**Project 3: Report**

**Chris Escobar, Chloe Malone, Dimo Kartelyan**

**I) Table of group activity**

|  |  |  |
| --- | --- | --- |
| **Meeting Day** | **Group Members** | **Discussed** |
| Wednesday, Oct 23 | Chloe, Chris, Dimo | Discussed plan of action, Assigned basic duties |
| Monday, Oct 28 | Chloe, Chris | Discussed written python code and who’s we were going to use  Explained special instructions to be on the same page |
| Wednesday, Oct 30 | Chris, Dimo | Discussed how written python code will translate into hardware designs (special instructions into ALU layout), etc. |
| Wednesday, Nov 6 | Chloe, Chris | Optimize the python code to reduce DIC, while keeping most of our hardware the same |
| Thursday, Nov 7 | Chloe, Chris, Dimo | Finished up Hardware Design  Made sure code was fully debugged  Finished Report |

**II) Individual activity list**

**Chloe:**

Small Hardware design

Small debugging

Minor corrections to python to reduce instruction count

FUNC python implementation

Initial 8-bit Instruction Implementation

**Chris:**

Almost all of Python Code

Reduced python to handle 8 bits

Reduced FUNC to be 8 bits as well

Fancy output

Final 8-bit Instruction Implementation

**Dimo:**

Main Hardware Designer

**III) Group Work Reflection**

1. How does this group differ from the previous one?
2. List out one new or important thing that you have learned from this group
3. If you were to do this project all by yourself, how different would it be? Why?

**[ISA Design]**

**Unique name:** smol\_8oi

**Memory Addressing Modes:** We do not load from memory, we just store with FUNC. FUNC uses mem[0x4 + (A-1)] = C to store.

**PC updating mechanism:** We used a special instruction that took an immediate, and added the immediate value to PC if the condition was met.

**PART A**

**Introduction & Instruction list:**

We named our new architecture smol\_8oi (small boy). Our architecture strives to work with the least possible amount of instructions. We managed to achieve a DIC of 515. This was a big goal of ours that we were able to achieve. Our instructions, formats, and opcodes are included below, as well as an example of one of our .asm codes.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instruction** | **Functionality** | **Encoding Format** | **MC Example** | **Asm Example** |
| **add** Rx, Ry | Rx = Rx + Ry; PC++ | 0100 xx yy | 01001001 | add $2, $1 |
| **sbnezR2** imm | R2 = R2 -1  If R2 != 0,  PC = PC + imm  Else, PC++  Imm: [-8, 7] | 1011 iiii | 10111110 | sbnezR2 -2 |
| **luiR1** imm | R1 = iiii0000 | 0110 iiii | 01101111 | luiR1 15 |
| **oriR1** imm | R1 = R1 | iiii | 0111 iiii | 01111111 | oriR1 15 |
| **func** Rx, Ry, Rz | Rx = hash(Rx, Ry) | 00 xx yy zz | 00111001 | func $3, $2, $1 |

**Register design:**

We have 5 supported registers. R0-R3, and R4, which is our PC. R1 is reserved for LUIR1 and ORIR1, then R1 is used in FUNC for our starting value to be folded.

**Branch design:**

We support SBNEZR2, which subtracts 1 from $2, then branches by adding immediate value to R3 (AKA PC) if $2 is not equal to zero. The maximum branch distance supported is [-8, 7], AKA back 8 instructions or forward 7 instructions.

**Data memory addressing modes:**

We use LUIR1 to access memory and load upper immediate to R1. Also, FUNC writes to memory when finished. We can access all 0x1000 (4096) addresses dedicated to this design

**Most significant features of our ISA:**

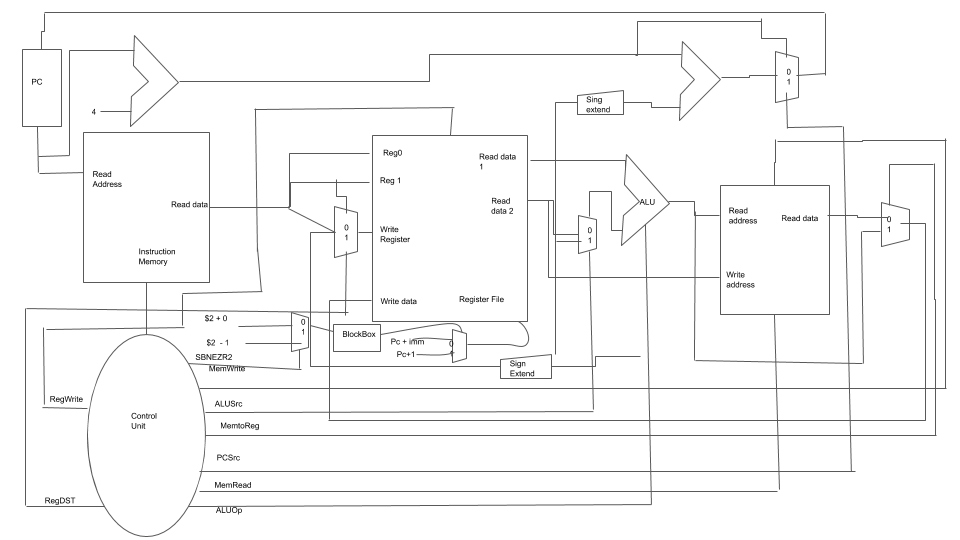
Hardware was a significant feature in our ISA since we had many special instructions. This was not difficult to implement, however. Our ISA is very specialized to hashing, so it was very easy to make an ASM file specifically for hashing, and our python was easy to write (and pretty straightforward) as well.

As for low DIC, we were definitely able to achieve it. We did this by having very specialized instructions and managed to bring our DIC to 515. As for HW, we needed more due to our special functions, however, it was not hard to implement. We used more ALU’s, MUX’s, and Black Boxes with descriptions of what they do.

Our limitations lie in our ISA overall. Since the instructions are very HASH focused, the ISA is pretty much only usable for HASH functions. We compromised by making very specific instructions that are only usable for a small number of things.

**PART B**

**CPU Datapath Design:**

****

SBNEZR2:

$r2 = $r2 - 1;

if $2 != 0 PC += imm;

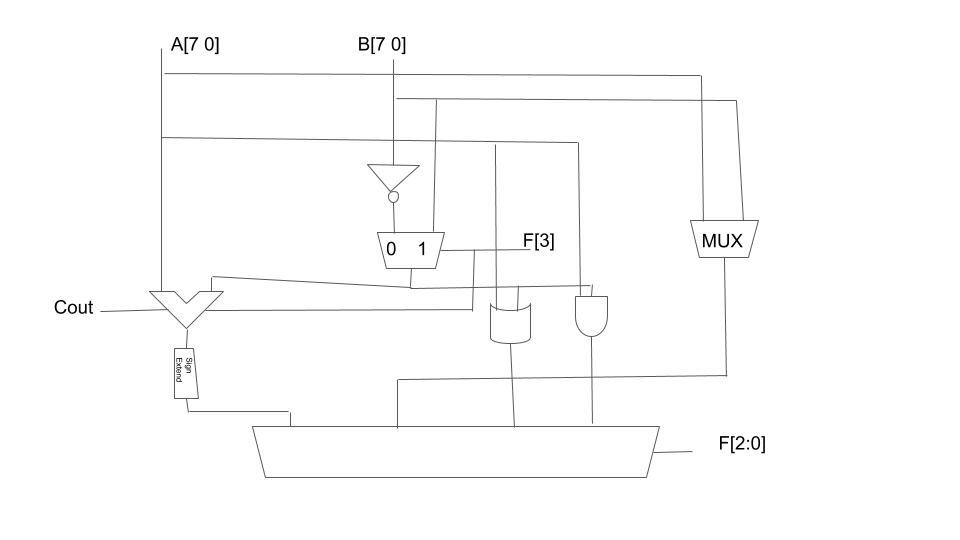
else PC++

**Control Logic Design:**

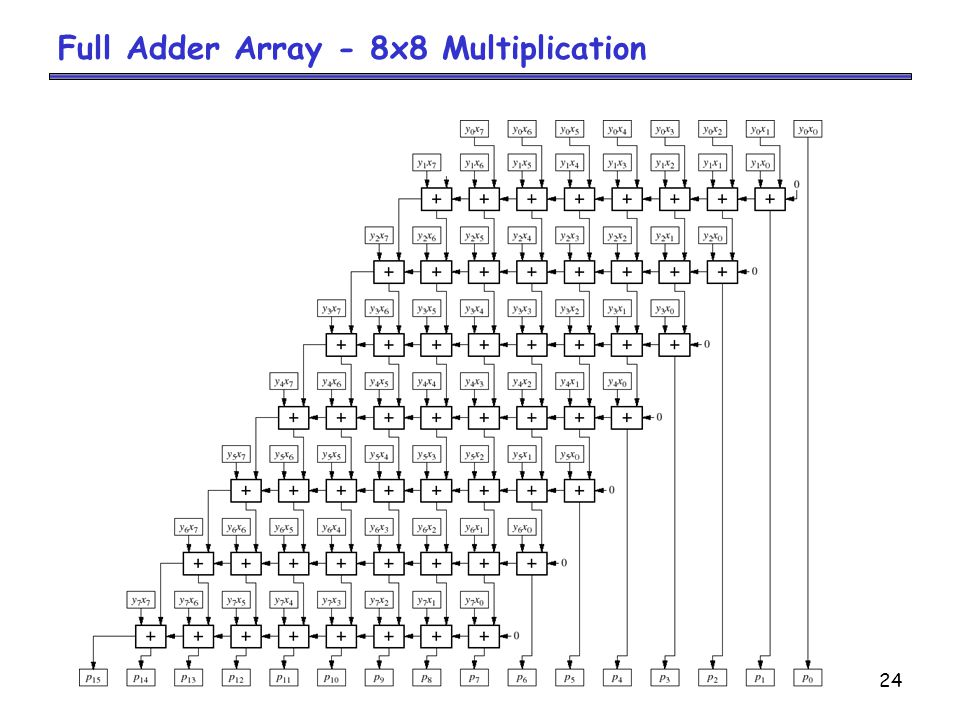
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Opcode** | **RegWrite** | **RegDst** | **ALUSrc** | **Branch** | **MemWrite** | **MemtoReg** | **ALUOp** |
| Add | 0100 | 1 | 1 | 0 | 0 | 0 | 0 | 01 |
| sbnezR2 | 1011 | 1 | 0 | 1 | 1 | 0 | 0 | 10 |
| luiR1 | 0110 | 1 | X | 1 | 0 | 0 | 0 | 01 |
| oriR1 | 0111 | 1 | X | 1 | 0 | 0 | 0 | 10 |
| func | 0000 | 1 | 1 | 0 | 0 | 1 | 0 | 00 |

**ALU Schematic:**

ALU:



Multiply:

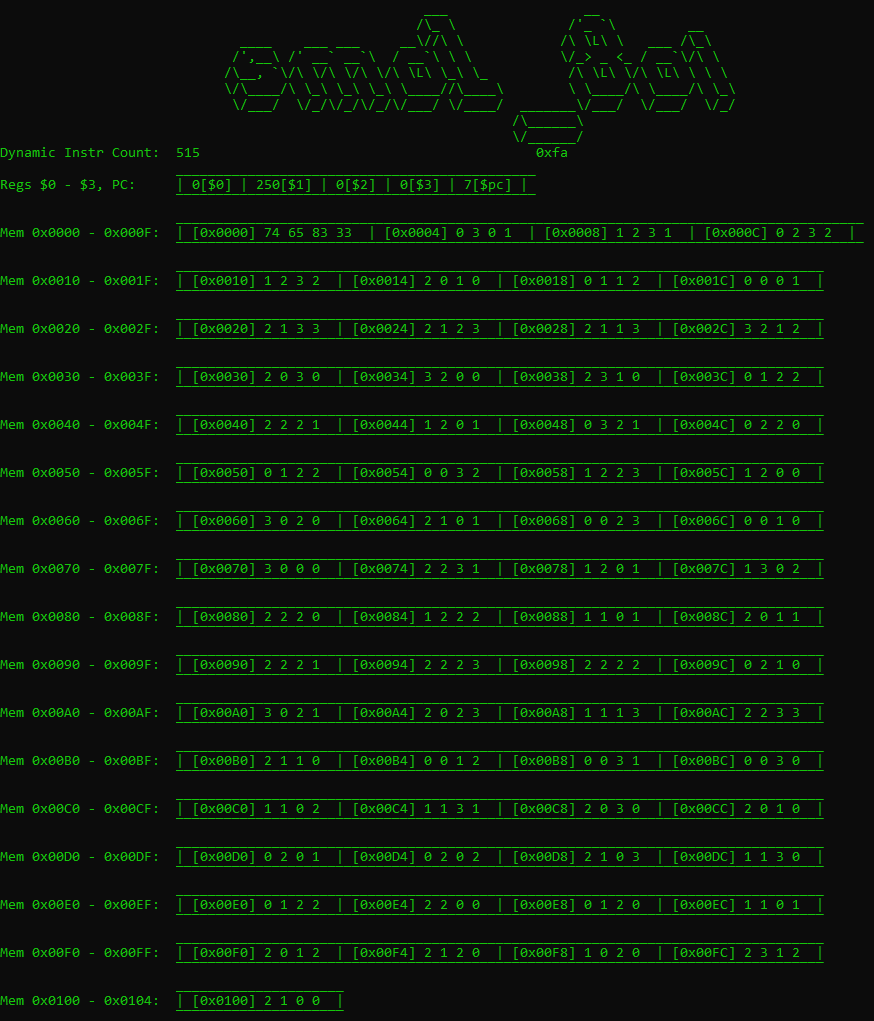
****

The multiplexer in the ALU takes the 8 bits and folds them until the hash reaches a final result of 2 bits.

**PART C**

**Provide results with the following values of B:**

1. 0xFA



Mem [0x0000] = Pattern Matching (0)

Mem [0x0001] = Pattern Matching (1)

Mem [0x0002] = Pattern Matching (2)

Mem [0x0003] = Pattern Matching (3)

ASM Code:

luiR1 15

oriR1 15 # R1 = 0xFF

add $2, $1 # R2 = R1 = 0xFF

luiR1 15

oriR1 10 # R1 = 0xFA

func $3, $2, $1 # hash( 0xFF - 0x01, 0xFA)

sbnezR2 -2 # branch back to func if R2 != 0

MC:

01101111

01111111

01001001

01101111

01111010

00111001

10111110

---------------------------------------------------------------------------------------------------------------------

1. 0x19



Mem [0x0000] = Pattern Matching (0)

Mem [0x0001] = Pattern Matching (1)

Mem [0x0002] = Pattern Matching (2)

Mem [0x0003] = Pattern Matching (3)

ASM Code:

luiR1 15

oriR1 15 # R1 = 0xFF

add $2, $1 # R2 = R1 = 0xFF

luiR1 1

oriR1 9 # R1 = 0x19

func $3, $2, $1 # hash( 0xFF - 0x01, 0xFA)

sbnezR2 -2 # branch back to func if R2 != 0

MC:

01101111

01111111

01001001

01100001

01111001

00111001

10111110

---------------------------------------------------------------------------------------------------------------------

1. 0xE3



Mem [0x0000] = Pattern Matching (0)

Mem [0x0001] = Pattern Matching (1)

Mem [0x0002] = Pattern Matching (2)

Mem [0x0003] = Pattern Matching (3)

ASM Code:

luiR1 15

oriR1 15 # R1 = 0xFF

add $2, $1 # R2 = R1 = 0xFF

luiR1 14

oriR1 3 # R1 = 0xE3

func $3, $2, $1 # hash( 0xFF - 0x01, 0xFA)

sbnezR2 -2 # branch back to func if R2 != 0

MC:

01101111

01111111

01001001

01101110

01110011

00111001

10111110

---------------------------------------------------------------------------------------------------------------------

1. 0x66

****

Mem [0x0000] = Pattern Matching (0)

Mem [0x0001] = Pattern Matching (1)

Mem [0x0002] = Pattern Matching (2)

Mem [0x0003] = Pattern Matching (3)

ASM Code:

luiR1 15

oriR1 15 # R1 = 0xFF

add $2, $1 # R2 = R1 = 0xFF

luiR1 6

oriR1 6 # R1 = 0x66

func $3, $2, $1 # hash( 0xFF - 0x01, 0xFA)

sbnezR2 -2 # branch back to func if R2 != 0

MC:

01101111

01111111

01001001

01100110

01110110

00111001

10111110

---------------------------------------------------------------------------------------------------------------------

Include your Python simulator code here, well commented/annotated so that it’s easy to see how it is corresponding to your ISA design.

# Author(s):

# Supported instrs:

# LUI, ORI, ADDIU, MULTU, MFHI, MFLO, XOR, SRL, ADDU, ANDI, ST, LD, BEZ, SLT, JMP

# register[4] = PC

**def** sim(program):

# Machine Code to Simulation

finished = False # Is the simulation finished?

PC = 0 # Program Counter

register = [0] \* 5 # Let's initialize 32 empty registers

mem = [0] \* 4096 # Let's initialize 0x1000 or 4096 spaces in memory.

#f = open("mc.txt","w+")

DIC = 0 # Dynamic Instr Count

**while** (**not** (finished)):

**if** PC == len(program) - 1:

finished = True

register[4] = PC + 1 # which register will be PC ?

fetch = program[PC]

DIC += 1

# FUNC - Sim

**if** fetch[0:2] == '00': # hashes register[ry] & register[rz], stores result into register[rz] and mem[0x4 + (register[ry] - 1)]

PC += 1

rx = int(fetch[2:4], 2)

ry = int(fetch[4:6], 2)

rz = int(fetch[6:], 2)

hash(int(register[ry]), int(register[rz]), rx, mem, register)

# ADD - Sim

**elif** fetch[0:4] == '0100':

PC += 1

rx = int(fetch[4:6], 2)

ry = int(fetch[6:], 2)

register[rx] = register[rx] + register[ry]

# SBNEZR2 - Sim

**elif** fetch[0:4] == '1011':

PC += 1

imm = -(16 - int(fetch[4:], 2)) **if** fetch[4] == '1' **else** int(fetch[4:], 2) # if $R2 != 0 then branch (PC + IMM) and subtract $R2 by 1, else PC + 1

register[2] = register[2] - 1

# Compare the registers and decide if jumping or not

**if** register[2] != 0:

PC = PC + imm

**if** (imm < 0):

finished = False

# LUIR1 - Sim

**elif** fetch[0:4] == '0110': # sets upper 4 bits of 8 bit number

PC += 1 # to specified imm from ASM

imm = int(fetch[4:], 2)

register[1] = imm << 4

# ORIR1 - Sim

**elif** fetch[0:4] == '0111': # sets lower 4 bits of 8 bit number

PC += 1 # to specified imm from ASM

imm = int(fetch[4:], 2)

register[1] = int(register[1]) | imm

**else**:

# This is not implemented on purpose

PC += 1

**print**('Not implemented')

# Finished simulations, print out stats

# \_\_\_ \_\_

# /\\_ \ /'\_ `\ \_\_

# \_\_\_\_ \_\_\_ \_\_\_ \_\_\_\//\ \ /\ \L\ \ \_\_\_ /\\_\

# /',\_\_\ /' \_\_` \_\_`\ / \_\_`\\ \ \ \/\_> \_ <\_ / \_\_`\/\ \

# /\\_\_, `\/\ \/\ \/\ \/\ \L\ \\\_\ \\_ /\ \L\ \/\ \L\ \ \ \

# \/\\_\_\_\_/\ \\_\ \\_\ \\_\ \\_\_\_\_//\\_\_\_\_\ \ \\_\_\_\_/\ \\_\_\_\_/\ \\_\

# \/\_\_\_/ \/\_/\/\_/\/\_/\/\_\_\_/ \/\_\_\_\_/ \_\_\_\_\_\_\_\/\_\_\_/ \/\_\_\_/ \/\_/

# /\\_\_\_\_\_\_\

# \/\_\_\_\_\_\_/

**print**(" \_\_\_ \_\_ ")

**print**(" /\\_ \ /'\_ `\ \_\_ ")

**print**(" \_\_\_\_ \_\_\_ \_\_\_ \_\_\//\ \ /\ \L\ \ \_\_\_ /\\_\ ")

**print**(" /',\_\_\ /' \_\_` \_\_`\ / \_\_`\\ \ \ \/\_> \_ <\_ / \_\_`\/\ \ ")

**print**(" /\\_\_, `\/\ \/\ \/\ \/\ \L\ \\\_\ \\_ /\ \L\ \/\ \L\ \ \ \ ")

**print**(" \/\\_\_\_\_/\ \\_\ \\_\ \\_\ \\_\_\_\_//\\_\_\_\_\ \ \\_\_\_\_/\ \\_\_\_\_/\ \\_\ ")

**print**(" \/\_\_\_/ \/\_/\/\_/\/\_/\/\_\_\_/ \/\_\_\_\_/ \_\_\_\_\_\_\_\/\_\_\_/ \/\_\_\_/ \/\_/ ")

**print**(" /\\_\_\_\_\_\_\ ")

**print**(" \/\_\_\_\_\_\_/ ")

**print**('Dynamic Instr Count: ', DIC, ' ', hex(register[1]))

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Regs $0 - $3, PC: | {}[$0] | {}[$1] | {}[$2] | {}[$3] | {}[$pc] |'.format(register[0], register[1], register[2], register[3], register[4]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0000 - 0x000F: | [0x0000] {} {} {} {} | [0x0004] {} {} {} {} | [0x0008] {} {} {} {} | [0x000C] {} {} {} {} |'.format(mem[0], mem[1], mem[2], mem[3], mem[4], mem[5], mem[6], mem[7], mem[8], mem[9], mem[10], mem[11], mem[12], mem[13], mem[14], mem[15]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0010 - 0x001F: | [0x0010] {} {} {} {} | [0x0014] {} {} {} {} | [0x0018] {} {} {} {} | [0x001C] {} {} {} {} |'.format(mem[16], mem[17], mem[18], mem[19], mem[20], mem[21], mem[22], mem[23], mem[24], mem[25], mem[26], mem[27], mem[28], mem[29], mem[30], mem[31]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0020 - 0x002F: | [0x0020] {} {} {} {} | [0x0024] {} {} {} {} | [0x0028] {} {} {} {} | [0x002C] {} {} {} {} |'.format(mem[32], mem[33], mem[34], mem[35], mem[36], mem[37], mem[38], mem[39], mem[40], mem[41], mem[42], mem[43], mem[44], mem[45], mem[46], mem[47]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0030 - 0x003F: | [0x0030] {} {} {} {} | [0x0034] {} {} {} {} | [0x0038] {} {} {} {} | [0x003C] {} {} {} {} |'.format(mem[48], mem[49], mem[50], mem[51], mem[52], mem[53], mem[54], mem[55], mem[56], mem[57], mem[58], mem[59], mem[60], mem[61], mem[62], mem[63]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0040 - 0x004F: | [0x0040] {} {} {} {} | [0x0044] {} {} {} {} | [0x0048] {} {} {} {} | [0x004C] {} {} {} {} |'.format(mem[64], mem[65], mem[66], mem[67], mem[68], mem[69], mem[70], mem[71], mem[72], mem[73], mem[74], mem[75], mem[76], mem[77], mem[78], mem[79]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0050 - 0x005F: | [0x0050] {} {} {} {} | [0x0054] {} {} {} {} | [0x0058] {} {} {} {} | [0x005C] {} {} {} {} |'.format(mem[80], mem[81], mem[82], mem[83], mem[84], mem[85], mem[86], mem[87], mem[88], mem[89], mem[90], mem[91], mem[92], mem[93], mem[94], mem[95]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0060 - 0x006F: | [0x0060] {} {} {} {} | [0x0064] {} {} {} {} | [0x0068] {} {} {} {} | [0x006C] {} {} {} {} |'.format(mem[96], mem[97], mem[98], mem[99], mem[100], mem[101], mem[102], mem[103], mem[104], mem[105], mem[106], mem[107], mem[108], mem[109], mem[110], mem[111]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0070 - 0x007F: | [0x0070] {} {} {} {} | [0x0074] {} {} {} {} | [0x0078] {} {} {} {} | [0x007C] {} {} {} {} |'.format(mem[112], mem[113], mem[114], mem[115], mem[116], mem[117], mem[118], mem[119], mem[120], mem[121], mem[122], mem[123], mem[124], mem[125], mem[126], mem[127]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0080 - 0x008F: | [0x0080] {} {} {} {} | [0x0084] {} {} {} {} | [0x0088] {} {} {} {} | [0x008C] {} {} {} {} |'.format(mem[128], mem[129], mem[130], mem[131], mem[132], mem[133], mem[134], mem[135], mem[136], mem[137], mem[138], mem[139], mem[140], mem[141], mem[142], mem[143]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0090 - 0x009F: | [0x0090] {} {} {} {} | [0x0094] {} {} {} {} | [0x0098] {} {} {} {} | [0x009C] {} {} {} {} |'.format(mem[144], mem[145], mem[146], mem[147], mem[148], mem[149], mem[150], mem[151], mem[152], mem[153], mem[154], mem[155], mem[156], mem[157], mem[158], mem[159]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x00A0 - 0x00AF: | [0x00A0] {} {} {} {} | [0x00A4] {} {} {} {} | [0x00A8] {} {} {} {} | [0x00AC] {} {} {} {} |'.format(mem[160], mem[161], mem[162], mem[163], mem[164], mem[165], mem[166], mem[167], mem[168], mem[169], mem[170], mem[171], mem[172], mem[173], mem[174], mem[175]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x00B0 - 0x00BF: | [0x00B0] {} {} {} {} | [0x00B4] {} {} {} {} | [0x00B8] {} {} {} {} | [0x00BC] {} {} {} {} |'.format(mem[176], mem[177], mem[178], mem[179], mem[180], mem[181], mem[182], mem[183], mem[184], mem[185], mem[186], mem[187], mem[188], mem[189], mem[190], mem[191]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x00C0 - 0x00CF: | [0x00C0] {} {} {} {} | [0x00C4] {} {} {} {} | [0x00C8] {} {} {} {} | [0x00CC] {} {} {} {} |'.format(mem[192], mem[193], mem[194], mem[195], mem[196], mem[197], mem[198], mem[199], mem[200], mem[201], mem[202], mem[203], mem[204], mem[205], mem[206], mem[207]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x00D0 - 0x00DF: | [0x00D0] {} {} {} {} | [0x00D4] {} {} {} {} | [0x00D8] {} {} {} {} | [0x00DC] {} {} {} {} |'.format(mem[208], mem[209], mem[210], mem[211], mem[212], mem[213], mem[214], mem[215], mem[216], mem[217], mem[218], mem[219], mem[220], mem[221], mem[222], mem[223]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x00E0 - 0x00EF: | [0x00E0] {} {} {} {} | [0x00E4] {} {} {} {} | [0x00E8] {} {} {} {} | [0x00EC] {} {} {} {} |'.format(mem[224], mem[225], mem[226], mem[227], mem[228], mem[229], mem[230], mem[231], mem[232], mem[233], mem[234], mem[235], mem[236], mem[237], mem[238], mem[239]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x00F0 - 0x00FF: | [0x00F0] {} {} {} {} | [0x00F4] {} {} {} {} | [0x00F8] {} {} {} {} | [0x00FC] {} {} {} {} |'.format(mem[240], mem[241], mem[242], mem[243], mem[244], mem[245], mem[246], mem[247], mem[248], mem[249], mem[250], mem[251], mem[252], mem[253], mem[254], mem[255]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**(' \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_')

**print**('Mem 0x0100 - 0x0104: | [0x0100] {} {} {} {} |'.format(mem[256], mem[257], mem[258], mem[259]))

**print**(' ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾')

**print**('~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~')

**print**('')

input()

**def** hash(A, B, rx, mem, register):

# MultFold1

C = A \* B

C\_hi = C >> 7

C\_hi = C\_hi >> 1 # for some reason SLR using even numers didn't work

C\_low = C & 0x00FF

C = C\_hi ^ C\_low

# MultFold2

C = B\*C

C\_hi = C >> 7

C\_hi = C\_hi >> 1

C\_low = C & 0x00FF

C = C\_hi^C\_low

# MultFold3

C = B\*C

C\_hi = C >> 7

C\_hi = C\_hi >> 1

C\_low = C & 0x00FF

C = C\_hi^C\_low

# MultFold4

C = B\*C

C\_hi = C >> 7

C\_hi = C\_hi >> 1

C\_low = C & 0x00FF

C = C\_hi^C\_low

# MultFold5

C = B\*C

C\_hi = C >> 7

C\_hi = C\_hi >> 1

C\_low = C & 0x00FF

C = C\_hi^C\_low

# Fold 8 -> 4 bits

C\_hi = C >> 3

C\_hi = C\_hi >> 1

C\_low = C & 0x0F

C = C\_hi^C\_low

# Fold 4 -> 2 bits

C\_hi = C >> 1

C\_hi = C\_hi >> 1

C\_low = C & 0x3

C = C\_hi^C\_low

# Pattern matching and storing to corresponding register

**if**(C == 0):

mem[0] = mem[0] + 1

**elif**(C == 1):

mem[1] = mem[1] + 1

**elif**(C == 2):

mem[2] = mem[2] + 1

**elif**(C == 3):

mem[3] = mem[3] + 1

mem[0x4 + (A - 1)] = C

register[rx] = C

# Remember where each of the jump label is, and the target location

**def** saveJumpLabel(asm,labelIndex, labelName):

lineCount = 0

countWithoutLabels = 0

**for** line **in** asm:

line = line.replace(" ","")

**if**(line.count(":")):

labelName.append(line[0:line.index(":")]) # append the label name

labelIndex.append(countWithoutLabels) # append the label's index

asm[lineCount] = line[line.index(":")+1:]

countWithoutLabels -= 1

lineCount += 1

countWithoutLabels += 1

**for** item **in** range(asm.count('\n')): # Remove all empty lines '\n'

asm.remove('\n')

**def** main():

labelIndex = []

labelName = []

f = open("mc.txt", "w+")

h = open("test0xE3.asm", "r")

asm = h.readlines()

currentline = 0

**for** item **in** range(asm.count('\n')): # Remove all empty lines '\n'

asm.remove('\n')

saveJumpLabel(asm, labelIndex, labelName) # Save all jump's destinations

**for** line **in** asm:

line = line.replace("\n", "") # Removes extra chars

line = line.replace("$", "")

line = line.replace(" ", "")

line = line.replace("zero", "0") # assembly can also use both $zero and $0

#

# This section of the program converts

# the ASM file into machine code

# which is the simulated above

# with the " XXXXX - Sim" statements

#

# = = = = ADD = = = = = =

**if** (line[0:3] == "add"):

line = line.replace("add", "")

line = line.split(",")

rx = format(int(line[0]), '02b') # make element 0 in the set, 'line' an int of 5 bits. (rt)

ry = format(int(line[1]), '02b')

f.write(str('0100') + str(rx) + str(ry) + '\n')

currentline += 1

# = = = = FUNC = = = = = =

**elif** (line[0:4] == "func"):

line = line.replace("func", "")

line = line.split(",")

rx = format(int(line[0]), '02b') # make element 0 in the set, 'line' an int of 5 bits. (rt)

ry = format(int(line[1]), '02b')

rz = format(int(line[2]), '02b')

f.write(str('00') + str(rx) + str(ry) + str(rz) + '\n')

currentline += 1

# = = = = SBNEZR2 = = = = = =

**elif** (line[0:7] == "sbnezR2"):

line = line.replace("sbnezR2", "")

line = line.split(",")

imm = format(int(line[0]), '04b') **if** (int(line[0]) >= 0) **else** format(16 + int(line[0]), '04b')

f.write(str('1011') + str(imm) + '\n')

currentline += 1

# = = = = LUIR1 = = = = = =

**elif** (line[0:5] == "luiR1"):

line = line.replace("luiR1", "")

line = line.split(",")

imm = format(int(line[0]), '04b')

f.write(str('0110') + str(imm) + '\n')

currentline += 1

# = = = = ORIR1 = = = = = =

**elif** (line[0:5] == "oriR1"):

line = line.replace("oriR1", "")

line = line.split(",")

imm = format(int(line[0]), '04b')

f.write(str('0111') + str(imm) + '\n')

currentline += 1

f.close()

#--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------#

# file opener and reader (Machine Code)

file = open('mc.txt')

program = []

**for** line **in** file:

**if** line.count('#'):

line = list(line)

line[line.index('#'):-1] = ''

line = ''.join(line)

**if** line[0] == '\n':

**continue**

line = line.replace('\n', '')

instr = line[:]

program.append(instr) # since PC increment every cycle

# We SHALL start the simulation!

sim(program)

**if** \_\_name\_\_ == '\_\_main\_\_':

main()

#fin