Single-cycle MIPS

Using C

Project #2

32181928 박찬호

Dankook University

Mobile Systems Engineering

2022 Spring

Left Freeday : 0

(금번 과제 제출로 남은 freeday 전부 사용)

서론

구현 목표

본 프로젝트의 구현 목표는 다음과 같습니다.

MIPS Simulator 를 구현할 것

세부 사항으로는

1. MIPS binary 를 읽어서 실행할 것
2. MIPS instruction 을 지원할 것
3. 32 개의 레지스터를 지원할 것
4. 5 개의 stage 를 통해 instruction 을 실행할 것
5. 최종 결과를 v0 (r2) 레지스터에 저장할 것
6. 샘플 프로그램을 모두 실행할 것

구현 언어는 C 로 설정하였으며, 본 프로젝트는 프로젝트 1 의 구성을 유지하며 개발하는 것을 목적으로 하였습니다. 따라서, 본 프로젝트에서는 4 번의 요구사항의 5 개 instruction stages(IF, ID, Ex, M, WB)의 단계를 명시적으로 나누지 않았습니다. 4 번 요구사항은 프로젝트 3 에 이르면 충분히 구현될 것입니다. 위와 관련된 내용은 본론 첫 부분에서 더 자세히 설명합니다

실행 방법

설명에 앞서 본 프로그램의 실행 방법을 설명합니다. out 파일은 실행 파일로

./out input4.bin

와 같이 input 이 되는 바이너리 파일을 지정해주면 정상적으로 동작합니다. 본 프로젝트는 assam 에서의 동작을 기준으로 하나, 실행 환경에 의존적인 코드를 최대한 제거하였으므로, 전체 소스를 Windows 10, Visual Studio 에서 실행해도 정상적으로 동작합니다 (인자 및 몇몇 함수 수정 필요)

본론

구현 방법

본 프로젝트의 핵심 아이디어는 다음과 같습니다.

Project 1 의 함수 포인터 배열 사용을 유지하면서, 전체 Instruction 을 타입별로 나누어 구조체를 설계하고, 이때 메모리 낭비를 막기 위해 비트 필드를 사용합니다. 이를 통해 전체 명령어를 종류별로 처리할 때 if 나 switch 를 남발하지 않을 수 있습니다. 이는 즉, instruction 과 데이터의 형태를 일관되게 구성하여 전체 프로그램이 if else 간에서 혼란스럽게 오가지 않고, 하나의 흐름 속에서 돌도록 합니다. 다만 이러한 구조를 유지하기 위해 부득이 Instruction Stage 를 명시적으로 나누지는 않았습니다. 그러나 추후 멀티사이클에서는 각 Stage 를 명시적으로 구분하고, 동시에 지금까지의 구현 컨셉을 곁들여볼 예정입니다.

이와 같이 로직의 일관성을 유지하는 주요한 구조체를 먼저 보이겠습니다.

Instruction

```
typedef struct {
    unsigned int opcode : 6;
    unsigned int rs     : 5;
    unsigned int rt     : 5;
    unsigned int rd     : 5;
    unsigned int shamt  : 5;
    unsigned int funct  : 6;
} R_INSTRUCTION;

typedef struct {
    unsigned int opcode : 6;
    unsigned int rs     : 5;
    unsigned int rt     : 5;
             int imm    : 16;
} I_INSTRUCTION;

typedef struct {
    unsigned int opcode : 6;
    unsigned int address: 26;
} J_INSTRUCTION;

typedef union {
    R_INSTRUCTION r;
    I_INSTRUCTION i;
    J_INSTRUCTION j;
} INSTRUCTION;
```

Instruction 은 위와 같은 구조체로 구성하였습니다. 각 Instrucion 의 전체 크기가 (unsigned) int 와 같은 4 Byte (32 bit)라는 점에서 착안하여, unsigned int 크기의 구조체에 bit field 를 이용하여 타입에 맞게 각 구성요소 (opcode, rt 등)을 담았습니다. Instruction 은 각 타입 별로 다른 구조체를 가지나, 이는 3 가지 타입의 Instruction 중 무조건 하나로 결정되는 것이므로, 이를 Union 으로 묶어 결정하도록 하였습니다.

본래 구상했던 컨셉은 Int 자료형으로 instruction 을 받고, 이를 통째로 구조체 포인터가 가리키는 주소에 복사하여 별도의 변환없이 구조체의 bit field 에 각 요소가 담기도록 하는 것이었습니다. 그러나 이는 Big Endian – Little Endian 의 차이와 메모리 상의 구조체 배치에 따라 side effect 를 만들어 낼 수 있다고 생각되었고, 이에 결과적으로는 한 번의 변환과정을 거치도록 하였습니다. 이는 decode 에서 진행됩니다.

또한 프로젝트 1 과 마찬가지로 함수 포인터의 배열을 사용하여 execution 을 처리하였습니다

```
void execute(INSTRUCTION* instruction, int registers[], int* pc, int memory[], int
opcode);

void ADDIU  (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void ADDU   (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void BEQ    (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void BNE    (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void JUMP   (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void JAL    (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void JR     (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void LUI    (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void LW     (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void ORI    (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void SLT    (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void SLTI   (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void SLL    (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void SW     (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
void SUBU   (INSTRUCTION* instruction, int registers[], int* pc, int memory[]);
```

이때 모든 연산에 순차적으로 opcode 를 부여할 수 있었던 프로젝트 1 과 달리, MIPS 에서는 연산별 opcode 가 정해져 있고 순차적이지도 않습니다. 따라서, 본 프로젝트에서는 dictionary 와 같이 동작하는 key 배열, value 배열을 이용하여 opcode 로 함수 포인터 배열을 조작할 수 있도록 하였습니다. 이는 추후 코드 설명에서 더 자세히 보이겠습니다.

파일 구성

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

I. main
II. decode
III. execute
IV. mips
V. 기타 영역 (mode, instruction 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 모듈을 초기화하는 영역입니다. main 영역은 main.c 파일 하나로 구성되어 있습니다.

main 영역은 다음의 순서대로 역할을 수행합니다.

1. 바이너리 파일 오픈
2. Register 배열 초기화
3. Memory 배열 초기화
4. File 을 Memory 로 Load
5. Pc 를 증가시켜 가며 한 cycle 씩 실행

main 은 프로그램의 진입점으로서 역할을 수행하며, 동시에 전체 cycle 을 관리합니다. 각 역할을 간단히 설명하겠습니다

1. 바이너리 파일 오픈

파일명은 argv[1]을 통해서 받고, rb 를 통해 바이너리 파일로 오픈합니다.

2. Register 배열 초기화

Register 배열은 길이를 32 로 선언하고, initialize_registers 함수로 초기화합니다. 이 함수는 registers 의 값을 조건에 맞게 초기화합니다 (LR, SP)

3. Memory 배열 초기화

Memory 배열은 힙 영역에 큰 공간을 (100000000 * sizeof(int)) 동적할당하는 것으로 선언하고, memset 으로 -1 을 전체 영역에 채워줍니다. 이는 혹시 모를 상황으로 인해 Memory 배열에 존재하는 모든 명령어를 실행한 이후의 시점을 확인하고, side effect 를 막기 위함입니다.

4. File 을 Memory 로 Load

File 을 Memory 로 load 하는 과정에서, Big Endian 으로 저장되어 있는 sample program 의 binary 를 현재의 little endian 실행환경에 맞게 바꾸는 과정이 필요합니다. 이는 수업 중 교수님이 제시해주신 아래 코드를 바탕으로 변환합니다.

```
int ret;
        char buffer[4];
        for (int i = 0; i < 4; i++) {
            ret = fread(&buffer[3 - i], 1, 1, fp);
        }
```

이렇게 변환한 코드는 int 크기로 묶어 memory 에 저장합니다.

```
        int input = *(int*)buffer;
        *mem_cursor = input;
```

5. pc 를 증가시켜 가며 한 cycle 씩 실행

이 부분이 main 의 핵심적인 부분입니다.

```
int pc = 0;

    char* const pc_cursor = (char*) memory;

    int count = 0;

    while (pc != -1) {
        clrscr();

        view_registers(registers);

        fprintf(stdout, "\n\n\n");

        //FETCH

        int temp_pc = pc;

        int inst = *(int*)(pc_cursor + pc);

        if (inst == -1) break;

#ifdef __DEBUG_MODE__
        printf("\n%#X\n\n", inst);
#endif

        if (inst == 0) {
            count++;
            pc += 4;
            continue;
        }

        int opcode;

        // DECODE

        INSTRUCTION* instruction = decode(inst, &opcode);

        // EXECUTE

        execute(instruction, registers, &pc, memory, opcode);

        free(instruction);

        if (temp_pc == pc) pc += 4;

        count++;
    }

    printf("\n\nR2 : %d  |  Count : %d", registers[2], count);
```

이 부분을 구현할 때, FETCH, DECODE, EXECUTE 의 세 과정으로 나누어, EXECUTE 이후의 MEMORY ACCESS 나 WRITE BACK 과정은 모두 함수 내에서 처리되도록 묶어서 구현하였습니다.

더 자세히 살펴보면, 각 사이클마다 pc_cursor 가 가리키는 memory 의 첫번째 명령어 주소에 pc 라는 값을 더하는 포인터 연산으로 pc 값을 활용하고, 각 루프가 끝날 때는 pc 값이 변하지 않았다면, 즉, PC 를 변화시키는 명령어(Jump, BEQ 등)가 사용되지 않았다면 pc + 4 를 통해 다음 명령어를 읽어들일 준비를 합니다.

루프 내부에서는 memory 에서 fetch 한 instruction 을 decode 하고, execute 에 전달하는 과정을 수행합니다.

II. Decode 영역

Decode 영역은 bit mask 와 shift 를 이용하여 각 instruction 의 type 에 맞게 decode 함수가 변환 과정을 수행합니다. 이를 위해

```
int shift_length[] = {
    26, 21, 16, 11, 6
};

int mask[] = {
    0b11111100000000000000000000000000,
    0b00000011111000000000000000000000,
    0b00000000000011111000000000000000,
    0b00000000000000000111110000000000,
    0b00000000000000000000011111000000,
    0b00000000000000000000000000111111
    };
```

Shift 에 필요한 length 와 bit mask 를 미리 준비하고, 이를 조합하여 instruction 의 각 bit 를 읽어들입니다. 이때 각 type 을 판별하기 위해, opcode_to_inst_type 함수가 key 배열과 value 배열을 이용하여 type 을 반환합니다. 이때 사용된 key 배열은 아래와 같습니다.

```
int op_dict_keys[] = {
    0x0, 0x2, 0x3, 0x4, 0x5, 0x8, 0x9, 0xa,
    0xb, 0xc, 0xd, 0xf, 0x23, 0x24, 0x25, 0x28,
    0x29, 0x2b, 0x30, 0x38
};
```

이 배열은 MIPS 에서 주어진 opcode 를 오름차순으로 담고 있으며, 추후 함수 포인터 배열 접근 시에도 동일하게 Key 로 사용됩니다.

전체 코드에서 R Type 의 decode 과정을 예시로 보이겠습니다.

```
case 'R':
        instruction->r.opcode   = opcode;
        instruction->r.rs       = (input & mask[1]) >> shift_length[1];
        instruction->r.rt       = (input & mask[2]) >> shift_length[2];
        instruction->r.rd       = (input & mask[3]) >> shift_length[3];
        instruction->r.shamt    = (input & mask[4]) >> shift_length[4];
        instruction->r.funct    = (input & mask[5]);

        #ifdef __DEBUG_MODE__
        printf("%#x %#x %#x %#x %#x %#x\n", instruction->r.opcode, instruction->r.rs,
instruction->r.rt, instruction->r.rd, instruction->r.shamt, instruction->r.funct);
        #endif

        break;
```

위와 같이 bit mask 와 shift 를 이용하여 decode 를 진행하며, I type 의 imm 과 같이 주어진 mask 보다 길이가 긴 경우에는 아래와 같이 조합하여 사용합니다.

```
instruction->i.imm      = input & (mask[3] + mask[4] + mask[5]);
```

이렇게 decode 를 마친 후, decode 함수는 INSTRUCTION 구조체 포인터를 반환합니다.

III. Execute 영역

Execute 영역은 각 instruction 을 opcode 에 맞게 실행하기 위해, 함수를 정의하고, 함수 포인터 배열을 준비하는 역할, 그리고 실제 execute 하는 역할까지 수행합니다.

이때, 함수 포인터 배열의 사용에는 한 가지 걸림돌이 존재하는데, 그것은 바로 모든 instruction 이 opcode 하나 만으로 결정되지 않는다는 것입니다. opcode 외에 funct 로 결정되는 instruction 이 존재하기 때문에 opcode 를 이용한 배열 만으로는 모든 instruction 에 접근할 수가 없습니다. 따라서, 최악의 경우에는 opcode 와 funct 를 모두 이용하여 함수 포인터 이중 배열 따위의 것을 만들어야 할지도 모릅니다.

다행히 본 프로젝트에서는 funct 로 접근되는 함수 포인터 배열을 추가로 만들고, 이 instruction 들(R type)을 다시금 execute_r 을 통해 실행하도록 하여 문제를 해결했습니다.

각 instruction 을 표현한 함수는 아래와 같은 꼴을 가집니다

```
void ADDIU(INSTRUCTION* inst, int registers[], int* pc, int memory[]) {
    registers[inst->i.rt] = registers[inst->i.rs] + inst->i.imm;
    return;
}
```

이런 식으로 정의된 함수들을 아래와 같이 opcode, funct 에 대한 함수 포인터 배열로 만들고

```
void (*op_inst[NUM_OF_OP])(INSTRUCTION*, int*, int*, int*) = {
    &execute_r, &JUMP,  &JAL,   &BEQ,   &BNE,   &NULLF, &ADDIU, &SLTI,
    &NULLF,     &NULLF, &ORI,   &LUI,   &LW,    &NULLF, &NULLF, &NULLF,
    &NULLF,     &SW,    &NULLF, &NULLF
};

void (*funct_inst[NUM_OF_FUNCT])(INSTRUCTION*, int*, int*, int*) = {
    &SLL,   &NULLF, &JR,    &NULLF, &ADDU,  &NULLF,
    &SUBU,  &NULLF, &NULLF, &NULLF, &SLT,   &NULLF
};
```

(이때 NULLF 는 형식만 동일한 빈 함수입니다. 본 프로그램에서 사용되지 않는 instruction 의 자리를 대신해주는 역할을 수행합니다.)

```
void execute(INSTRUCTION* inst, int registers[], int* pc, int memory[], int opcode) {

    int key = find_opcode_index(opcode);

    if (key == -1) {
        printf("ERROR with opcode : %#x\n", opcode);
        printf("%d", *pc);
    }

    op_inst[key](inst, registers, pc, memory);
}


void execute_r(INSTRUCTION* inst, int registers[], int* pc, int memory[]) {
    int key = find_funct_index(inst->r.funct);

    if (key == -1) {
        printf("ERROR with funct : %#x\n", inst->r.funct);
        printf("%d", *pc);
    }

    funct_inst[key](inst, registers, pc, memory);
}
```

위의 두 함수를 이용하여 실행합니다. 이를 통해 if-else 와 switch-case 의 무분별한 반복을 피할 수 있습니다.

execute.h 에서는 위의 함수들의 prototype 을 선언함과 동시에, jump address, zero immediate 등의 계산에 필요한 플래그를 담고 있습니다

```
#define JUMP_ADDRESS_FLAG ((int)0b11110000000000000000000000000000)
#define ZERO_IMMIDIATE_FLAG 0b1111111111111111
```

IV. MIPS 영역

MIPS 영역은 이 외에 부가적으로 필요한 함수들, 즉 opcode 를 index 로 변환하거나, register 를 초기화 하는 등 간단하지만 반복 사용되는 함수를 담고 있습니다. 특별한 아이디어나 로직을 담고 있지 않은 부분이라 별도의 설명은 생략하겠습니다.

V. 기타 영역

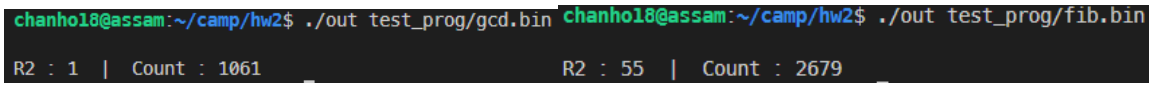
기타 영역에는 mode.h, instruction.h 등이 있습니다.

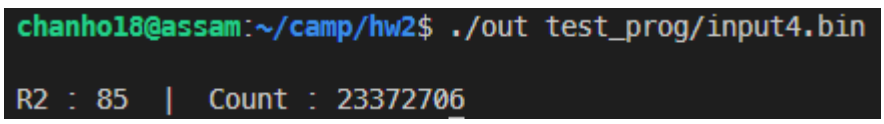
Mode 영역에서는 debug mode 를 끄고 켬으로써 코드 실행 시에 output stream 으로 내용이 출력되는 것을 끄고 켤 수 있습니다.

Instruction 영역에서는 본 보고서의 초반 부에 제시한 것과 같이 instruction 의 형태를 정의합니다.

실행 결과

본 프로그램의 실행 결과는 debug mode 를 켰을 경우 전체 instruction 을 하나씩 실행하며 확인할 수 있습니다. 본 보고서에서는 각 프로그램의 r2 value 와 cycle 횟수만을 캡쳐로 제시합니다. 대표적으로 gcd, fib, input4 세개의 캡쳐 만을 첨부하였습니다.





결론

느낀 점

최초에 프로젝트를 시작할 시점에는 함수 포인터 배열 등의 구현 상의 목표와, 싱글 사이클의 각 단계라는 구조 상의 목표를 모두 갖고 있었으나, 구현 상의 목표는 다음 프로젝트 보다는 현재 프로젝트에서 달성했을 때 더 의미있다는 점, 그리고 두 목표를 모두 수행하기는 어렵다는 생각에 구현 상의 목표에 조금 더 치중하여 진행하였습니다. 프로젝트 1 의 방식을 확장하여 구현하였기 때문에 코드는 더 간결해졌고. 함수 포인터 배열과 관련된 문제를 해결하며 위와 같은 구현에 더 익숙해지는 성장의 기회였습니다.