Project 1

-Simple ISA Calculator -

Prepared by

Lee Chang Yoon

32183641@dankook.ac.kr

Left Days: 23
March 11, 2022

Introduction

ISA is an abbreviation of Instruction Set Architecture, and it works as an interface between software and hardware. Each microprocessor has its own ISA, and these ISAs include all executable instructions that the processor can execute.

In this report, we will discuss the implementation of our own ISA and calculation program that executes each instruction sequentially. This program follows the sequence of executing an instruction of traditional computer architecture, von Neumann computer. The program has four phases of execution: Load, Fetch, and Execute. In the Load, it loads all of the instructions in “input.txt” and stores them in the memory array. Next, in the Fetch, IR gets the instructions from Memory which PC points at and decodes instruction into a set of [opcode operand1, operand2]. In the Execute, the program executes the instruction, prints out the result of the instruction to the console screen, and records log in “output.txt”. Execute phase includes binary arithmetic operations (+, -, *, /), mover instruction (M), halt instruction (H), jump instruction (J), compare instruction (C), and branch instruction (B).

This report consists of 4 sections. First, we will talk about concepts used in the calculator program including the instruction set manual. Next is the program structure. We will discuss the sequence of program execution. Third, we will state the details for instructions. In the last of the report, we will present the build environment and image files for the results.

1. Concepts – Von Neumann Architecture

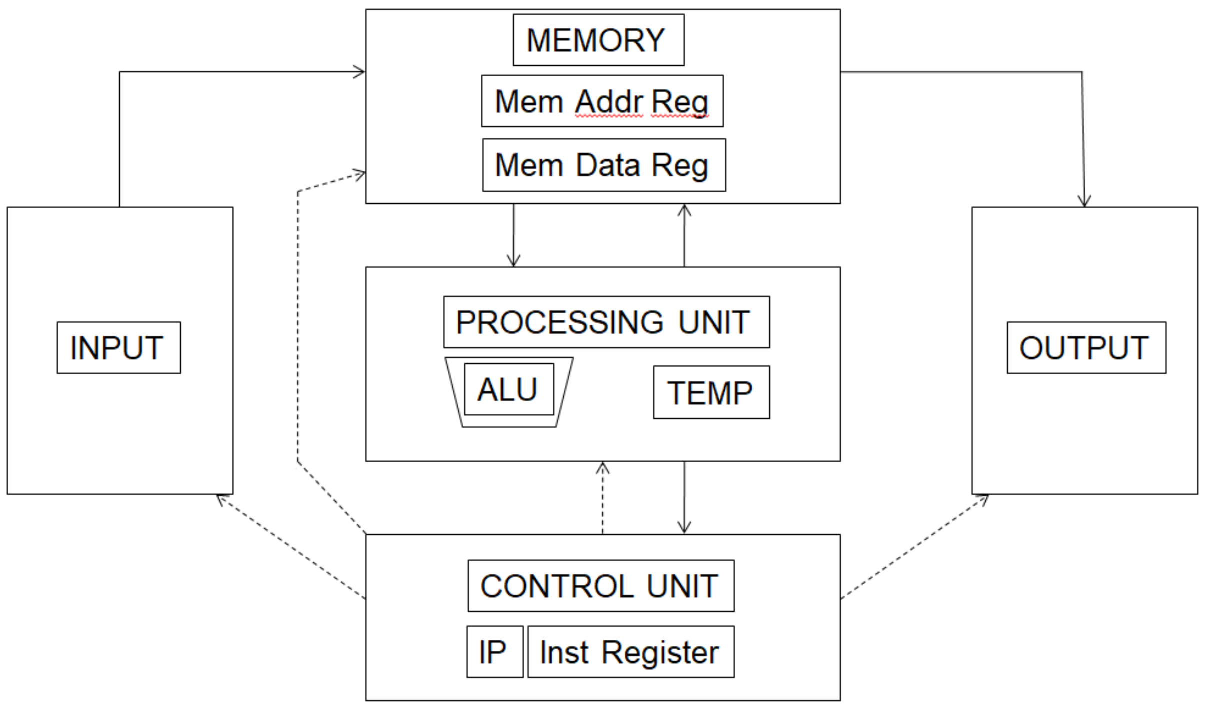


Figure 1 - Von Neumann Model

The Von Neumann architecture is a computer architecture based on a description by John Von Neumann in 1945. Figure 1 shows the model of the Von Neumann architecture. According to Figure 1, the Von Neumann computer consists of a processing unit containing an arithmetic logic unit (ALU) and processor register, a controlling unit containing an instruction register and instruction pointer (program counter), a memory to store both data and instructions, and input and output mechanisms. The meaning has evolved to be any stored-program computer in which instruction fetch and a data operation cannot occur at the same time because of the share of the common bus.

Von Neumann's machine executes a program in the following sequence: fetch – decode – execute – store. First, the control unit fetches the next instruction from the memory. The instruction pointer (IP) or program counter (PC) in the control unit contains the address of the next instruction to fetch. The memory unit then reads the bytes stored at the specified address and sends them to the control unit on the data bus. The instruction register (IR) stores the bytes of the instructions received from the memory unit. The control unit also increments the IP or PC value to store the address of the new next instruction to fetch. Next, the control unit decodes the instructions stored in IR. Third, the processing unit executes the instruction. Lastly, the control unit stores the results in memory.

2. Program Structure

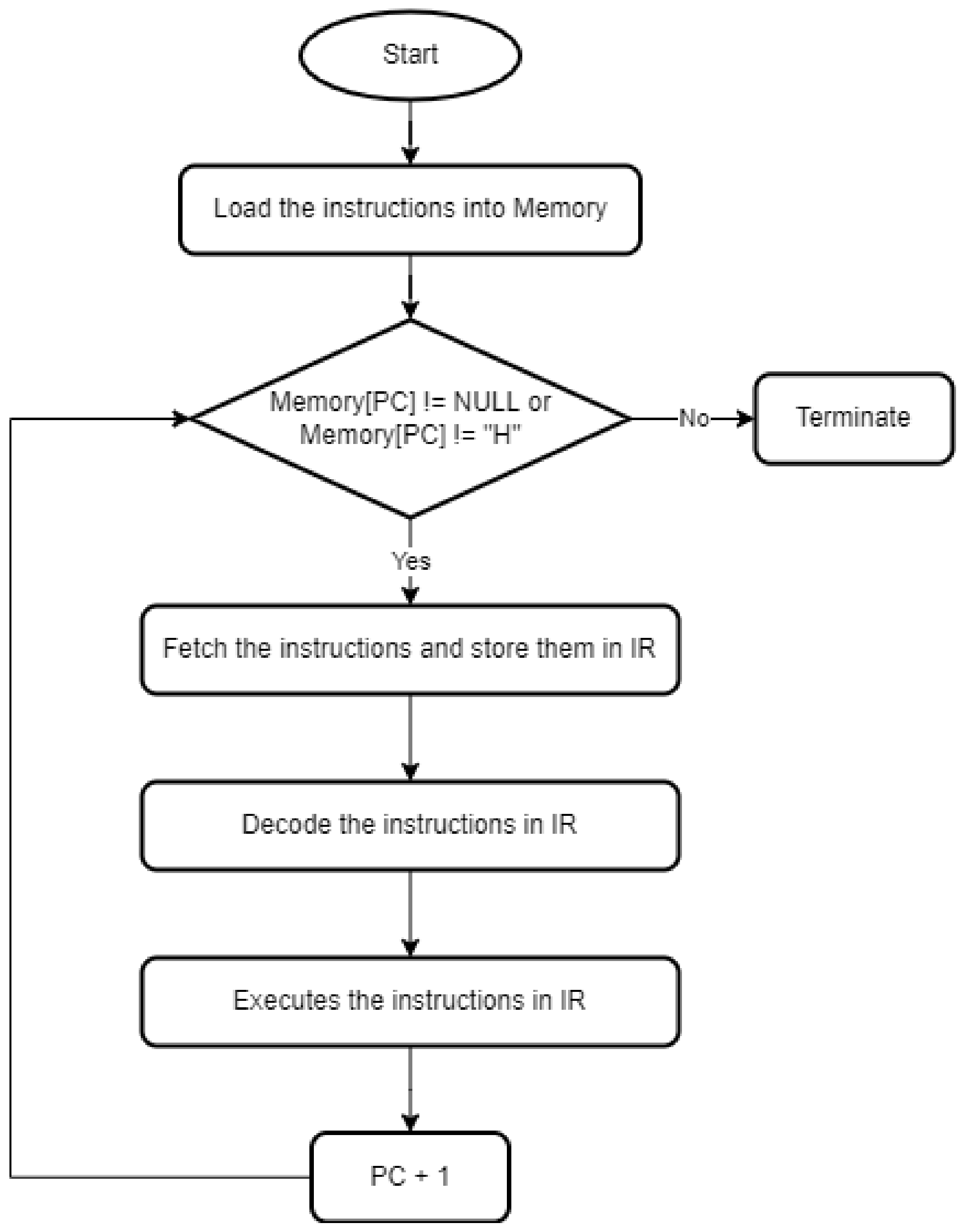
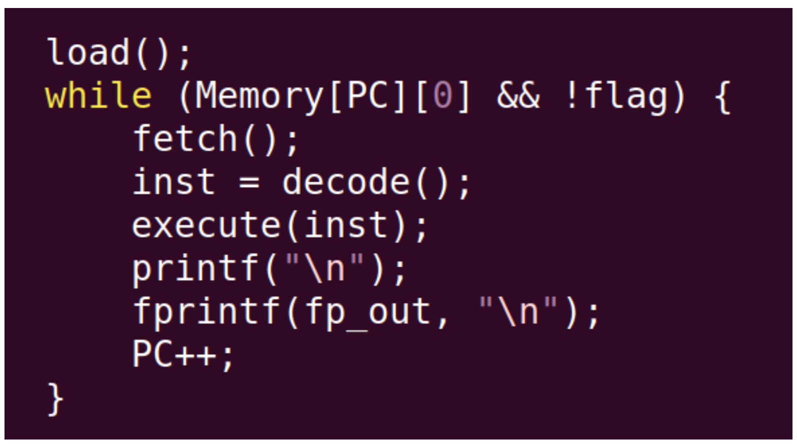

Start
Load the instructions into Memory
Memory[PC] != NULL or Memory[PC] != "H"
No
Terminate
Yes
Fetch the instructions and store them in IR
Decode the instructions in IR
Executes the instructions in IR
PC + 1

Figure 2 - Flow Chart of Calculator Program

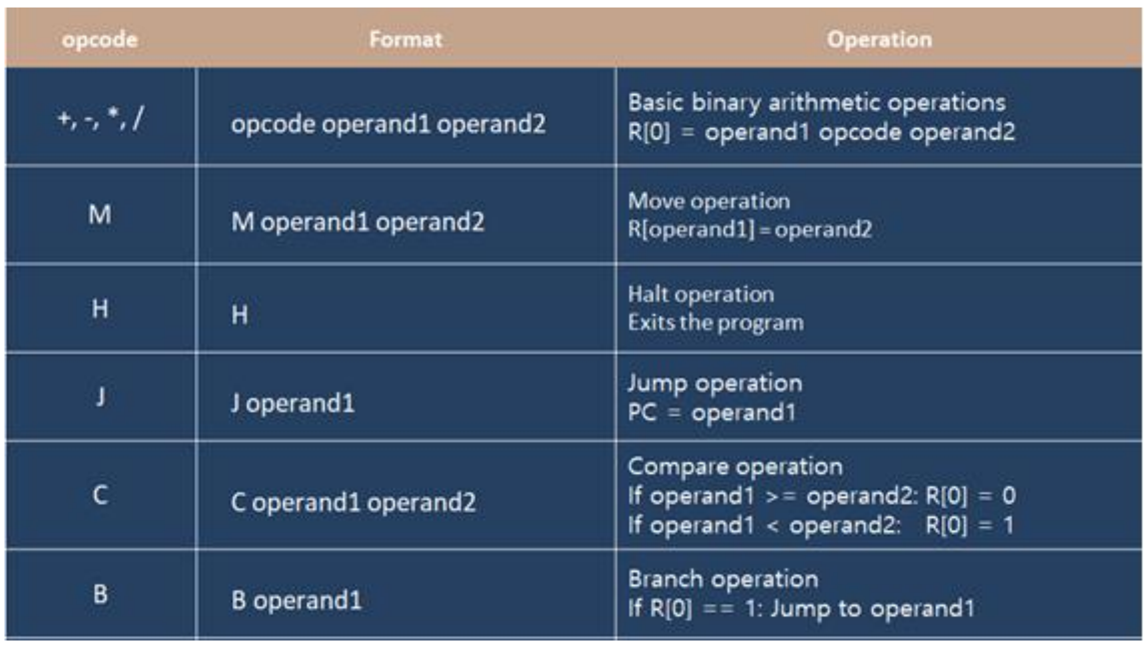


```
load();
while (Memory[PC][0] && !flag) {
    fetch();
    inst = decode();
    execute(inst);
    printf("\n");
    fprintf(fp_out, "\n");
    PC++;
}
```

Figure 3 - Main

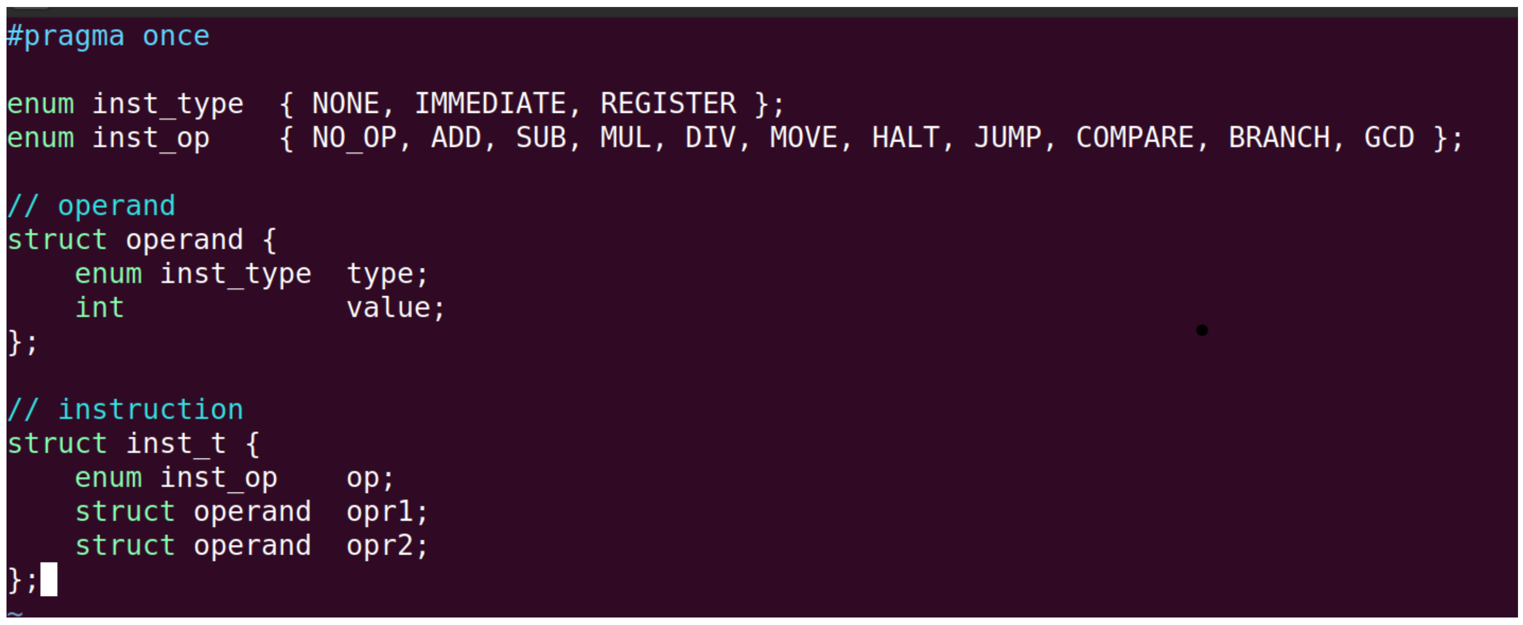
According to the sequence of executing a program in the Von Neumann machine (section 1), we can check that it follows the phase of fetch – decode – execute – store. In the calculator program, we appended the load, fetch, decode, and execute operations. Figure 2 shows the flow of the calculator program. In Figure 3, the calculator program has four phases of executing the program: load – fetch – decode – execute. First, in load, the program reads all of the instructions one by one from the "input.txt" by using the fgets() function and stores those instructions in the Memory array. In fetch, the program gets the instruction from Memory[PC] and stores them in the instruction register (IR). In decode, the program decodes IR in a certain method. The methods of decoding the instructions will be described in detail in section 3. Then, the program executes decoded instructions in following rules. These rules are also explained in section 3. In each phase of fetch, decode, and execute the operation, after processing their operations, the program prints the result on the console screen and writes (stores) result in "output.txt". Once done, the PC will be updated. This consequence of the execution is operated until it reaches the end of the program file (EOF) or gets the instruction of halt (H).

3. Instructions



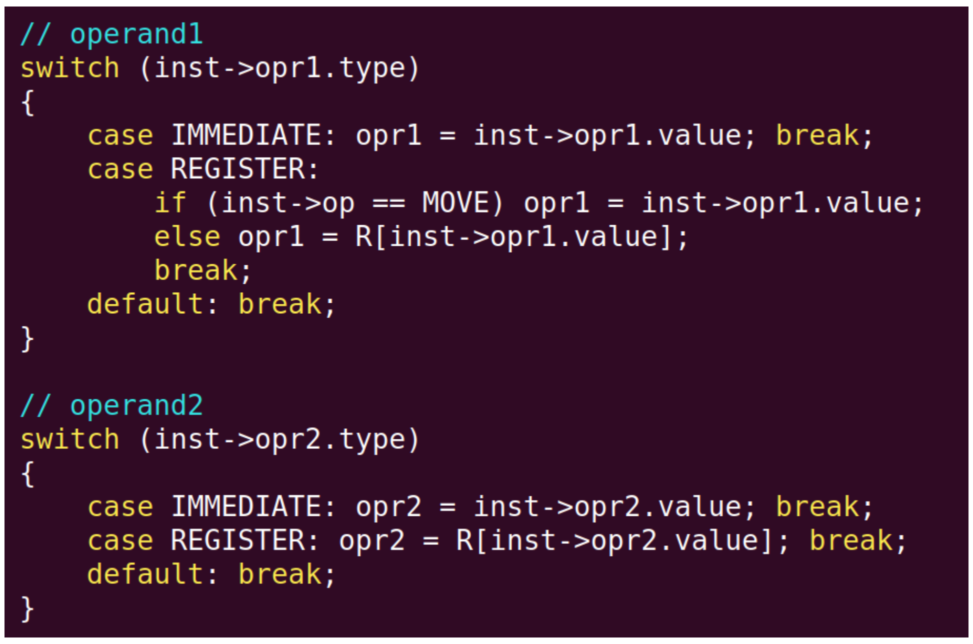
opcode	Format	Operation
+, -, *, /	opcode operand1 operand2	Basic binary arithmetic operations R[0] = operand1 opcode operand2
M	M operand1 operand2	Move operation R[operand1] = operand2
H	H	Halt operation Exits the program
J	J operand1	Jump operation PC = operand1
C	C operand1 operand2	Compare operation If operand1 >= operand2: R[0] = 0 If operand1 < operand2: R[0] = 1
B	B operand1	Branch operation If R[0] == 1: Jump to operand1

Figure 4 - Instruction Table of Calculator



```
#pragma once

enum inst_type  { NONE, IMMEDIATE, REGISTER };
enum inst_op    { NO_OP, ADD, SUB, MUL, DIV, MOVE, HALT, JUMP, COMPARE, BRANCH, GCD };

// operand
struct operand {
    enum inst_type  type;
    int             value;
};

// instruction
struct inst_t {
    enum inst_op    op;
    struct operand  opr1;
    struct operand  opr2;
};
```

Figure 5 - Instruction Structure



```
// operand1
switch (inst->opr1.type)
{
    case IMMEDIATE: opr1 = inst->opr1.value; break;
    case REGISTER:
        if (inst->op == MOVE) opr1 = inst->opr1.value;
        else opr1 = R[inst->opr1.value];
        break;
    default: break;
}

// operand2
switch (inst->opr2.type)
{
    case IMMEDIATE: opr2 = inst->opr2.value; break;
    case REGISTER: opr2 = R[inst->opr2.value]; break;
    default: break;
}
```

Figure 6 - Decode of operand1 and operand2

Figure 4 shows the instruction set of the calculator program. It represents the format and operation of the particular instructions of the program.

According to Figure 4, in the case of arithmetic (+, -, *, /), move (M), and compare (C), they need opcode, operand1, and operand2. However, in halt (H), jump (J), and branch (B) operations, they only need either opcode and operand1 or only opcode. The difference in instructions needs to be handled in the execute phase.

Figure 5 shows the enumerations of instruction type and operation and the structure of operand and instruction that is used in the decode phase. Instruction type (inst_type), represents whether the operand is NULL, immediate, or registers. Instruction operation (inst_op) represents the type of operation. Operand structure stores the type of operand. If the operand type is registered, it stores the index of the register. If the operand type is immediate, it stores hexadecimal numbers itself. If it is none, it just stores 0, which represents nothing. The instruction structure stores all of the information mentioned above. In the decode phase, the program decodes the instruction and stores data in the following format. After decoding, it sends the following format to execute.c.

Figure 6 shows how we implemented this function in execute.c. The execute.c takes an argument of decoded instruction in the format represented in Figure 5. In the case of operand1, if the type of operand1 is register, we have to check if the operation type is move or not. If the operation is move, the program stores the index of the following register into opr1, which is the variable that is used for the operations. Else, it stores the data in the following register. If the type of operand1 is immediate, it stores the decimal number in opr1. The same operations are implemented for operand2. However, because the program does not have to care about move, jump, and branch operation for operand2, the program stores whether the index of the register or decimal number into opr2, which takes the same role of opr1.

By implementing codes in Figure 5 and Figure 6, we can execute the instructions by using the enumerations and instructions efficiently.

4. Build Environment and Results

- Build Environments:

  Linux environment – Vi editor, GCC complier

  Program is built by using the Makefile.

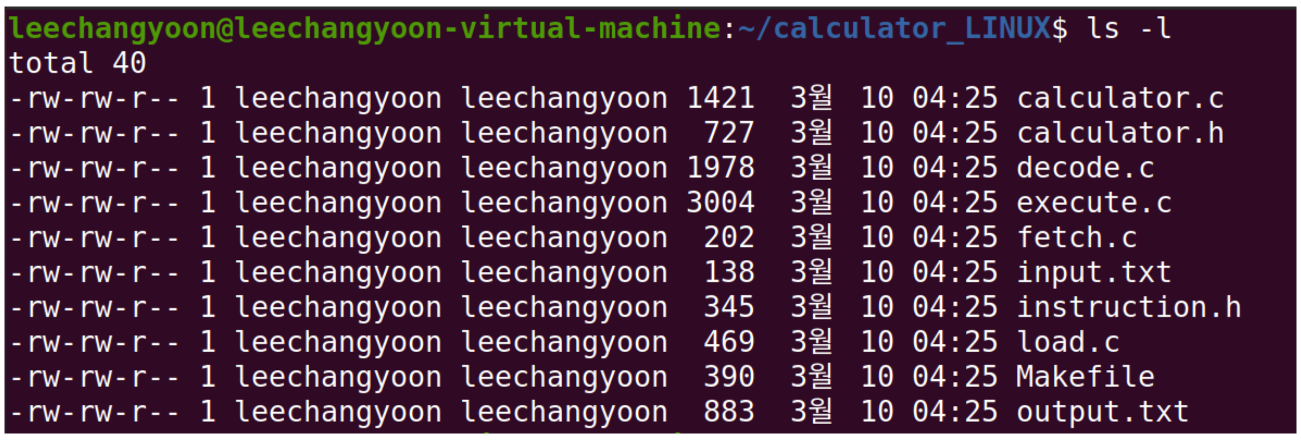
Make command:

1. $make main -> build the execution program

2. $make clean -> clean the object files of calculator.c, load.c, fetch.c, decode.c, and execute.c

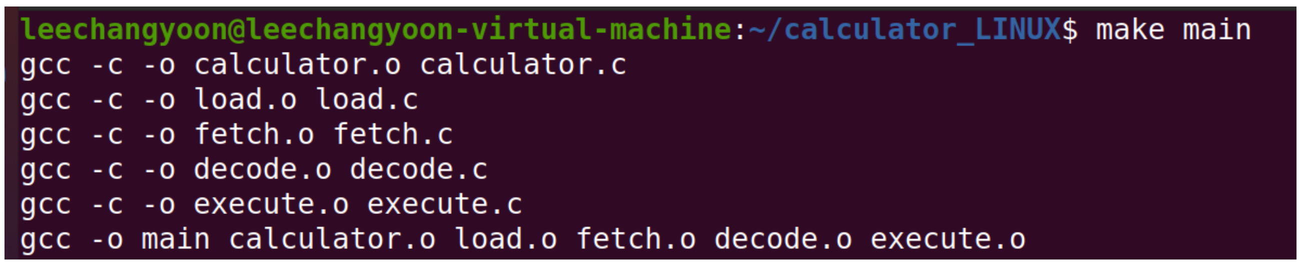
- Results:

Files:



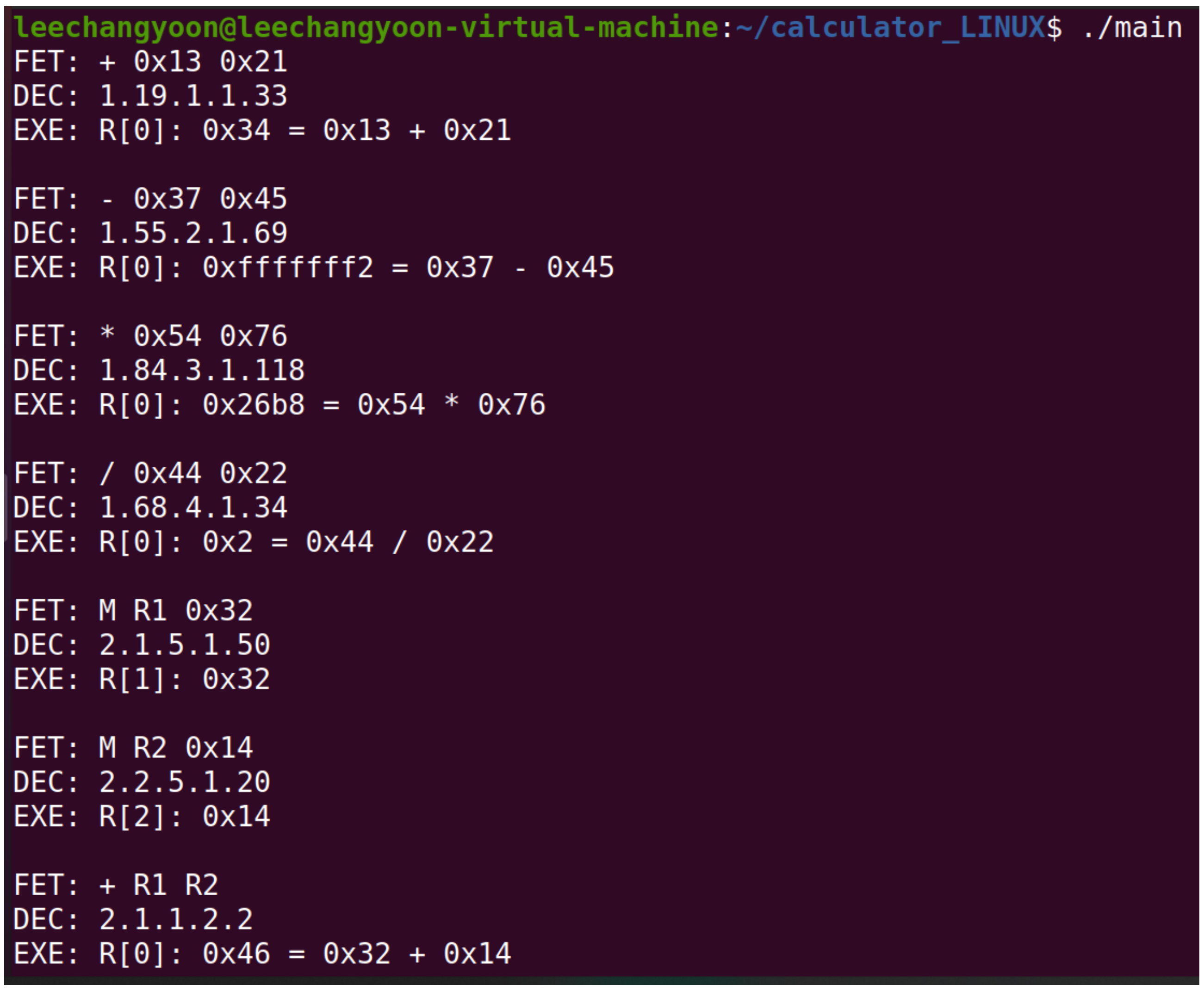
```
leechangyoon@leechangyoon-virtual-machine:~/calculator_LINUX$ ls -l
total 40
-rw-rw-r-- 1 leechangyoon leechangyoon 1421  3월  10 04:25 calculator.c
-rw-rw-r-- 1 leechangyoon leechangyoon  727  3월  10 04:25 calculator.h
-rw-rw-r-- 1 leechangyoon leechangyoon 1978  3월  10 04:25 decode.c
-rw-rw-r-- 1 leechangyoon leechangyoon 3004  3월  10 04:25 execute.c
-rw-rw-r-- 1 leechangyoon leechangyoon  202  3월  10 04:25 fetch.c
-rw-rw-r-- 1 leechangyoon leechangyoon  138  3월  10 04:25 input.txt
-rw-rw-r-- 1 leechangyoon leechangyoon  345  3월  10 04:25 instruction.h
-rw-rw-r-- 1 leechangyoon leechangyoon  469  3월  10 04:25 load.c
-rw-rw-r-- 1 leechangyoon leechangyoon  390  3월  10 04:25 Makefile
-rw-rw-r-- 1 leechangyoon leechangyoon  883  3월  10 04:25 output.txt
```

$make main

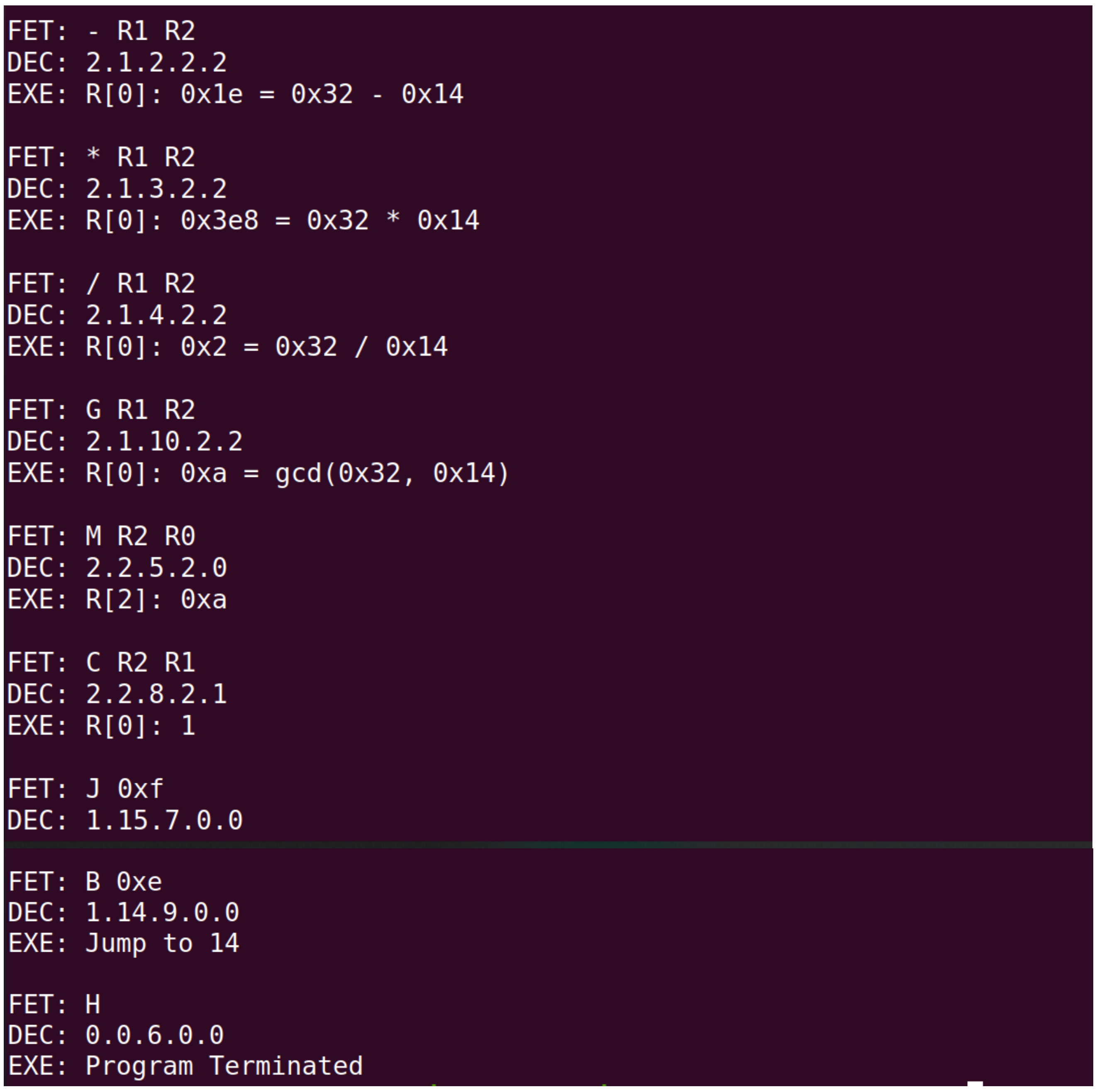


```
leechangyoon@leechangyoon-virtual-machine:~/calculator_LINUX$ make main
gcc -c -o calculator.o calculator.c
gcc -c -o load.o load.c
gcc -c -o fetch.o fetch.c
gcc -c -o decode.o decode.c
gcc -c -o execute.o execute.c
gcc -o main calculator.o load.o fetch.o decode.o execute.o
```

Result of $./main and output.txt

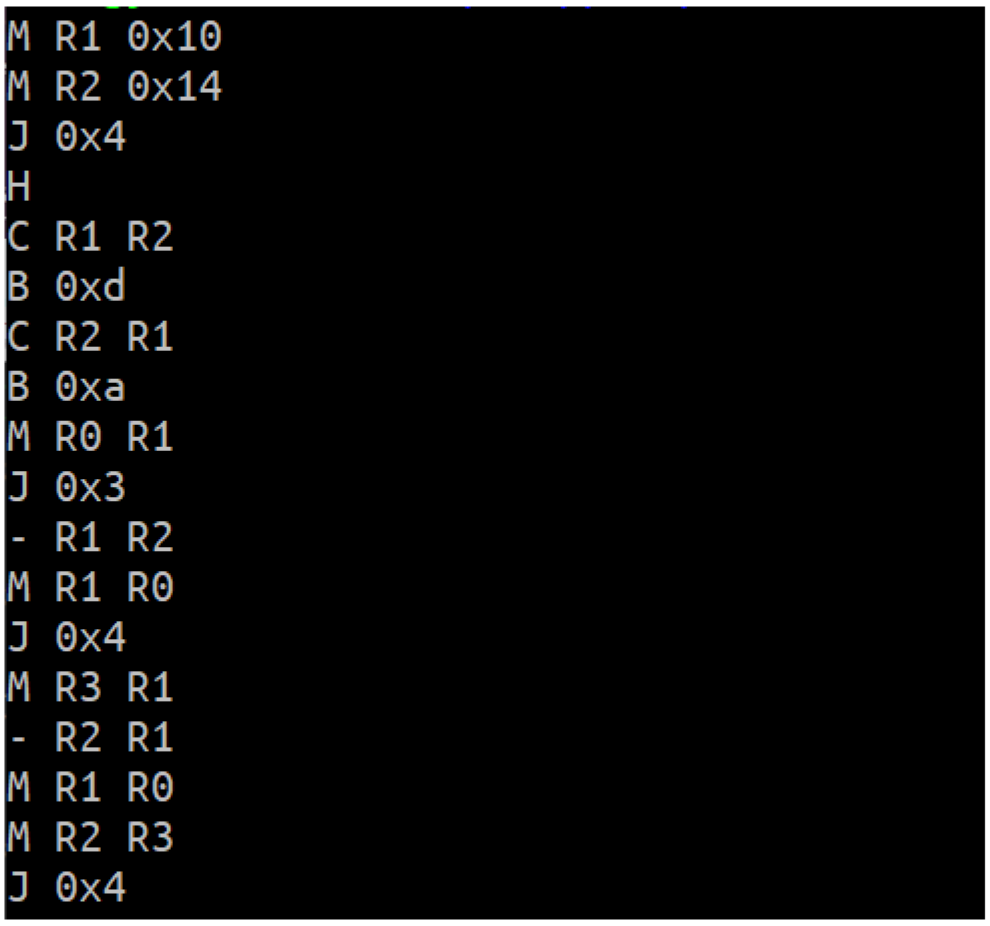


```
leechangyoon@leechangyoon-virtual-machine:~/calculator_LINUX$ ./main
FET: + 0x13 0x21
DEC: 1.19.1.1.33
EXE: R[0]: 0x34 = 0x13 + 0x21

FET: - 0x37 0x45
DEC: 1.55.2.1.69
EXE: R[0]: 0xfffffff2 = 0x37 - 0x45

FET: * 0x54 0x76
DEC: 1.84.3.1.118
EXE: R[0]: 0x26b8 = 0x54 * 0x76

FET: / 0x44 0x22
DEC: 1.68.4.1.34
EXE: R[0]: 0x2 = 0x44 / 0x22

FET: M R1 0x32
DEC: 2.1.5.1.50
EXE: R[1]: 0x32

FET: M R2 0x14
DEC: 2.2.5.1.20
EXE: R[2]: 0x14

FET: + R1 R2
DEC: 2.1.1.2.2
EXE: R[0]: 0x46 = 0x32 + 0x14
```



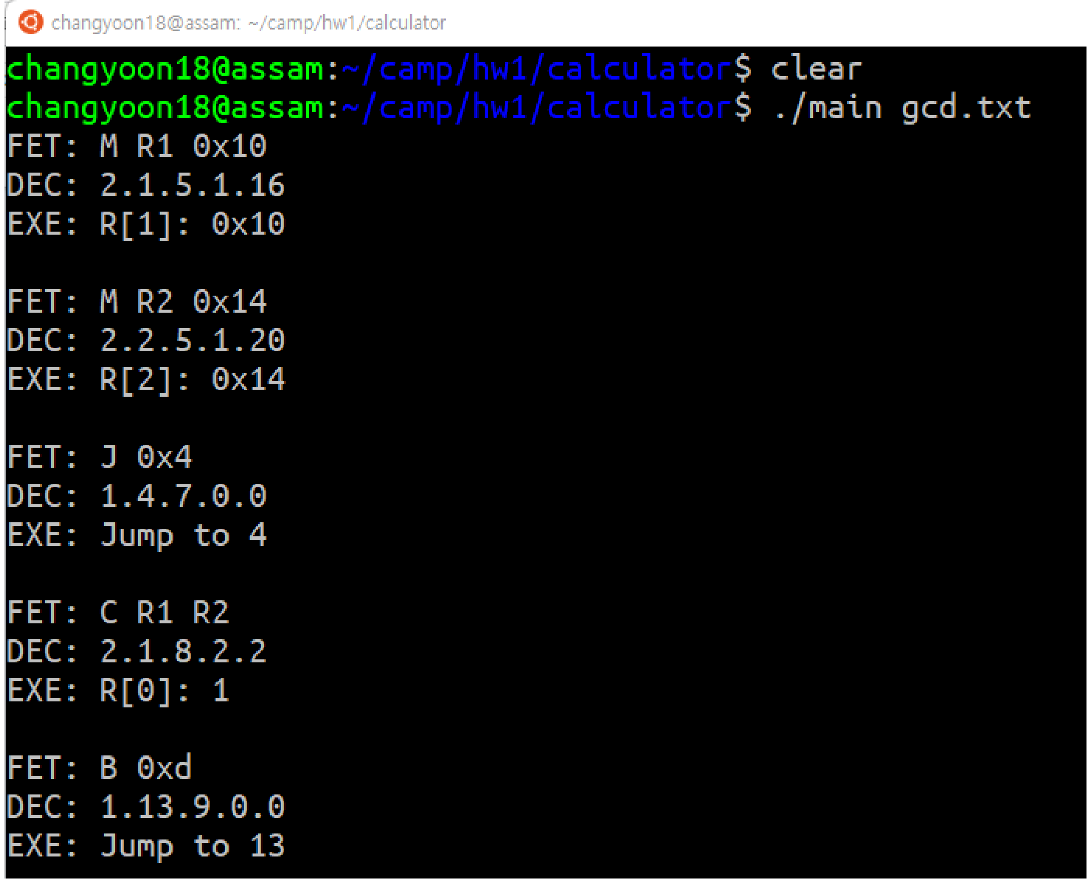
```
FET: - R1 R2
DEC: 2.1.2.2.2
EXE: R[0]: 0x1e = 0x32 - 0x14

FET: * R1 R2
DEC: 2.1.3.2.2
EXE: R[0]: 0x3e8 = 0x32 * 0x14

FET: / R1 R2
DEC: 2.1.4.2.2
EXE: R[0]: 0x2 = 0x32 / 0x14

FET: G R1 R2
DEC: 2.1.10.2.2
EXE: R[0]: 0xa = gcd(0x32, 0x14)

FET: M R2 R0
DEC: 2.2.5.2.0
EXE: R[2]: 0xa

FET: C R2 R1
DEC: 2.2.8.2.1
EXE: R[0]: 1

FET: J 0xf
DEC: 1.15.7.0.0

FET: B 0xe
DEC: 1.14.9.0.0
EXE: Jump to 14

FET: H
DEC: 0.0.6.0.0
EXE: Program Terminated
```

GCD program



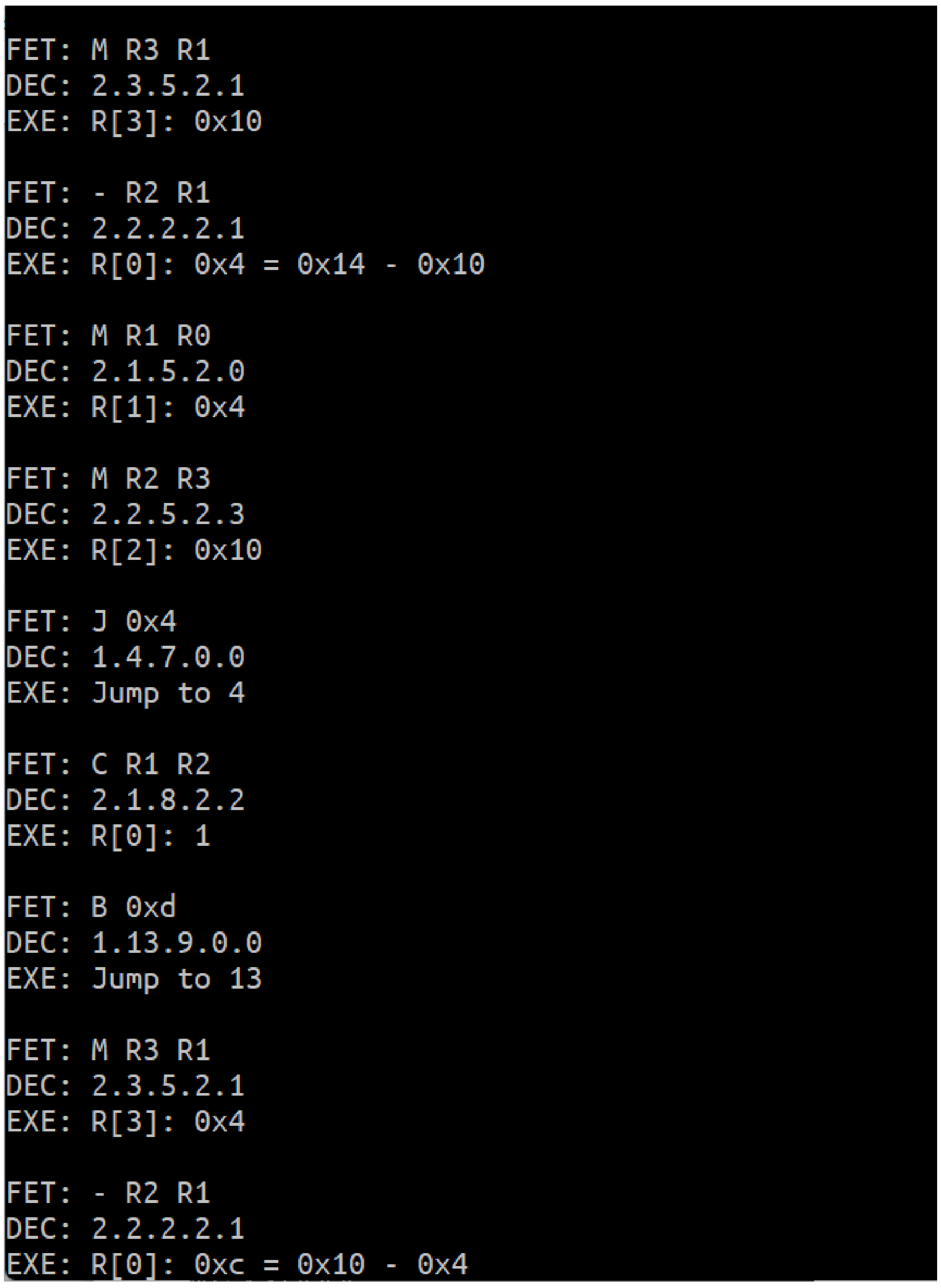
```
M R1 0x10
M R2 0x14
J 0x4
H
C R1 R2
B 0xd
C R2 R1
B 0xa
M R0 R1
J 0x3
- R1 R2
M R1 R0
J 0x4
M R3 R1
- R2 R1
M R1 R0
M R2 R3
J 0x4
```

Result of $./main and gcd.txt

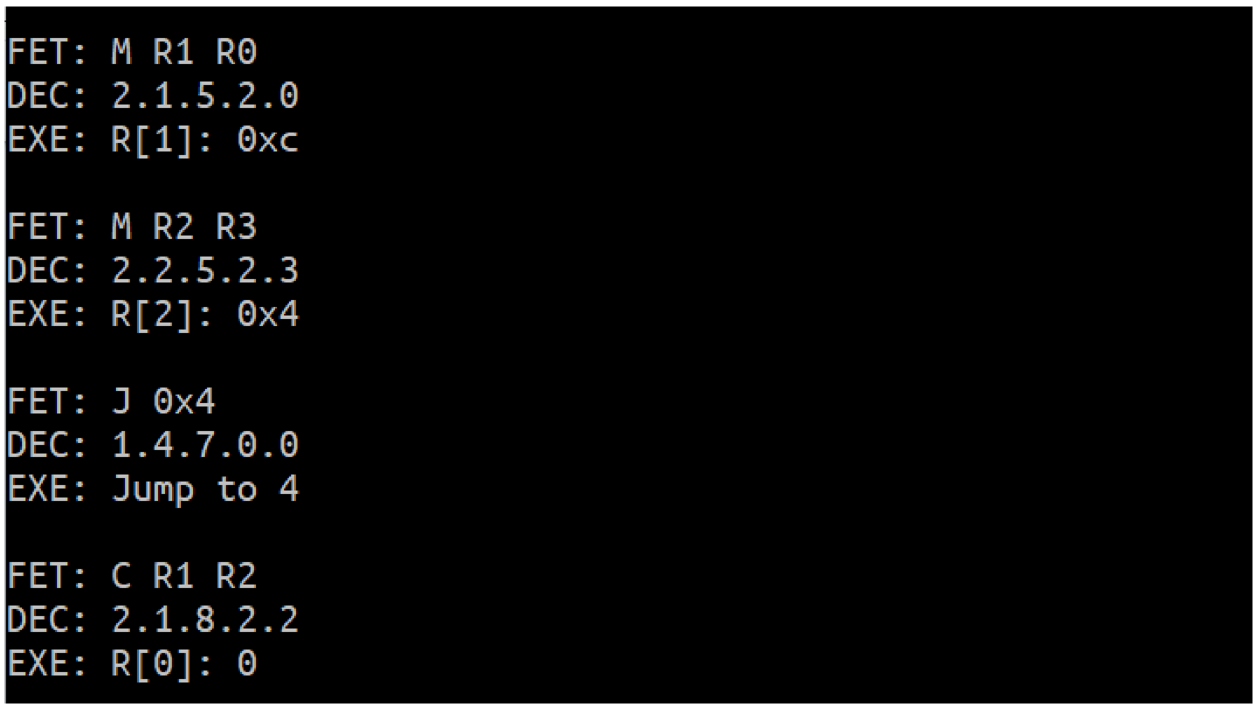


changyoon18@assam: ~/camp/hw1/calculator

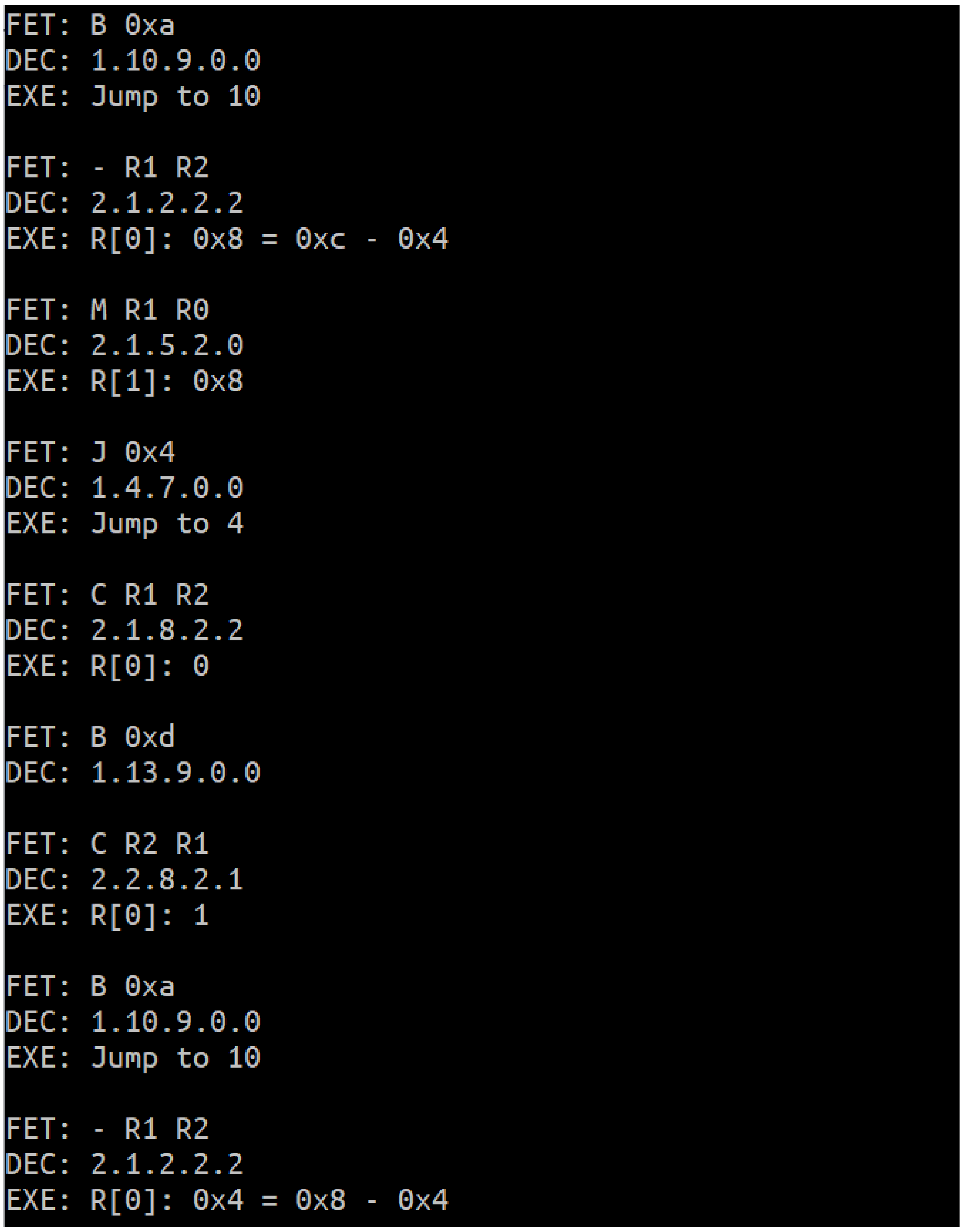
```
changyoon18@assam:~/camp/hw1/calculator$ clear
changyoon18@assam:~/camp/hw1/calculator$ ./main gcd.txt
FET: M R1 0x10
DEC: 2.1.5.1.16
EXE: R[1]: 0x10

FET: M R2 0x14
DEC: 2.2.5.1.20
EXE: R[2]: 0x14

FET: J 0x4
DEC: 1.4.7.0.0
EXE: Jump to 4

FET: C R1 R2
DEC: 2.1.8.2.2
EXE: R[0]: 1

FET: B 0xd
DEC: 1.13.9.0.0
EXE: Jump to 13
```



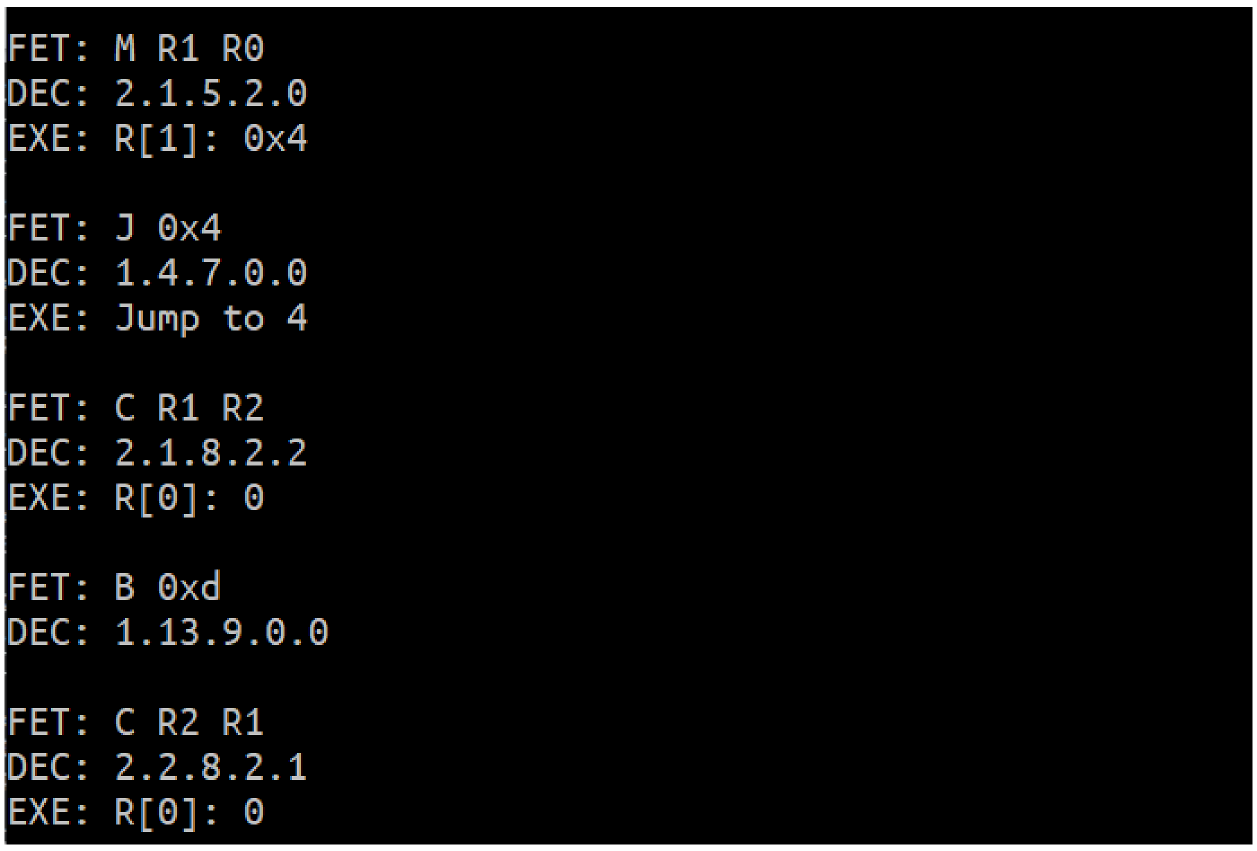
```
FET: M R3 R1
DEC: 2.3.5.2.1
EXE: R[3]: 0x10

FET: - R2 R1
DEC: 2.2.2.2.1
EXE: R[0]: 0x4 = 0x14 - 0x10

FET: M R1 R0
DEC: 2.1.5.2.0
EXE: R[1]: 0x4

FET: M R2 R3
DEC: 2.2.5.2.3
EXE: R[2]: 0x10

FET: J 0x4
DEC: 1.4.7.0.0
EXE: Jump to 4

FET: C R1 R2
DEC: 2.1.8.2.2
EXE: R[0]: 1

FET: B 0xd
DEC: 1.13.9.0.0
EXE: Jump to 13

FET: M R3 R1
DEC: 2.3.5.2.1
EXE: R[3]: 0x4

FET: - R2 R1
DEC: 2.2.2.2.1
EXE: R[0]: 0xc = 0x10 - 0x4
```



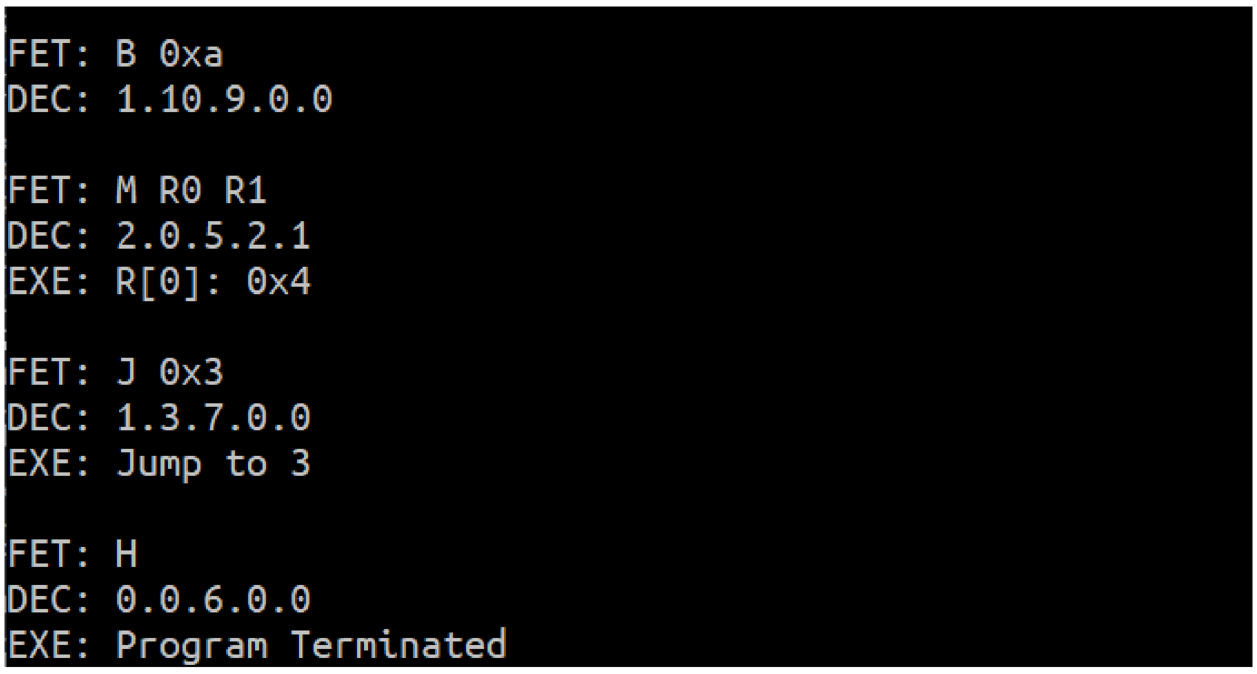
```
FET: M R1 R0
DEC: 2.1.5.2.0
EXE: R[1]: 0xc

FET: M R2 R3
DEC: 2.2.5.2.3
EXE: R[2]: 0x4

FET: J 0x4
DEC: 1.4.7.0.0
EXE: Jump to 4

FET: C R1 R2
DEC: 2.1.8.2.2
EXE: R[0]: 0
```



```
FET: B 0xa
DEC: 1.10.9.0.0
EXE: Jump to 10

FET: - R1 R2
DEC: 2.1.2.2.2
EXE: R[0]: 0x8 = 0xc - 0x4

FET: M R1 R0
DEC: 2.1.5.2.0
EXE: R[1]: 0x8

FET: J 0x4
DEC: 1.4.7.0.0
EXE: Jump to 4

FET: C R1 R2
DEC: 2.1.8.2.2
EXE: R[0]: 0

FET: B 0xd
DEC: 1.13.9.0.0

FET: C R2 R1
DEC: 2.2.8.2.1
EXE: R[0]: 1

FET: B 0xa
DEC: 1.10.9.0.0
EXE: Jump to 10

FET: - R1 R2
DEC: 2.1.2.2.2
EXE: R[0]: 0x4 = 0x8 - 0x4
```



```
FET: M R1 R0
DEC: 2.1.5.2.0
EXE: R[1]: 0x4

FET: J 0x4
DEC: 1.4.7.0.0
EXE: Jump to 4

FET: C R1 R2
DEC: 2.1.8.2.2
EXE: R[0]: 0

FET: B 0xd
DEC: 1.13.9.0.0

FET: C R2 R1
DEC: 2.2.8.2.1
EXE: R[0]: 0
```



```
FET: B 0xa
DEC: 1.10.9.0.0

FET: M R0 R1
DEC: 2.0.5.2.1
EXE: R[0]: 0x4

FET: J 0x3
DEC: 1.3.7.0.0
EXE: Jump to 3

FET: H
DEC: 0.0.6.0.0
EXE: Program Terminated
```

Conclusion

The purpose of this project is to build a simple calculating program that works with ISA. When we started to implement the program, we just decided to make it as simple as possible. However, the reason that it works with ISA, we built the program in a similar way to the execution in Von Neumann computer. By implementing the program, we have learned about what happens in each phase of the simple Von Neumann architecture. ~~We are glad that we finished the project with desirable results in the Visual Studio Code environment by using GCC compiler. However, because we have failed the same program with the same compiler in the Linux environment by using Makefile, we will handle the outputs in future work. We will attach the file with the Visual Studio Code version and also the Linux version.~~ In short, we have implemented the program that runs in Linux environment. One problem that we suffered was that when we send the files from the Windows environment to the Linux server, the new line characters in the files, which are written in the Windows environment, are read differently in the Linux server. In Windows, it uses CR and LF for the newline character. However, in Linux, it only uses LF for the newline character. Therefore, in the input.txt file, there were ^M characters at the end of the lines. In short, by removing those ^M characters by using commands “:%s/^M//g” in the VI editor, we handled this problem.

Citations

- ”Von Neumann Architecture.” Von Neumann Architecture - an overview | ScienceDirect Topics. Accessed March 9, 2022. https://www.sciencedirect.com/topics/computer-science/von-neumann-architecture.

- ”The Von Neumann Architecture.” Dive into systems. Accessed March 9, 2022. https://diveintosystems.org/book/C5-Arch/von.html.