Group 12: Claire Chappee and Zhongy Chen
ECE 366 Project 3

Activity Log

Part I: Table of group activity

Meeting 1 {
Time/Location: Friday, October 25th 4:00-5:00pm in the CS lounge
Activity: We need to plan out/distribute work
Achieved/To-Do: We assigned our work to do for the project. Brainstorming/planning, also created a GitHub repo for the project was achieved at the meeting.
Attended: Zhongyi & Claire
}
Meeting 2 {
Time/Location: Friday, November 1st 4:00-5:00pm in the CS lounge
Activity: We need to check in on progress and brainstorm hardware implementation drawings.
Achieved/To-Do: We looked at the GitHub to see our individual progress and adjusted our timeline of completed milestones.
Attended: Zhongyi & Claire
}

Part II: Individual activity list

Claire:

I was responsible for the python simulator and working on correctly interpreting the machine code to implement our new ISA. I also helped on various other little things like the hardware drawings and the report.

Zhongyi:

- Decided what instructions to use, their encoding, how they worked, etc.
- Designed CPU datapath, control logic (truth table), and ALU
- Tweaked python simulator for minor bugs and formatted memory output

Part III: Group work reflection

a) How does this group differ from the previous one?

The group was smaller this time since Chris told us that she was dropping the class right before our first meeting together to distribute the work.

b) List out one new or important thing that you have learned from this group.

We really learned how having a less structured meeting agenda and not strict milestone deadlines for ourselves allowed us each to work at our own pace with the project and actually was able to complete it faster than expected.

c) If you were to do this project all by yourself, how different would it be? Why?

Doing this project as a “one-man-show” would have been too much for the project deadline. We would have felt overwhelmed to get so much done in a short time frame. By having a partner it allowed us each to take some weight off of our shoulders and realistically complete the project.

Project Content

Part A) ISA intro, Q&A

1) Introduction & Instruction list. Name of the architecture, overall philosophy, specific goals strived for and achieved, all the instructions, their formats, opcodes, and an example.

The name of our architecture is Minimal-Hash-ISA. The overall philosophy was to keep it simple and eliminate branching if possible. We did achieve an architecture without branching. No branching meant an easier hardware implementation and also reduces the dynamic instruction count.

Instruction	Functionality	Opcode	Encoding Format	MC Example	Asm Example
lui Rx, imm	Rx = (imm << 4); PC++; imm: [0, 15]	00	00 xx iiii	00001111	lui $0, 0xF
addi Rx, imm	Rx = Rx + imm; PC++; imm: [0, 15]	01	01 xx iiii	01001010	addi $0, 0xA
hash Rx, Ry, Rz	Rx = H(Ry, Rz); PC++	10	10 xx yy zz	10100100	hash $2, $1, $0
ldinc Rx, (Ry)	Rx = Mem[Ry] + 1; PC++	1100	1100 xx yy	11001110	ldinc $3, ($2)
st Rx, (Ry)	Mem[Ry] = Rx; PC++;	1110	1110 xx yy	11101110	st $3, ($2)
sto3inc Rx, (Ry)	Mem[Ry + 3] = Rx; Ry++; PC++;	1111	1111 xx yy	11111001	sto3inc $2, ($1)

2) Register design. How many registers are supported? Is there anything special about the registers?

We supported 5 registers: $0, $1, $2, $3, and pc. There is nothing special about these registers.

3) Branch design. What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported?

The special thing about the way that we did this project is that we do not support branches at all!

4) Data memory addressing modes. What kind of instructions are used to access data memory? What is the range of addresses that can be accessed with your design?

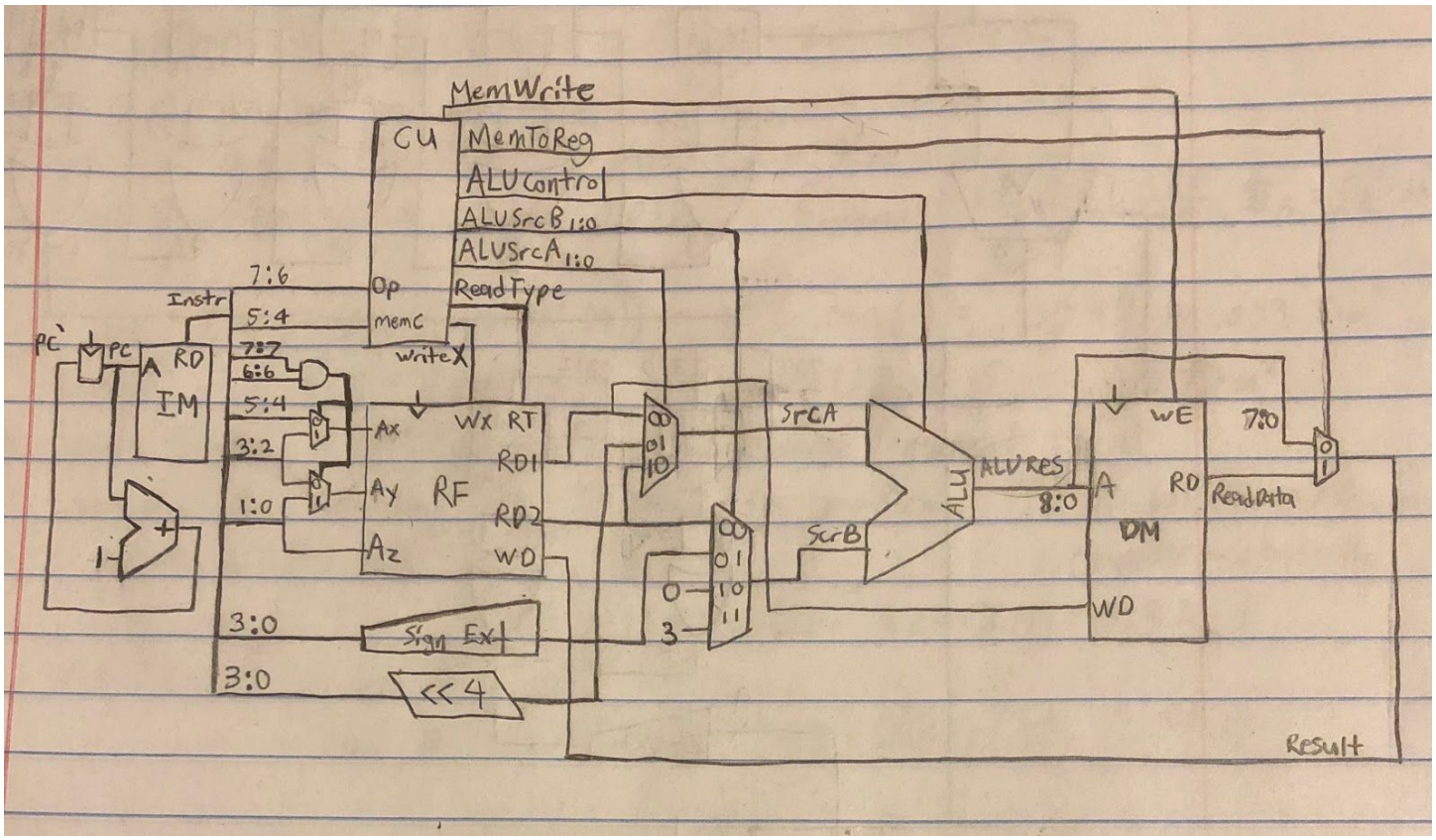
ldinc, st, and sto3inc are used to access/modify data memory. With our design, ldinc and st can reach address 0x0 - 0xFF and sto3inc can reach addresses 0x3 - 0x102.

5) What are the most significant features of your ISA (with regard to the Hash program, hardware implementation, ease of programming, etc)? What have you done towards the goals of low DIC and HW simplification? What are the main limitations? What are the main compromises that you have done to make things work, rather than perfecting everything?

The most significant thing about our ISA is it does not use branching and the ALU is designed to fully support this project's hash functionality in one cycle. No branching makes programming the simulator easier since there will be no possibility for the code to get stuck in an endless loop. Eliminating branching reduces DIC by at least the number of values that needs to be hashed. Some instructions automatically increment registers/values, which reduces the need to use extra instructions for incrementing. Also, there are only a few instructions either doing unsigned add or hash, minimizing what is actually needed in hardware. Main limitation with no branching functionality is no loop functionality, so we have to copy and paste a lot of machine code. The main compromise we made is to not use immediate value in instruction when calculating memory address. This allowed for more instruction types (ldinc, st, sto3inc) that can do more specific things.

Part B) Hardware implementations

1) CPU Datapath design. A schematic including your register file, ALU, PC logic, and memory components. Clearly mark out all the signal lines, their names, and how many bits for each.

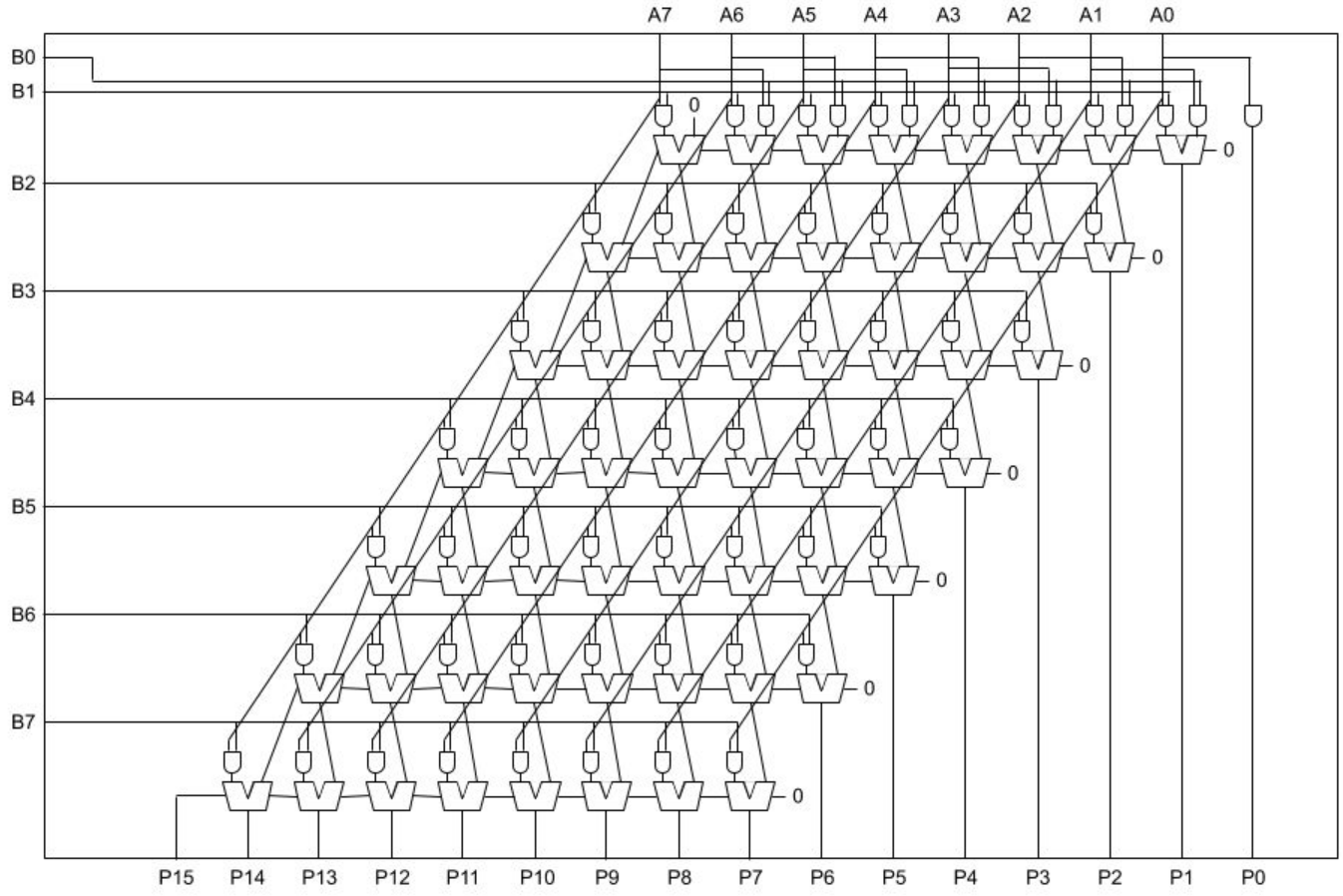


2) Control logic design. Decoder truth-table indicating how each control signal (one per column) is specified (0, 1, or X) from each instruction (one per row). If you have special instructions or register design, explain the control signals briefly.

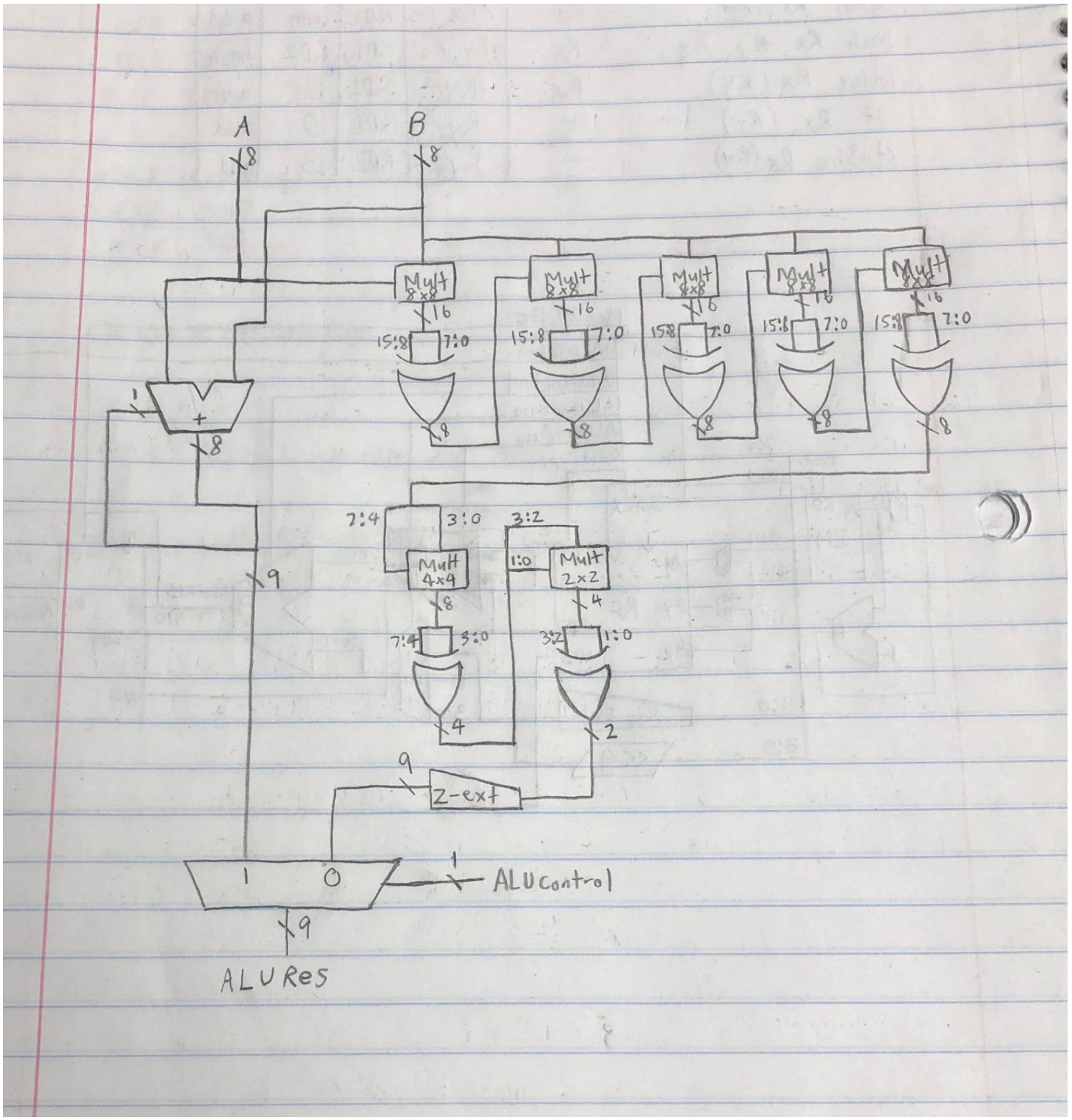
Instruction	MemWrite	MemToReg	ALUControl	ALUSrcB	ALUSrcA	ReadType	WriteX
lui	0	0	1	10	01	X	1
addi	0	0	1	01	00	0	1
hash	0	0	0	00	00	1	1
ldinc	0	1	1	10	10	0	1
st	1	X	1	10	10	0	0
sto3inc	1	X	1	11	10	0	0

Control Signal	Description
MemWrite	Should memory be modified? 0 - No, 1 - Yes
MemToReg	What should be written to register? 0 - ALURes, 1 - Value from memory
ALUControl	What operation? 0 - hash, 1 - add
ALUSrcB	Select value for srcB
ALUSrcA	Select value for srcA
ReadType	Determine values for RD1/RD2. 0 - RD1 = reg[Ax], RD2 = reg[Ay] 1 - RD1 = reg[Ay], RD2 = reg[Az]
WriteX	Should value be written to Rx? 0 - No, 1 - Yes

3) ALU schematic. A hierarchical sketch of your Arithmetic Logic Unit which implements whatever computation that your ISA instructions use.



This is a multiplication block that supports multiplying two 8 bit numbers to get a 16 bit result. This is represented by “Mult 8x8” in the ALU drawing. There are also “Mult 4x4” and “Mult 2x2” but they are just smaller versions of “Mult 8x8”.


A
B
8
8
Mult 8x8
16
15:8
7:0
8
1
+
9
7:4
3:0
3:2
1:0
Mult 4x4
Mult 2x2
4
2
Z-ext
1
ALUControl
ALURes

Part C) Software package

In an attempt to reduce the number of pages for the report, this little section will describe how assembly and machine code are written. Since our code does not use branching, there are 255 values to hash, and each hash computation is 4 instructions, this will result in over 8000 lines of code being copied onto this report, which is really unreasonable for a reader to look at.

This is how we are presenting our assembly and machine code:

hash $2, $1, $0 ldinc $3, ($2) st $3, ($2) sto3inc $2, ($1)	x 3	Is equivalent to…	hash $2, $1, $0 ldinc $3, ($2) st $3, ($2) sto3inc $2, ($1) hash $2, $1, $0 ldinc $3, ($2) st $3, ($2) sto3inc $2, ($1) hash $2, $1, $0 ldinc $3, ($2) st $3, ($2) sto3inc $2, ($1)
10100100 11001110 11101110 11111001	x 3	Is equivalent to...	10100100 11001110 11101110 11111001 10100100 11001110 11101110 11111001 10100100 11001110 11101110 11111001

i) 0xFA

1. Assembly code (should be easy to read) of your program. Make sure your assembly format is either obvious or well described, and that the code is well commented.

Line Numbers	Code	Times occuring	Description

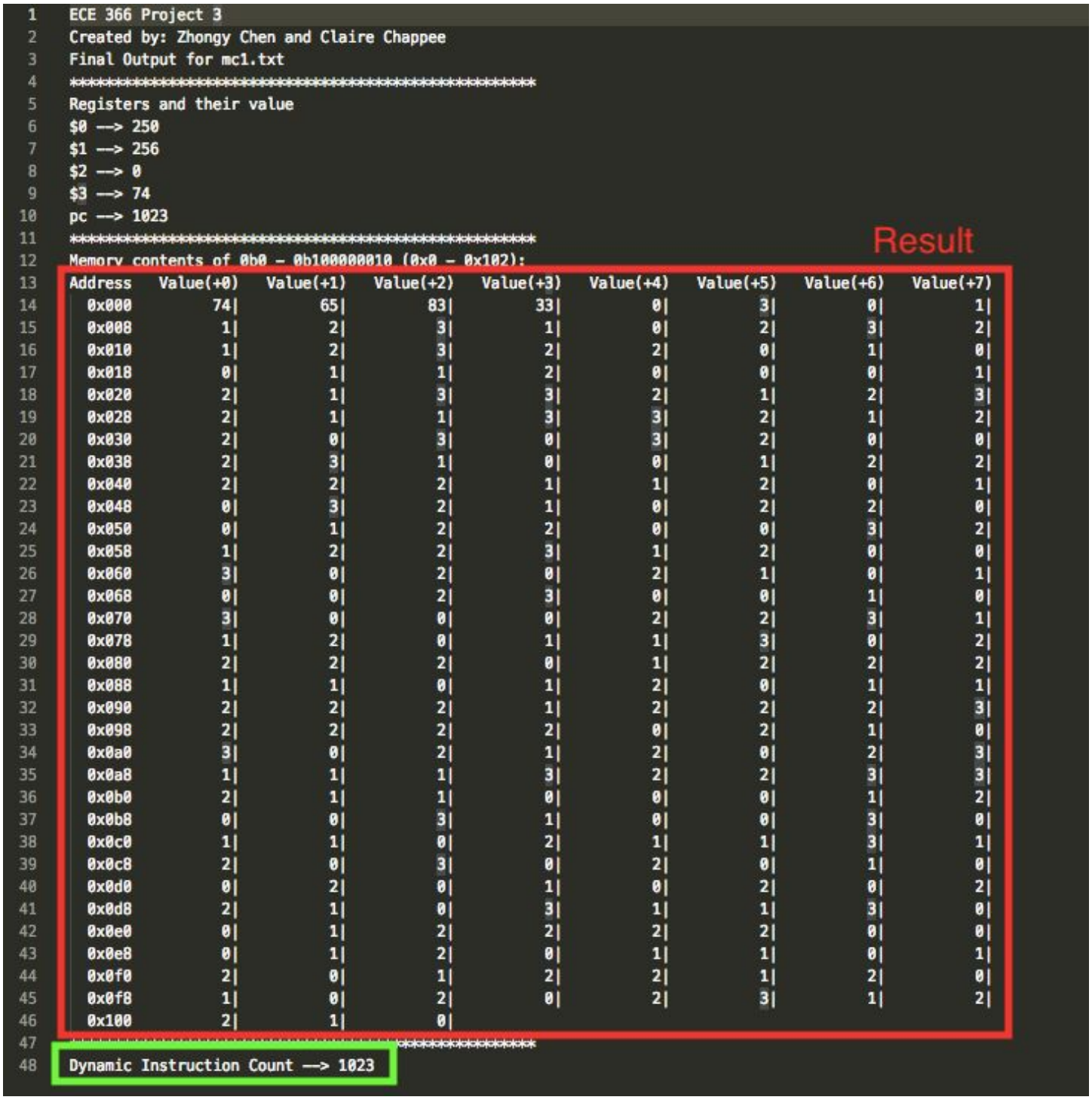
1 - 3	lui $0, 0xF
	addi $0, 0xA
	addi $1, 0x1
		x 1	Set B value 0xFA into register $0

4 - 1023	hash $2, $1, $0 ldinc $3, ($2) st $3, ($2) sto3inc $2, ($1)	x 255	Compute hash, Increment the times this pattern appeared, Store incremented value back to memory, Store hash value and increment iterator

2. Machine code (either in binary or hex) of your program. This should be the input to your python simulator.

Line Numbers	Code	Times occuring	Description
1 - 3	00001111 01001010 01010001	x 1	Set B value 0xFA into register $0
4 - 1023	10100100 11001110 11101110 11111001	x 255	Compute hash, Increment the times this pattern appeared, Store incremented value back to memory, Store hash value and increment iterator

3. Screenshots of your Python simulator's output for your program. Highlight the final results (pattern counts, all C values in memory) and Dynamic Instruction Count. You should provide sufficient screenshots to convince us that your ISA + Python package work correctly for any B values of the Hash program.



```
ECE 366 Project 3
Created by: Zhongy Chen and Claire Chappee
Final Output for mc1.txt
**************************************************
Registers and their value
$0 --> 250
$1 --> 256
$2 --> 0
$3 --> 74
pc --> 1023
**************************************************
Memory contents of 0b0 - 0b100000010 (0x0 - 0x102):
```

Result

Address	Value(+0)	Value(+1)	Value(+2)	Value(+3)	Value(+4)	Value(+5)	Value(+6)	Value(+7)
0x000	74	65	83	33	0	3	0	1
0x008	1	2	3	1	0	2	3	2
0x010	1	2	3	2	2	0	1	0
0x018	0	1	1	2	0	0	0	1
0x020	2	1	3	3	2	1	2	3
0x028	2	1	1	3	3	2	1	2
0x030	2	0	3	0	3	2	0	0
0x038	2	3	1	0	0	1	2	2
0x040	2	2	2	1	1	2	0	1
0x048	0	3	2	1	0	2	2	0
0x050	0	1	2	2	0	0	3	2
0x058	1	2	2	3	1	2	0	0
0x060	3	0	2	0	2	1	0	1
0x068	0	0	2	3	0	0	1	0
0x070	3	0	0	0	2	2	3	1
0x078	1	2	0	1	1	3	0	2
0x080	2	2	2	0	1	2	2	2
0x088	1	1	0	1	2	0	1	1
0x090	2	2	2	1	2	2	2	3
0x098	2	2	2	2	0	2	1	0
0x0a0	3	0	2	1	2	0	2	3
0x0a8	1	1	1	3	2	2	3	3
0x0b0	2	1	1	0	0	0	1	2
0x0b8	0	0	3	1	0	0	3	0
0x0c0	1	1	0	2	1	1	3	1
0x0c8	2	0	3	0	2	0	1	0
0x0d0	0	2	0	1	0	2	0	2
0x0d8	2	1	0	3	1	1	3	0
0x0e0	0	1	2	2	2	2	0	0
0x0e8	0	1	2	0	1	1	0	1
0x0f0	2	0	1	2	2	1	2	0
0x0f8	1	0	2	0	2	3	1	2
0x100	2	1	0					

```
**************************************************
Dynamic Instruction Count --> 1023
```

ii) 0x19

1. Assembly code (should be easy to read) of your program. Make sure your assembly format is either obvious or well described, and that the code is well commented.

Line Numbers	Code	Times occuring	Description

1 - 3	lui $0, 0x1
	addi $0, 0x9
	addi $1, 0x1
		x 1	Set B value 0x19 into register $0

4 - 1023	hash $2, $1, $0 ldinc $3, ($2) st $3, ($2) sto3inc $2, ($1)	x 255	Compute hash, Increment the times this pattern appeared, Store incremented value back to memory, Store hash value and increment iterator

2. Machine code (either in binary or hex) of your program. This should be the input to your python simulator.

Line Numbers	Code	Times occuring	Description
1 - 3	00000001 01001001 01010001	x 1	Set B value 0x19 into register $0
4 - 1023	10100100 11001110 11101110 11111001	x 255	Compute hash, Increment the times this pattern appeared, Store incremented value back to memory, Store hash value and increment iterator

3. Screenshots of your Python simulator's output for your program. Highlight the final results (pattern counts, all C values in memory) and Dynamic Instruction Count. You should provide sufficient screenshots to convince us that your ISA + Python package work correctly for any B values of the Hash program.

```
ECE 366 Project 3
Created by: Zhongy Chen and Claire Chappee
Final Output for mc2.txt
**************************************************
Registers and their values
$0 --> 25
$1 --> 256
$2 --> 0
$3 --> 146
pc --> 1023
**************************************************
Memory contents of 0b0 - 0b100000010 (0x0 - 0x102):
```

Result

Address	Value(+0)	Value(+1)	Value(+2)	Value(+3)	Value(+4)	Value(+5)	Value(+6)	Value(+7)
0x000	146	52	46	11	2	0	3	0
0x008	0	3	0	0	2	0	1	0
0x010	1	0	1	0	0	1	0	0
0x018	2	0	0	0	1	0	1	0
0x020	1	0	2	0	2	0	1	1
0x028	0	0	0	0	2	2	0	1
0x030	1	1	1	2	2	1	0	3
0x038	1	1	0	0	0	0	0	1
0x040	0	3	0	0	0	0	3	1
0x048	0	2	0	2	0	0	0	0
0x050	0	0	3	1	0	0	0	0
0x058	1	2	2	1	0	0	0	1
0x060	0	0	0	1	0	1	1	2
0x068	0	0	2	0	3	0	0	0
0x070	0	2	2	0	0	2	2	0
0x078	0	3	0	0	0	2	2	0
0x080	2	0	0	0	1	0	0	0
0x088	0	0	0	1	2	2	0	1
0x090	0	1	0	2	0	0	0	1
0x098	2	0	2	0	0	2	0	0
0x0a0	0	1	0	0	0	2	0	1
0x0a8	2	0	0	0	0	0	0	0
0x0b0	0	0	2	0	0	1	1	2
0x0b8	0	1	1	0	1	2	0	0
0x0c0	2	0	0	2	2	1	0	0
0x0c8	0	1	0	1	2	0	0	0
0x0d0	3	0	0	1	0	3	0	2
0x0d8	1	0	0	2	0	0	1	1
0x0e0	0	0	2	0	0	2	0	1
0x0e8	3	0	0	0	1	0	1	0
0x0f0	0	2	2	2	0	0	1	1
0x0f8	0	2	2	1	0	0	1	0
0x100	1	0	0					

```
**************************************************
Dynamic Instruction Count --> 1023
```

iii) 0xE3

1. Assembly code (should be easy to read) of your program. Make sure your assembly format is either obvious or well described, and that the code is well commented.

Line Numbers	Code	Times occuring	Description

1 - 3	lui $0, 0xE
	addi $0, 0x3
	addi $1, 0x1
		x 1	Set B value 0xE3 into register $0

4 - 1023	hash $2, $1, $0 ldinc $3, ($2) st $3, ($2) sto3inc $2, ($1)	x 255	Compute hash, Increment the times this pattern appeared, Store incremented value back to memory, Store hash value and increment iterator

2. Machine code (either in binary or hex) of your program. This should be the input to your python simulator.

Line Numbers	Code	Times occuring	Description
1 - 3	00001110 01000011 01010001	x 1	Set B value 0xE3 into register $0
4 - 1023	10100100 11001110 11101110 11111001	x 255	Compute hash, Increment the times this pattern appeared, Store incremented value back to memory, Store hash value and increment iterator

3. Screenshots of your Python simulator's output for your program. Highlight the final results (pattern counts, all C values in memory) and Dynamic Instruction Count. You should provide sufficient screenshots to convince us that your ISA + Python package work correctly for any B values of the Hash program.

```
ECE 366 Project 3
Created by: Zhongy Chen and Claire Chappee
Final Output for mc3.txt
**************************************************
Registers and their values
$0 --> 227
$1 --> 256
$2 --> 0
$3 --> 74
pc --> 1023
**************************************************
Memory contents of 0b0 - 0b100000010 (0x0 - 0x102):
```

Result

Address	Value(+0)	Value(+1)	Value(+2)	Value(+3)	Value(+4)	Value(+5)	Value(+6)	Value(+7)
0x000	74	68	70	43	0	2	1	3
0x008	1	3	3	0	0	2	2	0
0x010	1	0	2	0	0	1	3	1
0x018	3	2	1	0	3	3	2	1
0x020	0	1	0	2	0	2	1	2
0x028	2	2	3	0	0	0	1	0
0x030	0	2	0	2	2	3	3	3
0x038	0	3	0	2	1	0	0	0
0x040	1	1	1	1	2	3	1	1
0x048	2	2	2	1	1	3	3	1
0x050	0	0	3	3	1	2	0	0
0x058	3	0	1	2	2	0	0	1
0x060	0	1	2	2	3	1	0	1
0x068	2	2	0	2	2	1	2	2
0x070	2	1	0	3	3	2	1	2
0x078	0	3	0	2	1	3	2	2
0x080	1	0	2	0	2	3	0	1
0x088	2	2	0	0	2	1	2	2
0x090	3	1	2	1	1	1	1	2
0x098	2	1	0	2	2	1	1	0
0x0a0	3	0	1	2	1	2	3	1
0x0a8	3	0	0	0	2	2	0	0
0x0b0	2	0	3	1	0	3	1	0
0x0b8	3	3	1	1	3	1	0	2
0x0c0	0	1	1	1	1	0	3	1
0x0c8	0	3	1	1	1	1	1	3
0x0d0	0	0	2	2	1	1	0	0
0x0d8	1	0	0	2	1	3	2	2
0x0e0	2	0	1	0	2	0	1	2
0x0e8	2	0	3	2	3	2	2	0
0x0f0	0	0	0	3	0	2	0	0
0x0f8	2	1	0	3	2	2	3	1
0x100	3	1	0					

```
**************************************************
Dynamic Instruction Count --> 1023
```

iv) 0x66

1. Assembly code (should be easy to read) of your program. Make sure your assembly format is either obvious or well described, and that the code is well commented.

Line Numbers	Code	Times occuring	Description

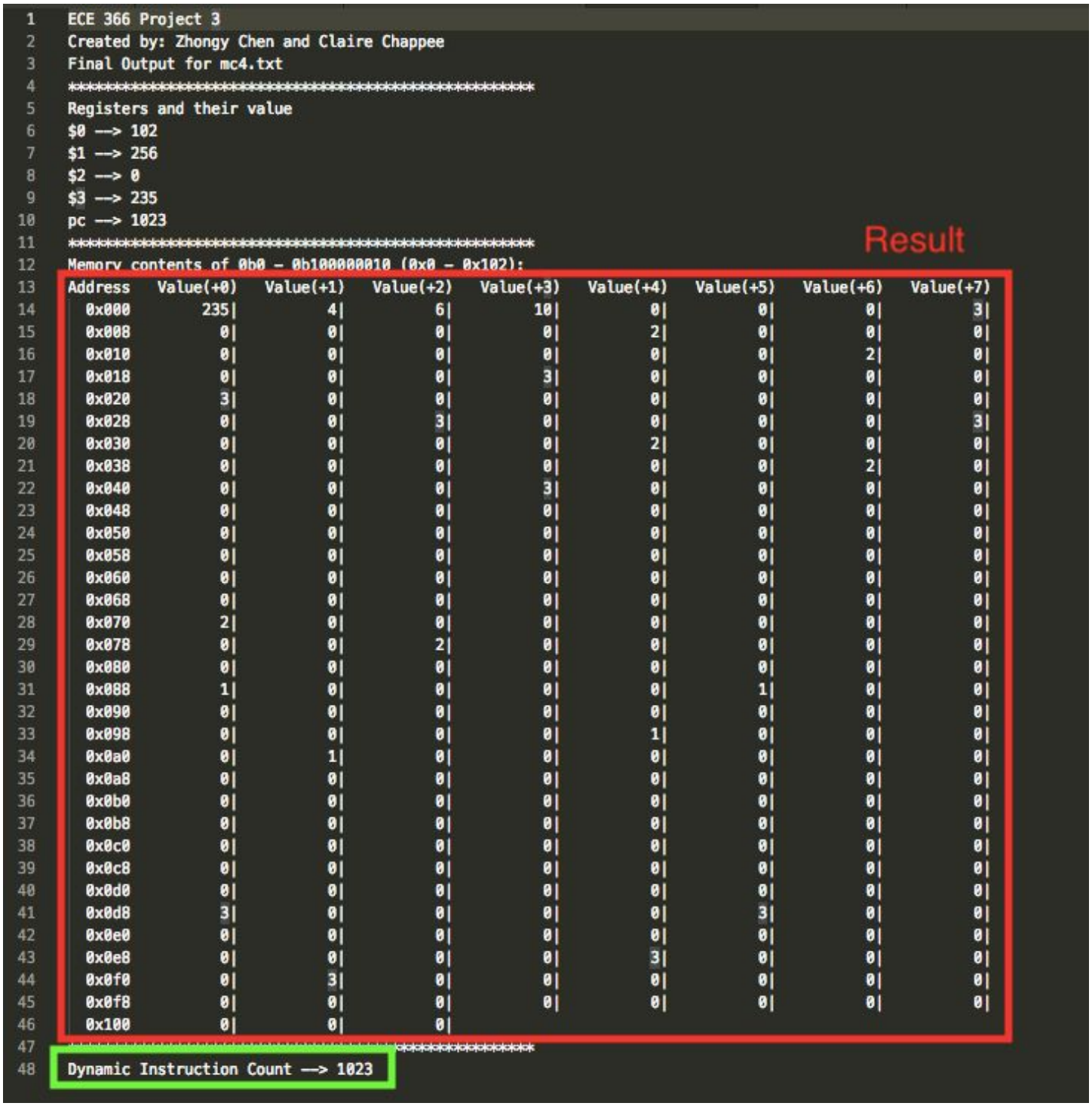
1 - 3	lui $0, 0x6
	addi $0, 0x6
	addi $1, 0x1
		x 1	Set B value 0x66 into register $0

4 - 1023	hash $2, $1, $0 ldinc $3, ($2) st $3, ($2) sto3inc $2, ($1)	x 255	Compute hash, Increment the times this pattern appeared, Store incremented value back to memory, Store hash value and increment iterator

2. Machine code (either in binary or hex) of your program. This should be the input to your python simulator.

Line Numbers	Code	Times occuring	Description
1 - 3	00000110 01000110 01010001	x 1	Set B value 0x66 into register $0
4 - 1023	10100100 11001110 11101110 11111001	x 255	Compute hash, Increment the times this pattern appeared, Store incremented value back to memory, Store hash value and increment iterator

3. Screenshots of your Python simulator's output for your program. Highlight the final results (pattern counts, all C values in memory) and Dynamic Instruction Count. You should provide sufficient screenshots to convince us that your ISA + Python package work correctly for any B values of the Hash program.



```
ECE 366 Project 3
Created by: Zhongy Chen and Claire Chappee
Final Output for mc4.txt
**************************************************
Registers and their value
$0 --> 102
$1 --> 256
$2 --> 0
$3 --> 235
pc --> 1023
**************************************************
Memory contents of 0b0 - 0b100000010 (0x0 - 0x102):
```

Result

Address	Value(+0)	Value(+1)	Value(+2)	Value(+3)	Value(+4)	Value(+5)	Value(+6)	Value(+7)
0x000	235	4	6	10	0	0	0	3
0x008	0	0	0	0	2	0	0	0
0x010	0	0	0	0	0	0	2	0
0x018	0	0	0	3	0	0	0	0
0x020	3	0	0	0	0	0	0	0
0x028	0	0	3	0	0	0	0	3
0x030	0	0	0	0	2	0	0	0
0x038	0	0	0	0	0	0	2	0
0x040	0	0	0	3	0	0	0	0
0x048	0	0	0	0	0	0	0	0
0x050	0	0	0	0	0	0	0	0
0x058	0	0	0	0	0	0	0	0
0x060	0	0	0	0	0	0	0	0
0x068	0	0	0	0	0	0	0	0
0x070	2	0	0	0	0	0	0	0
0x078	0	0	2	0	0	0	0	0
0x080	0	0	0	0	0	0	0	0
0x088	1	0	0	0	0	1	0	0
0x090	0	0	0	0	0	0	0	0
0x098	0	0	0	0	1	0	0	0
0x0a0	0	1	0	0	0	0	0	0
0x0a8	0	0	0	0	0	0	0	0
0x0b0	0	0	0	0	0	0	0	0
0x0b8	0	0	0	0	0	0	0	0
0x0c0	0	0	0	0	0	0	0	0
0x0c8	0	0	0	0	0	0	0	0
0x0d0	0	0	0	0	0	0	0	0
0x0d8	3	0	0	0	0	3	0	0
0x0e0	0	0	0	0	0	0	0	0
0x0e8	0	0	0	0	3	0	0	0
0x0f0	0	3	0	0	0	0	0	0
0x0f8	0	0	0	0	0	0	0	0
0x100	0	0	0					

```
**************************************************
Dynamic Instruction Count --> 1023
```

Python Code:

```
# ECE 366 Project 3 Fall 2019
# Group 12: Zhongyi Chen & Claire Chappee
f = 0;
instr_logging = True

registers = {
    '00': 0,
    '01': 0,
    '10': 0,
    '11': 0,
    'pc': 0
}
register_names = {
    '00': '$0',
    '01': '$1',
    '10': '$2',
    '11': '$3',
    'pc': 'pc'
}

# 2^9 needs to be the size of our memory
memory = [0] * 512
instr_memory = []

def executeLine(line):
    if(line[0:2] == "00"): #lui
        rx = line[2:4]
        imm = int(line[4:8], 2)
        immExt = imm << 4
        registers[rx] = immExt
        registers['pc'] += 1
        if (instr_logging == True):
            f.write("Instruction: lui " + register_names[rx] + ", " + str(imm) + '\n')
            f.write("Register[" + register_names[rx] + "] now has value " + str(registers[rx]) +
"\n")
            f.write("PC is now: " + str(registers['pc']) + "\n")

    if(line[0:2] == "01"): #addi
        rx = line[2:4]
        imm = int(line[4:8], 2)
        registers[rx] = registers[rx] + imm
```

```
        registers['pc'] += 1
        if (instr_logging == True):
            f.write("Instruction: addi " + register_names[rx] + ", " + str(imm) + "\n")
            f.write("Register[" + register_names[rx] + "] now has value " + str(registers[rx]) +
"\n")
            f.write("PC is now: " + str(registers['pc']) + "\n")

    if(line[0:2] == "10"): #hash
        rx = line[2:4]
        ry = line[4:6]
        rz = line[6:8]
        #registers[rx] = H(ry, rz)
        srcA = registers[ry]
        srcB = registers[rz]
        for i in range(0, 5):
            product = srcA * srcB
            hi = (product & 0xFF00) >> 8
            lo = product & 0xFF
            srcA = hi ^ lo
        srcB = srcA & 0xF
        srcA = (srcA & 0xF0) >> 4
        c = srcA ^ srcB
        registers[rx] = ((c & 0xC) >> 2) ^ (c & 0x3)
        registers['pc'] += 1
        if (instr_logging == True):
            f.write("Instruction: hash " + register_names[rx] + ", " + register_names[ry] + ", " +
register_names[rz] + "\n")
            f.write("Register[" + register_names[rx] + "] now has value " + str(registers[rx]) +
"\n")
            f.write("PC is now: " + str(registers['pc']) + "\n")

    if(line[0:4] == "1100"): #ldinc
        rx = line[4:6]
        ry = line[6:8]
        registers[rx] = memory[registers[ry]] + 1
        registers['pc'] += 1
        if (instr_logging == True):
            f.write("Instruction: ldinc " + register_names[rx] + ", " + "(" + register_names[ry] +
")\n")
            f.write("Register[" + register_names[rx] + "] now has value " + str(registers[rx]) +
"\n")
            f.write("PC is now: " + str(registers['pc']) + "\n")
```

```
    if(line[0:4] == "1110"): #st
        rx = line[4:6]
        ry = line[6:8]
        memory[registers[ry]] = registers[rx]
        registers['pc'] += 1
        if (instr_logging == True):
            f.write("Instruction: st " + register_names[rx] + ", " + "(" + register_names[ry] +
")\n" )
            f.write("Register[" + register_names[rx] + "] now has value " + str(registers[rx]) +
"\n")
            f.write("PC is now: " + str(registers['pc']) + "\n")

    if(line[0:4] == "1111"): #sto3inc
        rx = line[4:6]
        ry = line[6:8]
        memory[registers[ry] + 3] = registers[rx]
        registers[ry] += 1
        registers['pc'] += 1
        if (instr_logging == True):
            f.write("Instruction: sto3inc " + register_names[rx] + ", " + "(" + register_names[ry] +
")\n" )
            f.write("Register[" + register_names[rx] + "] now has value " + str(registers[rx]) +
"\n")
            f.write("PC is now: " + str(registers['pc']) + "\n")

def initializeInstrMemory(instr_mem_array, mCode):
    for line in mCode:
        line = line.strip()
        if (line == ''):
            continue
        else:
            instr_mem_array.append(line)

def main():
    global f
    global registers
    global register_names
    global memory
    global instr_memory

    f1 = open("outputFA.txt","w+")
    h1 = open("mc1.txt","r")
    f2 = open("output19.txt","w+")
```

```
    h2 = open("mc2.txt","r")
    f3 = open("outputE3.txt","w+")
    h3 = open("mc3.txt","r")
    f4 = open("output66.txt","w+")
    h4 = open("mc4.txt","r")

    if (instr_logging == True):
        f = f1
    mCode = h1.readlines()
    initializeInstrMemory(instr_memory, mCode)
    instrCount = len(instr_memory)
    dynamInstrCount = 0
    while (registers['pc'] < instrCount):
        mcLine = instr_memory[registers['pc']]
        executeLine(mcLine)
        dynamInstrCount += 1
    #Output for B = FA
    f1.write('\nECE 366 Project 3\n')
    f1.write('Created by: Zhongy Chen and Claire Chappee\n')
    f1.write('Final Output for mc1.txt\n')
    f1.write('*************************************************\n')
    f1.write('Registers and their value\n')
    for key in registers:
        f1.write(register_names[key] + ' --> ' + str(registers[key]) + '\n')
    f1.write('*************************************************\n')
    f1.write('Memory contents of 0b0 - 0b100000010 (0x0 - 0x102):\n')
    memOutputSize = 10
    f1.write('Address   Value(+0)   Value(+1)   Value(+2)   Value(+3)   Value(+4)   Value(+5)
Value(+6)   Value(+7)\n')
    for i in range(0, 259):
        if (i != 0 and i % 8 == 0):
            f1.write('\n')
        if (i % 8 == 0):
            f1.write('  0x%03x ' % i)
        for j in range(0, memOutputSize - len(str(memory[i]))):
            f1.write(' ')
        f1.write(str(memory[i]) + '| ')
    f1.write('\n*************************************************\n')
    f1.write('Dynamic Instruction Count --> ' + str(dynamInstrCount))
    #clear memory
    for key in registers:
        registers[key] = 0
    memory = [0] * 512
```

```
    instr_memory = []




    if (instr_logging == True):
        f = f2
    mCode = h2.readlines()
    initializeInstrMemory(instr_memory, mCode)
    instrCount = len(instr_memory)
    dynamInstrCount = 0
    while (registers['pc'] < instrCount):
        mcLine = instr_memory[registers['pc']]
        executeLine(mcLine)
        dynamInstrCount += 1
    #Output for B = 19
    f2.write('\nECE 366 Project 3\n')
    f2.write('Created by: Zhongy Chen and Claire Chappee\n')
    f2.write('Final Output for mc2.txt\n')
    f2.write('*************************************************\n')
    f2.write('Registers and their values\n')
    for key in registers:
        f2.write(register_names[key] + ' --> ' + str(registers[key]) + '\n')
    f2.write('*************************************************\n')
    f2.write('Memory contents of 0b0 - 0b100000010 (0x0 - 0x102):\n')
    memOutputSize = 10
    f2.write('Address   Value(+0)   Value(+1)   Value(+2)   Value(+3)   Value(+4)   Value(+5)
Value(+6)   Value(+7)\n')
    for i in range(0, 259):
        if (i != 0 and i % 8 == 0):
            f2.write('\n')
        if (i % 8 == 0):
            f2.write('  0x%03x ' % i)
        for j in range(0, memOutputSize - len(str(memory[i]))):
            f2.write(' ')
        f2.write(str(memory[i]) + '| ')
    f2.write('\n*************************************************\n')
    f2.write('Dynamic Instruction Count --> ' + str(dynamInstrCount))
    #clear memory
    for key in registers:
        registers[key] = 0
    memory = [0] * 512
    instr_memory = []
```

```
    if (instr_logging == True):
        f = f3
    mCode = h3.readlines()
    initializeInstrMemory(instr_memory, mCode)
    instrCount = len(instr_memory)
    dynamInstrCount = 0
    while (registers['pc'] < instrCount):
        mcLine = instr_memory[registers['pc']]
        executeLine(mcLine)
        dynamInstrCount += 1
    #Output for B = E3
    f3.write('\nECE 366 Project 3\n')
    f3.write('Created by: Zhongy Chen and Claire Chappee\n')
    f3.write('Final Output for mc3.txt\n')
    f3.write('*************************************************\n')
    f3.write('Registers and their values\n')
    for key in registers:
        f3.write(register_names[key] + ' --> ' + str(registers[key]) + '\n')
    f3.write('*************************************************\n')
    f3.write('Memory contents of 0b0 - 0b100000010 (0x0 - 0x102):\n')
    memOutputSize = 10
    f3.write('Address   Value(+0)   Value(+1)   Value(+2)   Value(+3)   Value(+4)   Value(+5)
Value(+6)   Value(+7)\n')
    for i in range(0, 259):
        if (i != 0 and i % 8 == 0):
            f3.write('\n')
        if (i % 8 == 0):
            f3.write('  0x%03x ' % i)
        for j in range(0, memOutputSize - len(str(memory[i]))):
            f3.write(' ')
        f3.write(str(memory[i]) + '| ')
    f3.write('\n*************************************************\n')
    f3.write('Dynamic Instruction Count --> ' + str(dynamInstrCount))
    #clear memory
    for key in registers:
        registers[key] = 0
    memory = [0] * 512
    instr_memory = []
```

```
    if (instr_logging == True):
        f = f4
    mCode = h4.readlines()
    initializeInstrMemory(instr_memory, mCode)
    instrCount = len(instr_memory)
    dynamInstrCount = 0
    while (registers['pc'] < instrCount):
        mcLine = instr_memory[registers['pc']]
        executeLine(mcLine)
        dynamInstrCount += 1
    #Output for B = 66
    f4.write('\nECE 366 Project 3\n')
    f4.write('Created by: Zhongy Chen and Claire Chappee\n')
    f4.write('Final Output for mc4.txt\n')
    f4.write('**************************************************\n')
    f4.write('Registers and their value\n')
    for key in registers:
        f4.write(register_names[key] + ' --> ' + str(registers[key]) + '\n')
    f4.write('**************************************************\n')
    f4.write('Memory contents of 0b0 - 0b100000010 (0x0 - 0x102):\n')
    memOutputSize = 10
    f4.write('Address   Value(+0)   Value(+1)   Value(+2)   Value(+3)   Value(+4)   Value(+5)
Value(+6)   Value(+7)\n')
    for i in range(0, 259):
        if (i != 0 and i % 8 == 0):
            f4.write('\n')
        if (i % 8 == 0):
            f4.write('  0x%03x ' % i)
        for j in range(0, memOutputSize - len(str(memory[i]))):
            f4.write(' ')
        f4.write(str(memory[i]) + '| ')
    f4.write('\n**************************************************\n')
    f4.write('Dynamic Instruction Count --> ' + str(dynamInstrCount))
    #clear memory
    for key in registers:
        registers[key] = 0
    memory = [0] * 512
    instr_memory = []

if __name__ == "__main__":
    main()
```