REPORT

Project #02 : Single Cycle MIPS

- **<기존에 남아있던 Freedays : 5일, 이번 과제에 사용할 Freeday : 4일>**

과목명	컴퓨터구조및모바일프로세서	담당교수	유시환 교수님
학 번	32204292	전 공	모바일시스템공학과

이 름 조민혁 제 출 일 2024/05/01

목 차

Ⅰ. 서 론

1. Single Cycle 이란?

Computer Architecture에는 2가지의 명령어 처리방식이 존재하는데, 그것은 Single Cycle과 Multi Cycle이다. Single Cycle은 어떤 하나의 명령어가 처리될 때 Cycle의 수가 1개라는 의미이다. 즉, Single Cycle은 명령어의 종류에 관계 없이 동일한 한 개의 Cycle을 가진다.

명령어가 처리될 때에는 5가지의 State를 가지는데, 명령어를 Memory에서 Register로 load하는 Fetch단계, 가져온 명령어를 해석하는 Decode단계, Decode한 명령어를 바탕으로 실행하는 Execution단계, Data Memory에 접근하여 Memory에 Read/Write의 작업을 하는 Memory Access 단계, Program Counter를 Update하거나, 필요한 경우 Register의 결과를 Update하는 Write Back단계가 있다. 이에 따라, 5개의 State는 하나의 Single Cycle을 이룬다.

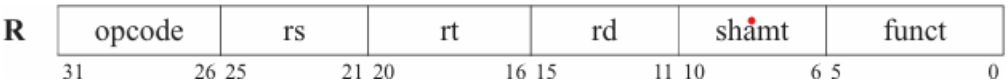
그러나, 명령어마다 필요한 State가 각각 다른데, 모든 명령어는 Single Cycle로 처리되므로 Decode의 단계까지만 필요한 'J Instruction'의 경우 불필요한 시간을 낭비해야한다는 장점이 있다.

어떤 프로그램의 Single Cycle의 State는 프로그램의 명령어들 중 가장 많은 State가 필요한 명령어를 기준으로 Cycle의 State가 결정된다. 이에 따라, 프로그램의 모든 명령어는 원하든 원하지 않든 기준이 되는 명령어의 State 수 만큼 불필요한 시간을 낭비해야만 한다.

2. MIPS란?

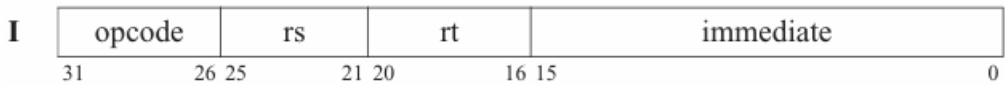
MIPS란 'Microprocessor without Interlocked Pipeline Stages'의 줄임말로 'MIPS Technologies'에서 개발한 RISC계열의 명령어 집합 체계이다. MIPS 명령어는 R-Type, I-Type, J-Type으로 3가지의 명령어 Type이 존재한다.

R-Type의 경우에는 다음 그림과 같이 이루어진다.



CPU는 기계어로 이루어진 명령어를 각 Part마다 6bits, 5bits, 5bits, 5bits, 5bits, 6bits로 나누어 계산한다. R-Type의 경우에는 opcode가 0으로 고정되는데, 이에 따라 funct를 통해 해당 명령어를 구분하는 특성이 있다.

I-Type의 경우에는 다음 그림과 같이 이루어진다.



CPU는 R-Type과는 다르게 각 Part를 6bits, 5bits, 5bits 16bits로 나누며 I-Type에서는 opcode가 구분되어 opcode를 통해 명령어를 구분한다. 또한 16bits의 immediate value를 이용하는데 16bits의 경우 컴퓨터 시스템에 따라 32bits 또는 64bits로 Sign Extend되는 특성이 있다.

CPU는 J-Type을 각각 5bits, 26bits로 나누며 address라고 쓰여있듯 opcode를 바탕으로 명령어를 구분한 후 address로 무조건 jump하는 특성이 있다.

또한 MIPS는 32개의 Register를 사용하는데 Register의 종류는 다음과 같다.

REGISTER NAME, NUMBER, USE, CALL CONVENTION

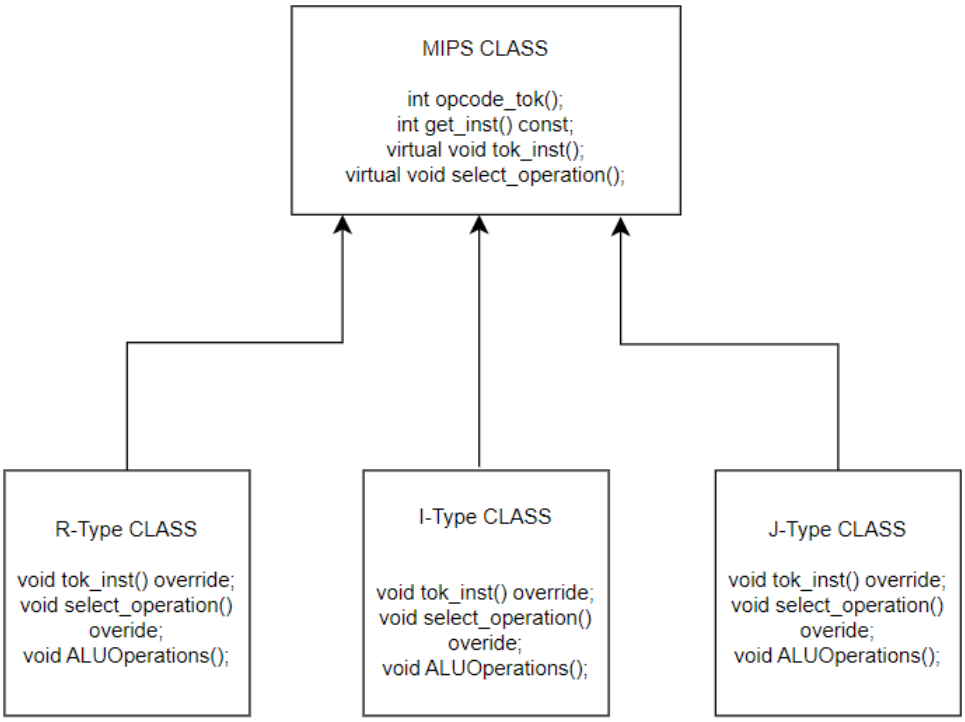
NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

그림에 나와 있듯 $zero Register 경우 0이라는 상수 값이 담겨 있고, $v0-$v1의 경우에는 함수의 실행 결과가 담겨있다. $a0-$a3의 경우 함수를 호출 시 Arguments로 사용하는 Register이고, $s0-$s7의 경우에는 범용 레지스터라고 생각하면 된다.

또한 $sp의 경우 Stack Pointer로 Stack을 가르키고, 관련 값이 들어있는 Pointer Register이고, $fp의 경우 Frame Pointer로 함수 호출시에 Base Point와 같이 기준점을 잡아주는 Frame을 관리해주는 Pointer이다. 마지막으로, $ra는 Return Address로 함수 실행이 종료된 후에 돌아갈 주소가 담겨있는 Register이다.

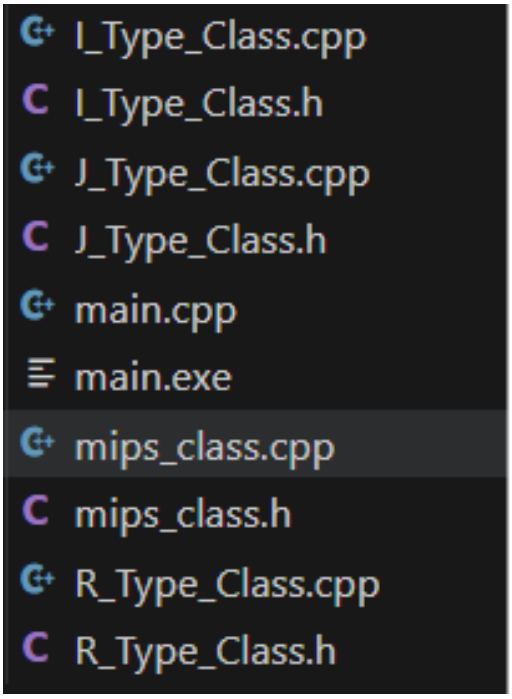
Ⅱ. 본 론

1. 프로그램 개요



<Program Class Diagram>

다음 그림과 같이 Program은 'MIPS Class'를 기초 Class로 두고, 그 아래에는 R-type, I-Type, J-Type Class를 유도 class를 두어 MIPS Class를 상속하도록 하였습니다. void tok_inst(), void select_operation() 두 함수의 경우에는 virtual 선언을 하여 유도 Class들이 override가 가능하도록 하게 하였습니다. 또한, 각 Class에는 ALUOperations()라고 각 Type에 맞는 명령어가 상이하기에 해당 Class에 따라 필요한 명령어들이 선택되고 실행되도록 하였습니다.



<Program을 구성하는 Header File & .cpp File>

각 class를 하나의 File에 담으면 Code의 길이와 가독성 및 유지보수성이 감소하기 때문에, 총 9개의 File로 나누어 구성하였습니다. 먼저, Class별로 각각의 File을 구성하도록 하였고, 그에 따라 필요한 Header File도 구성하였습니다.

File의 Read는 main.cpp에서 이루어지도록 하였고, 최종 결과도 main에서 출력되도록 구성하였습니다.

Visual Studio Code

- 작업 환경 : Visual Studio Code
- 사용한 프로그래밍 언어 : C++
- 사용한 Compiler :

```
{
    "tasks": [
        {
```

```
            "type": "cppbuild",
            "label": "C/C++: g++ build active file",
            "command": "/usr/bin/g++",
            "args": [
                "-fdiagnostics-color=always",
                "-g",
                "${fileDirname}/*.cpp",
                "-o",
                "${fileDirname}/${fileBasenameNoExtension}.exe"
            ],
            "options": {
                "cwd": "${workspaceFolder}"
            },
            "problemMatcher": [
                "$gcc"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "detail": "Task generated by Debugger."
        }
    ],
    "version": "2.0.0"
```

<tasks.json>

- 주의사항 : Test File 사용 시 Input File 경로를 반드시 지켜줘야함

2. 중요 부분 설명

[2-1] 사용한 변수들

```
extern int Memory[0xfffffff]; // Memory Max Size
extern int R[32]; // Register 0 ~ 31
extern int pc; // Program Counter -> PC = PC + 4 (int = 4byte 이므로 코드
상에서는 PC = PC + 1 로 작성)
extern int idx; // Program 을 Memory 에 등록할 때, idx 를 통해 등록 시킴
extern int memory_access_count; // Memory Access State Count
extern int branch_count; // Branch Count
extern int r_count ,i_count, j_count; // R,I,J Create Count
extern int cycle_count; // Total Program Cycle Count
extern int temp_pc;
extern int jump_inst, jump_opcode; // Jump Instruction & Jump Opcode
```

<Program 의 전역변수들>

먼저 Computer Architecture 의 가장 중요한 구성 요소들인 Memory, Register 와 Program Counter 를 전역 변수로 선언하였다. 모두 int Type 으로 선언하였고, MIPS Green Sheet 에 나왔듯, Register 32 개와 Memory 의 크기를 0xfffffff 의 크기로 선언해주었다. Fetch 단계에서 Memory 를 참조하기 위해 idx 를 선언하였고, 프로그램이 실행을 마친 후 Memory 접근과 Branch 횟수, Instruction 이 실행된 횟수 ,그리고 Single Cycle 의 총 횟수를 출력하기 위해 count 변수들을 선언해주었다. 또한, jump 명령어 또는 Branch 될 시 해당 주소의 명령어를 파악하기 위해 jump_inst 와 jump_opcode 변수를 선언해주었다.

[2-2] mips_class.h

```
        virtual void tok_inst(){}; // Pure Virtual Funcion
        virtual void select_operation(){}; // Pure Virtual Function
```

<Pure Virtual Function>

mips_class.h 에서는 먼저, 두개의 순수 가상 함수를 선언하였다. tok_inst() 함수는 나중에 각 명령어 Type 에서 MIPS 의 형식에 맞게 명령어를 tok 하도록 도움을 주는 함수이다. 그리고, select_operation()은 tok 된 명령어로부터 해당 명령어가 어떤 명령어를 수행해야하는지 고르도록 해주는 함수이다.

```
int MIPS_::opcode_tok(){
    _opcode = (_instruction >> 26) & 0x3f; // Extract 6-bits
    return _opcode;
}
```

<opcode_tok()의 구현>

MIPS Class 의 opcode_tok() 함수는 Fetch 단계에서 전달받은 명령어로부터 opcode 가 6bits 이기에 6bits 를 추출하여 return 하도록하였다.

[2-3] Fetch State

Fetch State 를 하기에 앞서 컴퓨터는 데이터를 Little Endian 으로 저장하고 있기에 Big Endian 으로 변환하는 작업이 필요했다.

```
        unsigned int b1, b2, b3, b4;
        b1 = ret & 0x000000ff;
        b2 = ret & 0x0000ff00;
        b3 = ret & 0x00ff0000;
        b4 = (ret>>24) & 0x000000ff;

        int res = (b1<<24) | (b2<<8) | (b3>>8) | b4;
        cout<<"Big Endian : 0x"<<hex<<res<<endl;
        int instruction = res;
```

<Convert Little Endian to Big Endian>

Bit Shift 를 이용하여서 기존의 bits 배치를 역순으로 전환해주는 작업을 진행해주었다.

```
    while(1){

        // Fetch
        cout<<"=======Fetch State======="<<endl;
        int opcode;
        MIPS_ * mips_inst = new MIPS_(Memory[pc]);
        opcode = mips_inst->opcode_tok();

        if(Memory[pc] == 0 && Memory[pc+1] == 0 || pc == 0xffffffff){
            cout<<"Program Halt"<<endl;
            break;
        }
```

<Fetch State>

PC 가 가르키는 Memory 의 명령어가 null 이고, 그 다음 pc 가 가르키는 Memory 의 명령어가 null 이면 프로그램의 끝 부분이라고 생각하여 while 문을 종료시키고 프로그램을 종료시켰습니다. 또한 pc 가 0xffffffff 를 가르키면 Program 의 마지막 부분이라 판단하였기에 이 역시 while 문을 탈출시켜 프로그램을 종료시켰습니다. 그리고, MIPS Class Pointer 가 pc 가 가르키는 Memory 의 명령어를 가르키도록 하여 Fetch 가 진행되도록 하였고, Fetch 된

명령어로부터 opcode 를 token 하여 opcode 를 추출하는 작업을 Fetch 단계에서 진행하도록 하였습니다.

```
// R-type
        if(opcode == 0){
            cout<<"R-Type is Created"<<endl;
            r_count++;
            MIPS_ * r_type_inst = new R_Type(Memory[pc]);
            r_type_inst->tok_inst();
            r_type_inst->select_operation();
            delete r_type_inst;
        }
        // J-type
        else if(opcode == 2 || opcode == 3){
            cout<<"J-Type is Created"<<endl;
            j_count++;
            MIPS_ * j_type_inst = new J_Type(Memory[pc]);
            j_type_inst->tok_inst();
            j_type_inst->select_operation();
            delete j_type_inst;
        }
        // I-type
        else{
            cout<<"I-Type is Created"<<endl;
            i_count++;
            MIPS_ * i_type_inst = new I_Type(Memory[pc]);
            i_type_inst->tok_inst();
            i_type_inst->select_operation();
            delete i_type_inst;
        }
        cout<<"Cycle "<<cycle_count++<<" is Complete"<<endl;
        delete mips_inst;
    }
```

<opcode 를 바탕으로 생성되는 Instruction Type Class>

Token 된 opcode 를 바탕으로 MIPS Green Sheet 에 나온 기준에 맞추어서 해당하는 Type 의 명령어 Class 가 형성되도록 하였고, MIPS Class 의 Type 으로 Pointer 로 선언하면서 모든 명령어가 하나의 개별 객체가 아닌 공통의 객체 Type 으로 바라보게 하였습니다. 각 명령어는 공통의 객체 Type 이기에 객체가 생성된 이후에 tok_inst()를 진행하여 Decode State 가 수행되도록 하였고, Decode 된 명령어를 바탕으로 Operation 을 선택하고 수행하는 Execution State 가 진행되도록 하였습니다.

[2-4] Decode State

```
void R_Type::tok_inst(){
    cout<<"=======Decode State======="<<endl;
    rs = (r_instruction >> 21) & 0x1f; // Extract 5-bits
    rt = (r_instruction >> 16) & 0x1f; // Extract 5-bits
    rd = (r_instruction >> 11) & 0x1f; // Extract 5-bits
    shamt = (r_instruction >> 6) & 0x1f; // Extract 5-bits
    funct = (r_instruction) & 0x3f;  // Extract 6-bits

    cout<<"Opcode : 0x"<<hex<<r_opcode<<", rs : "<<rs<<", rt : "<<rt<<", rd :
"<<rd<<", shamt : "<<shamt<<", funct : "<<funct<<endl;
}
```

```
void I_Type::tok_inst(){
    cout<<"=======Decode State======="<<endl;
    rs = (i_instruction >> 21) & 0x1f;
    rt = (i_instruction >> 16) & 0x1f;
    imm = (i_instruction & 0xffff);
    s_imm = (imm & 0x8000) ? (imm | 0xffff0000) : (imm); // Extend 32bits
```

```
void J_Type::tok_inst(){
    cout<<"=======Decode State======="<<endl;
    address = (j_instruction & 0x03ffffff);
```

<Decode State>

각 Type 에 맞게 생성된 명령어 객체들은 공통적으로 tok_inst()를 수행하도록 하였고, Bit Shift 와 Bitwise operation 을 통해 각 Part 들을 추출하도록 하였습니다. 특히, I-Type 의 경우 Immediate Value 는 Sign Extend 가 필요했기에 MSB 를 기준으로 Bit 확장이 발생하도록 설계하였습니다.

[2-4-1] 주소값 계산

이번 Project 에서 가장 어려웠던 부분이라고 생각하는 것은 바로 '주소값 계산' 입니다. 실제 Memory 와 Program Counter 가 아닌 임의로 생성한 Memory 와 Program Counter 로 주소를 계산해야만 했기에 실제 주소와는 다르게 코드가 흘러가는 걸 보면서 실제 주소와의 관계를 파악해야했습니다.

< Branch Address & Jump Address 의 구성>

(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }

```
    address = (j_instruction & 0x03ffffff);
    address = address << 2;
    JumpAddr = (pc & 0x3f000000) | address;
```

<JUMP 명령어의 주소값 계산>

jump 명령어의 경우 address 는 26bits 로 이루어져있고, Shift left 2 bits 만큼 이동한 후, 증가된 pc 의 [31:28] 부분과 합쳐져 구성되었습니다. 제작된 주소는 JumpAddr 에 할당되도록 하였고,

```
pc = JumpAddr / 4;
```

J-Type 명령어인 'J' 와 'JAL' 명령어에서 4 로 나누는 작업을 진행해주었습니다. 4 로 나눈 이유는 실제로는 명령어가 Memory 에 4 bytes 만큼의 공간 씩 할당되어있지만, 코드 상에는 Memory 에 명령어를 1 index 씩 할당해주었으며, Memory 의 Data Type 은 Int 기에 기본적으로 1 index 당 4bytes 를 차지하고 있기에 4 로 나누면서 형식을 맞춰주는 작업이 필요했기에 4 로 나누었습니다.

```
    // BranchAddr = s_imm << 2; // BranchAddr just is composed by shifting
left 2bits the s_imm
    BranchAddr = (s_imm << 2);
```

I-Type 의 BranchAddr 의 경우에는 MIPS 에서 {14'imm[15], 16'Imm, b'00}로 이루어져있기에 Sign Extend 된 Immediate 를 바탕으로 Shift Left 2 를 진행하여 구성하도록 하였습니다.

```
        pc = ((pc*4 + 4) + _branchaddr) / 4;
```

I-Type 의 분기 명령어인 'BEQ'와 'BNE'의 경우에는 MIPS Architecture 에서 (pc+4)를 한 후 BranchAddress 와 더하여 구성되었기에, (pc+1)을 해준으로 4 를 곱해주었고, 이 후 BranchAddress 를 더하여 마지막에 Int Data Type 과의 형식을 맞추기 위해 4 로 나누는 작업을 해주어 Program Counter 의 값을 계산해주었습니다.

[2-5] Execution State

```
void R_Type::select_operation(){
    cout<<"=======Execution State========"<<endl;
    switch (funct){ // R-type is classified using funct
        case 0x20: // add rs, rt, rd
            add(rs, rt, rd);
            break;
        case 0x21: // addu rs, rt, rd; if rt == 0, then move rd, rs
            addu(rs, rt, rd);
            break;
        case 0x24:
            _and(rs, rt, rd); // AND rd rs rt
            break;
        case 0x08:
            jump_register(rs); // J rs
            break;
        case 0x27:
            _nor(rs, rt, rd); // NOR rd, rs, rt
            break;
        case 0x25:
            _or(rs, rt, rd); // OR rd, rs, rt
            break;
        case 0x2a:
            slt(rs, rt, rd); // SLT rd, rs, rt
            break;
        case 0x2b:
            sltu(rs, rt,    rd); // SLTU rd, rs, rt
```

```
            break;
        case 0x00:
            sll(rt, rd, shamt); // SLL rd, rt, shamt
            break;
        case 0x02:
            srl(rt, rd, shamt); // SRL rd, rt, shamt
            break;
        case 0x22:
            sub(rs, rt, rd); // SUB rd, rs, rt
            break;
        case 0x23:
            subu(rs, rt, rd); // SUBU rd, rs, rt
            break;
        default:
            cout<<"There is no matching operation"<<endl;
            break;

    }
}
```

<R-Type's select_operation()>

```
void I_Type::select_operation(){
    cout<<"=======Execute State======="<<endl;
    switch(i_opcode){ // I-Type is classified using opcode
        case 0x08:
            addi(rs, rt, s_imm); // addi rs rt imm
            break;
        case 0x09:
            addiu(rs, rt, s_imm); // addiu rs rt imm
            break;
        case 0x0c:
            andi(rs, rt, imm); // andi rs rt imm
            break;
        case 0x04:
            beq(rs, rt, BranchAddr); // beq rs rt imm
            break;
        case 0x05:
            bne(rs, rt, BranchAddr); // bne rs rt imm
            break;
        case 0x0f:
            lui(rt, imm); // lui rt imm
            break;
        case 0x23:
            lw(rs, rt, s_imm); // lw rt imm(rs)
            break;
        case 0x0d:
            ori(rs, rt, imm); // ori rt rs imm
```

```
            break;
        case 0x0a:
            slti(rs, rt, s_imm); // slti rt rs imm
            break;
        case 0x0b:
            sltiu(rs, rt, s_imm); // sltiu rt rs imm
            break;
        case 0x2b:
            sw(rs, s_imm, rt); // sw rt imm(rs)
            break;
        default:
            cout<<"There is no matching Instruction"<<endl;
            break;
    }
}
```

<I-Type's select_operation()>

```
void J_Type::select_operation(){
    cout<<"=======Execute State======="<<endl;
    switch(j_opcode){
        case 0x02:
            jump();
            return;
        case 0x03:
            jump_and_link();
            return;
        default:
            cout<<"There is no matching operation"<<endl;
            return;
    }
}
```

<J-Type's select_operation()>

모든 명령어 객체들은 select_operation() 함수를 override 함으로써 명령어를 수행하도록 하였으며, I-Type 과 J-Type 의 경우에는 opcode 를 통해 switch~case 문에서 선택하도록 하였지만, R-Type 의 경우에는 funct 를 통해 operation 을 선택하기에 funct 를 switch~case 문에 할당해주었습니다.

이후 각 명령어는 해당되는 명령어 함수를 통해 연산이 이루어는 과정을 거

```
void J_Type::jump_and_link(){
    cout<<"=======Jump And Link Operantion======="<<endl;
    R[31] = pc + 2;
```

```
    pc = JumpAddr / 4;
    do{
        jump_inst = Memory[pc];
        MIPS_ * jump_mips = new MIPS_(jump_inst);
        jump_opcode = jump_mips ->opcode_tok();

        // R-type
        if(jump_opcode == 0){
                cout<<"R-Type is Created"<<endl;
                r_count++;
                MIPS_ * r_type = new R_Type(Memory[pc]);
                r_type->tok_inst();
                r_type->select_operation();
                delete r_type;
            }
            // J-type
            else if(jump_opcode == 2 || jump_opcode == 3){
                cout<<"J-Type is Created"<<endl;
                j_count++;
                MIPS_ * j_type = new J_Type(Memory[pc]);
                j_type->tok_inst();
                j_type->select_operation();
                delete j_type;
            }
            // I-type
            else{
                cout<<"I-Type is Created"<<endl;
                i_count++;
                MIPS_ * i_type = new I_Type(Memory[pc]);
                i_type->tok_inst();
                i_type->select_operation();
                delete i_type;
            }
        delete jump_mips;
    }while(pc != R[31] - 2);

    pc = R[31];
    cout<<"Return PC: "<<pc<<endl;
    cout<<"=======Write Back======="<<endl;
    cout<<"PC: "<<pc<<endl;
}
```

<J-Type's 'JAL' Instruction>

이 중 J-Type 의 'JAL' 명령어를 살펴보면 pc 에 target_address 를 할당해준 후 target_address 를 바탕으로 객체를 생성하여 기존에 진행했던 Fetch 와

Decode 를 수행해주는 과정을 추가하여 해당 주소에 맞는 명령어가 올바르게 Decode 되어 수행되도록 하였습니다. Target Address 를 간 후 해당 부분의 명령어만 수행하고 return 하는 것이 아닌 Target Address 를 시작으로 순차적으로 명령어가 수행되어야만 했기에 while 문을 통해 반복적으로 수행하게 하였고, 특정 조건이 성립할 시 탈출하도록 하였습니다. 이외에도 분기하는 명령어들인 'JR', 'J', 'BEQ', 'BNE' 역시 이와 같은 로직으로 수행되도록 하였습니다.

[2-6] Memory Access State

Memory Access State 의 경우에는 Memory Access 하는 명령어는 한정되어 있었습니다. 'SW' Instruction 과 'LW' Instuction 으로 Memory 에 Access 할 수 있었고,

```
void I_Type::sw(uint32_t _rs, int32_t _imm, uint32_t _rt){
    cout<<"=======SW Operation======="<<endl;
    cout<<"=======Memory Access======="<<endl;
    Memory[static_cast<uint32_t>(R[_rs] + _imm)] = R[_rt];
    cout<<"Memory : "<<Memory[static_cast<uint32_t>(R[_rs] + _imm)]<<endl;
    cout<<"Memory["<<R[_rs] + _imm<<"] = "<<"R["<<_rt<<"]"<<endl;
    pc = pc + 1;
    memory_access_count++;

    cout<<"=======Write Back======="<<endl;
    cout<<"R["<<_rt<<"] is "<<R[_rt]<<endl;
    cout<<"PC: "<<pc<<endl;
}
```

<SW Instruction>

```
void I_Type::lw(uint32_t _rs, uint32_t _rt, int32_t _imm){
    cout<<"=======LW Operation======="<<endl;
    cout<<"=======Memory Access======="<<endl;
    R[_rt] = Memory[static_cast<uint32_t>(R[_rs] + _imm)];
    cout<<"Memory : "<<Memory[static_cast<uint32_t>(R[_rs] + _imm)]<<endl;

    cout<<"R["<<_rt<<"] = Memory["<<"R["<<_rs<<"] + "<<_imm<<"]"<<endl;
    pc = pc + 1;
    memory_access_count++;
    cout<<"=======Write Back======="<<endl;
    cout<<"R["<<_rt<<"] is "<<R[_rt]<<endl;
    cout<<"PC: "<<pc<<endl;
}
```

<LW Instruction>

SW 명령어와 LW 명령어는 이와 같은 형식으로 이루어져있습니다. 먼저 Memory 의 Index 는 양수여야했기에 unsigned 로 변환해주는 작업이 필요했기에, 그 과정을 수행하였고, 이후 해당 Index 에 접근하여 값을 Read 하여서 Register 에 Write 하거나 Register 의 값을 해당 Memory Index 에 Write 하도록 하였습니다. 이후 Memory Access State 를 진행하였기에 memory_access_count 를 1 증가하였습니다.

[2-7] Write Back State

Wrtie Back State 의 경우에는 Execution State, Memory Access State 가 끝난 후 결과를 Update 해주는 과정과 PC 를 증가시켜주는 과정이기에 그에 따른 과정이 필요했습니다.

```
    cout<<"=======Write Back======="<<endl;
    pc = pc + 1;
    cout<<"R["<<_rd<<"] is "<<R[_rd]<<endl;
    cout<<"PC: "<<pc<<endl;
```

<R-Type's Write Back>

R-Type 의 Write Back 같은 경우에는 pc 를 4bytes 인 1 만큼 증가시켜주었고, 필요한 경우 연산의 결과를 R[rd]에 update 해주었습니다.

```
    pc = pc + 1;
    cout<<"=======Write Back======="<<endl;
    cout<<"R["<<_rt<<"] is "<<R[_rt]<<endl;
    cout<<"PC: "<<pc<<endl;
```

<I-Type's Write Back>

I-Type 의 Write Back 같은 경우에도 pc 를 증가시켜주었으며, I-Type 에서는 Destination 의 Register 가 R[rt]이기에 필요한 경우 결과를 R[rt]에 update 해주었고, Write Back 수행 결과를 출력하여 확인하도록 하였습니다.

J-Type 의 경우에는 Decode State 와 Execution State 의 단계만 필요했기에 Write Back State 를 가지지 않아 해당하는 부분이 필요 없었습니다.

[3-1] 실행 결과

```
R[1e] = Memory[R[1d] + 4]
=======Write Back=======
R[1e] is 0
PC: 6
Cycle 6 is Complete
=======Fetch State=======
The instruction is 27bd0008
I-Type is Created
=======Decode State=======
Opcode : 0x9, rs : 1d, rt : 1d, imm : 8
=======Execute State=======
=======ADDIU Operation=======
=======Write Back=======
R[1d] is 1000000
PC: 7
Cycle 7 is Complete
=======Fetch State=======
The instruction is 3e00008
R-Type is Created
=======Decode State=======
Opcode : 0x0, rs : 1f, rt : 0, rd : 0, shamt : 0, funct : 8
=======Execution State=======
=======JUMP Register Operation=======
Program Finish!!!
Cycle 8 is Complete
=======Fetch State=======
Program Halt
=======The Program is Finished=======
=======Completion of The Program=======
The Result (r2) : 0
Total Executed Instruction : 8
R-Type Instruction : 4
I-Type Instruction : 4
J-Type Instruction : 0
Memory Access Instructions : 2
Branch Instructions : 0
=====================================
Bye
========================================================================
```

<Simple.bin 의 프로그램 실행 결과 화면>

```
R[1e] = Memory[R[1d] + 14]
=======Write Back=======
R[1e] is 0
PC: 8
Cycle 8 is Complete
=======Fetch State=======
The instruction is 27bd0018
I-Type is Created
=======Decode State=======
Opcode : 0x9, rs : 1d, rt : 1d, imm : 18
=======Execute State=======
=======ADDIU Operation=======
=======Write Back=======
R[1d] is 1000000
PC: 9
Cycle 9 is Complete
=======Fetch State=======
The instruction is 3e00008
R-Type is Created
=======Decode State=======
Opcode : 0x0, rs : 1f, rt : 0, rd : 0, shamt : 0, funct : 8
=======Execution State=======
=======JUMP Register Operation=======
Program Finish!!!
Cycle a is Complete
=======Fetch State=======
Program Halt
=======The Program is Finished=======
=======Completion of The Program=======
The Result (r2) : 100
Total Executed Instruction : 10
R-Type Instruction : 3
I-Type Instruction : 7
J-Type Instruction : 0
Memory Access Instructions : 4
Branch Instructions : 0
====================================
Bye
================================================================
```

<Simple2.bin 의 프로그램 실행 결과 화면>

```
R[1e] = Memory[R[1d] + 14]
=======Write Back=======
R[1e] is 0
PC: 19
Cycle 4ca is Complete
=======Fetch State=======
The instruction is 27bd0018
I-Type is Created
=======Decode State=======
Opcode : 0x9, rs : 1d, rt : 1d, imm : 18
=======Execute State=======
=======ADDIU Operation=======
=======Write Back=======
R[1d] is 1000000
PC: 1a
Cycle 4cb is Complete
=======Fetch State=======
The instruction is 3e00008
R-Type is Created
=======Decode State=======
Opcode : 0x0, rs : 1f, rt : 0, rd : 0, shamt : 0, funct : 8
=======Execution State=======
=======JUMP Register Operation=======
Program Finish!!!
Cycle 4cc is Complete
=======Fetch State=======
Program Halt
=======The Program is Finished=======
=======Completion of The Program=======
The Result (r2) : 5050
Total Executed Instruction : 1330
R-Type Instruction : 409
I-Type Instruction : 920
J-Type Instruction : 1
Memory Access Instructions : 613
Branch Instructions : 101
====================================
Bye
======================================================================
```

<Simple3.bin 의 프로그램 실행 결과 화면>

이로써 Simple.bin, Simple2.bin, Simple3.bin 까지 올바르게 출력되는 걸 확인할 수 있었습니다.

Ⅲ. 결 론

3. 소감 및 아쉬운 점

이번 Project2 : Single Cycle Processor를 진행하면서 중간고사 기간과 맞물려 있어서 비교적 이전 과제만큼 크게 신경을 쓰지 못하였습니다. 많은 시간을 투자하지 못하였기에 프로그램의 완성도도 비교적 떨어진다고 생각합니다. 그에 따라 simple1.bin, simple2.bin, simple3.bin까지 올바르게 실행되지만 simple4.bin의 경우 원하는 결과가 출력되지 않는 아쉬운 점이 존재합니다.

앞으로도 프로젝트 과제를 진행할 때 시험기간이 겹치더라도, 일정 시간은 프로젝트 과제에 할애해야겠다고 생각했습니다. 짧은 시간이었지만 Single Cycle Processor를 구현하기 위해 많은 정보를 찾아보고, 이론으로만 배웠던 Single Cycle의 개념을 직접 프로그래밍 언어로 코딩하면서 명확히 나의 것으로 만드는 시간이었던 것 같아서 이 점에서는 프로그램의 완성도를 떠나 보람차다고 생각합니다.

오늘날에야 당연하게 사용하는 명령어들과, 항상 어렵다고만 생각했던 MIPS 어셈블리어의 내부 동작을 직접 구현해보면서 한 치의 Error 없이 작동되는 MIPS Architecture와 Processor에 개인적으로는 대단하다는 느낌을 받았고, 겉으로는 간단해보여도 내부적으로는 수많은 코드들을 통해 실행되고 있다는 점과, Computer Architecture를 직접 그려보고 MIPS 명령어와의 관계를 살펴보면서 완벽히 호환되는 놀라움에 한번 더 감탄하였습니다.

개인적으로도 크게 성장하는 시간을 가졌고, 어려움이 있었지만 그래도 끝까지 해결하려고 하였던 저의 모습에는 스스로 뿌듯하게 생각하고 있습니다. 다음 과제에는 모든 Option까지 구현해보는 완벽한 Program을 구현하는 것을 목표로 프로젝트를 진행하고자합니다. 늘 좋은 과제와 훌륭한 강의를 해주시는 교수님께도 감사드립니다.