Nathanael Bhooshi
Daniel Hodczak
ECE 366
Project 3: ISA Design for Perceptron

Activity Log:
I) Table of group activity

Worked on the project throughout the week Did hardware last week

II) Individual activity list

Daniel Dodczak: Coded in python the perceptron and worked on hardware	Nathanael Bhooshi Coded in python the perceptron and worked on hardware

III) Group work reflection
a) How does this group differ from your previous one?
This group one of our group members wanted to do his own activity for the project
b) List out one new or important thing that you have learned from this group.
I learned how to create hardware for a new cpu
c) If you were to do this project entirely by yourself, how would it be different?
I would of used less functions

Part A) ISA intro, Q&A

Introduction & Instruction list:

Instruction	Op	InputControl[2:0]	InputAControl[2:0]	InputBControl[2:0]	RegWrite	MemRegF	FlagControl	SetStatus	IncCounter	ALUControl	yALUControl[2:0]	MemWrite	BypassMem	MemInc
set Rg	0000 00XX	10	10	xx	1	0	XX	0	1	0000	1111	0	1	X
add1 Rg	0000 01XX	10	10	xx	1	0	XX	0	0	0100	1111	0	1	X
sub1 Rg	0000 10XX	10	10	XX	1	0	XX	0	0	0101	1111	0	1	X
seqc gg	0000 11XX	XX	XX	XX	0	1	01	1	0	0000	XXXX	0	1	0
addlc gg	0001 01XX	XX	XX	XX	0	1	XX	0	0	1000	XXXX	1	X	1
swf Rg	0001 10XX	XX	XX	XX	0	1	XX	0	0	0000	XXXX	1	X	0
lwf Rg	0001 11XX	00	XX	XX	1	0	XX	0	0	0000	1111	0	0	X
sneg Rg	0010 00XX	XX	10	XX	0	0	10	1	0	1001	XXXX	0	X	X
snotneg Rg	0010 01XX	XX	10	XX	0	0	11	1	0	1001	XXXX	0	X	X
lwfaddreg	0010 10XX	00	XX	00	1	1	XX	0	0	0000	0001	0	0	X
lwfsubreg	0010 11XX	00	XX	00	1	1	XX	0	0	0000	0010	0	0	X
lwmultinc	0011 00XX	00	10	00	1	0	XX	0	0	0000	0011	0	0	X
lwprev	0011 01XX	00	10	00	1	0	XX	0	0	0000	0101	0	0	X
lwmult	0011 10XX	00	10	00	1	0	XX	0	0	0000	0011	0	0	X
hold	0011 1100	11	00	XX	1	X	XX	0	0	0000	1111	0	1	X
addback	0011 1101	00	11	XX	1	X	XX	0	0	0000	1111	0	1	X
add Ra, Rb	0100 XXXX	01	01	01	1	0	XX	0	0	0001	1111	0	1	X
swinc Ra, Rb	0101 XXXX	10	01	01	1	0	XX	0	0	0000	1110	1	1	0
init aabb	0110 XXXX	XX	XX	XX		X	XX	0	1	XXXX	XXXX	0	x	X
loop imm	100X XXXX	XX	XX	xx		X	XX	0	0	XXXX	XXXX	0	X	X
jif imm	110X XXXX	XX	XX	xx		X	XX	0	0	XXXX	XXXX	0	X	X
j imm	1100 0000	XX	XX	xx		X	XX	0	0	XXXX	XXXX	0	X	X

Instruction	Functionality	Encoding Form	MC Example	asm example
set Rg	Rg = result	0000 00 gg	0000 0000	set $0
add1 Rg	Rg = Rg + 1	0000 01 gg	0000 0100	add1 $0
sub1 Rg	Rg = Rg - 1	0000 10 gg	0000 1000	sub1 $0

seqc gg	if mem(4 +gg) == mem(8 +gg)
	  status = 1;
	  mem(gg) = 0
	else:
	  status = 0;
	  mem(gg) += 1;
		0000 11 gg	0000 1100	seqc 0

add1c gg	mem(8+gg) += 1	0001 01 gg	0001 0100	add1c 0
swf Rg	mem(gg) = $0	0001 10gg	0001 1000	swf 0
lwf Rg	$0 = mem(gg)	0001 11gg	0001 1100	lwf 0

sneg Rg	if Rg < 0:
	  status = 1
	else:
	  status = 0
		0010 00gg	0010 0010	sneg $2

snotneg Rg	if Rg > 0:
	  status = 1
	else:
	  status = 0
		0010 01gg	0010 0110	snotneg $2

lwfaddreg	$0 = $0 + mem(gg)	0010 10gg	0010 1000	lwfaddreg 0
lwfsubreg	$0 = $0 - mem(gg)	0010 11gg	0010 1100	lwfsubreg 0
lwmult	$0 = $0 * mem(Rg);	0011 00gg	0011 0010	lwmultinc $2
lwprev	$0 = mem(Rg-1)	0011 01gg	0011 0110	lwprev $2
lwmult	$0 = $0 * mem(Rg)	0011 10gg	0011 1010	lwmult $2
hold	result = $0	0011 1100	0011 1100	hold
addback	$0 = result	0011 1101	0011 1101	addback
add Ra, Rb	Ra = Ra + Rb	0100 aabb	0100 0001	add $0, $1
swinc Ra, R	mem(Ra) = Rb; Ra++;	0101 aabb	0101 0110	swinc $2, $1

init aabb	init = init | (aabb << 4 * counter
	if counter == 3:
	  result = temp;
	  init = 0;
	  counter = 0;
	else:
	  counter++;
		0110 aabb	0110 1111	init F

loop imm	LUT[imm] = pc + 4	100i iiii	1000 0000	loop:
jif imm	if (status): pc = LUT[imm];	110i iiii	1010 0000	jif loop
j imm	pc = LUT[imm]	111i iiii	1100 0000	j loop

Name: 110%

Philosophy: was to create a cpu that can perform the same task but faster pc count

Goals: decrease the amount of instructions we had to use.

Significant features: we can combine multiple instructions into 1 instruction

What was done: We combined multiple instructions into 1 instruction

Main limitations:the amount of registers and bits we had to work with

Compromises: making the multiplication requires more resources

Register design:

Registers		InputControl[2:0]		OutputControl[2:0]		ALUControl[3:0]	
Name	Number	Encoding	Meaning	Encoding	Meaning	A	0000
$0	0	00	$0	00	$0	A+B	0001
$1	1	01	$Ra	01	$Rb	A-B	0010
$2	2	10	$Rg	10	$Rg	A*B	0011
$3	3	11	result	11	result	A+1	0100
result	4					A-1	0101
						A<<B	0110
						B+1	1110
						A+8	1000
						0-A	1001
						B	1111

There are 5 registers. In order to account for the different unique instructions that I didn't think I'd have so much difficulty in hardware implementing, designating what register will be written to is decided by INPC. Likewise, what registers SrcA and SrcB output is determined by OutputAControl and OutputBControl, respectively. The fact that the registers the register file puts out is controlled by the Control Unit holds a risk for race conditions; for this implementation to work, if at all, the register file must wait for the control unit to output its signals before it outputs its own signals, inheritly adding delay to the computer.
There is no A3 because of the 8-bit instructions; there will never be a third register in an instruction. Rb and Rg are synonymous, however Rg is just used when there is no Ra from an instruction to decrease ambiguity. Addressing of the 5 registers uses 2 bits for registers $0 to $3. 'Result' is a special register that is only used in a couple of functions for additional functionality.

Branch design:
No branches are supported. Target addresses are achieved using direct-addressing (jump), rather than relative offset addressing. Because of the 8-bit limitation, it is rather difficult to direct address all of the potential PC addresses for a program of reasonable length in this architecture. The J op is 3 bits in length, leaving 5 bits of addressing. Without any workarounds, only up to PC=31 would be able to jumped to. Therefore, it was necessary to implement jump addressing with a lookup table that returns addresses larger than 5 bits in length.
\paragraph{} This architecture runs in two passes: on the first pass, it tests for any 'loop' instructions in the instruction set. Any other instruction is completely ignored. Once a 'loop' instruction is detected, it stores a pairing of the first 5 bits of the instruction, the 'jjjjj', and the current pc value. Therefore, up to 32 possible pairings of {jjjjj: pc} are possible, and the PC can jump to any 16-bit address, but there are only 32 discrete locations it can jump to.

It is beyond the scope of this project to determine how machine knows the PC has run out of the instruction memory. Therefore, I left it as a cloud at the top of the datapath architecture. On the second path, if a 'j' or 'jif' operation is called, the pc will be set to whatever value is held for the first 5 bits of that instruction in the jump memory.

Data memory addressing modes:
The instructions that access data memory are 'seqc', 'add1c', 'swf', 'lwf','lwfaddreg', 'lwfsubreg', 'lwmult', and 'swinc'. There are so many different instructions to maximize dynamic memory efficiency of the script. Instructions that include 'f' in the name are \emph{fast} instructions, so they use direct word-to-memory addressing. These instructions use the first 2 bits of the word to access the first 4 words in memory, rather than referencing those bits against the register file. The 'inc' instructions increment the register after it has been used to address the data. The 'add', 'sub', and 'mult' instructions add, subtract, or multiply the memory to/from $0, respectively. All of these special, complex instructions work to decrease the dynamic instruction count.

Results for Perceptron:

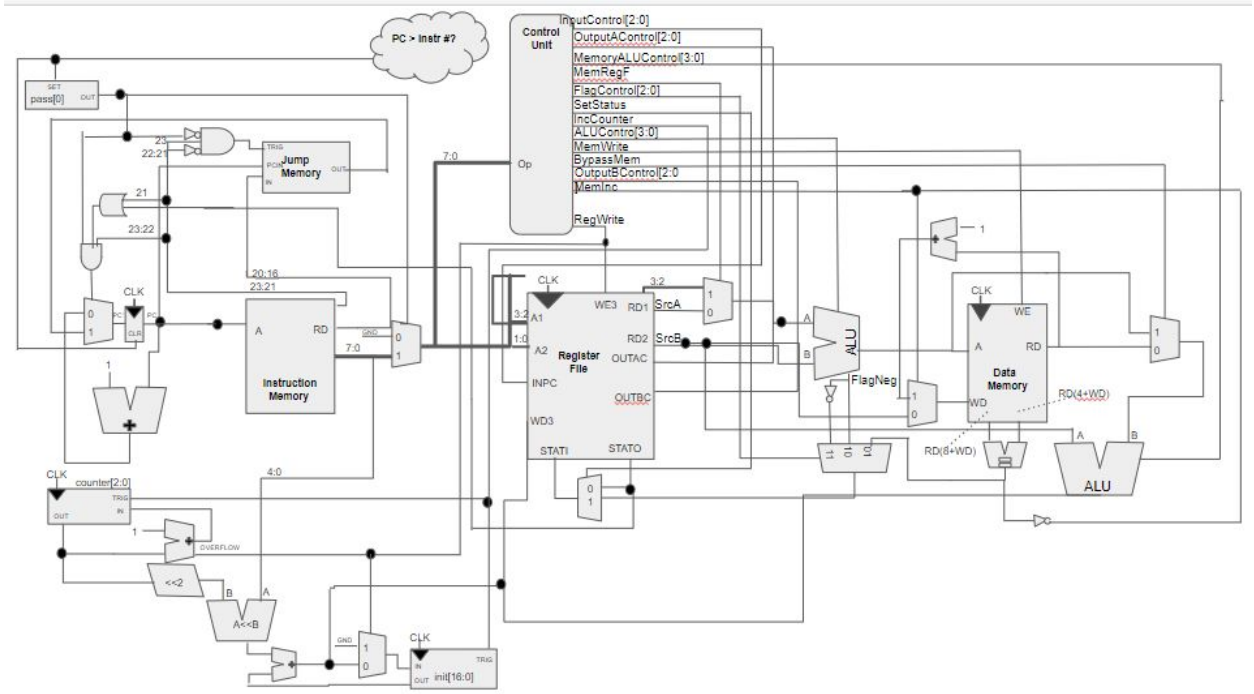
```
(venv) [danielh@daniel-pc Project_3]$ cat mcv1.txt | python ISA-sim.py -showcode=False
Data[0]: 4
Data[1]: -5
Data[2]: 5
Data[3]: -6
Dynamic instruction count:3045
[]
(venv) [danielh@daniel-pc Project_3]$
```

```
(venv) [danielh@daniel-pc Project_3]$ cat mcv2.txt | python ISA-sim.py -showcode=False
Data[0]: 0
Data[1]: 15
Data[2]: -17
Data[3]: 80
Dynamic instruction count:7425
[]
(venv) [danielh@daniel-pc Project_3]$
```

For V1 we have 3045 instruction counts and V2 we have 7425instruction counts. It would be V1 because that used less memory access.

Part B) Hardware sketches

CPU Datapath design:

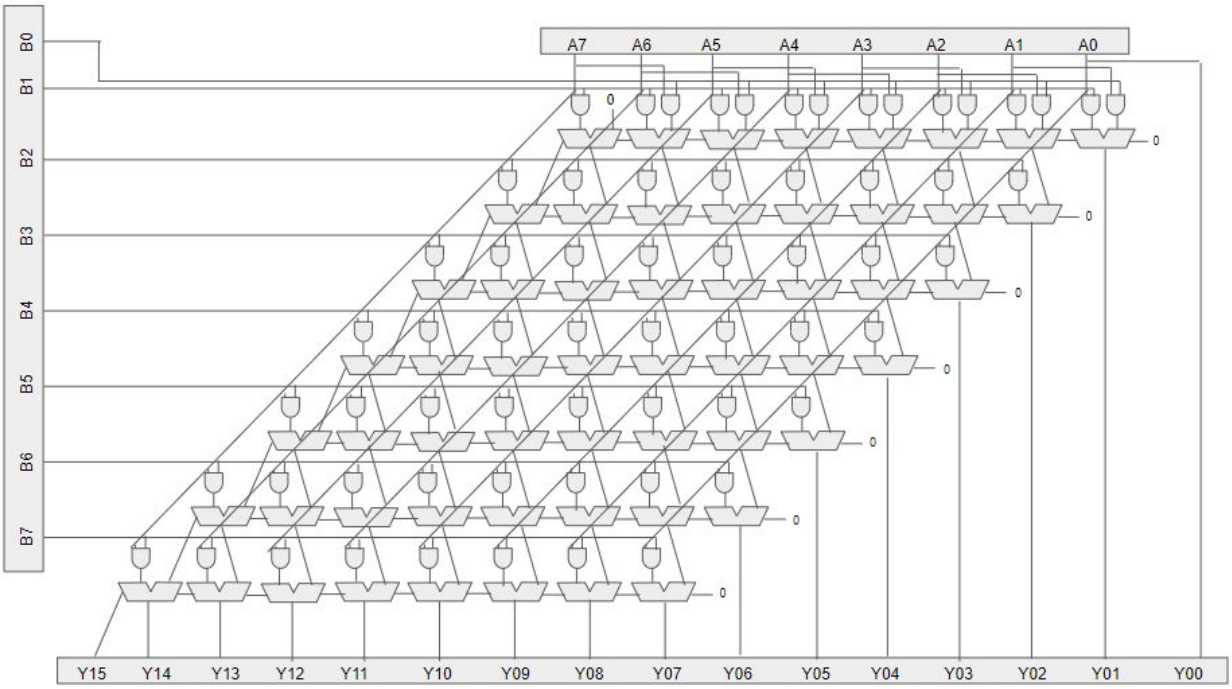


Control logic design:

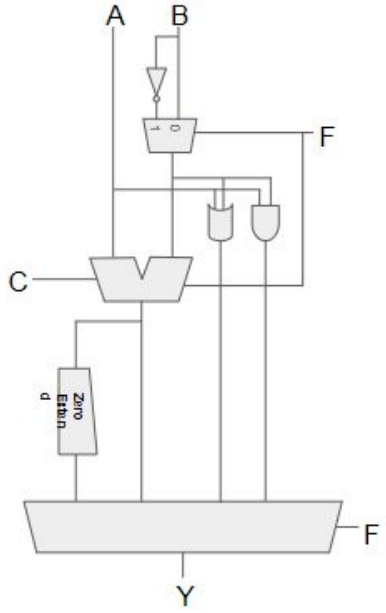
Registers		InputControl[2:0]		OutputControl[2:0]		ALUControl[3:0]	
Name	Number	Encoding	Meaning	Encoding	Meaning	A	0000
$0	0	00	$0	00	$0	A+B	0001
$1	1	01	$Ra	01	$Rb	A-B	0010
$2	2	10	$Rg	10	$Rg	A*B	0011
$3	3	11	result	11	result	A+1	0100
result	4					A-1	0101
						A<<B	0110
						B+1	1110
						A+8	1000
						0-A	1001
						B	1111

ALU schematic:

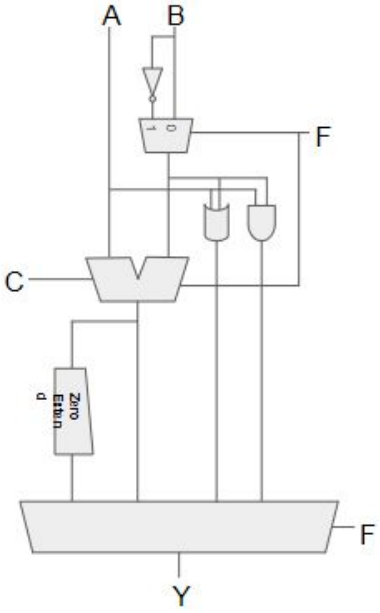
Multiply ALU

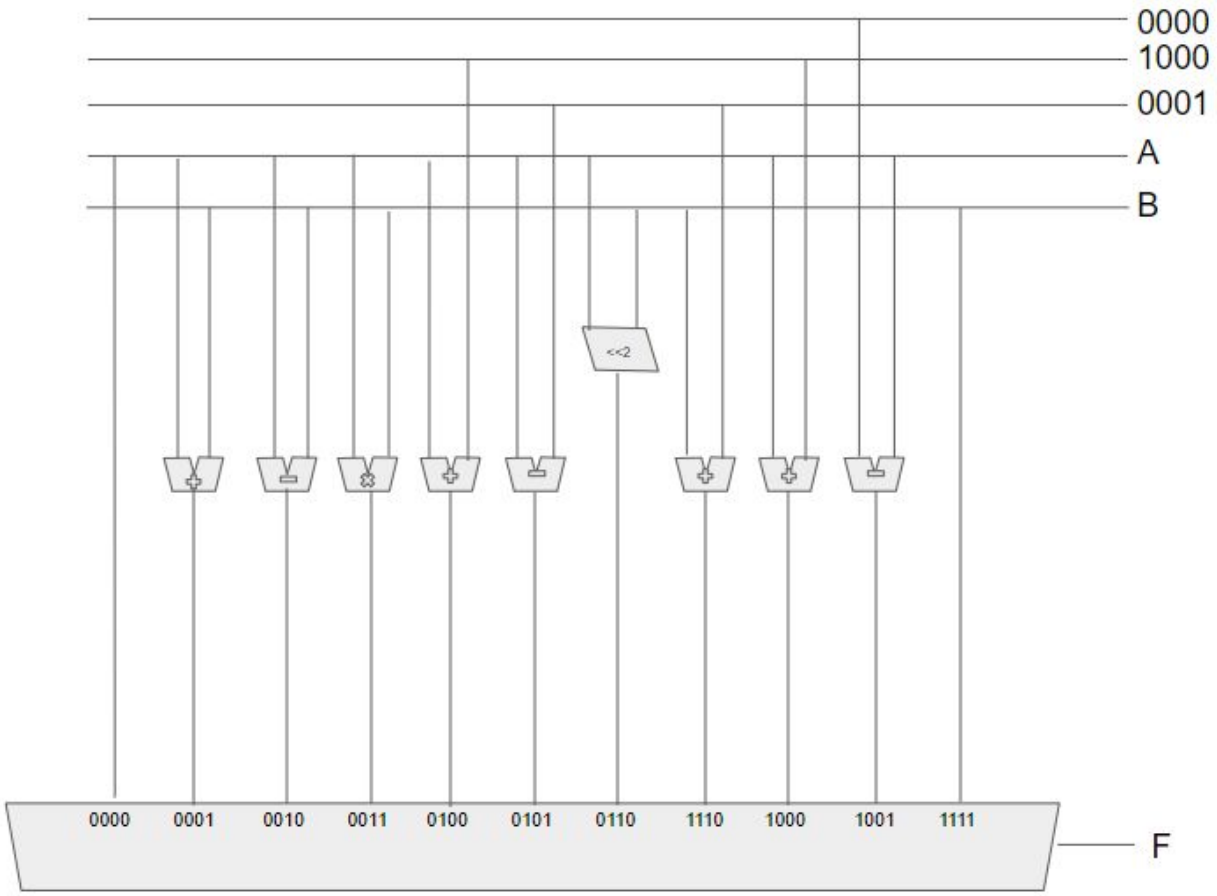


ADD
F2: 0
F1,0: 10



SUB
F2: 1
F1,0: 10





Part C) Software Package: Perceptron code + Python simulator

V1:
Assembly:

```
init 8
init 0
init 0
init 0
set $3 # $3 = 8
init 4
init 6
init 0
init 0
set $2
swinc $2, $3 # mem(8) = N = 100
init 0
init 0
```

```
init 1
init 0
set $2 # $2 = 256
init 0
init 0
init 2
init 0
set $3 # $3 = 512

loop_rightup:
add1c 1 # mem(9)++

loop_right:

seqc 1
jif loop_up
seqc 0
jif part_2


add1 $0
swinc $0, $2
swinc $1, $3

j loop_right

loop_up:

seqc 1
jif loop_leftdown
seqc 0
jif part_2

add1 $1
swinc $0, $2
swinc $1, $3

j loop_up


loop_leftdown:
add1c 1 # mem(9)++
```

```
loop_left:


seqc 1
jif loop_down
seqc 0
jif part_2

sub1 $0
swinc $0, $2
swinc $1, $3

j loop_left

loop_down:

seqc 1
jif loop_rightup
seqc 0
jif part_2

sub1 $1
swinc $0, $2
swinc $1, $3

j loop_down

part_2:

init 0
init 0
init 1
init 0
set $2 # $2 = 256
init 0
init 0
init 2
init 0
set $3 # $3 = 512


init 1
init 0
```

```
init 0
init 0
set $0
swf 0 # mem(0) = a = 1

init 1
init 0
init 0
init 0
set $0
swf 1 # mem(0) = b = 1


init 5
init 0
init 0
init 0
set $0
swf 2 # mem(2) = A = 5

init A
init F
init F
init F
set $0
swf 3 # mem(3) = B = -6


improve_ab:

seqc 0 # mem(4) = mem(8)?
jif done

# Is f(x,y) negative?

lwf 0 # $0 = a
lwmult $2 # $0 = ax[i]
hold

lwf 1 # $0 = b
lwmult $3 # $0 = by[i]
addback
```

```
slt $0

# F[X,Y] negative?

lwf 2 # $0 = A
lwmult $2 # $0 = Ax[i]
add1 $2 # $2 += 1
hold

lwf 3 # $0 = B
lwmult $3 # $0 = By[i]
add1 $3 # $3 += 1
addback


jif pred_neg
j pred_notneg


# c[i] is negative.
pred_neg:
slt $0
jif improve_ab

lwprev $2 # $0 = x[i]
lwfaddreg 0 # $0 = a + x[i]
swf 0 # a = $0

lwprev $3 # $0 = y[i]
lwfaddreg 1 # $0 = b + y[i]
swf 1 # b = $0

j improve_ab

# c[i] is not negative.
pred_notneg:
snotneg $0
jif improve_ab

lwprev $2 # $0 = x[i]
lwfsubreg 0 # $0 = a - x[i]
swf 0 # a = $0
```

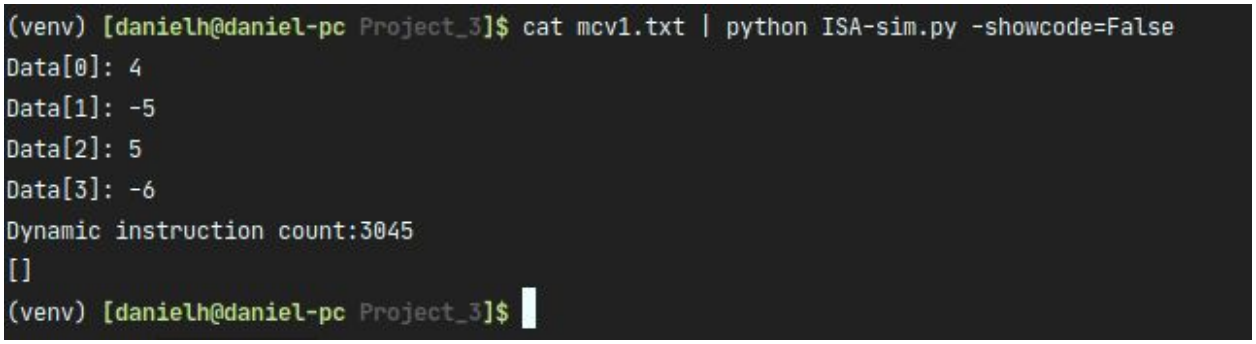
```
lwprev $3 # $0 = y[i]
lwfsubreg 1 # $0 = b - y[i]
swf 1 # b = $0

j improve_ab



done:


Machine:
0 ['init 8', '01101000']
1 ['init 0', '01100000']
2 ['init 0', '01100000']
3 ['init 0', '01100000']
4 ['set $3 # $3 = 8', '00000011']
5 ['init 4', '01100100']
6 ['init 6', '01100110']
7 ['init 0', '01100000']
8 ['init 0', '01100000']
9 ['set $2', '00000010']
10 ['swinc $2, $3 # mem(8) = N = 100', '01011011']
11 ['init 0', '01100000']
12 ['init 0', '01100000']
13 ['init 1', '01100001']
14 ['init 0', '01100000']
15 ['set $2 # $2 = 256', '00000010']
16 ['init 0', '01100000']
17 ['init 0', '01100000']
18 ['init 2', '01100010']
19 ['init 0', '01100000']
20 ['set $3 # $3 = 512', '00000011']
21 ['loop_rightup:', '10000000']
22 ['add1c 1 # mem(9)++', '00010101']
23 ['loop_right:', '10000001']
24 ['seqc 1', '00001101']
25 ['jif loop_up', '11000010']
26 ['seqc 0', '00001100']
27 ['jif part_2', '11000110']
28 ['add1 $0', '00000100']
29 ['swinc $0, $2', '01010010']
30 ['swinc $1, $3', '01010111']
```

31 ['j loop_right', '11100001']
32 ['loop_up:', '10000010']
33 ['seqc 1', '00001101']
34 ['jif loop_leftdown', '11000011']
35 ['seqc 0', '00001100']
36 ['jif part_2', '11000110']
37 ['add1 $1', '00000101']
38 ['swinc $0, $2', '01010010']
39 ['swinc $1, $3', '01010111']
40 ['j loop_up', '11100010']
41 ['loop_leftdown:', '10000011']
42 ['add1c 1 # mem(9)++', '00010101']
43 ['loop_left:', '10000100']
44 ['seqc 1', '00001101']
45 ['jif loop_down', '11000101']
46 ['seqc 0', '00001100']
47 ['jif part_2', '11000110']
48 ['sub1 $0', '00001000']
49 ['swinc $0, $2', '01010010']
50 ['swinc $1, $3', '01010111']
51 ['j loop_left', '11100100']
52 ['loop_down:', '10000101']
53 ['seqc 1', '00001101']
54 ['jif loop_rightup', '11000000']
55 ['seqc 0', '00001100']
56 ['jif part_2', '11000110']
57 ['sub1 $1', '00001001']
58 ['swinc $0, $2', '01010010']
59 ['swinc $1, $3', '01010111']
60 ['j loop_down', '11100101']
61 ['part_2:', '10000110']
62 ['init 0', '01100000']
63 ['init 0', '01100000']
64 ['init 1', '01100001']
65 ['init 0', '01100000']
66 ['set $2 # $2 = 256', '00000010']
67 ['init 0', '01100000']
68 ['init 0', '01100000']
69 ['init 2', '01100010']
70 ['init 0', '01100000']
71 ['set $3 # $3 = 512', '00000011']
72 ['init 1', '01100001']
73 ['init 0', '01100000']

74 ['init 0', '01100000']
75 ['init 0', '01100000']
76 ['set $0', '00000000']
77 ['swf 0 # mem(0) = a = 1', '00011000']
78 ['init 1', '01100001']
79 ['init 0', '01100000']
80 ['init 0', '01100000']
81 ['init 0', '01100000']
82 ['set $0', '00000000']
83 ['swf 1 # mem(0) = b = 1', '00011001']
84 ['init 5', '01100101']
85 ['init 0', '01100000']
86 ['init 0', '01100000']
87 ['init 0', '01100000']
88 ['set $0', '00000000']
89 ['swf 2 # mem(2) = A = 5', '00011010']
90 ['init A', '01101010']
91 ['init F', '01101111']
92 ['init F', '01101111']
93 ['init F', '01101111']
94 ['set $0', '00000000']
95 ['swf 3 # mem(3) = B = -6', '00011011']
96 ['improve_ab:', '10000111']
97 ['seqc 0 # mem(4) = mem(8)?', '00001100']
98 ['jif done', '11001010']
99 ['lwf 0 # $0 = a', '00011100']
100 ['lwmult $2 # $0 = ax[i]', '00111010']
101 ['hold', '00111100']
102 ['lwf 1 # $0 = b', '00011101']
103 ['lwmult $3 # $0 = by[i]', '00111011']
104 ['addback', '00111101']
105 ['slt $0', '00100000']
106 ['lwf 2 # $0 = A', '00011110']
107 ['lwmult $2 # $0 = Ax[i]', '00111010']
108 ['add1 $2 # $2 += 1', '00000110']
109 ['hold', '00111100']
110 ['lwf 3 # $0 = B', '00011111']
111 ['lwmult $3 # $0 = By[i]', '00111011']
112 ['add1 $3 # $3 += 1', '00000111']
113 ['addback', '00111101']
114 ['jif pred_neg', '11001000']
115 ['j pred_notneg', '11101001']
116 ['pred_neg:', '10001000']

117 ['slt $0', '00100000']
118 ['jif improve_ab', '11000111']
119 ['lwprev $2 # $0 = x[i]', '00110110']
120 ['lwfaddreg 0 # $0 = a + x[i]', '00101000']
121 ['swf 0 # a = $0', '00011000']
122 ['lwprev $3 # $0 = y[i]', '00110111']
123 ['lwfaddreg 1 # $0 = b + y[i]', '00101001']
124 ['swf 1 # b = $0', '00011001']
125 ['j improve_ab', '11100111']
126 ['pred_notneg:', '10001001']
127 ['snotneg $0', '00100100']
128 ['jif improve_ab', '11000111']
129 ['lwprev $2 # $0 = x[i]', '00110110']
130 ['lwfsubreg 0 # $0 = a - x[i]', '00101100']
131 ['swf 0 # a = $0', '00011000']
132 ['lwprev $3 # $0 = y[i]', '00110111']
133 ['lwfsubreg 1 # $0 = b - y[i]', '00101101']
134 ['swf 1 # b = $0', '00011001']
135 ['j improve_ab', '11100111']
136 ['done:', '10001010']

Screenshot:



V2:

Assembly:
init 8
init 0
init 0
init 0
set $3 # $3 = 8
init A
init F

```
init 0
init 0
set $2
swinc $2, $3 # mem(8) = N = 100
init 0
init 0
init 1
init 0
set $2 # $2 = 256
init 0
init 0
init 2
init 0
set $3 # $3 = 512

loop_rightup:
add1c 1 # mem(9)++

loop_right:

seqc 1
jif loop_up
seqc 0
jif part_2


add1 $0
swinc $0, $2
swinc $1, $3

j loop_right

loop_up:

seqc 1
jif loop_leftdown
seqc 0
jif part_2

add1 $1
swinc $0, $2
swinc $1, $3
```

```
j loop_up


loop_leftdown:
add1c 1 # mem(9)++

loop_left:


seqc 1
jif loop_down
seqc 0
jif part_2

sub1 $0
swinc $0, $2
swinc $1, $3

j loop_left

loop_down:

seqc 1
jif loop_rightup
seqc 0
jif part_2

sub1 $1
swinc $0, $2
swinc $1, $3

j loop_down

part_2:

init 0
init 0
init 1
init 0
set $2 # $2 = 256
init 0
init 0
init 2
```

```
init 0
set $3 # $3 = 512


init 1
init 0
init 0
init 0
set $0
swf 0 # mem(0) = a = 1

init 1
init 0
init 0
init 0
set $0
swf 1 # mem(0) = b = 1


init F
init E
init F
init F
set $0
swf 2 # mem(2) = A = -17

init 0
init 5
init 0
init 0
set $0
swf 3 # mem(3) = B = 80


improve_ab:

seqc 0 # mem(4) = mem(8)?
jif done

# Is f(x,y) negative?

lwf 0 # $0 = a
lwmult $2 # $0 = ax[i]
```

```
hold

lwf 1 # $0 = b
lwmult $3 # $0 = by[i]
addback

slt $0

# F[X,Y] negative?

lwf 2 # $0 = A
lwmult $2 # $0 = Ax[i]
add1 $2 # $2 += 1
hold

lwf 3 # $0 = B
lwmult $3 # $0 = By[i]
add1 $3 # $3 += 1
addback


jif pred_neg
j pred_notneg


# c[i] is negative.
pred_neg:
slt $0
jif improve_ab

lwprev $2 # $0 = x[i]
lwfaddreg 0 # $0 = a + x[i]
swf 0 # a = $0

lwprev $3 # $0 = y[i]
lwfaddreg 1 # $0 = b + y[i]
swf 1 # b = $0

j improve_ab

# c[i] is not negative.
pred_notneg:
snotneg $0
```

```
jif improve_ab

lwprev $2 # $0 = x[i]
lwfsubreg 0 # $0 = a - x[i]
swf 0 # a = $0

lwprev $3 # $0 = y[i]
lwfsubreg 1 # $0 = b - y[i]
swf 1 # b = $0

j improve_ab



done:
```

Machine:

```
0 ['init 8', '01101000']
1 ['init 0', '01100000']
2 ['init 0', '01100000']
3 ['init 0', '01100000']
4 ['set $3 # $3 = 8', '00000011']
5 ['init A', '01101010']
6 ['init F', '01101111']
7 ['init 0', '01100000']
8 ['init 0', '01100000']
9 ['set $2', '00000010']
10 ['swinc $2, $3 # mem(8) = N = 200', '01011011']
11 ['init 0', '01100000']
12 ['init 0', '01100000']
13 ['init 1', '01100001']
14 ['init 0', '01100000']
15 ['set $2 # $2 = 256', '00000010']
16 ['init 0', '01100000']
17 ['init 0', '01100000']
18 ['init 2', '01100010']
19 ['init 0', '01100000']
20 ['set $3 # $3 = 512', '00000011']
21 ['loop_rightup:', '10000000']
22 ['add1c 1 # mem(9)++', '00010101']
23 ['loop_right:', '10000001']
24 ['seqc 1', '00001101']
25 ['jif loop_up', '11000010']
```

26 ['seqc 0', '00001100']
27 ['jif part_2', '11000110']
28 ['add1 $0', '00000100']
29 ['swinc $0, $2', '01010010']
30 ['swinc $1, $3', '01010111']
31 ['j loop_right', '11100001']
32 ['loop_up:', '10000010']
33 ['seqc 1', '00001101']
34 ['jif loop_leftdown', '11000011']
35 ['seqc 0', '00001100']
36 ['jif part_2', '11000110']
37 ['add1 $1', '00000101']
38 ['swinc $0, $2', '01010010']
39 ['swinc $1, $3', '01010111']
40 ['j loop_up', '11100010']
41 ['loop_leftdown:', '10000011']
42 ['add1c 1 # mem(9)++', '00010101']
43 ['loop_left:', '10000100']
44 ['seqc 1', '00001101']
45 ['jif loop_down', '11000101']
46 ['seqc 0', '00001100']
47 ['jif part_2', '11000110']
48 ['sub1 $0', '00001000']
49 ['swinc $0, $2', '01010010']
50 ['swinc $1, $3', '01010111']
51 ['j loop_left', '11100100']
52 ['loop_down:', '10000101']
53 ['seqc 1', '00001101']
54 ['jif loop_rightup', '11000000']
55 ['seqc 0', '00001100']
56 ['jif part_2', '11000110']
57 ['sub1 $1', '00001001']
58 ['swinc $0, $2', '01010010']
59 ['swinc $1, $3', '01010111']
60 ['j loop_down', '11100101']
61 ['part_2:', '10000110']
62 ['init 0', '01100000']
63 ['init 0', '01100000']
64 ['init 1', '01100001']
65 ['init 0', '01100000']
66 ['set $2 # $2 = 256', '00000010']
67 ['init 0', '01100000']
68 ['init 0', '01100000']

69 ['init 2', '01100010']
70 ['init 0', '01100000']
71 ['set $3 # $3 = 512', '00000011']
72 ['init 1', '01100001']
73 ['init 0', '01100000']
74 ['init 0', '01100000']
75 ['init 0', '01100000']
76 ['set $0', '00000000']
77 ['swf 0 # mem(0) = a = 1', '00011000']
78 ['init 1', '01100001']
79 ['init 0', '01100000']
80 ['init 0', '01100000']
81 ['init 0', '01100000']
82 ['set $0', '00000000']
83 ['swf 1 # mem(0) = b = 1', '00011001']
84 ['init F', '01101111']
85 ['init E', '01101110']
86 ['init F', '01101111']
87 ['init F', '01101111']
88 ['set $0', '00000000']
89 ['swf 2 # mem(2) = A = -17', '00011010']
90 ['init 0', '01100000']
91 ['init 5', '01100101']
92 ['init 0', '01100000']
93 ['init 0', '01100000']
94 ['set $0', '00000000']
95 ['swf 3 # mem(3) = B = 80', '00011011']
96 ['improve_ab:', '10000111']
97 ['seqc 0 # mem(4) = mem(8)?', '00001100']
98 ['jif done', '11001010']
99 ['lwf 0 # $0 = a', '00011100']
100 ['lwmult $2 # $0 = ax[i]', '00111010']
101 ['hold', '00111100']
102 ['lwf 1 # $0 = b', '00011101']
103 ['lwmult $3 # $0 = by[i]', '00111011']
104 ['addback', '00111101']
105 ['slt $0', '00100000']
106 ['lwf 2 # $0 = A', '00011110']
107 ['lwmult $2 # $0 = Ax[i]', '00111010']
108 ['add1 $2 # $2 += 1', '00000110']
109 ['hold', '00111100']
110 ['lwf 3 # $0 = B', '00011111']
111 ['lwmult $3 # $0 = By[i]', '00111011']

112 ['add1 $3 # $3 += 1', '00000111']
113 ['addback', '00111101']
114 ['jif pred_neg', '11001000']
115 ['j pred_notneg', '11101001']
116 ['pred_neg:', '10001000']
117 ['slt $0', '00100000']
118 ['jif improve_ab', '11000111']
119 ['lwprev $2 # $0 = x[i]', '00110110']
120 ['lwfaddreg 0 # $0 = a + x[i]', '00101000']
121 ['swf 0 # a = $0', '00011000']
122 ['lwprev $3 # $0 = y[i]', '00110111']
123 ['lwfaddreg 1 # $0 = b + y[i]', '00101001']
124 ['swf 1 # b = $0', '00011001']
125 ['j improve_ab', '11100111']
126 ['pred_notneg:', '10001001']
127 ['snotneg $0', '00100100']
128 ['jif improve_ab', '11000111']
129 ['lwprev $2 # $0 = x[i]', '00110110']
130 ['lwfsubreg 0 # $0 = a - x[i]', '00101100']
131 ['swf 0 # a = $0', '00011000']
132 ['lwprev $3 # $0 = y[i]', '00110111']
133 ['lwfsubreg 1 # $0 = b - y[i]', '00101101']
134 ['swf 1 # b = $0', '00011001']
135 ['j improve_ab', '11100111']
136 ['done:', '10001010']

Screenshot:

Python code:

ISA-sim.py:

```
import fileinput
```

```
import sys
from instruction_set import instructions
from Assembler import assembler

instructions = {v: k for k, v in instructions.items()}  # invert the instructions

instruction, machine_code, gg, aa, bb, jjjjj = [""] * 6
pc = 0
cs = []

def reg(n):
    return "$" + str(int(n,2))

def binstrip(i, n=4):
    return format(int(i), f'#00{n+2}b')[2:]

lookup_table = {}
for x in range(32):
    lookup_table.update({binstrip(x,5): 0})

registers = {}
for x in range(4):
    registers.update({f"${x}": 0})
registers.update({'result': 0, 'return': 0, 'counter': 0, 'init': 0, 'status':0})


data_memory = {}
for x in range(0, 2304):
    data_memory.update({x: 0})

code = {}
argv = {}


def set_machine_code(code):
    global machine_code
    if len(code) != 8:
        raise Exception("Invalid length of machine code.")
    machine_code = code


def set_pc(n=0):
    global pc
    pc = int(n)


def get_pc():
    return pc
```

```
def increment_pc(n=1):
    global pc
    pc += n


def twos_comp(val, bits):
    """compute the 2's complement of int value val"""
    if (val & (1 << (bits - 1))) != 0:  # if sign bit is set e.g., 8bit: 128-255
        val = val - (1 << bits)  # compute negative value
    return val  # return positive value as is


class ShowUpdate(object):

    def __init__(self, f):
        self.f = f

    def __call__(self, *args, **kwargv):
        if (argv["-showcode"]):
            old_registers = dict(registers)
            old_pc = int(get_pc())
            old_memory = dict(data_memory)
        increment_pc()
        # show_ab()
        self.f()

        if(argv["-showcode"]):
            if get_pc() >= 61:
                for key in registers.keys():
                    if registers[key] != old_registers[key]:
                        print(f"\t\t{key}: {old_registers[key]} -> {registers[key]}")
                for key in data_memory.keys():
                    if data_memory[key] != old_memory[key]:
                        print(f"\t\tmem[{key}]: {old_memory[key]} -> {data_memory[key]}")
                if old_pc != get_pc():
                    print(f"\t\tpc: {'0x' + format(old_pc, '004X')} -> {'0x' + format(get_pc(), '004X')}")
                    print()


def set_words():
    global jjjjj, aa, bb, gg, instruction

    jjjjj = machine_code[-5:]
    aa = machine_code[-4:-2]
    bb = machine_code[-2:]
    gg = machine_code[-2:]
```

```
    instruction = None
    for encoding in instructions:
        if machine_code.startswith(encoding):
            instruction = globals()[instructions[encoding]]
            break
    if not instruction:
        raise Exception("Instruction not found")


@ShowUpdate
def init():
    global registers

    registers['init'] = registers['init'] | int((aa+bb),2) << 4 * registers['counter']
    if registers['counter'] == 3:
        registers['result'] = twos_comp(registers['init'], 16)
        registers['counter'] = 0
        registers['init'] = 0
    else:
        registers['counter'] = registers['counter'] + 1


@ShowUpdate
def set():
    global registers, data_memory, lookup_table
    registers[reg(gg)] = registers['result']

@ShowUpdate
def add1():
    global registers, data_memory, lookup_table
    registers[reg(gg)] += 1

@ShowUpdate
def sub1():
    global registers, data_memory, lookup_table
    registers[reg(gg)] -= 1

@ShowUpdate
def add1c():
    global registers, data_memory, lookup_table
    data_memory[int(gg,2)+8] += 1

@ShowUpdate
def hold():
    global registers
    registers['result'] = registers['$0']
```

```
@ShowUpdate
def addback():
    global registers
    registers['$0'] += registers['result']


@ShowUpdate
def seqc():
    global registers, data_memory, lookup_table
    if data_memory[int(gg,2)+4] == data_memory[int(gg,2)+8]:
        registers['status'] = 1
        data_memory[int(gg,2)+4] = 0
    else:
        registers['status'] = 0
        data_memory[int(gg,2)+4] += 1

@ShowUpdate
def swinc():
    global registers, data_memory
    data_memory[registers[reg(bb)]] = registers[reg(aa)]
    registers[reg(bb)] += 1

@ShowUpdate
def lwinc():
    global registers
    registers['$0'] = data_memory[registers[reg(gg)]]
    registers[reg(gg)] += 1

@ShowUpdate
def lwmult():
    global registers
    registers['$0'] *= data_memory[registers[reg(gg)]]

@ShowUpdate
def lwmultinc():
    global registers
    registers['$0'] *= data_memory[registers[reg(gg)]]
    registers[reg(gg)] += 1

@ShowUpdate
def lwprev():
    global registers
    registers['$0'] = data_memory[registers[reg(gg)]-1]

@ShowUpdate
def swprev():
    global registers
    data_memory[registers[reg(gg)]-1] = registers['$0']
```

```
@ShowUpdate
def swf():
    global data_memory
    data_memory[int(gg,2)] = registers['$0']

@ShowUpdate
def lwf():
    global registers
    registers['$0'] = data_memory[int(gg,2)]

@ShowUpdate
def lwfaddreg():
    global registers
    registers['$0'] = data_memory[int(gg,2)] + registers['$0']

@ShowUpdate
def lwfsubreg():
    global registers
    registers['$0'] = data_memory[int(gg,2)] - registers['$0']

@ShowUpdate
def add():
    global registers
    registers[reg(aa)] += registers[reg(bb)]

@ShowUpdate
def sub():
    global registers
    registers[reg(aa)] -= registers[reg(bb)]

@ShowUpdate
def mult():
    global registers
    registers[reg(aa)] *= registers[reg(bb)]

@ShowUpdate
def loop():
    pass

@ShowUpdate
```

```
def jif():
    global registers
    if registers['status']:
        set_pc(lookup_table[jjjjj])


@ShowUpdate
def j():
    set_pc(lookup_table[jjjjj])


@ShowUpdate
def slt():
    global registers
    registers['status'] = 1 if registers[reg(gg)] < 0 else 0

@ShowUpdate
def snotneg():
    global registers
    registers['status'] = 0 if registers[reg(gg)] < 0 else 1

@ShowUpdate
def snotneg():
    global registers
    registers['status'] = 1 if registers[reg(gg)] >= 0 else 0

@ShowUpdate
def negpos():
    global registers
    registers['$0'] = -1 if registers['$0'] < 0 else 1

def first_pass():
    for pc in code.keys():
        if code[pc][1].startswith('100'):
            lookup_table.update({code[pc][1][3:]: pc+1})
    if argv["-showlookup"]:
        print(lookup_table)


def second_pass():
    set_pc()
    max_pc = max(code.keys())
    while get_pc() <= max_pc:
        set_machine_code(code[get_pc()][1])
        set_words()
        if argv["-showcode"]:
            print("{:<16}".format(pc) + "{:<16}".format(str(code[pc])))
        instruction()
```

```
def register_contents():
    print("{:>10}".format("Register") + "{:>10}".format("Value"))
    for key in registers.keys():
        print("{:>10}".format(key) + "{:>10}".format(registers[key]))
    print("{:>10}".format("PC") + "{:>10}".format(get_pc()))

def memory_contents():
    string = "{:>16}".format(f"Base Address")
    for n in range(0,8):
        string += "{:>16}".format(f"[+{n}]")
    print(string)
    for x in range(1024,1124,8):
        string = "{:>16}".format(f"[{x}]")
        for y in range(0, 8):
            string += "{:>16}".format(f"{data_memory[x+y]}")
        print(string)

def counter_contents():
    for n in range(0,4):
        print(f"Data[{n+4}]: {data_memory[n+4]}, Data[{n+8}]: {data_memory[n+8]}")
    print()

def fast_mem():
    for n in range(0,4):
        print(f"Data[{n}]: {data_memory[n]}")

def show_ab():
    for n in range(0,2):
        print(f"Data[{n}]: {data_memory[n]}")

def coordinate_pairs():
    string = "{:>16}".format(f"Base Coords")
    for n in range(0,8):
        string += "{:>16}".format(f"[+{n}]")
    print(string)
    for x in range(0,100,8):
        string = "{:>16}".format(f"[{256+x},{512+x},{768+x}]")
        for y in range(0,8):
            string += "{:>16}".format("{:<4}".format(f"[{data_memory[256+x+y]},") +
                                      "{:>4}".format(f"{data_memory[512+x+y]},") +
                                      "{:>4}".format(f"{data_memory[768 + x + y]}]"))
        print(string)
    print()
```

```
if __name__ == '__main__':
    code = []
    argv = {"-showcode": True, "-showmem": False, "-showcounter": False, "-showfmem": False,
            "-showreg": False, "-showcoord": False, "-showlookup": False}

    for arg in sys.argv[1:]:
        if arg.split("=")[0] in argv:
            try:
                argv[arg.split("=")[0]] = False if arg.split("=")[1].lower() == 'false' else True
            except:
                argv[arg.split("=")[0]] = True

    sys.argv.clear()


    #code = assembler(fileinput.input())
    code = assembler(open('code.txt'))

    first_pass()
    second_pass()

    if(argv["-showcoord"]):
        coordinate_pairs()
    if(argv["-showmem"]):
        memory_contents()
    if (argv["-showreg"]):
        register_contents()
    if (argv["-showcounter"]):
        counter_contents()
    if (argv["-showfmem"]):
        fast_mem()

    print(cs)
```

Assembler:

```
import fileinput
import re
from instruction_set import instructions

lookup_table = {}
assembly = []


def toBin(i, n=2, hex=False):
    return format(int(i,16), f"00{n}b") if hex else format(int(i), f"00{n}b")

def to_machine_language(line):
```

```
    assembly = line.split()
    hextoint = lambda h: int(h, base=16)
    regstrip = lambda s: re.search('((?<=\$)|(?<=^))([0-3])', s).group(2)

    op = instructions[assembly[0]]
    if op.startswith('001111'):
        appendix = ""
    elif op.startswith('00'): # g-types start with 00
        appendix = toBin(regstrip(assembly[1]))
    elif op.startswith('1'): # j-types start with 1
        appendix = lookup_table[assembly[1]]
    elif op.startswith('01'): # ab-types start with 01
        if assembly[0] == 'init':
            if hextoint(assembly[1]) > 0xF:
                raise Exception("Invalid init immediate")
            appendix = toBin(assembly[1], 4, hex=True)
        else:
            appendix = (toBin(regstrip(assembly[1]))) + toBin(regstrip(assembly[2]))
    return op + appendix


def loop_to_machine_language(line):
    name = re.search('(.*?):', line).group(1)
    return "100" + lookup_table[name]


def build_lookup(line):
    if 'index' not in build_lookup.__dict__:
        build_lookup.index = 0

    if build_lookup.index >= 32:
        raise Exception('ISA does not support more than 32 loops.')

    name = re.search('(.*?):', line).group(1)
    if name in lookup_table:
        raise Exception('Loop assigned several times; causes ambiguity in machine code.')

    lookup_table.update({name: toBin(build_lookup.index, 5)})
    build_lookup.index += 1

def assemble(line):
    if line.startswith('#'):
        return None
    elif ":" in line:
        return loop_to_machine_language(line)
    else:
        return to_machine_language(line)
```

```
def assembler(lines):
    global assembly
    for line in lines:
        if line.strip():
            assembly.append(line.rstrip())

    for line in assembly:
        if line.startswith('#'):
            continue
        elif ":" in line:
            build_lookup(line)

    codepairs = {}
    pc = 0
    for line in assembly:
        if assemble(line):
            codepairs.update({pc: [line,assemble(line)]})
            pc += 1
    return codepairs

if __name__ == '__main__':
    #code = (assembler(fileinput.input()))
    code = assembler(open('code.txt'))
    print(lookup_table)
    for pc in code:
        print(pc, code[pc])
```

Instruction set:

```
instructions = {"set": "000000", "add1": "000001", "sub1": "000010",
                "hold": "00111100", "addback":"00111101", "negpos": "00111110",
                "seqc": "000011",
                "lwinc": "000100", "add1c": "000101", "swf": "000110", "lwf": "000111",
                "slt": "001000", "snotneg": "001001", "lwfaddreg": "001010", "lwfsubreg": "001011",
                "lwmultinc": "001100", "lwprev": "001101", "lwmult":"001110",
                "add": "0100", "swinc": "0101", "init": "0110",
                "loop": "100", "jif": "101", "j": "110",

                }
```