

Computer Architecture Assignment-2

Course Name: EG 212, Computer Architecture

By: Kavya Gupta (IMT2023016), Pragya Rai (IMT2023529), Ananya Vundavalli (IMT2023537)

26 February 2024

1 Introduction

In this project we have designed a MIPS processor to implement the following programs:

- Sum of squares of digits of a number.
- Factorial of a number.
- Maximum element of an array.

2 Programs

2.1 Sum of squares of digits of a number

We take a number and find sum of the squares of the digits in the number. *Assumption: We assume the given number is a non-negative integer.*

2.2 Factorial of a number

We multiply all the numbers from 1 to the given number. *Assumption: We assume given number is a non-negative integer.*

2.3 Maximum element in an array

We take the maximum element to be 0 and compare it with every element of the array. If it is bigger than the max element then that number becomes the max element. In the end we get the maximum element of the array. *Assumption: The array can have a maximum of 10 integers.*

3 Assembly Code

3.1 Sum of squares of digits of a number

We initialize `$t1` ,i.e.,sum to 0 and `$t2` ,i.e., the divisor to 10. If `$t0` is 0, we exit the loop. Otherwise we divide `$t0` by 10. The remainder is stored in `$t3` and quotient in `$t0`. We square the value in `$t3`. We then add the value in `$t3` to `$t1`.

```
.data
prompt:      .ascii "Enter a number: "
sum_msg:     .ascii "Sum of squares of digits: "
result:      .word 4      # Allocate 4 bytes for the result

.text
main:
    lw $t0, 0($s0)
    # Find the sum of squares of digits
    addi $t1, $0, 0      # initialize sum to 0
    addi $t2, $0, 10     # initialize divisor to 10

sum_of_squares_loop:
    beq $t0, $0, store_result # if $t0 is 0, exit loop
    div $t0, $t2          # divide $t0 by 10
    mfhi $t3             # remainder (last digit)
    mflo $t0             # quotient (removed last digit)
    mul $t3, $t3, $t3     # square the remainder
    add $t1, $t1, $t3     # add the squared remainder to the sum
    j sum_of_squares_loop # jump back to the beginning of the loop

store_result:
    # Store the sum in the result memory location
    sw $t1, result      # store the sum in the result
    # Exit program
    addi $v0, $0, 10     # syscall code for exit
    syscall
```

3.2 Factorial of a number

In the loop we increment the value by 1. We have \$t1 ,i.e., the result,initialized to 1 and \$t2 ,i.e., loop counter to 1. If the loop counter is equal to user input+1, the exit loop. Otherwise multiply the result by the loop counter, then increment the loop counter by 1.

```
.data
prompt:      .ascii "Enter a non-negative integer: "
result_msg:  .ascii "Factorial is: "
user_input:  .word 0 # space for storing user input

.text
main:
    # Read user input (set user input to 5)
    addi $t0, $0, 6      # set user input to 5
    sw $t0, user_input   # store user input in memory
    lw $t0, user_input   # load user input from memory into $t0
    addi $t0, $t0, 1     # for comparison purposes

    # Calculate factorial
    addi $t1, $0, 1      # initialize result to 1
    addi $t2, $0, 1      # initialize loop counter to 1

factorial_loop:
    slt $at, $t2, $t0    # if loop counter equals user input, exit loop
    beq $at, $0, end_factorial

    mul $t1, $t1, $t2    # multiply result by loop counter
    addi $t2, $t2, 1     # increment loop counter
    j factorial_loop     # jump back to the beginning of the loop

end_factorial:
    # Display the result
    addi $v0, $0, 4      # syscall code for print_str
    la $a0, result_msg   # load address of result message string
    syscall

    addi $v0, $0, 1      # syscall code for print_int
    add $a0, $t1, $0     # load the result into $a0
    syscall

    # Exit program
    addi $v0, $0, 10     # syscall code for exit
    syscall
```

3.3 Maximum element in an array

In loop, we load the word from array into \$t3. Set \$t4 to 1 if value in \$t2 < \$t3, else set the value to 0. if \$t4 is 0, then \$t3 is not the max, so let it go to not_max label, then update max to \$t3. In the not_max label, move to the next element, decrement the array size and if array size is not 0, the go to the loop. Otherwise, store max value in \$t2.

```
.data
array: .word 5, 3, 101, 1, 7, 9, 6, 2, 4, 10 # example array
array_size: .word 10 # size of the array
max: .word 0 # initialize max to the smallest possible integer

.text
main:

    la $t0, array # load address of array into $t0
    lw $t1, array_size # load array size into $t1
    lw $t2, max # load max into $t2

    # loop through the array
loop:
    lw $t3, 0($t0) # load word from array into $t3
    slt $t4, $t2, $t3 # set $t4 to 1 if $t2 < $t3, else 0
    beq $t4, $0, not_max # if $t4 is 0, then $t3 is not the max
    add $t2, $t3, $0 # update max to $t3
not_max:
    addi $t0, $t0, 4 # move to next array element
    addi $t1, $t1, -1 # decrement array size
    bne $t1, $0, loop # if array size is not zero, loop again

    sw $t2, max # store the max value in max

    # print max value

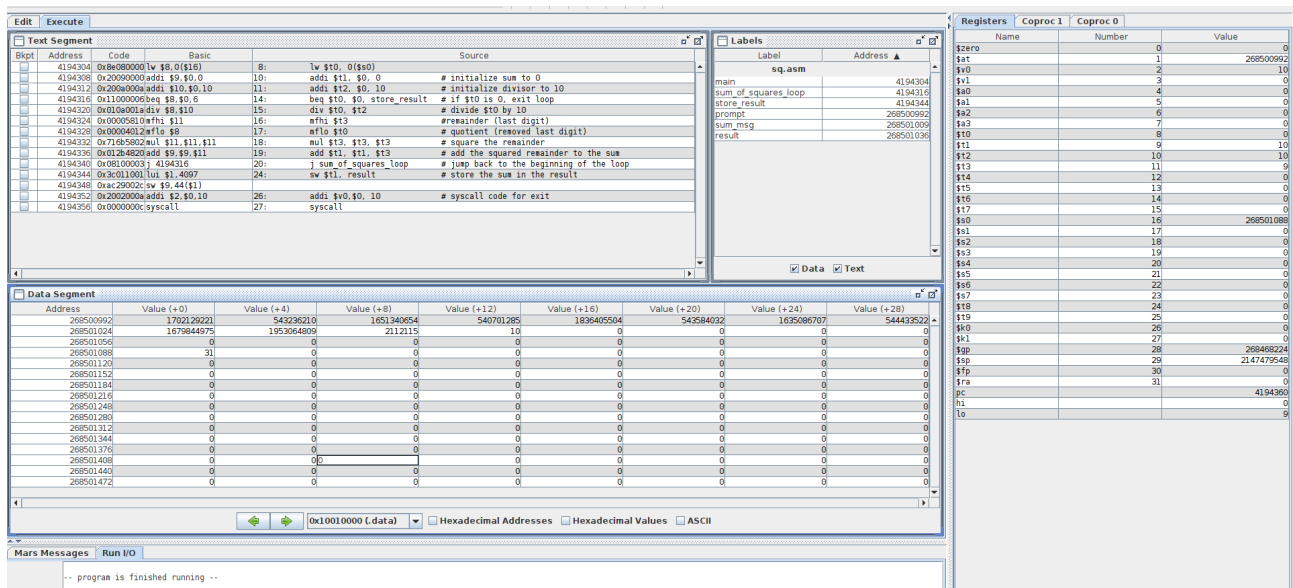
    add $v0, $0, 1
    add $a0, $t2, $0
    syscall

    # exit program
    addi $v0, $0, 10
    syscall
```

4 Assembler

We have used MARS simulator as our assembler.

4.1 Sum of squares of digits of a number



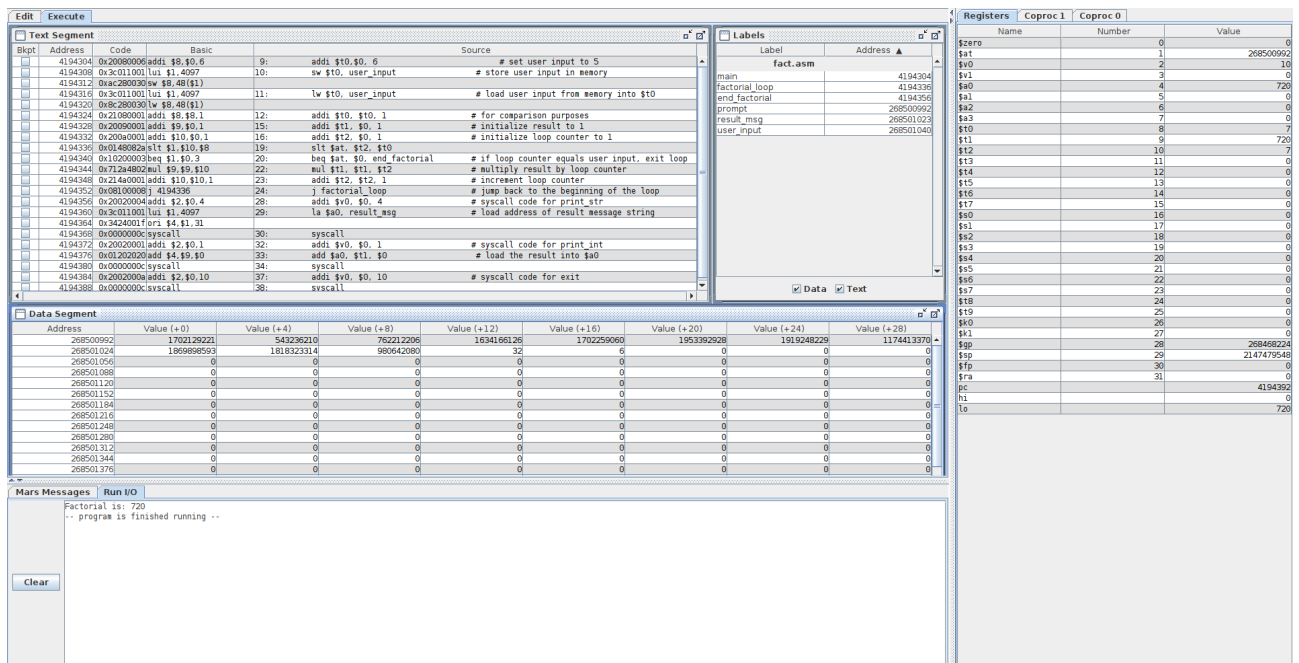
The screenshot shows the MARS simulator interface. The Text Segment contains the assembly code for the sum of squares of digits of a number. The Data Segment shows the memory layout, with the array of digits stored in memory. The Registers window shows the values of the registers, including \$t0, \$t1, \$t2, \$t3, \$t4, \$v0, and \$a0.

Register	Name	Number	Value
\$t0		0	0
\$t1		1	268500992
\$t2		2	16
\$t3		3	0
\$t4		4	0
\$v0		5	0
\$a0		6	0
\$f0		7	0
\$f1		8	0
\$f2		9	0
\$f3		10	0
\$f4		11	0
\$f5		12	0
\$f6		13	0
\$f7		14	0
\$f8		15	0
\$f9		16	0
\$f10		17	0
\$f11		18	0
\$f12		19	0
\$f13		20	0
\$f14		21	0
\$f15		22	0
\$f16		23	0
\$f17		24	0
\$f18		25	0
\$f19		26	0
\$f20		27	0
\$f21		28	0
\$f22		29	0
\$f23		30	0
\$f24		31	0
\$PC			4194360
\$LO			0
\$HI			0

Machine Code:

```
For sum of squares:
instruction memory={
4194304 : "10001110000010000000000000000000",
4194308 : "00100000000010010000000000000000",
4194312 : "00100000000010100000000000001010",
4194316 : "0001000100000000000000000000110",
4194320 : "00000001000010100000000000011010",
4194324 : "000000000000000010110000010000",
4194328 : "0000000000000000100000000010010",
4194332 : "01110001011010110101100000000010",
4194336 : "00000001001010110100100000000000",
4194340 : "00001000000100000000000000000011",
4194344 : "0011110000000010001000000000001",
4194348 : "10101100001010010000000000101100",}
```

4.2 Factorial of a number



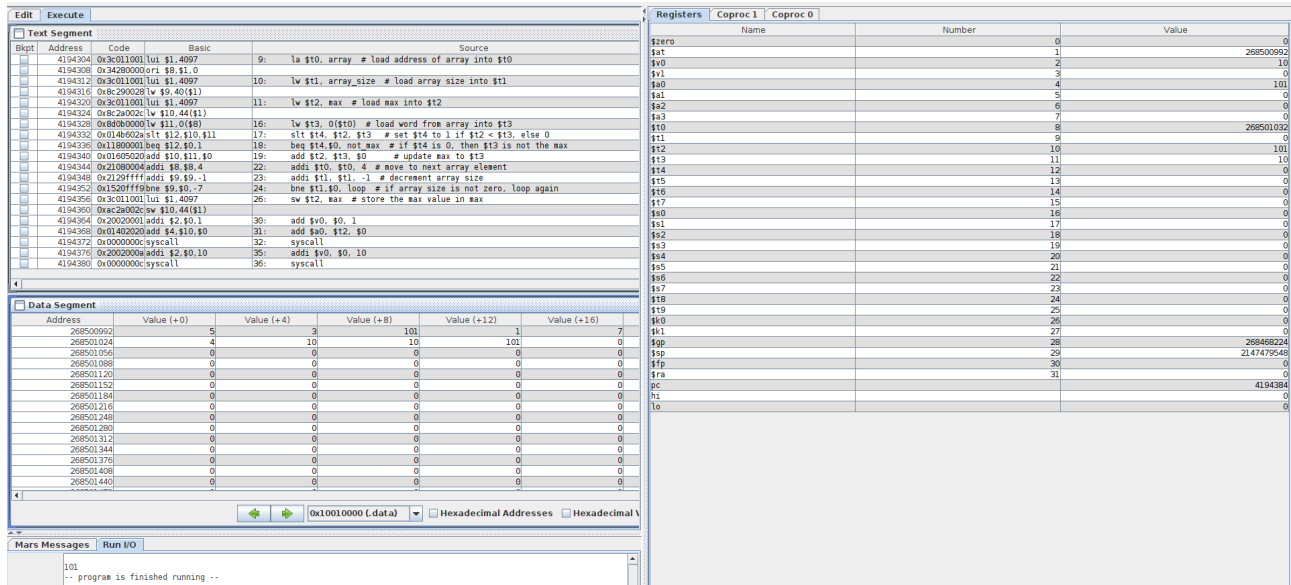
The screenshot displays a MIPS simulator interface. The **Text Segment** window shows assembly code for calculating the factorial of 5. The **Data Segment** window shows memory addresses and values. The **Registers** window shows the state of registers, with \$a0 containing the result 720. The **Messages** window shows the output 'Factorial is: 720'.

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
26850092	170228221	543296210	762212206	1634166126	1702259060	195392926	1919248229	1174413370
26850104	166969693	181812314	90642080	0	0	0	0	0
26850106	0	0	0	0	0	0	0	0
26850108	0	0	0	0	0	0	0	0
26850110	0	0	0	0	0	0	0	0
26850112	0	0	0	0	0	0	0	0
26850114	0	0	0	0	0	0	0	0
26850116	0	0	0	0	0	0	0	0
26850118	0	0	0	0	0	0	0	0
26850120	0	0	0	0	0	0	0	0
26850122	0	0	0	0	0	0	0	0
26850124	0	0	0	0	0	0	0	0
26850126	0	0	0	0	0	0	0	0
26850128	0	0	0	0	0	0	0	0
26850130	0	0	0	0	0	0	0	0
26850132	0	0	0	0	0	0	0	0
26850134	0	0	0	0	0	0	0	0
26850136	0	0	0	0	0	0	0	0

Machine Code:

```
For factorial:
instruction_memory = {
4194304: "0010000000001000000000000000101",
4194308: "00111100000000010001000000000001",
4194312: "10101100001010000000000000110000",
4194316: "00111100000000010001000000000001",
4194320: "10001100001010000000000000110000",
4194324: "00100000100001000000000000000001",
4194328: "00100000000010010000000000000001",
4194332: "00100000000010100000000000000001",
4194336: "00000001010010000000100000101010",
4194340: "00001000000100000000000000000011",
4194344: "01110001001010100100100000000010",
4194348: "00100000101001010000000000000001",
4194352: "00001000000100000000000000001000",}
```

4.3 Maximum element in an array



The screenshot displays the MARS MIPS simulator interface. The **Text Segment** shows assembly code for finding the maximum element in an array. The **Data Segment** shows memory values. The **Registers** window shows the state of registers.

Text Segment:

Address	Code	Basic	Source
4194304	0x3c01000	lui \$1, 4097	9: la \$t0, array # load address of array into \$t0
4194308	0x3420000	ori \$0, \$1, 0	
4194312	0x3c01000	lui \$1, 4097	10: lw \$t1, array_size # load array size into \$t1
4194316	0x8c20000	lv \$9, 40(\$t1)	
4194320	0x3c01000	lui \$1, 4097	11: lw \$t2, max # load max into \$t2
4194324	0x8c20000	lv \$10, 44(\$t1)	
4194328	0x8c20000	lv \$t3, 0(\$t0)	16: lw \$t3, 0(\$t0) # load word from array into \$t3
4194332	0x0140000	slt \$t4, \$t2, \$t3	17: slt \$t4, \$t2, \$t3 # set \$t4 to 1 if \$t2 < \$t3, else 0
4194336	0x1180000	beq \$t4, \$0, not_max	18: beq \$t4, \$0, not_max # if \$t4 is 0, then \$t2 is not the max
4194340	0x0160000	add \$t2, \$t2, \$t3	19: add \$t2, \$t2, \$t3 # update max to \$t3
4194344	0x2100000	addi \$t0, \$t0, 4	22: addi \$t0, \$t0, 4 # move to next array element
4194348	0x2120000	addi \$t1, \$t1, -1	23: addi \$t1, \$t1, -1 # decrement array size
4194352	0x1520000	bne \$t1, \$0, loop	24: bne \$t1, \$0, loop # if array size is not zero, loop again
4194356	0x3c01000	lui \$1, 4097	25: sw \$t2, max # store the max value in max
4194360	0x8c20000	lv \$t0, 44(\$t1)	
4194364	0x2002000	addi \$t0, \$t0, 1	30: addi \$t0, \$t0, 1
4194368	0x0140000	add \$t0, \$t0, \$t2	31: add \$t0, \$t0, \$t2
4194372	0x0000000	syscall	32: syscall
4194376	0x2002000	addi \$t0, \$t0, 10	35: addi \$t0, \$t0, 10
4194380	0x0000000	syscall	36: syscall

Data Segment:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)
268500992	5	3	101	1	7
268501024	4	10	10	101	0
268501056	0	0	0	0	0
268501088	0	0	0	0	0
268501120	0	0	0	0	0
268501152	0	0	0	0	0
268501184	0	0	0	0	0
268501216	0	0	0	0	0
268501248	0	0	0	0	0
268501280	0	0	0	0	0
268501312	0	0	0	0	0
268501344	0	0	0	0	0
268501376	0	0	0	0	0
268501408	0	0	0	0	0
268501440	0	0	0	0	0

Registers:

Name	Number	Value
\$zero	0	0
\$at	1	268500992
\$v0	2	10
\$v1	3	0
\$a0	4	101
\$a1	5	0
\$a2	6	0
\$a3	7	0
\$t0	8	268501032
\$t1	9	0
\$t2	10	101
\$t3	11	10
\$t4	12	0
\$t5	13	0
\$t6	14	0
\$t7	15	0
\$t8	16	0
\$t9	17	0
\$s0	18	0
\$s1	19	0
\$s2	20	0
\$s3	21	0
\$s4	22	0
\$s5	23	0
\$s6	24	0
\$s7	25	0
\$s8	26	0
\$s9	27	0
\$fp	28	268468224
\$sp	29	214747544
\$gp	30	0
\$ra	31	0
\$pc		4194384
\$hi		0
\$lo		0

Mars Messages:

```
-- program is finished running --
```

Machine Code:

```
For max element of array:
instruction memory={
4194304 : "00111100000000010001000000000001",
4194308 : "00110100001010000000000000000000",
4194312 : "00111100000000010001000000000001",
4194316 : "10001100001010010000000000101000",
4194320 : "00111100000000010001000000000001",
4194324 : "10001100001010100000000000101100",
4194328 : "10001101000010110000000000000000",
4194332 : "000000010010110110000000101010",
4194336 : "00010001100000000000000000000001",
4194340 : "00000001011000000101000001000000",
4194344 : "001000010000100000000000000000100",
4194348 : "001000010010101111111111111111",
4194352 : "00010101001000001111111111111001",
4194356 : "00111100000000010001000000000001",
4194360 : "1010110000101010000000000101100", }
```

5 Processor

We have written the functions in our code as the different stages of the MIPS processor.

The first function is **Instruction Fetch (IF)** which fetches instruction at the current PC from **Instruction Memory (IM)**.

The next function is **Instruction Decode (ID)** which decodes each instruction into different fields like op-code, rs, rt etc., and returns them. This calls the Control function to generate control signals for each instruction.

```

1 #This function is used to fetch the instructions
2 def IF():
3     global PC, instruction_memory
4     inst = instruction_memory[PC]
5     PC = PC + 4 #increment PC by 4
6     return inst
7
8
9 #this function decodes each instruction into different fields
10 def ID(inst):
11     global opcode, rs, rt, rd, shamt, funct, target, imm
12     #declare the different fields as empty strings
13     opcode = ""
14     rs = ""
15     rt = ""
16     rd = ""
17     shamt = ""
18     funct = ""
19     target = ""
20     imm = ""
21     opcode = inst[0:6] #first 6 bits of instruction
22     control(opcode) #function call
23
24     if opcode == "000010": # jump instruction
25         target = "0000" + inst[6:32] + "00"
26         return opcode, rs, rt, rd, shamt, funct, target, imm
27
28     elif opcode == "000000" or opcode == "011100": # R-type instructions
29         rs = inst[6:11]
30         rt = inst[11:16]
31         rd = inst[16:21]
32         shamt = inst[21:26]
33         funct = inst[26:32]
34         return opcode, rs, rt, rd, shamt, funct, target, imm
35
36     else: # I-type instruction
37         rs = inst[6:11]
38         rt = inst[11:16]
39         imm = inst[16:32]
40         return opcode, rs, rt, rd, shamt, funct, target, imm
41

```

Figure 1: processor 1

bin_to_int function is used to convert binary string to a signed integer and also acts as the sign extend component in MIPS.

The Execute function(EX) is similar to ALU part to generate ALU output based on different ALU control signals generated by the ALUControlUnit function.

```

42 #This function converts a binary string to a signed integer
43 def bin_to_int(bin_str):
44
45     if bin_str[0] == '1': # For negative number
46         if bin_str == '1' + '0' * (len(bin_str) - 1): # Check if it's the minimum negative number
47             return -2**(len(bin_str) - 1)
48         bin_str = ''.join('1' if b == '0' else '0' for b in bin_str) # Flip the bits
49         return -1 * (int("0b"+bin_str, 2) + 1) # Add 1 and then negate it
50
51     else: # For positive number
52         return int("0b"+bin_str, 2)
53
54
55
56 #This function will perform the various operations based on ALUOp, funct
57 def EX(rs, rt, imm, funct, target, ALUOp):
58
59     global ALUSrc, Branch, Jump, RegDst, reg, opcode, PC
60
61     ALU_in1 = 0
62     ALU_in2 = 0
63     ALU_res = 0
64     ALU_res1 = 0
65     ALU_res2 = 0
66
67     ALU_ctrl = ALU_controlUnit(ALUOp, funct) #function call
68
69     # jump
70     if Jump == 1:
71         PC = int("0b"+target, 2)
72         return 0
73     if rs != "":
74         ALU_in1 = reg[int("0b"+rs, 2)] # input1 to ALU
75
76     if ALUSrc == 0:
77         ALU_in2 = reg[int("0b"+rt, 2)] # input2 to ALU
78
79     else:
80         ALU_in2 = bin_to_int(imm)
81

```

Figure 2: processor 2

```

81
82 # lui
83 if ALU_ctrl == "001":
84     ALU_res = int("0b"+imm, 2) * (2**16)
85     return ALU_res
86
87 # ori
88 elif ALU_ctrl == "000":
89     ALU_res = ALU_in1 | ALU_in2
90     return ALU_res
91
92 # add
93 if ALU_ctrl == "010":
94     ALU_res = ALU_in1 + ALU_in2
95
96 # slt
97 elif ALU_ctrl == "100":
98     ALU_res = 1 if ALU_in1 < ALU_in2 else 0
99
100 # mul
101 elif ALU_ctrl == "111":
102     ALU_res = ALU_in1 * ALU_in2
103
104 #div
105 elif ALU_ctrl == "110":
106     ALU_res1 = ALU_in1 // ALU_in2
107     ALU_res2 = ALU_in1 % ALU_in2
108     return (ALU_res1,ALU_res2)
109
110 # sub
111 elif ALU_ctrl == "011":
112     ALU_res = ALU_in1 - ALU_in2
113
114 if Branch == 1:
115     # beq
116     if ALU_res==0 and opcode=="000100":
117         PC = PC + (4*bin_to_int(imm))
118     # bne
119     elif ALU_res !=0 and opcode=="000101":
120         PC=PC+ 4*bin_to_int(imm)
121
122 return ALU_res

```

Figure 3: processor 3

Memory function (MEM) reads from the memory or writes into the memory as per control signals `memwr` and `mmemrd`. Also it has `memtoreg` control signal to write the data into the writeback phase. WriteBack function (WB) writes back the data returned by the MEM function into the register destination. *We have assumed regwr=2 as our div instruction ALU stage needs to return 2 outputs and read them into 2 special registers hi and lo.*

```

125 #This function describes memory access
126 def MEM(ALU_res, rt):
127     global MemRd, MemWr, MentoReg, data_memory, hi, lo
128
129     # if MemRd == 1 then read from the address produced by ALU
130     if MemRd == 1:
131         return data_memory[ALU_res]
132
133     # if MemWr == 1 then write into the address produced by ALU
134     elif MemWr == 1:
135         data_memory[ALU_res] = reg[int("0b"+rt, 2)]
136
137     if MentoReg == 0:
138         return ALU_res
139     elif MentoReg == 1:
140         return data_memory[ALU_res]
141
142 #This function writes back to the reg
143 def WB(rd, rt, data):
144     global reg, hi, lo, RegWr, RegDst
145
146     if (RegWr == 2): # if condition for division
147         hi=data[1]
148         print("hi",hi)
149         lo=data[0]
150         print("lo",lo)
151
152     elif RegWr == 1:
153         if RegDst == 1:
154             if (func=="010000"):
155                 reg[int("0b"+rd, 2)] = hi #this hi is used in case of div, it stores remainder
156             elif (func=="010010"):
157                 reg[int("0b"+rd, 2)] = lo #this lo is also used in case of div, it stores quotient
158             else:
159                 reg[int("0b"+rd, 2)] = data
160         else:
161             reg[int("0b"+rt, 2)] = data
162
163
164
165
166

```

Figure 4: processor 4

The control function sets the control signals of `regwr`, `memwr`, `memrd` etc., according to the instruction. Some of the instructions whose `ALUOp` is not well-known, we have assumed the `ALUOps` and `ALUctrl`.

```

169 #This function sets the control signal based on the opcodes
170 #Control signals RegDst, Branch, MemRd, MentoReg, ALUOp, MemWr, ALUSrc, RegWr, Jump
171 def control(opcode):
172     global RegDst, Branch, MemRd, MentoReg, ALUOp, MemWr, ALUSrc, RegWr, Jump
173
174     if opcode == "100011": # lw instruction
175         ALUOp = "00"
176         ALUSrc = 1
177         Branch = 0
178         MemRd = 1
179         MentoReg = 1
180         MemWr = 0
181         Jump = 0
182         RegDst = 0
183         RegWr = 1
184
185     elif opcode == "101011": # sw instruction
186         ALUOp = "00"
187         ALUSrc = 1
188         Branch = 0
189         MemRd = 0
190         MentoReg = 0
191         MemWr = 1
192         Jump = 0
193         RegDst = 0
194         RegWr = 0
195
196     elif opcode == "000000": # R-type instruction
197         ALUOp = "10"
198         ALUSrc = 0
199         Branch = 0
200         MemRd = 0
201         MentoReg = 0
202         MemWr = 0
203         Jump = 0
204         RegDst = 1
205         RegWr = 1
206

```

Figure 5: processor 5


```

207 elif opcode == "011100": # mul instruction
208     ALUOp = "10"
209     ALUSrc = 0
210     Branch = 0
211     MemRd = 0
212     MemtoReg = 0
213     MemWr = 0
214     Jump = 0
215     RegDst = 1
216     RegWr = 1
217
218 elif opcode == "001111": # lui instruction
219     ALUOp = "11"
220     ALUSrc = 0
221     Branch = 0
222     MemRd = 0
223     MemtoReg = 0
224     MemWr = 0
225     Jump = 0
226     RegDst = 0
227     RegWr = 1
228
229 elif opcode == "001101": # ori instruction
230     ALUOp = "1"
231     ALUSrc = 1
232     Branch = 0
233     MemRd = 0
234     MemtoReg = 0
235     MemWr = 0
236     Jump = 0
237     RegDst = 0
238     RegWr = 1
239

```

Figure 6: processor 6

```

240 elif opcode == "001000": # addi instruction
241     ALUOp = "00"
242     ALUSrc = 1
243     Branch = 0
244     MemRd = 0
245     MemtoReg = 0
246     MemWr = 0
247     Jump = 0
248     RegDst = 0
249     RegWr = 1
250
251 elif opcode == "000100": # beq instruction
252     ALUOp = "01"
253     ALUSrc = 0
254     Branch = 1
255     MemRd = 0
256     MemtoReg = 0
257     MemWr = 0
258     Jump = 0
259     RegDst = 0
260     RegWr = 0
261
262 elif opcode == "000101": # bne instruction
263     ALUOp = "01"
264     ALUSrc = 0
265     Branch = 1
266     MemRd = 0
267     MemtoReg = 0
268     MemWr = 0
269     Jump = 0
270     RegDst = 0
271     RegWr = 0
272

```

Figure 7: processor 7

```

272
273 elif opcode == "000010": # j-type instruction
274     ALUSrc = "0"
275     Branch = 0
276     MemRd = 0
277     MemtoReg = 0
278     MemWr = 0
279     Jump = 1
280     RegDst = 0
281     RegWr = 0
282

```

Figure 8: processor 8

```

283 #This function generates ALU control signals which decide what operation is to be done by the ALU
284 def ALU_controlUnit(ALUOp, funct):
285     global RegWr
286     # lw, sw, addi
287     if ALUOp == "00":
288         return "010"
289
290     # lui
291     elif ALUOp == "11":
292         return "001"
293
294     elif ALUOp == "10": # R format
295         # add
296         if funct == "100000":
297             return "010"
298
299         # slt
300         elif funct == "101010":
301             return "100"
302
303         # mul : alu control is the same as add(mul is repeated add)
304         elif funct == "000010":
305             return "111"
306
307         #div
308         elif funct=="011010":
309             RegWr=2
310             return "110"
311
312         #mfhi - moves hi to rd
313         elif funct=="010000":
314             return "010"
315
316         #mflo - moves lo to rd
317         elif funct == "010010":
318             return "010"
319
320         # sub
321         elif funct == "100010":
322             return "011"
323

```

Figure 9: processor 9

```

326 # beq,bne
327 elif ALUOp == "01":
328     return "011"
329
330 #ori
331 elif ALUOp == "1":
332     return "000"
333
334 # Now we will initialize the state of the processor
335 #initialise PC to point to the first instruction
336 PC = 4194304
337
338 #initialise the various fields of instruction with empty strings
339 opcode = rs = rt = rd = shamt = funct = target = imm = ""
340
341 #initialise control signals of instructions with 0
342 RegDst = Branch = MemRd = MemtoReg = ALUOp = MemWr = ALUSrc = RegWr = Jump = 0
343
344 #initialise reg
345 reg = [0] * 32
346
347
348 #Controlling which program is to be run
349 inp=input('Enter a letter from a,b,c which decides which program is to be run: ')
350 a.Sum of squares
351 b.Factorial
352 c.Max element of an array : '')
353
354 print("\n")
355
356
357 #-----

```

Figure 10: processor 10

```

Open  ▾  final_processor_v6.py  Ln 115, Col 1
~Documents/mps

358 #for sum of squares
359 if (inp=="a"):
360
361
362     reg[16]=268501088 # $s0 value is initialized to address in data memory that contains our number
363     global hi
364     hi=0
365     global lo
366     lo=0
367
368
369     data_memory = {
370         268500992: 0,
371         268500996: 0,
372         268501000: 0,
373         268501004: 0,
374         268501008: 0,
375         268501012: 0,
376         268501016: 0,
377         268501020: 0,
378         268501024: 0,
379         268501028: 0,
380         268501032: 0,
381         268501036: 0, #stores the final sum
382         268501040: 0,
383         268501044: 0,
384         268501048: 0,
385         268501052: 0,
386         268501056: 0,
387         268501060: 0,
388         268501064: 0,
389         268501068: 0,
390         268501072: 0,
391         268501076: 0,
392         268501080: 0,
393         268501084: 0,
394         268501088: 31, #number whose sum of squares of digits is to be found(give input)
395         268501092: 0,
396     }

```

Figure 11: processor 11

```

397
398     instruction_memory={
399         4194304 : "10001110000010000000000000000000",
400         4194308 : "00100000000010010000000000000000",
401         4194312 : "00100000000010100000000000001010",
402         4194316 : "00010001000000000000000000000110",
403         4194320 : "00000001000010100000000000011010",
404         4194324 : "0000000000000000101100000010000",
405         4194328 : "000000000000000010000000010010",
406         4194332 : "0111000101101011010110000000010",
407         4194336 : "0000000100101011010010000010000",
408         4194340 : "0000100000010000000000000000011",
409         4194344 : "001111000000001000100000000001",
410         4194348 : "1010110000101001000000000101100",
411     }
412
413 #-----

```

Figure 12: processor 12

```

414 # for factorial
415 if (inp=="b"):
416     data_memory = {
417         268500992: 0,
418         268500996: 0,
419         268501000: 0,
420         268501004: 0,
421         268501008: 0,
422         268501012: 0,
423         268501016: 0,
424         268501020: 0,
425         268501024: 0,
426         268501028: 0,
427         268501032: 0,
428         268501036: 0,
429         268501040: 0,
430         268501044: 0,
431         268501048: 0,
432         268501052: 0,
433         268501056: 0,
434         268501060: 0,
435         268501064: 0,
436         268501068: 0,
437         268501072: 0,
438         268501076: 0,
439         268501080: 0,
440         268501084: 0,
441         268501088: 0,
442         268501092: 0,
443     }

```

Figure 13: processor 13

```

444 instruction_memory = {
445     4194304: "001000000000100000000000000001", #number whose factorial is to be found(give input)
446     4194308: "001111000000001000100000000001",
447     4194312: "101011000010100000000000011000",
448     4194316: "001111000000001000100000000001",
449     4194320: "100011000010100000000000011000",
450     4194324: "001000010000100000000000000001",
451     4194328: "001000000000100100000000000001",
452     4194332: "001000000000101000000000000001",
453     4194336: "00000001010010000000100000101010",
454     4194340: "000100000100000000000000000001",
455     4194344: "0111000100101010010010000000010",
456     4194348: "001000010100101000000000000001",
457     4194352: "000010000001000000000000000100",
458 }
459 #-----
460 -----

```

Figure 14: processor 14

```

461 #max element of array(array can take upto 10 elements)
462 if (inp=="c"):
463     data_memory = {
464         268500992: 88, #array elements (give input)
465         268500996: 3,
466         268501000: 66,
467         268501004: 723,
468         268501008: 9,
469         268501012: 0,
470         268501016: 0,
471         268501020: 0,
472         268501024: 0,
473         268501028: 0,
474         268501032: 5, #size of array
475         268501036: 0, #stores the final result
476         268501040: 0,
477         268501044: 0,
478         268501048: 0,
479         268501052: 0,
480         268501056: 0,
481         268501060: 0,
482         268501064: 0,
483         268501068: 0,
484         268501072: 0,
485         268501076: 0,
486         268501080: 0,
487         268501084: 0,
488         268501088: 0,
489         268501092: 0,
490     }

```

Figure 15: processor 15

```

491 instruction_memory={
492 4194384 : "001111000000001000100000000001",
493 4194388 : "001101000010100000000000000000",
494 4194312 : "00111100000000010001000000000001",
495 4194316 : "10001100001010010000000000101000",
496 4194320 : "00111100000000010001000000000001",
497 4194324 : "10001100001010100000000000101100",
498 4194328 : "10001101000010110000000000000000",
499 4194332 : "000000001010010110110000000101010",
500 4194336 : "00010001100000000000000000000001",
501 4194340 : "00000001011000000101000000100000",
502 4194344 : "001000010000100000000000000000100",
503 4194348 : "0010000100101001111111111111111",
504 4194352 : "00010101001000001111111111111001",
505 4194356 : "00111100000000010001000000000001",
506 4194360 : "1010110000101010000000000101100",
507 }
508
509

```

Figure 16: processor 16

```

513 def print_result():
514
515     print("Final result:\n")
516     print("Reg : \n" , reg)
517     print("\n")
518     print("Data memory : \n" , data_memory)
519     print("\n")
520
521     if (inp=="a"):
522         print("Sum of squares of digits:", data_memory[268501036])
523     elif (inp=="b"):
524         print("Factorial is ",reg[9])
525     elif (inp=="c"):
526         print("Max element of array is :",reg[10])
527
528 # Main simulation loop
529 while (1):
530     #perform the 5 stages fetch , decode , execute , memory access and writeback
531     #print value of PC at every step
532     print("Current PC : " , PC)
533
534     #instruction fetch
535     inst = IF()
536
537     #instruction decode
538     opcode, rs, rt, rd, shamt, funct, target, imm = ID(inst)
539
540     #instruction execute
541     ALU_res = EX(rs, rt, imm, funct, target, ALUOp) # Pass ALUOp to EX function
542
543     #memory access
544     mem_data = MEM(ALU_res, rt) #function call
545
546     #finally writeback
547     WB(rd, rt, mem_data) #function call
548
549     print("Reg : \n" , reg)
550     print("\n")
551     print("Data memory : \n" , data_memory)
552     print("\n")
553     print("-----")
554

```

Figure 17: processor 17

```

555     if (inp=="a"):
556         if PC > 4194348:
557             break
558     elif (inp=="b"):
559         if PC > 4194352:
560             break
561     elif (inp=="c"):
562         if PC > 4194360:
563             break
564
565     print("-----")
566     print("\n")
567
568     print_result()

```

Figure 18: processor 18

6 Result

6.1 Sum of squares of digits of a number

Here the input number is 31.

[illegible]

Page 14

[illegible]

Figure 21: result 3

[illegible]

Figure 22: result 4

```

.....
Final result:
Reg :
[0, 268500992, 0, 0, 0, 0, 0, 0, 10, 10, 9, 0, 0, 0, 0, 268501088, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
(268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 10, 268501040: 0, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 31, 268501092: 0)

Sum of squares of digits: 10

```

Figure 23: result 5

6.2 Factorial of a number

Here the input number is 5.

```

Enter a letter from a,b,c which decides which program is to be run:
a.Sum of squares
b.Factorial
c.Max element of an array : b

Current PC : 4194304
Reg :
[0, 0, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
(268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 0, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0)

.....
Current PC : 4194308
Reg :
[0, 268500992, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
(268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 0, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0)

.....
Current PC : 4194312
Reg :
[0, 268500992, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
(268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 5, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0)

.....
Current PC : 4194316
Reg :
[0, 268500992, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
(268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 5, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0)

.....
Current PC : 4194320
Reg :
[0, 268500992, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

```

Figure 24: result 1

[illegible]

Page 17

[illegible]

Page 18

```

-----
Current PC : 4194340
Reg :
[0, 1, 0, 0, 0, 0, 0, 0, 6, 24, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
{268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 5, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0}

-----
Current PC : 4194344
Reg :
[0, 1, 0, 0, 0, 0, 0, 0, 6, 120, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
{268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 5, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0}

-----
Current PC : 4194348
Reg :
[0, 1, 0, 0, 0, 0, 0, 0, 6, 120, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
{268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 5, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0}

-----
Current PC : 4194352
Reg :
[0, 1, 0, 0, 0, 0, 0, 0, 6, 120, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
{268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 5, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0}

-----
Current PC : 4194336
Reg :
[0, 0, 0, 0, 0, 0, 0, 0, 6, 120, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
{268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 5, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0}

```

Figure 29: result 6

```

-----
Current PC : 4194340
Reg :
[0, 0, 0, 0, 0, 0, 0, 0, 6, 120, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
{268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 5, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0}

-----
Final result:
Reg :
[0, 0, 0, 0, 0, 0, 0, 0, 6, 120, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

Data memory :
{268500992: 0, 268500996: 0, 268501000: 0, 268501004: 0, 268501008: 0, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 0, 268501036: 0, 268501040: 5, 268501044: 0, 268501048: 0,
268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0}

Factorial is 120

```

Figure 30: result 7

6.3 Maximum element in an array

Here the input array is [88,3,66,723,9].

[illegible]

Page 20

[illegible]

Figure 33: result 3

[illegible]

Figure 34: result 4

[illegible]

Figure 35: result 5

[illegible]

Figure 36: result 6

[illegible]

Page 23

```
.....  
.....  
Final result:  
Reg :  
[0, 268500992, 0, 0, 0, 0, 0, 268501012, 0, 723, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  
Data memory :  
{268500992: 88, 268500996: 3, 268501000: 66, 268501004: 723, 268501008: 9, 268501012: 0, 268501016: 0, 268501020: 0, 268501024: 0, 268501028: 0, 268501032: 5, 268501036: 723, 268501040: 0, 268501044: 0, 268501048: 0, 268501052: 0, 268501056: 0, 268501060: 0, 268501064: 0, 268501068: 0, 268501072: 0, 268501076: 0, 268501080: 0, 268501084: 0, 268501088: 0, 268501092: 0}
```

Figure 39: result 9