RISC V Simulator

Michael Reda - 900203291
Abanoub Emad - 900201630
Youssef Ashraf

Department of Computer Science and Engineering

The American University in Cairo

Spring 2023

Abstract : Assembly language is one of the fundamentals of computer science as it is the bridge between the high level languages and the hardware inside the computer. In this project, we built a RISC V assembly language simulator that is capable of keeping track of the executions of all the instructions and the contents of the registers and the memory. This report presents a RISC-V simulator that was developed as part of a project. The simulator was implemented in Python and provides a user-friendly command-line interface for interacting with the simulated processor. The report details the design and implementation of the simulator, as well as its features and limitations. The simulator was tested using various RISC-V programs and benchmarks, and the results were compared with the expected behavior. Overall, the simulator demonstrated a high degree of accuracy and performance, making it a useful tool for RISC-V development and experimentation.

Keywords: RISC V, Python, Instruction formats, Registers, Memory, Program counter, Stack, Binary, Decimal, Hexadecimal.

I. Introduction.

The RISC-V Instruction Set Architecture (ISA) has gained significant attention in recent years due to its open-source nature and its potential to become a standard for computer architecture. The RISC-V ISA is designed to be modular, scalable, and extensible, allowing designers to customize processors for specific use cases. This flexibility has made the RISC-V ISA a popular choice for a wide range of applications, including embedded systems, mobile devices, and high-performance computing. Simulating the execution of RISC-V instructions is an essential tool for developing and testing RISC-V programs. Simulation allows designers to evaluate the performance and correctness of their programs without the need for physical hardware. For this purpose, we created a RISC V simulator which is capable of keeping track of the instructions along with the contents of the 32 registers and the memory contents including its stack. Moreover, we tested the simulator against a variety of RISC V programs to ensure the correctness of this simulator.

II. Problem Definition.

Simulating RISC V ISA requires a deep understanding of all the instructions, their formats and how they deal with the RISC V memory and hardware. This simulator must simulate how each instruction in RISC V works correctly and must spot any incorrect inputs or instruction formats. Also, it must perform well in the different test cases. It should also manage the contents of the memory and the registers as what RISC V ISA does.

III. Methodology.

Implementation Choices.

First the programming language.

We decided to use python as the programming language in building this simulator for various reasons. First of all, python provides a lot of string methods which will help us in dealing with input files and to extract the instructions and their operands in an easy and optimal way. Second, reading from files is easy in python which saved our time and made us focus on more important parts in the project. Third, python provides a variety of data structures such as the dictionaries which help us in representing the memory and the registers easily. Finally, dealing with exceptions in python is very simple unlike other programming languages such as C++. And since many exceptions might occur, we decided to choose python to enable us to handle those exceptions in a straightforward way.

Second, Representation of memory and registers.
The main components of RISC V ISA which are responsible for storage are the registers and the memory. We decided to use hash maps (Dictionaries in python) data structures to represent the registers and the memory as maps provide accurate and fast retrieval of the data. Also, when dealing with the memory which can grow up to 4 GBs, regular arrays will be a bottleneck as its huge size will affect the performance of the simulator and will be a huge load on the computer the simulator runs.

Third, using a single python file and not using OOP.
It is known that OOP is very efficient in dealing with programs and data and provides a modular and easy way of programming. However, we did feel that our program needs to be implemented using OOP. However, it is just a series of function executions that does the reading of the instructions and their executions.

Implementation.

General flow of the program:
When the user runs the program, it will read the contents of the code text file which is in the same folder of the python program. It will then read the instructions and extract all the important information such as the actual instruction and its operands and will store them. The instructions along with their operands will be stored in an array. The program will also keep track of the address of each instruction. It will then populate a hashmap in which the addresses are the keys and the values are the instructions. Also, the program will search for the labels and their corresponding addresses. After reading the assembly code, the program will then open the data file and initialize the memory hashmap according to the contents of the data file. Then the python code will execute the instructions. The first executed instruction will be the first instruction at the main block. The value of the program counter will be updated according to the instruction executed. The program will then continue the execution of the instructions until the instructions ends. It should be noted that after the execution of each instruction, the simulator will output the contents of the registers and the memory in decimal, binary and hexadecimal.

Implemented Bonus features:

We implemented two bonus features in our code which are printing the contents of the registers and the memory in hexadecimal and binary in addition to the default which is the decimal value. Also, we tested our program on 6 programs instead of 3 programs.

Main functions in the python code.

1. Read code file:
   This function reads the file containing the assembly code and from this file it extracts the instructions and labels.
2. Instructions tokenization:
   This function extracts the fields of the instructions. It identifies the instruction itself (i.e Add, Jal, etc) and then according to the instruction, it determines what are the operands of this instruction and extracts them.
3. Read and initialize memory:
   This function reads the data file and places its contents in the memory. The data file contains the contents of the memory words in decimal. In other words, the memory addresses in this file should be divisible by 4 (words) and their contents should be in decimal. On the other hand, the hashmap of the memory is divided into bytes. This function reads the initialized memory words and their contents and then divides the words into bytes and assigns them their corresponding contents.
4. Initialize Registers:
   This function initializes all registers and sets them to zeros and the stack pointer to 40000000 before the execution of any instructions.
5. Execute instructions:
   This function executes the instructions starting from the label main. According to the instruction, this function decides how to execute it and deal with the contents of the registers and the memory. This function also decided how to deal with the program counter depending on the instructions. This function will stop the execution when it reaches the end of the code file.

Assumptions and Decision Choices.

1. The user will specify the starting address of the program with the main label. The user should put the instructions he wants to execute first in the main block and then the simulator will go directly to start the execution.
2. The data file should include the contents in words not half words or bytes and the value should be in decimal.
3. The Ecall, Fence and the Ebreak instructions should stop the execution of the program immediately.

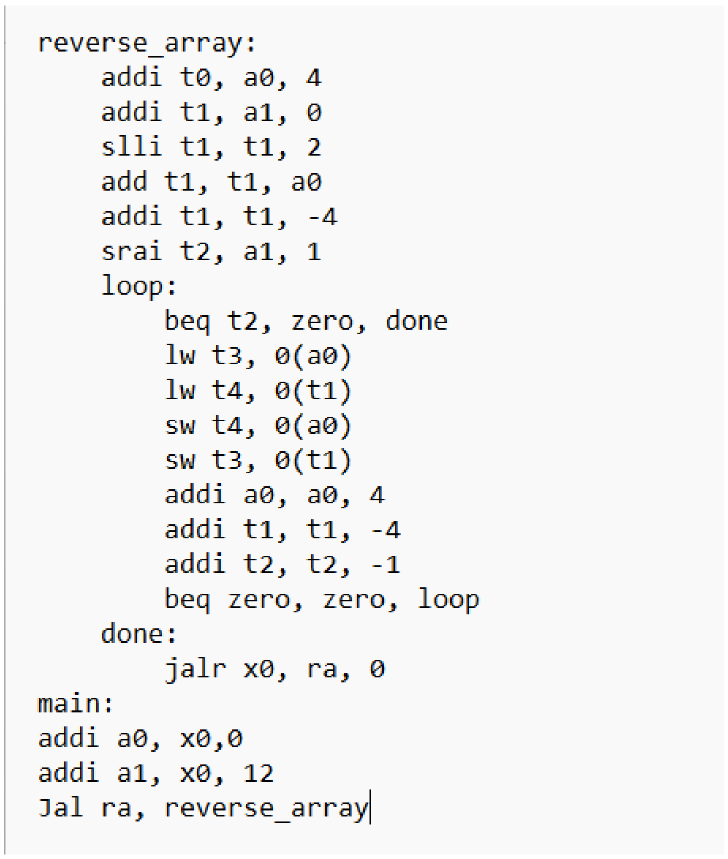
4. The stack will be placed in the same map of the memory. However, the stack pointer will be given a very large address to give enough space for the other contents of the memory and to avoid overriding. We decided to implement the stack in this way as it is the closest to how actual memory works. Also, it would make the coding of the other instructions much easier as the stack pointer register will be dealt with the same way of the other registers.

IV- User Interaction with the program

There are two main files that the user needs to interact with in order to start using our simulator. The first is the code file where the user puts the riscV instructions that he wants to see its simulation on our simulator. Second, he needs to fill the data file where he puts any information or data that is needed while executing the program.

For example the reverse array function:

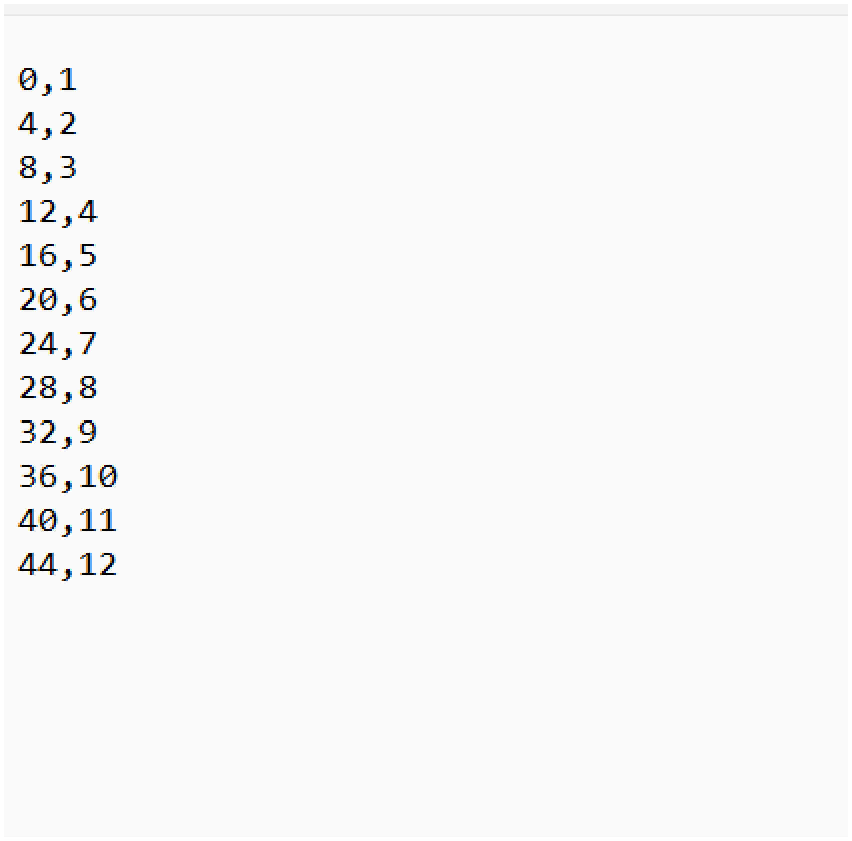
1st: This is the code file for the function:



```
reverse_array:
    addi t0, a0, 4
    addi t1, a1, 0
    slli t1, t1, 2
    add t1, t1, a0
    addi t1, t1, -4
    srai t2, a1, 1
    loop:
        beq t2, zero, done
        lw t3, 0(a0)
        lw t4, 0(t1)
        sw t4, 0(a0)
        sw t3, 0(t1)
        addi a0, a0, 4
        addi t1, t1, -4
        addi t2, t2, -1
        beq zero, zero, loop
    done:
        jalr x0, ra, 0
main:
addi a0, x0,0
addi a1, x0, 12
Jal ra, reverse_array
```

Here, this is the riscV instructions to implement the reverse of an array.

2nd: This is the data file for the function:

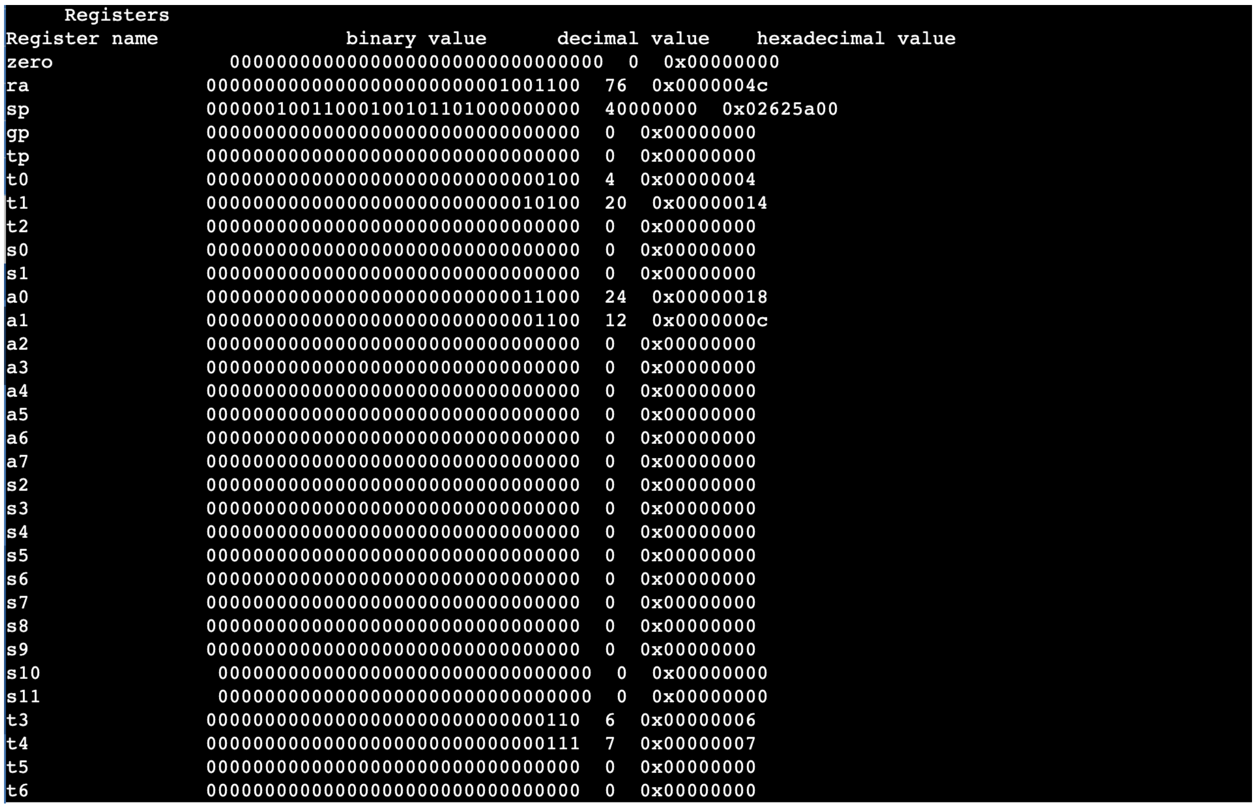


```
0,1
4,2
8,3
12,4
16,5
20,6
24,7
28,8
32,9
36,10
40,11
44,12
```

In this example, the data file consists of memory addresses, and the content of the array is distributed among the whole addresses.

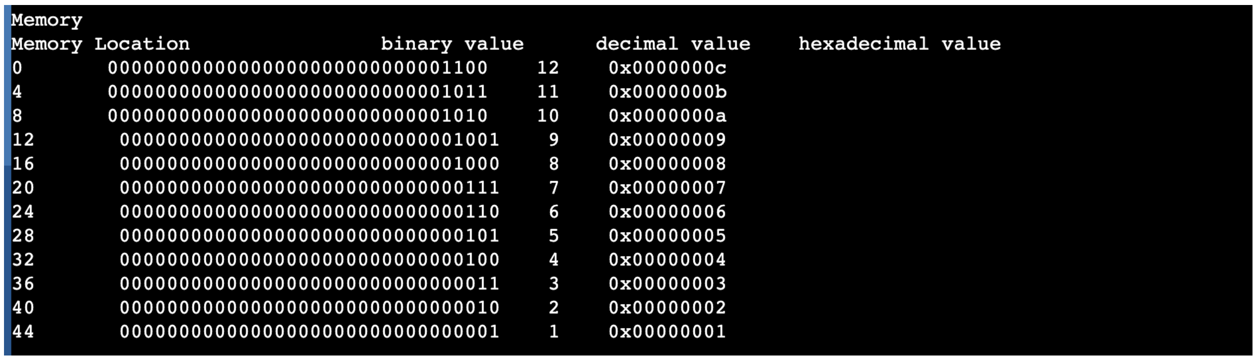
After filling these two files, the user will then run the simulator and he will see the contents of the registers and the memory after each instruction execution.

Registers



```
     Registers
Register name               binary value      decimal value    hexadecimal value
zero               00000000000000000000000000000000  0  0x00000000
ra               00000000000000000000000001001100  76  0x0000004c
sp               00000010011000100101101000000000  40000000  0x02625a00
gp               00000000000000000000000000000000  0  0x00000000
tp               00000000000000000000000000000000  0  0x00000000
t0               00000000000000000000000000000100  4  0x00000004
t1               00000000000000000000000000010100  20  0x00000014
t2               00000000000000000000000000000000  0  0x00000000
s0               00000000000000000000000000000000  0  0x00000000
s1               00000000000000000000000000000000  0  0x00000000
a0               00000000000000000000000000011000  24  0x00000018
a1               00000000000000000000000000001100  12  0x0000000c
a2               00000000000000000000000000000000  0  0x00000000
a3               00000000000000000000000000000000  0  0x00000000
a4               00000000000000000000000000000000  0  0x00000000
a5               00000000000000000000000000000000  0  0x00000000
a6               00000000000000000000000000000000  0  0x00000000
a7               00000000000000000000000000000000  0  0x00000000
s2               00000000000000000000000000000000  0  0x00000000
s3               00000000000000000000000000000000  0  0x00000000
s4               00000000000000000000000000000000  0  0x00000000
s5               00000000000000000000000000000000  0  0x00000000
s6               00000000000000000000000000000000  0  0x00000000
s7               00000000000000000000000000000000  0  0x00000000
s8               00000000000000000000000000000000  0  0x00000000
s9               00000000000000000000000000000000  0  0x00000000
s10               00000000000000000000000000000000  0  0x00000000
s11               00000000000000000000000000000000  0  0x00000000
t3               00000000000000000000000000000110  6  0x00000006
t4               00000000000000000000000000000111  7  0x00000007
t5               00000000000000000000000000000000  0  0x00000000
t6               00000000000000000000000000000000  0  0x00000000
```

Memory



```
Memory
Memory Location                 binary value      decimal value    hexadecimal value
0       00000000000000000000000000001100    12     0x0000000c
4       00000000000000000000000000001011    11     0x0000000b
8       00000000000000000000000000001010    10     0x0000000a
12       00000000000000000000000000001001    9     0x00000009
16       00000000000000000000000000001000    8     0x00000008
20       00000000000000000000000000000111    7     0x00000007
24       00000000000000000000000000000110    6     0x00000006
28       00000000000000000000000000000101    5     0x00000005
32       00000000000000000000000000000100    4     0x00000004
36       00000000000000000000000000000011    3     0x00000003
40       00000000000000000000000000000010    2     0x00000002
44       00000000000000000000000000000001    1     0x00000001
```

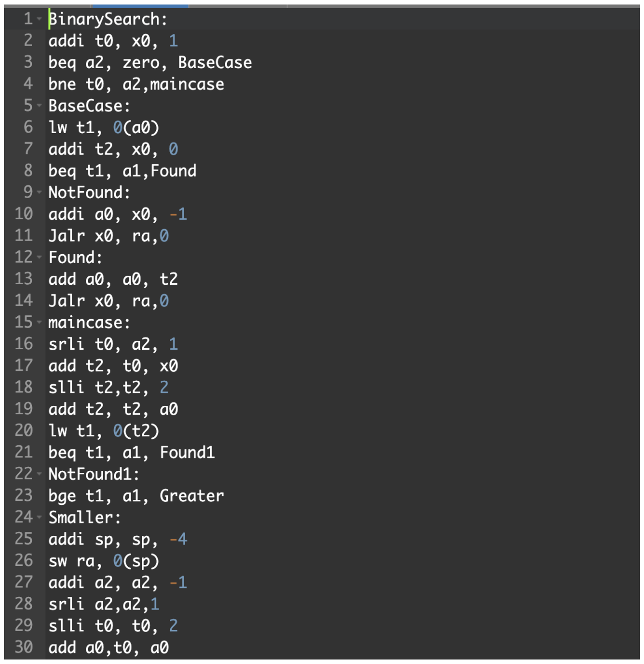
V- Testing

In the testing partition we tested up to six programs in order to cover all functions. Firstly, in Binary search we used an array to search inside it, and compare the middle element with the target. Also, we added a reverse array program to reverse the digits which also uses an array. Moreover, a fibonacci function is recursive to make sure that the simulator can run recursive functions normally. Also we implemented insertion sort which compares each value and resorts to them in the right way. As we made a block of code to test the bitwise functions and it was working successfully. Finally, we did a program to test which is string copy which copies the string from the source location to a destination in the memory. In other words, we build six programs and they are all successfully run without any problem as we expected and also they runned with the expected values that we were expecting as a result and I will give examples of this:

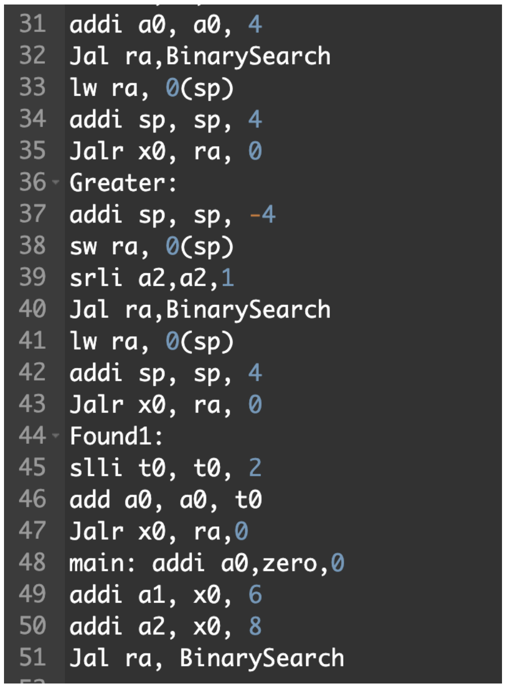
1- Binary Search

We wrote a RISC V binary search code and initialized a sorted array in the memory at address = 0 and it contained 8 elements from 1 to 8. We then run the code to search for number 6 and it.

Code file

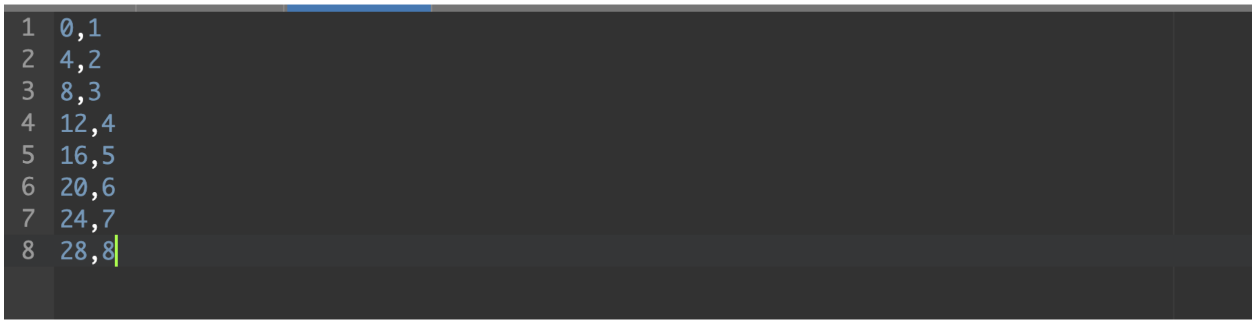


```
BinarySearch:
addi t0, x0, 1
beq a2, zero, BaseCase
bne t0, a2,maincase
BaseCase:
lw t1, 0(a0)
addi t2, x0, 0
beq t1, a1,Found
NotFound:
addi a0, x0, -1
Jalr x0, ra,0
Found:
add a0, a0, t2
Jalr x0, ra,0
maincase:
srli t0, a2, 1
add t2, t0, x0
slli t2,t2, 2
add t2, t2, a0
lw t1, 0(t2)
beq t1, a1, Found1
NotFound1:
bge t1, a1, Greater
Smaller:
addi sp, sp, -4
sw ra, 0(sp)
addi a2, a2, -1
srli a2,a2,1
slli t0, t0, 2
add a0,t0, a0
```



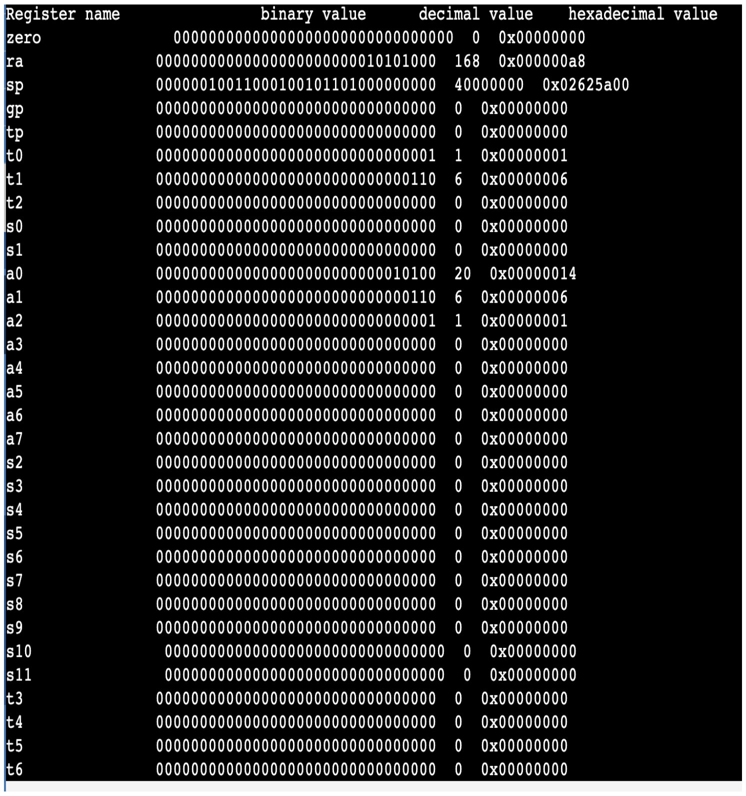
```
addi a0, a0, 4
Jal ra,BinarySearch
lw ra, 0(sp)
addi sp, sp, 4
Jalr x0, ra, 0
Greater:
addi sp, sp, -4
sw ra, 0(sp)
srli a2,a2,1
Jal ra,BinarySearch
lw ra, 0(sp)
addi sp, sp, 4
Jalr x0, ra, 0
Found1:
slli t0, t0, 2
add a0, a0, t0
Jalr x0, ra,0
main: addi a0,zero,0
addi a1, x0, 6
addi a2, x0, 8
Jal ra, BinarySearch
```

Data file



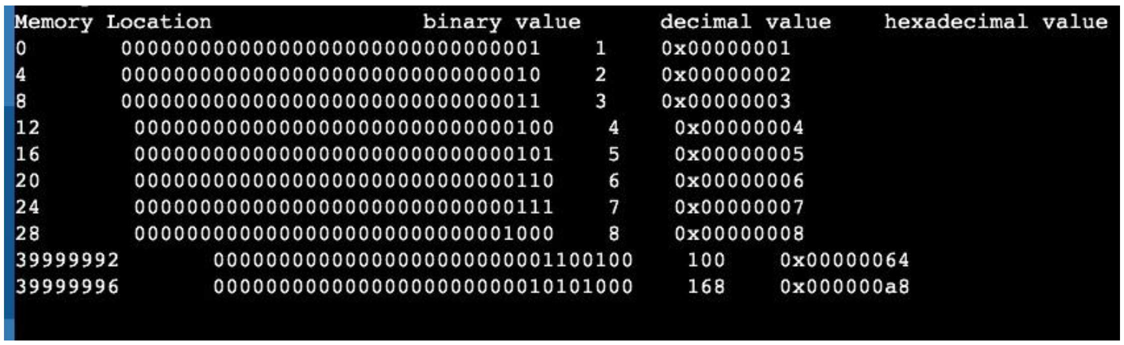
```
0,1
4,2
8,3
12,4
16,5
20,6
24,7
28,8
```

Registers output (the result is in register a0)



```
Register name                binary value      decimal value    hexadecimal value
zero               00000000000000000000000000000000  0  0x00000000
ra               00000000000000000000000010101000  168  0x000000a8
sp               00000010011000100101101000000000  40000000  0x02625a00
gp               00000000000000000000000000000000  0  0x00000000
tp               00000000000000000000000000000000  0  0x00000000
t0               00000000000000000000000000000001  1  0x00000001
t1               00000000000000000000000000000110  6  0x00000006
t2               00000000000000000000000000000000  0  0x00000000
s0               00000000000000000000000000000000  0  0x00000000
s1               00000000000000000000000000000000  0  0x00000000
a0               00000000000000000000000000010100  20  0x00000014
a1               00000000000000000000000000000110  6  0x00000006
a2               00000000000000000000000000000001  1  0x00000001
a3               00000000000000000000000000000000  0  0x00000000
a4               00000000000000000000000000000000  0  0x00000000
a5               00000000000000000000000000000000  0  0x00000000
a6               00000000000000000000000000000000  0  0x00000000
a7               00000000000000000000000000000000  0  0x00000000
s2               00000000000000000000000000000000  0  0x00000000
s3               00000000000000000000000000000000  0  0x00000000
s4               00000000000000000000000000000000  0  0x00000000
s5               00000000000000000000000000000000  0  0x00000000
s6               00000000000000000000000000000000  0  0x00000000
s7               00000000000000000000000000000000  0  0x00000000
s8               00000000000000000000000000000000  0  0x00000000
s9               00000000000000000000000000000000  0  0x00000000
s10               00000000000000000000000000000000  0  0x00000000
s11               00000000000000000000000000000000  0  0x00000000
t3               00000000000000000000000000000000  0  0x00000000
t4               00000000000000000000000000000000  0  0x00000000
t5               00000000000000000000000000000000  0  0x00000000
t6               00000000000000000000000000000000  0  0x00000000
```

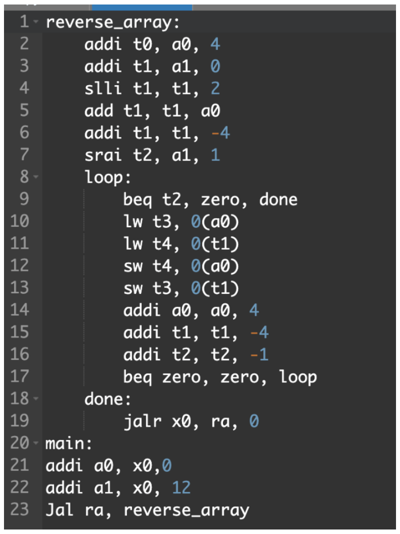
Memory output



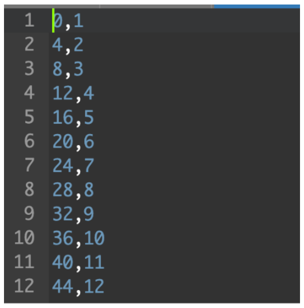
```
Memory Location                 binary value       decimal value    hexadecimal value
0        00000000000000000000000000000001    1    0x00000001
4        00000000000000000000000000000010    2    0x00000002
8        00000000000000000000000000000011    3    0x00000003
12        00000000000000000000000000000100    4    0x00000004
16        00000000000000000000000000000101    5    0x00000005
20        00000000000000000000000000000110    6    0x00000006
24        00000000000000000000000000000111    7    0x00000007
28        00000000000000000000000000001000    8    0x00000008
39999992        00000000000000000000000001100100    100    0x00000064
39999996        00000000000000000000000010101000    168    0x000000a8
```

2. Array Reverse (this code reverse an array initialized at the memory)

Code file Data file

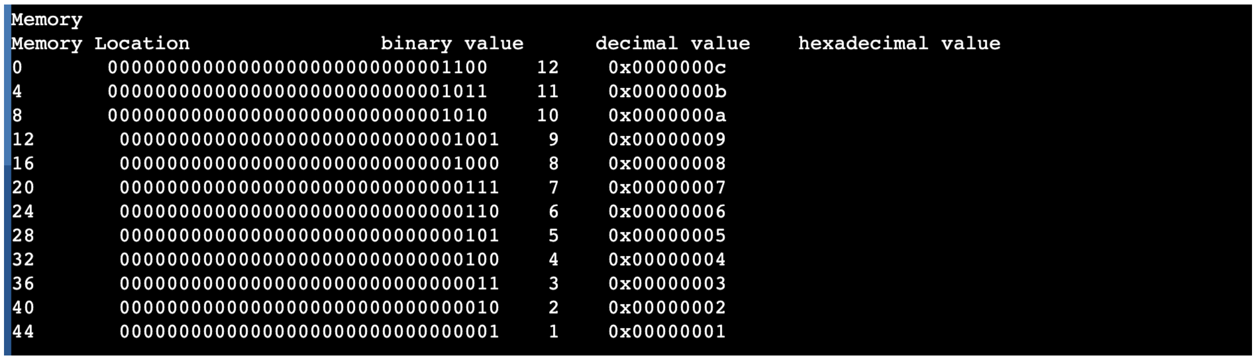


```
reverse_array:
    addi t0, a0, 4
    addi t1, a1, 0
    slli t1, t1, 2
    add t1, t1, a0
    addi t1, t1, -4
    srai t2, a1, 1
    loop:
        beq t2, zero, done
        lw t3, 0(a0)
        lw t4, 0(t1)
        sw t4, 0(a0)
        sw t3, 0(t1)
        addi a0, a0, 4
        addi t1, t1, -4
        addi t2, t2, -1
        beq zero, zero, loop
    done:
        jalr x0, ra, 0
main:
addi a0, x0,0
addi a1, x0, 12
Jal ra, reverse_array
```



```
0,1
4,2
8,3
12,4
16,5
20,6
24,7
28,8
32,9
36,10
40,11
44,12
```

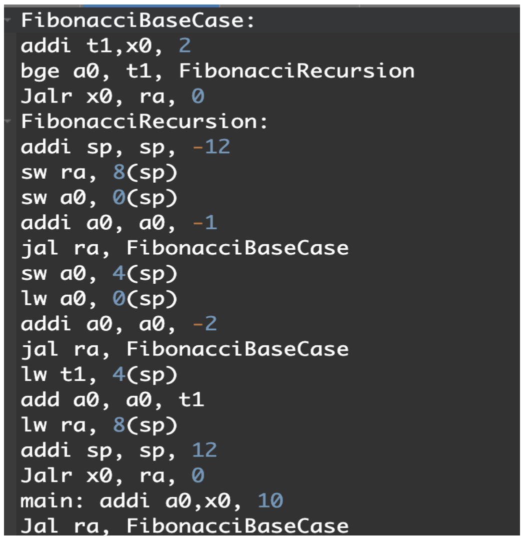
Memory output



```
Memory
Memory Location                binary value       decimal value    hexadecimal value
0       00000000000000000000000000001100    12    0x0000000c
4       00000000000000000000000000001011    11    0x0000000b
8       00000000000000000000000000001010    10    0x0000000a
12       00000000000000000000000000001001     9    0x00000009
16       00000000000000000000000000001000     8    0x00000008
20       00000000000000000000000000000111     7    0x00000007
24       00000000000000000000000000000110     6    0x00000006
28       00000000000000000000000000000101     5    0x00000005
32       00000000000000000000000000000100     4    0x00000004
36       00000000000000000000000000000011     3    0x00000003
40       00000000000000000000000000000010     2    0x00000002
44       00000000000000000000000000000001     1    0x00000001
```

3. Fibonacci Problem (Returns the nth term of the fibonacci series, the number n is initialized in a0)

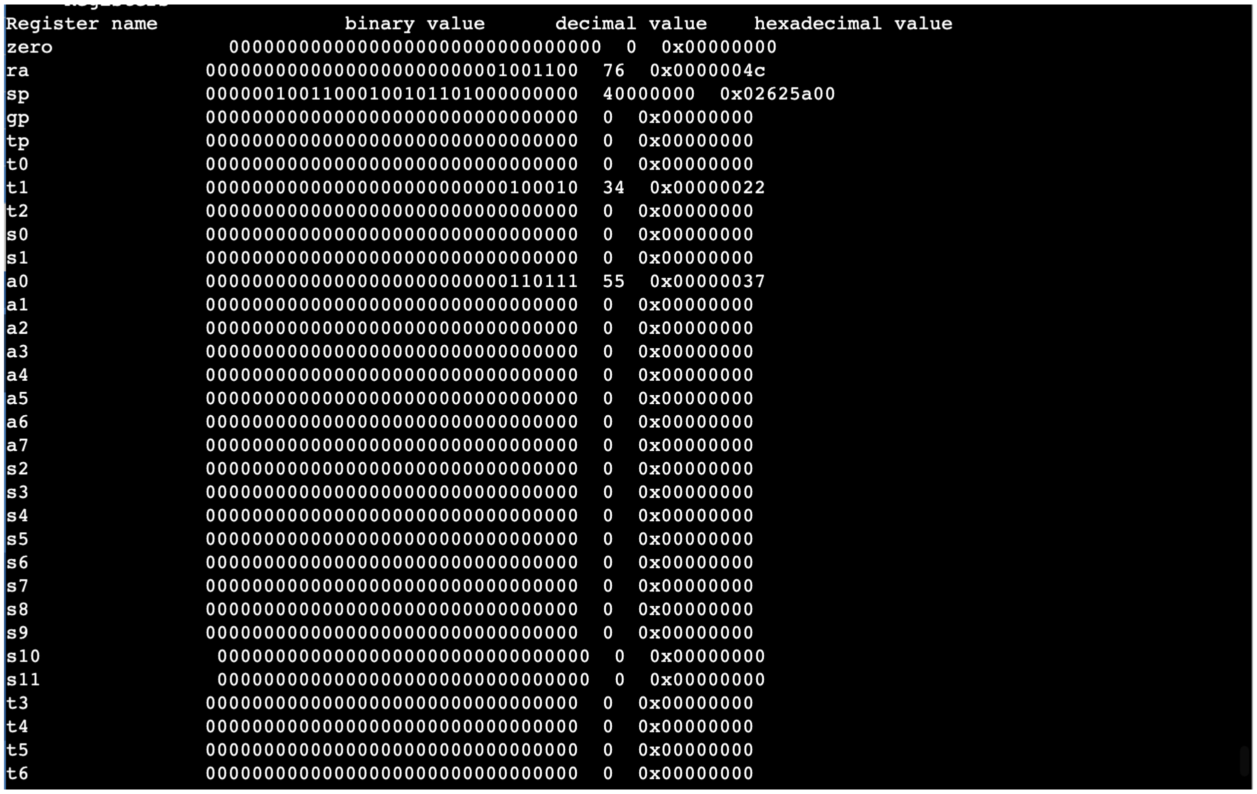
Code file



```
FibonacciBaseCase:
addi t1,x0, 2
bge a0, t1, FibonacciRecursion
Jalr x0, ra, 0
FibonacciRecursion:
addi sp, sp, -12
sw ra, 8(sp)
sw a0, 0(sp)
addi a0, a0, -1
jal ra, FibonacciBaseCase
sw a0, 4(sp)
lw a0, 0(sp)
addi a0, a0, -2
jal ra, FibonacciBaseCase
lw t1, 4(sp)
add a0, a0, t1
lw ra, 8(sp)
addi sp, sp, 12
Jalr x0, ra, 0
main: addi a0,x0, 10
Jal ra, FibonacciBaseCase
```

There are no required operands in the memory to test this code. So, we left the memory file empty.

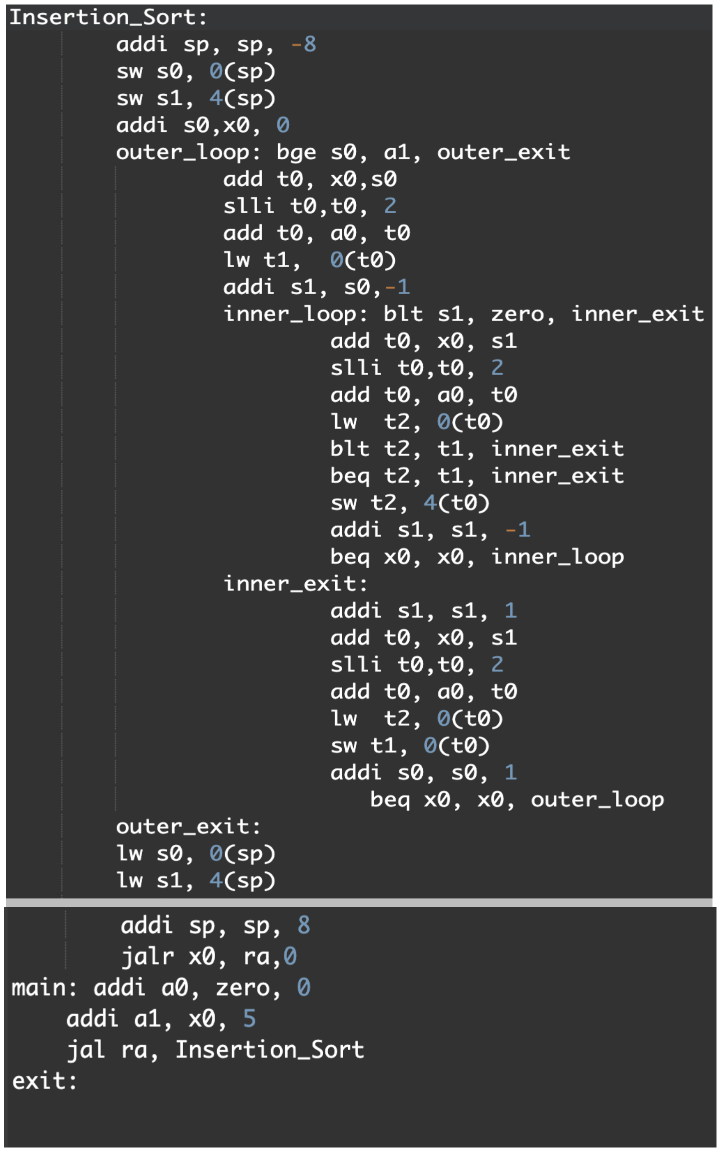
Registers Output.



```
Register name                binary value       decimal value    hexadecimal value
zero              00000000000000000000000000000000  0  0x00000000
ra              00000000000000000000000001001100  76  0x0000004c
sp              00000010011000100101101000000000  40000000  0x02625a00
gp              00000000000000000000000000000000  0  0x00000000
tp              00000000000000000000000000000000  0  0x00000000
t0              00000000000000000000000000000000  0  0x00000000
t1              00000000000000000000000000100010  34  0x00000022
t2              00000000000000000000000000000000  0  0x00000000
s0              00000000000000000000000000000000  0  0x00000000
s1              00000000000000000000000000000000  0  0x00000000
a0              00000000000000000000000000110111  55  0x00000037
a1              00000000000000000000000000000000  0  0x00000000
a2              00000000000000000000000000000000  0  0x00000000
a3              00000000000000000000000000000000  0  0x00000000
a4              00000000000000000000000000000000  0  0x00000000
a5              00000000000000000000000000000000  0  0x00000000
a6              00000000000000000000000000000000  0  0x00000000
a7              00000000000000000000000000000000  0  0x00000000
s2              00000000000000000000000000000000  0  0x00000000
s3              00000000000000000000000000000000  0  0x00000000
s4              00000000000000000000000000000000  0  0x00000000
s5              00000000000000000000000000000000  0  0x00000000
s6              00000000000000000000000000000000  0  0x00000000
s7              00000000000000000000000000000000  0  0x00000000
s8              00000000000000000000000000000000  0  0x00000000
s9              00000000000000000000000000000000  0  0x00000000
s10              00000000000000000000000000000000  0  0x00000000
s11              00000000000000000000000000000000  0  0x00000000
t3              00000000000000000000000000000000  0  0x00000000
t4              00000000000000000000000000000000  0  0x00000000
t5              00000000000000000000000000000000  0  0x00000000
t6              00000000000000000000000000000000  0  0x00000000
```

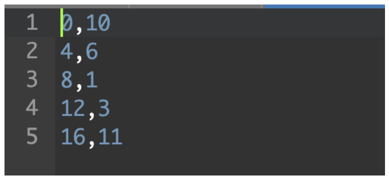
4. Insertion Sort (Use the insertion sort algorithm to sort an array initialized in the memory)

Code File



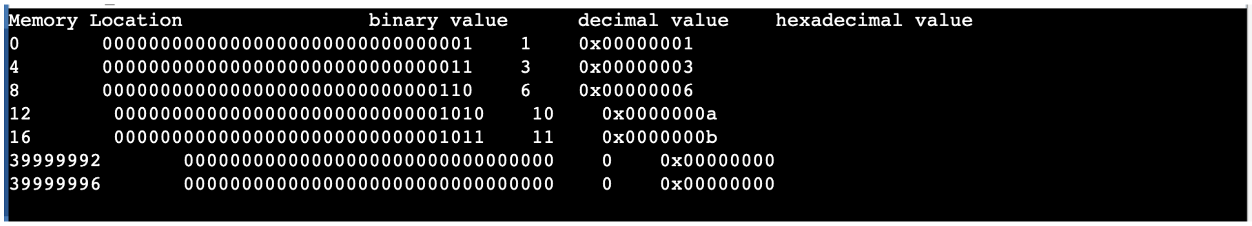
```
Insertion_Sort:
        addi sp, sp, -8
        sw s0, 0(sp)
        sw s1, 4(sp)
        addi s0,x0, 0
        outer_loop: bge s0, a1, outer_exit
                add t0, x0,s0
                slli t0,t0, 2
                add t0, a0, t0
                lw t1,  0(t0)
                addi s1, s0,-1
                inner_loop: blt s1, zero, inner_exit
                        add t0, x0, s1
                        slli t0,t0, 2
                        add t0, a0, t0
                        lw  t2, 0(t0)
                        blt t2, t1, inner_exit
                        beq t2, t1, inner_exit
                        sw t2, 4(t0)
                        addi s1, s1, -1
                        beq x0, x0, inner_loop
                inner_exit:
                        addi s1, s1, 1
                        add t0, x0, s1
                        slli t0,t0, 2
                        add t0, a0, t0
                        lw  t2, 0(t0)
                        sw t1, 0(t0)
                        addi s0, s0, 1
                           beq x0, x0, outer_loop
        outer_exit:
        lw s0, 0(sp)
        lw s1, 4(sp)
        addi sp, sp, 8
        jalr x0, ra,0
main: addi a0, zero, 0
    addi a1, x0, 5
    jal ra, Insertion_Sort
exit:
```

Data file



```
0,10
4,6
8,1
12,3
16,11
```

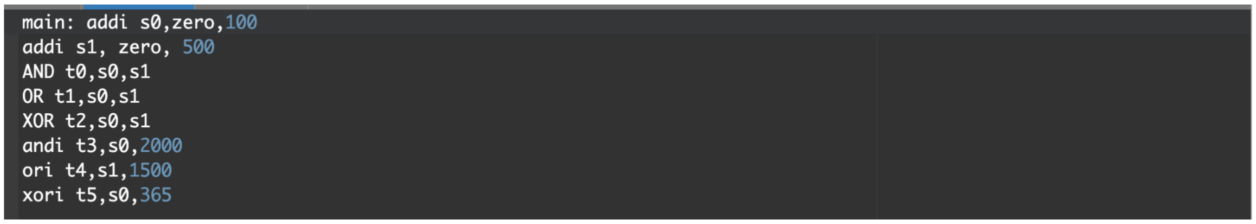
Memory output



```
Memory Location                 binary value        decimal value     hexadecimal value
0       00000000000000000000000000000001    1    0x00000001
4       00000000000000000000000000000011    3    0x00000003
8       00000000000000000000000000000110    6    0x00000006
12       00000000000000000000000000001010    10    0x0000000a
16       00000000000000000000000000001011    11    0x0000000b
39999992       00000000000000000000000000000000    0    0x00000000
39999996       00000000000000000000000000000000    0    0x00000000
```

5. Bitwise operation The bitwise will find the program and we used AND,OR,XOR,XORI,ANDI,ORI. They all worked properly as you can see.

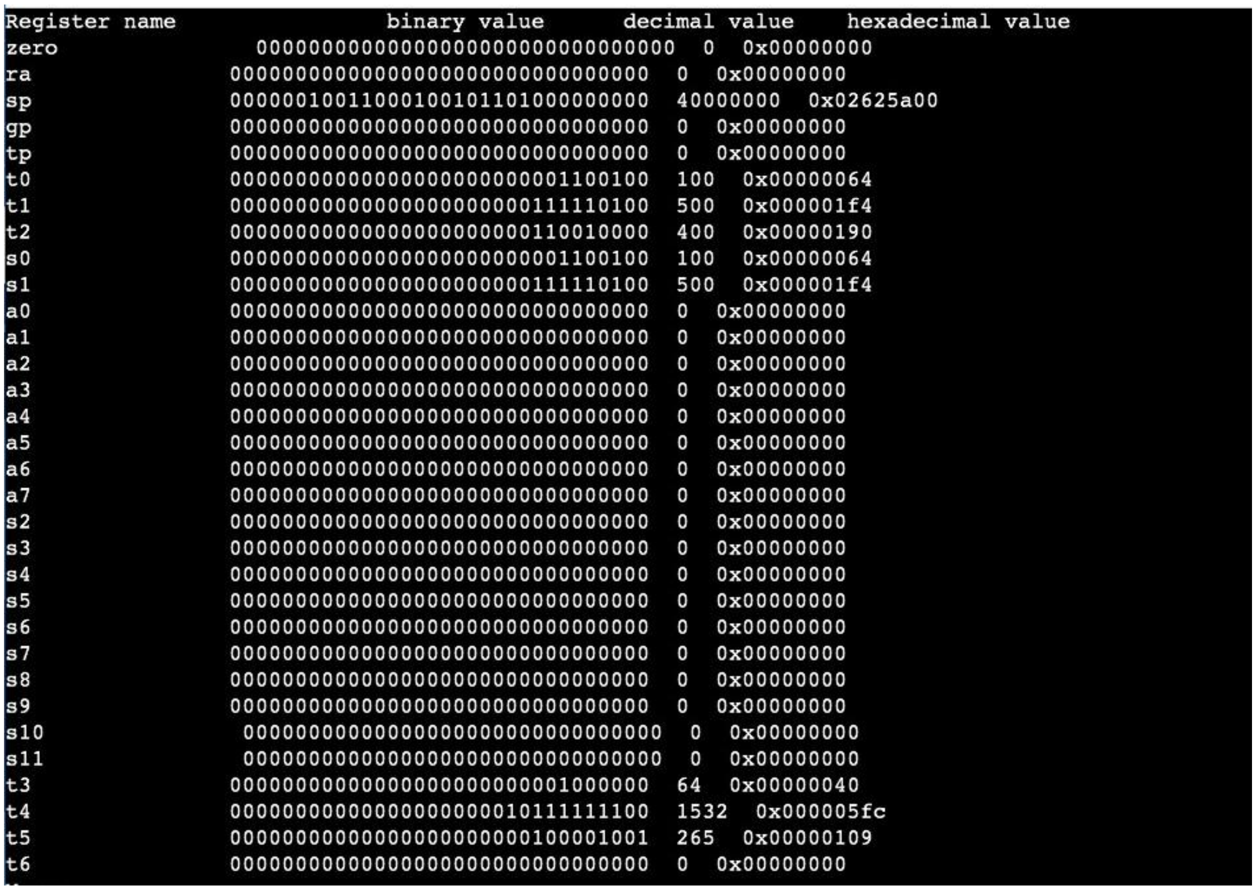
Code File



```
main: addi s0,zero,100
addi s1, zero, 500
AND t0,s0,s1
OR t1,s0,s1
XOR t2,s0,s1
andi t3,s0,2000
ori t4,s1,1500
xori t5,s0,365
```

There are no memory operands required in this example. So, we left it empty.

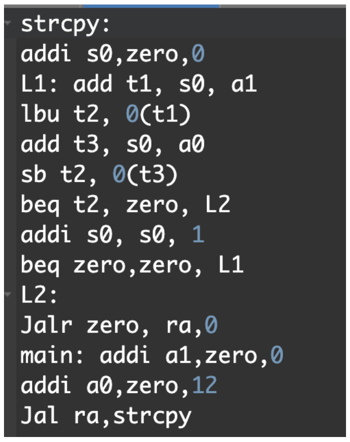
Output Registers.



```
Register name               binary value       decimal value    hexadecimal value
zero                  00000000000000000000000000000000  0  0x00000000
ra                  00000000000000000000000000000000  0  0x00000000
sp                  00000010011000100101101000000000  40000000  0x02625a00
gp                  00000000000000000000000000000000  0  0x00000000
tp                  00000000000000000000000000000000  0  0x00000000
t0                  00000000000000000000000001100100  100  0x00000064
t1                  00000000000000000000000111110100  500  0x000001f4
t2                  00000000000000000000000110010000  400  0x00000190
s0                  00000000000000000000000001100100  100  0x00000064
s1                  00000000000000000000000111110100  500  0x000001f4
a0                  00000000000000000000000000000000  0  0x00000000
a1                  00000000000000000000000000000000  0  0x00000000
a2                  00000000000000000000000000000000  0  0x00000000
a3                  00000000000000000000000000000000  0  0x00000000
a4                  00000000000000000000000000000000  0  0x00000000
a5                  00000000000000000000000000000000  0  0x00000000
a6                  00000000000000000000000000000000  0  0x00000000
a7                  00000000000000000000000000000000  0  0x00000000
s2                  00000000000000000000000000000000  0  0x00000000
s3                  00000000000000000000000000000000  0  0x00000000
s4                  00000000000000000000000000000000  0  0x00000000
s5                  00000000000000000000000000000000  0  0x00000000
s6                  00000000000000000000000000000000  0  0x00000000
s7                  00000000000000000000000000000000  0  0x00000000
s8                  00000000000000000000000000000000  0  0x00000000
s9                  00000000000000000000000000000000  0  0x00000000
s10                  00000000000000000000000000000000  0  0x00000000
s11                  00000000000000000000000000000000  0  0x00000000
t3                  00000000000000000000000001000000  64  0x00000040
t4                  00000000000000000000010111111100  1532  0x000005fc
t5                  00000000000000000000000100001001  265  0x00000109
t6                  00000000000000000000000000000000  0  0x00000000
```

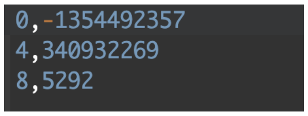
6. String Copy (This program copies the contents of a string initialized at the memory to the consecutive memory location)

Code File



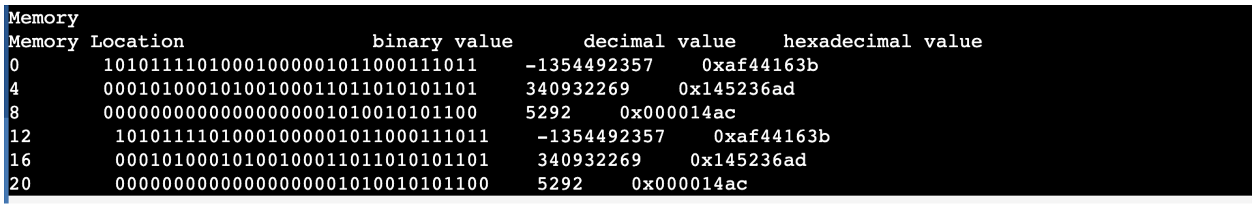
```
strcpy:
addi s0,zero,0
L1: add t1, s0, a1
lbu t2, 0(t1)
add t3, s0, a0
sb t2, 0(t3)
beq t2, zero, L2
addi s0, s0, 1
beq zero,zero, L1
L2:
Jalr zero, ra,0
main: addi a1,zero,0
addi a0,zero,12
Jal ra,strcpy
```

Memory File



```
0,-1354492357
4,340932269
8,5292
```

Output Memory



```
Memory
Memory Location                 binary value      decimal value    hexadecimal value
0       10101111010001000001011000111011    -1354492357    0xaf44163b
4       00010100010100100011011010101101    340932269    0x145236ad
8       00000000000000000001010010101100    5292    0x000014ac
12       10101111010001000001011000111011    -1354492357    0xaf44163b
16       00010100010100100011011010101101    340932269    0x145236ad
20       00000000000000000001010010101100    5292    0x000014ac
```

VI -Bugs and issues

According to the provided test programs, the simulator outputted all the correct values. However, there is always room for improvement. There are a lot of modifications and enhancements that could be added to this program. First of all, the Ecall, Fence, and Ebreak instructions could be adopted to match their real behavior instead of just stopping the execution. Moreover, many pseudo instructions could be added like li, la, j, etc. Also, directives could be added such as the data directive and the text directive. Furthermore, this project could be extended to make it output the equivalent machine code of the inputted instructions. Finally, the set of instructions supported could be extended to include RV32IM instruction set.