ECE 366 Project 3 Report

Fall 2019 Group #9 - Lil' Monster ISA

Sheng Chen
Daniel Serrano
Arsalan Sheikh

Summary of group activity

Time / Location	Activity	Achieved / To Do	Attended
Thursday Oct 24th 10:00am - 11:30 pm @ Library	In-person	**Done:** ❏ Basic ISA designing. ❏ Discussed HASH design. ❏ Brainstorm on the 8 bit instructions. **To Do:** ❏ Figure out our final 8 instructions ❏ Hash function ❏ Pattern Matching ❏ Op Code structure	Sheng Chen Daniel Serrano Arsalan Sheikh
Thursday Oct 31st 12:00pm - 2:00pm @ Library	In-person	**Done:** ❏ Main ISA structure and designing ❏ Op code structure **To Do:** ❏ BNE and SB instructions ❏ Memory Addressing ❏ Pattern Matching	Sheng Chen Daniel Serrano Arsalan Sheikh
Monday Nov 4th 6:00pm - 7:30pm @ Library	In-person	**Done:** ❏ Changes to Op code structure ❏ BNE and SB instructions ❏ Pattern Matching ❏ Hardware addressing **To Do:** ❏ Simulator needs to be finished up	Sheng Chen Daniel Serrano
Tuesday Nov 5th 3:30pm - 9:30pm @ Library	In-person	**Done:** ❏ Changes in Memory instructions **To Do:** ❏ Debugging on Hash and simulator.	Sheng Chen Daniel Serrano Arsalan Sheikh

Individual activity list

Sheng: Instruction and ALU design
Daniel: Assembly code implementation, Python simulation
Arsalan: Hardware Implementation

Group work reflection

How does this group differ from the previous one?

Members of this group appear to have more of an idea of what class-related skills they are comfortable with. We communicated very well which helped us understand each individuals strong areas. From a personal perspective, the members were more understanding of each others' prior commitments that were also competing with this project for time and effort. Despite everyones' busy schedules, the group managed to spare some time and get in some thoughts and ideas about our approach.

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

Constant, meaningful communication with simple, baby-step goals set afterwards helps to motivate group members to accomplish a lot together. More frequent meetups really helped us get a good idea of our progress and timeline.

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

Individual work on the project would become very time-consuming as students would need to strengthen every skill needed to complete this complex project. Not having someone to consult and corrections on small mistakes is very key and is very helpful. Furthermore, without 24-hour access to the professor and assisting TAs, the lengthy procedures to finish the project would be very error-prone, and errors would be time-consuming to resolve.

Part A) Lil' Monster ISA Introduction

Overview:

The Lil' Monster ISA is an instruction set architecture designed to hash two 8-bit numbers using a byte-sized system for calculation with as few instructions as possible, with hash results that are easy to read. After some careful thought, our group decided on an ISA with only 8 instructions to learn, and an easily-accessible memory structure.

The Lil' Monster ISA is single-cycle instruction set architecture with 8-bit instruction length, utilizing an 8-bit register file and byte-addressable data memory (loading and storing 8 bits per memory instruction). Given an incrementing 8-bit number **[A]** and a user-specified 8-bit number **[B]**, CPUs using the Lil' Monster ISA are designed to hash **[A,B]** together to produce a 2-bit result **[C]**. In addition, the CPU implementation produces a running count of all four of the possible 2-bit results of C from computing the hash function for every value of A.

Instruction List:

Name	Opcode	ADDIR		Imm	Notes
ADDI	000	X (R1 = A or B only)		ZZZZ	R1: 0 for A, 1 for B
Name	**Opcode**	**R1**	**Code**	**Mode**	**Notes**
BNE	001	XX	YY	Z	**Code:** As unsigned binary, zero-extended **Mode:** 0 for compare-to-255 and possible branch to PC = 0, 1 for compare to Code and possible branch to PC + 3
Name	**Opcode**	**R1**	**R2**	**Special**	**Notes**
SLL	010	XX	YY	Z	**Special:** 0 for shift 2 bits, 1 for shift 4 bits
SRL	011	XX	YY	Z	**Special:** 0 for shift 2 bits, 1 for shift 4 bits
MULT	100	XX	YY	Don't Care	MSB of product overwrites hi, LSB of product overwrites lo
XOR	101	XX	YY	Don't Care	Result overwrites hi

Name	Opcode	R1	Code	0-or-A	Notes
LBU	110	XX	YY	Z	**Code:** As unsigned binary, zero-extended **0-or-A:** 0 for 0, 1 for A (See Data Memory Addressing)
SBU	111	XX	YY	Z	**Code:** As unsigned binary, zero-extended **0-or-A:** 0 for 0, 1 for A (See Data Memory Addressing)

Instruction explanations and examples:

Add Immediate: `addi $addir, imm`
Adds a zero-extended, 4-bit immediate number to either Register A or B.
Opcode: `000` **Example:** `addi $B, 15`

Branch Not Equal to Code or Constant: `bne $R1, code, mode`
If Mode == 0, compares the contents of R1 to 0b11111111. If not equal, PC is set to 0.
If Mode == 1, compares the contents of R1 to zero-extended Code. If not equal, PC +3.
Opcode: `001` **Example (Compare to Constant):** `bne $A, 0, 0`
Example (Compare to Code): `bne $hi, 2, 1`

Shift Left Logical, 2 or 4 bits: `sll $R1, $R2, special`
Shifts the contents of R2 to the left, placing the results in R1.
Value of Special bit directs instruction to shift 2 bits if 0, 4 bits if 1.
Opcode: `010` **Example:** `sll $lo, $hi, 1`

Shift Right Logical, 2 or 4 bits: `srl $R1, $R2, special`
Shifts the contents of R2 to the right, placing the results in R1.
Value of Special bit directs instruction to shift 2 bits if 0, 4 bits if 1.
Opcode: `011` **Example:** `srl $lo, $lo, 1`

Multiply: `mult $R1, $R2`
Multiplies R1 and R2 to get a 16-bit result. Most significant 8 bits of result are placed in Register hi, least significant 8 bits of result are placed in Register lo.
Opcode: `100` **Example:** `mult $A, $B`

XOR: `xor $R1, $R2`
Takes the logical XOR of R1 and R2 and places the result in Register hi.
Opcode: `101` **Example:** `xor $hi, $lo`

Load Byte, Unsigned: `lbu $R1, code(0orA)`
Loads the contents of DataMemory[code + (0 or contents of Register A)] into R1.
A is interpreted as 0-or-A bit = 1 in machine code.
Opcode: `110` **Example:** `lbu $B, 1(0)`

Store Byte, Unsigned: `sbu $R1, code(0orA)`
Stores the contents of R1 into DataMemory[code + (0 or contents of Register A)].
A is interpreted as 0-or-A bit = 1 in machine code.
Opcode: `111` **Example:** `sbu $hi, 3(A)`

Register design & table:

Lil' Monster CPU register files contain 2 general-purpose registers called A and B, and 2 semi-specialized registers called hi and lo.

A specialized 8-bit PC register is initialized to 0 at the beginning of machine code execution (see next section, Branch Design).

Register name	2-bit register alias (R1, R2)	Special purposes, if any
A	00	None
B	01	None
hi	10	Most-sig 8 bits of MULT instruction overwrites here. Result of XOR instruction overwrites here.
lo	11	Least-sig 8 bits of MULT instruction overwrites here.
PC	———	Increments by 1 per instruction fetched. Modifiable by BNE instruction.

Branch design:

The Lil' Monster ISA has one type of branch instruction, BNE, that has a specialized manner of execution (and happens to be very different from a MIPS BNE instruction, by comparison). This ISA's BNE instruction compares the contents of **one register** to one out of 5 numbers in two different modes of comparison, differentiated by a special bit y (see Instruction List). If **y = 0**, the **register** will be **compared to 0b11111111** (binary representation of 255), and will branch directly to

InstructionMemory[3] if the two values are not equal. If **y = 1**, the **register** will be compared to the 2-bit value specified after bit y (zero-extended to 8 bits), known as **zz** (see Instruction List). In this mode, if the register is **not equal to 0x000000zz**, the BNE instruction will add 2 to PC in addition to its regular increment-per-instruction, resulting in **PC + 3**.

Data memory addressing modes:

The Lil' Monster ISA loads and stores to data memory with byte addressing. Maximum data memory supported by the ISA and its instructions is 259 Bytes (2,072 bits among 259 addresses). The access address **M[address]** is calculated by a Lil' Monster CPU's ALU as a sum of the contents of a register [A, B, hi, or lo] with a 2-bit immediate number.

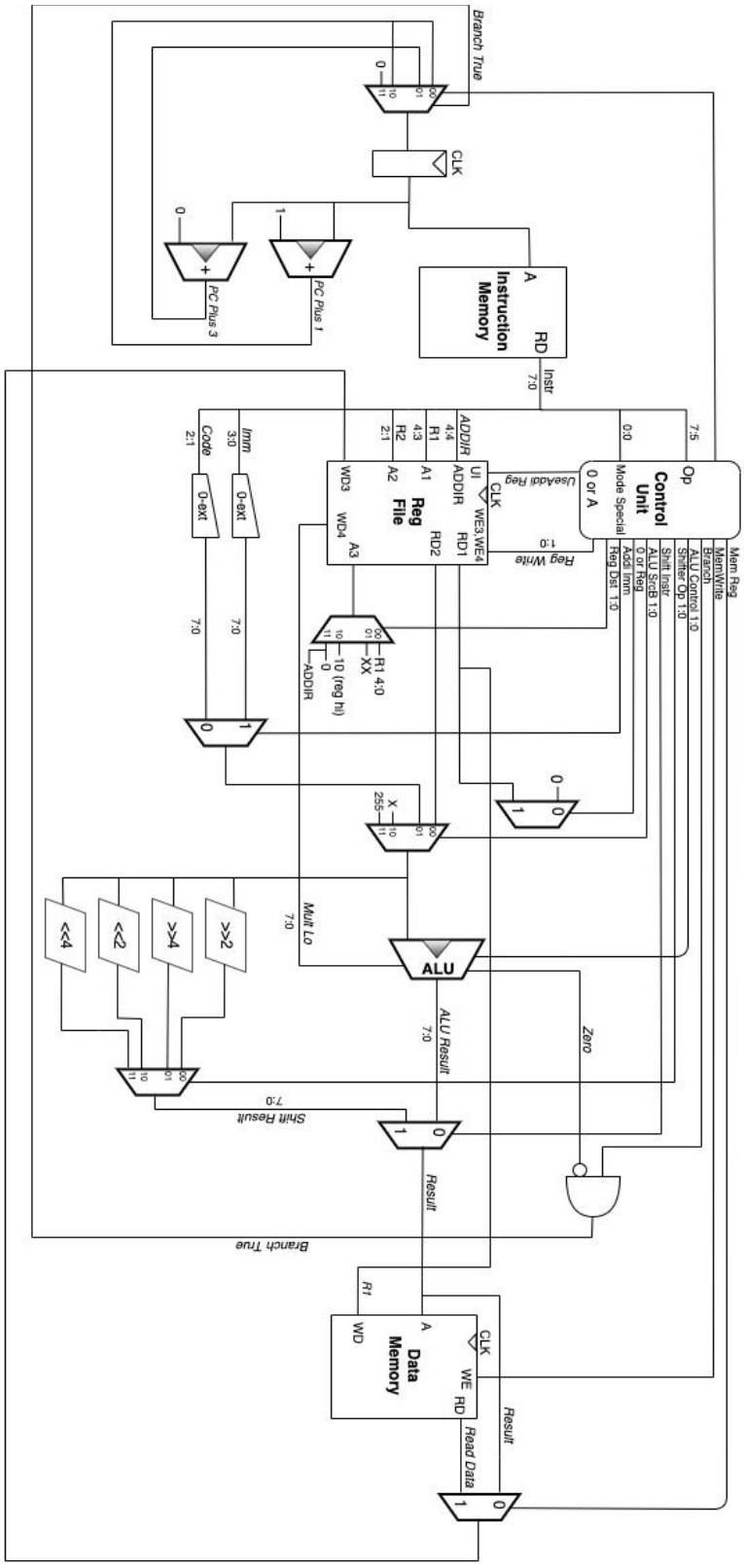
Significant features of ISA:

Five out of the eight instructions in the set are implemented with a mode-switching bit in their least-significant position. This bit is labeled **Mode/Special/0-or-A**. The bit can modify the associated instruction to target specific register locations or hard-coded immediate values not specified in other fields of the instruction, or the bit can specify the amount of bits to shift in a shift instruction. The bit is ignored in the MULT and XOR instructions, and is used otherwise as a part of a 4-bit immediate value in the ADDI instruction.

A number of values have been hardcoded for the purposes of register addressing and branching. In addition, to reduce the number of number of multiplexers used outside of the ALU, the Mode/Special/0-orA bit has been routed into the Control Unit for abstraction of the special instruction modifications mentioned in the previous paragraph. Finally, to slightly reduce the complexity of the ALU, dedicated shifters were implemented so that the ALU only needs to handle four different types of arithmetic/logical operations.

Part B) Hardware Implementation

Full CPU datapath schematic:



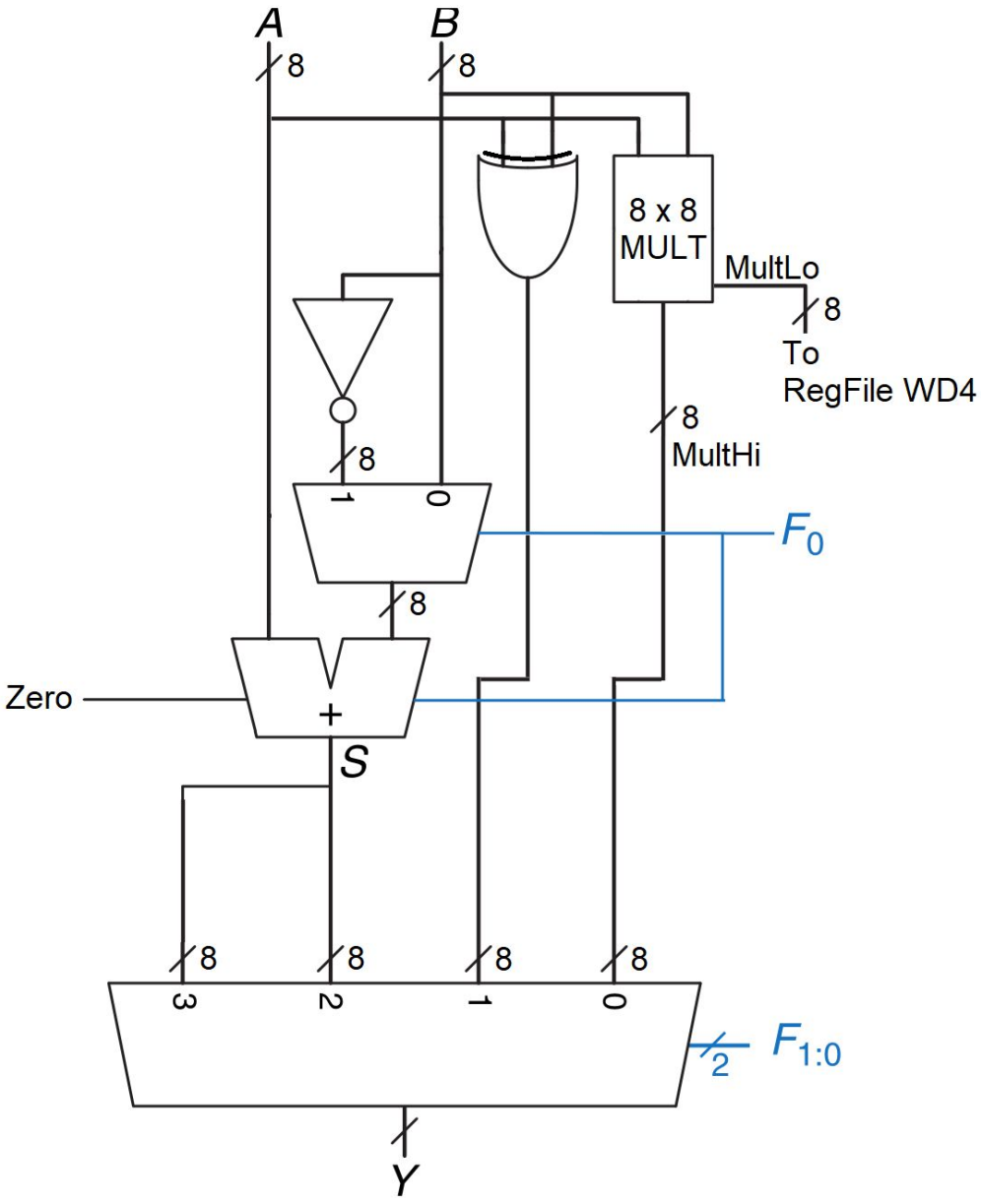
Control unit signal tables, by instruction:

Op	UseAddiReg	RegWrite (1:0)	RegDst (1:0)	AddiImm	0orReg	ALUSrcB (1:0)
ADDI	1	10	11 (0,addir)	1	1	01
BNE	0	00	XX	1	1	If Special == 0 then 11 If Special == 1 then 01
SLL	0	10	00 (R1)	X	X	XX
SRL	0	10	00 (R1)	X	X	XX
MULT	0	11	10 (hi)	X	1	00
XOR	0	10	10 (hi)	X	1	00
LBU	0	10	00 (R1)	0	1	01
SBU	0	00	XX	0	1	01

Control unit signal tables, continued:

Op	Shift Instr	ShifterOp (1:0)	ALUControl (1:0)	Branch	Mem Write	Memt oReg	BranchMode
ADDI	0	XX	10 (add)	0	0	0	X
BNE	X	XX	11 (sub)	1	0	X	If Special == 0 then 1 If Special == 1 then 0
SLL	1	If Special == 0 then 10 If Special == 1 then 11	XX	0	0	0	X
SRL	1	If Special == 0 then 00 If Special == 1 then 01	XX	0	0	0	X
MULT	0	XX	00 (mult)	0	0	0	X
XOR	0	XX	01 (xor)	0	0	0	X
LBU	0	XX	10 (add)	0	0	1	X
SBU	0	XX	10 (add)	0	1	X	X

ALU schematic:



ALU operation table (ALUControl 1:0):

ALUControl (F)	Operation
00	Multiply
01	XOR
10	Add
11	Subtract

Result Screenshots

B = 0xFA

```
Dynamic Instr Count:  9180
Registers $0 - $3:    | 255[$A] | 33[$B] | 0[$Hi] | 0[$Low] | 42[$PC] |
Memory contents in Decimal:
Mem[0]:     [74]   [65]   [83]   [33]
Mem[4]:     [0]    [3]    [0]    [1]    [1]    [2]
Mem[10]:    [3]    [1]    [0]    [2]    [3]    [2]    [1]    [2]    [3]    [2]
Mem[20]:    [2]    [0]    [1]    [0]    [0]    [1]    [1]    [2]    [0]    [0]
Mem[30]:    [0]    [1]    [2]    [1]    [3]    [3]    [2]    [1]    [2]    [3]
Mem[40]:    [2]    [1]    [1]    [3]    [3]    [2]    [1]    [2]    [2]    [0]
Mem[50]:    [3]    [0]    [3]    [2]    [0]    [0]    [2]    [3]    [1]    [0]
Mem[60]:    [0]    [1]    [2]    [2]    [2]    [2]    [2]    [1]    [1]    [2]
Mem[70]:    [0]    [1]    [0]    [3]    [2]    [1]    [0]    [2]    [2]    [0]
Mem[80]:    [0]    [1]    [2]    [2]    [0]    [0]    [3]    [2]    [1]    [2]
Mem[90]:    [2]    [3]    [1]    [2]    [0]    [0]    [3]    [0]    [2]    [0]
Mem[100]:   [2]    [1]    [0]    [1]    [0]    [0]    [2]    [3]    [0]    [0]
Mem[110]:   [1]    [0]    [3]    [0]    [0]    [0]    [2]    [2]    [3]    [1]
Mem[120]:   [1]    [2]    [0]    [1]    [1]    [3]    [0]    [2]    [2]    [2]
Mem[130]:   [2]    [0]    [1]    [2]    [2]    [2]    [1]    [1]    [0]    [1]
Mem[140]:   [2]    [0]    [1]    [1]    [2]    [2]    [2]    [1]    [2]    [2]
Mem[150]:   [2]    [3]    [2]    [2]    [2]    [2]    [0]    [2]    [1]    [0]
Mem[160]:   [3]    [0]    [2]    [1]    [2]    [0]    [2]    [3]    [1]    [1]
Mem[170]:   [1]    [3]    [2]    [2]    [3]    [3]    [2]    [1]    [1]    [0]
Mem[180]:   [0]    [0]    [1]    [2]    [0]    [0]    [3]    [1]    [0]    [0]
Mem[190]:   [3]    [0]    [1]    [1]    [0]    [2]    [1]    [1]    [3]    [1]
Mem[200]:   [2]    [0]    [3]    [0]    [2]    [0]    [1]    [0]    [0]    [2]
Mem[210]:   [0]    [1]    [0]    [2]    [0]    [2]    [2]    [1]    [0]    [3]
Mem[220]:   [1]    [1]    [3]    [0]    [0]    [1]    [2]    [2]    [2]    [2]
Mem[230]:   [0]    [0]    [0]    [1]    [2]    [0]    [1]    [1]    [0]    [1]
Mem[240]:   [2]    [0]    [1]    [2]    [2]    [1]    [2]    [0]    [1]    [0]
Mem[250]:   [2]    [0]    [2]    [3]    [1]    [2]    [2]    [1]    [0]
```

B = 0x19

```
Simulation Finished
Dynamic Instr Count:  9180
Registers $0 - $3:    | 255[$A] | 11[$B] | 0[$Hi] | 0[$Low] | 42[$PC] |
Memory contents in Decimal:
Mem[0]:     [146]   [52]   [46]   [11]
Mem[4]:     [2]    [0]    [3]    [0]    [0]    [3]
Mem[10]:    [0]    [0]    [2]    [0]    [1]    [0]    [1]    [0]    [1]    [0]
Mem[20]:    [0]    [1]    [0]    [0]    [2]    [0]    [0]    [0]    [1]    [0]
Mem[30]:    [1]    [0]    [1]    [0]    [2]    [0]    [2]    [0]    [1]    [1]
Mem[40]:    [0]    [0]    [0]    [0]    [2]    [2]    [0]    [1]    [1]    [1]
Mem[50]:    [1]    [2]    [2]    [1]    [0]    [3]    [1]    [1]    [0]    [0]
Mem[60]:    [0]    [0]    [0]    [1]    [0]    [3]    [0]    [0]    [0]    [0]
Mem[70]:    [3]    [1]    [0]    [2]    [0]    [2]    [0]    [0]    [0]    [0]
Mem[80]:    [0]    [0]    [3]    [1]    [0]    [0]    [0]    [0]    [1]    [2]
Mem[90]:    [2]    [1]    [0]    [0]    [0]    [1]    [0]    [0]    [0]    [1]
Mem[100]:   [0]    [1]    [1]    [2]    [0]    [0]    [2]    [0]    [3]    [0]
Mem[110]:   [0]    [0]    [0]    [2]    [2]    [0]    [0]    [2]    [2]    [0]
Mem[120]:   [0]    [3]    [0]    [0]    [0]    [2]    [2]    [0]    [2]    [0]
Mem[130]:   [0]    [0]    [1]    [0]    [0]    [0]    [0]    [0]    [0]    [1]
Mem[140]:   [2]    [2]    [0]    [1]    [0]    [1]    [0]    [2]    [0]    [0]
Mem[150]:   [0]    [1]    [2]    [0]    [2]    [0]    [0]    [2]    [0]    [0]
Mem[160]:   [0]    [1]    [0]    [0]    [0]    [2]    [0]    [1]    [2]    [0]
Mem[170]:   [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [2]    [0]
Mem[180]:   [0]    [1]    [1]    [2]    [0]    [1]    [1]    [0]    [1]    [2]
Mem[190]:   [0]    [0]    [2]    [0]    [0]    [2]    [2]    [1]    [0]    [0]
Mem[200]:   [0]    [1]    [0]    [1]    [2]    [0]    [0]    [0]    [3]    [0]
Mem[210]:   [0]    [1]    [0]    [3]    [0]    [2]    [1]    [0]    [0]    [2]
Mem[220]:   [0]    [0]    [1]    [1]    [0]    [0]    [2]    [0]    [0]    [2]
Mem[230]:   [0]    [1]    [3]    [0]    [0]    [0]    [1]    [0]    [1]    [0]
Mem[240]:   [0]    [2]    [2]    [2]    [0]    [0]    [1]    [1]    [0]    [2]
Mem[250]:   [2]    [1]    [0]    [0]    [1]    [0]    [1]    [0]    [0]
```

B = 0xE3

```
Dynamic Instr Count:  9180
Registers $0 - $3:    | 255[$A] | 43[$B] | 0[$Hi] | 0[$Low] | 42[$PC] |
Memory contents in Decimal:
Mem[0]:     [74]   [68]   [70]   [43]
Mem[4]:     [0]    [2]    [1]    [3]    [1]    [3]
Mem[10]:    [3]    [0]    [0]    [2]    [2]    [0]    [1]    [0]    [2]    [0]
Mem[20]:    [0]    [1]    [3]    [1]    [3]    [2]    [1]    [0]    [3]    [3]
Mem[30]:    [2]    [1]    [0]    [1]    [0]    [2]    [0]    [2]    [1]    [2]
Mem[40]:    [2]    [2]    [3]    [0]    [0]    [0]    [1]    [0]    [0]    [2]
Mem[50]:    [0]    [2]    [2]    [3]    [3]    [3]    [0]    [3]    [0]    [2]
Mem[60]:    [1]    [0]    [0]    [0]    [1]    [1]    [1]    [1]    [2]    [3]
Mem[70]:    [1]    [1]    [2]    [2]    [2]    [1]    [1]    [3]    [3]    [1]
Mem[80]:    [0]    [0]    [3]    [3]    [1]    [2]    [0]    [0]    [3]    [0]
Mem[90]:    [1]    [2]    [2]    [0]    [0]    [1]    [0]    [1]    [2]    [2]
Mem[100]:   [3]    [1]    [0]    [1]    [2]    [2]    [0]    [2]    [2]    [1]
Mem[110]:   [2]    [2]    [2]    [1]    [0]    [3]    [3]    [2]    [1]    [2]
Mem[120]:   [0]    [3]    [0]    [2]    [1]    [3]    [2]    [2]    [1]    [0]
Mem[130]:   [2]    [0]    [2]    [3]    [0]    [1]    [2]    [2]    [0]    [0]
Mem[140]:   [2]    [1]    [2]    [2]    [3]    [1]    [2]    [1]    [1]    [1]
Mem[150]:   [1]    [2]    [2]    [1]    [0]    [2]    [2]    [1]    [1]    [0]
Mem[160]:   [3]    [0]    [1]    [2]    [1]    [2]    [3]    [1]    [3]    [0]
Mem[170]:   [0]    [0]    [2]    [2]    [0]    [0]    [2]    [0]    [3]    [1]
Mem[180]:   [0]    [3]    [1]    [0]    [3]    [3]    [1]    [1]    [3]    [1]
Mem[190]:   [0]    [2]    [0]    [1]    [1]    [1]    [1]    [0]    [3]    [1]
Mem[200]:   [0]    [3]    [1]    [1]    [1]    [1]    [1]    [3]    [0]    [0]
Mem[210]:   [2]    [2]    [1]    [1]    [0]    [0]    [1]    [0]    [0]    [2]
Mem[220]:   [1]    [3]    [2]    [2]    [2]    [0]    [1]    [0]    [2]    [0]
Mem[230]:   [1]    [2]    [2]    [0]    [3]    [2]    [3]    [2]    [2]    [0]
Mem[240]:   [0]    [0]    [0]    [3]    [0]    [2]    [0]    [0]    [2]    [1]
Mem[250]:   [0]    [3]    [2]    [2]    [3]    [1]    [3]    [1]    [0]
```

B = 0x66

```
Dynamic Instr Count:  9180
Registers $0 - $3:    | 255[$A] | 10[$B] | 0[$Hi] | 0[$Low] | 42[$PC] |
Memory contents in Decimal:
Mem[0]:     [235]   [4]   [6]   [10]
Mem[4]:     [0]    [0]    [0]    [3]    [0]    [0]
Mem[10]:    [0]    [0]    [2]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[20]:    [0]    [0]    [2]    [0]    [0]    [0]    [0]    [3]    [0]    [0]
Mem[30]:    [0]    [0]    [3]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[40]:    [0]    [0]    [3]    [0]    [0]    [0]    [0]    [3]    [0]    [0]
Mem[50]:    [0]    [0]    [2]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[60]:    [0]    [0]    [2]    [0]    [0]    [0]    [0]    [3]    [0]    [0]
Mem[70]:    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[80]:    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[90]:    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[100]:   [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[110]:   [0]    [0]    [2]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[120]:   [0]    [0]    [2]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[130]:   [0]    [0]    [0]    [0]    [0]    [0]    [1]    [0]    [0]    [0]
Mem[140]:   [0]    [1]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[150]:   [0]    [0]    [0]    [0]    [0]    [0]    [1]    [0]    [0]    [0]
Mem[160]:   [0]    [1]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[170]:   [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[180]:   [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[190]:   [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[200]:   [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[210]:   [0]    [0]    [0]    [0]    [0]    [0]    [3]    [0]    [0]    [0]
Mem[220]:   [0]    [3]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[230]:   [0]    [0]    [0]    [0]    [0]    [0]    [3]    [0]    [0]    [0]
Mem[240]:   [0]    [3]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
Mem[250]:   [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]    [0]
```

ISA Python Simulator Code:

```
def sim(program):
    # Machine Code to Simulation

    finished = False  # Is the simulation finished?
    PC = 0  # Program Counter
    register = [0] * 5  # Let's initialize 4 empty registers #[A,B,Hi,Low]
    #[A,B,Hi,Low]
    mem = [0] * 259  # Let's initialize 259 spots in memory

    DIC = 0  # Dynamic Instr Count
    while (not (finished)):

        if PC == len(program) - 1:
            finished = True
            register[4] = PC + 1

        fetch = program[PC]
        DIC += 1

        # ADDI  - Sim
        if fetch[0:3] == '000':
            PC += 1
            r1 = int(fetch[3], 2)
            imm = int(fetch[4:], 2)

            register[r1] += imm

        # mult  - Sim
        elif fetch[0:3] == '100':
            PC += 1
            r1 = int(fetch[3:5], 2)
            r2 = int(fetch[5:7], 2)

            result = register[r1] * register[r2]
            register[3] = result & 0x00FF # LOW
            register[2] = result >> 8 # HI



        # SBU - Sim
        elif fetch[0:3] == '111':
            PC += 1
            r1 = int(fetch[3:5], 2)
            imm = int(fetch[5:7], 2)
            Zero_A = int(fetch[7], 2)
            if Zero_A == 0:
```

```
        mem[0 + imm] = register[r1]
    else:
        mem[register[0] + imm] = register[r1]

# LBU - Sim
elif fetch[0:3] == '110':
    PC += 1
    r1 = int(fetch[3:5], 2)
    imm = int(fetch[5:7], 2)
    Zero_A = int(fetch[7], 2)
    if Zero_A == 0:
        register[r1] = mem[0 + imm]
    else:
        register[r1] = mem[register[0] + imm]
# SRL
elif fetch[0:3] == '011':
    PC += 1
    r1 = int(fetch[3:5], 2)
    r2 = int(fetch[5:7], 2)
    Z = int(fetch[7], 2)
    # if Z is 0 then we rigth shift 2 bits and if its 1 then we right shift 4 bits
    if Z == 0:
        register[r1] = int(register[r2] / 4)
    else:
        register[r1] = int(register[r2] / 16)


# SLL
elif fetch[0:3] == '010':
    PC += 1
    r1 = int(fetch[3:5], 2)
    r2 = int(fetch[5:7], 2)
    Z = int(fetch[7], 2)
    if Z == 0:
        register[r1] = int(register[r2] * 4)
    else:
        register[r1] = int(register[r2] * 16)
    temp = register[r1]

    # if number is larger than 8 bits, we need to cut off the extra bits
    if temp > 255:
        temp = format(temp, '016b')
        length = len(temp)
        start = length - 8
        temp = temp[start:]  # this "function" if it
        i = int(temp, 2)     # overflows past 8 digits
        register[r1] = i

    elif temp > 15 and r1 == 3: # so this only happens in the last 2 folds of the hashing
```

```
            temp = format(temp, '08b')
            length = len(temp)
            if Z == 0: # if we shifted by 2 bits, then this is the last fold, we cut off if it overflows 4 bits
                start = length - 4
            elif Z == 1: # if we shifted by 4 bits, then this is the second to last fold and we cut off it it overflows 8 bits
                start = length - 8
            temp = temp[start:]
            i = int(temp, 2)
            register[r1] = i



    # XOR  - Sim
    elif fetch[0:3] == '101':
        PC += 1
        r1 = int(fetch[3:5], 2)
        r2 = int(fetch[5:7], 2)
        register[2] = int(register[r1]) ^ int(register[r2])



    # BNE  - Sim
    elif fetch[0:3] == '001':
        PC += 1
        r1 = int(fetch[3:5], 2)
        YY = int(fetch[5:7],2)
        Z = int(fetch[7],2)
        if Z == 0: #if special bit is 0, then we jump back to the beginning of the loop
            if register[r1] != 255:
                PC = 0
                finished = False
        else: # if Z = 1, then this is pattern matching part

            if YY == 1 and register[r1] != 1:
                PC += 2
            elif YY == 0 and register[r1] != 0:
                PC += 2
            elif YY == 3 and register[r1] != 3:
                PC += 2
            elif YY == 2 and register[r1] != 2:
                PC += 2



# Finished simulations. Let's print out some stats
print('***Simulation finished***')

print('Dynamic Instr Count: ', DIC)
```

```
    print('Registers $0 - $3:    | {}[$A] | {}[$B] | {}[$Hi] | {}[$Low] | {}[$PC] |'.format(
        register[0], register[1], register[2], register[3], register[4]))

    print('Memory contents in Decimal:')

    print('Mem[0]:     ', end='')

    for x in range(len(mem)):
        if x % 10 == 0 and x != 0:
            print(' ')
            if x >= 100:
                print('Mem[' + str(x) + ']:  ', end=' ')
            else:
                print('Mem[' + str(x) + ']:   ', end=' ')
        if x >= 4:
            if x < 100:
                print('[' + str(mem[x]) + ']   ', end=' ')
            else:
                print('[' + str(mem[x]) + ']   ', end=' ')
        else:
            print('[' + str(mem[x]) + ']  ', end=' ')

        if x == 3:
            print(' ')
            print('Mem[4]:     ', end='')

    input()


def main():

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

    asm = h.readlines()

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

    for line in asm:

        if line[0] == "#":
            continue

        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
```

```
# = = = = ADDI = = = = = = = = =
if (line[0:4] == "addi"):
    line = line.replace("addi", "")
    line = line.split(",")

    imm = format(int(line[1]), '04b') if (int(line[1]) >= 0) else format(16 + int(line[1]), '04b')
    r1 = format(0, '01b')  # Assuming its A
    if (line[0] == "B"):
        r1 = format(1, '01b')


    f.write(str('000') + str(r1) + str(imm) + '\n')


# = = = = MULT = = = = = = = = =
elif (line[0:4] == "mult"):
    line = line.replace("mult", "")
    line = line.split(",")

    # defaults to register[0] or A
    r1 = 0
    r2 = 0

    # Since we use letters in assembly code
    if (line[0] == "B"):
        r1 = 1
    elif (line[0] == "hi"):
        r1 = 2
    elif (line[0] == "lo"):
        r1 = 3

    if (line[1] == "B"):
        r2 = 1
    elif (line[1] == "hi"):
        r2 = 2
    elif (line[1] == "lo"):
        r2 = 3

    r1 = format(r1, '02b')  # make element 1 in the set, 'line' an int of 2 bits. (r1)
    r2 = format(r2, '02b')  # make element 2 in the set, 'line' an int of 2 bits. (r2)

    f.write(str('100') + str(r1) + str(r2) + str('0') + '\n')

# = = = = SRL = = = = = = = = =
elif (line[0:3] == "srl"):
    line = line.replace("srl", "")
    line = line.split(",")

    # defaults to register[0] or A
```

```
    r1 = 0
    r2 = 0

    # Since we use letters in assembly code
    if (line[0] == "B"):
        r1 = 1
    elif (line[0] == "hi"):
        r1 = 2
    elif (line[0] == "lo"):
        r1 = 3

    if (line[1] == "B"):
        r2 = 1
    elif (line[1] == "hi"):
        r2 = 2
    elif (line[1] == "lo"):
        r2 = 3
    r1 = format(r1, '02b')  # make element 0 in the set, 'line' an int of 2 bits. (r1)
    r2 = format(r2, '02b')  # make element 1 in the set, 'line' an int of 2 bits. (r2)
    two_or_four = format(int(line[2]), '01b')  # make element 2 in the set, 'line' an int of 1 bits. (sh)

    f.write(str('011') + str(r1) + str(r2) + str(two_or_four) + '\n')


elif (line[0:3] == "sll"):
    line = line.replace("sll", "")
    line = line.split(",")

    # defaults to register[0] or A
    r1 = 0
    r2 = 0

    # Since we use letters in assembly code
    if (line[0] == "B"):
        r1 = 1
    elif (line[0] == "hi"):
        r1 = 2
    elif (line[0] == "lo"):
        r1 = 3

    if (line[1] == "B"):
        r2 = 1
    elif (line[1] == "hi"):
        r2 = 2
    elif (line[1] == "lo"):
        r2 = 3
    r1 = format(r1, '02b')  # make element 0 in the set, 'line' an int of 2 bits. (r1)
    r2 = format(r2, '02b')  # make element 1 in the set, 'line' an int of 2 bits. (r2)
    two_or_four = format(int(line[2]), '01b')  # make element 2 in the set, 'line' an int of 1 bits.
```

```
    f.write(str('010') + str(r1) + str(r2) + str(two_or_four) + '\n')


# SBU
elif (line[0:3] == "sbu"):
    line = line.replace(")", "")  # remove the ) param entirely.
    line = line.replace("(", ",")  # replace ( left parem with comma
    line = line.replace("sbu", "")
    line = line.split(
        ",")  # split the 1 string 'line' into a string array of many strings, broken at the comma.

    # defaults to register[0] or A
    r1 = 0
    r2 = 0

    # Since we use letters in assembly code
    if (line[0] == "B"):
        r1 = 1
    elif (line[0] == "hi"):
        r1 = 2
    elif (line[0] == "lo"):
        r1 = 3

    if (line[2] == "A"):
        r2 = 1

    r1 = format(r1, '02b')  # make element 0 in the set, 'line' an int of 2 bits. (r1)
    r2 = format(r2, '01b')  # make element 2 in the set, 'line' an int of 1 bits. (r2)
    imm = format(int(line[1]), '02b')

    f.write(str('111') + str(r1) + str(imm) + str(r2) + '\n')

 # LBU
elif (line[0:3] == "lbu"):
    line = line.replace(")", "")  # remove the ) paran entirely.
    line = line.replace("(", ",")  # replace ( left paren with comma
    line = line.replace("lbu", "")
    line = line.split(
        ",")  # split the 1 string 'line' into a string array of many strings, broken at the comma.

    # defaults to register[0] or A
    r1 = 0
    r2 = 0

    # Since we use letters in assembly code
    if (line[0] == "B"):
        r1 = 1
    elif (line[0] == "hi"):
```

```
        r1 = 2
    elif (line[0] == "lo"):
        r1 = 3

    if (line[2] == "A"):
        r2 = 1

    r1 = format(r1, '02b')  # make element 0 in the set, 'line' an int of 2 bits. (r1)
    r2 = format(r2, '01b')  # make element 2 in the set, 'line' an int of 1 bits. (r2)
    imm = format(int(line[1]), '02b')
    f.write(str('110') + str(r1) + str(imm) + str(r2) + '\n')

# = = = = XOR = = = = = = = = =
elif (line[0:3] == "xor"):
    line = line.replace("xor", "")
    line = line.split(",")

    # defaults to register[0] or A
    r1 = 0
    r2 = 0

    # Since we use letters in assembly code
    if (line[0] == "B"):
        r1 = 1
    elif (line[0] == "hi"):
        r1 = 2
    elif (line[0] == "lo"):
        r1 = 3

    if (line[1] == "B"):
        r2 = 1
    elif (line[1] == "hi"):
        r2 = 2
    elif (line[1] == "lo"):
        r2 = 3

    r1 = format(r1, '02b')  # make element 1 in the set, 'line' an int of 2 bits. (r1)
    r2 = format(r2, '02b')  # make element 2 in the set, 'line' an int of 2 bits. (r2)

    f.write(str('101') + str(r1) + str(r2) + str('0') + '\n')


    # = = = = BNE = = = = = = = = = =
elif (line[0:3] == "bne"):
    line = line.replace("bne", "")
    line = line.split(",")

    # defaults to register[0] or A
    r1 = 0
```

```
            if (line[0] == "B"):
                r1 = 1
            elif (line[0] == "hi"):
                r1 = 2
            elif (line[0] == "lo"):
                r1 = 3

            r1 = format(r1, '02b')
            YY = format(int(line[1]), '02b')
            Z = format(int(line[2]), '01b')

            f.write(str('001') + str(r1) + str(YY) + str(Z) + '\n')


    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 by 4 every cycle,


    # We SHALL start the simulation!
    sim(program)


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

Assembly Code:

```
# Instruction M[0]:
# Hardwired start of hash-and-pattern-match loop
# Clear reg B and initialize to project assigned value
sll $B, $B, 1
sll $B, $B, 1
addi $B, 6
sll $B, $B, 1
addi $B, 6

# Increment A by 1 (starting at 0 + 1 = 1)
addi $A, 1

# Multiply-and-fold 5x
# operands begin in regs A and B but are then processed in
# regs hi and lo with B participating in multiplications
mult $A, $B
xor $hi, $lo

mult $hi, $B
xor $hi, $lo

mult $hi, $B
xor $hi, $lo

mult $hi, $B
xor $hi, $lo

mult $hi, $B
xor $hi, $lo

# Final fold to reduce 8-bit C to 4 bits
sll $lo, $hi, 1
srl $lo, $lo, 1
srl $hi, $hi, 1
xor $hi, $lo

# Final fold to reduce 4 bits to 2 bits
sll $lo, $hi, 0
srl $lo, $lo, 0
srl $hi, $hi, 0
xor $hi, $lo

# Store results of hash (hi) to memory location
# based on current value of A (loop iteration)
sbu $hi, 3(A)

# Pattern-matching section:
# load current current match count for 00 and compare 00 to hi
# skip incrementation of match count if no match (PC+3)
lbu $B, 0(0)
bne $hi, 0, 1
addi $B, 1
sbu $B, 0(0)

# load current current match count for 01 and compare 01 to hi
lbu $B, 1(0)
bne $hi, 1, 1
```

```
addi $B, 1
sbu $B, 1(0)

# load current current match count for 10 and compare 10 to hi
lbu $B, 2(0)
bne $hi, 2, 1
addi $B, 1
sbu $B, 2(0)

# load current current match count for 11 and compare 11 to hi
lbu $B, 3(0)
bne $hi, 3, 1
addi $B, 1
sbu $B, 3(0)

# return to Instruction M[0] unless max value of A reached
bne $A, 0, 0
```