Simple Calculator

Using C

Project #1

32181928 박찬호

Dankook University

Mobile Systems Engineering

2022 Spring

Left Freeday : 4

(금번 과제 제출로 freeday 1 사용)

Contents 목차

서론

구현 목표

본 프로젝트의 구현 목표는 다음과 같습니다.

1. 10 개의 레지스터를 지원한다 (R[0] ~ R[9])
2. 더하기 +, 빼기 -, 곱하기 *, 나누기 /, 나머지% 의 기본 수리연산을 지원한다
3. 레지스터 이동 M 연산을 지원한다
4. 점프 J 연산을 지원한다
5. 비교 C 연산을 지원한다. (C A B 를 실행하면 R[0] = A < B ? 1 : 0 로 동작)
6. 브랜치 B 연산을 지원한다. (R[0]이 1 이면 J 연산 실행)
7. 중지 H 연산을 지원한다.

위의 목표를 수행하기 위한 구현 언어는 C 로 설정했습니다. 본 프로젝트에서 개인적인 목표를 설정하자면, C 의 라이브러리와 문법을 제대로 활용하면서 최대한 많은 것을 구현하는 것입니다.

실행 방법

설명에 앞서 본 프로그램의 실행 방법을 설명합니다.

main 파일은 실행 파일로 Ubuntu 20.04 LTS 에서 정상 동작을 확인했습니다.

./main input-gcd.camp --reg 8 123456789 --reg 9 987654321

와 같이 input 파일과 필요한 레지스터의 값을 지정해주면 정상적으로 동작합니다.

(123456789 와 987654321 의 gcd 값을 찾기 위한 인자)

Windows 를 위해서는 전체 파일을 컴파일하여 같은 방식으로 인자 값을 주어 실행하면 됩니다.

본론

구현 방법

본 프로젝트의 핵심 아이디어는 다음과 같습니다.

Memory 와 Register 라는 일종의 모듈을 만든다. Memory 는 Instruction 을 담는 배열과 같은 형태이고, Register 는 Data 를 담는 배열이다.

따라서, 전체 Memory size 는 담을 수 있는 Instruction 개수와 같고, 사용 중인 Memory size 는 load 된 Instruction 개수입니다.

각 요소는 각 헤더 안에서 다음과 같이 정의됩니다.

Instruction

```
typedef struct _instruction {
    int opcode;
    OPERAND* op1;
    OPERAND* op2;
} INSTRUCTION;
```

Memory

```
typedef struct _memory {
    size_t mem_size;
    size_t used_size;
    INSTRUCTION** insts;
} MEMORY;
```

Data

```
typedef size_t DATA;
```

Register

```
DATA registers[10] = { 0, };
```

이 요소들을 이용하여, 전체 프로그램의 흐름은 아래와 같이 진행됩니다.

1. Memory 를 할당하고 초기화한다.
2. 파일에서 Instruction 을 하나씩 읽어 Memory 에 저장한다
3. Instruction 을 실행하는 Operation 함수에 Memory 를 넘긴다.
4. Operation 함수는 전체 Instruction 을 실행한다.
5. Memory 를 할당 해제한다.

세부적으로는

2 의 과정 중에 파일에서 읽은 문자열을 토큰화하여, Opcode 와 피연산자 1, 피연산자 2 로 분리하는 과정

4 의 과정 중에 Memory 와 Register 를 시각화하는 과정 등이 포함되어 있습니다.

용어 정리

이후 사용되는 용어 중 Memory 와 Register 는 실제 컴퓨터의 Memory 와 Register 가 아닌 본 프로그램에서 구현한 Memory 와 Register 를 말합니다.

파일 구성

전체 파일 구성은 크게 6 개의 영역으로 되어 있습니다. 그 영억은 다음과 같습니다

I. main
II. memory
III. operand
IV. operator
V. operation
VI. 기타 영역 (exception, keyboard 등)

본론의 전개는 위에 소개된 순서대로 영역 별 코드를 설명하면서 진행하도록 하겠습니다.

I. Main 영역

main 영역은 프로그램의 entry point 인 main 함수가 존재하며, memory, register 등 본 프로그램에서 사용하는 가상 모듈을 초기화하는 영역입니다. main 영역은 main.c 파일 하나로 구성되어 있습니다.

main 영역은 다음의 순서대로 역할을 수행됩니다.

1. Register 초기화
2. 프로그램 실행 인자 설정
3. Memory 관련 코드 호출
4. 실행 함수 호출

main 은 프로그램의 진입점으로서 역할을 수행하는 것 외에 많은 기능을 수행하지 않고, 다른 파일의 함수를 호출하는 역할입니다. 따라서, main 의 주요한 역할인 실행 인자 설정만 짚고 넘어가겠습니다.

1. Register 초기화

10 개의 레지스터 (R[0] ~ R[9]) 모두는 초기값으로 0 을 가집니다. 그러나 프로그램 실행에 있어 Register 에 초기값을 정해줘야 할 필요가 있는 경우, --reg 라는 인자를 이용해 설정할 수 있습니다.

--reg A B 꼴로 인자를 입력할 경우, R[A] = B 가 실행되어 A 번째 Register 를 원하는 값으로 초기화 할 수 있습니다.

2. 메모리 크기 설정

memory.h 에 초기 memory size 는 30 으로 정해져 있는데, --mem-size 라는 flag 를 사용하면 실행 시의 memory size 를 원하는 값으로 결정할 수 있습니다. (--mem-size 100 과 같은 형식으로 사용)

3. 파일 이름 설정

파일 이름은 별도의 option flag 없이 입력하면 됩니다.

따라서 프로그램을 실행할 때 다음과 같은 형식으로 인자를 입력할 수 있습니다

./main input.txt --mem-size 100 –reg 8 12345 –reg 9 4321

위의 명령은 다음과 같은 의미입니다

현재 위치의 main 이란 실행 파일을 실행하는데, input.txt 를 input file 로 하고, mem size 는 100 으로, 8 번 register 의 값은 12345 로, 9 번 register 의 값은 4321 로 설정

II. Memory 영역

Memory 영역은 memory 와 instruction 의 형식을 설정하고, memory 에 관련된 기능을 구현한 영역입니다.

Memory 영역의 구현을 설명하기 전에 다시 한 번 Memory 와 Instruction 의 형식을 보이고 넘어가겠습니다

Instruction

```
typedef struct _instruction {
    int opcode;
    OPERAND* op1;
    OPERAND* op2;
} INSTRUCTION;
```

Memory

```
typedef struct _memory {
    size_t mem_size;
    size_t used_size;
    INSTRUCTION** insts;
} MEMORY;
```

이 두 구조체는 memory.h 에 정의되어 있습니다.

주목할 점은, 첫째로 MEMORY 는 INSTRUCTION 의 포인터를 원소로 갖는 스택과 사실상 동일한 역할이라는 것, 둘째로 Instruction 은 단순한 문자열이 아닌 opcode 와 두 연산자를 갖는 구조체라는 것입니다

이어서 memory 영역의 함수들을 영역 외부로 제공하는 함수부터 살펴보겠습니다.

먼저 영역 외부로 제공하는 함수는 initialize_memory, load_file, visualize_memory 입니다.

이중 visualize_memory 는 메모리를 시각화하는 역할만 수행하므로 별도의 설명을 생략합니다.

첫째로 initialize_memory 는 memory 와 instruction 배열을 동적할당하는 역할입니다. 이때 int 형의 memsize 를 인자로 받아 초기 memsize 를 설정해줄 수 있습니다. 코드는 동적할당 외의 별다른 내용이 없기에 별도의 코드 설명은 생략하겠습니다.

둘째로 load_file 은 파일을 읽어 Instruction 형태로 변형하여 Memory 에 저장하는 함수입니다.

함수의 형식은 다음과 같습니다.

MEMORY* load_file (MEMORY* memory, char* file_name);

본 함수의 주요 부분은 아래와 같습니다.

```
    int memory_begin = memory->used_size;

    while (feof(fp) == 0)
    {
       if (buffer != fgets(buffer, BUFLEN, fp)) {

       (예외처리 생략)
       }

        buffer[strcspn(buffer, "\r\n")] = 0;

       (예외처리 생략)

        memory->insts[memory->used_size] = to_instruction(buffer);
        memory->used_size += 1;
    }

    fclose(fp);
    return memory;
}
```

예외처리를 하는 부분을 생략하면 주요 부분의 로직은

버퍼에 파일을 한 줄 씩 읽고 -> Instruction 형태로 변환하여 -> 메모리에 추가한다.

입니다.

이때 스트링을 Instruction 으로 변환하는 함수는 to_instruction 이라는 static 함수입니다.

```
static INSTRUCTION* to_instruction(char* buffer) {

    INSTRUCTION* instruction = malloc(sizeof(*instruction));
    (에러 처리 생략)
    char** str_token = malloc(sizeof(*str_token) * MAX_NUM_TOKEN);
    (에러 처리 생략)

    for (int i = 1; i < 3; i++) str_token[i] = NULL;

    char* ptr;
    int num_of_tokens;

    for (int i = 0, ptr = strtok(buffer, " "); ptr != NULL; num_of_tokens =
i++, ptr = strtok(NULL, " ")) str_token[i] = ptr;

    instruction->opcode = encode_token(*str_token[0]);

    OPERAND* operands[2];

    for (int i = 1; i <= 2; i++) {

        if (str_token[i] == NULL) {
            operands[i - 1] = NULL;
            break;
        }
        operands[i - 1] = malloc(sizeof(*operands[i - 1]));
        operands[i - 1] = operand_initializer(str_token[i]);
    }

    instruction->op1 = operands[0];
    instruction->op2 = operands[1];

    return instruction;
}
```

이 함수의 로직은 다음과 같습니다. Instruction 을 할당하고, 문자열 token 을 저장할 문자열 배열 str_token 을 만듭니다.

이후

```
    for (int i = 0, ptr = strtok(buffer, " "); ptr != NULL; num_of_tokens =
i++, ptr = strtok(NULL, " ")) str_token[i] = ptr;
```

이 부분 코드에서 한 줄의 문자열의 token 들로 쪼개어 str_token 배열에 저장하고,

opcode 는 operation 영역의 encode_token 이 변환해주고

```
    instruction->opcode = encode_token(*str_token[0]);
```

피연산자는 operand 영역의 operand_initializer 가 변환하여 초기화해줍니다.

```
    for (int i = 1; i <= 2; i++) {

        if (str_token[i] == NULL) {
            operands[i - 1] = NULL;
            break;
        }
        operands[i - 1] = malloc(sizeof(*operands[i - 1]));
        operands[i - 1] = operand_initializer(str_token[i]);
    }
```

III. Operand 영역

Operand 영역은 operand 의 형식을 설정하고, operand 와 관련된 기능을 구현한 영역입니다.

Operand 영역의 구현을 설명하기 전에 다시 한 번 Operand 의 형식을 보이겠습니다.

```
typedef enum {
    NONE = -1,
    REGISTER = 'R',
    HEXA = '0'
} TYPE;

typedef struct {

    TYPE type;
    DATA value;

} OPERAND;
```

위와 같이 OPERAND 는 Register, Hexa 와 같이 타입을 갖고 타입에 맞는 값을 가집니다. 타입이 Register 라면 value 는 Register 의 번호, HEXA 라면 정수 값입니다. 각 타입의 enum 정수 값으로는 문자를 할당하여 파일 입력 시 읽어 들이는 문자를 그대로 사용할 수 있도록 했습니다.

OPERAND 는 그 자체로 사용하지 않고, OPERAND 를 초기화할 때는 operand_initializer 함수를 사용하고, OPERAND 가 의도하는 값은 value_by_type 으로 찾아 사용합니다. 이는 Register 타입을 고려한 것으로, 실제 명령어 수행 시에 Register 타입의 OPERAND 는 Register[value]에 들어있는 값을 사용해야 합니다. 이러한 역할을 value_by_type 이 수행합니다.

Operand 영역은 단순한 코드로 이루어져 있으므로 별도의 코드 설명을 생략합니다.

IV. Operator 영역

Operator 영역은 operator 의 형식을 설정하고, operator 와 관련된 기능을 구현한 영역입니다.

이 영역의 주요 목표는 연산자 실행 시 switch 를 통한 연산자 실행을 피하는 것에 있습니다. 현재 제공되는 연산자는 총 10 개인데, 10 개의 연산을 모두 switch 로 처리할 경우 코드가 난잡해집니다. 또한, 추후 다른 연산자를 추가할 경우 switch 문 내에 case 를 추가하며 더욱 난잡한 코드를 만들게 됩니다.

따라서, 본 영역에서는 각 연산자에 opcode 를 부여하여, opcode 를 이용하면 직접적으로 함수를 실행할 수 있게 했습니다. 이를 구현하기 위한 방법으로는 함수 포인터를 사용하였습니다.

구현 방법은 다음과 같습니다

먼저 전역 상수로 연산자를 담는 상수 배열을 만듭니다. 이는 다른 영역에서도 필요 시 extern 으로 가져다 쓸 수 있게 하기 위함입니다.

```
const char operators[] = { '+', '-', '*', '/', '%', 'M', 'C', 'J', 'B', 'H',
'\0' };
```

파일을 로드하여 Instruction 으로 변환할 때, encode_token 은 위의 배열에서 연산자를 찾아 그 인덱스를 opcode 로 반환합니다.

```
int encode_token(char token) {
    return strchr(operators, token) - operators;
}
```

이렇게 Instruction 은 opcode 를 가지고 있고, 이후 instruction 실행 시에 opcode 로 함수를 호출하게 됩니다.

이를 위해 전체 연산자들에 해당하는 별도의 함수들을 만들어야 하는데, 이 함수들은 피연산자의 개수가 다릅니다. +, -, *, /, %, M, C 는 2 개의 피연산자를 필요로 하고, J, B 는 1 개의 피연산자를, H 는 0 개의 피연산자를 필요로 합니다.

이때 H 는 Operation 과정을 종료하기만 하면 되므로, 별도의 함수를 만들지 않고, 이외의 연산들은 모두 인자의 형식을 통일하여 함수를 생성했습니다.

그 형식은 (OPERAND*, void*, DATA*) 입니다. 각 인자 별로 설명하겠습니다.

먼저 H 를 제외했으므로 모든 연산자는 공통적으로 첫 번째 피연산자를 받기 때문에 첫 번째 인자는 OPERAND 포인터입니다.

두 번째 인자는 void* 입니다. 먼저 두 개의 피연산자를 갖는 연산자는 void*를 OPERAND*로 캐스팅하여 사용하면 되고, 한 개의 피연산자를 갖는 연산자 J 와 B 는 pc (program counter)를 수정해야 하므로, void* 자리에 int 타입인 pc 의 주소를 받아 int*로 캐스팅하여 사용하도록 했습니다.

세 번째 인자는 register 를 사용하는 경우가 많으므로, register 배열을 받도록 했습니다.

둘의 예시로 add 와 jmp 의 구현을 보이겠습니다.

```
void add(OPERAND* operand1, void* operand2, DATA* registers) {

    DATA op1 = value_by_type(operand1, registers);

    DATA op2 = value_by_type((OPERAND*)operand2, registers);

    registers[0] = op1 + op2;

    return;
}
```

```
void jmp(OPERAND* op1, void* pc, DATA* registers) {

    int* program_counter = (int*)pc;
    *program_counter = op1->value;
    (*program_counter)--;
}
```

이렇게 구현하면 jmp 와 같이 register 를 사용하지 않는 함수는 불필요한 값을 인자로 받는다는 단점이 있지만, jmp 를 제외하면 브랜치 기능을 포함하여 모든 함수가 register 를 사용하므로 형식이 통일된다는 장점이 있습니다.

또한 형식이 통일되었기 때문에, 앞서 소개한 것과 같이 함수 포인터를 사용하면 전체 연산자를 함수 포인터 배열에 다 담을 수 있습니다. 이는 operation 영역으로 넘어가 바로 설명하겠습니다.

V. Operation 영역

Operation 영역은 operation 이라는 함수가 주요 부분입니다. 앞서 설명한 흐름을 이어가기 위해 바로 코드 설명으로 넘어가겠습니다.

Operation 함수는 처음에 함수 포인터 배열을 만듭니다. 그리고 앞서 설명한 바와 같이 해당 배열 내에 operators 의 인덱스와 일치하게 만들어 둔 연산자 함수를 할당합니다.

```
void operation(MEMORY* memory, DATA* registers)
{
    void (*fp[9])(OPERAND*, void*, DATA*);

    fp[0] = &add;
    fp[1] = &sub;
    fp[2] = &mul;
    fp[3] = &quo;
    fp[4] = &mod;
    fp[5] = &mov;
    fp[6] = &cmp;
    fp[7] = &jmp;
    fp[8] = &brn;
```

이와 같은 방식을 통해 opcode 를 이용하여 직접 연산자를 실행할 수 있게 되고, switch-case 의 반복 사용으로 난잡한 코드를 피할 수 있게 됩니다.

Operation 안에서는 for 문으로 pc 값이 0 부터 증가하며

```
    for (int pc = 0; pc < size; pc++) {

        (중간 생략)
        int opcode = memory->insts[pc]->opcode;
        if (opcode == 9) break;
        OPERAND* operand1 = memory->insts[pc]->op1;
        void* operand2 = opcode < 7 ? memory->insts[pc]->op2 : &pc;
        fp[opcode](operand1, operand2, registers);
        (생략)
    }
}
```

각 instruction 의 opcode 와 피연산자를 읽어들여

```
fp[opcode](operand1, operand2, registers);
```

와 같은 식으로 연산을 실행하게 됩니다.

VI. 기타 영역

기타 영역에는 exception.c exception.h keyboard.c keyboard.h color.h data.h 가 있습니다.

Exception 은 메시지를 받아 stderr 로 출력하는 print_exception 함수가 있습니다

Keyboard 는 키보드로 입력 받은 키를 알려주는 getch 함수가 있으며, 이는 Windows 와 달리 Linux 계열에서는 제공해주지 않는 함수입니다. 따라서 널리 알려진 Linux 용 getch 코드를 사용했습니다.

Color 는 fprintf 에서 글자의 색깔을 define 하였고, data 는 typedef 로 data 의 자료형을 정했습니다.

결론

느낀 점

최초에 목표했던 것처럼 실제 잘 쓰여진 C 코드, 예컨대 coreutils 나 linux kernel 에서 C 를 사용하는 것처럼 좋은 코드를 짤 수는 없었습니다. 그러나 switch-case 문제를 해결하거나, 문자열을 토큰화하는 등의 일을 수행하며 더 나은 C 코드를 작성할 수 있게 되었습니다. 또한, mips 의 green sheet 를 참고하며 전체 프로그램의 방향을 설계하는 과정에서 조금 잊고 있었던 mips 어셈블리의 동작 방식도 다시금 체화할 수 있는 계기가 되었습니다.

아래는 동작 사진입니다.