Faculty of engineering

Ain-shams University

MIPS DATAPATH AND CONTROL UNIT SIMULATOR

Computer Architecture

Prepared by:

- Ayman Hesham Mohamed (16P3037)
- Yara Hossam Mohamed (16P3002)
- Caroleen Mourad (16P6019)
- Noha Ibrahim Hassanien (16P2001)
- Nada Tarek Abdallah (16P6053)

Submitted to: Dr Cherif Salama

Eng. Ahmed Fathi

Table of contents

1. Brief Description:

- The programming language used is Java. The project is divided to classes: “Instruction”, “InstructionMemory”, “RFormat”, “IFormat”, “JFormat”, “Register”, “RegisterFile”, “Controls”, “ALUControl”, “ALUOperation”, “Memory”, and the main class named “Architecture_proj”.

- We have created an abstract class “Instruction” that have 3 children:

- RFormat: Comprise the majority of commands of MIPS Assembly, for example addition, subtraction, division and multiplication, and all logical operations.
  This format divides the 32 bits into: “opcode” (6 bits),”rs” (5 bits),” rt” (5 bits),” rd” (5 bits),” shamt” (5 bits),” FunctionCode” (6 bits).

- IFormat: Comprise the instructions where one of the values are directly passed in the code, the immediate instruction I-Type are characterized by operations with one value in the register and another in a part of the code, with the data of the maximum length of 16 bits. This format divides the 32 bits into: “opcode” (6 bits),”rs” (5 bits),” rt” (5 bits),” Offset” (16 bits).

- JFormat: Represents the instructions of unconditional jump) that extends from it. All instructions that should be supported are subclasses of one of the three super classes. This format divides the 32 bits into: “opcode” (6 bits),” jump” (26 bits).

- Each of them contains its fields and methods and attributes. When we create an object from one of them, it would have its own controls based on the opcode or function code if it is RFormat because that the opcode of this type of instruction is (000000) so the indentation of the instruction is by observing the function code.

- We have created also a class named “InstructionMemory” which have an array of String as fields and are initialize as empty array and have also methods for example “Instruction_Decode” that divide the 32 bits instruction based on its format. At the beginning of the program, this class set the PC using a method named “SetPC ()” with the value 0. And After every instruction the PC is incremented by 1. So, if we want to jump or branch we have to get the PC using the method “getPC ()”.

- We have created a class named “Register” that holds String name and an integer value. And contains its setters and getters that we use at the program then.

- The program holds also a class named “RegisterFile” that is composed of an array of 32 “Register” that we have mentioned before. Noting that at the initialization of the program, we create an object of this class that contains 32 Registers named: $0, $at, $v0…. and holds a value “0” at the beginning.

- The program have also a class named “Controls” that take the opcode from the instruction (either it is RFormat, IFormat or JFormat) and generates the controls signals to the different units of the path. Control signals are set according to the class of the instruction when it is created with those controls we can determine if we would write in a register or if we would read from the memory and other controls.

- We have also a class named “ALUControl” that take control signal (ALUOp) as an input and based on this input it sends another signal to the class named “ALUOperation” so it can execute the required operation (add, sub…).

- The “ALUOperation” takes input from the register file or from the “SignExtend” method based on another control signal(ALUSrc) and (ALUOp) as we mentioned before, so the answer of the operation is ready and also the “ZeroFlag” that we use if the current Instruction is branch instructions “beq”.

- In addition of a class “Mem” that is composed of integer ArrayList of size 100 that are initialized “0”. The memory could support many instructions as (lw,sw,lb,lbu,sb) because of the methods supported in this class as(loadByte(),storeByte()..)that take input from ALUOperation and controls signals (MemRead,MemWrite…) and based on those controls it can either load from the memory(Byte or a word) in a specific address that we take from the “ALUOperation” or store in it the result of the “ALUOperation”.
  Each line in this ArrayList contains 1 Byte = 8 bits.So when we load a word we load 4 lines of the memory and then we concatenate them to form the word. As the store word, we store the 4 bytes that form the word, each byte in his line. So the word takes 4 lines then.

- For the Jump instruction (j, jal) when the “Controls” detect the instruction and sends the “jump” control. The last 26 bits of the instruction pass by the “SignExtend” as it is an JFormat and is shifted by 2 and the concatenated with the first 4 bits of the updated PC and the result would be the next address we would jump for. finally, the PC would be updated with the new address.
- For the branch instruction (beq), when the instruction arrive at the ALUOperation if the “ZeroFlag” =1 and the control signal “Branch” is 1 also, the PC have to be updated. We can update it when the 16 bits of the offset are SignExtend the shifted by 2 then added to the PC+4.

- The “main” class is the one that uses all these components to simulate the MIPS data path, first the we create a “RegisterFile” and the “Memory”. We ask the user to enter the number of instructions “noi” he would enter so that he could fill the array of strings of the “InstructionMemory” by entering “noi” String that represents the instruction in binary. It has to be 32 bits else the program would terminate. Once we have the “InstructionMemory” is ready with all instructions filled by the user, we start the decoding and dividing the current instruction based on its format then sending the controls to the units of the path and prints on the console the controls of the current instruction. Then the execution takes place at the “ALUOperation” after receiving all the signals it needs and the data stored in the chosen registers as source.

2. The datapath:

At first, the program creates an empty “Instruction Memory” then it creates the “Register File” that contains the 32 “Registers”. After the program creates an empty “Memory” then the program asks the user to enter the number of instructions “Please enter the instruction (IN BINARY!)” and to enter the instructions in binary form. Then the program checks if the instruction entered by user is 32 bits, the instruction is saved in the “Instruction Memory”, else the program shows a message to the user that the instruction is invalid “INVALID INSTRUCTION”. After filling the “Instruction Memory” with all the instructions, the program starts by fetching the first instruction, then the instruction decodes at the “Instruction Memory”, after that the program sends the “Opcode” to the “Control Unit”. Then the program sends to the “Register File” the first source “rs”, the second source “rt”, the destination of the result “rd”, the control signal “Memory to Register” and the control signal “RegisterWrite”. After sending every information needed to “Register File”, the program starts by executing the instruction by sending to the “ALUControl” the control signal needed as the “ALUOp” signal from the “Control Unit”, and the “Function Code”. Then the “ALUOperation” receives the result signal “ALUControl”, the value of the first source “rs”, the value of the second source “rt”, the control signal “ALUSource”, the “Offset” and the “Shift Amount”. After that, the program checks the “Opcode”, if it needs to load a byte (unsigned) or load a word or store a byte or store a word and also checks the “Opcode”, if the program needs to jump to another instruction or another function. Then the program goes to the next instruction to execute it and repeat all the stages until the program finishes all the instructions. Finally, the program shows the results of all the control signals used by every instruction.

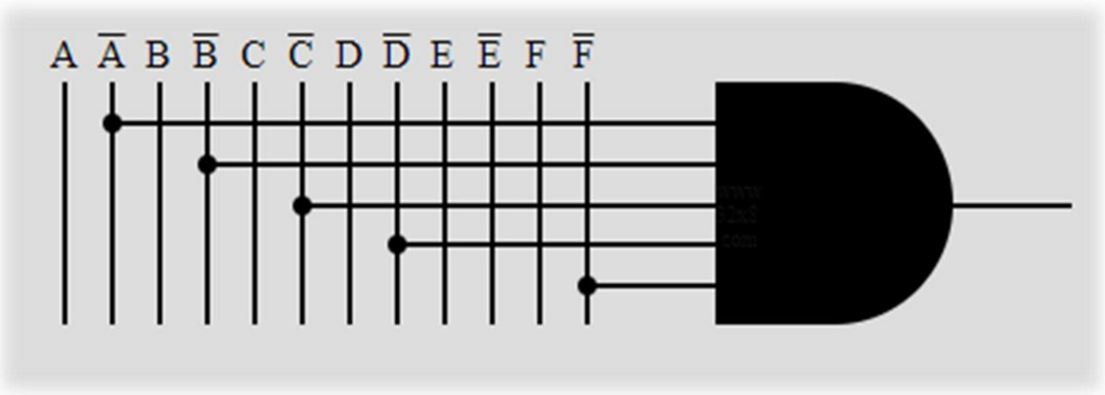
3. Truth table:

opcode						Controls									
						Reg Dest	jump	branch	Mem read	ALU Op		Mem ToReg	Mem Write	ALU Src	RegWrite
0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	X	0	1	X	0	1	X	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	1	0	1	0	0	0	0	0	1	1	0	0	1	1
0	0	0	0	1	0	X	1	0	0	X	X	X	0	X	0
1	0	0	0	1	1	X	1	0	0	X	X	X	0	X	1
1	0	0	0	1	1	0	0	0	1	0	0	1	0	1	1
1	0	1	0	0	0	0	0	0	1	0	0	1	0	1	1
1	0	1	0	0	1	0	0	0	1	0	0	1	0	1	1
1	0	1	0	1	1	X	0	0	0	0	0	X	1	1	0
1	0	0	1	1	1	X	0	0	0	0	0	X	1	1	0

3.1. RegDst

ABC \ DEF	000	001	011	010	100	101	111	110
000	1	0	x	x	x	0	0	0
001	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	x	0
101	0	0	x	0	0	0	0	0
111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0

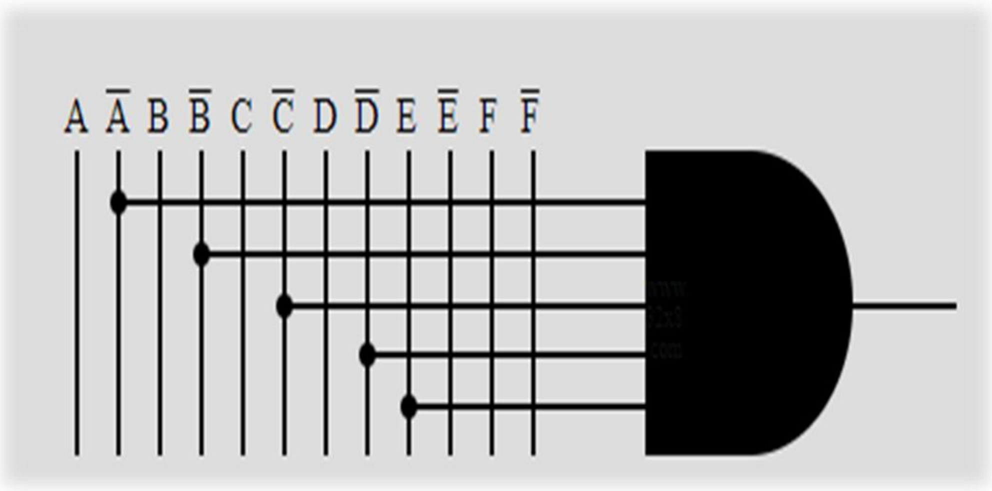
Y=A'B'C'D'F'



3.2. Jump

DEF / ABC	000	001	011	010	100	101	111	110
000	0	0	1	1	0	0	0	0
001	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0
101	0	0	0	0	0	0	0	0
111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0

Y=A'B'C'D'E

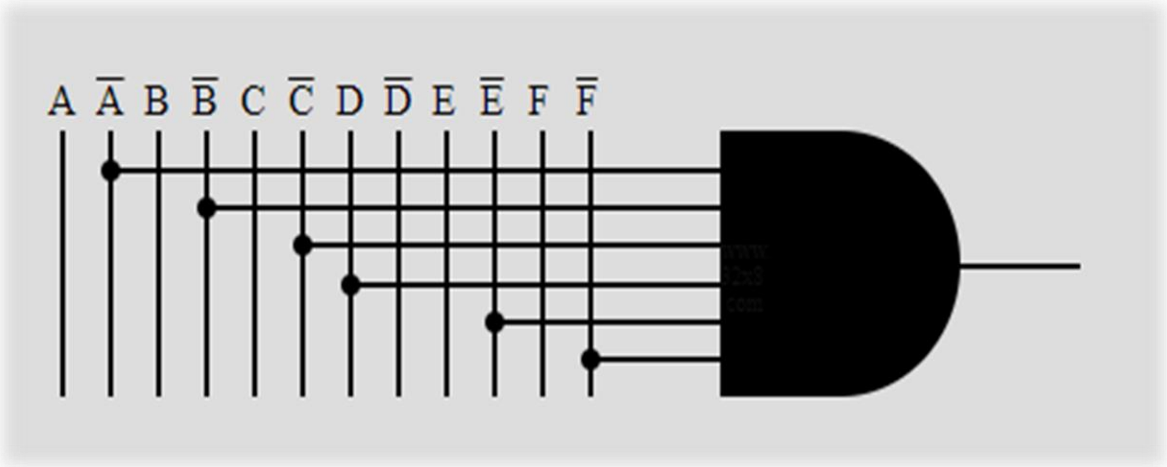


A $\bar{A}$ B $\bar{B}$ C $\bar{C}$ D $\bar{D}$ E $\bar{E}$ F $\bar{F}$

3.3. Branch

ABC \ DEF	000	001	011	010	100	101	111	110
000	0	0	0	0	1	0	0	0
001	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0
101	0	0	0	0	0	0	0	0
111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0

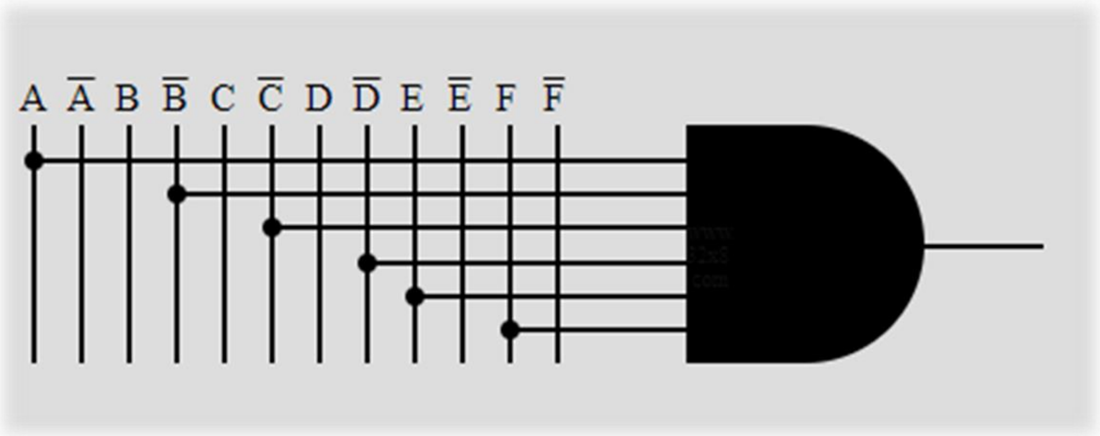
Y=A'B'C'DE'F'



3.4. MemRead

DEF / ABC	000	001	011	010	100	101	111	110
000	0	0	0	0	0	0	0	0
001	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
100	0	0	1	0	0	0	0	0
101	0	0	0	0	0	0	0	0
111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0

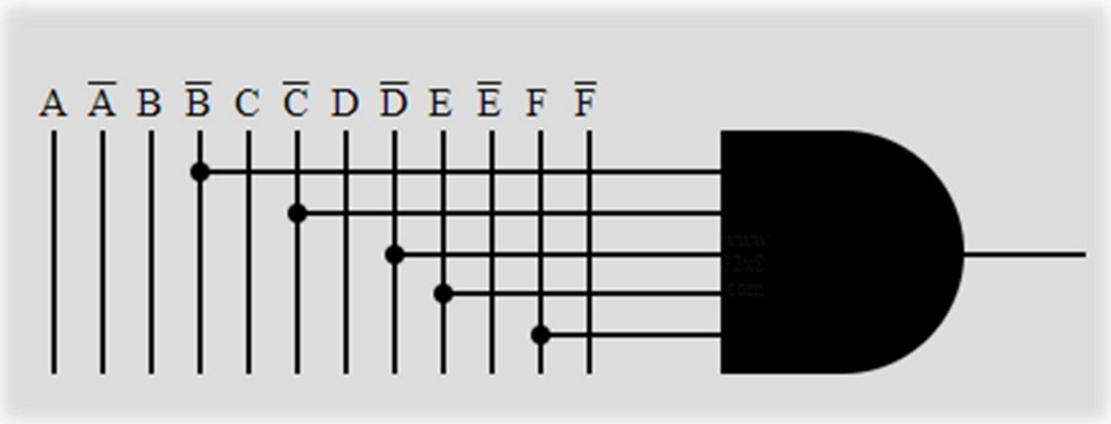
Y=AB’C’D’EF



3.5. MemtoReg

DEF / ABC	000	001	011	010	100	101	111	110
000	0	0	x	x	x	0	0	0
001	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
100	0	0	1	0	0	0	0	0
101	0	0	x	0	0	0	0	0
111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0

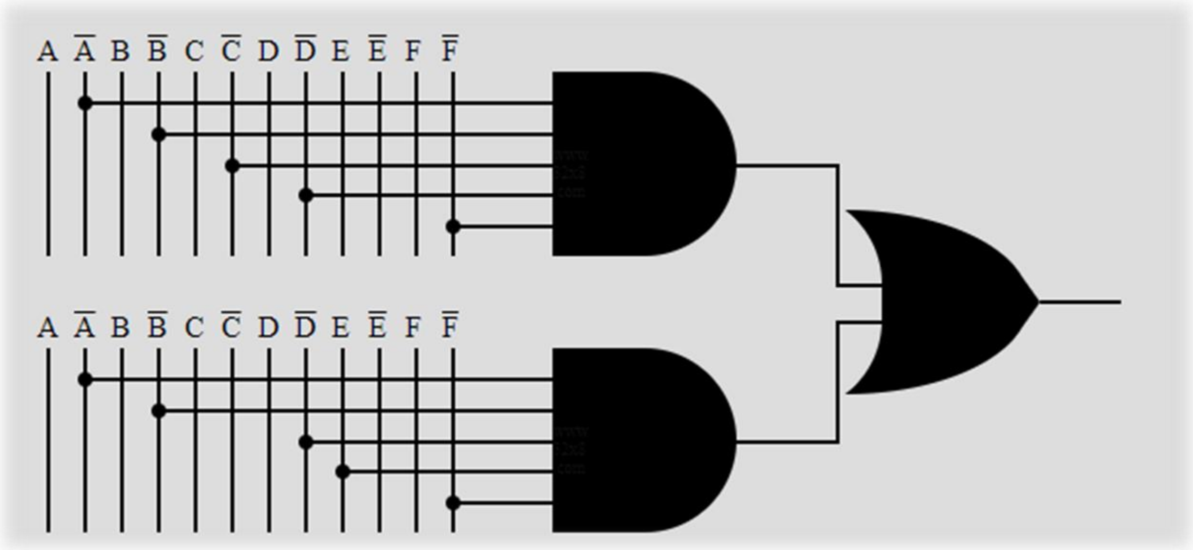
Y=B'C'D'EF



3.6. Alu-op[1]

DEF / ABC	000	001	011	010	100	101	111	110
000	1	0	x	x	0	0	0	0
001	0	0	0	1	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0
101	0	0	0	0	0	0	0	0
111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0

Y=A’B’C’D’F’+A’B’D’EF’

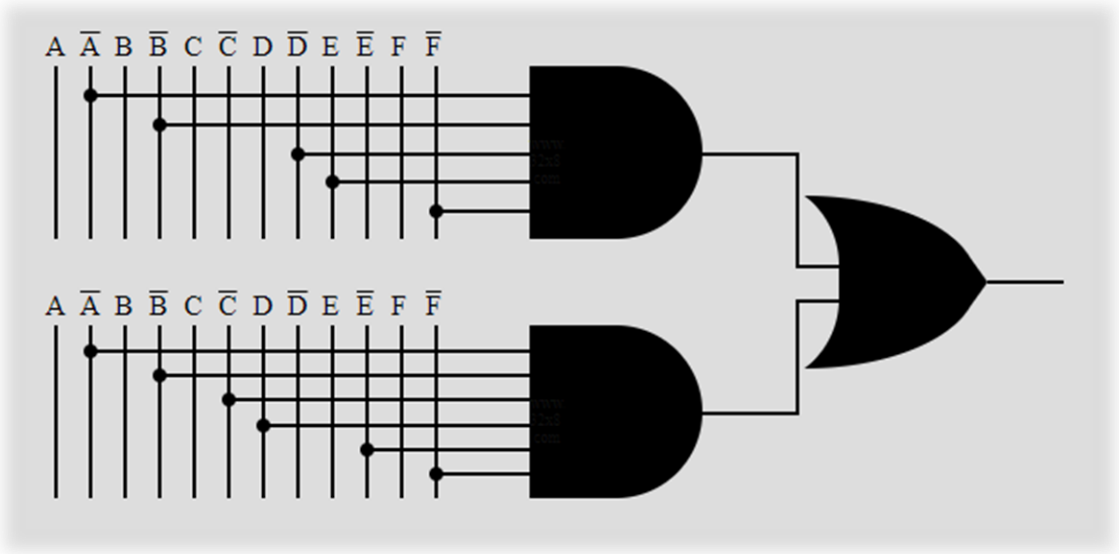


$A \ \bar{A} \ B \ \bar{B} \ C \ \bar{C} \ D \ \bar{D} \ E \ \bar{E} \ F \ \bar{F}$
$A \ \bar{A} \ B \ \bar{B} \ C \ \bar{C} \ D \ \bar{D} \ E \ \bar{E} \ F \ \bar{F}$

3.7. AluOp[2]

ABC \ DEF	000	001	011	010	100	101	111	110
000	0	0	x	x	1	0	0	0
001	0	0	0	1	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0
101	0	0	0	0	0	0	0	0
111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0

Y=A’B’D’EF’+A’B’C’DE’F’

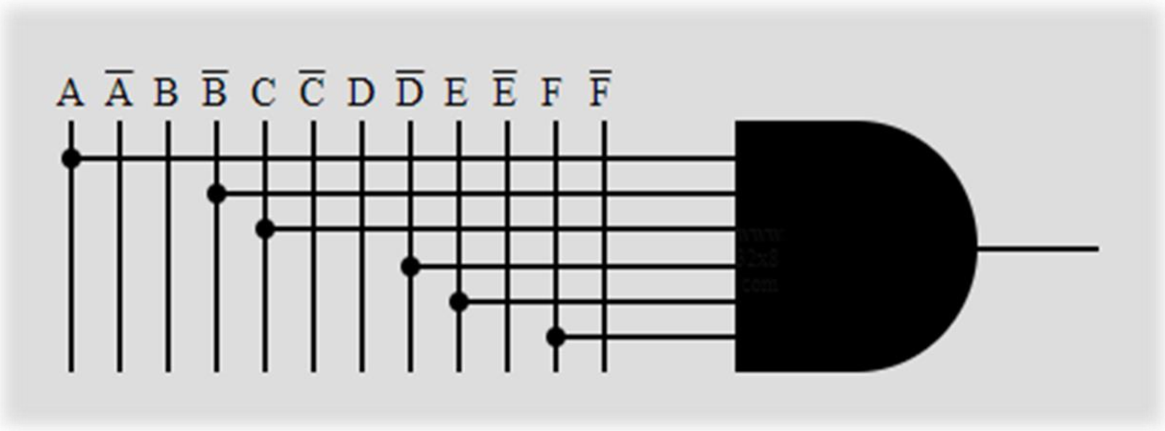


$A\ \bar{A}\ B\ \bar{B}\ C\ \bar{C}\ D\ \bar{D}\ E\ \bar{E}\ F\ \bar{F}$
$A\ \bar{A}\ B\ \bar{B}\ C\ \bar{C}\ D\ \bar{D}\ E\ \bar{E}\ F\ \bar{F}$

3.8. MemWrite

ABC \ DEF	000	001	011	010	100	101	111	110
000	0	0	0	0	0	0	0	0
001	0	0	0	0	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
100	0	0	0	0	0	0	0	0
101	0	0	1	0	0	0	0	0
111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0

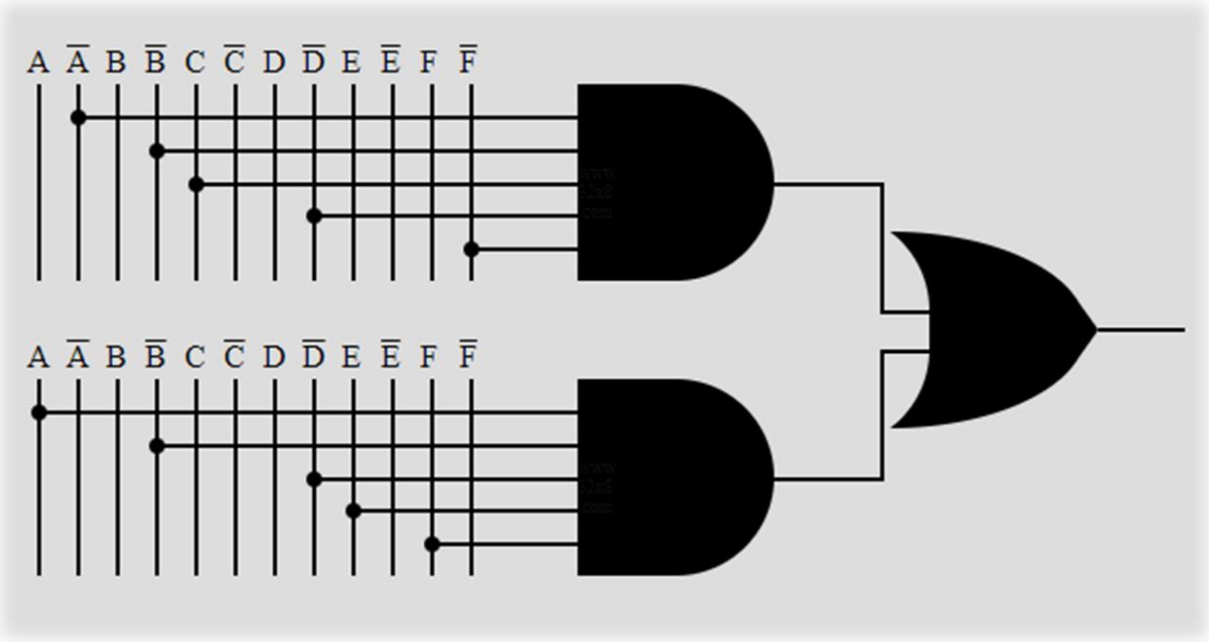
Y=AB’CD’EF



3.9. AluSrc

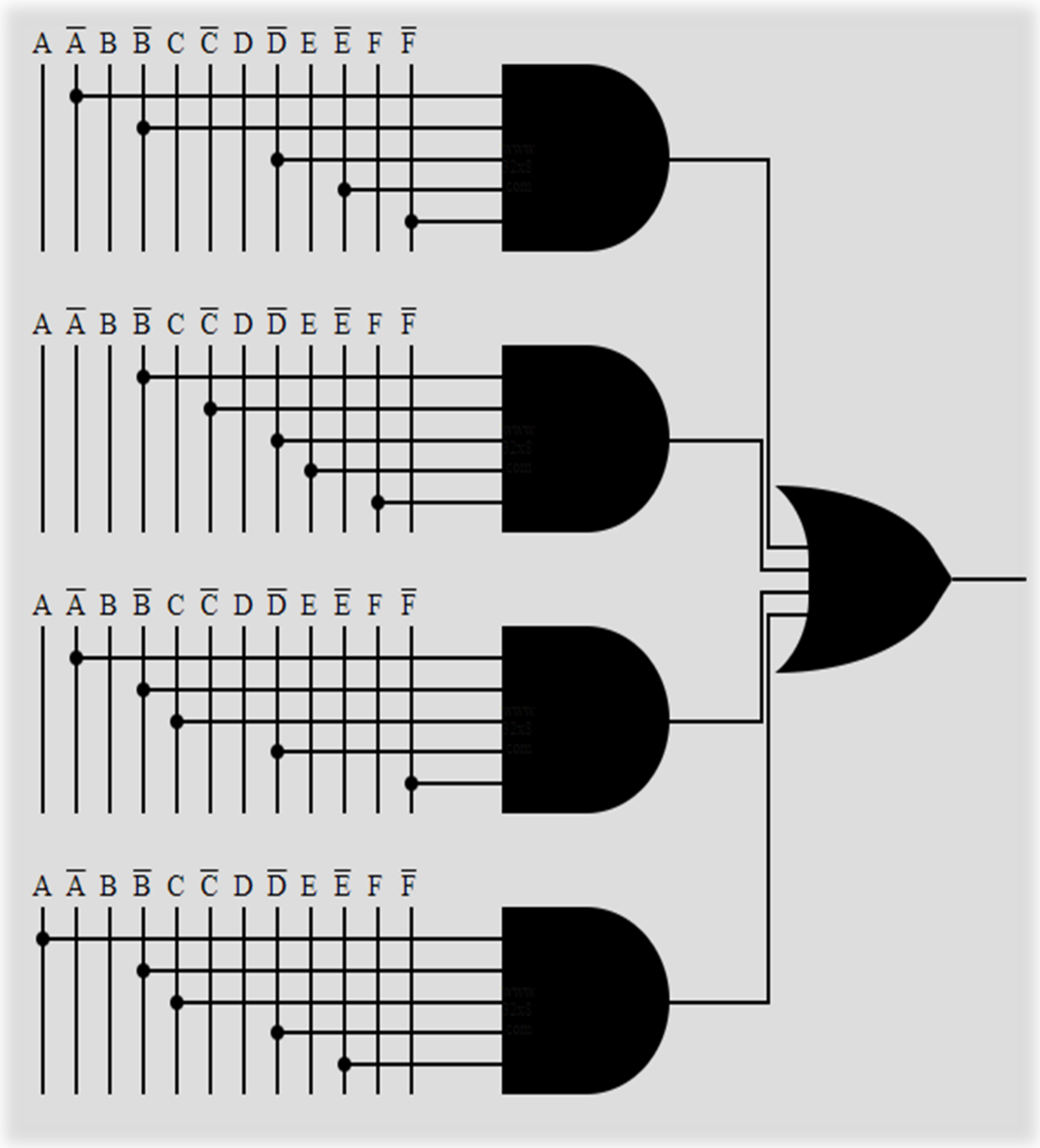
ABC \ DEF	000	001	011	010	100	101	111	110
000	0	0	x	x	0	0	0	0
001	1	0	0	1	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
100	0	0	1	0	0	0	0	0
101	0	0	1	0	0	0	0	0
111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0

Y=A’B’CD’F’+AB’D’EF


$A\ \overline{A}\ B\ \overline{B}\ C\ \overline{C}\ D\ \overline{D}\ E\ \overline{E}\ F\ \overline{F}$
$A\ \overline{A}\ B\ \overline{B}\ C\ \overline{C}\ D\ \overline{D}\ E\ \overline{E}\ F\ \overline{F}$

3.10. RegWrite

ABC \ DEF	000	001	011	010	100	101	111	110
000	1	0	1	0	0	0	0	0
001	1	0	0	1	0	0	0	0
011	0	0	0	0	0	0	0	0
010	0	0	0	0	0	0	0	0
100	0	0	1	0	0	0	0	0
101	1	1	0	0	0	0	0	0
111	0	0	0	0	0	0	0	0
110	0	0	0	0	0	0	0	0

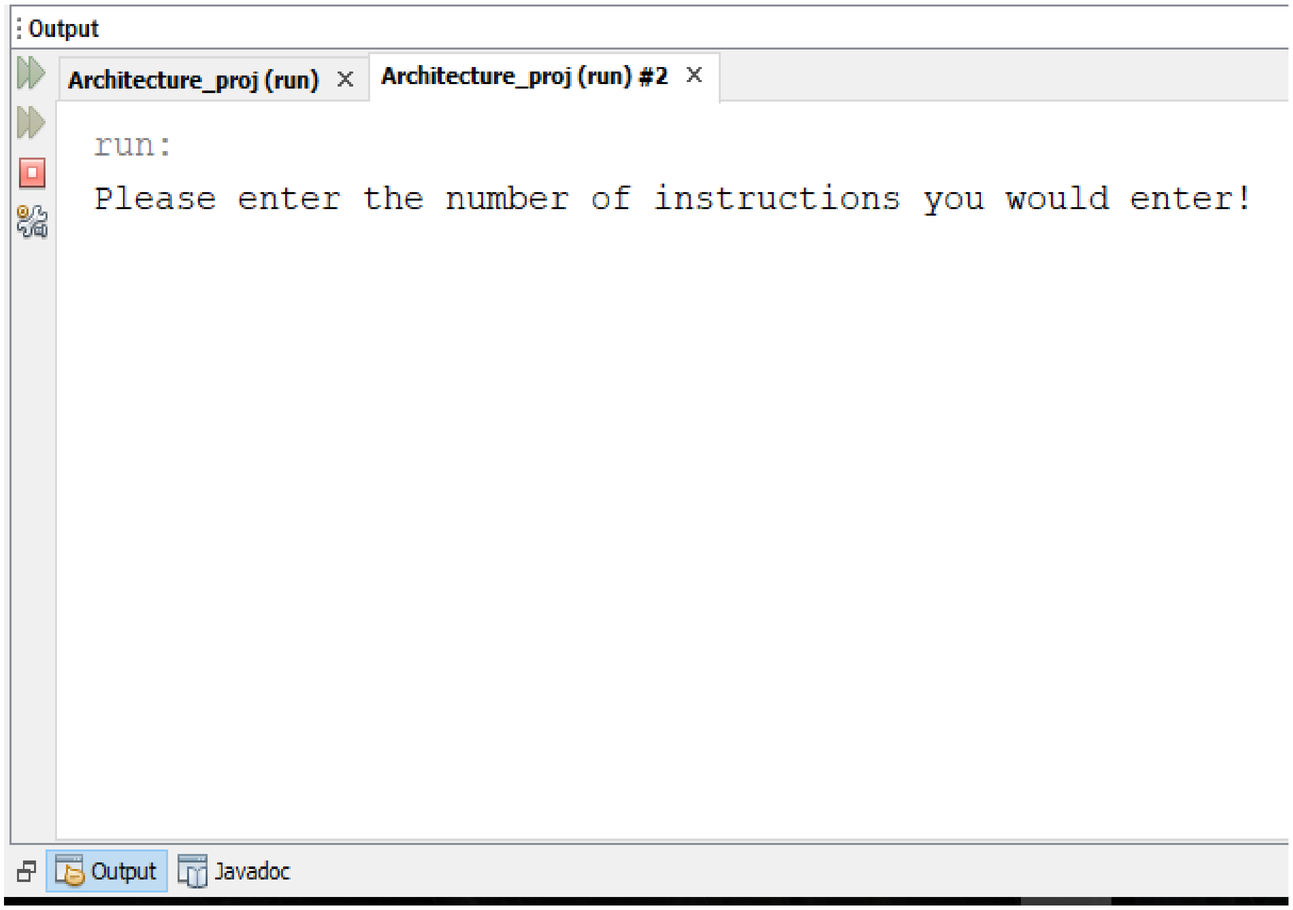

A $\overline{A}$ B $\overline{B}$ C $\overline{C}$ D $\overline{D}$ E $\overline{E}$ F $\overline{F}$
A $\overline{A}$ B $\overline{B}$ C $\overline{C}$ D $\overline{D}$ E $\overline{E}$ F $\overline{F}$
A $\overline{A}$ B $\overline{B}$ C $\overline{C}$ D $\overline{D}$ E $\overline{E}$ F $\overline{F}$
A $\overline{A}$ B $\overline{B}$ C $\overline{C}$ D $\overline{D}$ E $\overline{E}$ F $\overline{F}$

4. The assumptions:

- The “Opcode” of the load byte instruction is:101000(40 in decimal).
- The “Opcode” of the load byte unsigned is: 101001(42 in decimal).
- The “Opcode” of the store byte is: 100111(39 in decimal).
- The control signal “ALUOp” of the “SLTI” instruction equal: 11.
- The “ALUControlFinal” signal of the “SLTI” instruction equal: 0.
- “rs”, “rt”, “rd”, “shift amount” are all integers.
- “Memory” is composed from ArrayList of Integers
- “Memory” size is: an 100
- Each line of the memory represent a byte, ex: Mem[0] holds byte number 1.
- To load a word we load 4 lines of the Memory
- To store a word we store each byte in a line in the memory
- “Instruction Memory” is composed from 1D array of strings.
- “Instruction Memory” size is: [100].
- Class “Instruction” is an abstract class.
- Classes “RFormat”, “IFormat” and “JFormat” inherit from class “Instruction”.

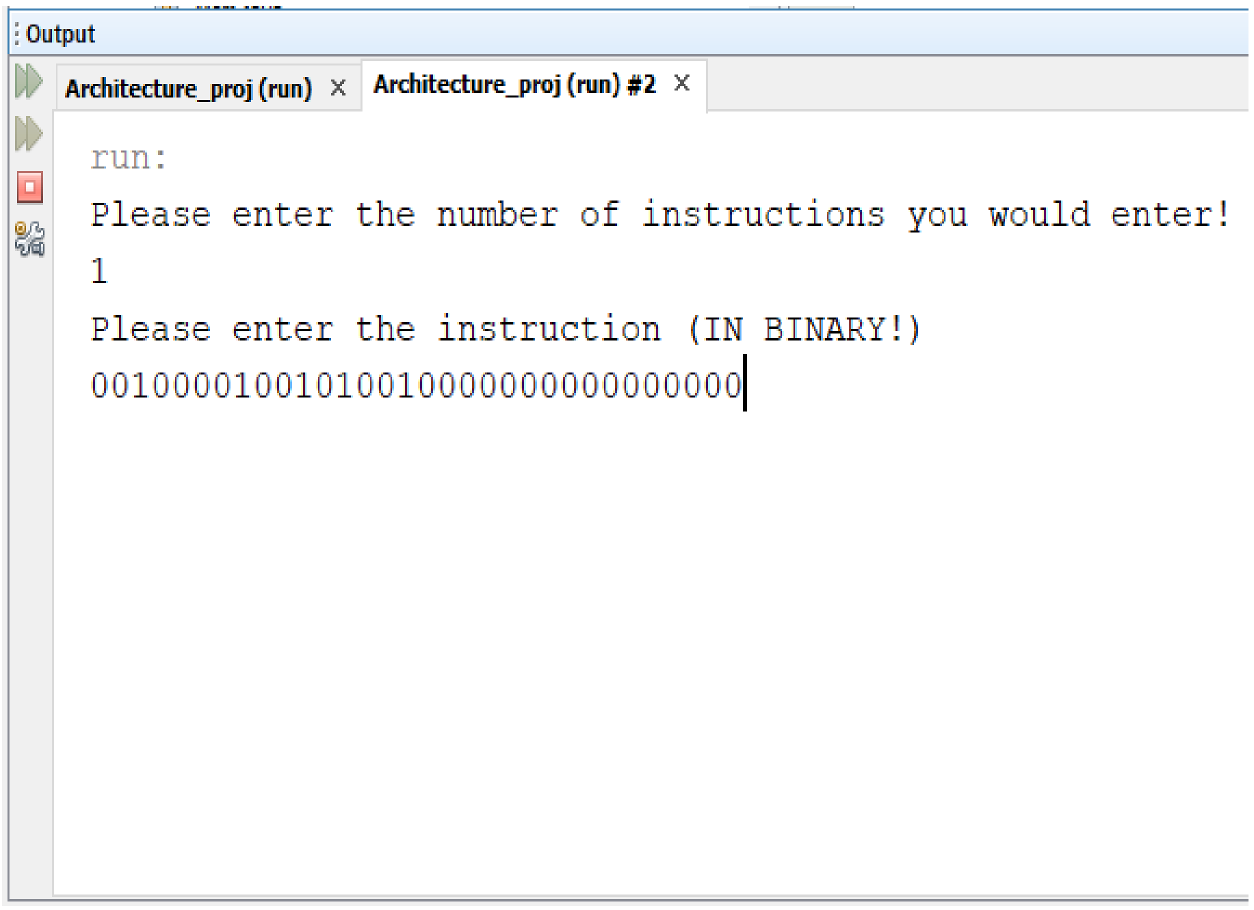
5. User guide:

- First you must enter an integer number representing the number of instructions

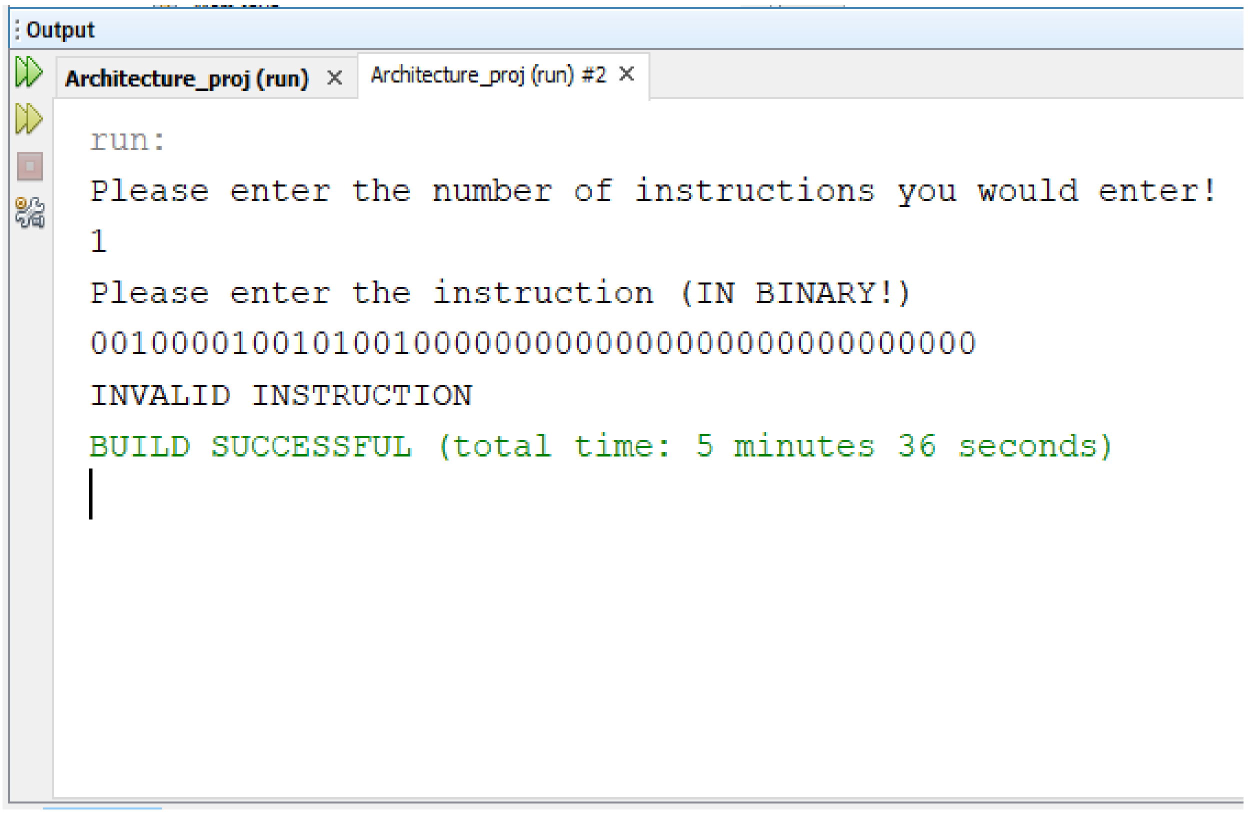


```
run:
Please enter the number of instructions you would enter!
```

- Second, you must fill the Instruction array with you're the number of instructions you have entered. You have to fill each instruction with a String composed of 32 bits (in Binary) Otherwise , a message will appear on the console
  "INVALID INSTRUCTION"



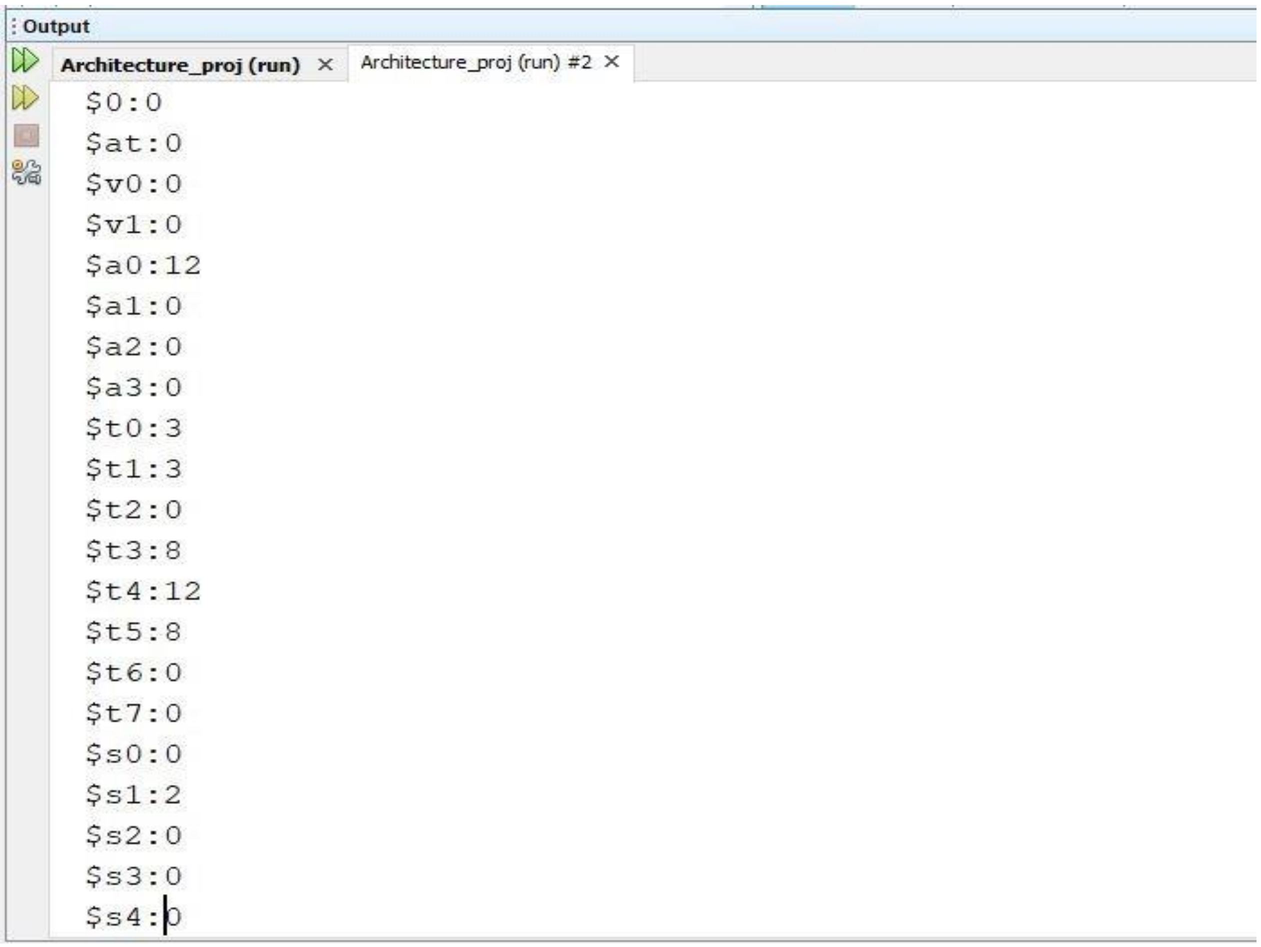
```
run:
Please enter the number of instructions you would enter!
1
Please enter the instruction (IN BINARY!)
00100001001010010000000000000000
```


Output

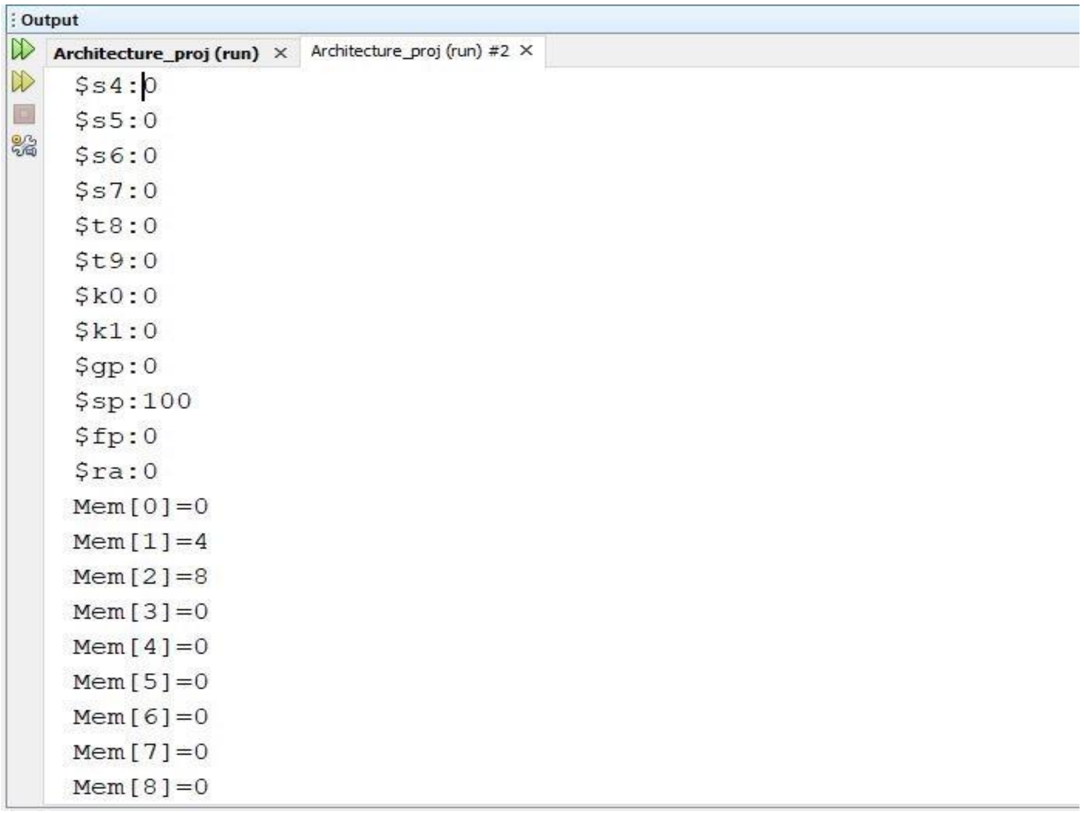
Architecture_proj (run) × Architecture_proj (run) #2 ×

```
run:
Please enter the number of instructions you would enter!
1
Please enter the instruction (IN BINARY!)
00100001001010010000000000000000000000000000
INVALID INSTRUCTION
BUILD SUCCESSFUL (total time: 5 minutes 36 seconds)
```

➢ After the execution of the instructions entered, The Controls of each instruction and the registers used(rs,rt,rd) are printed in the Console, and the Register File (name and values of the 32 Registers) and the Memory. So, you could find the change of the values in the Register File or the memory after the execution.



```
$0:0
$at:0
$v0:0
$v1:0
$a0:12
$a1:0
$a2:0
$a3:0
$t0:3
$t1:3
$t2:0
$t3:8
$t4:12
$t5:8
$t6:0
$t7:0
$s0:0
$s1:2
$s2:0
$s3:0
$s4:0
```



```
$s4:0
$s5:0
$s6:0
$s7:0
$t8:0
$t9:0
$k0:0
$k1:0
$gp:0
$sp:100
$fp:0
$ra:0
Mem[0]=0
Mem[1]=4
Mem[2]=8
Mem[3]=0
Mem[4]=0
Mem[5]=0
Mem[6]=0
Mem[7]=0
Mem[8]=0
```

List of programs:

Program 1: a0 should finally be equal to (45 decimal)

```
00100000000010000000000000000000
00100000000010010000000000001100
00010001000010010000000000000101
00000001000000000101100010000000
00000010000010001000100000100000
10101110001010110000000000000000
00100001000010000000000000000100
00001000000000000000000000000010
00000000000000000100000000100000
00100000000001100000000000000000
00010001000010010000000000000101
00000010000010001000100000100000
10001110001011010000000000000000
00000001100011010110000000100000
00100001000010000000000000000100
00001000000000000000000000001010
00100001100001000000000000000000
```

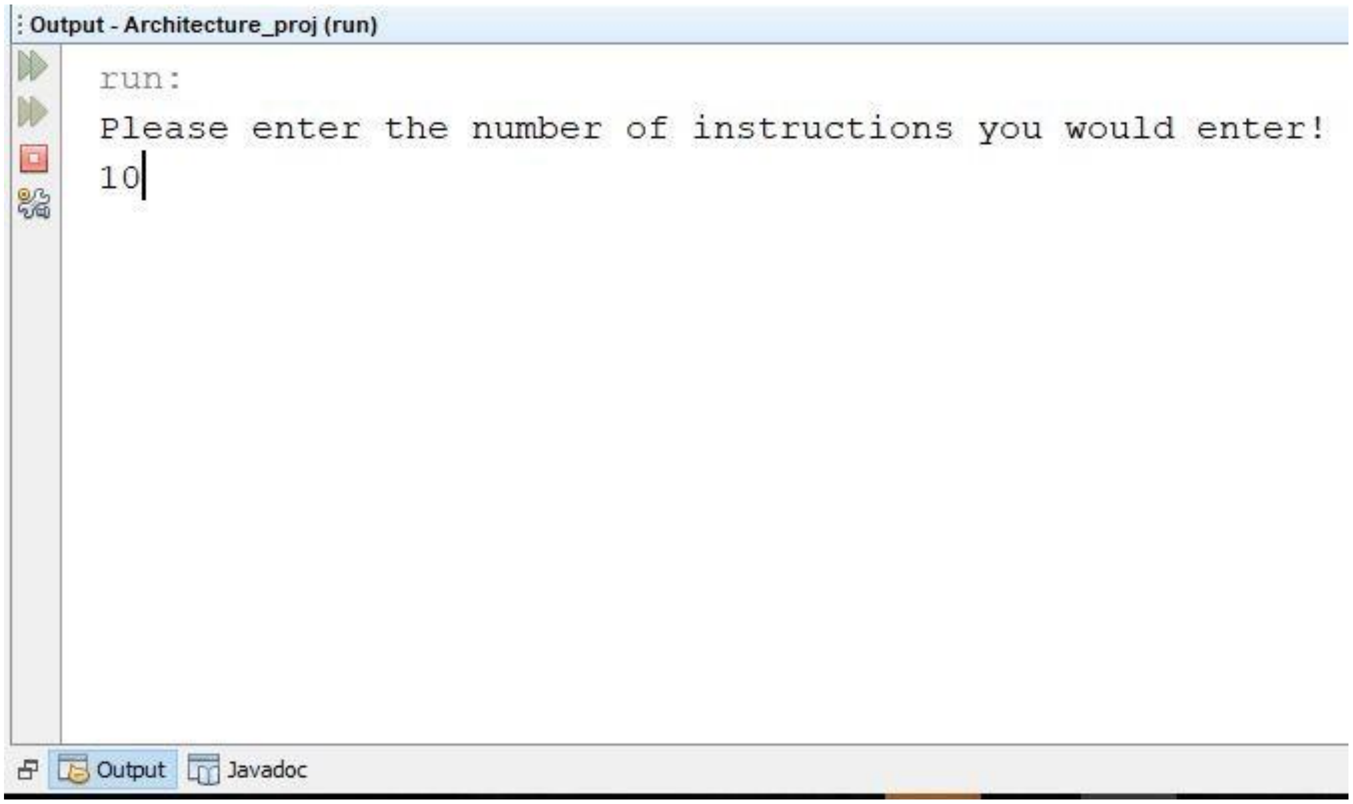
Assembly:

```
Addi $t4 $t4 , 1
addi $t1 ,$t1 ,0  # sum=0
addi $t2 , $t2 , 0 #i=0
Loop : slti $t3 , $t2 , 10 # if i<10 $ t3 = 1
beq $t3 $t4 label 1 if(t3=0) e
jump exit
label1 : add $t1 $t1 $t2 #sum=sum+i
```

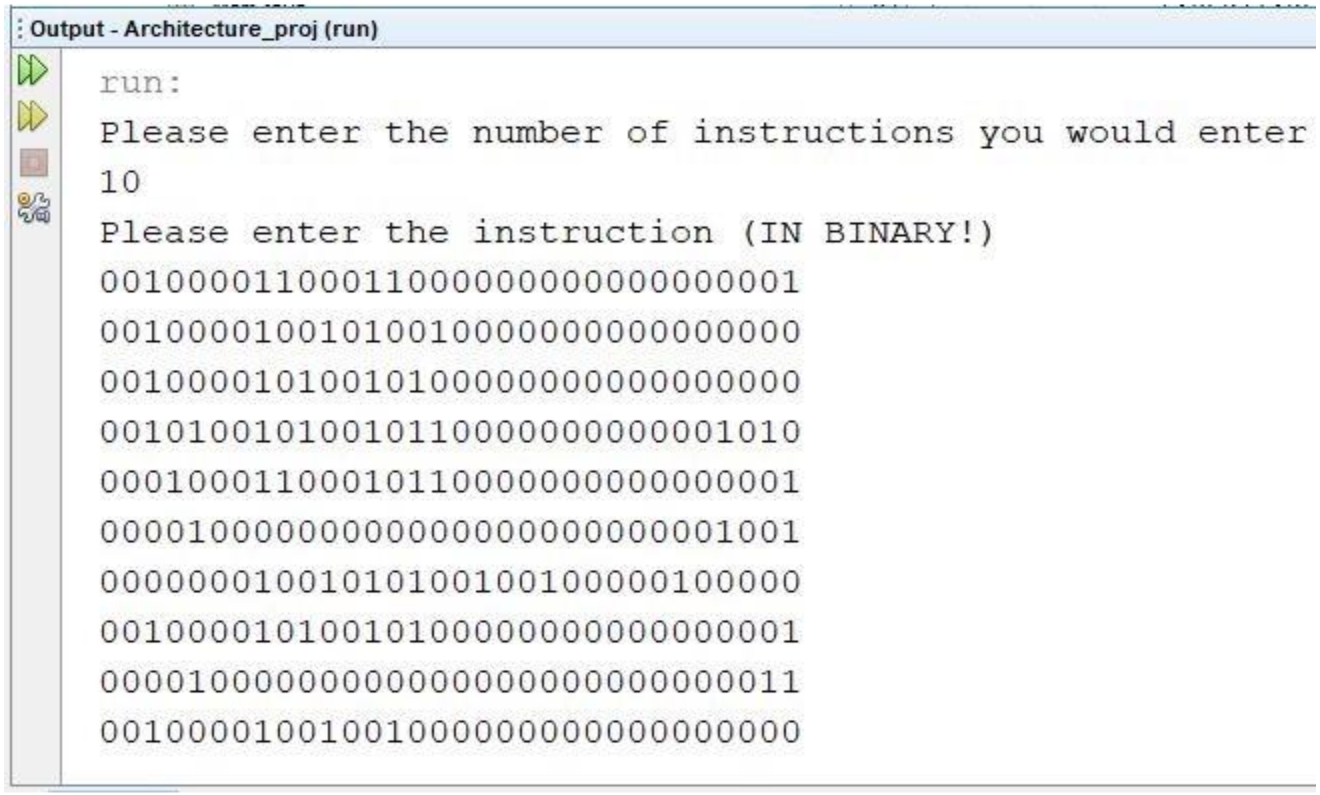
addi $t2 , $t2 , 1 #i++

jump loop

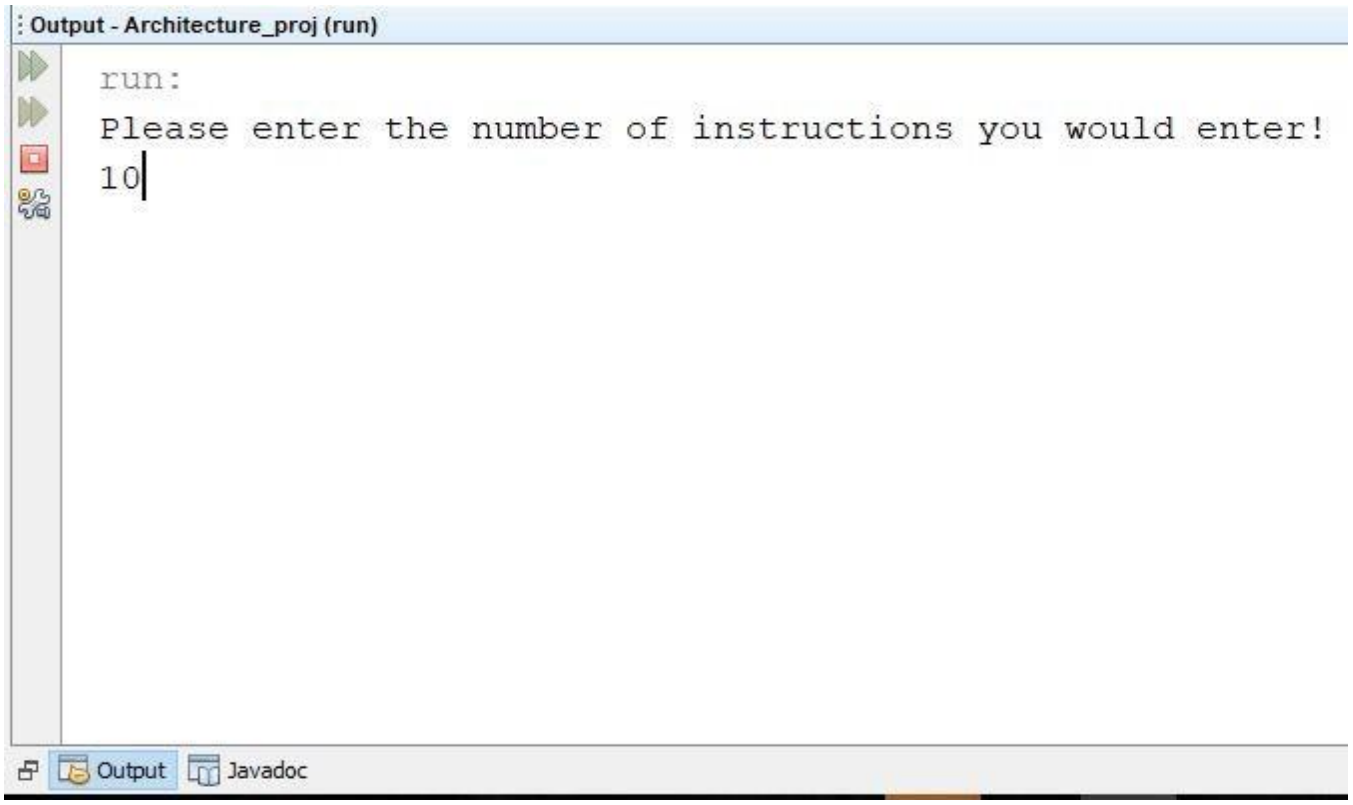
exit : Addi $a0 ,$t1,0



```
run:
Please enter the number of instructions you would enter!
10
```

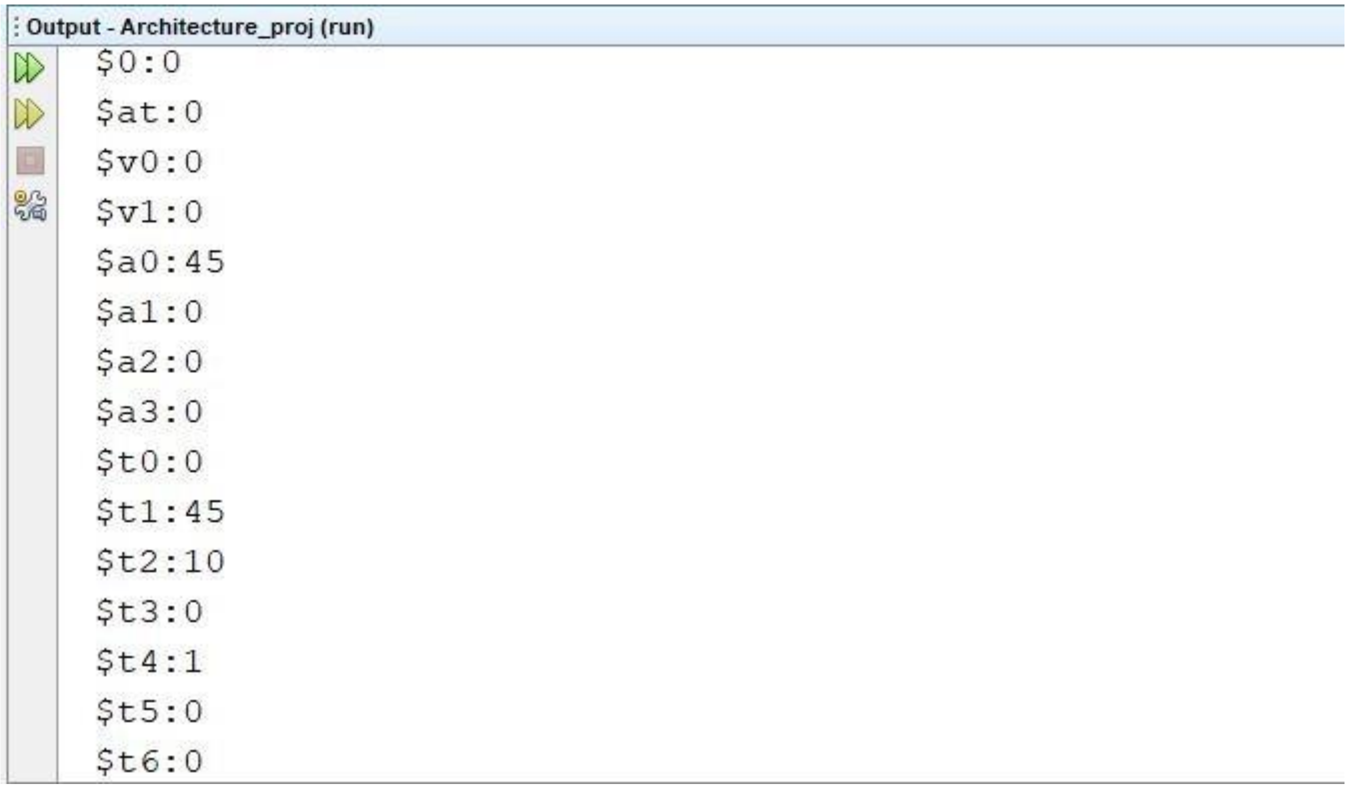

Output - Architecture_proj (run)

```
run:
Please enter the number of instructions you would enter
10
Please enter the instruction (IN BINARY!)
00100001100011000000000000000001
00100001001010010000000000000000
00100001010010100000000000000000
00101001010010110000000000001010
00010001100010110000000000000001
00001000000000000000000000001001
00000001001010100100100000100000
00100001010010100000000000000001
00001000000000000000000000000011
00100001001001000000000000000000
```

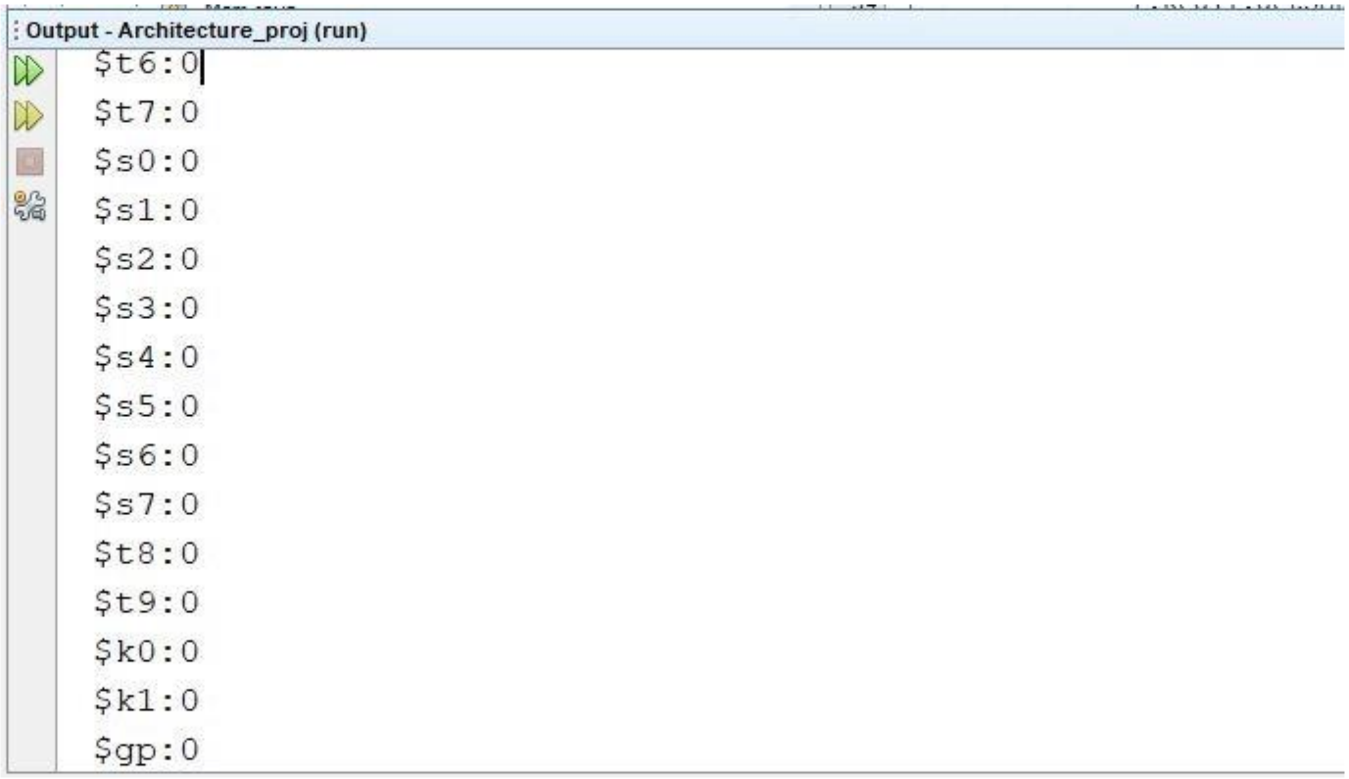

Output - Architecture_proj (run)

```
run:
Please enter the number of instructions you would enter!
10
```

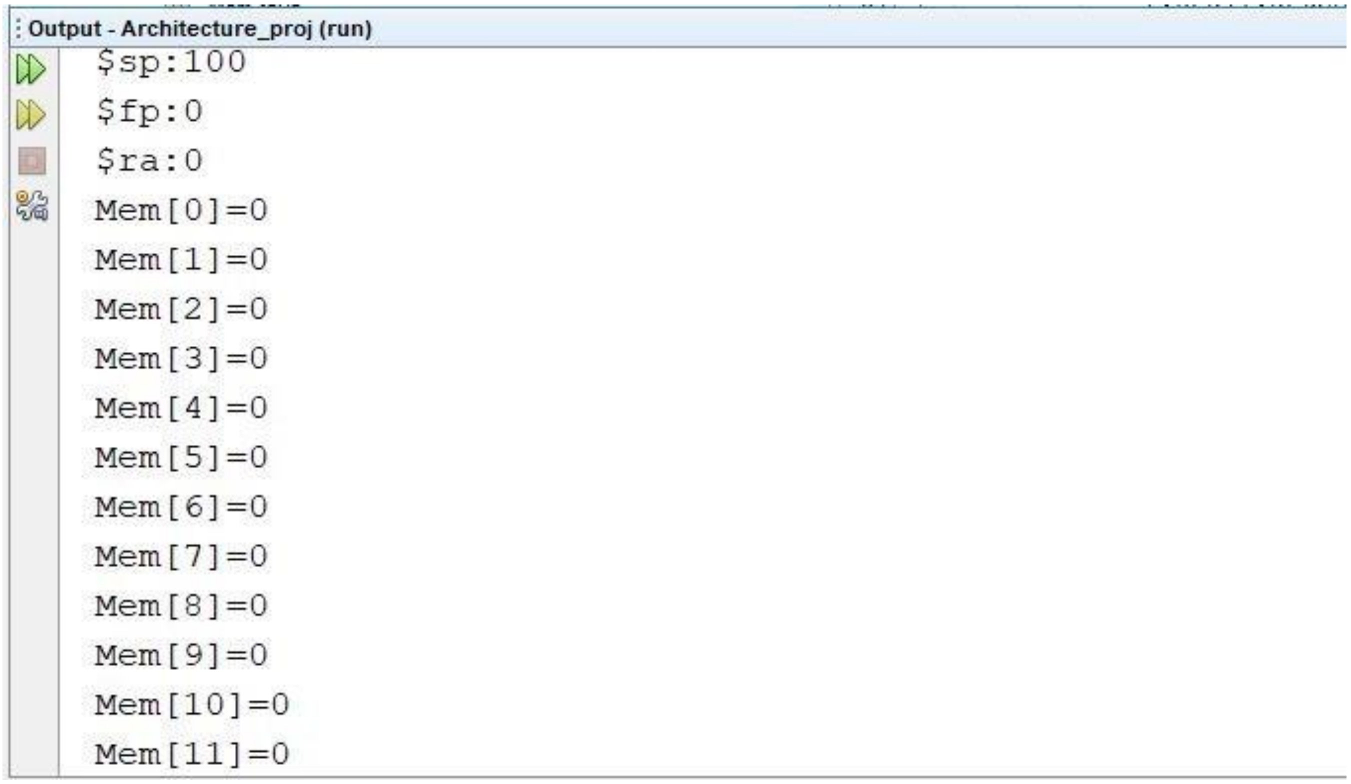
Output Javadoc


Output - Architecture_proj (run)

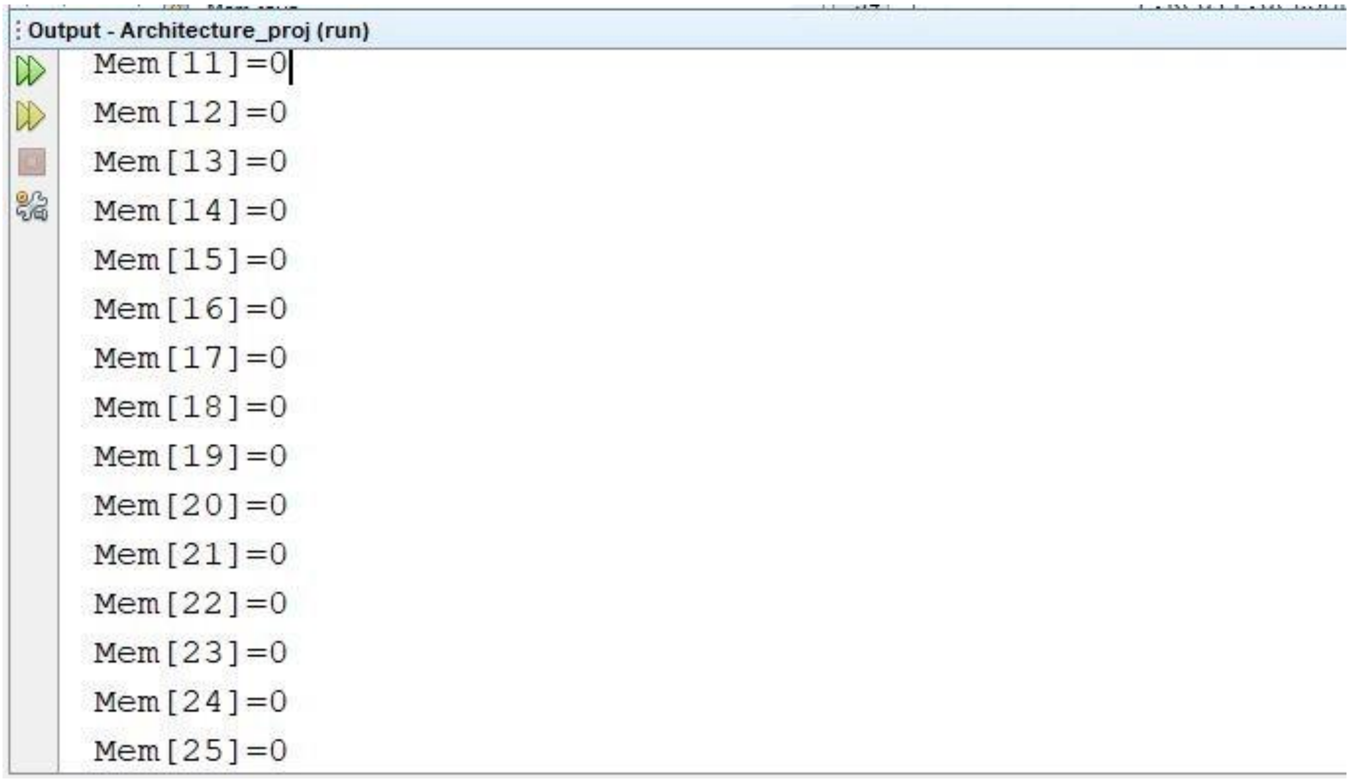
```
$0:0
$at:0
$v0:0
$v1:0
$a0:45
$a1:0
$a2:0
$a3:0
$t0:0
$t1:45
$t2:10
$t3:0
$t4:1
$t5:0
$t6:0
```


Output - Architecture_proj (run)

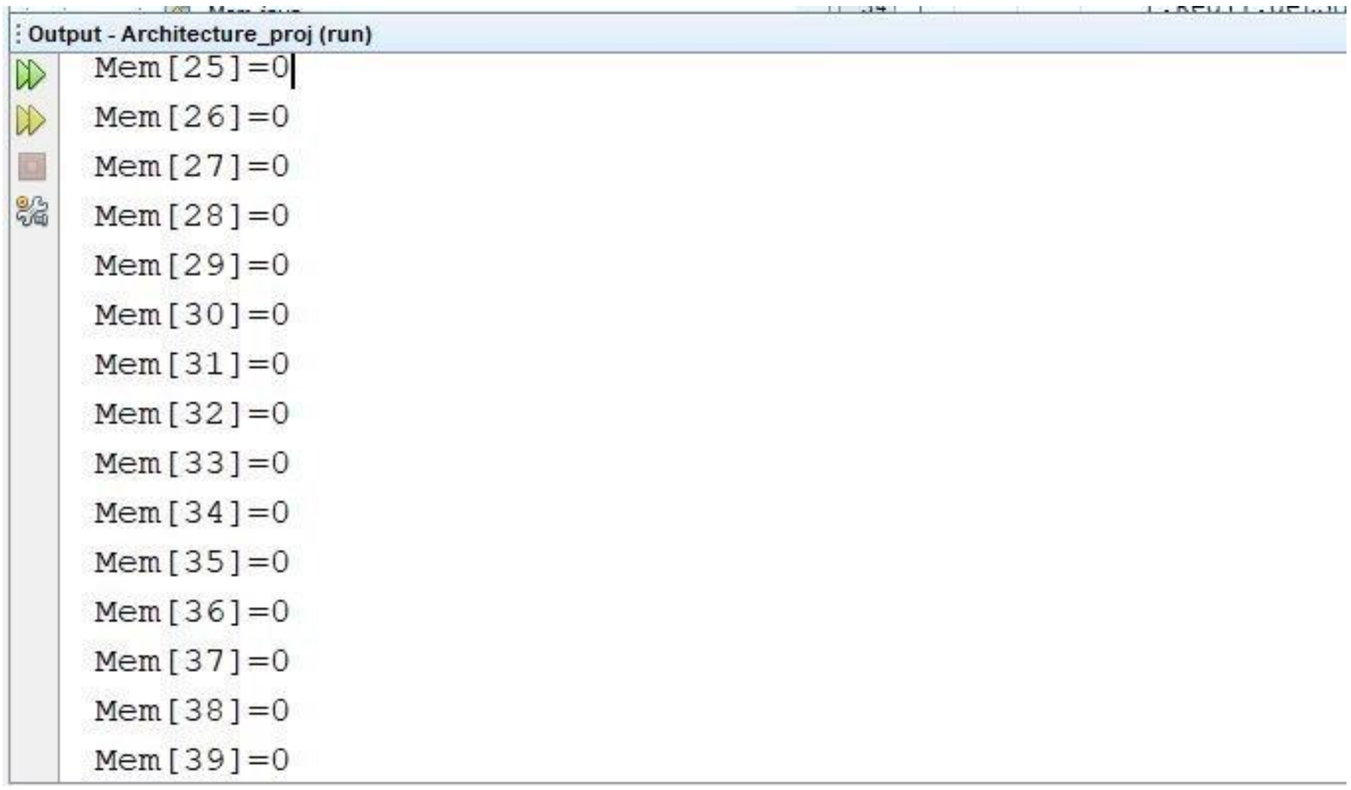
```
$t6:0
$t7:0
$s0:0
$s1:0
$s2:0
$s3:0
$s4:0
$s5:0
$s6:0
$s7:0
$t8:0
$t9:0
$k0:0
$k1:0
$gp:0
```


Output - Architecture_proj (run)

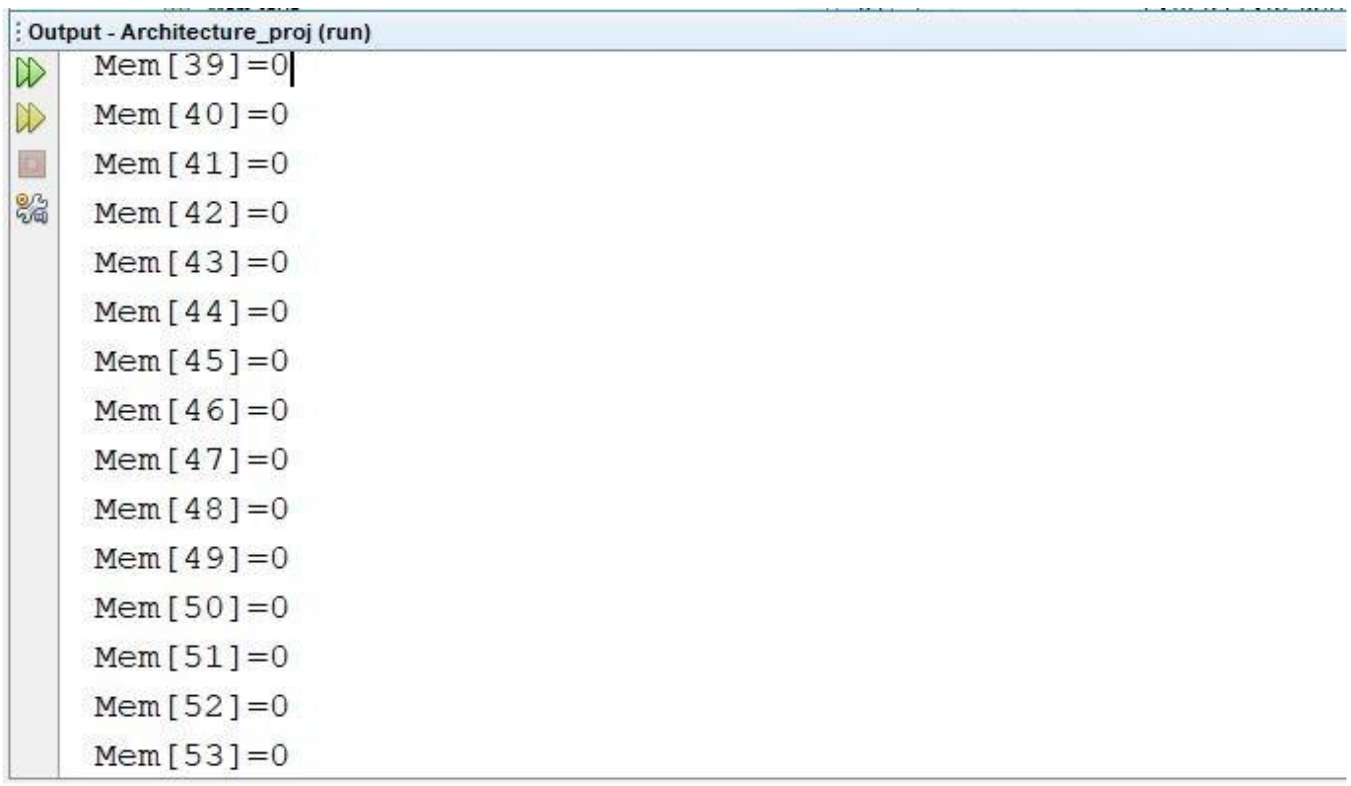
```
$sp:100
$fp:0
$ra:0
Mem[0]=0
Mem[1]=0
Mem[2]=0
Mem[3]=0
Mem[4]=0
Mem[5]=0
Mem[6]=0
Mem[7]=0
Mem[8]=0
Mem[9]=0
Mem[10]=0
Mem[11]=0
```


Output - Architecture_proj (run)

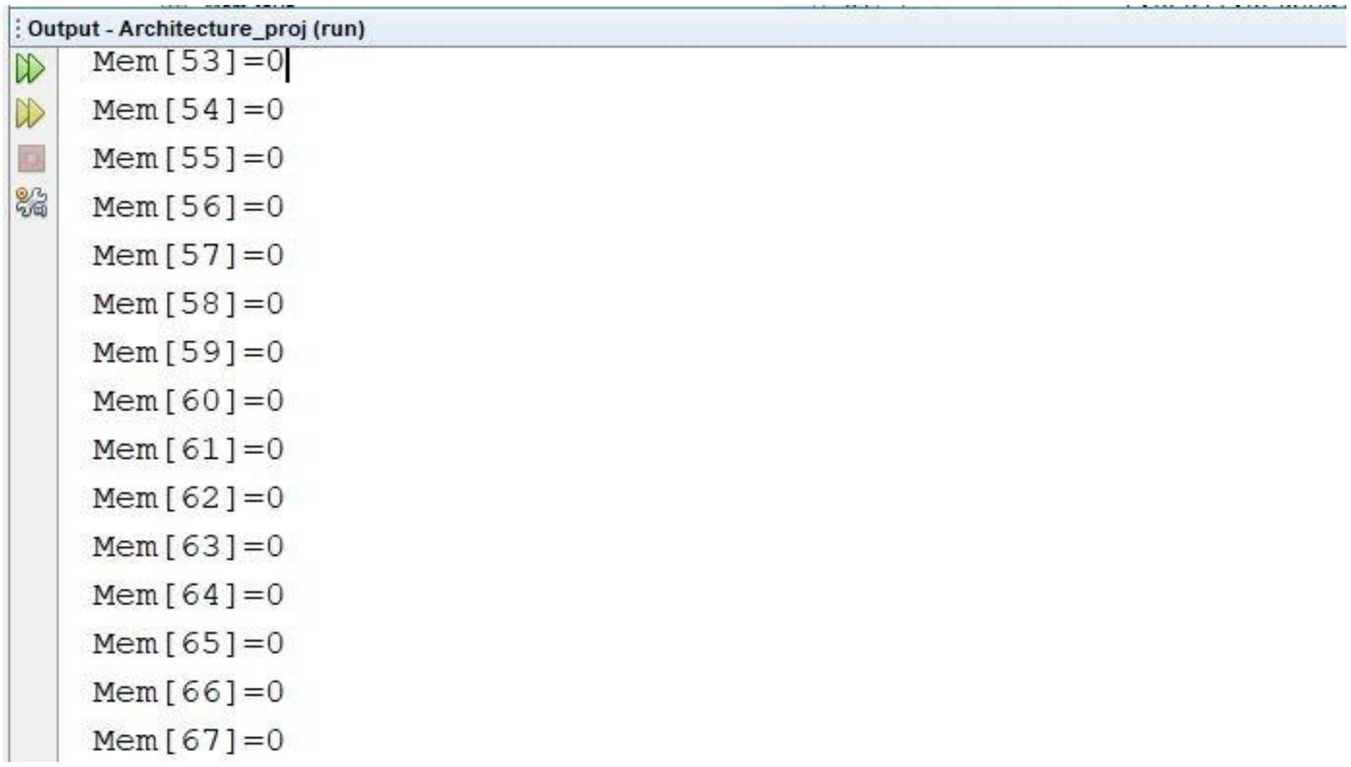
```
Mem[11]=0
Mem[12]=0
Mem[13]=0
Mem[14]=0
Mem[15]=0
Mem[16]=0
Mem[17]=0
Mem[18]=0
Mem[19]=0
Mem[20]=0
Mem[21]=0
Mem[22]=0
Mem[23]=0
Mem[24]=0
Mem[25]=0
```


Output - Architecture_proj (run)

```
Mem[25]=0
Mem[26]=0
Mem[27]=0
Mem[28]=0
Mem[29]=0
Mem[30]=0
Mem[31]=0
Mem[32]=0
Mem[33]=0
Mem[34]=0
Mem[35]=0
Mem[36]=0
Mem[37]=0
Mem[38]=0
Mem[39]=0
```

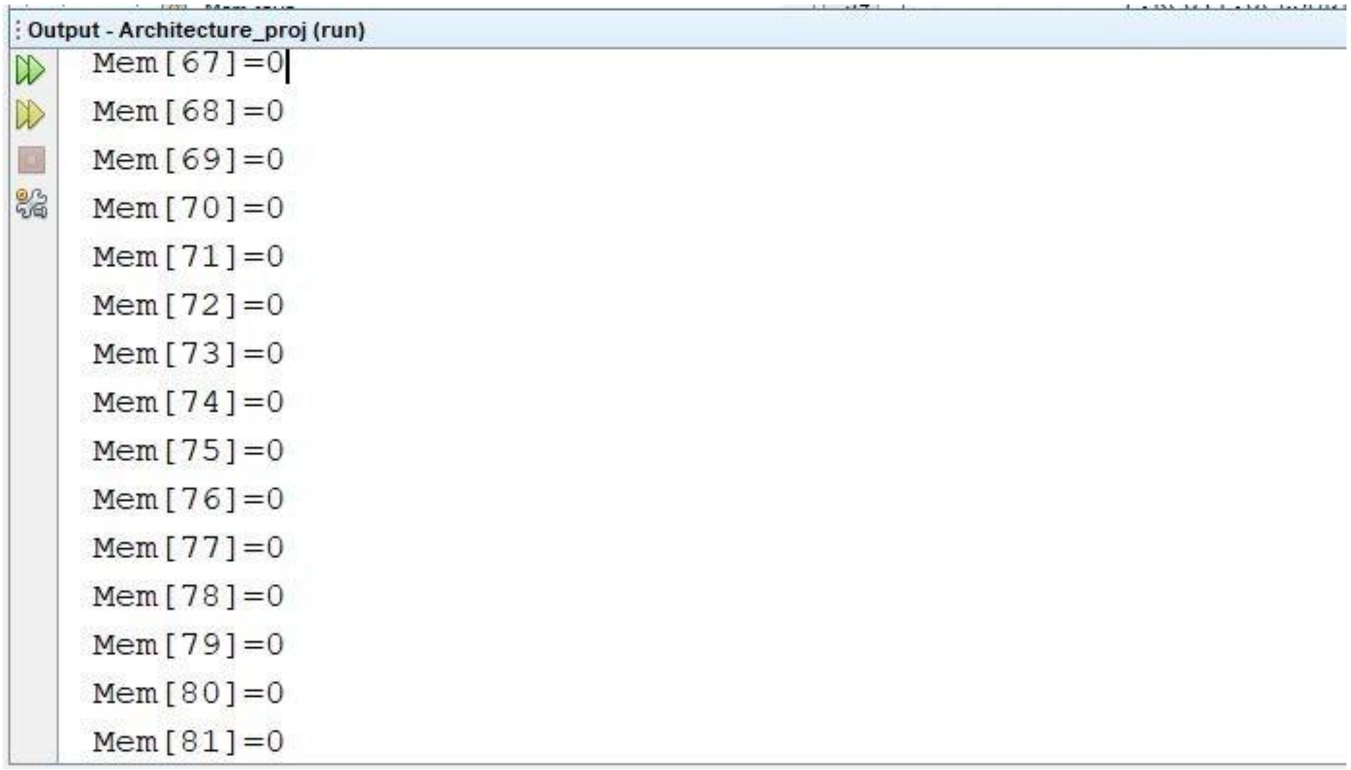

Output - Architecture_proj (run)

```
Mem[39]=0
Mem[40]=0
Mem[41]=0
Mem[42]=0
Mem[43]=0
Mem[44]=0
Mem[45]=0
Mem[46]=0
Mem[47]=0
Mem[48]=0
Mem[49]=0
Mem[50]=0
Mem[51]=0
Mem[52]=0
Mem[53]=0
```



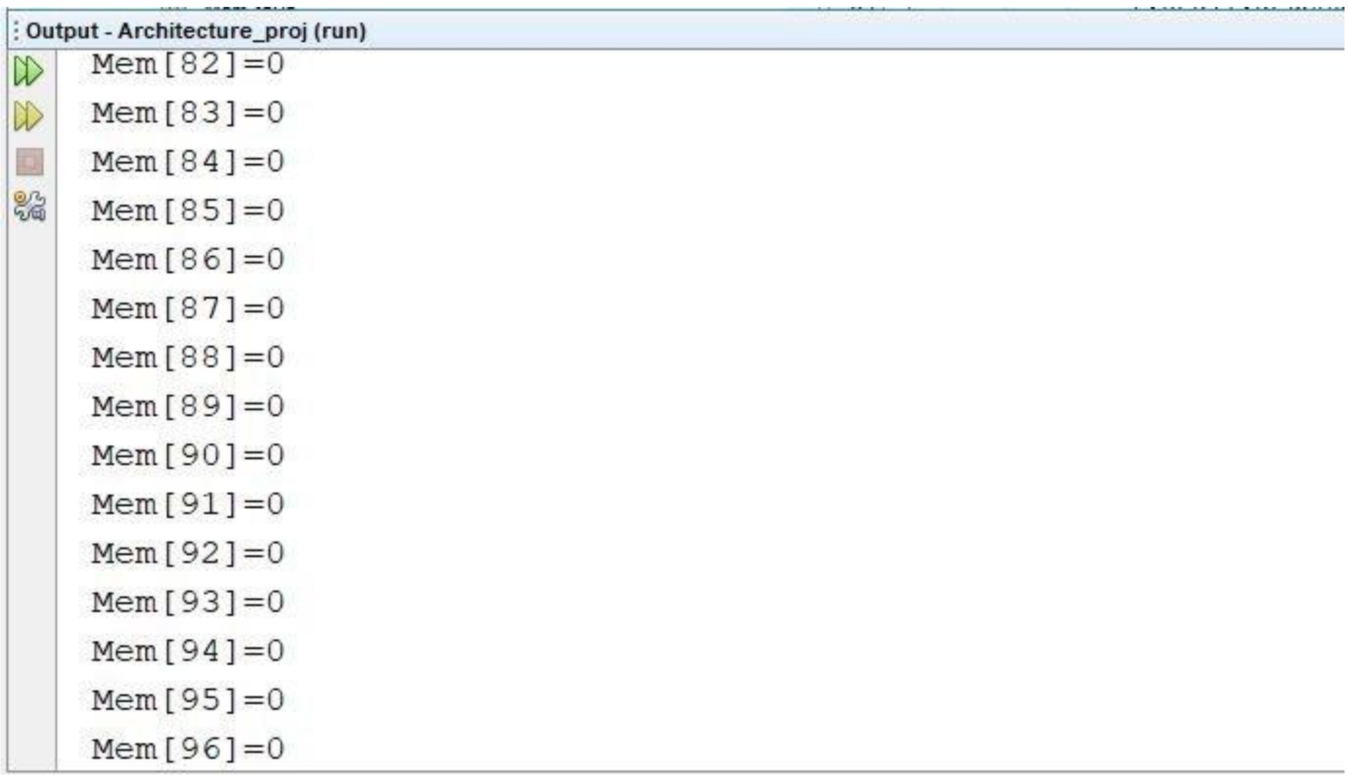
Output - Architecture_proj (run)

```
Mem[53]=0
Mem[54]=0
Mem[55]=0
Mem[56]=0
Mem[57]=0
Mem[58]=0
Mem[59]=0
Mem[60]=0
Mem[61]=0
Mem[62]=0
Mem[63]=0
Mem[64]=0
Mem[65]=0
Mem[66]=0
Mem[67]=0
```

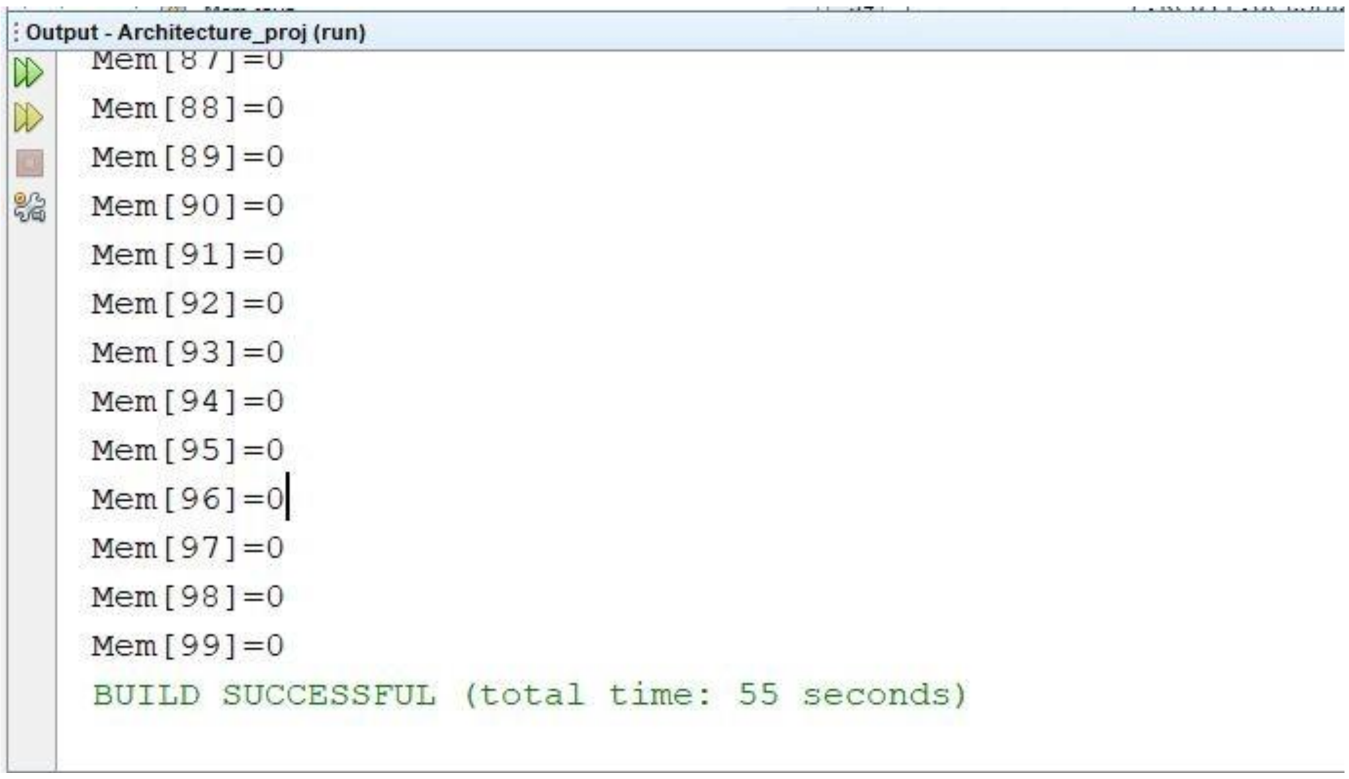


Output - Architecture_proj (run)

```
Mem[67]=0
Mem[68]=0
Mem[69]=0
Mem[70]=0
Mem[71]=0
Mem[72]=0
Mem[73]=0
Mem[74]=0
Mem[75]=0
Mem[76]=0
Mem[77]=0
Mem[78]=0
Mem[79]=0
Mem[80]=0
Mem[81]=0
```



```
Mem[82]=0
Mem[83]=0
Mem[84]=0
Mem[85]=0
Mem[86]=0
Mem[87]=0
Mem[88]=0
Mem[89]=0
Mem[90]=0
Mem[91]=0
Mem[92]=0
Mem[93]=0
Mem[94]=0
Mem[95]=0
Mem[96]=0
```



```
Mem[87]=0
Mem[88]=0
Mem[89]=0
Mem[90]=0
Mem[91]=0
Mem[92]=0
Mem[93]=0
Mem[94]=0
Mem[95]=0
Mem[96]=0
Mem[97]=0
Mem[98]=0
Mem[99]=0
BUILD SUCCESSFUL (total time: 55 seconds)
```

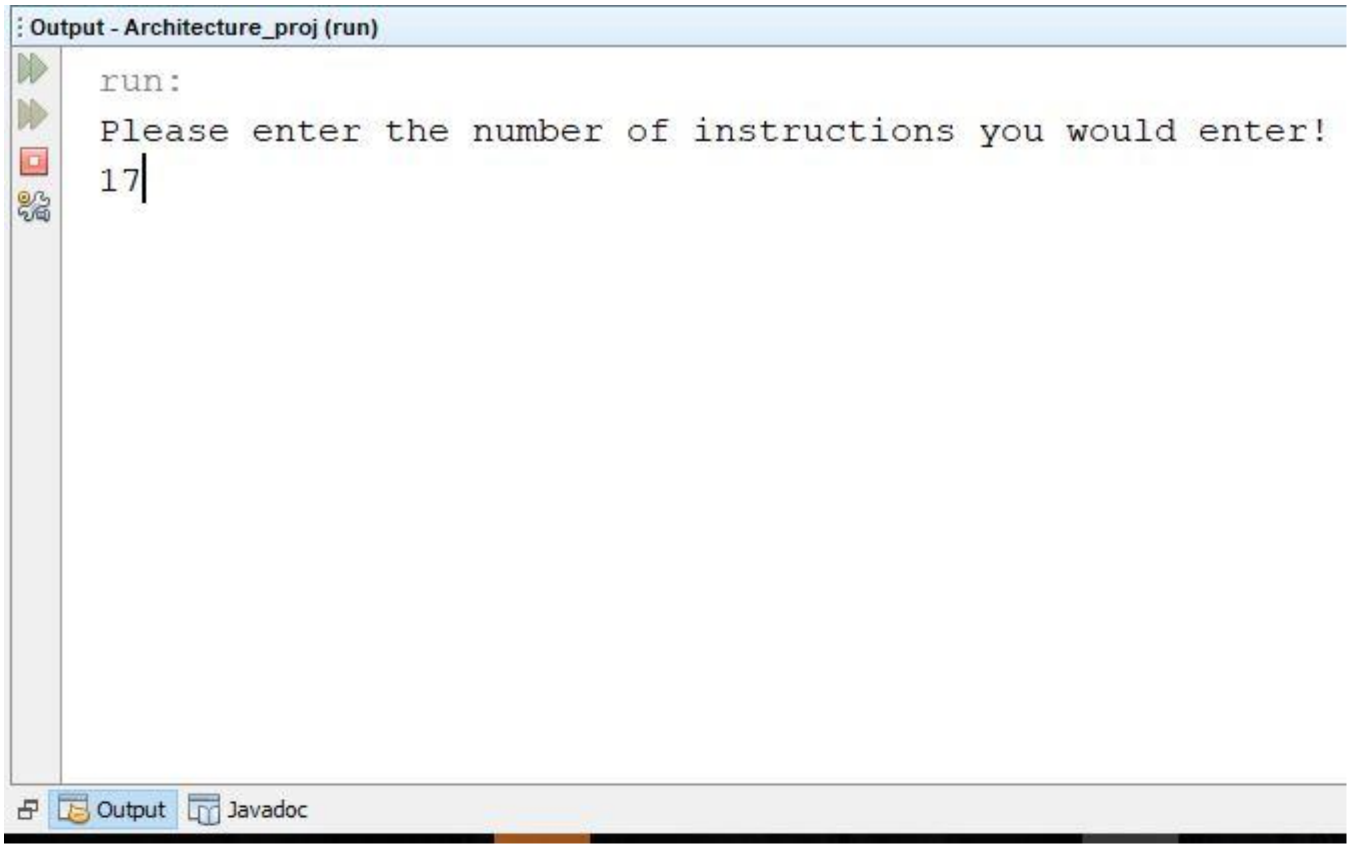
<u>Program 2: a0 should finally be equal to 0xc (12 decimal)</u>

```
00100000000010000000000000000000

00100000000010010000000000001100

00010001000010010000000000000101

00000001000000000101100010000000

00000010000010001000100000100000

10101110001010110000000000000000

00100001000010000000000000000100

00001000000000000000000000000010

00000000000000001000000000100000

00100000000011000000000000000000

00010001000010010000000000000101

00000010000010001000100000100000

10001110001011010000000000000000

00000001100011010110000000100000

00100001000010000000000000000100

00001000000000000000000000001010

00100001100001000000000000000000
```

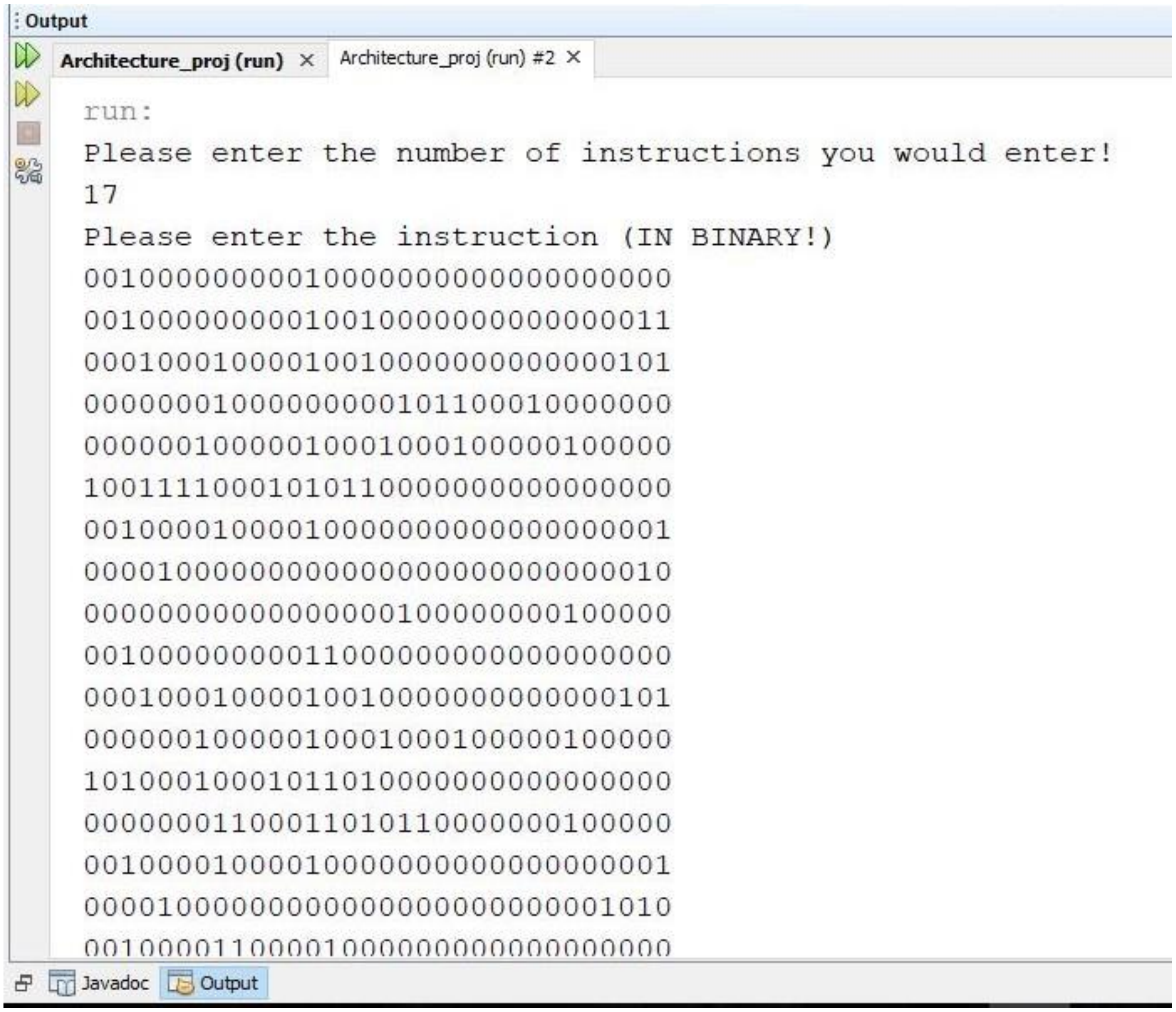
<u>Assembly:</u>

```
addi $t0 , $0 , 0  # i = 0

addi $t1, $0 , 3 # count_Max

loop1 : beq $t0 , $t1 , finish1

sll $t3, $t0, 2  # t3 = 4*i

add $s1, $s0, $t0 # base + offset

sb $t3, 0($s1)

addi $t0, $t0 ,1

j loop1
```

```
finish1:
add $t0 , $0 , $0 # i = 0
addi $t4 , $0, 0 # sum = 0
loop2 :beq $t0, $t1 , finish2
add $s1, $s0, $t0
lb $t5,0($s1)
add $t4, $t4, $t5
addi $t0 , $t0 ,1
j loop2
finish2:
addi $a0 , $t4 , 0
```



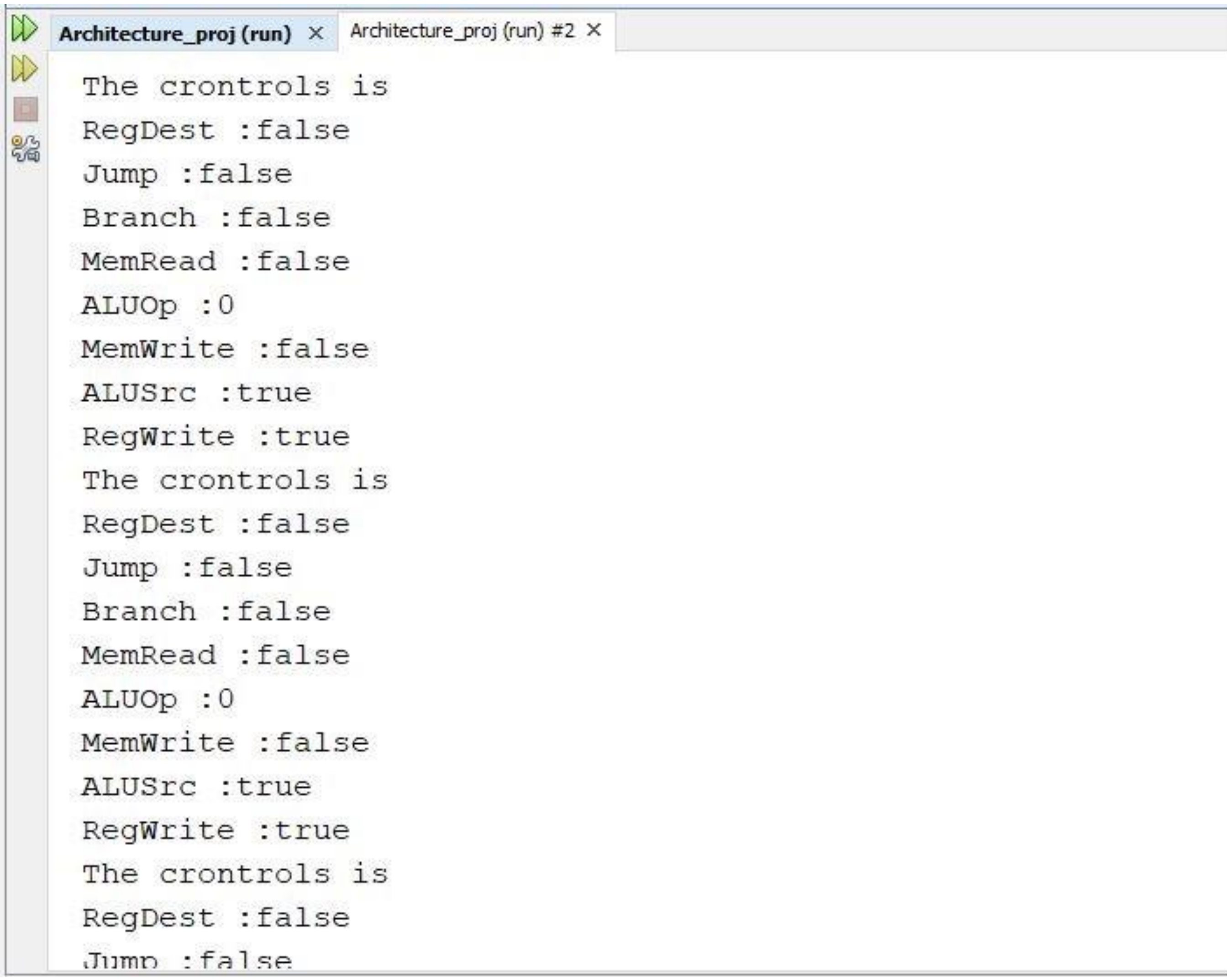
```
run:
Please enter the number of instructions you would enter!
17
```



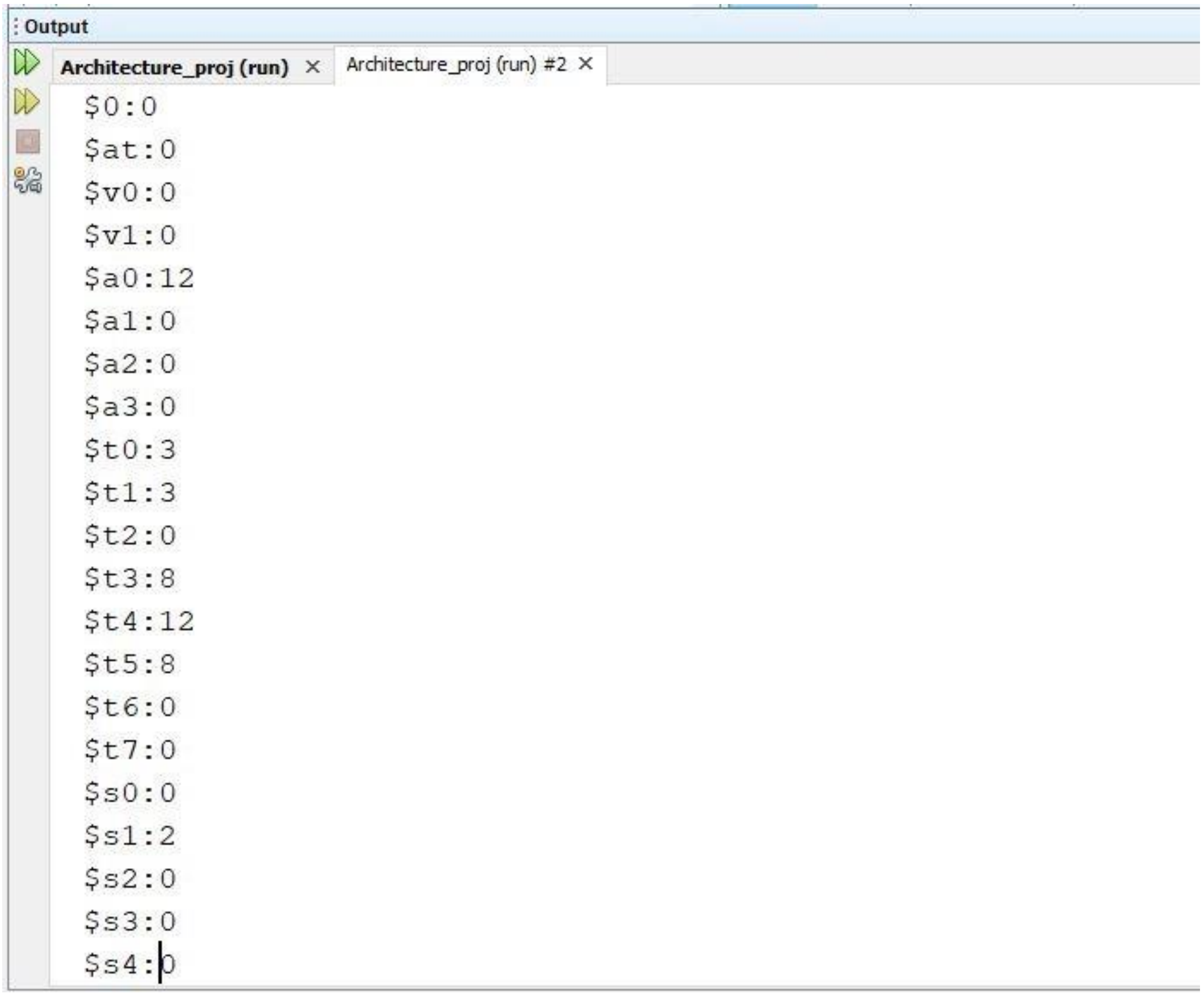
Output

Architecture_proj (run) × Architecture_proj (run) #2 ×

```
run:
Please enter the number of instructions you would enter!
17
Please enter the instruction (IN BINARY!)
00100000000010000000000000000000
00100000000010010000000000000011
00010001000010010000000000000101
00000001000000000101100010000000
00000010000010001000100000100000
10011110001010110000000000000000
00100001000010000000000000000001
00001000000000000000000000000010
00000000000000000100000000100000
00100000000011000000000000000000
00010001000010010000000000000101
00000010000010001000100000100000
10100010001011010000000000000000
00000001100011010110000000100000
00100001000010000000000000000001
00001000000000000000000000001010
00100001100001000000000000000000
```

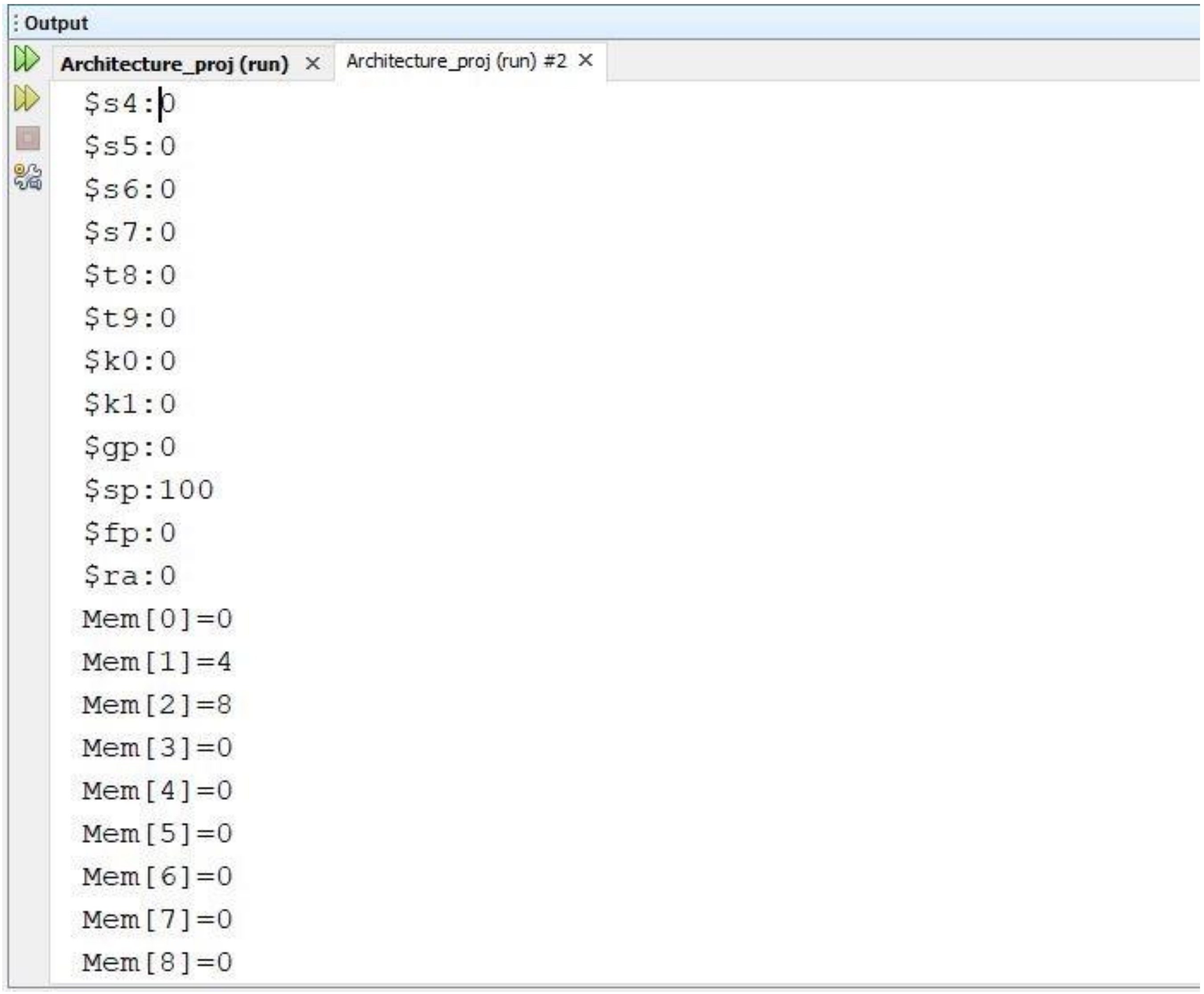


```
The crontrols is
RegDest :false
Jump :false
Branch :false
MemRead :false
ALUOp :0
MemWrite :false
ALUSrc :true
RegWrite :true
The crontrols is
RegDest :false
Jump :false
Branch :false
MemRead :false
ALUOp :0
MemWrite :false
ALUSrc :true
RegWrite :true
The crontrols is
RegDest :false
Jump :false
```

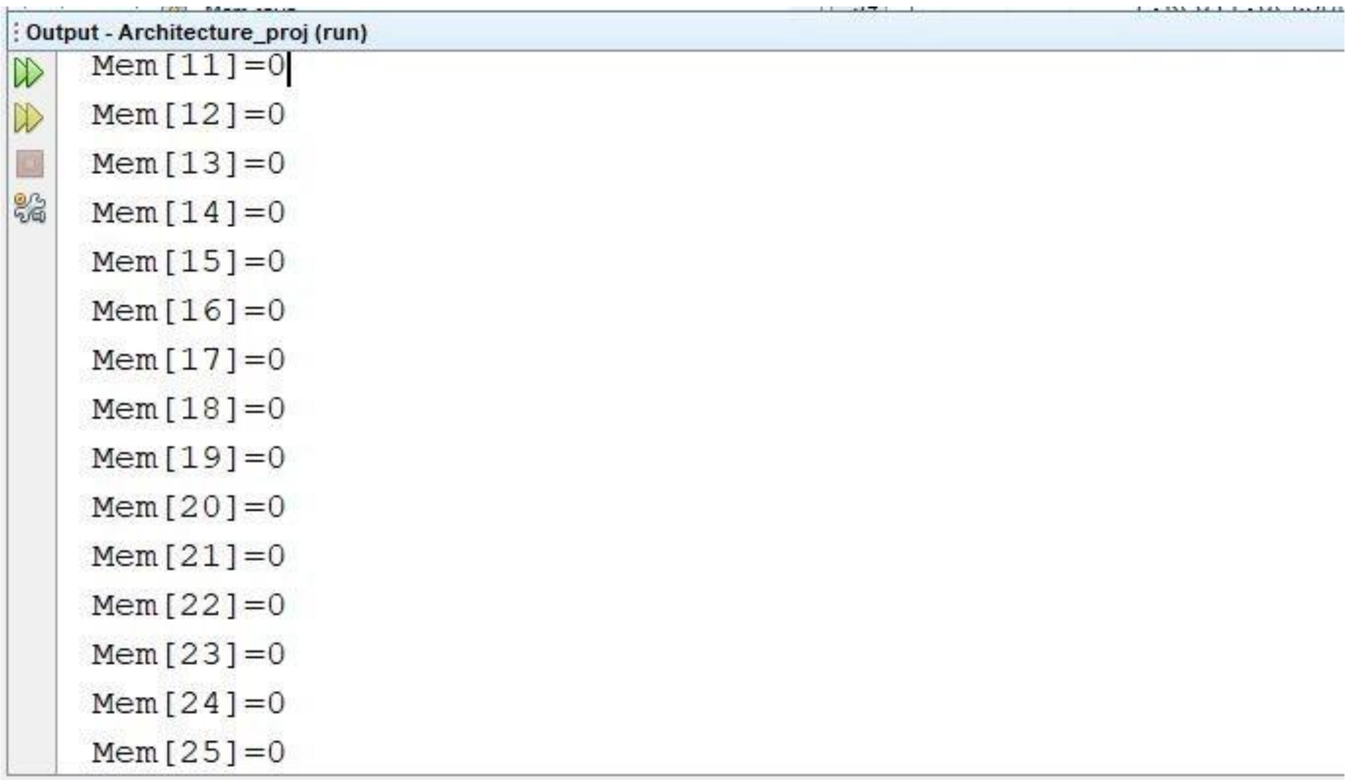

Output

Architecture_proj (run) × Architecture_proj (run) #2 ×

```
$0:0
$at:0
$v0:0
$v1:0
$a0:12
$a1:0
$a2:0
$a3:0
$t0:3
$t1:3
$t2:0
$t3:8
$t4:12
$t5:8
$t6:0
$t7:0
$s0:0
$s1:2
$s2:0
$s3:0
$s4:0
```

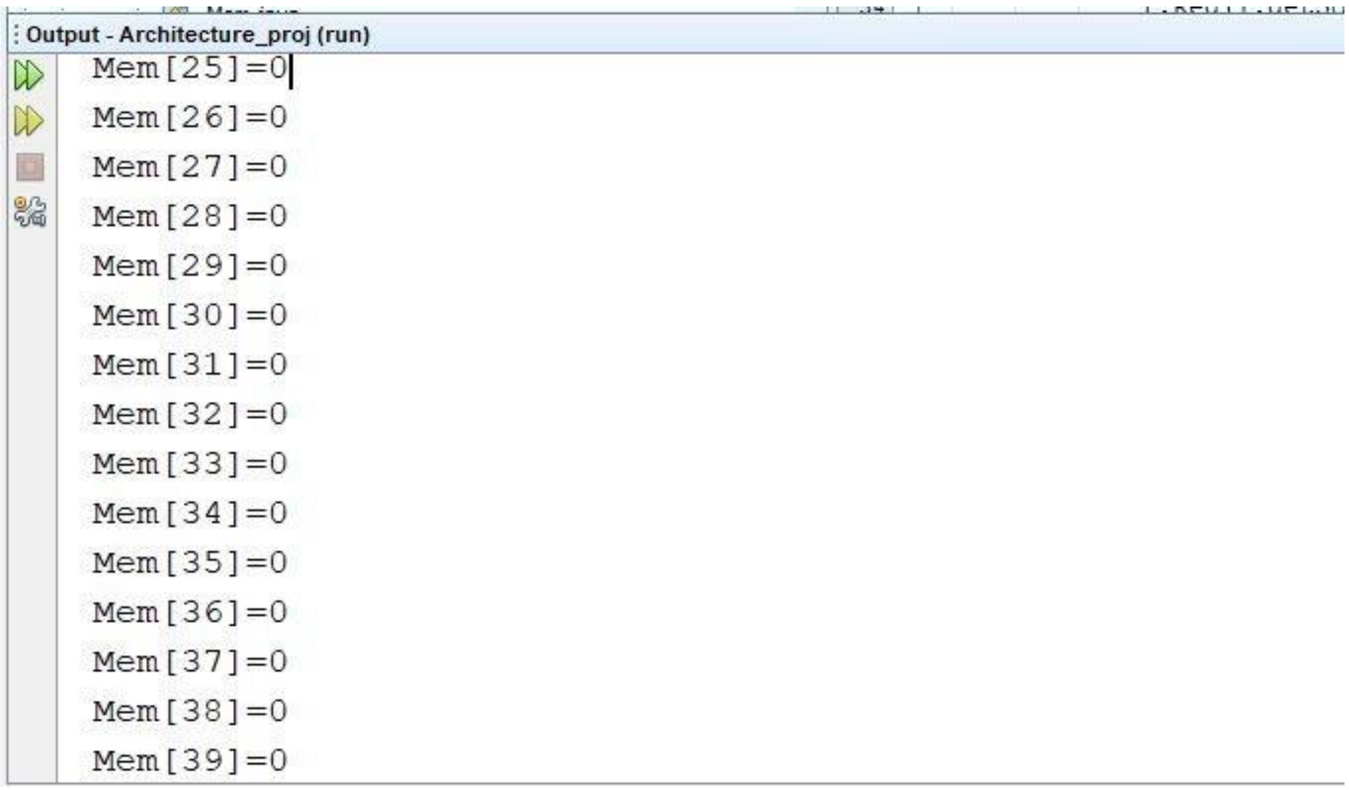


```
$s4:0
$s5:0
$s6:0
$s7:0
$t8:0
$t9:0
$k0:0
$k1:0
$gp:0
$sp:100
$fp:0
$ra:0
Mem[0]=0
Mem[1]=4
Mem[2]=8
Mem[3]=0
Mem[4]=0
Mem[5]=0
Mem[6]=0
Mem[7]=0
Mem[8]=0
```



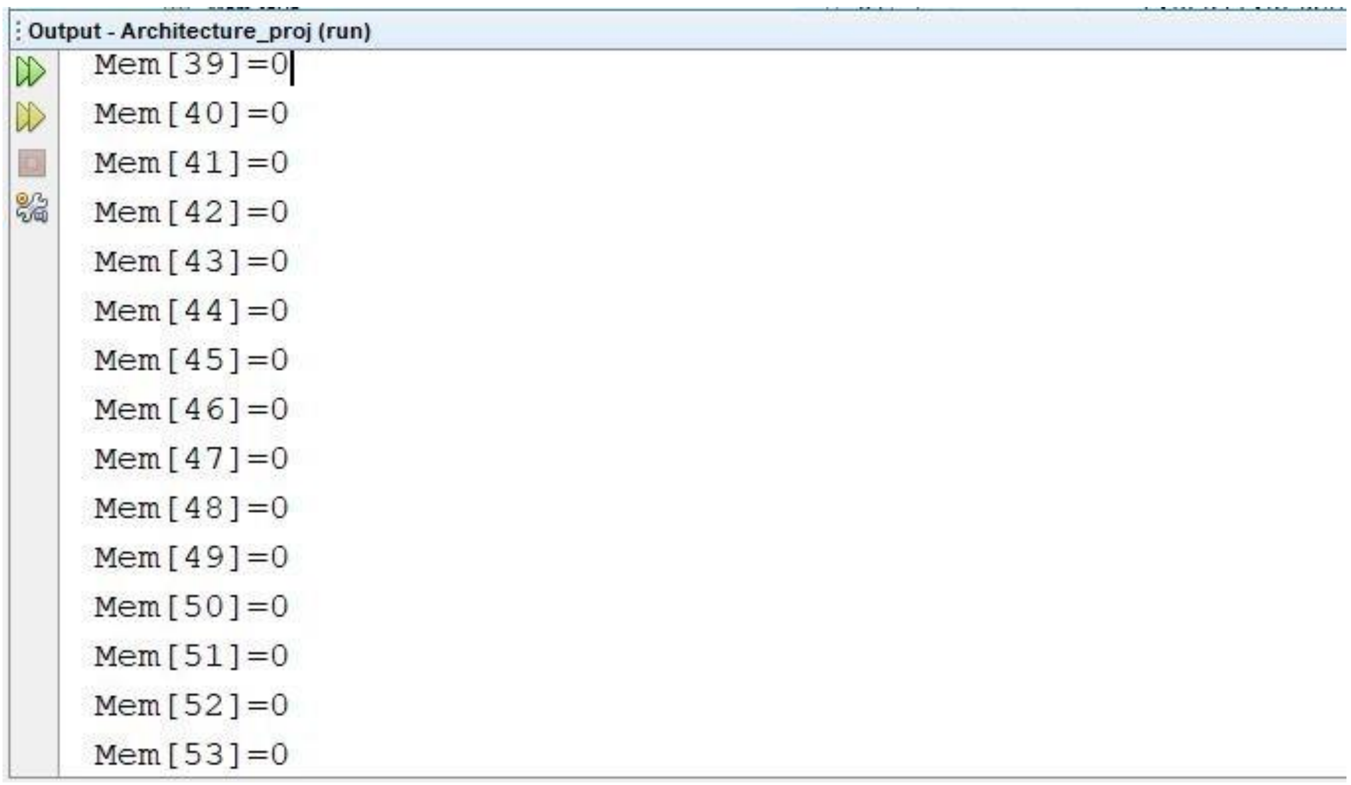
Output - Architecture_proj (run)

```
Mem[11]=0
Mem[12]=0
Mem[13]=0
Mem[14]=0
Mem[15]=0
Mem[16]=0
Mem[17]=0
Mem[18]=0
Mem[19]=0
Mem[20]=0
Mem[21]=0
Mem[22]=0
Mem[23]=0
Mem[24]=0
Mem[25]=0
```

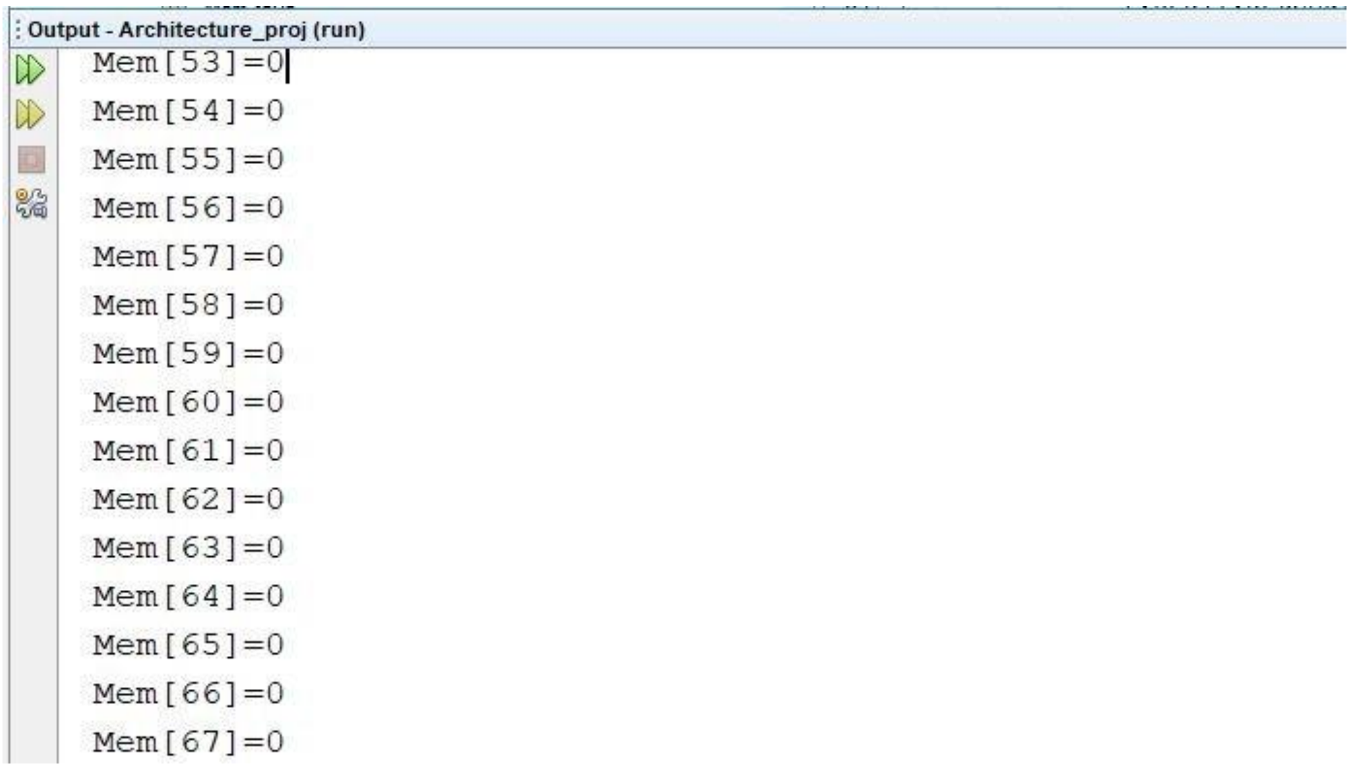


Output - Architecture_proj (run)

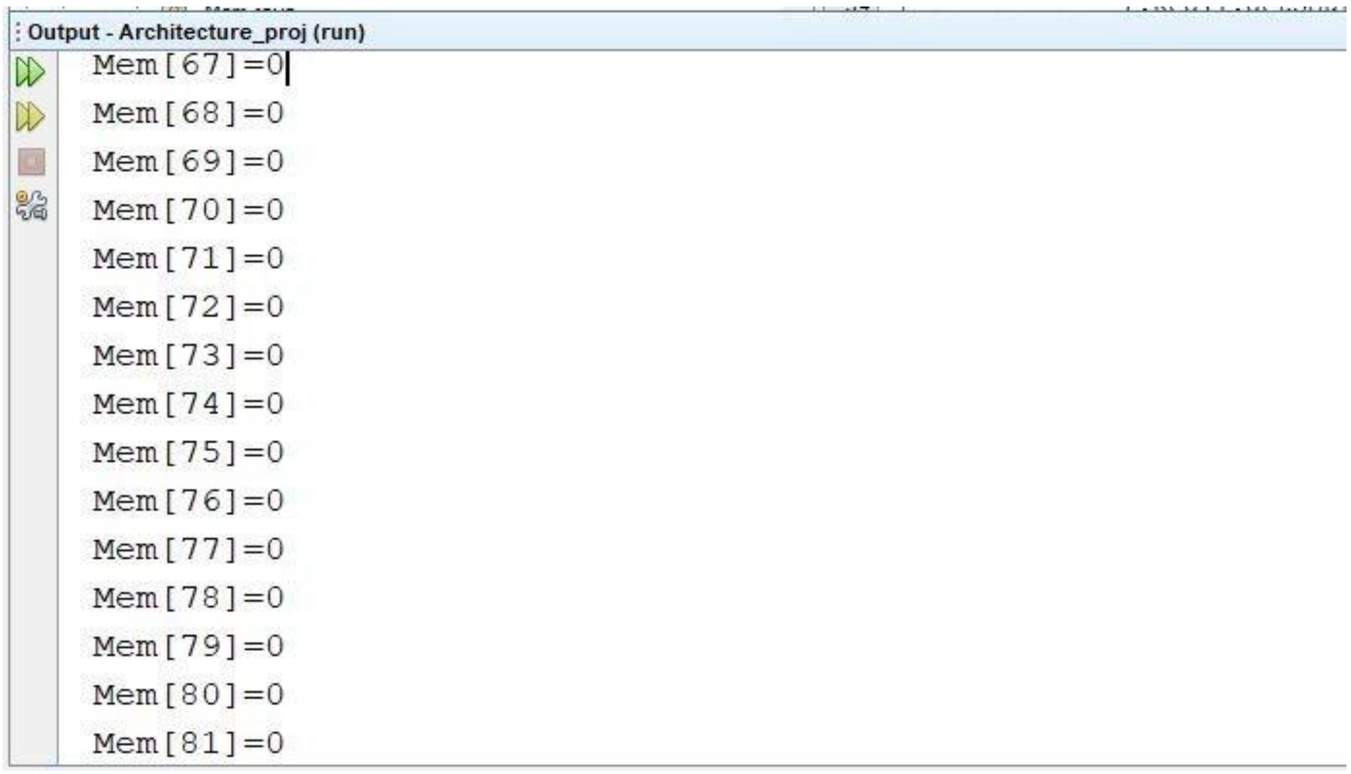
```
Mem[25]=0
Mem[26]=0
Mem[27]=0
Mem[28]=0
Mem[29]=0
Mem[30]=0
Mem[31]=0
Mem[32]=0
Mem[33]=0
Mem[34]=0
Mem[35]=0
Mem[36]=0
Mem[37]=0
Mem[38]=0
Mem[39]=0
```


Output - Architecture_proj (run)

```
Mem[39]=0
Mem[40]=0
Mem[41]=0
Mem[42]=0
Mem[43]=0
Mem[44]=0
Mem[45]=0
Mem[46]=0
Mem[47]=0
Mem[48]=0
Mem[49]=0
Mem[50]=0
Mem[51]=0
Mem[52]=0
Mem[53]=0
```

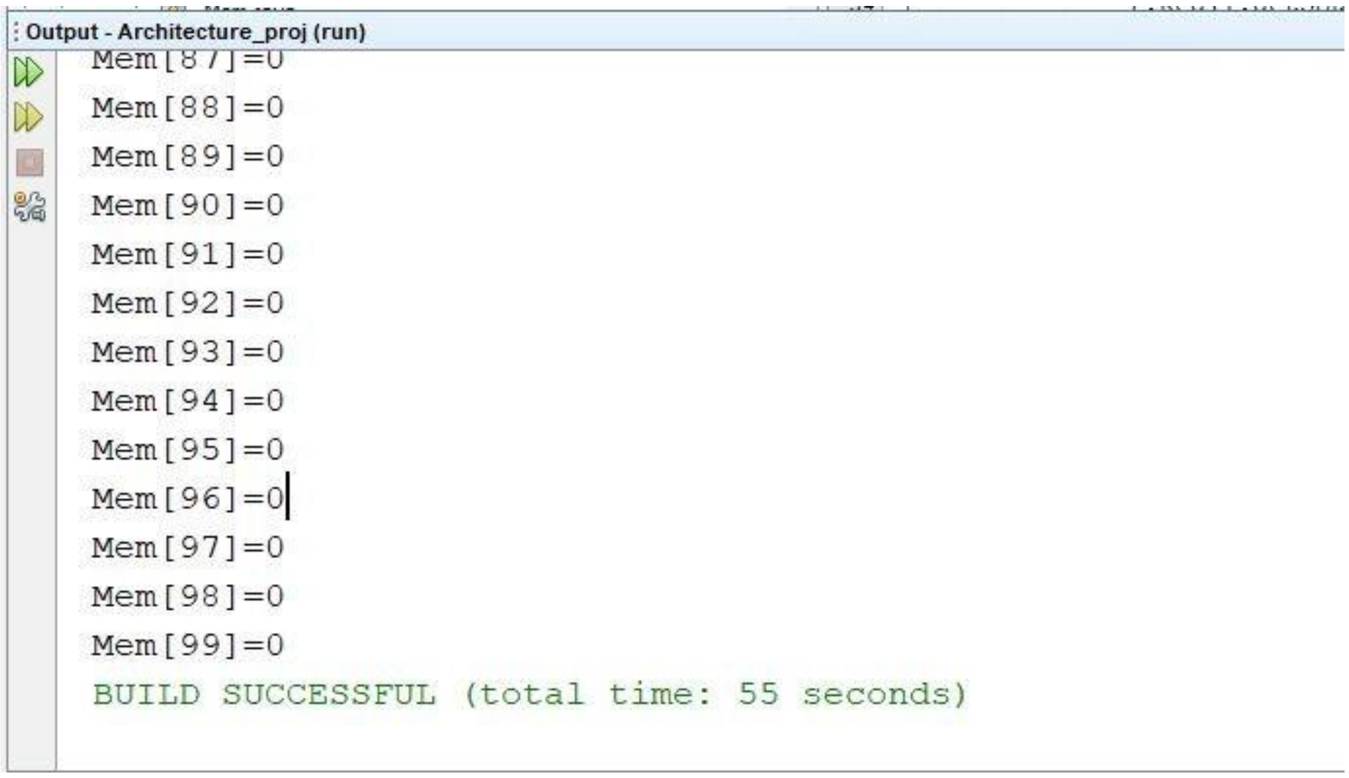

Output - Architecture_proj (run)

```
Mem[53]=0
Mem[54]=0
Mem[55]=0
Mem[56]=0
Mem[57]=0
Mem[58]=0
Mem[59]=0
Mem[60]=0
Mem[61]=0
Mem[62]=0
Mem[63]=0
Mem[64]=0
Mem[65]=0
Mem[66]=0
Mem[67]=0
```



```
Mem[67]=0
Mem[68]=0
Mem[69]=0
Mem[70]=0
Mem[71]=0
Mem[72]=0
Mem[73]=0
Mem[74]=0
Mem[75]=0
Mem[76]=0
Mem[77]=0
Mem[78]=0
Mem[79]=0
Mem[80]=0
Mem[81]=0
```

```
Mem[82]=0
Mem[83]=0
Mem[84]=0
Mem[85]=0
Mem[86]=0
Mem[87]=0
Mem[88]=0
Mem[89]=0
Mem[90]=0
Mem[91]=0
Mem[92]=0
Mem[93]=0
Mem[94]=0
Mem[95]=0
Mem[96]=0
```



```
Mem[87]=0
Mem[88]=0
Mem[89]=0
Mem[90]=0
Mem[91]=0
Mem[92]=0
Mem[93]=0
Mem[94]=0
Mem[95]=0
Mem[96]=0
Mem[97]=0
Mem[98]=0
Mem[99]=0
BUILD SUCCESSFUL (total time: 55 seconds)
```

6. Summary:

The program has been done by all the group. We haven’t distributed a specified task to each one.