Project 3

ISA Design for Perceptron Algorithm

Part A) ISA Intro, Q&A

1. **Introduction & Instruction list:**

   Our ISA is called MEDP and our overall philosophy is “functionality first, then efficiency”. A specific goal we strived for was efficiency and to really use the 8-bits given to their fullest potential. Significant features of our ISA are that the instructions only contain 8 total bits, the register can contain up to 16-bit values, and the perceptron algorithm can be performed accurately with only using 13 various instructions. To minimize instruction count we used special registers several times with a lot of our instructions, this allows us to not waste an instruction or two setting a value equal to another. We also added an increment function that saves us on not having to initialize a register to 1 or -1 only to increment/decrement a looping value. For hardware simplification we tried to mimic MIPS with the instructions yet fulfilling the needs of the perceptron algorithm. Some compromises we did to make things work were messing with the jump address and using our jump function along with our branch function together to loop.

Instruction	Functionality	Encoding Format	Machine Code Example	Assembly Example
j imm	PC = imm*2 + PC imm = [-64,63]	**00 iiiiii**	**00 000001**	j 2
bgeqz Rx **Rx[0,15]**	If (Rx >= 0) PC = PC + 2 else PC++	**0100 xxxx**	**0100 0011**	beqz r3
incr Rx, imm **Rx[0,7]**	Rx = Rx + imm; PC++ imm = -1 or 1 Used to increment or decrement a register's value by 1	**0101 xxx i**	**0101 010 0** **Inc if i = 1** **Dec if i = 0**	incr r2, 0 (does r2 = r2 -1) incr r2, 1 (does r2 = r2 +1)
ld Rx, (Ry) **Ry[0,1,2,3]** **Rx[6,7,8,9]**	Rx = MEM[Ry]; PC++	**0110 xx yy**	**0100 00 11**	ld r6, (r3)
st Rx, (Ry) **Ry[0,1,2,3]** **Rx[6,7,8,9]**	MEM[Ry] = Rx; PC++	**0111 xx yy**	**0101 11 01**	st r9, (r1)
mult Rx, Ry **Ry[4,5,6,7]** **Rx[4,5,6,7]**	Rx = Rx × Ry; PC++	**1000 xx yy**	**1000 00 01**	mult r4, r5
div2 Rx **Rx[0,15]**	Rx = Rx / 2: PC++ R15 = remainder	**1001 xxxx**	**1001 1000**	div r8
stsum Rx **Rx[0,15]**	Rx = sum; sum = 0; PC++	**1010 xxxx**	**1010 1100**	stsum r12
sub Rx **Rx[0,15]**	Sum = sum - Rx; PC++	**1011 xxxx**	**1011 1100**	sub r12
add Rx **Rx[0,15]**	sum = sum + Rx; PC++	**1100 xxxx**	**1100 1101**	add r13
stb Rx **Rx[0,15]**	Rx = tempReg ; PC++	**1101 xxxx**	**1101 1101**	stb r13
appb [4-bits]	tempReg = $[tempReg]_2$ [4-bits] (appends 4 bits to whatever bits are currently in tempReg); PC++	**1110 bbbb**	**1101 1001**	appb 1001
initb [4-bits]	tempReg = [4-bits]; PC++ Initializes implicit register tempReg with 4 given bits	**1111 bbbb**	**1111 1010**	initb 1010

2. **Register design:**

There are 18 registers in our ISA, two of them being implicit registers. Register 16 is named “sum” and is tied to instructions sub, add, and stsum. Register 17 is named “tempReg” and is tied to instructions initb, appb, and stb.

Reg #	Description of Registers as used within perceptron program
0	Used for indexing n
1	Memory addrs of x values
2	Memory addrs of y values
3	Memory addrs of C values
4	Used for temporarily storing values
5	Used for temporarily storing values
6	Used for temporarily storing values
7	Used for temporarily storing values
8	Used for C values
9	Used for c values
10	Used for calculations (C-c)
11	Stores B value to be later accessed
12	Stores A value to be later accessed
13	Stores n value to be later accessed
14	Used with
15	Used to store remainder of div2 instruction
sum	Functions with add and sub to allow these instructions to work for all 16 registers
tempReg	Works with initb, appb, and stb to manipulate strings of up to 16-bit binary.

3. **Branch design:**

   Our ISA supports two branches: j and bgeqz. J, or jump, works like a branch in MIPS where the address is calculated using PC, but in MEDP we multiply the immediate number given by two to extend its reach. Bgeqz is not a regular kind of branch, the address is calculated using PC, but it only branches at maximum by two (or one instruction) if the specified register is greater or equal than 0. This type of branch is mostly used to prevent the jump branch from being used in certain situations. The maximum branch distance supported is -64.

4. **Data memory addressing modes:**

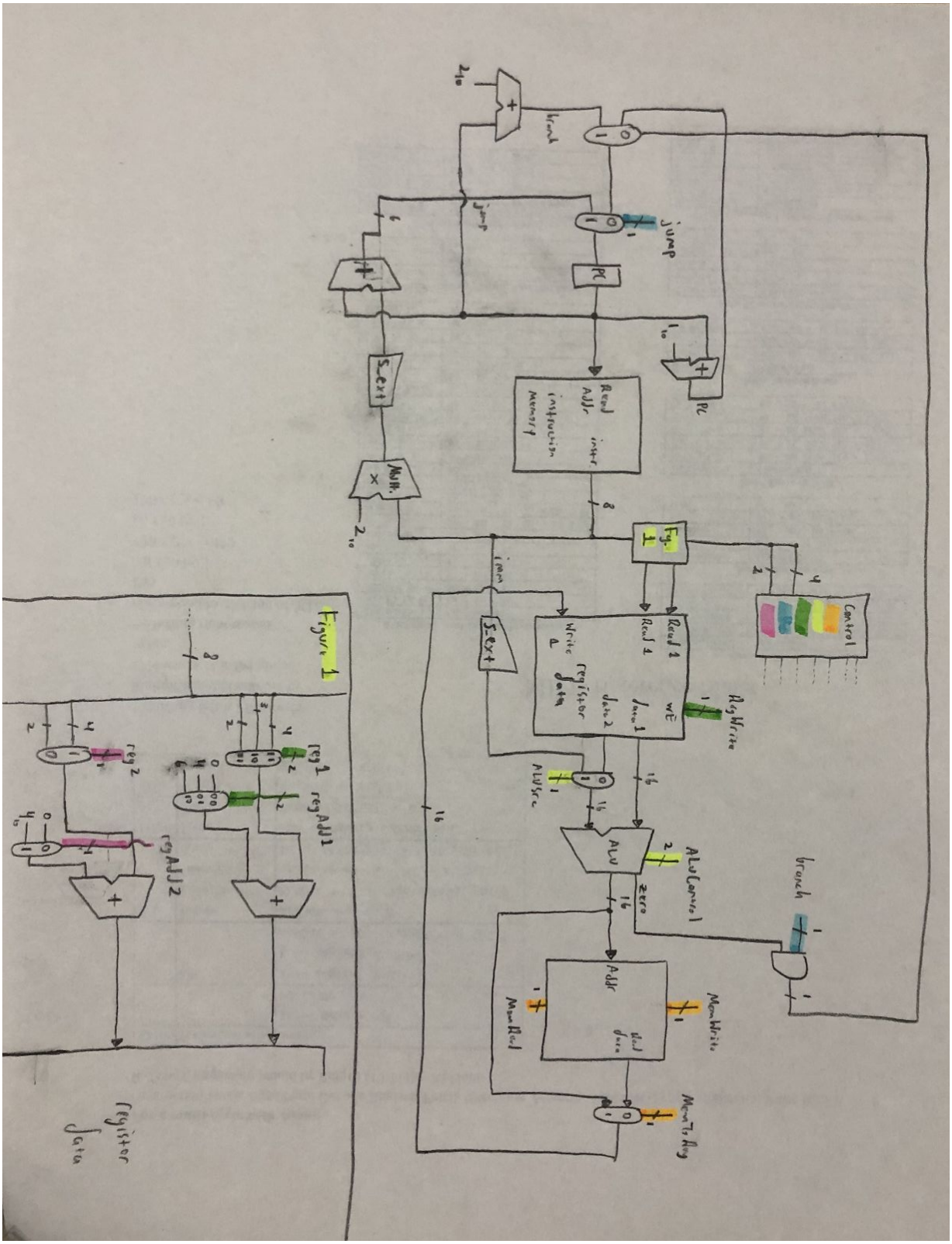
   The instructions we made to access data memory are st and ld. Both of these instructions have an address range of [0, 1023].

5. **Results for Perceptron:**

   The instruction count for case V1 is 7176, while V2 is 17707. The value n has the biggest impact on the instruction count. This value can be at most 16-bits, but it can only be positive. Therefore the value of n for which the instruction count is highest is 2^16 = 65536. A and B's impact on the instruction count is constant, because they are initialized using 4-bits at a time and then immediately stored into registers. Whenever they need to be accessed, it is done through the use of these registers.

Part B) Hardware Sketches

1. CPU Datapath design:

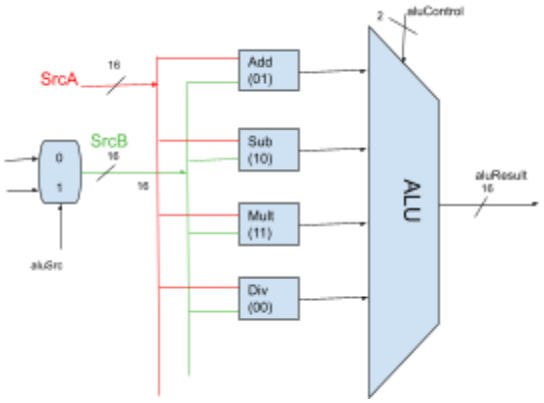


2. **Control logic design:**

	opCode	**regWrite**	**memToReg (ld)**	**memWrite (st)**	**branch (beqR0)**	**jump**	**aluSrc**	**aluControl**
j	00	0	X	0	X	1	X	01 (add)
bgeqz	0100	0	X	0	1	0	0	10 (sub) (Rx - 0 = 0)
incr	0101	1 (Rx)	0	0	0	0	1 (imm)	01 (add)
ld	0110	1 (Rx)	1	0	0	0	0	XX
st	0111	0	0	1	0	0	0	XX
mult	1000	1 (Rx)	0	0	0	0	0 (reg)	11 (multiply)
div2	1001	1 (Rx, R15)	0	0	0	0	1 (imm, 2)	00 (div)
stsum	1010	1 (Rx)	0	0	0	0	0	XX
sub	1011	1 (sum)	0	0	0	0	0 (reg)	10 (sub)
add	1100	1 (sum)	0	0	0	0	0 (reg)	01 (add)
stb	1101	1 (Rx)	0	0	0	0	XX	XX
appB	1110	1 (tempReg)	0	0	0	0	1 (imm 4bit)	01
initB	1111	1 (tempReg)	0	0	0	0	1 (imm 4 bit)	01

	opCode	reg1	regAdd1	reg2	regAdd2
j	00	XX	XX	X	X
bgeqz	0100	XX	XX	X	X
incr	0101	10 3-bits	00	X	X
ld	0110	11 2-bits	10 (Rx [6, 9])	0 2-bits	0 (Ry [0, 3])
st	0111	11	10	0	0
mult	1000	11	01 (Rx [4, 7])	1	01 (Ry [4, 7])
div2	1001	11 (sum)	00	1 (Rx)	0
stsum	1010	11	00	1	0
sub	1011	11	00	1	0
add	1100	11	00	1	0
stb	1101	11 (tempReg)	00	1	0
appB	1110	11 (tempReg)	XX (uses imm)	X	X
initB	1111	11 (tempReg)	XX	X	X

3. ALU schematic:



Part C) Software Package: Perceptron code + Python Simulator

1. Results with the given n, A, B values:

I. Assembly code of our programs:

Perceptron V1

```
1     initb   0000              CORRECT: ALL
2     appb    0000
3     appb    0110
4     appb    0100
5     stb     r13                         # r13 contains n

6     initb   0000
7     appb    0000
8     appb    0000
9     appb    0101
10    stb     r12                         # r12 contains A

11    initb   1111
12    appb    1111
13    appb    1111
14    appb    1010
15    stb     r11         # r11 contains B
#ONLY HIGHLIGHTED CODE CHANGES w/ n, A, and B

16    initb   0000
17    appb    0001
18    appb    0000
19    appb    0000
20    stb     r1                  # r1 contains 256 (mem addr of first x value)
```

Perceptron V2

```
1     initb   0000
2     appb    0000
3     appb    1111
4     appb    1010
5     stb     r13         # r13 contains n

6     initb   1111
7     appb    1111
8     appb    1110
9     appb    1111
10    stb     r12         # r12 contains A

11    initb   0000
12    appb    0000
13    appb    0101
14    appb    0000
15    stb     r11         # r11 contains B
```

The rest is exactly the same as perceptron v1.

```
21      initb   0000
22      appb    0010
23      appb    0000
24      appb    0000
25      stb     r2                      # r2 contains 512 (mem addr of first y value)

26      initb   0001
27      stb     r4                              # r4 is current step size


28      add     r13
29      stsum   r0
30      incr    r0, -1                          # r0 set to n-1

# step1:
31      initb   0000
32      sub     r4
33      stsum   r3                      # reset indexing variable (r3) to  -(cur step size)
34      add     r4
35      stsum   r14                             # set r14 to current step size
36      div2    r14
37      bgeqz   r15     # if cur step size is even (remainder when r14 is div2), branch
fails
38      j       30 (odd)                # if cur step size is odd (jump to “odd” loop)
# even:
39      initb   0000
40      incr    r6, -1
41      st      r6, (r1)
42      st      r7, (r2)
43      incr    r1, 1
44      incr    r2, 1
45      incr    r3, 1
46      incr    r0, -1
47      bgeqz   r0              # r0 = -1 when n points have been created (bgez fails)
48      j       48 (step 2)
49      bgeqz   r3              # r3 = 0 when loop has been performed (cur step size)
times
50      j       -10 (even)

51      sub     r4                              # sum = -(current step)
52      stsum   r3                              # reset indexing value to cur step size
# down:
53      initb   0000
54      incr    r7, -1
55      st      r6, (r1)
56      st      r7, (r2)
57      incr    r1, 1
58      incr    r2, 1
59      incr    r3, 1
```

```
60      incr    r0, -1
61      bgeqz   r0          # branch fails (jump to step 2) when n points created (r0 = -1)
62      j       34 (step 2)
63      bgeqz   r3              # branch when loop performed step size times (r3 = 0)
64      j       -10 (down)
65      incr    r4, 1
66      j       -34 (step1)
# odd:
67      initb   0000
68      incr    r6, 1
69      st      r6, (r1)
70      st      r7, (r2)
71      incr    r1, 1
72      incr    r2, 1
73      incr    r3, 1
74      incr    r0, -1
75      bgeqz   r0          # branch fails (jump to step 2) when n points created (r0 = -1)
76      j       20 (step2)
77      bgeqz   r3              # branch when loop performed step size times (r3 = 0)
78      j       -10 (odd)                       # otherwise repeat current loop
79      sub     r4                              # sum = -(current step size)
80      stsum   r3                              # reset indexing value to cur step size
# up:
81      initb   0000
82      incr    r7, 1
83      st      r6, (r1)
84      st      r7, (r2)
85      incr    r1, 16
86      incr    r2, 1
87      incr    r3, 1
88      incr    r0, -1
89      bgeqz   r0          # branch fails (jump to step 2) when n points created (r0 = -1)
90      j       6 (step2)
91      bgeqz   r3              # branch when loop performed step size times (r3 = 0)
92      j       -10 (up)            # otherwise repeat current loop
93      incr    r4, 1
94      j       -62 (step1)     # increase step size and repeat step 1 (switch to
left-down)
# step2:
95      initb   0000
96      add     r13
97      stsum   r0                              # r0 set to n
98      incr    r0, -1                          # r0 initialized to n-1 (for indexing)

99      initb   0000
100     appb    0001
101     appb    0000
102     appb    0000
103     stb     r1              # r1 reinitialized to 256 (mem addr of first x value)
```

```
104     initb   0000
105     appb    0010
106     appb    0000
107     appb    0000
108     stb     r2                              # r2 reinitialized to 512 (mem addr of
first y value)

109     initb   0000
110     appb    0011
111     appb    0000
112     appb    0000
113     stb     r3          # r3 initialized to 768 (memory addr for first C values)

114     add     r12
115     stsum   r4                              # initialize r4 to A
116     add     r11
117     stsum   r5                              # initialize r5 to B

# C_pos:
118     initb   0000
119     ld      r6, (r1)
120     ld      r7, (r2)
121     mult    r6, r4                          # r4 = Ax
122     mult    r7, r5                          # r5 = By

123     add     r6
124     add     r7
125     stsum   r10                             # r10 = Ax + By
126     bgeqz   r10
127     j       14 (C_neg)                      # jump to C neg if Ax+By < 0

128     initb   0000
129     appb    0000
130     appb    0000
131     appb    0001
132     stb     r8                              # r8 = 1

133     st      r8, (r3)
134     incr    r0, -1
135     incr    r1, 1
136     incr    r2, 1
137     incr    r3, 1                           # increase memory addrs
138     bgeqz   r0                  # if all data points have been classified then go to step 3
139     j       16      # (step3)
140     j       - 22 #   (C_pos),     otherwise repeat classification step

# C_neg:
141     initb   0000
```

```
142     initb   1111
143     appb    1111
144     appb    1111
145     appb    1111
146     stb     r8                                  # r8 = -1

147     st      r8, (r3)
148     inc     r1, 1
149     inc     r2, 1
150     inc     r3, 1                               # increase memory addresses
151     inc     r0, -1
152     bgeqz   r0          # if all data points have been classified then go to step 3
153     J       2(step3)
154     j       -34 (C_pos)
# step3:
155     initb   0000
156     initb   0000
157     appb    0000
158     appb    0000
159     appb    0001
160     stb     r4                                  # r4 = a initially set to 1
161     stb     r5                                  # r5 = b initially set to 1

162     initb   0000
163     appb    0001
164     appb    1111
165     appb    1111
166     stb     r1          # r1 initialized to 251 (mem addr of first x value)

167     initb   0000
168     appb    0001
169     appb    1111
170     appb    1111
171     stb     r2          # r2 reinitialized to 511 (mem addr of first y value)

172     initb   0000
173     appb    0010
174     appb    1111
175     appb    1111
176     stb     r3          # r3 initialized to 767 (memory addr for first C values)

177     add     r13
178     stsum   r0                                  # r0 = n
179     incr    r0, -1                              # r0 = n-1 (for indexing)


# alg:
180     initb   0000
181     inc     r1, 1
```

```
182     inc     r2, 1
183     inc     r3, 1                           # increment mem addr at start of loop
184     ld      r6, (r1)
185     ld      r7, (r2)
186     ld      r8, (r3)
187     mult    r6, r4                          # r6 = x*a = ax
188     mult    r7, r5                          # r7 = y*b = by

189     add     r6
190     add     r7
191     stsum   r10                             # r10 = ax +by
192     bgeqz   r10
193     j       26 (ltz)                        # jump to ltz if ax + by < 0

194     initb   0001
195     stb     r9                              # r9 = c = 1

196     add     r8
197     sub     r9
198     stsum   r10                             # r10 = C-c
199     div2    r10         # r10 = [(C-c)/2]; if c > C, r7 = -1; if c < C, r7 = 1; if c == C, r7 =0

200     add     r10
201     stsum   r7                              # set r7 to change factor

202     ld      r6, (r1)                        # reset r6 to x value
203     mult    r6, r7              # r6 = x, -x, or 0 depending on change factor
204     add     r4
205     add     r6
206     stsum   r4                              # update a

207     add     r10
208     stsum   r6                              # set r6 to change factor

209     ld      r7,(r2)                         # reset r7 to y value
210     mult    r7, r6              # r7 = y, -y, or 0 depending on change factor

211     add     r5
212     add     r7
213     stsum   r5                              # update b

214     incr    r0, -1                          # increase n indexing value
215     bgeqz   r0              # if a and b have been adjusted n amount of times (r0 = -1)
# jump to final
216     j       28 (final)
217     j       -36 (alg)                   # otherwise repeat adjusting process

# Ltz:
218     initb   0000
```

```
219     initb   1111
220     stb     r9                              # r9 = c = -1

221     add     r8
222     sub     r9
223     stsum   r10                             # r10 = C-c
224     div2    r10         # r10 = [(C-c)/2]; if c > C, r7 = -1; if c < C, r7 = 1; if c == C, r7 =0

225     add     r10
226     stsum   r7                              # set r7 to change factor

227     ld      r6, (r1)                        # reset r6 to x value
228     mult    r6, r7                  # r6 = x, -x, or 0 depending on change factor
229     add     r4
230     add     r6
231     st      r4                              # update a

232     add     r10
233     stsum   r6                              # set r6 to change factor

234     ld      r7,(r2)                         # reset r7 to y value
235     mult    r7, r6                  # r7 = y, -y, or 0 depending on change factor
236     add     r5
237     add     r7
238     st      r5                              # update b

239     incr    r0, -1                          # decrease n indexing value
240     bgeqz   r0
241     j       2 final                         # jump to after n loops (when r0 = n
-1)
242     j       -62 (alg)               # loop until n loops have been performed


final:
243     initb   0000
244     Initb   0000
245     stb     r1                              # set mem addr to 0 in order to store
final a

246     add     r4
247     stsum   r9          # store r4 into r9 so that it can be stored into memory using st
248     st      r9, (r1)
249     incr    r1, 1

250     add     r5
251     stsum   r9
252     st      r9, (r1)                        # store final b at mem addr 1
```

II. Machine code in binary:

Perceptron V1	**Perceptron V2**
11110000 11100000 11100110 11100100 11011101 11110000 11100000 11100000 11100101 11011100 11111111 11101111 11101111 11101010 11011011 11110000 11100001 11100000 11100000 11010001 11110000 11100010 11100000 11100000 11010010 11110001 11010100 11001101 10100000 01010000 11110000 10110100 10100011 11000100 10101110 10011110 01001111 00001111 11110000 01011100 01110001 01110110 01010011 01010101 01010111 01010000 01000000 00011000 01000011 00111011 10110100 10100011 11110000 01011110 01110001	11110000 11100000 11101111 11101010 11011101 11111111 11101111 11101110 11101111 11011100 11110000 11100000 11100101 11100000 11011011 11110000 11100001 11100000 11100000 11010001 11110000 11100010 11100000 11100000 11010010 11110001 11010100 11001101 10100000 01010000 11110000 10110100 10100011 11000100 10101110 10011110 01001111 00001111 11110000 01011100 01110001 01110110 01010011 01010101 01010111 01010000 01000000 00011000 01000011 00111011 10110100 10100011 11110000 01011110 01110001

01110110	01110110
01010011	01010011
01010101	01010101
01010111	01010111
01010000	01010000
01000000	01000000
00010001	00010001
01000011	01000011
00111011	00111011
01011001	01011001
00101111	00101111
11110000	11110000
01011101	01011101
01110001	01110001
01110110	01110110
01010011	01010011
01010101	01010101
01010111	01010111
01010000	01010000
01000000	01000000
00001010	00001010
01000011	01000011
00111011	00111011
10110100	10110100
10100011	10100011
11110000	11110000
01011111	01011111
01110001	01110001
01110110	01110110
01010011	01010011
01010101	01010101
01010111	01010111
01010000	01010000
01000000	01000000
00000011	00000011
01000011	01000011
00111011	00111011
01011001	01011001
00100001	00100001
11110000	11110000
11001101	11001101
10100000	10100000
01010000	01010000
11110000	11110000
11100001	11100001
11100000	11100000
11100000	11100000
11010001	11010001
11110000	11110000
11100010	11100010
11100000	11100000
11100000	11100000
11010010	11010010
11110000	11110000
11100011	11100011
11100000	11100000
11100000	11100000
11010011	11010011
11001100	11001100
10100100	10100100
11001011	11001011
10100101	10100101
11110000	11110000
01100001	01100001
01100110	01100110

10001000	10001000
10001101	10001101
11000110	11000110
11000111	11000111
10101010	10101010
01001010	01001010
00000111	00000111
11110000	11110000
11100000	11100000
11100000	11100000
11100001	11100001
11011000	11011000
01111011	01111011
01010000	01010000
01010011	01010011
01010101	01010101
01010111	01010111
01000000	01000000
00001000	00001000
00110101	00110101
11110000	11110000
11111111	11111111
11101111	11101111
11101111	11101111
11101111	11101111
11011000	11011000
01111011	01111011
01010011	01010011
01010101	01010101
01010111	01010111
01010000	01010000
01000000	01000000
00000001	00000001
00101100	00101100
11110000	11110000
11110000	11110001
11100000	11010100
11100000	11010101
11100001	11110000
11010100	11100000
11010101	11101111
11110000	11101111
11100000	11010001
11101111	11110000
11101111	11100001
11010001	11101111
11110000	11101111
11100001	11010010
11101111	11110000
11101111	11100010
11010010	11101111
11110000	11101111
11100010	11010011
11101111	11001101
11101111	10100000
11010011	01010000
11001101	11110000
10100000	01010011
01010000	01010101
11110000	01010111
01010011	01100001
01010101	01100110
01010111	01101011
01100001	10001000
01100110	10001101

01101011	11000110
10001000	11000111
10001101	10101010
11000110	01001010
11000111	00001101
10101010	11110001
01001010	11011001
00001101	11001000
11110001	10111001
11011001	10101010
11001000	10011010
10111001	11001010
10101010	10100111
10011010	01100001
11001010	10001011
10100111	11000100
01100001	11000110
10001011	10100100
11000100	11001010
11000110	10100110
10100100	01100110
11001010	10001110
10100110	11000101
01100110	11000111
10001110	10100101
11000101	01010000
11000111	01000000
10100101	00001110
01010000	00101110
01000000	11110000
00001110	11111111
00101110	11011001
11110000	11001000
11111111	10111001
11011001	10101010
11001000	10011010
10111001	11001010
10101010	10100111
10011010	01100001
11001010	10001011
10100111	11000100
01100001	11000110
10001011	10100100
11000100	11001010
11000110	10100110
10100100	01100110
11001010	10001110
10100110	11000101
01100110	11000111
10001110	10100101
11000101	01010000
11000111	01000000
10100101	00000001
01010000	00100001
01000000	11110000
00000001	11110000
00100001	11010001
11110000	11000100
11110000	10101001
11010001	01111101
11000100	01010011
10101001	11000101
01111101	10101001
01010011	01111101
11000101	

10101001 01111101	

III. Screenshots of your Python simulator's output for your program

Perceptron V1:

Instruction Count with final register values:

```
Translate instructions into writefile.txt (y/n)? n

    MEDP Simulator


Registers:

$0:       -1
$1:       1
$2:       611
$3:       867
$4:       4
$5:       -5
$6:       0
$7:       0
$8:       -1
$9:       -5
$10:      0
$11:      -6
$12:      5
$13:      100
$14:      5
$15:      0
sum:      0
tempReg: 0000
PC: 252

Instruction Count: 7176
```

Final Result:

```
Memory Content:

0:      4   1:      -5  2:      0   3:      0   4:      0   5:      0   6:      0   7:      0
8:      0   9:      0   10:     0   11:     0   12:     0   13:     0   14:     0   15:     0
16:     0   17:     0   18:     0   19:     0   20:     0   21:     0   22:     0   23:     0
24:     0   25:     0   26:     0   27:     0   28:     0   29:     0   30:     0   31:     0
```

x-values:

```
248:    0   249:    0   250:    0   251:    0   252:    0   253:    0   254:    0   255:    0
256:    1   257:    1   258:    0   259:    -1  260:    -1  261:    -1  262:    0   263:    1
264:    2   265:    2   266:    2   267:    2   268:    1   269:    0   270:    -1  271:    -2
272:    -2  273:    -2  274:    -2  275:    -2  276:    -1  277:    0   278:    1   279:    2
280:    3   281:    3   282:    3   283:    3   284:    3   285:    3   286:    2   287:    1
288:    0   289:    -1  290:    -2  291:    -3  292:    -3  293:    -3  294:    -3  295:    -3
296:    -3  297:    -3  298:    -2  299:    -1  300:    0   301:    1   302:    2   303:    3
304:    4   305:    4   306:    4   307:    4   308:    4   309:    4   310:    4   311:    4
312:    3   313:    2   314:    1   315:    0   316:    -1  317:    -2  318:    -3  319:    -4
320:    -4  321:    -4  322:    -4  323:    -4  324:    -4  325:    -4  326:    -4  327:    -4
328:    -3  329:    -2  330:    -1  331:    0   332:    1   333:    2   334:    3   335:    4
336:    5   337:    5   338:    5   339:    5   340:    5   341:    5   342:    5   343:    5
344:    5   345:    5   346:    4   347:    3   348:    2   349:    1   350:    0   351:    -1
352:    -2  353:    -3  354:    -4  355:    -5  356:    0   357:    0   358:    0   359:    0
360:    0   361:    0   362:    0   363:    0   364:    0   365:    0   366:    0   367:    0
```

y-values:

```
512:    0   513:    1   514:    1   515:    1   516:    0   517:    -1  518:    -1  519:    -1
520:    -1  521:    0   522:    1   523:    2   524:    2   525:    2   526:    2   527:    2
528:    1   529:    0   530:    -1  531:    -2  532:    -2  533:    -2  534:    -2  535:    -2
536:    -2  537:    -1  538:    0   539:    1   540:    2   541:    3   542:    3   543:    3
544:    3   545:    3   546:    3   547:    3   548:    2   549:    1   550:    0   551:    -1
552:    -2  553:    -3  554:    -3  555:    -3  556:    -3  557:    -3  558:    -3  559:    -3
560:    -3  561:    -2  562:    -1  563:    0   564:    1   565:    2   566:    3   567:    4
568:    4   569:    4   570:    4   571:    4   572:    4   573:    4   574:    4   575:    4
576:    3   577:    2   578:    1   579:    0   580:    -1  581:    -2  582:    -3  583:    -4
584:    -4  585:    -4  586:    -4  587:    -4  588:    -4  589:    -4  590:    -4  591:    -4
592:    -4  593:    -3  594:    -2  595:    -1  596:    0   597:    1   598:    2   599:    3
600:    4   601:    5   602:    5   603:    5   604:    5   605:    5   606:    5   607:    5
608:    5   609:    5   610:    5   611:    5   612:    0   613:    0   614:    0   615:    0
616:    0   617:    0   618:    0   619:    0   620:    0   621:    0   622:    0   623:    0
```

c-values:

```
760:    0   761:    0   762:    0   763:    0   764:    0   765:    0   766:    0   767:    0
768:    1   769:    -1  770:    -1  771:    -1  772:    -1  773:    1   774:    1   775:    1
776:    1   777:    1   778:    1   779:    -1  780:    -1  781:    -1  782:    -1  783:    -1
784:    -1  785:    -1  786:    -1  787:    1   788:    1   789:    1   790:    1   791:    1
792:    1   793:    1   794:    1   795:    1   796:    1   797:    -1  798:    -1  799:    -1
800:    -1  801:    -1  802:    -1  803:    -1  804:    -1  805:    -1  806:    -1  807:    -1
808:    -1  809:    1   810:    1   811:    1   812:    1   813:    1   814:    1   815:    1
816:    1   817:    1   818:    1   819:    1   820:    1   821:    1   822:    1   823:    -1
824:    -1  825:    -1  826:    -1  827:    -1  828:    -1  829:    -1  830:    -1  831:    -1
832:    -1  833:    -1  834:    -1  835:    -1  836:    -1  837:    -1  838:    -1  839:    1
840:    1   841:    1   842:    1   843:    1   844:    1   845:    1   846:    1   847:    1
848:    1   849:    1   850:    1   851:    1   852:    1   853:    1   854:    1   855:    1
856:    1   857:    -1  858:    -1  859:    -1  860:    -1  861:    -1  862:    -1  863:    -1
864:    -1  865:    -1  866:    -1  867:    -1  868:    0   869:    0   870:    0   871:    0
872:    0   873:    0   874:    0   875:    0   876:    0   877:    0   878:    0   879:    0
```

Perceptron V2:

Instruction Count with final register values:

```
Translate instructions into writefile.txt (y/

    MEDP Simulator


Registers:

$0:      -1
$1:      1
$2:      761
$3:      1017
$4:      0
$5:      15
$6:      0
$7:      0
$8:      1
$9:      15
$10:     0
$11:     80
$12:     -17
$13:     250
$14:     8
$15:     0
sum:     0
tempReg: 0000
PC: 249

Instruction Count: 17707
```

Final Result:

```
Memory Content:

0:      0    1:      15   2:      0    3:      0    4:      0    5:      0    6:      0    7:      0
8:      0    9:      0    10:     0    11:     0    12:     0    13:     0    14:     0    15:     0
16:     0    17:     0    18:     0    19:     0    20:     0    21:     0    22:     0    23:     0
```

x-values:

```
248:    0    249:    0    250:    0    251:    0    252:    0    253:    0    254:    0    255:    0
256:    1    257:    1    258:    0    259:    -1   260:    -1   261:    -1   262:    0    263:    1
264:    2    265:    2    266:    2    267:    2    268:    1    269:    0    270:    -1   271:    -2
272:    -2   273:    -2   274:    -2   275:    -2   276:    -1   277:    0    278:    1    279:    2
280:    3    281:    3    282:    3    283:    3    284:    3    285:    3    286:    2    287:    1
288:    0    289:    -1   290:    -2   291:    -3   292:    -3   293:    -3   294:    -3   295:    -3
296:    -3   297:    -3   298:    -2   299:    -1   300:    0    301:    1    302:    2    303:    3
304:    4    305:    4    306:    4    307:    4    308:    4    309:    4    310:    4    311:    4
312:    3    313:    2    314:    1    315:    0    316:    -1   317:    -2   318:    -3   319:    -4
320:    -4   321:    -4   322:    -4   323:    -4   324:    -4   325:    -4   326:    -4   327:    -4
328:    -3   329:    -2   330:    -1   331:    0    332:    1    333:    2    334:    3    335:    4
336:    5    337:    5    338:    5    339:    5    340:    5    341:    5    342:    5    343:    5
344:    5    345:    5    346:    4    347:    3    348:    2    349:    1    350:    0    351:    -1
352:    -2   353:    -3   354:    -4   355:    -5   356:    -5   357:    -5   358:    -5   359:    -5
360:    -5   361:    -5   362:    -5   363:    -5   364:    -5   365:    -5   366:    -4   367:    -3
368:    -2   369:    -1   370:    0    371:    1    372:    2    373:    3    374:    4    375:    5
376:    6    377:    6    378:    6    379:    6    380:    6    381:    6    382:    6    383:    6
384:    6    385:    6    386:    6    387:    6    388:    5    389:    4    390:    3    391:    2
392:    1    393:    0    394:    -1   395:    -2   396:    -3   397:    -4   398:    -5   399:    -6
400:    -6   401:    -6   402:    -6   403:    -6   404:    -6   405:    -6   406:    -6   407:    -6
408:    -6   409:    -6   410:    -6   411:    -6   412:    -5   413:    -4   414:    -3   415:    -2
416:    -1   417:    0    418:    1    419:    2    420:    3    421:    4    422:    5    423:    6
424:    7    425:    7    426:    7    427:    7    428:    7    429:    7    430:    7    431:    7
432:    7    433:    7    434:    7    435:    7    436:    7    437:    7    438:    6    439:    5
440:    4    441:    3    442:    2    443:    1    444:    0    445:    -1   446:    -2   447:    -3
448:    -4   449:    -5   450:    -6   451:    -7   452:    -7   453:    -7   454:    -7   455:    -7
456:    -7   457:    -7   458:    -7   459:    -7   460:    -7   461:    -7   462:    -7   463:    -7
464:    -7   465:    -7   466:    -6   467:    -5   468:    -4   469:    -3   470:    -2   471:    -1
472:    0    473:    1    474:    2    475:    3    476:    4    477:    5    478:    6    479:    7
480:    8    481:    8    482:    8    483:    8    484:    8    485:    8    486:    8    487:    8
488:    8    489:    8    490:    8    491:    8    492:    8    493:    8    494:    8    495:    8
496:    7    497:    6    498:    5    499:    4    500:    3    501:    2    502:    1    503:    0
504:    -1   505:    -2   506:    0    507:    0    508:    0    509:    0    510:    0    511:    0
```

y-values:

```
512:    0    513:    1    514:    1    515:    1    516:    0    517:    -1   518:    -1   519:    -1
520:    -1   521:    0    522:    1    523:    2    524:    2    525:    2    526:    2    527:    2
528:    1    529:    0    530:    -1   531:    -2   532:    -2   533:    -2   534:    -2   535:    -2
536:    -2   537:    -1   538:    0    539:    1    540:    2    541:    3    542:    3    543:    3
544:    3    545:    3    546:    3    547:    3    548:    2    549:    1    550:    0    551:    -1
552:    -2   553:    -3   554:    -3   555:    -3   556:    -3   557:    -3   558:    -3   559:    -3
560:    -3   561:    -2   562:    -1   563:    0    564:    1    565:    2    566:    3    567:    4
568:    4    569:    4    570:    4    571:    4    572:    4    573:    4    574:    4    575:    4
576:    3    577:    2    578:    1    579:    0    580:    -1   581:    -2   582:    -3   583:    -4
584:    -4   585:    -4   586:    -4   587:    -4   588:    -4   589:    -4   590:    -4   591:    -4
592:    -4   593:    -3   594:    -2   595:    -1   596:    0    597:    1    598:    2    599:    3
600:    4    601:    5    602:    5    603:    5    604:    5    605:    5    606:    5    607:    5
608:    5    609:    5    610:    5    611:    5    612:    4    613:    3    614:    2    615:    1
616:    0    617:    -1   618:    -2   619:    -3   620:    -4   621:    -5   622:    -5   623:    -5
624:    -5   625:    -5   626:    -5   627:    -5   628:    -5   629:    -5   630:    -5   631:    -5
632:    -5   633:    -4   634:    -3   635:    -2   636:    -1   637:    0    638:    1    639:    2
640:    3    641:    4    642:    5    643:    6    644:    6    645:    6    646:    6    647:    6
648:    6    649:    6    650:    6    651:    6    652:    6    653:    6    654:    6    655:    6
656:    5    657:    4    658:    3    659:    2    660:    1    661:    0    662:    -1   663:    -2
664:    -3   665:    -4   666:    -5   667:    -6   668:    -6   669:    -6   670:    -6   671:    -6
672:    -6   673:    -6   674:    -6   675:    -6   676:    -6   677:    -6   678:    -6   679:    -6
680:    -6   681:    -5   682:    -4   683:    -3   684:    -2   685:    -1   686:    0    687:    1
688:    2    689:    3    690:    4    691:    5    692:    6    693:    7    694:    7    695:    7
696:    7    697:    7    698:    7    699:    7    700:    7    701:    7    702:    7    703:    7
704:    7    705:    7    706:    7    707:    7    708:    6    709:    5    710:    4    711:    3
712:    2    713:    1    714:    0    715:    -1   716:    -2   717:    -3   718:    -4   719:    -5
720:    -6   721:    -7   722:    -7   723:    -7   724:    -7   725:    -7   726:    -7   727:    -7
728:    -7   729:    -7   730:    -7   731:    -7   732:    -7   733:    -7   734:    -7   735:    -7
736:    -7   737:    -6   738:    -5   739:    -4   740:    -3   741:    -2   742:    -1   743:    0
744:    1    745:    2    746:    3    747:    4    748:    5    749:    6    750:    7    751:    8
752:    8    753:    8    754:    8    755:    8    756:    8    757:    8    758:    8    759:    8
760:    8    761:    8    762:    0    763:    0    764:    0    765:    0    766:    0    767:    0
```

c-values:

```
768:    -1   769:    1    770:    1    771:    1    772:    1    773:    -1   774:    -1   775:    -1
776:    -1   777:    -1   778:    1    779:    1    780:    1    781:    1    782:    1    783:    1
784:    1    785:    1    786:    -1   787:    -1   788:    -1   789:    -1   790:    -1   791:    -1
792:    -1   793:    -1   794:    -1   795:    1    796:    1    797:    1    798:    1    799:    1
800:    1    801:    1    802:    1    803:    1    804:    1    805:    1    806:    1    807:    -1
808:    -1   809:    -1   810:    -1   811:    -1   812:    -1   813:    -1   814:    -1   815:    -1
816:    -1   817:    -1   818:    -1   819:    -1   820:    1    821:    1    822:    1    823:    1
824:    1    825:    1    826:    1    827:    1    828:    1    829:    1    830:    1    831:    1
832:    1    833:    1    834:    1    835:    1    836:    -1   837:    -1   838:    -1   839:    -1
840:    -1   841:    -1   842:    -1   843:    -1   844:    -1   845:    -1   846:    -1   847:    -1
848:    -1   849:    -1   850:    -1   851:    -1   852:    -1   853:    -1   854:    1    855:    1
856:    1    857:    1    858:    1    859:    1    860:    1    861:    1    862:    1    863:    1
864:    1    865:    1    866:    1    867:    1    868:    1    869:    1    870:    1    871:    1
872:    1    873:    1    874:    -1   875:    -1   876:    -1   877:    -1   878:    -1   879:    -1
880:    -1   881:    -1   882:    -1   883:    -1   884:    -1   885:    -1   886:    -1   887:    -1
888:    -1   889:    -1   890:    -1   891:    -1   892:    -1   893:    -1   894:    -1   895:    1
896:    1    897:    1    898:    1    899:    1    900:    1    901:    1    902:    1    903:    1
904:    1    905:    1    906:    1    907:    1    908:    1    909:    1    910:    1    911:    1
912:    1    913:    1    914:    1    915:    1    916:    1    917:    1    918:    1    919:    -1
920:    -1   921:    -1   922:    -1   923:    -1   924:    -1   925:    -1   926:    -1   927:    -1
928:    -1   929:    -1   930:    -1   931:    -1   932:    -1   933:    -1   934:    -1   935:    -1
936:    -1   937:    -1   938:    -1   939:    -1   940:    -1   941:    -1   942:    -1   943:    -1
944:    1    945:    1    946:    1    947:    1    948:    1    949:    1    950:    1    951:    1
952:    1    953:    1    954:    1    955:    1    956:    1    957:    1    958:    1    959:    1
960:    1    961:    1    962:    1    963:    1    964:    1    965:    1    966:    1    967:    1
968:    1    969:    1    970:    1    971:    1    972:    -1   973:    -1   974:    -1   975:    -1
976:    -1   977:    -1   978:    -1   979:    -1   980:    -1   981:    -1   982:    -1   983:    -1
984:    -1   985:    -1   986:    -1   987:    -1   988:    -1   989:    -1   990:    -1   991:    -1
992:    -1   993:    -1   994:    -1   995:    -1   996:    -1   997:    -1   998:    -1   999:    -1
1000:   -1   1001:   1    1002:   1    1003:   1    1004:   1    1005:   1    1006:   1    1007:   1
1008:   1    1009:   1    1010:   1    1011:   1    1012:   1    1013:   1    1014:   1    1015:   1
1016:   1    1017:   1    1018:   0    1019:   0    1020:   0    1021:   0    1022:   0    1023:   0
```

2. **Include your Python simulator code here:**

```
# MEDP Simulator    #
# Project 3         #
# Group 6, ECE 366  #

import math

# All my registers (0-18) #
# Initialized to equal 0  #
Registers = [0] * 18
# 16 = sum                #
# 17 = tempReg            #

# All of our instructions          #
# Used to write into writefile.txt #
Instructions = []

# Memory[]   #
Mem = {}                    # Dictionary
memAddr = range(0, 1024, 1) # 16-bit memory addresses
for i in memAddr:
 # Initialzing every Memory Address to 0
 Mem[i] = 0

# All the supported functions #
# Used for printing what instructions happen in writefile.txt #
func = {}
func["00"] = "j"
func["0100"] = "bgeqz"
func["0101"] = "incr"
func["0110"] = "ld"
func["0111"] = "st"
func["1000"] = "mult"
func["1001"] = "div2"
func["1010"] = "stsum"
func["1011"] = "sub"
func["1100"] = "add"
func["1101"] = "stb"
func["1110"] = "appb"
func["1111"] = "initb"

# ************************************** #
```

```
# twos_comp:
# Returns the two's comp of a given value
def twos_comp(val, bits):

 if (val & (1 << (bits - 1))) != 0:
   val = val - (1 << bits)
 return val

# formatChecker:
# Checks if given binary instr is good
def formatChecker(instr):
 checker = False

 if any(n > '1' or n < '0' for n in instr): # if not 0 or 1
   print("Binary only uses 1's and 0's")
 elif len(instr) != 8: # if too long or short
   print("Only 8-bit instructions are supported")
 else:
   checker = True
  return checker

# PC:
# counts the number of written instructions in
# the .asm file provided
def PC(dataFile):
 PC = 0
 for instr in dataFile:
   PC = PC + 1
 return PC

# simulator:
# actually simulates the assembly code
def simulator(dataFile, functions):
 # acts like a for loop in C++/C/Java
 # did this so we can easily go backwards and forwards
 # where i sort of acts like pc
 i = 0
 instructionCount = 0
 while i < len(dataFile):

   # For debug purposes #
   # print("i " + str(i) + ", r10: " + str(Registers[10]))
```

```
#if (i == 206):
#  break
#if (instructionCount == 3779):
  # break

# Getting the instruction
instr = dataFile[i]
instr = instr.replace(" ", "") # Gets rid of spaces
instr = instr.replace("\n", "") # Gets rid of newlines

# If the format isn't right for at least one instruction
if (formatChecker(instr) == False):
  print("\nAborting...")
  exit()

# Incrementing now
# If we don't then our jump breaks
i = i + 1

# Instruction counter, does not equal i because
# i changes with each jump and branch
instructionCount = instructionCount + 1

# j imm
# works more like branch
if(instr[0:2] == "00"):
  # 00 iiiiii
  # imm = [-64, 63], done by multiplying the twos_comp by 2
  # PC = imm + PC
  imm = instr[2:8]
  newImm = twos_comp(int(instr[2:8], 2), 6)
  newImm = newImm * 2

  # So we can print what instructions happen #
  Instructions.append(functions[instr[0:2]] + " " + str(newImm))

  i = i + (newImm) - 1 # jumping

# bgeqz Rx
if(instr[0:4] == "0100"):
  # 0100 xxxx
  # if (Rx >= 0) PC = PC + 2
  # else PC++
```

```
    Rx = int(instr[4:8], 2)

    # So we can print what instructions happen #
    Instructions.append(functions[instr[0:4]] + " $" + str(Rx))

    if (int(Registers[Rx]) >= 0):
      i = i + 1 # Only 1 because we already incremented i by 1 in the very
beginning of this loop

  # incr Rx, imm
  if(instr[0:4] == "0101"):
    # 0101 xxx i
    # Rx = Rx + i
    # i = -1 or 1

    Rx = int(instr[4:7], 2)
    imm = int(instr[7:8], 2)
    # So we can print what instructions happen #
    Instructions.append(functions[instr[0:4]] + " $" + str(Rx) + ", " +
str(imm))

    if (imm == 0):
      Registers[Rx] = Registers[Rx] - 1
    if (imm == 1):
      Registers[Rx] = Registers[Rx] + 1

  # ld Rx, (Ry)
  if(instr[0:4] == "0110"):
    # 0110 xx, yy
    # xx = [6, 9]
    # yy = [0, 3]
    Rx = int(instr[4:6], 2) + 6
    Ry = int(instr[6:8], 2)

    # So we can print what instructions happen #
    Instructions.append(functions[instr[0:4]] + " $" + str(Rx) + ", ($" +
str(Ry) + ")")

    Registers[Rx] = Mem[Registers[Ry]]

  # st Rx, (Ry)
  if(instr[0:4] == "0111"):
```

```
      # Mem[Ry] = Rx
      # 0111 xx, yy
      # xx = [6, 9]
      # yy = [0, 3]
      Rx = int(instr[4:6], 2) + 6
      Ry = int(instr[6:8], 2)

      # So we can print what instructions happen #
      Instructions.append(functions[instr[0:4]] + " $" + str(Rx) + ", ($" +
str(Ry) + ") Addr:" + str(Registers[Ry]) + " = " +  str(Registers[Rx]))

      Mem[Registers[Ry]] = Registers[Rx]

   # mult Rx, Ry
   if(instr[0:4] == "1000"):
      # 1000 xx yy
      # xx, yy = [4-7]
      Rx = int(instr[4:6], 2) + 4
      Ry = int(instr[6:8], 2) + 4

      # So we can print what instructions happen #
      Instructions.append(functions[instr[0:4]] + " $" + str(Rx) + ", $" +
str(Ry))

      Registers[Rx] = Registers[Rx] * Registers[Ry]

   # div2 Rx
   if(instr[0:4] == "1001"):
      # Rx = Rx / 2
      # R15 = Remainder
      # 1001 xxxx
      Rx = int(instr[4:8], 2)

      # So we can print what instructions happen #
      Instructions.append(functions[instr[0:4]] + " $" + str(Rx))

      R15 = Registers[Rx] % 2
      R15 = R15 * -1
      Registers[15] = R15

      Registers[Rx] = math.floor(Registers[Rx] / 2)

   # stsum Rx
```

```
  if(instr[0:4] == "1010"):
    # Rx = sum
    # 1010 xxxx
    Rx = int(instr[4:8], 2)

    # So we can print what instructions happen #
    Instructions.append(functions[instr[0:4]] + " $" + str(Rx))

    Registers[Rx] = Registers[16]
    # Reset sum
    Registers[16] = 0

  # sub Rx
  if(instr[0:4] == "1011"):
    # 1011 xxxx
    # sum = sum - Rx
    Rx = int(instr[4:8], 2)

    # So we can print what instructions happen #
    Instructions.append(functions[instr[0:4]] + " $" + str(Rx))

    Registers[16] = Registers[16] - Registers[Rx]

  # add Rx
  if(instr[0:4] == "1100"):
    # 1100 xxxx
    # sum = sum + Rx
    Rx = int(instr[4:8], 2)

    # So we can print what instructions happen #
    Instructions.append(functions[instr[0:4]] + " $" + str(Rx))

    Registers[16] = Registers[16] + Registers[Rx]

  # stb Rx
  if(instr[0:4] == "1101"):
    # Rx = tempReg
    # 1101 xxxx
    Rx = int(instr[4:8], 2)

    # So we can print what instructions happen #
    Instructions.append(functions[instr[0:4]] + " $" + str(Rx))
```

```
      # THIS ALWAYS TAKES THE twos_comp
      # EVEN IF tempReg is 4, 8, 16 BITS
      Registers[Rx] = twos_comp(int(Registers[17], 2), len(Registers[17]))
      # print(Registers[Rx])

    # appb bbbb
    if(instr[0:4] == "1110"):
      # tempReg = [tempReg][bbbb]
      # 1110 bbbb

      # So we can print what instructions happen #
      Instructions.append(functions[instr[0:4]] + " " + instr[4:8])

      Registers[17] = str(Registers[17]) + instr[4:8]
      # print(Registers[17])

    # initb bbbb
    if(instr[0:4] == "1111"):
      # tempReg = bbbb
      # 1111 bbbb

      # So we can print what instructions happen #
      Instructions.append(functions[instr[0:4]] + " " + instr[4:8])

      Registers[17] = instr[4:8]
      # print(Registers[17])

  return instructionCount



# ****************************************** #
# *************** "M A I N" *************** #
print("Welcome to MEDP Simulator!\n\n")
dataFile = str(input("Enter the filename with the MEDP instructions: "))
printInstr = str(input("\nTranslate instructions into writefile.txt (y/n)? "))
print("\n\tMEDP Simulator\n\n")
try:
  with open(dataFile) as myFile:

    # Open file to write in
    w = open('writefile.txt', 'w+')
    dataFile = myFile.readlines()    # Read my file
```

```
# Call the simulator
# instructionCount is the amount of instructions done
instructionCount = simulator(dataFile, func)

# Basically a count of the instructions written in the file
pc = PC(dataFile)

# Prints our registers
print("Registers: \n")
count = 0
for reg in Registers:
  # our special registers
  if (count == 16):
    register = "sum:"
  elif (count == 17):
    register = "tempReg:"
  else:
    # regular registers
    register = "$" + str(count) + ":"

  # Formats so they print nicely
  print("{:<8} {:<5}".format(register, str(Registers[count])))
  count = count + 1

print("PC: " + str(pc))

# printing the amount of instructions done
print("\nInstruction Count: " + str(instructionCount) + "\n")

# Writes instructions into a file
if(printInstr.lower() == 'y'):
  w.write("Instructions:\n\n")
  for instr in Instructions:
    w.write(str(instr) + "\n")          # Write instructions in the file
  w.write("--------------------\n")

w.write("Memory Content:\n\n")

# Writes memory into a file
# Full-screened txt file looks the best #
printCounter = 0
for addr, content in Mem.items():
```

```
    printCounter = printCounter + 1
    addr = str(addr) + ":"
    # Formats so it prints nicely while fullscreened
    writeMe = "{:<8} {:<5}".format(str((addr)), str(content))

    # Just limits the amount of addresses per line to 8, which looks the best
when you give the writefile.txt all the space you can on repl.it
    if (printCounter == 8):
      w.write(writeMe)
      w.write("\n")
      printCounter = 0
    else:
      w.write(writeMe)

  w.close()                       # Close the file
except IOError:
 # This happens if the filename is wrong
 print("Uh oh, this file either does not exist or we can not reach
it...\n\nAborting...")
 exit()
```