Vishal Parikh: Group_9

ECE 366 PROJECT 3

[ISA Design]

Part A) ISA Intro

1. **Introduction. This should include the name of the architecture, overall philosophy, specific goals strived for and achieved.**
   - The name of our ISA is the Uncreative Name Instruction Set (UNIS). It was designed to support random number generation and theoretically can support basic arithmetic and logic tasks. One additional goal was to make the syntax as simple and clear as possible

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

Opcode				Function	Format	Result
0	0	0	0	addi_r0	addi_r0 imm	r0 = imm
0	0	0	1	addi_r1	addi_r1 imm	r1 = r1 + imm
0	0	1	0	half_lui	half_lui imm	r3 = imm << 4
0	0	1	1	andi_r3	andi_r3 imm	r3 = r3 & imm
0	1	0	0	sw	sw rd rs	M[rs] = rd
0	1	0	1	lw	lw rd rs	rd = M[rs]
0	1	1	0	mult	mult rd rs	rd = rd * rs
0	1	1	1	mask_top	mask_top rd rs	rd = bitmask_4MSB(rs)
1	0	0	0	srl	srl rd imm	if imm = 0, shift 1, if imm = 1, shift 8
1	0	0	1	or_r	or_r rd rs	rd = rd or rs
1	0	1	0	branch_nz	branch_nz rd	pc += rd
1	0	1	1	count_reset	count_reset	r6 = 15
1	1	0	0	add_r	add_r rd rs	rd = rd + rs
1	1	0	1	subi_r1	subi_r1 imm	r1 = r1 - imm
1	1	1	0	sub_count	sub_count imm	r5 = r5 - 1
1	1	1	1	zero	zero rd	rd = 0

3. **Register design. How many registers are supported? Is there anything special about the registers?**
    - Four registers are supported in terms of what can be accessed by the programmer. There are also 5 implicit registers which hold either counter variables or constants to be used by certain instructions, such as mask top which relies on a register holding the value 0xF000. Other constants include registers that hold the length of branch jumps for resetting PC with branches.

4. **Branch design. What types of branches are supported? How are the target addresses calculated? What is the maximum branch distance supported?**
    - There is one kind of branch which checks the implicit counter variable in register 6. If it is not zero it takes an argument 0-3 which selects what register to update PC's value with. Since our registers are 8bit data types, the maximum distance is -128 or 127.
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?**
    - Data memory can support any 8bit unsigned address, so 256 addresses. The simulator was programmed to support an array of 43 memory locations. Following the example of MIPS, the two memory accessing instructions are lw (load word) and sw (store word.)

6. **What would you have done differently if you had 1 more bit for instructions?**
    - I would have given it to opcodes, allowing more than 16 instructions, which could give a more robust set of basic arithmetic and logic operations, but also room for more special instructions.

7. **How about 1 fewer bit?**
    - I don't think I could have completed the PRPG project with any fewer bits

8. **What are the most significant advantages of your ISA (about 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 advantage of this ISA for the PRPG is that it was written with this program in mind. That made it possible to have a very small instruction set (only 16 instructions) and thus a small footprint. However, the weakness is that the amount of special instructions meant that the hardware implementation became more complicated.

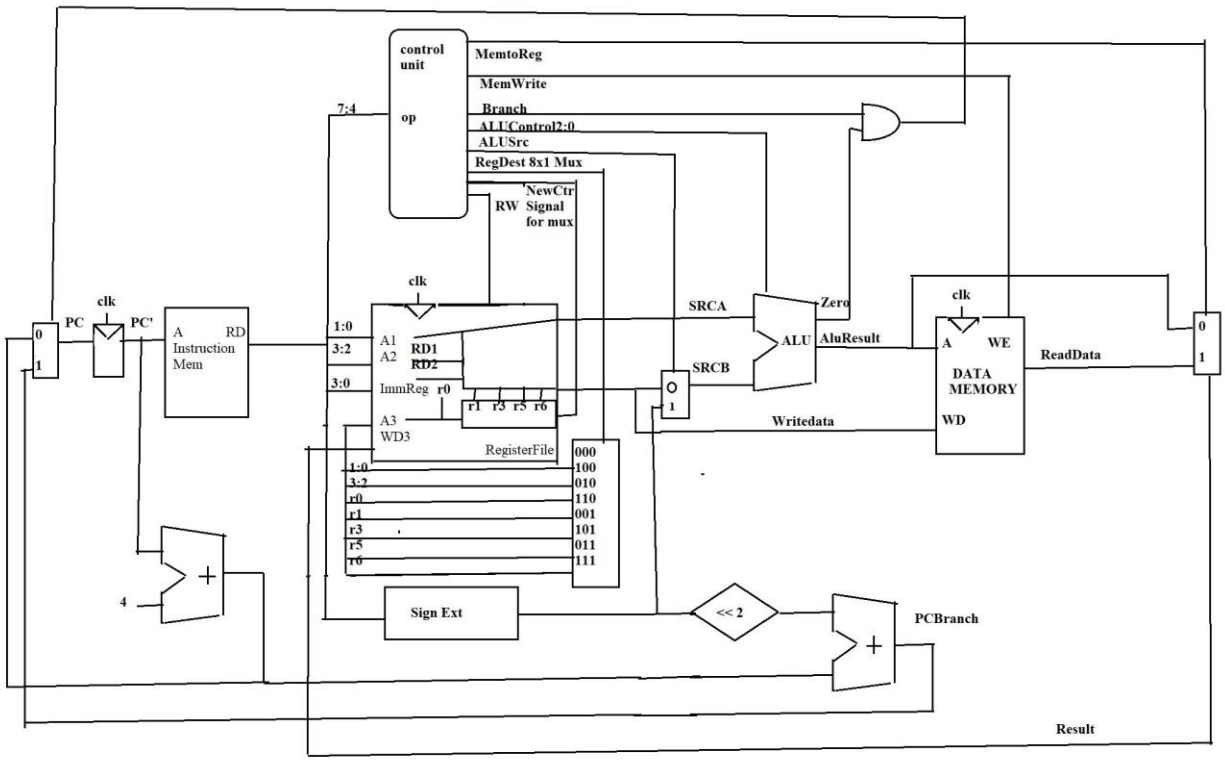
9. **What have you done towards the goals of low DIC and HW simplification? What could have been done differently to better optimize for each of the two goals, if to start over?**
    - The original plan was to keep it down to 8 instructions to allow larger immediate and more registers. In this implementation the ability to branch was not there, and so the

code was very long and hard to modify. When we redesigned the structure of the ISA to allow 16 instructions, we were able to include a branch instruction which made the program instruction count much more efficient. In terms of hardware, we tried to keep it as similar to the basic class example as possible. So where there were areas where we could simply add inputs to a MUX instead of creating whole new control signals and bypasses, that's what we did.

10. **If you are given a chance to restart this project afresh with 3 weeks' time, how would your group have done it differently?**
    - It's a good question and I'm not sure the answer to it. The constraints were so tight and hard to meet as it was, I would really need to think about it a lot more to try to improve on the design.

Part B) Hardware Implementation

1. **CPU Datapath design.** A schematic including register file, ALU, PC logic, and memory components (see textbook ch 7.3.1).



Vishal Parikh: Group_9

Opcode				Function	Regwrite	RegDst	AluSrc	Branch	MemWrite	MemtoReg	ALUop	NewCtrlSgnl
0	0	0	0	addi_r0	1	110	1	0	0	0	00	xx
0	0	0	1	addi_r1	1	001	0	0	0	0	00	00
0	0	1	0	half_lui	1	101	1	0	0	0	10	10
0	0	1	1	andi_r3	1	101	1	0	0	0	1x	10
0	1	0	0	sw	0	xxx	1	0	1	x	00	xx
0	1	0	1	lw	1	010	1	0	0	1	00	xx
0	1	1	0	mult	1	010	1	0	0	0	11	xx
0	1	1	1	mask_top	1	010	1	0	0	0	1x	xx
1	0	0	0	srl	1	010	x	0	0	x	1x	xx
1	0	0	1	or_r	1	010	0	0	0	0	1x	xx
1	0	1	0	branch_nz	0	xxx	x	1	0	0	xx	xx
1	0	1	1	count_reset	1	111	x	0	0	0	xx	xx
1	1	0	0	add_r	1	010	0	0	0	0	00	xx
1	1	0	1	subi_r1	1	001	1	0	0	0	x1	00
1	1	1	0	sub_count	1	011	1	0	0	0	x1	01
1	1	1	1	zero	1	010	x	0	0	0	xx	xx

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.

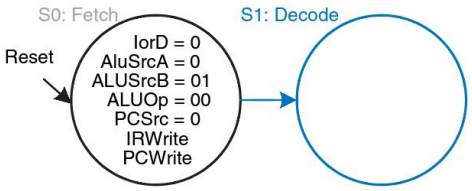


Figure 7.31 Decode

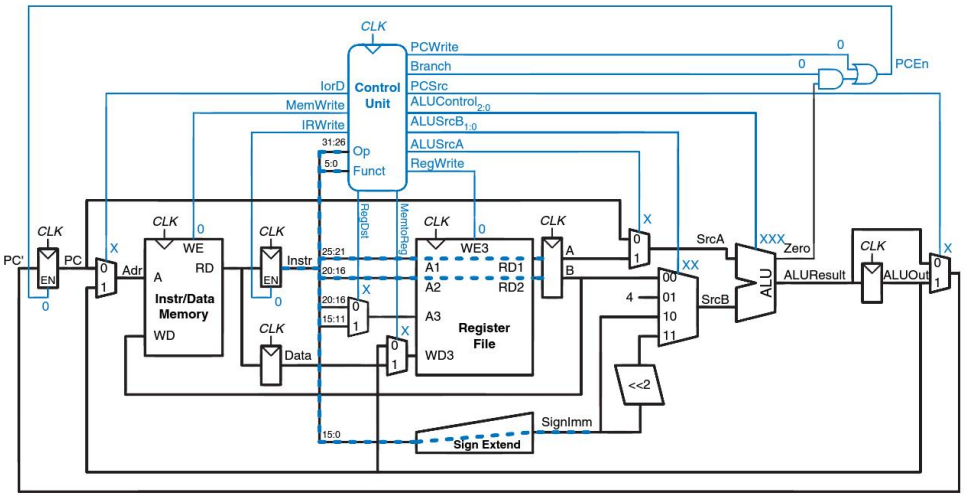
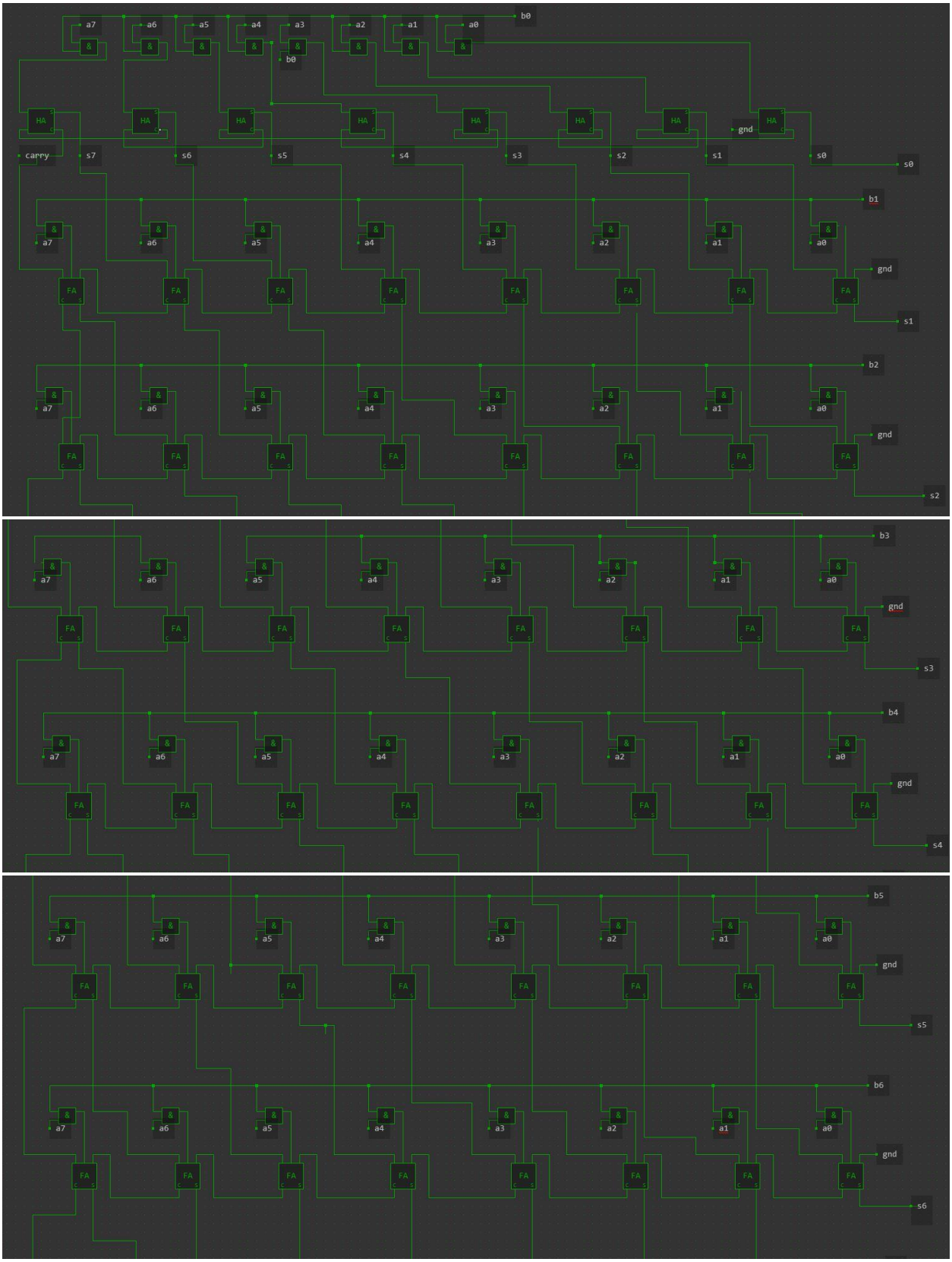


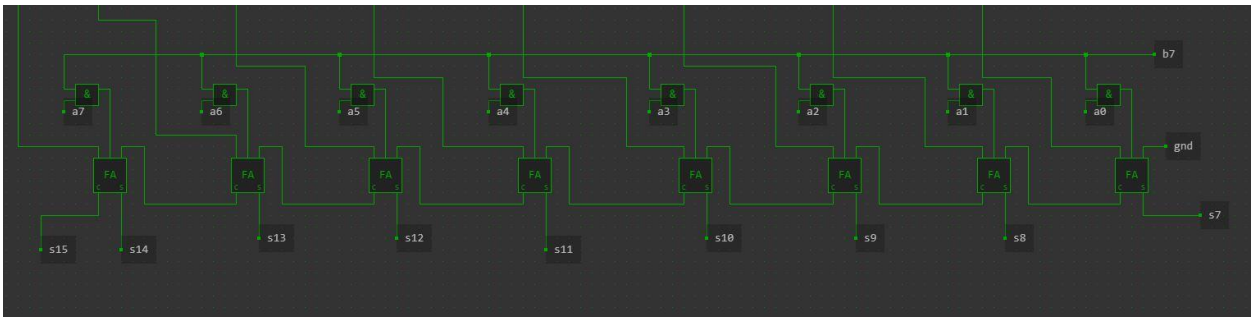
Figure 7.32 Data flow during the decode step

3. **ALU schematic.** A hierarchical sketch of your Arithmetic Logic Unit which implements whatever computation operations that your ISA instructions use (See textbook ch 5.2.4).

Our multiplier schematic, which multiplies [a7...a0] and [b7...b0]. Inputting the same number, it is effectively a combinational squaring circuit. I chose an array multiplier.

Vishal Parikh: Group_9





Part C) Software Package

- S0 = 251
- S0 = 118
- S0 = 79
- Your (non-trivial) choice of S0

For each of the above cases, show the following:

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

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

3. **Screenshots of your Python simulator's output for your PRPG program.** This should convince people that your ISA + Python package works correctly for your PRPG program.

Seed = 251:
Assembly Code:

```
#the initial seed code goes here, will be different for each seed
half_lui 8       # has implicit register use, r3 = x << 4
addi_r0 11              # r0 = y
or_r 0 3        # 3 is register destination, or of r3 r0
half_lui 7
or_r 3 0

#store s0
sw 3 1  #contents of r3 stored at address in r1
mult 3 3                #square the seed
mask_top 2 3     #stores mask of most significant 3 bits of r3 in r2
```

Vishal Parikh: Group_9

```
andi_r3 15            #bit mask the least significant 3 bits of r3, store in r3
srl 2 1               # r2 = r2 >> 8
or_r 3 2              # r3 = r3 or r2
addi_r1 1             # r1 = r1 + 1  memory address increment
sub_count 1         #decriment counter register by 1
sw 3 1
branch_nz 7         #if implicit counter register != 0, pc -= r7
count_reset

#seed sum
subi_r1 1
lw 2 1
add_r 3 2
sub_count 1
branch_nz 9

#average
addi_r1 1
srl 3 2
addi_r1 15
addi_r1 1
sw 3 1
subi_r1 15
subi_r1 1     #memory address in r1 back at 2008
count_reset

#hamming weight, memory address is back at start
lw 3 1
zero 2      #r2 = 0 (number of set bits counter)
andi_r3 1         #check bit 0
add_r 2 3   #r2 = r2 + r3
lw 3 1    #reload value at address in r1
srl 3 0
andi_r3 1 #check bit 1
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
andi_r3 1 #check bit 2
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
```

Vishal Parikh: Group_9

```
andi_r3 1 #check bit 3
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 4
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 5
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 6
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 7
add_r 2 3   #r2 = r2 + r3
addi_r1 15
addi_r1 3   #move forward 18 bytes in memory
sw 2 1      #store r2 in that address
subi_r1 15
subi_r1 2   #memory update to next seed value, requires subtraction by 16
sub_count 1
branch_nz 8   #if implicit counter register != 0, pc -= r8
count_reset
```

Vishal Parikh: Group_9

```
addi_r1 1     # address should now be M[18]
lw 2 1        #load h0 to r2

#hamming sum
addi_r1 1     # address of next hamming number
lw 3 1        #load to temp
add_r 2 3       #add_r to total sum
sub_count 1   #decrement counter
branch_nz 10   #if counter != 0, loop

#average
srl 2 2       #r2 = r2 >> 4
sw 2 1         #store r2 in address in r1
```

Machine Code:

```
00101000
00001011
10010011
00100111
10011100
01001101
01101111
01111011
00111111
10001001
10011110
00010001
11100001
01001101
10100111
10110111
11010001
01011001
11001110
11100001
10101001
00010001
10001110
00011111
00010001
01001101
11011111
11010001
10110001
```

Vishal Parikh: Group_9

01011101
11111000
00110001
11001011
01011101
10001100
00110001
11001011
01011101
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
10001100
10001100
00110001
11001011
01011101

Vishal Parikh: Group_9

10001100
10001100
10001100
10001100
10001100
10001100
10001100
00110001
11001011
00011111
00010011
01001001
11011111
11010010
11100010
10101000
10111000
00010001
01011001
00010001
01011101
11001011
11100001
10101010
10001010
01001001

Screenshots:

Vishal Parikh: Group_9

Atom

objects.py | main.py | simulator.py | machine_s2... | README.md | MIPS-2-ma... | machine.txt... | instru

```python
import objects
import instructions
from simulator import simulate

def main():
    my_registers = objects.registers()      # create registers object
    instr_list = []
    my_memory = objects.memory()
    filename = 'machine_s251.txt'

    print("loading instructions from " + filename)
    file = open(filename, 'r')  #open the instruction file
    i = 0
    #endofinstructions =
    for instr in file:
        if (instr == '\n' or instr[0] == '#'):
            continue
        #instr = instr[0:10]
        #print(instr)
        temp = objects.instr_parsed(instr)
        #temp.print_instr() #pseudocode translation
        instr_list.append(temp)
        i += 1
    instr_list.append(objects.instr_parsed('11111111'))
    simulate(instr_list, my_registers, my_memory)
    return
main()
```

C:\Users\saman\AppData\Local\Programs...

```
loading instructions from machine_s251.txt
***Simulation started***
***Simulation Finished***
Instruction count:1276

The contents of the registers are:
$0: 139
$1: 41
$2: 2
$3: 1
$5: 61440
$7: -8
$8: -59
$9: -4
$10: -4
PC: 99

 The contents of the memory are:
0x8:    251
0x9:    249
0xa:    241
0xb:    225
0xc:    193
0xd:    145
0xe:    81
0xf:    17
0x10:   1
0x11:   1
0x12:   1
0x13:   1
0x14:   1
0x15:   1
0x16:   1
0x17:   1
0x18:   88
0x1a:   7
0x1b:   6
0x1c:   5
0x1d:   4
0x1e:   3
0x1f:   3
0x20:   3
0x21:   2
```

Atom

objects.py | main.py | simulator.py | machine_s2... | README.md | MIPS-2-ma... | machine.txt... | instru

```python
import objects
import instructions
from simulator import simulate

def main():
    my_registers = objects.registers()      # create registers object
    instr_list = []
    my_memory = objects.memory()
    filename = 'machine_s251.txt'

    print("loading instructions from " + filename)
    file = open(filename, 'r')  #open the instruction file
    i = 0
    #endofinstructions =
    for instr in file:
        if (instr == '\n' or instr[0] == '#'):
            continue
        #instr = instr[0:10]
        #print(instr)
        temp = objects.instr_parsed(instr)
        #temp.print_instr() #pseudocode translation
        instr_list.append(temp)
        i += 1
    instr_list.append(objects.instr_parsed('11111111'))
    simulate(instr_list, my_registers, my_memory)
    return
main()
```

C:\Users\saman\AppData\Local\Programs...

$9: -4
$10: -4
PC: 99

 The contents of the memory are:
0x8:   251
0x9:   249
0xa: 241
0xb: 225
0xc: 193
0xd: 145
0xe: 81
0xf: 17
0x10:   1
0x11:   1
0x12:   1
0x13:   1
0x14:   1
0x15:   1
0x16:   1
0x17:   1
0x18:   88
0x1a: 7
0x1b: 6
0x1c: 5
0x1d: 4
0x1e: 3
0x1f: 3
0x20:   3
0x21:   2
0x22:   1
0x23:   1
0x24:   1
0x25:   1
0x26:   1
0x27:   1
0x28:   1
0x29:   2

Process returned 0 (0x0)   execution time :
 0.604 s
Press any key to continue . . .

Vishal Parikh: Group_9

Seed = 118:
Assembly Code:

```
#the initial seed code goes here, will be different for each seed
half_lui 7       # has implicit register use, r3 = x << 4
addi_r0 6              # r0 = y
or_r 3 0       # 3 is register destination, or of r3 r0

#store s0
sw 3 1  #contents of r3 stored at address in r1
mult 3 3               #square the seed
mask_top 2 3     #stores mask of most significant 3 bits of r3 in r2
andi_r3 15             #bit mask the least significant 3 bits of r3, store in r3
srl 2 1                # r2 = r2 >> 8
or_r 3 2           # r3 = r3 or r2
addi_r1 1              # r1 = r1 + 1  memory address increment
sub_count 1        #decriment counter register by 1
sw 3 1
branch_nz 7        #if implicit counter register != 0, pc -= r7
count_reset

#seed sum
subi_r1 1
lw 2 1
add_r 3 2
sub_count 1
branch_nz 9

#average
addi_r1 1
srl 3 2
addi_r1 15
addi_r1 1
sw 3 1
subi_r1 15
subi_r1 1     #memory address in r1 back at 2008
count_reset

#hamming weight, memory address is back at start
lw 3 1
zero 2      #r2 = 0 (number of set bits counter)
andi_r3 1        #check bit 0
add_r 2 3   #r2 = r2 + r3
lw 3 1    #reload value at address in r1
srl 3 0
```

Vishal Parikh: Group_9

```
andi_r3 1 #check bit 1
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
andi_r3 1 #check bit 2
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 3
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 4
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 5
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 6
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
```

```
srl 3 0
andi_r3 1 #check bit 7
add_r 2 3   #r2 = r2 + r3
addi_r1 15
addi_r1 3   #move forward 18 bytes in memory
sw 2 1      #store r2 in that address
subi_r1 15
subi_r1 2   #memory update to next seed value, requires subtraction by 16
sub_count 1
branch_nz 8   #if implicit counter register != 0, pc -= r8
count_reset
addi_r1 1     # address should now be M[18]
lw 2 1        #load h0 to r2

#hamming sum
addi_r1 1     # address of next hamming number
lw 3 1        #load to temp
add_r 2 3       #add_r to total sum
sub_count 1   #decrement counter
branch_nz 10   #if counter != 0, loop

#average
srl 2 2       #r2 = r2 >> 4
sw 2 1         #store r2 in address in r1
```

Machine Code:

```
00100111
00000110
10011100
01001101
01101111
01111011
00111111
10001001
10011110
00010001
11100001
01001101
10100111
10110111
11010001
01011001
11001110
11100001
```

Vishal Parikh: Group_9

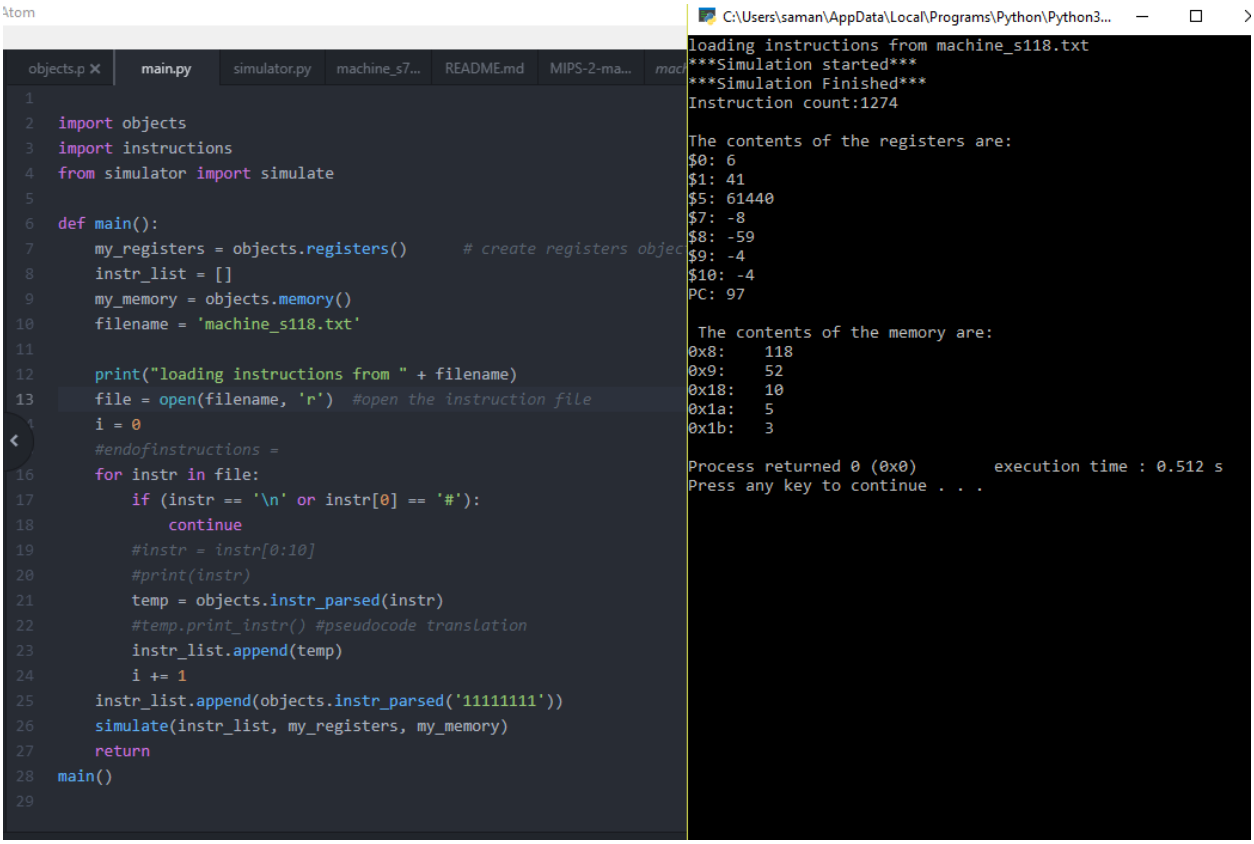
10101001
00010001
10001110
00011111
00010001
01001101
11011111
11010001
10110001
01011101
11111000
00110001
11001011
01011101
10001100
00110001
11001011
01011101
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
10001100
00110001
11001011
01011101

Vishal Parikh: Group_9

10001100
10001100
10001100
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
10001100
10001100
10001100
00110001
11001011
00011111
00010011
01001001
11011111
11010010
11100010
10101000
10111000
00010001
01011001
00010001
01011101
11001011
11100001
10101010
10001010
01001001

Screenshots:

Vishal Parikh: Group_9



```
import objects
import instructions
from simulator import simulate

def main():
    my_registers = objects.registers()      # create registers objec
    instr_list = []
    my_memory = objects.memory()
    filename = 'machine_s118.txt'

    print("loading instructions from " + filename)
    file = open(filename, 'r')  #open the instruction file
    i = 0
    #endofinstructions =
    for instr in file:
        if (instr == '\n' or instr[0] == '#'):
            continue
        #instr = instr[0:10]
        #print(instr)
        temp = objects.instr_parsed(instr)
        #temp.print_instr() #pseudocode translation
        instr_list.append(temp)
        i += 1
    instr_list.append(objects.instr_parsed('11111111'))
    simulate(instr_list, my_registers, my_memory)
    return
main()
```

```
C:\Users\saman\AppData\Local\Programs\Python\Python3...
loading instructions from machine_s118.txt
***Simulation started***
***Simulation Finished***
Instruction count:1274

The contents of the registers are:
$0: 6
$1: 41
$5: 61440
$7: -8
$8: -59
$9: -4
$10: -4
PC: 97

 The contents of the memory are:
0x8:    118
0x9:    52
0x18:   10
0x1a:   5
0x1b:   3

Process returned 0 (0x0)        execution time : 0.512 s
Press any key to continue . . .
```

Seed = 79:
Assembly Code:

```
#the initial seed code goes here, will be different for each seed
half_lui 4      # has implicit register use, r3 = x << 4
addi_r0 15              # r0 = y
or_r 3 0        # 3 is register destination, or of r3 r0


#store s0
sw 3 1  #contents of r3 stored at address in r1
mult 3 3                #square the seed
mask_top 2 3    #stores mask of most significant 3 bits of r3 in r2
andi_r3 15              #bit mask the least significant 3 bits of r3, store in r3
srl 2 1                 # r2 = r2 >> 8
or_r 3 2            # r3 = r3 or r2
addi_r1 1               # r1 = r1 + 1  memory address increment
sub_count 1         #decriment counter register by 1
sw 3 1
branch_nz 7         #if implicit counter register != 0, pc -= r7
count_reset

#seed sum
```

Vishal Parikh: Group_9

```
subi_r1 1
lw 2 1
add_r 3 2
sub_count 1
branch_nz 9

#average
addi_r1 1
srl 3 2
addi_r1 15
addi_r1 1
sw 3 1
subi_r1 15
subi_r1 1     #memory address in r1 back at 2008
count_reset

#hamming weight, memory address is back at start
lw 3 1
zero 2      #r2 = 0 (number of set bits counter)
andi_r3 1         #check bit 0
add_r 2 3   #r2 = r2 + r3
lw 3 1    #reload value at address in r1
srl 3 0
andi_r3 1 #check bit 1
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
andi_r3 1 #check bit 2
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 3
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 4
add_r 2 3   #r2 = r2 + r3
lw 3 1
```

```
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 5
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 6
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 7
add_r 2 3   #r2 = r2 + r3
addi_r1 15
addi_r1 3   #move forward 18 bytes in memory
sw 2 1      #store r2 in that address
subi_r1 15
subi_r1 2   #memory update to next seed value, requires subtraction by 16
sub_count 1
branch_nz 8   #if implicit counter register != 0, pc -= r8
count_reset
addi_r1 1     # address should now be M[18]
lw 2 1        #load h0 to r2

#hamming sum
addi_r1 1     # address of next hamming number
lw 3 1        #load to temp
add_r 2 3       #add_r to total sum
sub_count 1   #decrement counter
branch_nz 10   #if counter != 0, loop
```

Vishal Parikh: Group_9

```
#average
srl 2 2       #r2 = r2 >> 4
sw 2 1         #store r2 in address in r1
```

Machine Code:

```
00100100
00001111
10011100
01001101
01101111
01111011
00111111
10001001
10011110
00010001
11100001
01001101
10100111
10110111
11010001
01011001
11001110
11100001
10101001
00010001
10001110
00011111
00010001
01001101
11011111
11010001
10110001
01011101
11111000
00110001
11001011
01011101
10001100
00110001
11001011
01011101
10001100
10001100
```

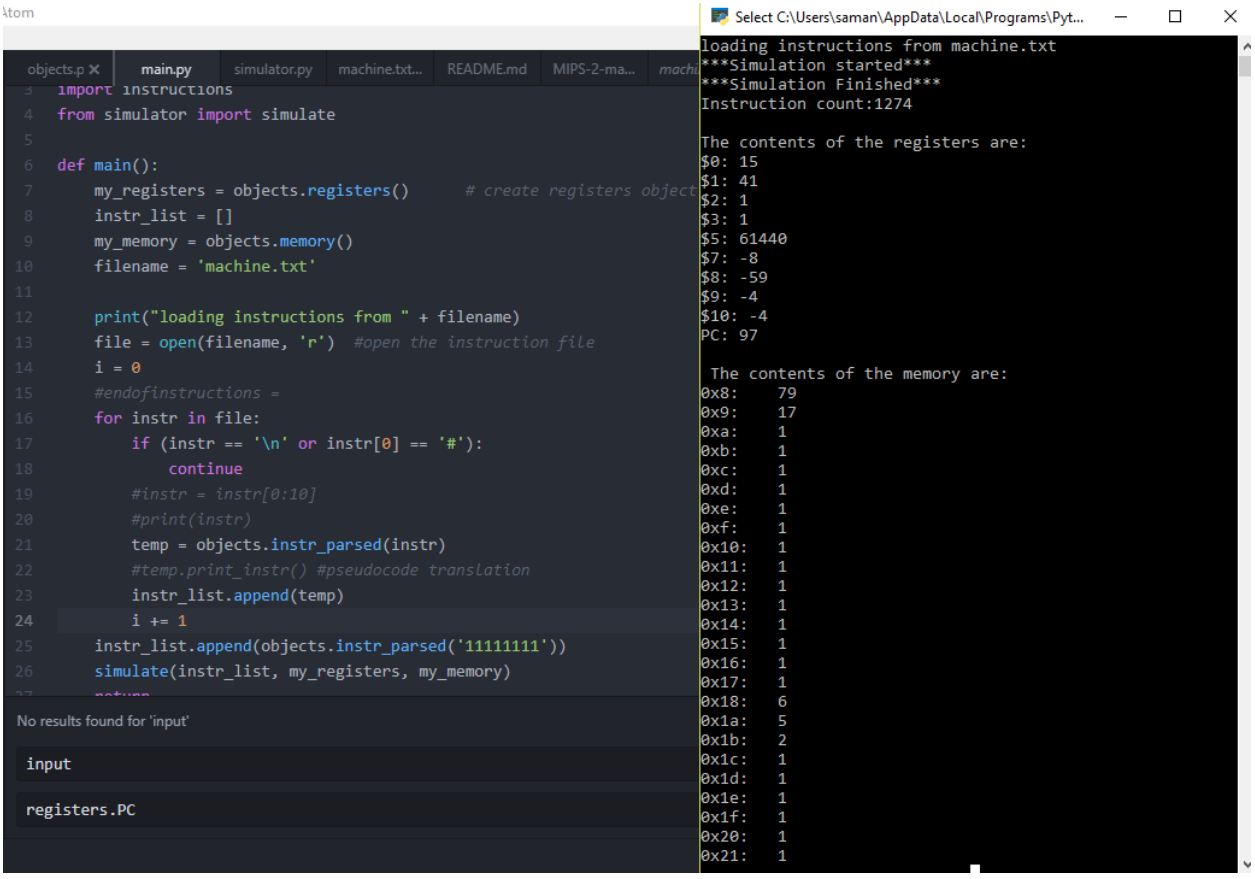
Vishal Parikh: Group_9

00110001
11001011
01011101
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
10001100
10001100
10001100
00110001
11001011
00011111
00010011

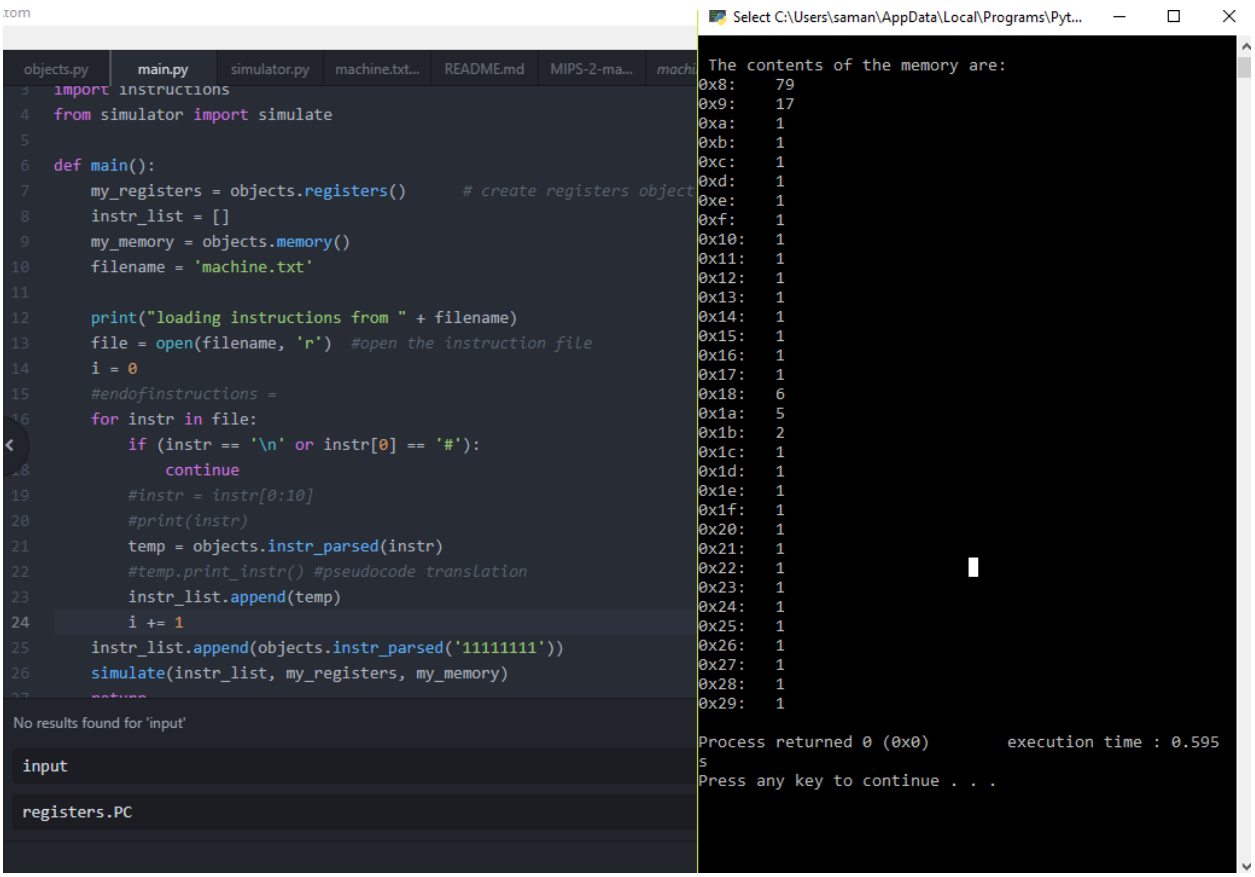
Vishal Parikh: Group_9

01001001
11011111
11010010
11100010
10101000
10111000
00010001
01011001
00010001
01011101
11001011
11100001
10101010
10001010
01001001

Screenshots:



Vishal Parikh: Group_9



Seed = 107 (self-testing)
Assembly Code:

```
#the initial seed code goes here, will be different for each seed
half_lui 6      # has implicit register use, r3 = x << 4
addi_r0 11          # r0 = 15
or_r 3 0      # 3 is register destination, or of r3 r0

#store s0
sw 3 1  #contents of r3 stored at address in r1
mult 3 3            #square the seed
mask_top 2 3    #stores mask of most significant 3 bits of r3 in r2
andi_r3 15          #bit mask the least significant 3 bits of r3, store in r3
srl 2 1             # r2 = r2 >> 8
or_r 3 2          # r3 = r3 or r2
addi_r1 1           # r1 = r1 + 1  memory address increment
sub_count 1       #decriment counter register by 1
sw 3 1
branch_nz 7       #if implicit counter register != 0, pc -= r7
count_reset
```

Vishal Parikh: Group_9

```
#seed sum
subi_r1 1
lw 2 1
add_r 3 2
sub_count 1
branch_nz 9

#average
addi_r1 1
srl 3 2
addi_r1 15
addi_r1 1
sw 3 1
subi_r1 15
subi_r1 1     #memory address in r1 back at 2008
count_reset

#hamming weight, memory address is back at start
lw 3 1
zero 2      #r2 = 0 (number of set bits counter)
andi_r3 1         #check bit 0
add_r 2 3   #r2 = r2 + r3
lw 3 1    #reload value at address in r1
srl 3 0
andi_r3 1 #check bit 1
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
andi_r3 1 #check bit 2
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 3
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 4
```

Vishal Parikh: Group_9

```
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 5
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 6
add_r 2 3   #r2 = r2 + r3
lw 3 1
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
srl 3 0
andi_r3 1 #check bit 7
add_r 2 3   #r2 = r2 + r3
addi_r1 15
addi_r1 3   #move forward 18 bytes in memory
sw 2 1      #store r2 in that address
subi_r1 15
subi_r1 2   #memory update to next seed value, requires subtraction by 16
sub_count 1
branch_nz 8   #if implicit counter register != 0, pc -= r8
count_reset
addi_r1 1     # address should now be M[18]
lw 2 1        #load h0 to r2

#hamming sum
addi_r1 1     # address of next hamming number
lw 3 1        #load to temp
add_r 2 3       #add_r to total sum
sub_count 1   #decrement counter
```

Vishal Parikh: Group_9

```
branch_nz 10   #if counter != 0, loop

#average
srl 2 2       #r2 = r2 >> 4
sw 2 1        #store r2 in address in r1
```

Machine Code:

```
00100110
00001011
10011100
01001101
01101111
01111011
00111111
10001001
10011110
00010001
11100001
01001101
10100111
10110111
11010001
01011001
11001110
11100001
10101001
00010001
10001110
00011111
00010001
01001101
11011111
11010001
10110001
01011101
11111000
00110001
11001011
01011101
10001100
00110001
11001011
01011101
10001100
```

Vishal Parikh: Group_9

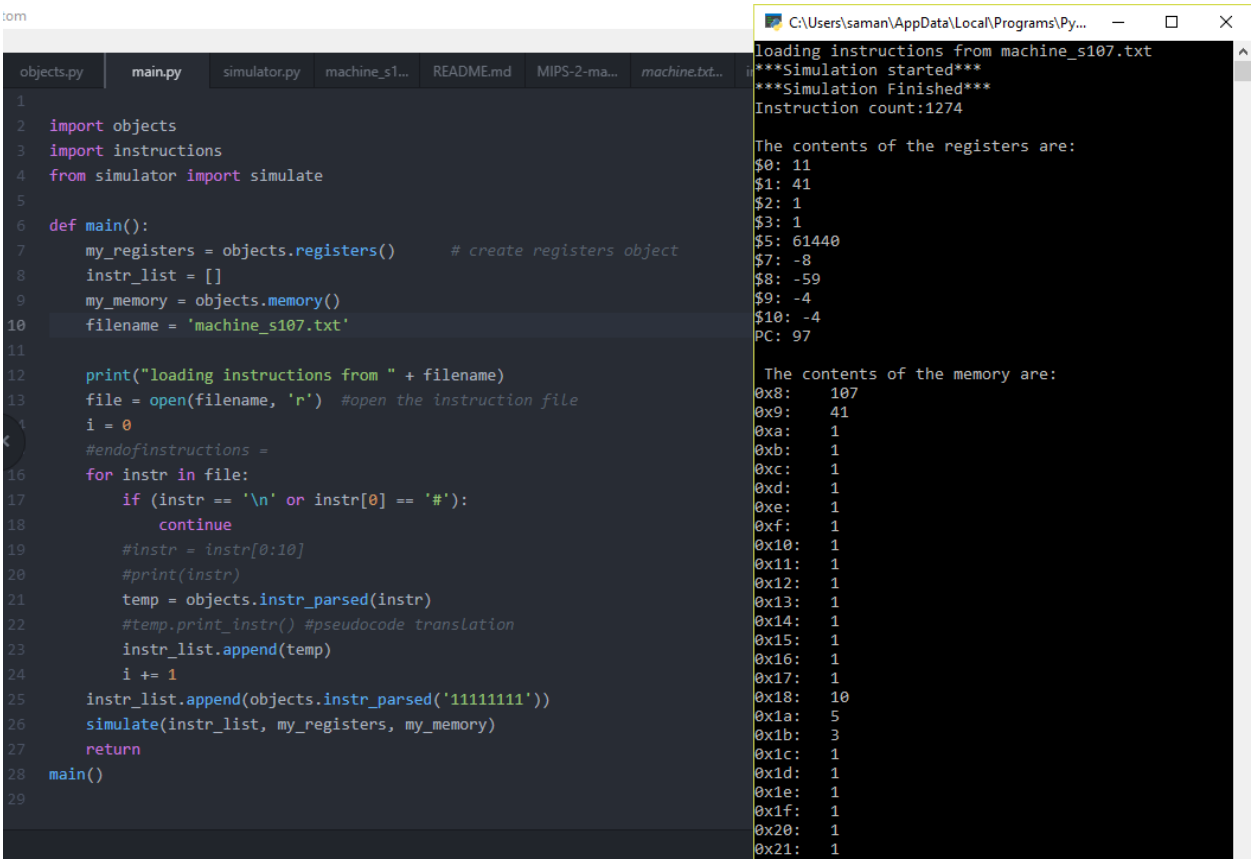
10001100
00110001
11001011
01011101
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
10001100
10001100
00110001
11001011
01011101
10001100
10001100
10001100
10001100
10001100
10001100
10001100
00110001
11001011
00011111

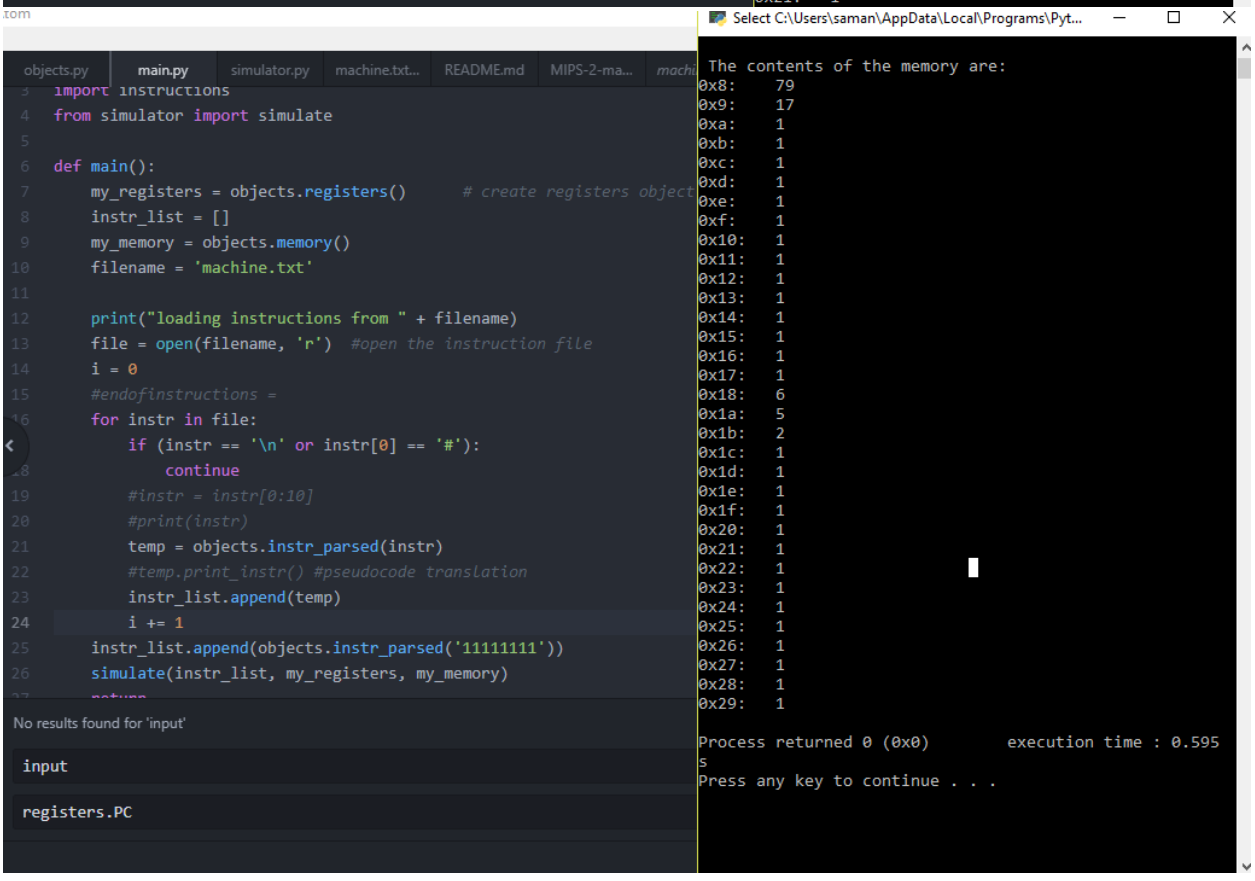
Vishal Parikh: Group_9

00010011
01001001
11011111
11010010
11100010
10101000
10111000
00010001
01011001
00010001
01011101
11001011
11100001
10101010
10001010
01001001

Screenshots:

Vishal Parikh: Group_9





Vishal Parikh: Group_9

Table of group activity

Time/Location	Activity	Achieved/To Do	Members
Week 8 / library	General planning	Set up github, discussed project, laid out tasks	all
Week 9 / whatsapp	Picked tasks	Decided who would do what, more general planning	all
Week 10 / library	ISA design	Began designing the structure of the ISA	all
Week 11/12 online and individual	Coding and diagrams	Coding the isa and simulator, drawing ALU and CPU diagrams, deciding on instruction set	all
Week 12 / library	finishing	Debugging, control signals table and writing report	all

Individual Activity Log

Time/Location	Activity	Achieved/To Do
Week 8	General planning	Set up github, discussed project, laid out tasks
Week 9	Picked tasks	Decided who would do what, more general planning
Week 10	ISA design	Began designing the structure of the ISA
Week 11/12 individua	Coding and diagrams	Coding the isa and simulator, drawing ALU and CPU diagrams, deciding on instruction set
Week 12	finishing	Debugging, control signals table and writing report

Vishal Parikh: Group_9