**Project 3**

By: Richard Remigoso, Mijiddorj (Mike) Enkhbaatar, Daniella Rodriguez

1. **Group Activity Log**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date / Location** | **Activity** | **Progress** | **Members** |
| 3/19/2019  4:00 pm - 4:45 pm @ UIC Library | In person Group meeting | * First meetup * Started GitHub   + Explained the layout of the repository * Listed possible instructions to do * Chose which/whose previous project to build on | Richard  Mike  Daniella |
| 3/22/2019  4:00 pm - 5:45 pm @ UIC Library | In person Group meeting | * Started encoding instructions | Richard  Mike  Daniella |
| 3/25/2019  12:00pm- 3:00pm  @UIC Library | In person Group meeting | * Brainstormed the 8-bit instruction architecture | Richard  Mike  Daniella |
| 3/27/2019  2:00pm- 4:20pm  @UIC Library | In person Group meeting | * Redid/fixed the 8-bit instruction architecture * Restructured and optimized the PRPG program to use only 4 registers | Richard  Mike  Daniella |
| 3/29/2019  2:00pm- 5:20pm  @UIC Library | In person Group meeting | * Worked on hardware and python * Designated each group members with tasks to do   + Architecture design & python simulator: Richard   + Hardware design:   Mike, Daniella | Richard  Mike  Daniella |
| 4/4/2019  6:00pm - 10:00pm  @UIC Library | In person Group meeting | * Project compilation * Final touch | Richard  Mike  Daniella |

1. **Individual Activity Log**

|  |  |  |  |
| --- | --- | --- | --- |
| **Time/ Location** | **Activity** | **Achieved/ To-Do** | **Member(s)** |
| 3/28/2019  10 AM – 2 PM  @ UIC SPH 11th floor | Individual approach for project | Finished my register recycling approach, redesigned referenced prpg.asm to use only 4 registers, created a new ISA using only 4 registers | Richard |
| 3/31/2019  10 AM – 10 PM  @ Apartment | Adding instructions to python and redesigned PRPG assembly file | Adapted our own ISA syntax using the PRPG assembly file, finished and debugged the python simulator for part A and B | Richard |
| 4/1/2019  6 PM – 10 PM  @ UIC library 1st floor | Debugging | Debugged python simulator, performed all seed inputs and successfully displayed their correct outputs | Richard |

**Part A) ISA intro**

1. **Introduction**. This should include the name of the architecture, overall philosophy, specific goals strived for and achieved.

Our architecture is name is Eggnogx8. The overall philosophy of this architecture is register recycling. In other words, the PRPG program only uses recycles 4 registers to perform the necessary operations. The goal for this project is to maximize the utility of 8-bit codes to create the software and hardware design of a PRPG program, which was achieved in this project.

2. **Instruction list / table**. Give all the instructions, their formats, opcodes, and an example.

4 registers:

|  |  |  |  |
| --- | --- | --- | --- |
| 00 | 01 | 10 | 11 |
| $0 | $1 | $2 | $3 |

Special registers (*rx)*:

|  |  |
| --- | --- |
| 0 | 1 |
| *uno* | *dos* |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Instr.** | **OP**  **code** | **Details** | | | | | **Example** |
| *Syntax* | *Machine code* | *Supported registers* | *Supported imm. range* | *General behavior* |
| refresh  (0) | 0000 | refresh rt imm. | 0000 ttii | [0, 1, 2, 3] | [-2, 1] | Initializes register rt into the given immediate.  rt = imm. | **refresh** $0, 0 |
|
| addi  (1) | 0001 | addi rt imm. | 0001 ttii | [0, 1, 2, 3] | [-2, 1] | Adds register rt with the given immediate and stores result to rt.  rt = rt + imm. | **addi** $0, 1 |
| addu  (2) | 0010 | addu rt, rs | 0010 ttss | [0, 1, 2, 3] | X | Adds registers rt and rs together and stores the unsigned result to rt.  rt = rt + rs | refresh $0, 0  refresh $1, 1  **addu** $1, $0 |
| store  (3) | 0011 | store rt, rs | 0011 ttss | [0, 1, 2, 3] | X | Stores the value of register rt into the memory address.  M[rs] = rt | refresh $2, 0  **store** $2, $2 |
| mult  (4) | 0100 | mult rt, rs | 0100 ttss | [0, 1, 2, 3] | X | Multiplies rt and rs into  a 16 bit number then stores the 8 MSB into $1 and the 8 LSB into $0.  temp16 = rt x rs  $0 = temp16 & 0x00FF  $1 = temp16 & 0xFF00 | **mult** $1, $1 |
| splice  (5) | 0101 | splice $1, $0 | 0101 0100 | [0, 1] | X | Drops the 4 LSB of $1 and 4 MSB of $0 then merge into $1.  $0 = $0 & 0x0F  $1 = $1 & 0xF0  $1 = $1 + $0 | **splice** $1, $0 |
| jne  (6) | 0110 | jne rt, imm., rx | 0110 ttix | [0, 1, 2, 3,]  Special registers:  *uno*  *dos* | [0, 1] | If register rt is not equal to the given imm., jump to the value of register *rx.*  If rt != imm.,  pc = rx  else  pc++ | addi $2, -2  **jne** $2, 0, 1 |
| mark  (7) | 0111 | mark rx | 0111 XXXx  \*X - don’t cares | Special registers:  *uno*  *dos* | X | Stores the next pc value into register rx.  rx = pc + 1 | **mark** 1 |
| slti  (8) | 1000 | slti rt, imm. | 1000 ttii | [0, 1, 2, 3] | [0, 3] | If register rt value is less than the given immediate, set value of register 4 to 1, else set to 0.  rt value < 0?,  $3 = 1:0 | **slti** $1, 0 |
| beq  (9) | 1001 | beq rt, imm. | 1001 ttii | [0, 1, 2, 3] | [-2, 1] | If register rt value is equal to 0, update pc with given immediate.  if rt == 0,  pc = pc + imm.  else  pc++ | **slti** $1, 0  **beq** $1, 1 |
| sll  (A) | 1010 | sll rt, imm. | 1010 ttii | [0, 1, 2, 3] | [0, 3] | Logic shift register rt to the left by the given immediate and store it back to rt.  rt = rt << imm. | **sll** $1, 1 |
| srl  (B) | 1011 | srl rt, rs | 1011 ttss | [0, 1, 2, 3] | X | Logic shift register rt to the right by the value of register rs and store it back to rt.  rt = rt >> rs | **srl** $3, $1 |
| load  (C) | 1100 | load rt, rs | 1100 ttss | [0, 1, 2, 3] | X | Loads the value of memory address into the register rt.  rt = M[rs] | refresh $2, 0  **load** $2, $2 |

3. **Register design**. How many registers are supported? Is there anything special about the registers?

There are 4 general registers but we have special registers *uno* and *dos*. The special registers are used store the PC values of an instruction line in the program. They can only be accessed by instructions mark and jne. There are total of 6 registers

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

Beq is supported in this design. Like MIPS, an immediate number is added to the PC whenever the branch is taken, otherwise regular PC operation considered if not taken. The maximum branch distance supported is -2 to go back and +1 to go forward.

5. **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?

Instructions load and store are used to access data memory. The range of addresses that can be accessed with the design is from M[0] → M[63], allowing 64 bytes of data memory.

6. What would you have done differently if you had 1 more bit for instructions?

If we had 1 more bit for instructions, then the amount of dedicated bits for immediate numbers will be larger, which will support a bigger range of immediates.

7. How about 1 fewer bit?

If we had 1 less bit for instructions, then the amount of instructions may have to be reduced, as well as the op-code size, and the design of the instruction syntaxes will have to change drastically.

8. What are the most significant advantages of your ISA (with regard to the PRPG program, hardware implementation, ease of programming, etc)? 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 advantage of our ISA is that it has a great possibility of being used for other programs, not just for PRPG. This is because majority of the instructions were not made only for PRPG. The main limitations of this architecture is the amount of immediate number supported which is only from [-2, 1] for signed operations and [0, 3] for unsigned operations. The main compromise is the drastic reduction for immediate number support, in order to accommodate the op-codes of the instructions. However, because of the utilization of 4-bit op-codes, it is much cleaner to program.

9. What have you done towards the goals of low DIC and HW simplification? What could have been done differently to better optimize for each of the two goals, if to start over?

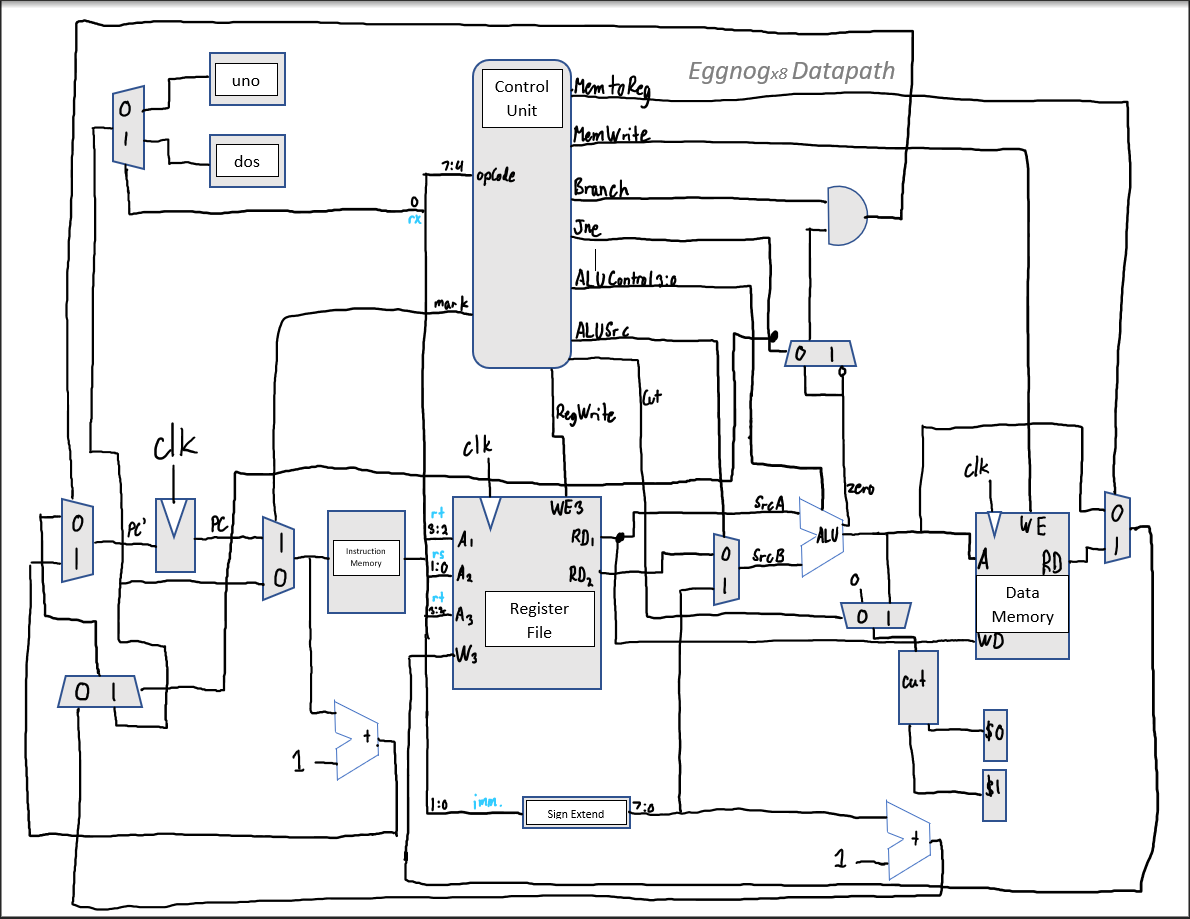
To reduce the DIC, we used a algorithm based on the fact that square root of any number n can be calculated by adding odd numbers exactly n times. For the hardware simplification, we gave more utility to our ALU design to simplify our datapath. If we were to start over, we would have better instructions that have more utility to lower the DIC this would result in more complex datapath and ALU design.

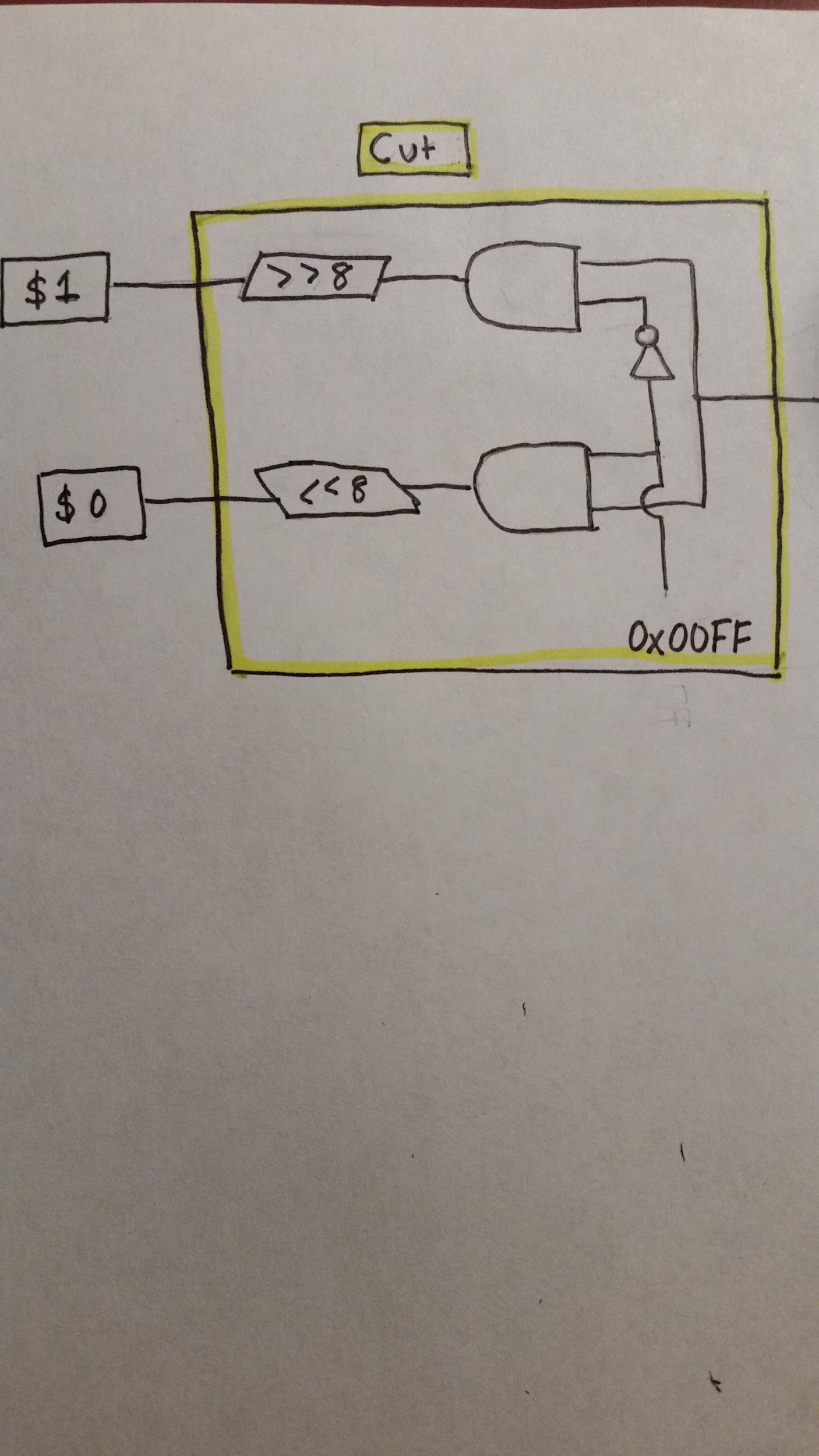
10. If you are given a chance to restart this project afresh with 3 weeks’ time, how would your group have done it differently?

We would have brainstormed more on the amount of bits dedicated for the op-codes of our instruction (most preferably to use 3). We would probably have approached our hardware design differently too since our current design can be simplified more.

**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.

Decoder Truth Table/Control Logic Design

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Instruction | Opcode | RegWrite | Jne | ALUSrc | Branch | MemWrite | MemReg | Cut | Mark | ALUOp |
| refresh | 0000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | XX |
| addi | 0001 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 00 |
| addu | 0010 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0010 |
| store | 0011 | 0 | 0 | 0 | 0 | 1 | x | 0 | 0 | XX |
| mult | 0100 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0111 |
| splice | 0101 | 0 | 0 | x | 0 | 0 | 0 | 0 | 0 | 0101 |
| jne | 0110 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | XX |
| mark | 0111 | 0 | 0 | x | 0 | 0 | 0 | 0 | 1 | XX |
| slti | 1000 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1011 |
| beq | 1001 | 0 | 0 | 0 | 1 | 0 | x | 0 | 0 | XX |
| sll | 1010 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0100 |
| srl | 1011 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0110 |
| load | 1100 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | XX |

Special Instructions:

Mark: It allows for rx (the special registers) to be written into with PC+1.

Refresh: It automatically puts a value into rt.

Mult: It multiplies what’s in rt and rs together to produce a sixteen bit number.

Splice: It takes the lsb and msb of $0 and $1 and puts them together in one register.

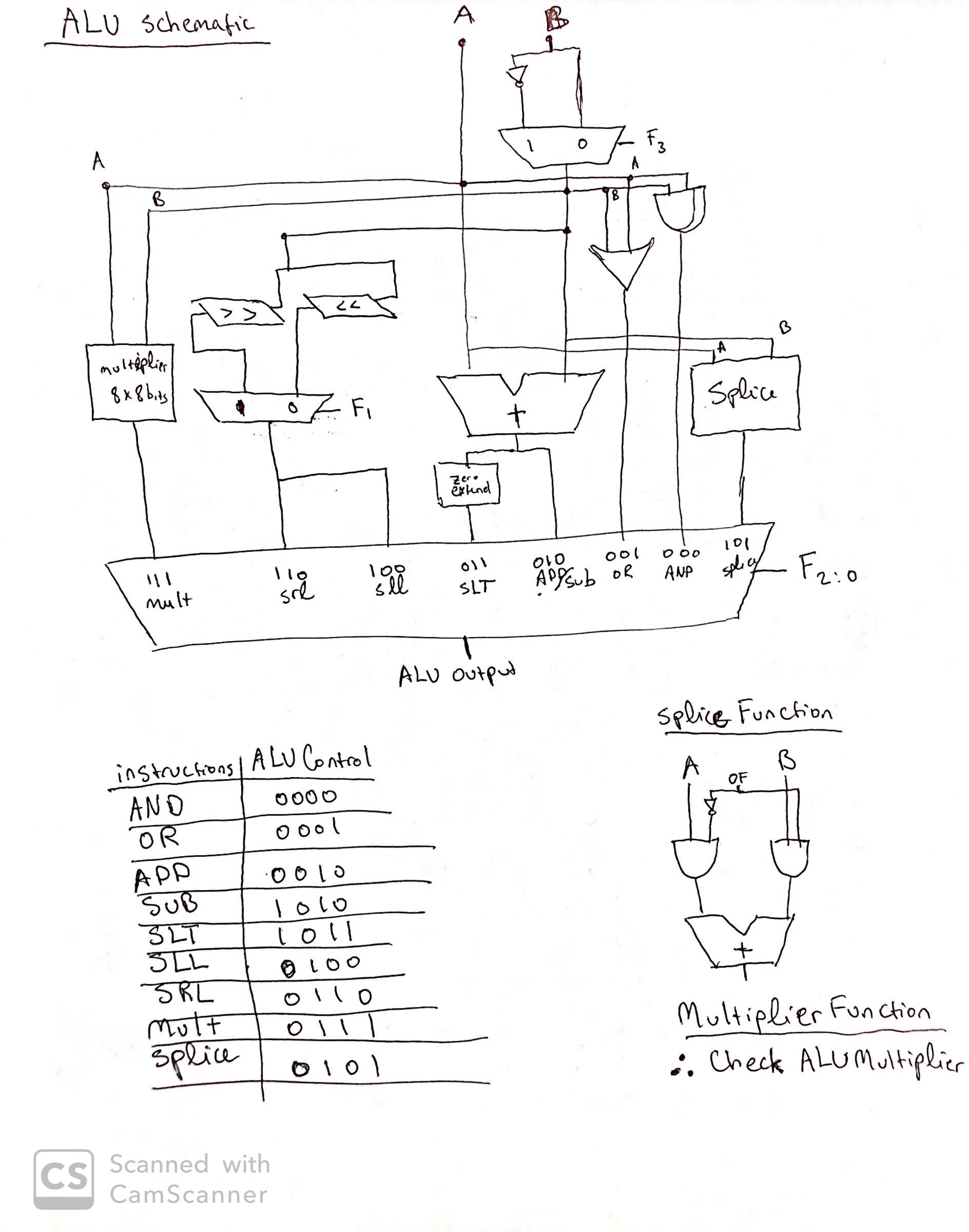
New Control Signals:

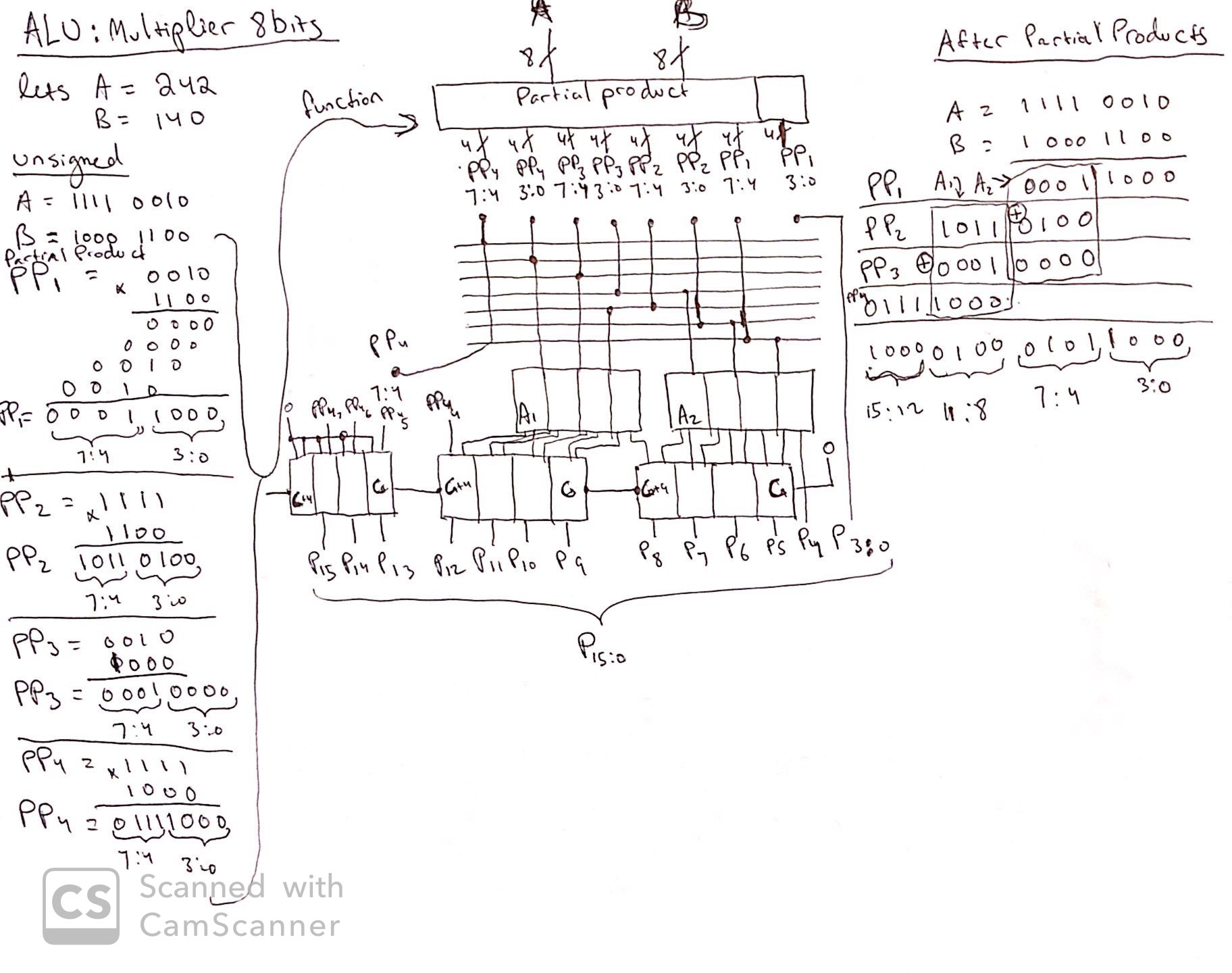
Jne: If this is high, then the value in rx can be used in place of PC+1. Else, PC is incremented normally.

Mark: If this is high, then rx can be written into. Else, rx does not change.

Cut: If this is high, then it takes the ALU result and divides it into $0 and $1. Else, the ALU result is not divided.

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

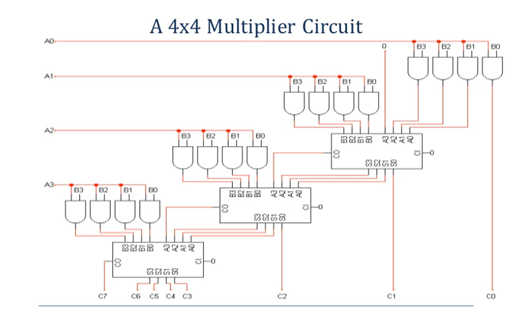


The goal of A1 is to calculate the final product bits from 8 to 11.

* A1 adds the partials of last half of PP2 with last half of PP3 and PP4.
* As result, A1 = 0010

The goal of A2 is to calculate the final products bits from 4 to 7.

* A2 adds the partials of last half of PP1 with first half of PP2 and PP3.
* As Result, A2 = 1000.



The 3 Rectangles under the A1 and A2, calculates the carry in order to get the correct values.

However, simpler way to describe the 8x8 Multiplier would be the circuit above. The only difference would is that A’s range will from A0 - A7, B’s range would B0 - B7 and there will lot more AND gates and adders.

**Part C) Software Package: PRPG Program + Python (20pts)**

1. S0 = 251

Assembly code

# Seed = 251

########################################## Part A

refresh $0, 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1 # initial seed 251

refresh $1, 0

addu $1, $0 # tempS = seed

refresh $2, 1 # address counter

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 8, point initial memory address to M[8]

refresh $3, 0

store $2, $3 # store address counter to M[0]

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # numS = 16

addi $3, 1

store $2, $3 # store numS to M[1]

refresh $0, 0

load $3, $0 # load address counter into $3

loop: mark 0 # store the next pc value into register *uno*

store $1, $3 # store value $1 into mem address

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 4

load $3, $2 # load sumS into $3

addu $3, $1 # sumS = sumS + seed

store $3, $2 # store sumS into M[4]

mult $1, $1 # sq = $1 x $1; $1 = 8 most significant bits; $0 = 8 least significant bits

splice $1, $0 # drops the 4 LSB of $1 and 4 MSB of $0, merge into $1

refresh $0, 0

addi $0, 1

load $2, $0 # load numS into $2

addi $2, -1 # numS--

store $2, $0 # store numS into M[1]

refresh $0, 0

load $3, $0 # load address counter into $3

addi $3, 1 # increment address counter by 1

store $3, $0 # store address counter into M[0]

jne $2, 0, 0 # if $2 != 0, jump to register *uno* value → loop

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 4

load $3, $2 # load sumS into $3

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 4

srl $3, $1 # logic shift to the right by 4, Average seed

refresh $0, 0

load $1, $0 # load address counter into $1

store $3, $1 # store Average seed into M[24]

#################################### Part B

refresh $0, 0 # point to first address

refresh $1, 0 # sum = 0

store $1, $0 # store sum into M[0]

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # numS = 16

addi $0, 1 # point to second address

store $2, $0 # store numS into M[1]

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 32

addi $0, 1 # point to third address

store $1, $0 # store memory address counter of seed Hamming weights into M[2]

refresh $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1 # point to 8th address

loop1: mark 0 # store the next pc value into register *uno*

load $1, $0 # load seed in current mem address

refresh $2, 0 # i = 0

loop2: mark 1 # store the next pc value into register *dos*

slti $1, 0 # $1 value < 0?, $3 = 1:0

beq $3, 1 # if $3 == 0 branch to skip

addi $2, 1 # i++ when $1 is negative; Hamming weight of a seed

skip: sll $1, 1 # logic shift to the left once for every loop

jne $1, 0, 1 # if $1 != 0 jump to register *dos* value → loop2

refresh $3, 1

addi $3, 1

load $1, $3 # load mem. addr. ctr. of seed Hamming weights into $1

store $2, $1 # store Hamming weight into M[$1]

addi $1, 1 # increment mem. addr. ctr. by 1

store $1, $3 # store mem. addr. ctr. into M[2]

refresh $1, 0 # point to the 1st memory address

load $3, $1 # load sum into $3

addu $3, $2 # sum of 1’s in all 16 seeds, Total Hamming weight

store $3, $1 # store sum into M[0]

refresh $1, 1 # point to the 2nd memory address

load $3, $1 # load numS into $3

addi $3, -1 # decrement numS for each iteration by 1

store $3, $1 # store numS into M[1]

addi $0, 1 # increment address counter by 1

jne $3, 0, 0 # if $3 != 0 jump to register *uno* value → loop1

refresh $0, 0

load $2, $0 # load sum into $2

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 4

srl $2, $1 # logic shift to the right by 4, Average Hamming weight

refresh $0, 1

addi $0, 1

load $1, $0 # load mem. addr. ctr. of seed Hamming weights into $1

store $2, $1 # store AHW into address M[48]

Machine Code

############################################################## PART A

0x01

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11 # initial seed 251

0x04

0x24 # tempS = seed

0x09 # address counter

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # $2 = 8, point initial memory address to M[8]

0x0C

0x3B # store address counter to M[0]

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # numS = 16

0x1D

0x3B # store numS to M[1]

0x00

0xCC # load address counter into $3

0x70 # store the next pc value into register uno

0x37 # store value $1 into mem address

0x09

0x19

0x19

0x19 # $2 = 4

0xCE # load sumS into $3

0x2D # sumS = sumS + seed

0x3E # store sumS into M[4]

0x45 # sq = $1 x $1; $1 = 8 most significant bits; $0 = 8 least significant bits

0x54 # drops the 4 LSB of $1 and 4 MSB of $0, merge into $1

0x00

0x11

0xC8 # load numS into $2

0x1B # numS--

0x38 # store numS into M[1]

0x00

0xCC # load address counter into $3

0x1D # increment address counter by 1

0x3C # store address counter into M[0]

0x68 # if $2 != 0, jump to register uno value → loop

0x09

0x19

0x19

0x19 # $2 = 4

0xCE # load sumS into $3

0x05

0x15

0x15

0x15 # $1 = 4

0xBD # logic shift to the right by 4, Average seed

0x00

0xC4 # load address counter into $1

0x3D # store Average seed into M[24]

############################################################## PART B

0x00 # point to first address

0x04 # sum = 0

0x38 # store sum into M[0]

0x09

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # numS = 16

0x11 # point to second address

0x38 # store numS into M[1]

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # $1 = 32

0x11 # point to third address

0x38 # store memory address counter of seed Hamming weights into M[2]

0x01

0x11

0x11

0x11

0x11

0x11

0x11

0x11 # point to 8th address

0x70 # store the next pc value into register uno

0xC4 # load seed in current mem address

0x08 # i = 0

0x71 # store the next pc value into register dos

0x84 # $1 value < 0?, $3 = 1:0

0x9D # if $3 == 0 branch to skip

0x19 # i++ when $1 is negative; Hamming weight of a seed

0xA5 # logic shift to the left once for every loop

0x65 # if $1 != 0 jump to register dos value → loop2

0x0D

0x1D

0xC7 # load mem. addr. ctr. of seed Hamming weights into $1

0x39 # store Hamming weight into M[$1]

0x15 # increment mem. addr. ctr. by 1

0x37 # store mem. addr. ctr. into M[2]

0x04 # point to the 1st memory address

0xCD # load sum into $3

0x2E # sum of 1’s in all 16 seeds, Total Hamming weight

0x3D # store sum into M[0]

0x05 # point to the 2nd memory address

0xCD # load numS into $3

0x1F # decrement numS for each iteration by 1

0x3D # store numS into M[1]

0x11 # increment address counter by 1

0x6C # if $3 != 0 jump to register uno value → loop1

0x00

0xC8 # load sum into $2

0x05

0x15

0x15

0x15 # $1 = 4

0xB9 # logic shift to the right by 4, Average Hamming weight

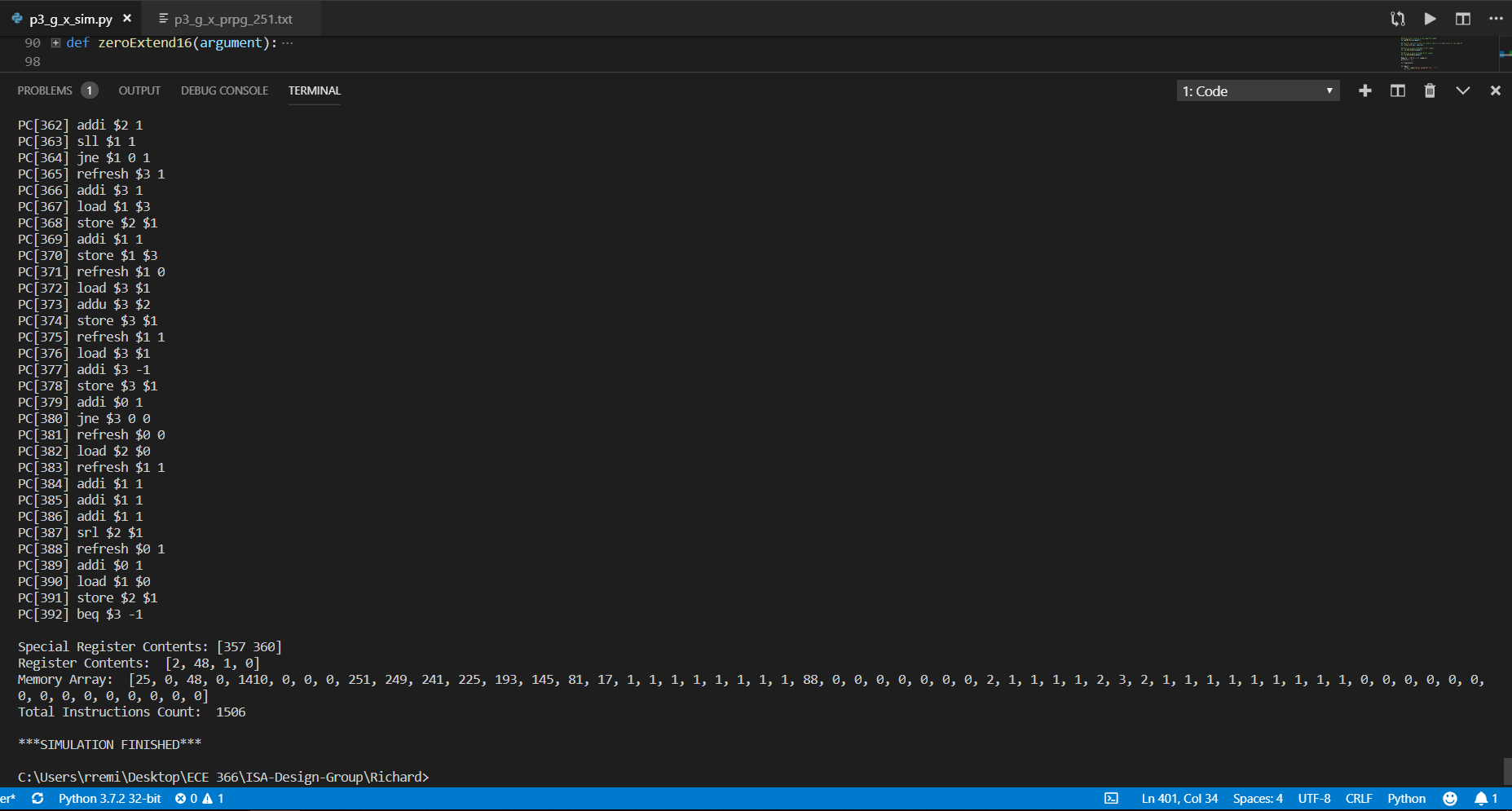
0x01

0x11

0xC4 # load mem. addr. ctr. of seed Hamming weights into $1

0x39 # store AHW into address M[48]

0x9F #STOP



2. S0 = 118

Assembly Code

# Seed = 118

########################################## Part A

refresh $0, 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1 # initial seed 118

refresh $1, 0

addu $1, $0 # tempS = seed

refresh $2, 1 # address counter

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 8, point initial memory address to M[8]

refresh $3, 0

store $2, $3 # store address counter to M[0]

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # numS = 16

addi $3, 1

store $2, $3 # store numS to M[1]

refresh $0, 0

load $3, $0 # load address counter into $3

loop: mark 0 # store the next pc value into register *uno*

store $1, $3 # store value $1 into mem address

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 4

load $3, $2 # load sumS into $3

addu $3, $1 # sumS = sumS + seed

store $3, $2 # store sumS into M[4]

mult $1, $1 # sq = $1 x $1; $1 = 8 most significant bits; $0 = 8 least significant bits

splice $1, $0 # drops the 4 LSB of $1 and 4 MSB of $0, merge into $1

refresh $0, 0

addi $0, 1

load $2, $0 # load numS into $2

addi $2, -1 # numS--

store $2, $0 # store numS into M[1]

refresh $0, 0

load $3, $0 # load address counter into $3

addi $3, 1 # increment address counter by 1

store $3, $0 # store address counter into M[0]

jne $2, 0, 0 # if $2 != 0, jump to register *uno* value → loop

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 4

load $3, $2 # load sumS into $3

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 4

srl $3, $1 # logic shift to the right by 4, Average seed

refresh $0, 0

load $1, $0 # load address counter into $1

store $3, $1 # store Average seed into M[24]

#################################### Part B

refresh $0, 0 # point to first address

refresh $1, 0 # sum = 0

store $1, $0 # store sum into M[0]

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # numS = 16

addi $0, 1 # point to second address

store $2, $0 # store numS into M[1]

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 32

addi $0, 1 # point to third address

store $1, $0 # store memory address counter of seed Hamming weights into M[2]

refresh $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1 # point to 8th address

loop1: mark 0 # store the next pc value into register *uno*

load $1, $0 # load seed in current mem address

refresh $2, 0 # i = 0

loop2: mark 1 # store the next pc value into register *dos*

slti $1, 0 # $1 value < 0?, $3 = 1:0

beq $3, 1 # if $3 == 0 branch to skip

addi $2, 1 # i++ when $1 is negative; Hamming weight of a seed

skip: sll $1, 1 # logic shift to the left once for every loop

jne $1, 0, 1 # if $1 != 0 jump to register *dos* value → loop2

refresh $3, 1

addi $3, 1

load $1, $3 # load mem. addr. ctr. of seed Hamming weights into $1

store $2, $1 # store Hamming weight into M[$1]

addi $1, 1 # increment mem. addr. ctr. by 1

store $1, $3 # store mem. addr. ctr. into M[2]

refresh $1, 0 # point to the 1st memory address

load $3, $1 # load sum into $3

addu $3, $2 # sum of 1’s in all 16 seeds, Total Hamming weight

store $3, $1 # store sum into M[0]

refresh $1, 1 # point to the 2nd memory address

load $3, $1 # load numS into $3

addi $3, -1 # decrement numS for each iteration by 1

store $3, $1 # store numS into M[1]

addi $0, 1 # increment address counter by 1

jne $3, 0, 0 # if $3 != 0 jump to register *uno* value → loop1

refresh $0, 0

load $2, $0 # load sum into $2

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 4

srl $2, $1 # logic shift to the right by 4, Average Hamming weight

refresh $0, 1

addi $0, 1

load $1, $0 # load mem. addr. ctr. of seed Hamming weights into $1

store $2, $1 # store AHW into address M[48]

Machine code

############################################################## PART A

0x01

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11 # initial seed 118

0x04

0x24 # tempS = seed

0x09 # address counter

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # $2 = 8, point initial memory address to M[8]

0x0C

0x3B # store address counter to M[0]

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # numS = 16

0x1D

0x3B # store numS to M[1]

0x00

0xCC # load address counter into $3

0x70 # store the next pc value into register uno

0x37 # store value $1 into mem address

0x09

0x19

0x19

0x19 # $2 = 4

0xCE # load sumS into $3

0x2D # sumS = sumS + seed

0x3E # store sumS into M[4]

0x45 # sq = $1 x $1; $1 = 8 most significant bits; $0 = 8 least significant bits

0x54 # drops the 4 LSB of $1 and 4 MSB of $0, merge into $1

0x00

0x11

0xC8 # load numS into $2

0x1B # numS--

0x38 # store numS into M[1]

0x00

0xCC # load address counter into $3

0x1D # increment address counter by 1

0x3C # store address counter into M[0]

0x68 # if $2 != 0, jump to register uno value → loop

0x09

0x19

0x19

0x19 # $2 = 4

0xCE # load sumS into $3

0x05

0x15

0x15

0x15 # $1 = 4

0xBD # logic shift to the right by 4, Average seed

0x00

0xC4 # load address counter into $1

0x3D # store Average seed into M[24]

############################################################## PART B

0x00 # point to first address

0x04 # sum = 0

0x38 # store sum into M[0]

0x09

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # numS = 16

0x11 # point to second address

0x38 # store numS into M[1]

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # $1 = 32

0x11 # point to third address

0x38 # store memory address counter of seed Hamming weights into M[2]

0x01

0x11

0x11

0x11

0x11

0x11

0x11

0x11 # point to 8th address

0x70 # store the next pc value into register uno

0xC4 # load seed in current mem address

0x08 # i = 0

0x71 # store the next pc value into register dos

0x84 # $1 value < 0?, $3 = 1:0

0x9D # if $3 == 0 branch to skip

0x19 # i++ when $1 is negative; Hamming weight of a seed

0xA5 # logic shift to the left once for every loop

0x65 # if $1 != 0 jump to register dos value → loop2

0x0D

0x1D

0xC7 # load mem. addr. ctr. of seed Hamming weights into $1

0x39 # store Hamming weight into M[$1]

0x15 # increment mem. addr. ctr. by 1

0x37 # store mem. addr. ctr. into M[2]

0x04 # point to the 1st memory address

0xCD # load sum into $3

0x2E # sum of 1’s in all 16 seeds, Total Hamming weight

0x3D # store sum into M[0]

0x05 # point to the 2nd memory address

0xCD # load numS into $3

0x1F # decrement numS for each iteration by 1

0x3D # store numS into M[1]

0x11 # increment address counter by 1

0x6C # if $3 != 0 jump to register uno value → loop1

0x00

0xC8 # load sum into $2

0x05

0x15

0x15

0x15 # $1 = 4

0xB9 # logic shift to the right by 4, Average Hamming weight

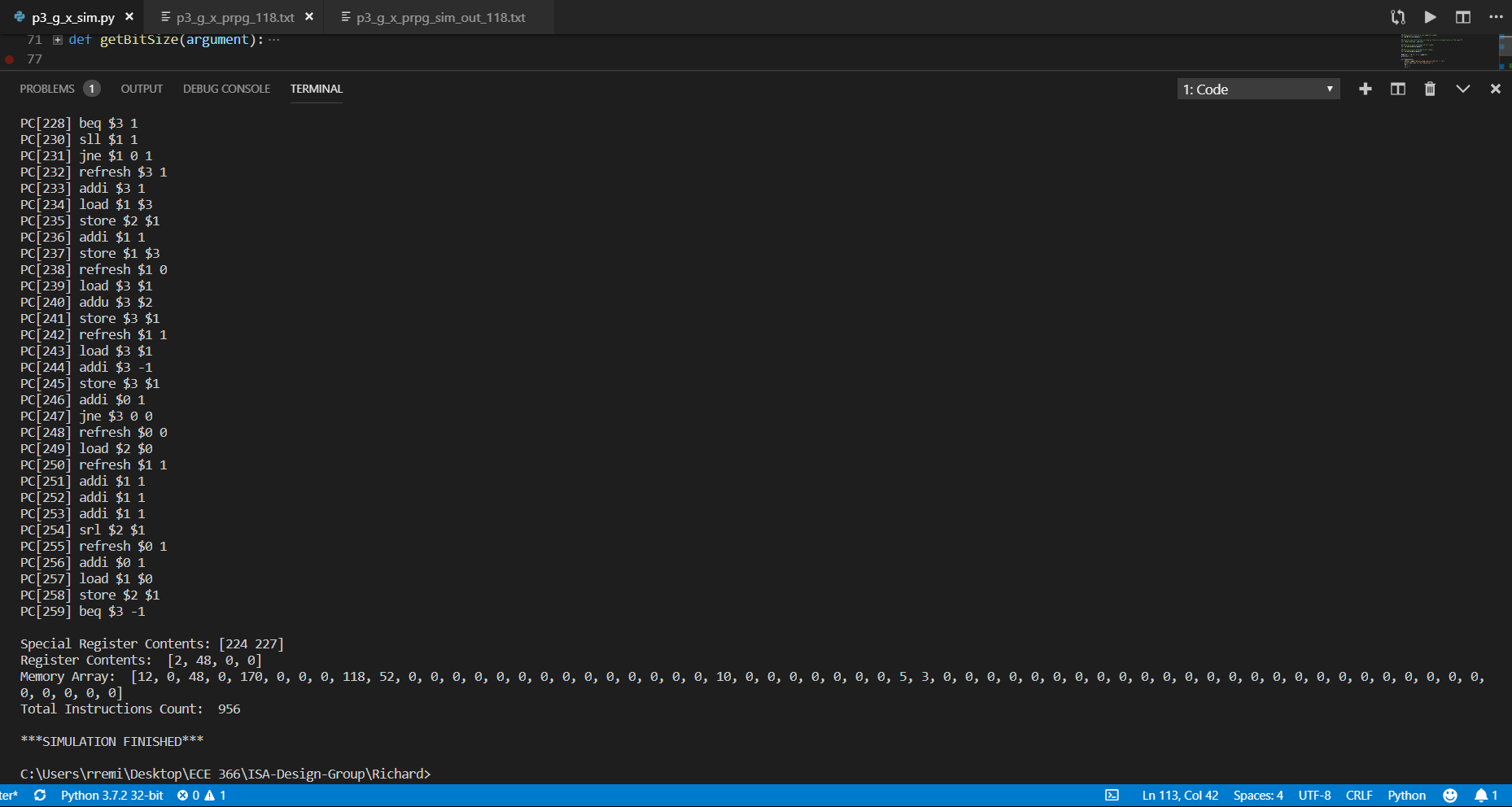
0x01

0x11

0xC4 # load mem. addr. ctr. of seed Hamming weights into $1

0x39 # store AHW into address M[48]

0x9F #STOP



3. S0 = 79

Assembly code

# Seed = 79

########################################## Part A

refresh $0, 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1

addi $0 1 # initial seed 79

refresh $1, 0

addu $1, $0 # tempS = seed

refresh $2, 1 # address counter

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 8, point initial memory address to M[8]

refresh $3, 0

store $2, $3 # store address counter to M[0]

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # numS = 16

addi $3, 1

store $2, $3 # store numS to M[1]

refresh $0, 0

load $3, $0 # load address counter into $3

loop: mark 0 # store the next pc value into register *uno*

store $1, $3 # store value $1 into mem address

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 4

load $3, $2 # load sumS into $3

addu $3, $1 # sumS = sumS + seed

store $3, $2 # store sumS into M[4]

mult $1, $1 # sq = $1 x $1; $1 = 8 most significant bits; $0 = 8 least significant bits

splice $1, $0 # drops the 4 LSB of $1 and 4 MSB of $0, merge into $1

refresh $0, 0

addi $0, 1

load $2, $0 # load numS into $2

addi $2, -1 # numS--

store $2, $0 # store numS into M[1]

refresh $0, 0

load $3, $0 # load address counter into $3

addi $3, 1 # increment address counter by 1

store $3, $0 # store address counter into M[0]

jne $2, 0, 0 # if $2 != 0, jump to register *uno* value → loop

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 4

load $3, $2 # load sumS into $3

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 4

srl $3, $1 # logic shift to the right by 4, Average seed

refresh $0, 0

load $1, $0 # load address counter into $1

store $3, $1 # store Average seed into M[24]

#################################### Part B

refresh $0, 0 # point to first address

refresh $1, 0 # sum = 0

store $1, $0 # store sum into M[0]

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # numS = 16

addi $0, 1 # point to second address

store $2, $0 # store numS into M[1]

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 32

addi $0, 1 # point to third address

store $1, $0 # store memory address counter of seed Hamming weights into M[2]

refresh $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1 # point to 8th address

loop1: mark 0 # store the next pc value into register *uno*

load $1, $0 # load seed in current mem address

refresh $2, 0 # i = 0

loop2: mark 1 # store the next pc value into register *dos*

slti $1, 0 # $1 value < 0?, $3 = 1:0

beq $3, 1 # if $3 == 0 branch to skip

addi $2, 1 # i++ when $1 is negative; Hamming weight of a seed

skip: sll $1, 1 # logic shift to the left once for every loop

jne $1, 0, 1 # if $1 != 0 jump to register *dos* value → loop2

refresh $3, 1

addi $3, 1

load $1, $3 # load mem. addr. ctr. of seed Hamming weights into $1

store $2, $1 # store Hamming weight into M[$1]

addi $1, 1 # increment mem. addr. ctr. by 1

store $1, $3 # store mem. addr. ctr. into M[2]

refresh $1, 0 # point to the 1st memory address

load $3, $1 # load sum into $3

addu $3, $2 # sum of 1’s in all 16 seeds, Total Hamming weight

store $3, $1 # store sum into M[0]

refresh $1, 1 # point to the 2nd memory address

load $3, $1 # load numS into $3

addi $3, -1 # decrement numS for each iteration by 1

store $3, $1 # store numS into M[1]

addi $0, 1 # increment address counter by 1

jne $3, 0, 0 # if $3 != 0 jump to register *uno* value → loop1

refresh $0, 0

load $2, $0 # load sum into $2

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 4

srl $2, $1 # logic shift to the right by 4, Average Hamming weight

refresh $0, 1

addi $0, 1

load $1, $0 # load mem. addr. ctr. of seed Hamming weights into $1

store $2, $1 # store AHW into address M[48]

Machine code

############################################################## PART A

0x01

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11

0x11 # initial seed 79

0x04

0x24 # tempS = seed

0x09 # address counter

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # $2 = 8, point initial memory address to M[8]

0x0C

0x3B # store address counter to M[0]

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # numS = 16

0x1D

0x3B # store numS to M[1]

0x00

0xCC # load address counter into $3

0x70 # store the next pc value into register uno

0x37 # store value $1 into mem address

0x09

0x19

0x19

0x19 # $2 = 4

0xCE # load sumS into $3

0x2D # sumS = sumS + seed

0x3E # store sumS into M[4]

0x45 # sq = $1 x $1; $1 = 8 most significant bits; $0 = 8 least significant bits

0x54 # drops the 4 LSB of $1 and 4 MSB of $0, merge into $1

0x00

0x11

0xC8 # load numS into $2

0x1B # numS--

0x38 # store numS into M[1]

0x00

0xCC # load address counter into $3

0x1D # increment address counter by 1

0x3C # store address counter into M[0]

0x68 # if $2 != 0, jump to register uno value → loop

0x09

0x19

0x19

0x19 # $2 = 4

0xCE # load sumS into $3

0x05

0x15

0x15

0x15 # $1 = 4

0xBD # logic shift to the right by 4, Average seed

0x00

0xC4 # load address counter into $1

0x3D # store Average seed into M[24]

############################################################## PART B

0x00 # point to first address

0x04 # sum = 0

0x38 # store sum into M[0]

0x09

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # numS = 16

0x11 # point to second address

0x38 # store numS into M[1]

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # $1 = 32

0x11 # point to third address

0x38 # store memory address counter of seed Hamming weights into M[2]

0x01

0x11

0x11

0x11

0x11

0x11

0x11

0x11 # point to 8th address

0x70 # store the next pc value into register uno

0xC4 # load seed in current mem address

0x08 # i = 0

0x71 # store the next pc value into register dos

0x84 # $1 value < 0?, $3 = 1:0

0x9D # if $3 == 0 branch to skip

0x19 # i++ when $1 is negative; Hamming weight of a seed

0xA5 # logic shift to the left once for every loop

0x65 # if $1 != 0 jump to register dos value → loop2

0x0D

0x1D

0xC7 # load mem. addr. ctr. of seed Hamming weights into $1

0x39 # store Hamming weight into M[$1]

0x15 # increment mem. addr. ctr. by 1

0x37 # store mem. addr. ctr. into M[2]

0x04 # point to the 1st memory address

0xCD # load sum into $3

0x2E # sum of 1’s in all 16 seeds, Total Hamming weight

0x3D # store sum into M[0]

0x05 # point to the 2nd memory address

0xCD # load numS into $3

0x1F # decrement numS for each iteration by 1

0x3D # store numS into M[1]

0x11 # increment address counter by 1

0x6C # if $3 != 0 jump to register uno value → loop1

0x00

0xC8 # load sum into $2

0x05

0x15

0x15

0x15 # $1 = 4

0xB9 # logic shift to the right by 4, Average Hamming weight

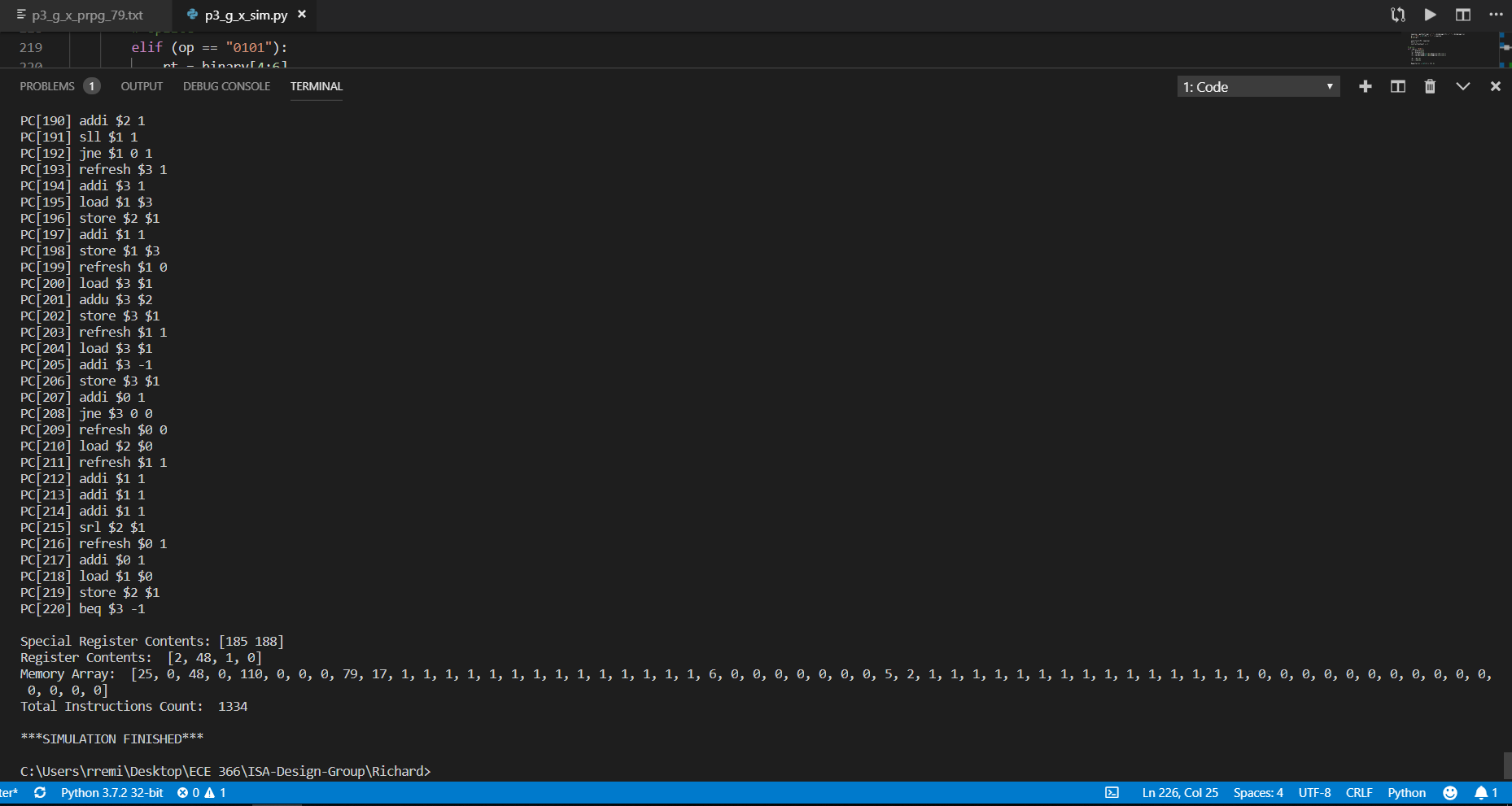
0x01

0x11

0xC4 # load mem. addr. ctr. of seed Hamming weights into $1

0x39 # store AHW into address M[48]

0x9F #STOP



4. S0 = 3

Assembly code

# Seed = 3

########################################## Part A

refresh $0, 1

addi $0, 1

addi $0, 1 # initial seed 3

refresh $1, 0

addu $1, $0 # tempS = seed

refresh $2, 1 # address counter

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 8, point initial memory address to M[8]

refresh $3, 0

store $2, $3 # store address counter to M[0]

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # numS = 16

addi $3, 1

store $2, $3 # store numS to M[1]

refresh $0, 0

load $3, $0 # load address counter into $3

loop: mark 0 # store the next pc value into register *uno*

store $1, $3 # store value $1 into mem address

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 4

load $3, $2 # load sumS into $3

addu $3, $1 # sumS = sumS + seed

store $3, $2 # store sumS into M[4]

mult $1, $1 # sq = $1 x $1; $1 = 8 most significant bits; $0 = 8 least significant bits

splice $1, $0 # drops the 4 LSB of $1 and 4 MSB of $0, merge into $1

refresh $0, 0

addi $0, 1

load $2, $0 # load numS into $2

addi $2, -1 # numS--

store $2, $0 # store numS into M[1]

refresh $0, 0

load $3, $0 # load address counter into $3

addi $3, 1 # increment address counter by 1

store $3, $0 # store address counter into M[0]

jne $2, 0, 0 # if $2 != 0, jump to register *uno* value → loop

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # $2 = 4

load $3, $2 # load sumS into $3

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 4

srl $3, $1 # logic shift to the right by 4, Average seed

refresh $0, 0

load $1, $0 # load address counter into $1

store $3, $1 # store Average seed into M[24]

#################################### Part B

refresh $0, 0 # point to first address

refresh $1, 0 # sum = 0

store $1, $0 # store sum into M[0]

refresh $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1

addi $2, 1 # numS = 16

addi $0, 1 # point to second address

store $2, $0 # store numS into M[1]

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 32

addi $0, 1 # point to third address

store $1, $0 # store memory address counter of seed Hamming weights into M[2]

refresh $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1

addi $0, 1 # point to 8th address

loop1: mark 0 # store the next pc value into register *uno*

load $1, $0 # load seed in current mem address

refresh $2, 0 # i = 0

loop2: mark 1 # store the next pc value into register *dos*

slti $1, 0 # $1 value < 0?, $3 = 1:0

beq $3, 1 # if $3 == 0 branch to skip

addi $2, 1 # i++ when $1 is negative; Hamming weight of a seed

skip: sll $1, 1 # logic shift to the left once for every loop

jne $1, 0, 1 # if $1 != 0 jump to register *dos* value → loop2

refresh $3, 1

addi $3, 1

load $1, $3 # load mem. addr. ctr. of seed Hamming weights into $1

store $2, $1 # store Hamming weight into M[$1]

addi $1, 1 # increment mem. addr. ctr. by 1

store $1, $3 # store mem. addr. ctr. into M[2]

refresh $1, 0 # point to the 1st memory address

load $3, $1 # load sum into $3

addu $3, $2 # sum of 1’s in all 16 seeds, Total Hamming weight

store $3, $1 # store sum into M[0]

refresh $1, 1 # point to the 2nd memory address

load $3, $1 # load numS into $3

addi $3, -1 # decrement numS for each iteration by 1

store $3, $1 # store numS into M[1]

addi $0, 1 # increment address counter by 1

jne $3, 0, 0 # if $3 != 0 jump to register *uno* value → loop1

refresh $0, 0

load $2, $0 # load sum into $2

refresh $1, 1

addi $1, 1

addi $1, 1

addi $1, 1 # $1 = 4

srl $2, $1 # logic shift to the right by 4, Average Hamming weight

refresh $0, 1

addi $0, 1

load $1, $0 # load mem. addr. ctr. of seed Hamming weights into $1

store $2, $1 # store AHW into address M[48]

Machine code

############################################################## PART A

0x01

0x11

0x11 # initial seed 3

0x04

0x24 # tempS = seed

0x09 # address counter

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # $2 = 8, point initial memory address to M[8]

0x0C

0x3B # store address counter to M[0]

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # numS = 16

0x1D

0x3B # store numS to M[1]

0x00

0xCC # load address counter into $3

0x70 # store the next pc value into register uno

0x37 # store value $1 into mem address

0x09

0x19

0x19

0x19 # $2 = 4

0xCE # load sumS into $3

0x2D # sumS = sumS + seed

0x3E # store sumS into M[4]

0x45 # sq = $1 x $1; $1 = 8 most significant bits; $0 = 8 least significant bits

0x54 # drops the 4 LSB of $1 and 4 MSB of $0, merge into $1

0x00

0x11

0xC8 # load numS into $2

0x1B # numS--

0x38 # store numS into M[1]

0x00

0xCC # load address counter into $3

0x1D # increment address counter by 1

0x3C # store address counter into M[0]

0x68 # if $2 != 0, jump to register uno value → loop

0x09

0x19

0x19

0x19 # $2 = 4

0xCE # load sumS into $3

0x05

0x15

0x15

0x15 # $1 = 4

0xBD # logic shift to the right by 4, Average seed

0x00

0xC4 # load address counter into $1

0x3D # store Average seed into M[24]

############################################################## PART B

0x00 # point to first address

0x04 # sum = 0

0x38 # store sum into M[0]

0x09

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # numS = 16

0x11 # point to second address

0x38 # store numS into M[1]

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19

0x19 # $1 = 32

0x11 # point to third address

0x38 # store memory address counter of seed Hamming weights into M[2]

0x01

0x11

0x11

0x11

0x11

0x11

0x11

0x11 # point to 8th address

0x70 # store the next pc value into register uno

0xC4 # load seed in current mem address

0x08 # i = 0

0x71 # store the next pc value into register dos

0x84 # $1 value < 0?, $3 = 1:0

0x9D # if $3 == 0 branch to skip

0x19 # i++ when $1 is negative; Hamming weight of a seed

0xA5 # logic shift to the left once for every loop

0x65 # if $1 != 0 jump to register dos value → loop2

0x0D

0x1D

0xC7 # load mem. addr. ctr. of seed Hamming weights into $1

0x39 # store Hamming weight into M[$1]

0x15 # increment mem. addr. ctr. by 1

0x37 # store mem. addr. ctr. into M[2]

0x04 # point to the 1st memory address

0xCD # load sum into $3

0x2E # sum of 1’s in all 16 seeds, Total Hamming weight

0x3D # store sum into M[0]

0x05 # point to the 2nd memory address

0xCD # load numS into $3

0x1F # decrement numS for each iteration by 1

0x3D # store numS into M[1]

0x11 # increment address counter by 1

0x6C # if $3 != 0 jump to register uno value → loop1

0x00

0xC8 # load sum into $2

0x05

0x15

0x15

0x15 # $1 = 4

0xB9 # logic shift to the right by 4, Average Hamming weight

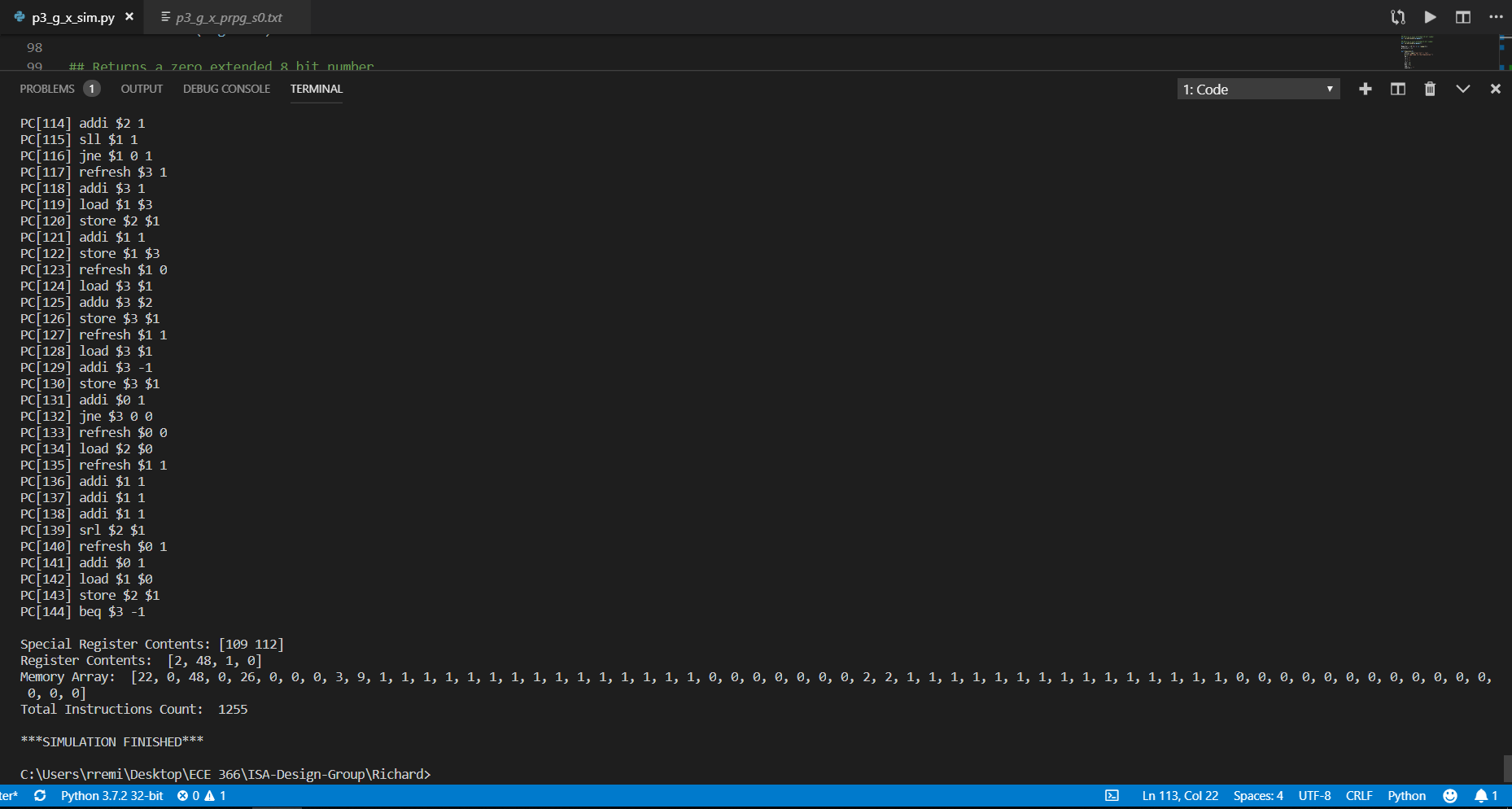
0x01

0x11

0xC4 # load mem. addr. ctr. of seed Hamming weights into $1

0x39 # store AHW into address M[48]

0x9F #STOP



Python Simulator

# Function for converting hex to binary

def hex2bin(argument):

switcher = {

'0': "0000",

'1': "0001",

'2': "0010",

'3': "0011",

'4': "0100",

'5': "0101",

'6': "0110",

'7': "0111",

'8': "1000",

'9': "1001",

'A': "1010",

'B': "1011",

'C': "1100",

'D': "1101",

'E': "1110",

'F': "1111",

}

return switcher.get(argument, "ERROR")

## Table for all instruction opcodes

def getInstr(argument):

switcher = {

"0000": "refresh",

"0001": "addi",

"0010": "addu",

"0011": "store",

"0100": "mult",

"0101": "splice",

"0110": "jne",

"0111": "mark",

"1000": "slti",

"1001": "beq",

"1010": "sll",

"1011": "srl",

"1100": "load",

}

return switcher.get(argument, "ERROR")

## Table for all general purpose registers

def bin2gregi(argument):

switcher = {

'00': "$0",

'01': "$1",

'10': "$2",

'11': "$3",

}

return switcher.get(str(argument), "ERROR")

## Two's complement for 2 bits

def getTwosComp2(argument):

if (argument[0] == '1'):

num = int(argument,2)

val = -2 + (num - 2)

else:

val = int(argument, 2)

return int(val)

## Two's complement for 8 bits

def getTwosComp8(argument):

if (argument[0] == '1'):

num = int(argument,2)

val = -128 + (num - 128)

else:

val = int(argument, 2)

return int(val)

## Two's complement for 32 bits

def getTwosComp32(argument):

if (argument[0] == '1'):

num = int(argument,2)

val = -2147483648 + (num - 2147483648)

else:

val = int(argument, 2)

return int(val)

## Returns bit size of a non-negative number

def getBitSize(argument):

sum = 0

while argument >> sum:

sum += 1

return sum

## Inserts new instruction so long as there is no duplicates on the same PC

def insertList(pc, newline):

check = False

for a, b in printList:

if (a == pc and b == newline):

check = True

if (check == False):

printList.append((pc, newline))

## Returns a zero extended 16 bit number

def zeroExtend16(argument):

zext = 16 - len(argument)

while (zext != 0):

argument = "0" + argument

zext -= 1

return argument

## Returns a zero extended 8 bit number

def zeroExtend8(argument):

zext = 8 - len(argument)

while (zext != 0):

argument = "0" + argument

zext -= 1

return argument

Register = [0 for i in range(4)]

printList = []

def Simulate(I):

oFile = open("p3\_g\_x\_prpg\_sim\_out\_251.txt.", "w")

print("Welcome to the Simulation!")

op = ""

rs = ""

rt = ""

rx = ""

uno = 0

dos = 0

imm = ""

newLine = ""

Memory = [0 for i in range(64)] # list for memory content

PC = 0

instructionCount = 0

finished = False

while(not(finished)):

binary = I[PC] # get instruction binary

if (binary == "10011111"): # END instruction

finished = True

op = binary[0:4]

# refresh

if (op == "0000"):

rt = binary[4:6]

imm = binary[6:8]

Register[int(rt, 2)] = getTwosComp2(imm)

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + str(getTwosComp2(imm))

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

# addi

elif (op == "0001"):

rt = binary[4:6]

imm = binary[6:8]

Register[int(rt, 2)] = Register[int(rt, 2)] + getTwosComp2(imm)

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + str(getTwosComp2(imm))

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

# addu

elif (op == "0010"):

rt = binary[4:6]

rs = binary[6:8]

Register[int(rt, 2)] = Register[int(rt, 2)] + Register[int(rs, 2)]

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + bin2gregi(rs)

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

# store

elif (op == "0011"):

rt = binary[4:6]

rs = binary[6:8]

Memory[abs(Register[int(rs, 2)])] = Register[int(rt, 2)]

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + bin2gregi(rs)

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

# mult

elif (op == "0100"):

rt = binary[4:6]

rs = binary[6:8]

temp16 = bin(Register[int(rt, 2)] \* Register[int(rs, 2)])[2:]

if (len(temp16) < 16):

temp16 = zeroExtend16(temp16)

print("temp16 ", temp16)

Register[0] = int(temp16[8:16], 2)

Register[1] = int(temp16[0:8], 2)

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + bin2gregi(rs)

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

# splice

elif (op == "0101"):

rt = binary[4:6]

rs = binary[6:8]

r0 = zeroExtend8(str(bin(Register[0]))[2:])

r1 = zeroExtend8(str(bin(Register[1]))[2:])

r0 = r0[4:8]

r1 = r1[0:4]

Register[1] = int(r1 + r0, 2)

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + bin2gregi(rs)

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

# jne

elif (op == "0110"):

rt = binary[4:6]

imm = binary[6]

rx = binary[7]

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + str(int(imm, 2)) + " " + rx

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

if (Register[int(rt, 2)] != int(imm, 2)):

if (rx == '0'):

PC = uno

print("Jump ------------> PC[" + str(PC) + "]")

elif (rx == '1'):

PC = dos

else:

PC += 1

insertList(PC, newLine)

instructionCount += 1

# mark

elif (op == "0111"):

rx = binary[7]

if (rx == '0'):

uno = PC + 1

elif (rx == '1'):

dos = PC + 1

newLine = getInstr(op) + " " + rx

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

# slti

elif (op == "1000"):

rt = binary[4:6]

imm = binary[6:8]

if (Register[int(rt, 2)] < int(imm, 2)):

Register[3] = 1

else:

Register[3] = 0

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + str(int(imm))

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

# beq

elif (op == "1001"):

rt = binary[4:6]

imm = binary[6:8]

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + str(getTwosComp2(imm))

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

if (Register[int(rt, 2)] == 0):

PC = PC + 1 + getTwosComp2(imm)

if (getTwosComp2(imm) == -1):

print("EXIT")

else:

print("Branch ------------> PC[" + str(PC) + "]")

else:

PC += 1

insertList(PC, newLine)

instructionCount += 1

# sll

elif (op == "1010"):

rt = binary[4:6]

imm = binary[6:8]

imm1 = Register[int(rt,2)] << int(imm,2)

if (imm1 < -2147483648):

hold = bin(imm1)[3:]

imm1 = getTwosComp32(hold)

Register[int(rt,2)] = -(imm1)

elif (imm1 >= 2147483648):

temp = bin(imm1)[2:]

imm1 = getTwosComp32(temp)

Register[int(rt,2)] = imm1

else:

Register[int(rt,2)] = imm1

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + str(int(imm))

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

# srl

elif (op == "1011"):

rt = binary[4:6]

rs = binary[6:8]

Register[int(rt,2)] = Register[int(rt,2)] >> Register[int(rs,2)]

temp = bin(Register[int(rt,2)])

if (temp[0] == '-'):

Register[int(rt,2)] = abs(Register[int(rt,2)])

ogBitSize = getBitSize(Register[int(rt,2)])

val = Register[int(rt,2)] ^ 255 # get the

val += 1 # two's complement

if(ogBitSize > getBitSize(val)): # check if there are missing zeroes

zeroes = ogBitSize - getBitSize(val)

temp = bin(val)[2:]

while zeroes!=0: # concatanate the missing zeroes

temp = '0' + temp

zeroes -= 1

temp = '1' + temp

Register[int(rt,2)] = int(temp, 2)

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + bin2gregi(rs)

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

# load

elif (op == "1100"):

rt = binary[4:6]

rs = binary[6:8]

Register[int(rt, 2)] = Memory[abs(Register[int(rs, 2)])]

newLine = getInstr(op) + " " + bin2gregi(rt) + " " + bin2gregi(rs)

pr = "PC[" + str(PC) + "] " + newLine

print(pr)

insertList(PC, newLine)

PC += 1

instructionCount += 1

print("\nSpecial Register Contents: [" + str(uno) + " " + str(dos) + "]")

print("Register Contents: ", Register)

print("Memory Array: ", Memory)

print("Total Instructions Count: ",instructionCount)

# Write all instructions to an output file

printList.sort()

for a, b in printList:

oFile.write(b + "\n")

oFile.close()

def main():

iFile = open("p3\_g\_x\_prpg\_251.txt", "r")

I = []

binary = ""

word = ""

for line in iFile:

if (line == "\n" or line[0] == "#" ):

continue

if (line == "0x9F"):

# prints the register contents

print("Registers contents:", Register)

print("\nThankYou")

exit()

word = word + line[2:4] # get each line, but ignore 0x

for i in word:

binary = binary + hex2bin(i) # convert to binary

I.append(binary)

word = ""

binary =""

Simulate(I)

print("\n\*\*\*SIMULATION FINISHED\*\*\*")

if \_\_name\_\_== "\_\_main\_\_":

main()