Mips Single Cycle

32213763 임석민

I. Introduction

프로젝트의 목표는 MIPS CPU 에뮬레이터를 구현하는 것이다. MIPS 에뮬레이터는 주어진 MIPS 프로그램을 이진 파일 형식으로 입력 받아 메모리에 load 후, MIPS 명령어를 실행한다. CPU는 싱글 사이클 구조로 동작하며, 명령어 실행을 위해 Instruction fetch, Instruction decode, Execution, Memory access, Write back과 같은 다섯 단계를 구현해야 한다. 각 클럭 사이클마다 상태 변화를 출력하며, 예외 상황에 대해서도 처리해야 한다. 이 프로젝트의 목표는 MIPS 명령어 세트의 주요 기능을 구현하여 프로그램을 정확하게 실행하는 것이다. 또한, 메모리 액세스, 레지스터 갱신, 분기 처리 등과 같은 중요한 구현 사항을 고려하여 MIPS CPU의 동작을 재현해야 한다. 이를 통해 컴퓨터 아키텍처와 명령어 처리에 대한 심층적인 이해를 도모하고, MIPS 아키텍처를 실제로 경험해 볼 수 있다. 또한, C 언어를 사용하여 프로그래밍 능력을 향상시킬 수 있을 것으로 생각됨.

II. Background

A. ISA (Instruction Set Architecture)

ISA는 컴퓨터 시스템에서 프로세서가 이해하고 실행하는 명령어 집합을 정의하는 인터페이스이다. ISA는 하드웨어와 소프트웨어 사이의 중간 계층으로 작동하여 프로세서의 동작과 명령어 형식, 레지스터 구성, 명령어 집합 등을 규정한다. 이를 통해 프로그래머와 컴퓨터 아키텍처 사이의 추상화를 제공하며, 소프트웨어 개발자가 하드웨어를 직접 다루지 않고도 프로그램을 작성할 수 있도록 한다. ISA는 명령어의 종류, 명령어의 길이, 데이터 유형, 레지스터의 수와 용도 등을 명시하며, 이를 기반으로 컴파일러, 어셈블러, 인터프리터 등이 프로그램을 생성하고 실행한다. 다양한 ISA가 존재하며, 대표적으로 x86, ARM, MIPS 등이 있다. ISA는 컴퓨터 아키텍처의 핵심 요소로, 시스템의 성능, 호환성, 확장성 등에 영향을 미친다.

B. MIPS

MIPS 아키텍처는 RISC기반의 컴퓨터 아키텍처로, 단순하고 효율적인 명령어 집합 구조를 갖추고 있다. MIPS 아키텍처는 명령어 길이가 32bit로 동일하며 I, J, R type들로 고정된 형식을 가지며, 명령어 실행을 위해 32개의 레지스터를 사용한다. MIPS 아키텍처는 단일 사이클, 다중 사이클, 파이프라인 등 다양한 실행 모델을 지원하며, 명령어의 효율적인 실행을 위해 분기 예측 등의 기술을 사용한다. MIPS 아키텍처는 명령어의 간결성과 파이프라인화의 장점으로 인해 다양한 응용 분야에서 사용되며, 학습용 컴퓨터 아키텍처로도 널리 활용된다.

C. MIPS single Cycle

MIPS Single Cycle은 MIPS 아키텍처를 기반으로 한 단일 사이클 구조의 CPU 설계이다. 이 구조는 각 명령어를 실행하는 데 필요한 단계를 하나의 클럭 사이클 동안 완료하는 방식으로 동작한다. 실행 단계는 명령어를 메모리에서 가져오는 Instruction Fetch, 명령어를 해석하고 제어 신호를 생성하는 Instruction Decode, 연산을 수행하는 Execution, Memory Access, 레지스터 갱신 및 결과를 저장하는 Write Back 단계로 구성된다.

MIPS Single Cycle은 명령어마다 고정된 사이클 수가 필요로 한다. 그로 인해 명령어마다 동일한 시간이 필요로 되어지고 이에 따라 명령어에 따라 성능차이가 날 수 있어 성능면에서 효율적이지 않을 수 있다. 그래서 Pipelined MIPS을 통해 한 번에 여러 명령어들을 실행시켜 효율성을 증가시키는 방법이 있다.

III. Implementation

A. Program Structure & Design

기본적으로 MIPS는 32bit명령어 체계를 가지고 있다 따라서 32bit크기를 가진 데이터형을 이용하여 구현하기 위해서는 int형을 이용하였다 하지만 signed int의 경우 msb에 의해 음수 양수가 정해지기에 데이터를 저장할 때 오류가 발생할 가능성이 있기에 unsigned int를 사용하여 대부분을 구현하였고 sign int가 필요한 경우 casting을 이용하였다.

i. Control Unit

MIPS Single Cycle 구조에서 Control Unit은 명령어 실행의 제어를 담당한다 Control Unit은 명령어의 세부 동작을 제어하고 필요한 신호를 생성하여 ALU, 레지스터 파일, 메모리 등과 상호 작용한다. 또한, 예외 상황을 감지하고 처리하며 분기 명령어의 조건을 확인하고 분기 동작을 수행한다. 이를 통해 Control Unit은 명령어 실행 흐름을 조정하고 CPU의 다른 하위 시스템과 상호 작용하여 명령어를 올바르게 실행할 수 있도록 한다.

```
void control_unit(Operands* operands, ControlSignals* cs, unsigned int* pc)
{
    switch (operands->opcode) { ... }

    unsigned int op0 = 0, op1 = 0, op2 = 0, op3 = 0, op4 = 0, op5 = 0;
    unsigned int rformat = 0, lw = 0, sw = 0, beq = 0, jump = 0;

    if (operands->opcode == 0 | operands->opcode == 0X23 | operands->opcode == 0X2B | operands->opcode == 0X04 | operands->opcode == 0X05 | operands->opcode == 0X02)
    {
        op0 = operands->opcode & 0x1; // 0x1 = 0000 0001
        op1 = (operands->opcode & 0x2) >> 1; // 0x1 = 0000 0010
        op2 = (operands->opcode & 0x4) >> 2; // 0x1 = 0000 0100
        op3 = (operands->opcode & 0x8) >> 3; // 0x1 = 0000 1000
        op4 = (operands->opcode & 0x10) >> 4; // 0x1 = 0001 0000
        op5 = (operands->opcode & 0x20) >> 5; // 0x1 = 0010 0000

        rformat = ((~op0) & 0x1) & ((~op1) & 0x1) & ((~op2) & 0x1) & ((~op3) & 0x1) & ((~op4) & 0x1) & ((~op5) & 0x1);
        lw = op0 & op1 & ((~op2) & 0x1) & ((~op3) & 0x1) & ((~op4) & 0x1) & op5;
        sw = op0 & op1 & ((~op2) & 0x1) & op3 & ((~op4) & 0x1) & op5;
        beq = ((~op0) & 0x1) & ((~op1) & 0x1) & op2 & ((~op3) & 0x1) & ((~op4) & 0x1) & ((~op5) & 0x1);
        jump = ((~op0) & 0x1) & ((~op1) & 0x1) & ((~op2) & 0x1) & ((~op3) & 0x1) & op4 & ((~op5) & 0x1);

        cs->RegDst = rformat;
        cs->ALUSrc = lw | sw;
        cs->MemtoReg = lw;
        cs->RegWrite = rformat | lw;
        cs->MemRead = lw;
        cs->MemWrite = sw;
        cs->Branch = beq;
        cs->Jump = jump;

        if (operands->funct == 0X08)
        {
            cs->RegDst = 0;
            cs->ALUSrc = 0;
            cs->MemtoReg = 0;
            cs->RegWrite = 0;
            cs->MemRead = 0;
            cs->MemWrite = 0;
            cs->Branch = 0;
            cs->Jump = 1;
        }
    }
```

그림 3-1-1

먼저 control unit 함수는 opcode를 통해 control signal을 만들어 내는 함수이다. 원래는 실제 control unit과 같이 모든 opcode의 control signal을 만들어 내야 하지만 정보의 부족으로 R-type일부, Beq, Jump, sw, lw의 control signal을 생성하는 PLA회로(그림3-1-2)만을 기반으로 코드를 작성하여 control signal을 생성하였다. (그림 3-1-1) 나머지 I-type, J-type instruction들은 아래 사진의 truth table(그림 3-1-4)을 바탕으로 코드에 직접 입력하였다. Bne, jr의 경우 일부 정보만 수정하면 되기에 PLA회로를 이용하여 생성된 control signal 수정하는 코드를 작성하였다. (그림 3-1-3)

```
else if (operands->opcode == 0X09 | operands->opcode == 0X0A)
{
    cs->RegDst = 0;
    cs->ALUSrc = 1;
    cs->MemtoReg = 0;
    cs->RegWrite = 1;
    cs->MemRead = 0;
    cs->MemWrite = 0;
    cs->Branch = 0;
    cs->Jump = 0;

}
else if (operands->opcode == 0X04)
{
    cs->RegDst = 0;
    cs->ALUSrc = 1;
    cs->MemtoReg = 0;
    cs->RegWrite = 0;
    cs->MemRead = 0;
    cs->MemWrite = 0;
    cs->Branch = 0;
    cs->Jump = 0;
}
else if (operands->opcode == 0X03)
{
    cs->RegDst = 1;
    cs->RegWrite = 0;
    cs->Jump = 1;
}
```

그림 3-1-3

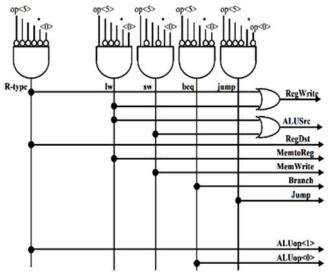


그림 3-1-2

Instruction	Opcode	Sh_B	S_zext	RegWrite	RegDst	Aluop	Branch	Brchne	Blez	Bltz	Bgtz	Memwrite	Memtoreg	Jump	Jal	ALUop	Ilt
R_type	000000	xx	x	1	01	00	0	0	0	0	0	0	00	0	0	110	0
lw	100011	xx	x	1	00	01	0	0	0	0	0	0	01	0	0	000	0
sw	101011	11	x	0	xx	01	0	0	0	0	0	1	xx	0	0	000	0
beq	000100	xx	x	0	xx	00	1	0	0	0	0	0	xx	0	0	001	0
bne	000101	xx	x	0	xx	00	0	1	0	0	0	0	xx	0	0	001	0
blez	000111	xx	x	0	xx	00	0	0	1	0	0	0	xx	0	0	001	0
bltz	000001	xx	x	0	xx	00	0	0	0	1	0	0	xx	0	0	001	0
bgtz	000110	xx	x	0	xx	00	0	0	0	0	1	0	xx	0	0	001	0
addi	001000	xx	x	1	00	01	0	0	0	0	0	0	00	0	0	000	0
addiu	001001	xx	x	1	00	01	0	0	0	0	0	0	00	0	0	000	0
j	000010	xx	x	0	xx	xx	x	x	x	x	x	0	xx	1	0	xxx	0
jal	000011	xx	x	1	10	xx	x	x	x	x	x	0	xx	1	1	xxx	0
andi	001100	xx	x	1	00	10	0	0	0	0	0	0	00	0	0	010	0
ori	001101	xx	x	1	00	10	0	0	0	0	0	0	00	0	0	011	0
xori	001110	xx	x	1	00	10	0	0	0	0	0	0	00	0	0	100	0
slti	001010	xx	x	1	00	01	0	0	0	0	0	0	00	0	0	101	0
sltiu	001011	xx	x	1	00	01	0	0	0	0	0	0	00	0	0	101	0
lui	001111	xx	x	1	00	11	0	0	0	0	0	0	00	0	0	000	0
lb	100000	xx	0	1	00	01	0	0	0	0	0	0	11	0	0	000	0
lbu	100100	xx	1	1	00	01	0	0	0	0	0	0	11	0	0	000	0
lh	100001	xx	0	1	00	01	0	0	0	0	0	0	10	0	0	000	0
lhu	100101	xx	1	1	00	01	0	0	0	0	0	0	10	0	0	000	0
sb	101000	00	x	0	xx	01	0	0	0	0	0	1	xx	0	0	000	0
sh	101001	01	x	0	xx	01	0	0	0	0	0	1	xx	0	0	000	0
hlt	111100	xx	x	0	xx	xx	x	x	x	X	x	0	xx	x	x	xxx	1

그림 3-1-4

ii. Instruction Memory

Instruction Memory는 실행될 명령어를 저장하며 읽기 전용인 메모리이며 주로 이진 형식 MIPS 명령어들을 저장한다. Instruction Memory는 프로그램의 시작 주소에서부터 순차적으로 명령어를 저장한다.

먼저, 주어진 파일의 이름을 사용하여 해당 파일의 크기를 읽어온다. 파일 크기를 토대로, 그 크기만큼의 배열을 생성한다. 이 배열은 Instruction Memory를 구성하는 데 사용된다.

그 다음, 우리는 1바이트씩 파일을 읽어와서 배열에 하나씩 저장합니다. 이렇게 하여 명령어 메모리에 프로그램의 명령어들을 load한다. 주소 값은 일반적으로 1워드(4바이트)를 기준으로 하며, 주소는 4의 배수이다. 따라서 1바이트씩 작성하는 경우, 중간 주소에서 어떤 순서로 저장할지 결정해야 한다.

```
// Open the binary file for reading
if (fopen_s(&file, "input1.bin", "rb") != 0 || file == NULL) {
    printf("Error opening file.\n");
    return 1;
}

// Determine the file size
fseek(file, 0, SEEK_END);
long file_size = ftell(file);
rewind(file);

// Allocate memory to store the file content
// Inst mem bigendian
unsigned int* inst_mem = (unsigned int*)malloc(file_size * sizeof(unsigned int));
if (inst_mem == NULL) {
    printf("Error allocating memory.\n");
    fclose(file);
    return 1;
}

while ((bin = fgetc(file)) != EOF)
{
    inst_mem[length++] = bin;
}
```

그림 3-2-1

여기서 사용되는 두 가지 방식은 Little Endian과 Big Endian이다. Little Endian은 낮은 주소에 낮은 자릿수 값을 저장하는 방식이고, Big Endian은 낮은 주소에 높은 자릿수 값을 저장하는 방식이며 이번에는 Big Endian 방식을 선택하여 작성하였다.

동일한 원리로 데이터 메모리도 구성해 Instruction, Data Memory 모두 일관된 나열 방식을 사용한다. (그림 3-2-1)

iii. Registers

Register는 고정된 크기의 저장 공간으로, CPU 내부에서 데이터를 일시적으로 보관하고 액세스 시간이 매우 짧으며, CPU가 명령어를 실행하고 계산을 수행하는 동안 데이터를 효율적으로 처리할 수 있도록 도와준다. MIPS 아키텍처에서는 총 32개의 레지스터가 있고, 각각 32bit를 저장할 수 있다. 레지스터는 숫자로 식별되며, 각 레지스터는 특정 목적을 가지고 있다.

```
//Initialize
*inst = 0;
*pc = 0;
memset(mem, 0, 4294967296 * sizeof(unsigned char));
memset(reg, 0, 32 * sizeof(unsigned int));
reg[31] = 0XFFFFFFFF;//ra
reg[29] = 0X10000000;//sp

//Start the simulation
while (*pc < length | *pc != 0XFFFFFFFF)
{
    //Zero registor
    reg[0] = 0;
```

Register를 unsigned int로 선언된 32개의 인자가 있는 배열을 할당하고 특정 register마다 특정 값을 가져야 하는 경우 RA, SP, Zero register의 경우 값을 할당한 후 실행한다. Zero register의 경우는 항상 0이기 때문에 cycle이 종료될 때마다 0으로 initialize해준다.

그림 3-3-1

iv. Arithmetic Logic Unit

ALU는 산술 연산과 논리 연산을 수행하여 데이터의 계산을 담당한다.

```
switch (cs->ALUOp)
{
case 0:
    //add
    *alu_result = data1 + data2;
    break;

case 1:
    //sub
    *alu_result = data1 - data2;
    break;

case 2:
    //and
    *alu_result = data1 & data2;
    break;

case 3:
    //or
    *alu_result = data1 | data2;
    break;

case 4:
    //nor
    *alu_result = ~(data1 | data2);
    break;

case 5:
    //slt
    if (data1 < data2)
        *alu_result = 1;
    else
        *alu_result = 0;
    break;

case 6:
    //shift sll
    *alu_result = data1 << operands->shamt;
    break;

case 7:
    //shift srl
    *alu_result = data1 >> operands->shamt;
    break;

default:
    break;
}
```

그림 3-4-1

먼저 control signal과 마찬가지로 ALU op signal을 이용하여 어떤 operation을 해야 하고 실행할 지 정해야 하지만 회로를 몰라서 임의의 ALU op signal을 할당하고 그에 해당하는 operation을 하도록 연결하여 ALU의 역할을 구현하였다. (그림 3-4-1)

그리고 Branch가 있는 경우 Bcond signal을 결과로 내기 때문에 그 신호를 설정하기 위한 연산도 구현하였다. (그림 3-4-2)

```
//b) Branch Evaluation
switch (operands->opcode)
{

case 0X04:
    if (reg[operands->rs] == reg[operands->rt])
        cs->Bcond = 1;
    else
        cs->Bcond = 0;
    break;

case 0X05:
    cs->Branch = 1;
    if (reg[operands->rs] != reg[operands->rt])
        cs->Bcond = 1;
    else
        cs->Bcond = 0;
    break;

default:
    cs->Bcond = 0;
    break;
}
```

그림 3-4-2

v. Data Memory

데이터 메모리는 Address와 Data로 구성된다. 주소는 고유한 메모리 위치를 나타내며, 데이터는 해당 주소에 저장되는 값이다. MIPS의 Data Memory의 주소 범위는 0X00000000부터 0XFFFFFFFF까지이으로 1Byte가 2^32개 있는 것이다. 따라서 배열의 크기가 2^32인 배열을 선언하고 데이터의 나열방식은 Big-endian방식을 통하여 값을 저장하게 된다. (그림 3-5-1)

```
unsigned char* mem = (unsigned char*)malloc(4294967296 * sizeof(unsigned char));
```

그림 3-5-1

```
//big endian
//b) Load Data
if (cs->MemRead == 1 && cs->MemWrite == 0) {
    *read_data = (mem[*alu_result] << 24) & 0XFF000000 | (mem[*alu_result + 1] << 16) & 0X00FF0000 | (mem[*alu_result + 2] << 8) & 0X0000FF00 | (mem[*alu_result + 3] << 0) & 0X000000FF;
    printf("addr: %X, data: %X %X %X %X, load data: %X\n", *alu_result, mem[*alu_result], mem[*alu_result + 1], mem[*alu_result + 2], mem[*alu_result + 3], *read_data);
}

//c) Store Data
else if (cs->MemRead == 0 && cs->MemWrite == 1) {
    mem[*alu_result] = (reg[operands->rt] & 0XFF000000) >> 24;
    mem[*alu_result + 1] = (reg[operands->rt] & 0X00FF0000) >> 16;
    mem[*alu_result + 2] = (reg[operands->rt] & 0X0000FF00) >> 8;
    mem[*alu_result + 3] = (reg[operands->rt] & 0X000000FF) >> 0;
    printf("addr: %X, data: %X %X %X %X, stored data: %X\n", *alu_result, mem[*alu_result], mem[*alu_result + 1], mem[*alu_result + 2], mem[*alu_result + 3], reg[operands->rt]);
}
```

그림 3-5-2

위 그림 (그림 3-5-2)는 Big-endian방식으로 Data memory에 저장 및 출력할 수 있도록 Store word, Load word를 구현한 것이다.

vi. Mux

Mux는 Multiplexer로, 다수의 입력 신호 중에서 하나의 출력을 선택하는 논리 회로이다. MIPS에서 register에 값이 쓰이거나 두 값 중 한 값을 선택하는데 사용된다. (그림 3-6-1)

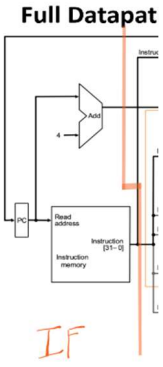
```
unsigned int Mux(unsigned int a, unsigned int b, unsigned int cs) //0,1
{
    if(cs==0)
        return a;
    else
        return b;
}
```

그림 3-6-1

B. Stages

i. Instruction Fetch

Program Counter에서 다음 명령어의 주소를 가져와 명령어 메모리에서 해당 명령어를 읽어온다. PC를 증가시켜 다음 명령어를 가리킨다. (그림 3-7-1)



해당 instruction을 읽어온 뒤 비트 연산을 통해 1Byte씩 나눠진 instruction들을 32bit 명령으로 전환시킨 뒤 pc를 다음 instruction으로 옮긴다. (그림 3-7-2)

```
void F1_If(unsigned int* pc, unsigned int* inst_mem, unsigned int* inst) {
    //Read the PC

    //Fetch instruction from memory & Store the fetched instruction
    *inst = 0;
    *inst = (inst_mem[*pc] << 24) + (inst_mem[*pc + 1] << 16) + (inst_mem[*pc + 2] << 8) + inst_mem[*pc + 3];
    //Prepare the next PC
    *pc += 4;
}
```

그림 3-7-2

그림 3-7-1

ii. Instruction Decode

가져온 명령어를 해석하여 어떤 연산이 수행되어야 하는지 파악한다. Instruction의 operands들을 식별하고, 필요한 레지스터 값을 읽어온다. (그림 3-8-2)

```
void F2_Id(unsigned int* inst, ControlSignals* cs, Operands* operands, unsigned int* pc){

    unsigned int funct = 0;
    unsigned int opcode = 0;
    memset(cs, 0, sizeof(ControlSignals)); // set all control signals to 0

    //a) Fetch the instruction
    //By using pointer

    //b) Decode the opcode
    opcode = (*inst & 0xFC000000) >> 26; // 0xFC000000 = 1111 1100 0000 0000 0000 0000 0000 0000
    operands->opcode= opcode;

    if (opcode == 0)
        funct = *inst & 0x3F; // 0x3F = 0011 1111
    else
        funct = ~0;

    //c) Read register operands
    operands->rs = (*inst & 0x03E00000) >> 21;    // 0x03E00000 = 0000 0011 1110 0000 0000 0000 0000 0000
    operands->rt = (*inst & 0x001F0000) >> 16;    // 0x001F0000 = 0000 0000 0001 1111 0000 0000 0000 0000
    operands->rd = (*inst & 0x0000F800) >> 11;    // 0x0000F800 = 0000 0000 0000 0000 1111 1000 0000 0000

    //d) Prepare operands
    operands->funct = (*inst & 0x0000003F);       // 0x0000003F = 0000 0000 0000 0000 0000 0000 0011 1111
    operands->shamt = (*inst & 0x000007C0) >> 6; // 0x000007C0 = 0000 0000 0000 0000 0000 0111 1100 0000
    operands->addr = (*inst & 0x03FFFFFF);        // 0x03FFFFFF = 0000 0011 1111 1111 1111 1111 1111 1111
    operands->imm = (*inst & 0x0000FFFF);         // 0x0000FFFF = 0000 0000 0000 0000 1111 1111 1111 1111

    //sign extension
    unsigned int signBit = (operands->imm >> 15) & 1; // Extract the sign bit
    if (signBit == 1)
    {
        int mask = -1 << 16; // Create a mask with 1s in the higher bits
        operands->imm = operands->imm | mask; // Perform sign extension
    }

    //e) Generate control signals
    control_unit(operands, cs, pc);
}
```

그림 3-8-1

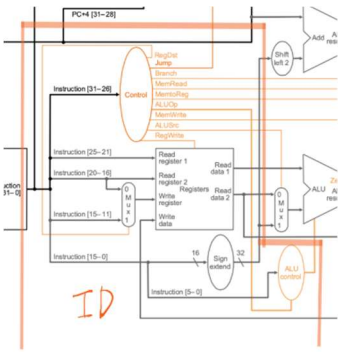


그림 3-8-2

Instruction을 읽어온 뒤 각operand들에 해당하는 비트를 읽어 저장하고 opcode를 분리하여 control unit으로 보내 control signal들을 생성한다. (그림 3-8-1)

iii. Execution

명령어에 따라 적절한 연산을 수행한다. 산술 논리 연산, 데이터 이동, 분기 등이 수행될 수 있다.

```
void F3_Ex(unsigned int* pc, unsigned int* alu_result,unsigned int* reg, ControlSignals* cs, Operands* operands) {

    //a) Calculation
    unsigned int data1 = reg[operands->rs];
    unsigned int data2 = Mux(reg[operands->rt], operands->imm, cs->ALUSrc);

    //Sign Unsigned
    if (operands->funct == 0x21 || operands->funct == 0x23 || operands->funct == 0x2B || operands->opcode == 0x09){}
    else
    {
        (int)data1;
        (int)data2;
    }

    switch (cs->ALUOp){ ... }

    printf("data1: %x     data2:%x\n", data1, data2);
    printf("ALU Result: %x\n", *alu_result);

    //b) Branch Evaluation
    switch (operands->opcode){ ... }
    //printf("%X %X\n", reg[operands->rs], reg[operands->rt]);
    //printf("bcond: %x branch:%d\n", cs->Bcond, cs->Branch);
}
```

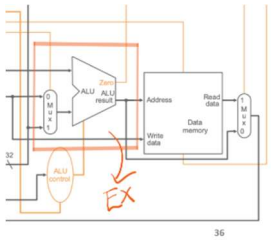


그림 3-9-1

<-그림 3-9-2

읽어온 operand들과 생성한 control signal을 바탕으로 ALU연산에 사용될 데이터를 Mux를 이용하여 선택하고 Instruction에 따른 sign, unsign type을 확인하여 정하고 ALUop signal에 따라서 연산을 진행하게 된다. 또한 Branch에 대한 연산의 결과를 Bcond signal에 보낸다. (그림 3-9-2)

iv. Memory

주로 Load/Store 명령어에서 사용되며, 데이터 메모리에 접근한다. 필요한 데이터를 읽거나 쓰기 위해 메모리를 접근하고 데이터를 주고받는다. (그림 3-10-1)

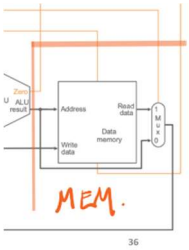


그림 3-10-1

```
void F4_Mem(ControlSignals* cs, Operands* operands,unsigned int* read_data, unsigned char* mem, unsigned int* alu_result, unsigned int* reg) {

    //a) Data Alignment
    if (*alu_result % 4 != 0) {
        //printf("Error: Memory address is not aligned\n");
        return ;
    }

    //big endian
    //b) Load Data
    if (cs->MemRead == 1 && cs->MemWrite == 0) {
        *read_data = (mem[*alu_result] << 24) & 0xFF000000 | (mem[*alu_result + 1] << 16) & 0x00FF0000 | (mem[*alu_result + 2] << 8) & 0x0000FF00 | (mem[*alu_result + 3] << 0) & 0x000000FF;
        printf("addr: %X, data: %X %X %X %X, load data: %X\n", *alu_result, mem[*alu_result], mem[*alu_result + 1], mem[*alu_result + 2], mem[*alu_result + 3], *read_data);
    }

    //c) Store Data
    else if (cs->MemRead == 0 && cs->MemWrite == 1) {
        mem[*alu_result] = (reg[operands->rt] & 0xFF000000) >> 24;
        mem[*alu_result + 1] = (reg[operands->rt] & 0x00FF0000) >> 16;
        mem[*alu_result + 2] = (reg[operands->rt] & 0x0000FF00) >> 8;
        mem[*alu_result + 3] = (reg[operands->rt] & 0x000000FF) >> 0;
        printf("addr: %X, data: %X %X %X %X, stored data: %X\n", *alu_result, mem[*alu_result], mem[*alu_result + 1], mem[*alu_result + 2], mem[*alu_result + 3], reg[operands->rt]);
    }

    else if (cs->MemRead == 0 && cs->MemWrite == 0) {
        //printf("No memory access\n");
    }
    else {
        //printf("Error: MemRead and MemWrite cannot be 1 at the same time\n");
    }
}
```

그림 3-10-2

Opcode에 따라 memory에 접근하여 load, store 중 어떤 일을 할 지 if 문으로 분기한 뒤 load의 경우 1byte씩 나뉘어 있는 값을 하나의 값으로 통합하여 load하고 store의 경우에는 4Byte 값을 1Byte씩 나눠서 각 memory에 Big-endian 방식으로 저장하게 된다.

v. Write Back

실행한 명령어의 결과를 레지스터에 기록한다. ALU 연산 결과나 메모리에서 읽어온 데이터를 register에 저장한다. (그림 3-11-1)

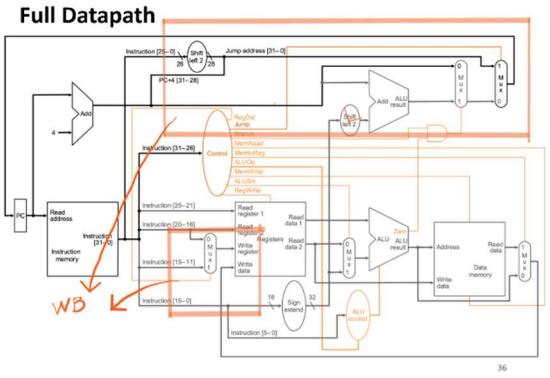


그림 3-11-1

register에 저장될 값이 있는 Instruction의 경우 if 문을 이용하여 register에 데이터를 저장하게 된다. 분기되어 pc의 값을 업데이트 해야 하는 instruction들 또한 if 문을 이용하여 처리해주었다. (그림 3-11-2)

```
void F5_Wb(ControlSignals* cs, Operands* operands, unsigned int* pc, unsigned int* read_data, unsigned int* alu_result, unsigned int* reg) {

    //a) register write
    if (operands->opcode == 0 | operands->opcode == 0x09 | operands->opcode == 0x0A | operands->opcode == 0x23) {
        unsigned int write_data = Mux(*alu_result, *read_data, cs->MemtoReg);
        reg[Mux(operands->rt, operands->rd, cs->RegDst)] = write_data;
        printf("reg[%d] write data: %X\n", Mux(operands->rt, operands->rd, cs->RegDst), write_data);
    }

    //b) Update program counter
    if (operands->opcode == 0x03)//jal
    {
        reg[31] = *pc + 4;
        *pc = operands->addr << 2 | (*pc & 0xF0000000);
    }
    else  if (operands->opcode == 0 && operands->funct == 0x08)//jr
    {
        *pc = reg[operands->rs];
        return;
    }
    else
    {
        unsigned int res = Mux(*pc, ((operands->imm << 2) + *pc), cs->Bcond & cs->Branch);
        *pc = Mux(res, operands->addr << 2 | (*pc & 0xF0000000), cs->Jump);
    }
    printf("Next Pc: %X\n", *pc);
}
```

그림 3-11-2

C. Implementation detail

먼저 Control unit, Mux, ALU 등의 논리회로를 코드로 가져올 때 일부회로만 작성되어 있어서 opcode만 control signal을 generate할 수 있었기에 Karnaugh Map이나 다른 회로이론을 바탕으로 그림3-1-4을 기반이나 다른 진리표를 바탕으로 논리 회로를 작성한 뒤 코드를 작성하면 좋았을 것이다.

그리고 예외 상황들에 대해서 data address에 대한 잘못된 접근을 하는 경우, 구현한 opcode이외의 opcode의 경우 예외 처리하여 사용자가 바로 인식할 수 있도록 하였다.

IV. Execution & Analysis

A. Program build environment & execution.

그림 4-1-1과 같은 개발 환경에서 program을 작성하였다.

그림 4-1-1

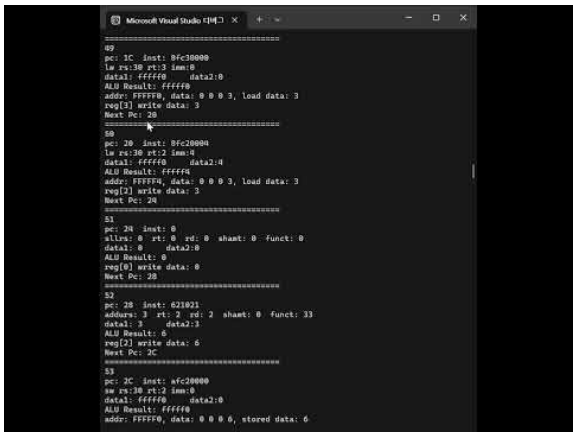
```
// Open the binary file for reading
if (fopen_s(&file, "input1.bin", "rb") != 0 || file == NULL) {
    printf("Error opening file.\n");
    return 1;
}
```

그림 4-11-2에 보라색 네모로 표시된 부분에 읽어올 바이너리 파일의 이름을 입력한 뒤 실행시키면 된다. 그리고 같은 폴더에 있어야 한다.

그림 4-11-2

B. Program result & analysis

i. Input1



영상 4-2-1

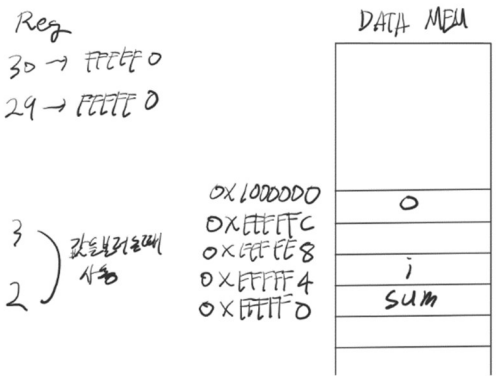
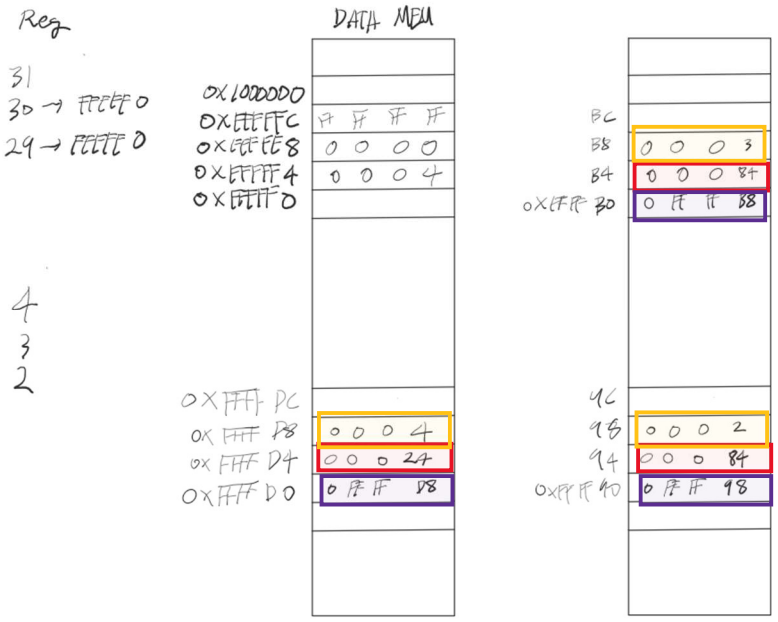


그림 4-2-1

영상 4-2-1을 바탕으로 메모리와 레지스터의 변화를 관찰하면 그림 4-2-1과 같은 구조에서 값들이 변

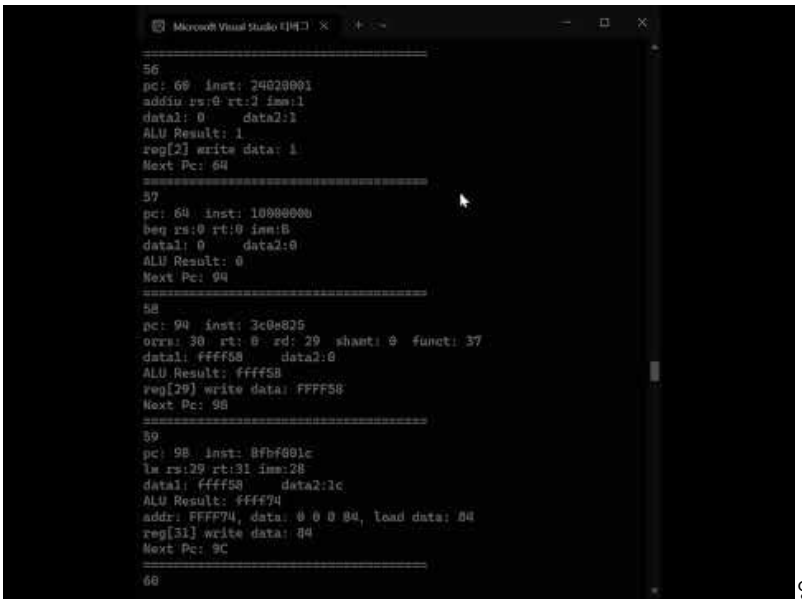
화하는 것을 관찰할 수 있는데 그림과 같이 0XFFFFF0에는 input1의 sum에 해당하는 값을 저장하고 0XFFFFF4에는 for문의 i에 해당하는 값을 저장하여 for문이 종료된 뒤에는 2D가 저장된 것을 확인할 수 있다. 그리고 처음 프로그램을 시작할 때 필요한 만큼의 저장 공간을 확보하는데 이것은 처음 SP를 -16만큼 빼서 변수들을 저장할 공간을 창출하는 것이다. 따라서, 프로그램이 종료되는 시점에는 다시 레지스터와 SP를 원상복구한 뒤 종료하는 것을 실행결과 영상 4-2-1을 통해 알 수 있다.

ii. Input2



Input2는 jal가 있어 Ra register를 활 발하게 사용하며 매 재귀 함수를 실행 할 때마다 SP를 이용하여 Memory에 필요한 만큼 저장 공간을 생성한다 그에 따른 저장 공간을 살펴보면 재귀가 끝나고 memory를 복구할 때 필요한 memory주소(보라색 박스)와 index값(주황색 박스) 그리고 점프할 주소(빨간색 박스)를 저장하는 것을 확인할 수 있다. 그리고 마지막에 재귀 탈출하며 합연산을 실행할 경우 register 2,3,4번을 이용하여 결과를 저장해가며 그 결과는 0XA이다. 이후 종료 프로세스는 Input1과 유사하다.

그림 4-3-1



영상 4-3-1

해당 유튜브 영상들은 제가 녹화하여 유튜브에 올린 후 링크를 건 것으로 두 영상의 상세설명에 제 이름과 학번이 기제되어 있습니다.

V. Lesson

이번 프로젝트를 하면서 MIPS구조와 CPU 아키텍처에 대해 좀 더 자세히 알게 되었다. 저번 학기에 배운 시스템 프로그래밍을 직접 C언어로 구현하고 실행시켜보면서 MIPS Single Cycle이 어떻게 작동하는 지를 배우면서 Single Cycle의 작동 방식과 단점을 명확히 이해할 수 있었다. 왜 Pipeline MIPS가 있는 지 느꼈다. 프로그램이 고급프로그래밍 언어를 실제로 컴퓨터가 작동하기 위해서 명령어를 인출하고 해석하며 실행하고, 메모리와 레지스터 간의 데이터 전송과 결과를 기록하는 과정을 시각화 할 수 있어서 좋았다. 또한, 이 프로젝트를 통해 ISA가 어떻게 프로세서의 동작과 성능에 영향을 미치는지 생각해보고 이해하였다. 요즘 다양한 프로세서들이 출시되고 있는데 사용자의 목적에 알맞은 칩과 그에 따른 ISA의 구성으로 좀 더 최적의 효율과 이익을 가져올 수 있기에 여러 프로세서들이 출시되고 있는 것 같다는 생각이 들었다.

Reference

Microprogram-Control.pdf (torontomu.ca) – PLA 3-1-2

main control truth table | Download Table (researchgate.net) – control signal truth table 3-1-4

(23) CA HW2 과제 input1 출력값 Microsoft Visual Studio 디버그 콘솔 2023 06 02 - YouTube 4-2-1

(23) CA HW2 과제 input2 출력값 Microsoft Visual Studio 디버그 콘솔 2023 06 02 - YouTube 4-3-1