jump       StX
/ALU_tb/LSFT       0
/ALU_tb/op         110        110
/ALU_tb/OUT        00000001   00000001
Now       12000 ps
Cursor 1  0 ps
```

Equal branch works as expected. The opcode for branches is 110 and to choose equals comparison, the branches need to be 10. This takes in two registers which are inputA And inputB. There were two cases we tested which is when the inputs actually equal each other and outputted 1 and then we tested the inputs that are not equal which did end up outputting 0.



```
VSIM 16> run -all
#                    1000 YAY!! inputs = 64 64, opcode = 110, LFST
= 0, branches = 10, answer = 00000001
#                    7000 YAY!! inputs = 64 02, opcode = 110, LFST
= 0, branches = 10, answer = 00000000
```



```
                    Msgs
/ALU_tb/branch      StX
/ALU_tb/branches    10         10
/ALU_tb/expected    00000000   00000001
/ALU_tb/INPUTA      01100100   01100100
/ALU_tb/INPUTB      00000010   01100100
/ALU_tb/jump        StX
/ALU_tb/LSFT        0
/ALU_tb/op          110        110
/ALU_tb/OUT         00000000   00000001

Now       12000 ps
Cursor 1  0 ps
```

**<u>ProgCntr.sv waveform</u>**
**Program Counter Test Bench:**

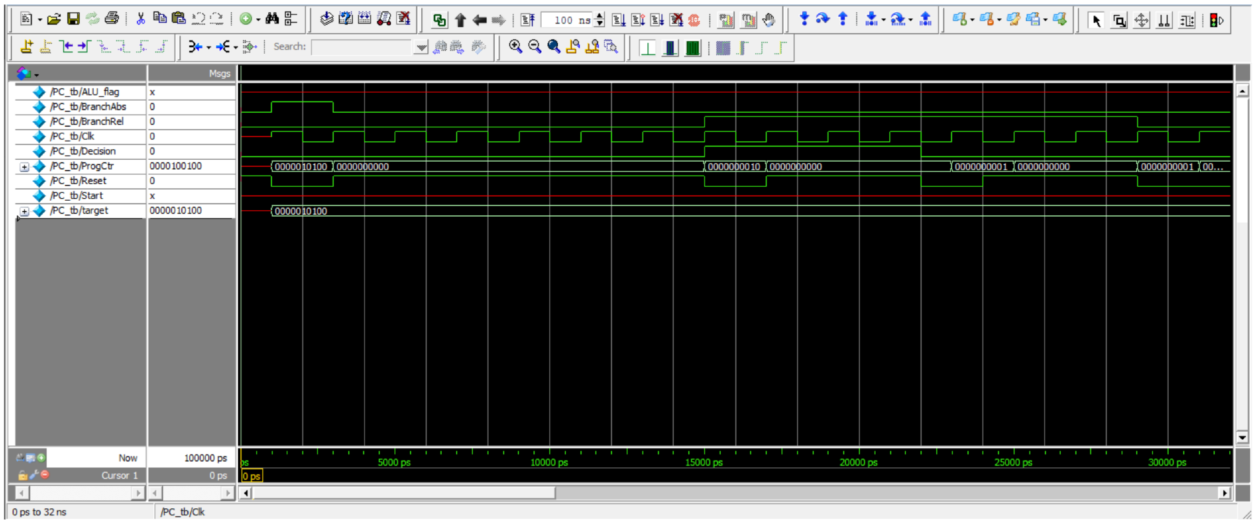
timeunit 1ns;
timeprecision 1ps;

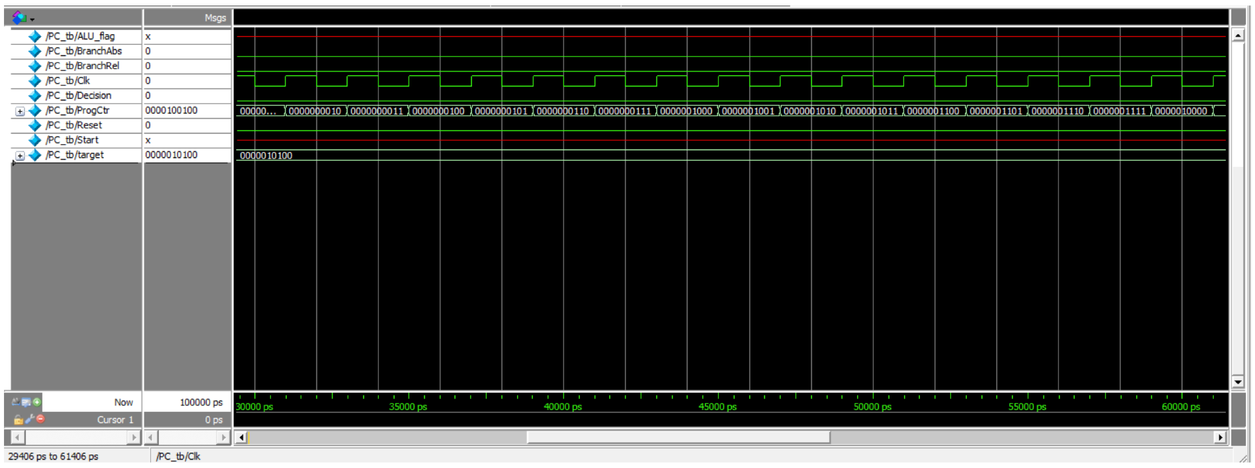
//Test bench
//Arithmetic Program Counter
/*
* INPUT: A, B
* op: 000, A ADD B
* op: 100, A_AND B
* ...
* Pleaser refer to definitions.sv for support ops(make changes if necessary)
* OUTPUT A op B
* equal: is A == B?

```
* even: is the output even?
*/


module PC_tb;
logic Reset;
logic Start;
logic Clk;
logic BranchRel;
logic BranchAbs;
logic Decision;
logic ALU_flag;
logic [9:0] target;
logic [9:0] ProgCtr;

// CONNECTION
ProgCtr uut(
  .Reset(Reset),
  .Start(Start),
  .Clk(Clk),
  .BranchRel(BranchRel),
  .BranchAbs(BranchAbs),
  .Decision(Decision),
  .ALU_flag(ALU_flag),
  .Target(target),
  .ProgCtr(ProgCtr)
    );


initial begin

 BranchRel = 0;
 BranchAbs = 0;
 Decision = 0;
 Reset = 1;
 #1;
 Reset = 0;
 BranchAbs = 1;
 target = 20;
 test_pc_func;
 #1;
 Reset = 1;
 BranchAbs = 0;
 #5;
 test_pc_func;
 #5;

 #1;
 Reset = 0;
 BranchRel = 1;
 Decision = 1;
 test_pc_func;
 #1;
 Reset = 1;
 #5;
 Reset = 0;
 Decision = 0;
 test_pc_func;
 #1;

 Reset = 1;
 #5;
 Reset = 0;
 BranchRel = 0;
 test_pc_func;
 end

 always begin
    #1 Clk = 'b1;
    #1 Clk = 'b0;
 end

 task test_pc_func;
 begin
   #1;
   $display("%t ProgCtr = %d", $time, $ProgCtr);
 end
 endtask

endmodule
```

**Explanation:**
The testbench will test the ProgCtr module for correct implementation based on various inputs. The first value of ProgCtr in the output of the waveform is the bit string representing the number 20. This is because on our first iteration of calling the test function, the program counter will take

the unconditional branch to the target value, which the testbench has set to 20. This means that this part of the program counter works. The test for when reset is 0 also works because various times, such as at time = 2ns, the ProgCtr value drops to 0 again, as is intended in the ProgCtr module. Next, we tested the relative branches. The first is if the decision is true/high which means that our program counter should increment by 2. Since we reset our ProgCtr value to 0 due to the reset, we should expect a value of 2 at time = 13ns, which is aptly represented in the first waveform. Afterwards, we reset back to 0, and then we set decision = 0/low to test the opposite case. In this case, we should only increment the ProgCtr value by 1 to go to the next instruction, which in the waveform happens at time = 19ns. Lastly, we test the case where all of the branches (relative and absolute branch) are low which means that the program counter defaults to the standard increment of 1 instruction. This is best shown in the second screenshot where the value of ProgCtr increments by 1 each clock cycle. The reason this keeps incrementing is because our testbench did not set the reset value back to high to stop updating the program counter values, and thus it continues to update until 100ns (since that is the runtime that we set for our testbench synthesis).





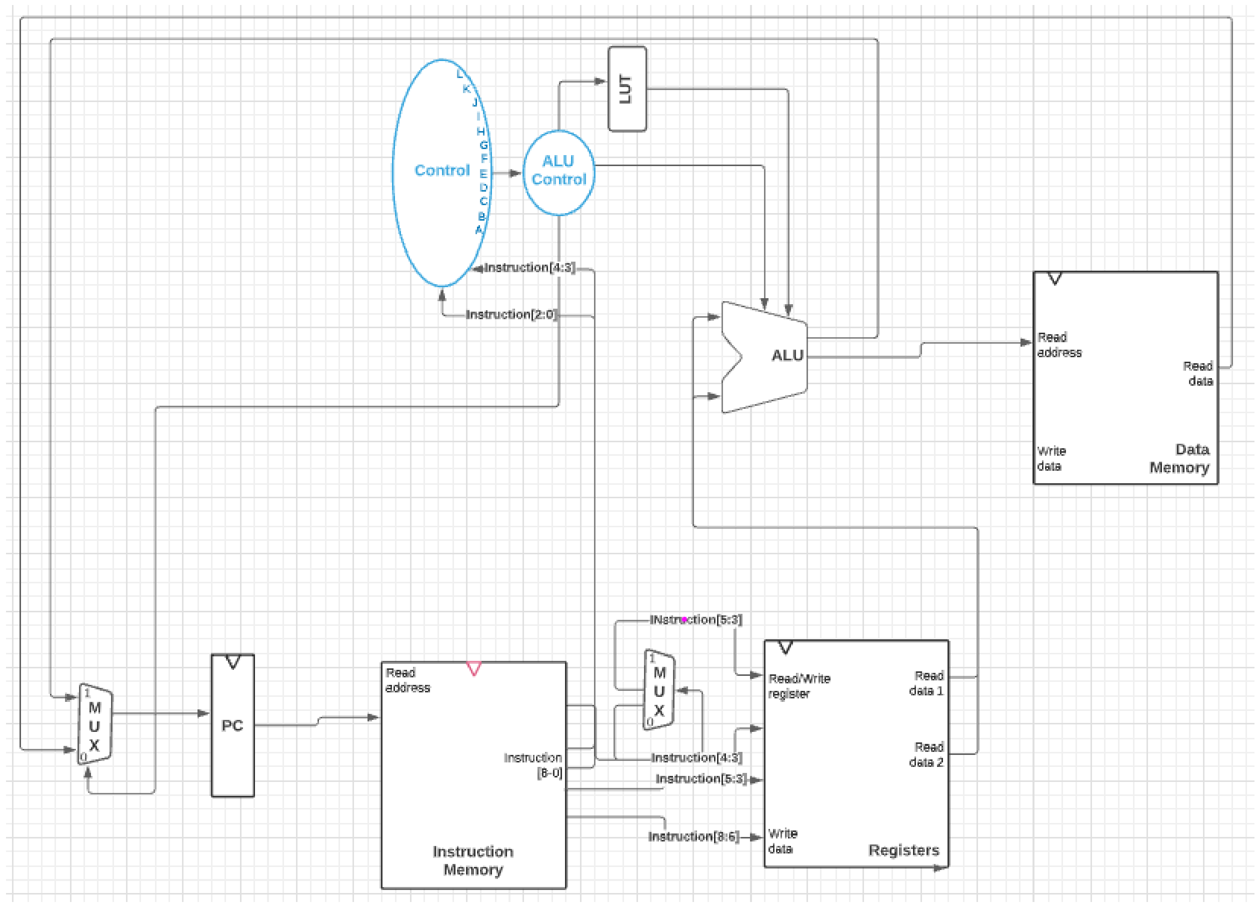
**All Verilog files:**

Attached source code

**Question:** 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?

We do use  non-arithmetic instructions for PC relative branches because for all our branches, we decided that if this was not taken, PC will go to the next instruction, which usually is a jump instruction. If it is taken, we need to go to the second instruction after. This complicates our design because it requires us to use jump instructions more often which may be harder to keep track of.

**Architectural Overview**



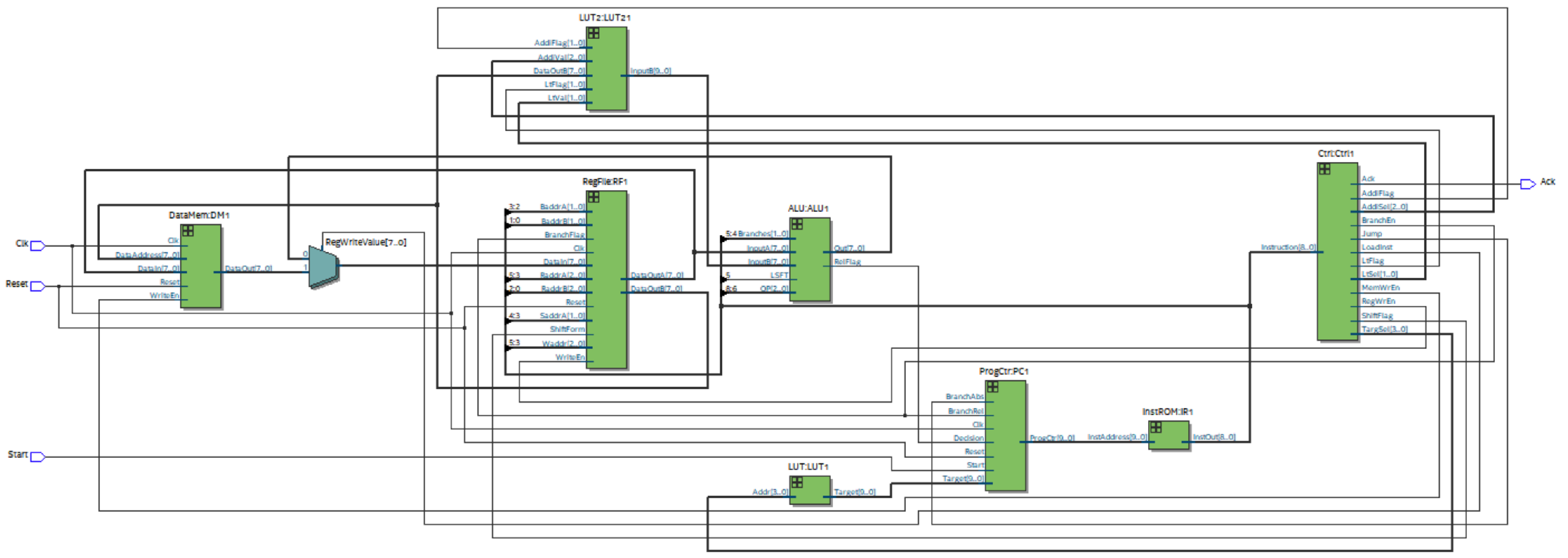
**Changelog:**
All new changes are highlighted in the file above. For quick reference, here are some of the major changes we made:

- We updated the instruction format along with the opcode information

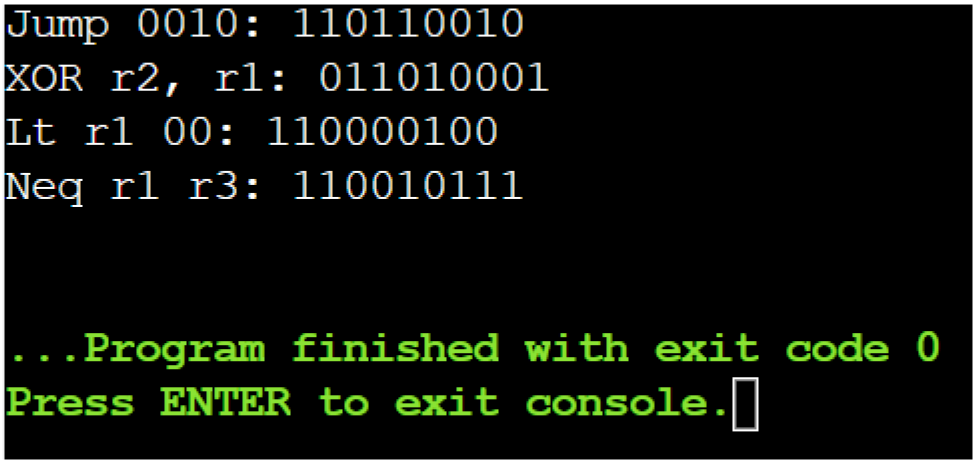
- We updated the control flow section in order to explain more about how target addresses are calculated
- Program 1 is completely rewritten from scratch

# **Milestone 3**

Architecture overview changes:



Example of Input and Output of assembler:



```
Jump 0010: 110110010
XOR r2, r1: 011010001
Lt r1 00: 110000100
Neq r1 r3: 110010111


...Program finished with exit code 0
Press ENTER to exit console.
```

ChangeLog:
- Updated the Architecture Overview
- Added in the machine code next to the assembly code back in milestone1

- Changed the assembly code, minor fixes for all program except for Program2 code, where most of the code was rearranged
- Created assembler.py

**Assembler.py**

```python
def assembler(insn):
    # file = open(filename)

    operands = insn.split(' ')

    # PARSE OPCODES
    opcode_switch = {
        'Add': '000',
        'Addi': '001',
        'AND': '010',
        'XOR': '011',
        'Store': '100',
        'Load': '101',
        'Lt': '11000',
        'Neq': '11001',
        'Eq': '11010',
        'Jump': '11011',
        '>>': '1110',
        '<<': '1111',
        'Ack': '111111111'
    }
    opcode = opcode_switch.get(operands[0])

    ### PARSE FIRST OPERAND ###
    # CASE 1: FIRST OPERAND IS REGISTER
    op1 = ''
    if ((operands[1])[0] == 'r'):
        decimal = int((operands[1])[1])
        binary = []
        while (decimal > 0):
            divided = decimal % 2
            binary.append(divided)
            decimal = decimal // 2

        while len(opcode) + len(binary) < 7 and len(binary) < 3:
```

```
        binary.append(0)
    binary.reverse()
    binaryString = ''
    for i in binary:
        binaryString += str(i)

    op1 = binaryString

# CASE 2: FIRST OPERAND IS LUT VALUE
else:
    op1 = operands[1]



### PARSE SECOND OPERAND ###
op2 = ''
if (opcode != '11011'):
    # CASE 1: SECOND OPERAND IS REGISTER
    if ((operands[2])[0] == 'r'):
        decimal = int((operands[2])[1])
        binary = []
        while (decimal > 0):
            divided = decimal % 2
            binary.append(divided)
            decimal = decimal // 2

        while len(opcode) + len(binary) < 7 and len(binary) < 3:
            binary.append(0)
        binary.reverse()
        binaryString = ''
        for i in binary:
            binaryString += str(i)

        op2 = binaryString

    # CASE 2: SECOND OPERAND IS LUT VALUE
    else:
        op2 = operands[2]
```

```python
    ### CONCATENATE MACHINE CODE TRANSLATION ###
    machine_code = opcode + op1
    if (op2 != ''):
        machine_code += op2

    # Add extra 0's if not filled
    while len(machine_code) < 9:
        machine_code += '0'

    return machine_code


### TESTING ###
test_insn = 'Jump 0010'
translation = assembler(test_insn)
print(test_insn + ": " + translation)

test_insn = 'XOR r2, r1'
translation = assembler(test_insn)
print(test_insn + ": " + translation)

test_insn = 'Lt r1 00'
translation = assembler(test_insn)
print(test_insn + ": " + translation)

test_insn = 'Neq r1 r3'
translation = assembler(test_insn)
print(test_insn + ": " + translation)
```