Project 1

Simple Calculator

Prepared by ChoJungHee

32194394@dankook.ac.kr

Introduction

The goal of this project is to implement a calculator using a simple ISA (Instruction Set Architecture).

ISA is an architecture that defines the set of instructions a processor can perform. In this project, I select the simple ISA required by the lecture and use it to implement a simple calculator.

I used von Neumann's architecture to complete this project. .The program has four step of execution : Load, Fetch, Decode and Execute.

First, in Load, it loads all of the instructions in “input.txt” and stores them in the instruction memory array. Next, in Fetch, IR(instruction register) get instructions from Memory which PC value and increase PC. In Decode, decodes instruction from IR register and seperate operand, opcode, optypes. Finally, in the Execute, the program executes the instruction and prints out the changes of register to the console screen.

This report has four table of contents. First, i will talk about concepts used in the calculator program and considerations for implementation. Next is the program implement. I will explain how I designed this program and why I used that design. Third, I will describe what environment Ideveloped program and how I built it. In the last of the report, I will explain what was hard, what I thought while doing the project.

Background

> Concept

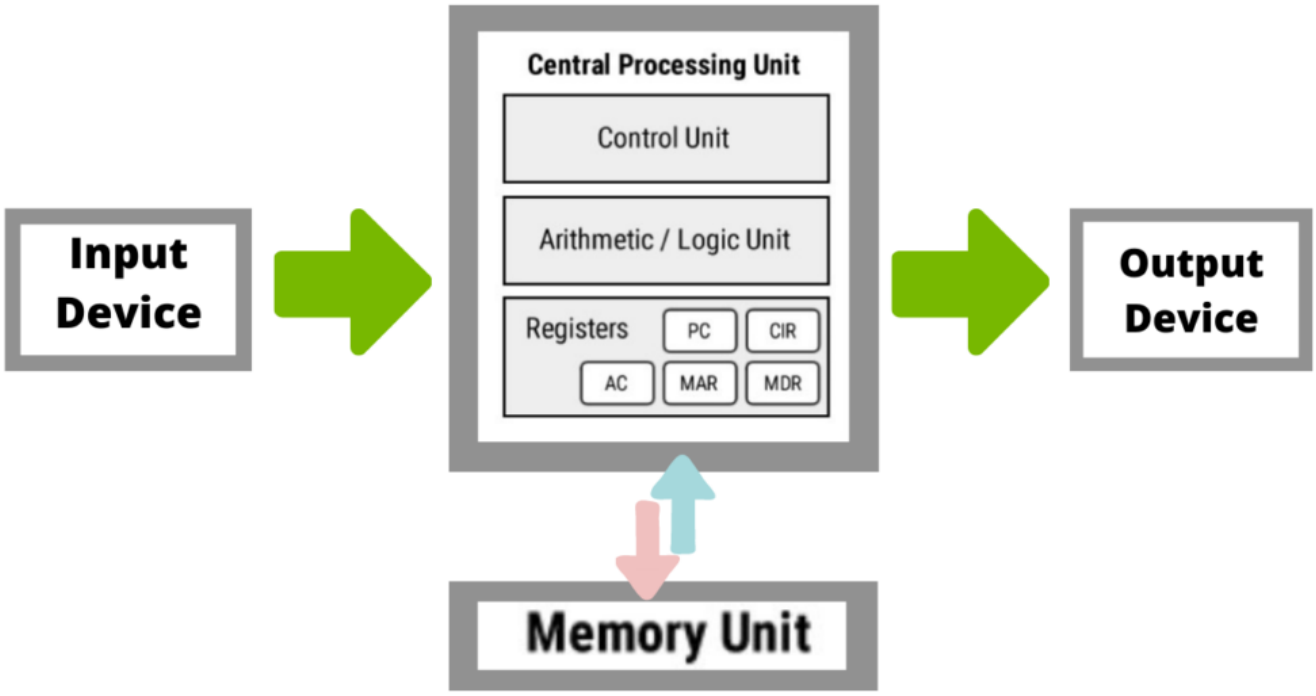


figure 1 : concept of von Neumann

The von Neumann's architecture consists of four main components: the memory, the CPU, the input/output devices, and the system bus.

- The CPU is responsible for executing instructions and performing calculations. It contains an ALU (Arithmetic Logic Unit) for performing arithmetic and logic operations, as well as registers for storing data and intermediate results. And the CPU consists of three major components: the control unit, the arithmetic logic unit (ALU), and the registers.
- Memory stores both data and instructions in the same address space, which allows the CPU to access them quickly.
- The bus is the communication pathway that allows data to be transferred between the CPU, memory, and input/output devices. It consists of three types of lines: data lines for transferring data, address lines for specifying memory locations, and control lines for managing the flow of data.

The fetch-decode-execute cycle is the fundamental cycle of operation in the von Neumann architecture. It consists of the following steps:

1. Fetch: The CPU retrieves instruction from memory at the address specified by the program counter (PC).
2. Decode: The CPU interprets the opcode of the instruction and determines the operation to be performed.
3. Execute: The CPU performs the operation specified by the instruction on the operands, which may be in memory or in registers.

This cycle is repeated continuously as long as the program is running, with the PC being increase after each instruction is executed to point to the next instruction in memory.

Implement

This program follows the fetch-decode-excute cycle flow of the von Neumann structure. The reasons for using the von Neumann structure in this program are as follows. The goal of this project is to create a simple calculator program. This simple calculator must process (calculate) information and store the results in a register. In general, this is the same as the root of computers. So I use of the von Neumann architecture, a modern general-purpose computer architecture, to design this program.

In addition, the ISA should be decoded in the program. To make this easier and more intuitive, it seemed good to use the von Neumann architecture.

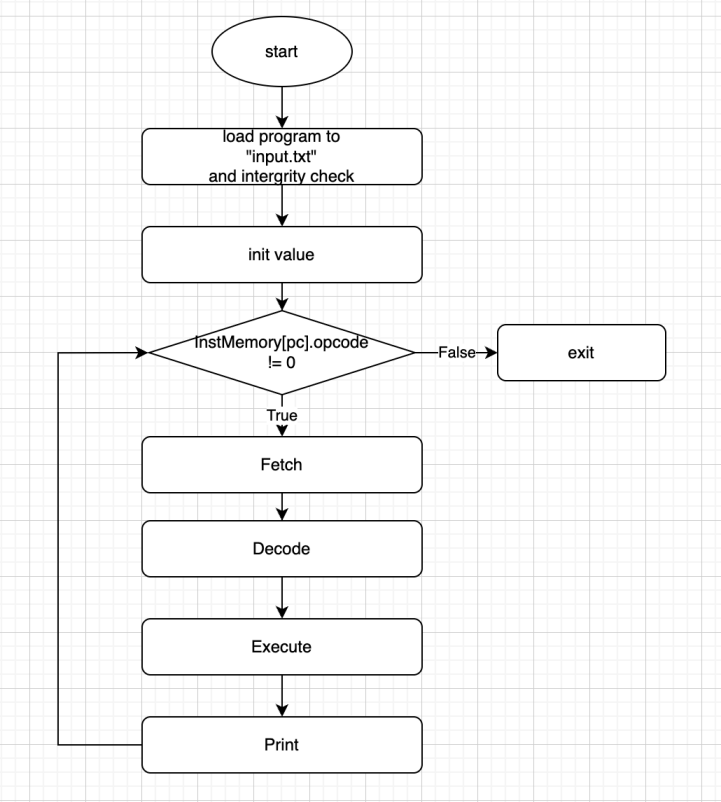


Figure 2 : flowchart

```
load(argv[1]);
init();
while(InstMemory[pc].opcode){
    printf("PC : %d\t\t| ",pc+1);

    fetch();
    decode();
    execute();

    print_register();
}
```

figure 3 : Main function (calculator.c)

So, this program follows the fetch-decode-excute cycle flow of the von Neumann architecture. Figure 2 shows the flow of the calculator program. In Figure 3, the calculator program has five phases of executing the program: load – fetch – decode – execute - print. First, in load, the program reads instructions one by one from the “input.txt” by using the fgets() function and stores those instructions in the Memory array. At this time, instead of storing a line of command immediately, the string is divided based on the white space. And it has a process of redefining the segmented string.

```
memno = 0;
while (fgets(buffer, sizeof(buffer), fp) != NULL) {
  count = 0;
  trim(buffer);
  ptr = strtok(buffer, " ");   //string tokenize at first token
  InstMemory[memno].operand_types = 0xFFF;
  while(ptr != NULL){
    tmp = (char*)malloc(BUF_SIZE);
    strcpy(tmp,ptr);
    switch (count){
      case 0: InstMemory[memno].opcode = STRING_TO_OPCODE(tmp); break;
      case 1: /*in case 1,2,3(operand), if operand is register, then bind register
pointer */
      case 2: /*else if operand is hex number, then bind malloc func pointer and put
value ,else NULL*/
      case 3: if( tmp[0] == 'r' ){
                if((num = hex_to_dec(tmp+1)) < 10){
                  InstMemory[memno].operand[count-1] = reg+num;
                  InstMemory[memno].operand_types &= (num<<(4 * (3-count))) | ~
(0xF<<(4*(3-count)));
                }
                else{
                  InstMemory[memno].operand[count-1] = NULL;
                  InstMemory[memno].operand_types &= (0xE<<(4 * (3-count))) | ~
(0xF<<(4*(3-count)));
                }
              }
              else if((tmp[0]=='0') && (tmp[1]=='x')){
                InstMemory[memno].operand[count-1] = (uint32_t*)malloc(sizeof(int));
                *(InstMemory[memno].operand[count-1]) = hex_to_dec(tmp+2);
                InstMemory[memno].operand_types &= (0xA << (4 * (3-count))) | ~(0xF<<(4
* (3-count)));
              }
              else if((*((unsigned long*)(tmp))) == STDOUT){
                InstMemory[memno].operand_types &= (0xB << (4 * (3-count))) | ~(0xF<<(4
* (3-count)));
              }
              else{
                InstMemory[memno].operand_types &= (0xE << (4 * (3-count))) | ~(0xF<<(4
* (3-count)));
              }
      default:
          break;
    }
    ptr = strtok(NULL, " ");     //next token
    free(tmp);
    count++;
  }
```

```
  if(integrity_check(InstMemory[memno])){
    printf("Invaild Instruction! \n");
    exit(0);
  }
  memno++;
}
```

figure 4 : load function (load.c)

```
void trim(char* origin){
    int str_len = strlen(origin);
    int i,j;
    char* tmp;
    char* str = (char*)malloc((str_len+1));
    strcpy(str, origin);
    for(i = 0; origin[i] == ' '; i++){}
    for(j = str_len-1; (origin[j] == ' ')||origin[j] == '\n'; j--){}
    tmp = str+i;
    str[j+1]='\0';
    strcpy(origin, tmp);
    free(str);
}
```

figure 5 : trim function (util.c)

```
//#define STRING_TO_OPCODE(opcode) ((*((unsigned int*)(opcode)))&0xffffff)
#define STRING_TO_OPCODE(opcode) ( opcode[0] | (opcode[1] << 8) | (opcode[2] << 16) )
```

figure 6 : STRING_TO_OPCODE macro function (instruction.h)

```
typedef struct _instruction{
    uint32_t* operand[3];
    uint32_t opcode : 24;
    uint16_t operand_types : 12;
}inst;
```

figure 7 : inst structure (instruction.h)

Figure 4 shows the load function. But it looks so complex. Because it has so many feature.

> This is what needs to be improved. There are too many features in one function.
>
> reason : In this project, it seems right to distinguish whether the operand is a register or a immediate in the decode code. But then it seemed that the integrity check should go into the decode code. I tried to solve it, but eventually the load function be complex(The process of knowing what type of operand is for integrity check). Therefore, the load function handled the integrity check. Because the load function performed an integrity check only once for each instruction. However, if the integrity check is performed in decode, the integrity test for the same command occurs several times due to jmp, bne, and beq. But there seems to be a better way than this.

Load function has six phase : file_open - get_string - trim - tokenize - instruction_redefine - store.

1. file_open : Open a file using fopen()
2. get_string : Get a single line of string from a file
3. trim : Remove front and back spaces of a string
4. tokenize : string divide based on the white space
5. instruction_redefine : Redefine a string as an integer or pointer
    1. if count == 0 (first token == opcode) then change string to int value (Show figure 6)

        > `#define STRING_TO_OPCODE(opcode) ((*((unsigned int*)(opcode)))&0xffffff)` is origin macro function. However, considering the endianness, the result value of explicit type conversion may be different, so fixed it.

    2. elif count == 1 or 2 or 3 (second, third, fourth token == operand)
        1. if token start 'r' then operand[count-1] == register pointer. and optype is 0x(register number)
        2. if token start '0' , and next 'x' then operand[count-1] == malloc(sizeof(int)) and chang string(hex) to int , store it. And optype is 0xA
        3. if token is "STDOUT" then optype is 0xB
        4. else optype is 0xE
6. store - Save the opcode,operands,optypes in the instruction structure.

```
/*
Optypes = 0x ? ? ?
             | | └ operand2 types
             | └ operand1 types
             └ operand0 types
operand type
0~9 regiseter = 0~9 hex value
immediate value = 0xA value
STDOUT = 0xB
NONE = 0xF
Error = 0xE
```

```
ex.
Optypes = 0x 2 0 1
             | | └ operand2 types == register[1]
             | └ operand1 types == register[0]
             └ operand0 types == register[2]

Optypes = 0x 0 A F
             | | └ operand2 types == None
             | └ operand1 types == immediate
             └ operand0 types == register[0]

Optypes = 0x 1 B F
             | | └ operand2 types == None
             | └ operand1 types == STDOUT
             └ operand0 types == register[1]
*/
int integrity_check(inst ins){
    int op_types = ins.operand_types;
    int op[3] = { (( op_types>> (8)) & 0xF) , (( op_types>> (4)) & 0xF), op_types &
0xF};

    switch (ins.opcode){
        // op1: r,  op2: r, op3: r
        case ADD:
        case SUB:
        case MUL:
        case DIV:   for(int i = 0; i < 3; i++)
                        if(op[i] > 0x9)
                            return 1;
                    break;

        // op1: r/n,  op2: r/n, op3: n
        case BEQ:
        case BNE:   if(op[0] > 0xA || op[1] > 0xA || op[2] != 0xA) return 1;
                    break;

        //op1: r,  op2: r/n, op3: n
        case SLT:   if(op[0] > 0x9 || op[1] > 0xA || op[2] > 0xA) return 1;
                    break;
        //op1: r,  op2: r/n, op3: X
        case MOV:   if(op[0] > 0x9 || op[1] > 0xA || op[2] != 0xF) return 1;
                    break;
        //op1: r,  op2: n, op3: X
        case  LW:   if(op[0] > 0x9 || op[1] != 0xA || op[2] != 0xF) return 1;
                    break;
        //op1: r,  op2: STDOUT, op3: X
        case  SW:   if(op[0] > 0x9 || op[1] != 0xB || op[2] != 0xF) return 1;
                    break;
        //op1: n,  op2: X, op3: X
```

```
        case JMP:   if(op[0] != 0xA || op[1] != 0xF || op[2] != 0xF) return 1;
                    break;
        //op1: X,  op2: X, op3: X
        case RST:   if(op[0] != 0xF || op[1] != 0xF || op[2] != 0xF) return 1;
                    break;

        default:
            break;
    }
    return 0;
}
```

figure 8 : integrity_check function (integrity_check.c)

```
void init(void){
    opcode      = 0x000000;
    pc          = 0;
    op_types    = 0xFFF;
    pflag       = 0b00000000000;
    op1         = NULL;
    op2         = NULL;
    op3         = NULL;
}
```

figure 9 : init function (init.c)

if finish load() then call init()

it just initializes value

```
void fetch(void){
    IR = InstMemory[pc];
    pc = pc + 1;
}
```

figure 10 : fetch function (fetch.c)

finish init() then call fetch() with while(memory[pc].opcode != 0)

Get instruction from memory and stored in IR. and increase PC value.

```
void decode(void){
    opcode = IR.opcode;
    op1 = IR.operand[0];
    op2 = IR.operand[1];
    op3 = IR.operand[2];
    op_types =IR.operand_types;
}
```

figure 11 : decode function (decode.c)

finish fetch() then call decode()

Get opcode and operands, optypes from IR register and store it.

```
void execute(void){
  switch (opcode){
  case ADD: *op1 = *op2 + *op3;           pflag = pflag | get_reg_index(op_types);
                                                                          break;
  case SUB: *op1 = *op2 - *op3;           pflag = pflag | get_reg_index(op_types);
                                                                          break;
  case MUL: *op1 = *op2 * *op3;           pflag = pflag | get_reg_index(op_types);
                                                                          break;
  case DIV: *op1 = *op2 / *op3;           pflag = pflag | get_reg_index(op_types);
                                                                          break;
  case SLT: *op1 = *op2 < *op3 ? 1 : 0;   pflag = pflag | get_reg_index(op_types);
                                                                          break;
  case MOV: *op1 = *op2;                  pflag = pflag | get_reg_index(op_types);
                                                                          break;
  case  LW: *op1 = *op2;                  pflag = pflag | get_reg_index(op_types);
                                                                          break;
  case  SW: printf("STDOUT : %d\t|",*op1);pflag = pflag | 0b10000000000;
                                                                          break;
  case RST: for(int i = 0; i < 10; i++) reg[i] = 0;       pflag = 0b00000000000;
                                                                          break;
  case JMP: pc = *op1 - 1; break;
  case BEQ: pc = *op1 == *op2 ? *op3 - 1 : pc; break;
  case BNE: pc = *op1 != *op2 ? *op3 - 1 : pc; break;

  default: break;
  }
}
```

figure 12 : execute function (execute.c)

Opcode	Operand (op1 | op2 | op3)	Operatrion
ADD, SUB, MUL, DIV	R | R | R	Basic binary arithmetic operations **op1=op2 operate op3**
MOV	R | R/N	Move value **op1 = op2**
LW	R | N	Load immediate value **op1=op2**
SW	R | STDOUT	Print out **op1**
RST		**reset** all register
BEQ	R/N | R/N | N	Branch Equal then **op1 == op2 then pc = op3**
BNE	R/N | R/N | N	Branch Not Equal then**op1 != op2 then pc = op3**
SLT	R |R/N | R/N	Set less than**op1 = op2 < op3 ? 1 : 0**
JMP	N	Jump **pc = op1**

figure 13 : ISA requirements

finish decode() then call execute()

In the execute(), the operation is performed according to the ISA requirements shown in figure 13. By the finished load function, register operands update result value through a pointer. Therefore, it does not consider updating the register value.

And if register changed then update pflag.

pflag's 0~9 bits means register number. if pflag is 0b0000001011 then register[0] , register[1] and register[3] are changed register.

pflag's 10bit means STDOUT.

```
void print_register(void){
    if((pflag>>10) == 1){
        pflag = pflag & 0b01111111111;
    }
    else{
        printf("\t\t|");
    }

    if(pflag){
        for(int i = 0; i < 10; i++){
            if(((pflag>>i) & 0b1)==0b1)
                printf(" reg[%d]=%d",i,reg[i]);
```

```
        }

    }
    printf("\n");
}
```

figure 14 : print_register function (cal_io.c)

print_regiset() print out register value according to the pflag. If pflag's STDOUT(10bit) is 0 then align the output.

Build Environment and Results

Ubuntu

> Ubuntu 22.04.2 LTS (GNU/Linux 5.15.0-69-generic aarch64)
>
> gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0

M2 MacBook Air

> Apple clang version 14.0.0 (clang-1400.0.29.202)
>
> Target: arm64-apple-darwin22.3.0
>
> Thread model: posix

Make command

> make main -> build the execution program
>
> make clean -> clean the object files

- File and make

```
root@ubuntu:~/share/CA# ls -la
total 68
drwxr-xr-x 19 501 dialout  608 Apr  6 17:40 .
drwxrwxrwx 20 501 dialout  640 Apr  6 17:39 ..
-rw-r--r--  1 501 dialout  417 Apr  6 17:39 calculator.c
-rw-r--r--  1 501 dialout  554 Apr  6 17:04 calculator.h
-rw-r--r--  1 501 dialout  605 Apr  6 17:01 cal_io.c
-rw-r--r--  1 501 dialout  310 Apr  6 17:02 decode.c
-rw-r--r--  1 501 dialout 1279 Apr  6 17:02 execute.c
-rw-r--r--  1 501 dialout   87 Apr  6 17:02 fetch.c
-rw-r--r--  1 501 dialout  224 Apr  6 17:02 init.c
-rw-r--r--  1 501 dialout  114 Apr  6 17:19 input1.txt
-rw-r--r--  1 501 dialout   76 Apr  6 13:05 input2.txt
-rw-r--r--  1 501 dialout   76 Apr  6 17:18 input3.txt
-rw-r--r--  1 501 dialout   90 Apr  6 17:17 input4.txt
-rw-r--r--  1 501 dialout  128 Apr  6 17:40 input.txt
-rw-r--r--  1 501 dialout  577 Apr  6 17:05 instruction.h
-rw-r--r--  1 501 dialout 2482 Apr  6 17:23 intergrity_check.c
-rw-r--r--  1 501 dialout 2728 Apr  6 17:20 load.c
-rw-r--r--  1 501 dialout  732 Apr  5 11:38 Makefile
-rw-r--r--  1 501 dialout  949 Apr  6 17:17 util.c
```

File List

```
root@ubuntu:~/share/CA# make main
gcc -O2 -c -o calculator.o calculator.c
gcc -O2 -c -o load.o load.c
gcc -O2 -c -o fetch.o fetch.c
gcc -O2 -c -o decode.o decode.c
gcc -O2 -c -o execute.o execute.c
gcc -O2 -c -o util.o util.c
gcc -O2 -c -o cal_io.o cal_io.c
gcc -O2 -c -o init.o init.c
gcc -O2 -c -o intergrity_check.o intergrity_check.c
gcc -O2 -o main calculator.o load.o fetch.o decode.o execute.o util.o cal_io.o init.o intergrity_check.o
```

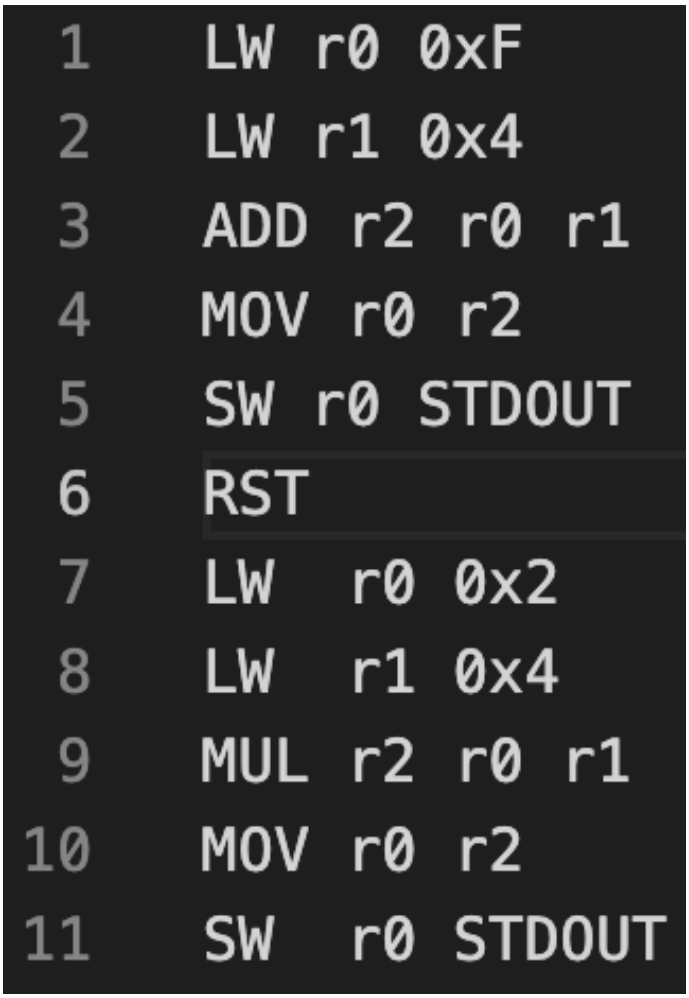
make main

```
root@ubuntu:~/share/CA# make clean
rm *.o main
```

make clean

Result

Basic input



```
 1    LW r0 0xF
 2    LW r1 0x4
 3    ADD r2 r0 r1
 4    MOV r0 r2
 5    SW r0 STDOUT
 6    RST
 7    LW   r0 0x2
 8    LW   r1 0x4
 9    MUL r2 r0 r1
10    MOV r0 r2
11    SW   r0 STDOUT
```

```
root@ubuntu:~/share/CA# ./main
PC : 1          |                | reg[0]=15
PC : 2          |                | reg[0]=15 reg[1]=4
PC : 3          |                | reg[0]=15 reg[1]=4 reg[2]=19
PC : 4          |                | reg[0]=19 reg[1]=4 reg[2]=19
PC : 5          | STDOUT : 19    | reg[0]=19 reg[1]=4 reg[2]=19
PC : 6          |                |
PC : 7          |                | reg[0]=2
PC : 8          |                | reg[0]=2 reg[1]=4
PC : 9          |                | reg[0]=2 reg[1]=4 reg[2]=8
PC : 10         |                | reg[0]=8 reg[1]=4 reg[2]=8
PC : 11         | STDOUT : 8     | reg[0]=8 reg[1]=4 reg[2]=8
root@ubuntu:~/share/CA#
```

JMP input

```
 1    LW r0 0x5
 2    LW r1 0xF
 3    ADD r2 r0 r1
 4    JMP 0x9
 5    RST
 6    LW r0 0x2
 7    LW r1 0x4
 8    MUL r2 r0 r0
 9    ADD r3 r1 r2
10    MOV r0 r3
11    SW  r0 STDOUT
```

```
root@ubuntu:~/share/CA# ./main
PC : 1          |               | reg[0]=5
PC : 2          |               | reg[0]=5 reg[1]=15
PC : 3          |               | reg[0]=5 reg[1]=15 reg[2]=20
PC : 4          |               | reg[0]=5 reg[1]=15 reg[2]=20
PC : 9          |               | reg[0]=5 reg[1]=15 reg[2]=20 reg[3]=35
PC : 10         |               | reg[0]=35 reg[1]=15 reg[2]=20 reg[3]=35
PC : 11         | STDOUT : 35   | reg[0]=35 reg[1]=15 reg[2]=20 reg[3]=35
root@ubuntu:~/share/CA#
```

BEQ input

```
1    LW r0 0x5
2    LW r1 0xA
3    BEQ r0 r1 0x6
4    MOV r0 0x0
5    JMP 0x7
6    MOV r0 0x1
7    SW r0 STDOUT
```

```
root@ubuntu:~/share/CA# ./main
PC : 1          |               | reg[0]=5
PC : 2          |               | reg[0]=5 reg[1]=10
PC : 3          |               | reg[0]=5 reg[1]=10
PC : 4          |               | reg[0]=0 reg[1]=10
PC : 5          |               | reg[0]=0 reg[1]=10
PC : 7          | STDOUT : 0    | reg[0]=0 reg[1]=10
root@ubuntu:~/share/CA#
```

BNE input

```
1    LW r0 0x5
2    LW r1 0xA
3    BNE r0 r1 0x6
4    MOV r0 0x0
5    JMP 0x7
6    MOV r0 0x1
7    SW r0 STDOUT
```

```
root@ubuntu:~/share/CA# ./main
PC : 1          |               | reg[0]=5
PC : 2          |               | reg[0]=5 reg[1]=10
PC : 3          |               | reg[0]=5 reg[1]=10
PC : 6          |               | reg[0]=1 reg[1]=10
PC : 7          | STDOUT : 1    | reg[0]=1 reg[1]=10
root@ubuntu:~/share/CA#
```

SLT input

```
1    LW r0 0x5
2    LW r1 0xA
3    SLT r2 r0 r1
4    BNE r2 0x0 0x7
5    MOV r0 0x0
6    JMP 0x7
7    MOV r0 0x1
8    SW r0 STDOUT
```

```
root@ubuntu:~/share/CA# ./main
PC : 1          |               | reg[0]=5
PC : 2          |               | reg[0]=5 reg[1]=10
PC : 3          |               | reg[0]=5 reg[1]=10 reg[2]=1
PC : 4          |               | reg[0]=5 reg[1]=10 reg[2]=1
PC : 7          |               | reg[0]=1 reg[1]=10 reg[2]=1
PC : 8          | STDOUT : 1    | reg[0]=1 reg[1]=10 reg[2]=1
root@ubuntu:~/share/CA#
```

Lesson

> One of the difficulties was writing a code report. It was harder than I thought to write my thoughts on the code and its grounds.
> It was also difficult to handle strings in C language.
> Strcmp function and if statement were used because the switch case could not be written because the equivalent operation was not possible between strings.
> However, I tried to use the switch case somehow because this method was so bad to look at and so hard to fix.
>
> Also, the exception handling process was harder than I thought.
> In particular, string exception handling has really a lot of cases. Case Upper Lower,white space problems, etc..
> I hard spent time dealing with these errors.
> In particular, the message to "gracefully" handle errors has fallen into the swamp of exception handling.
> Of course, it was a good experience.