Mobile processor Programming

assignment #3: Simple pipelined MIPS

모바일시스템공학과 김태경 (32211203)

목차

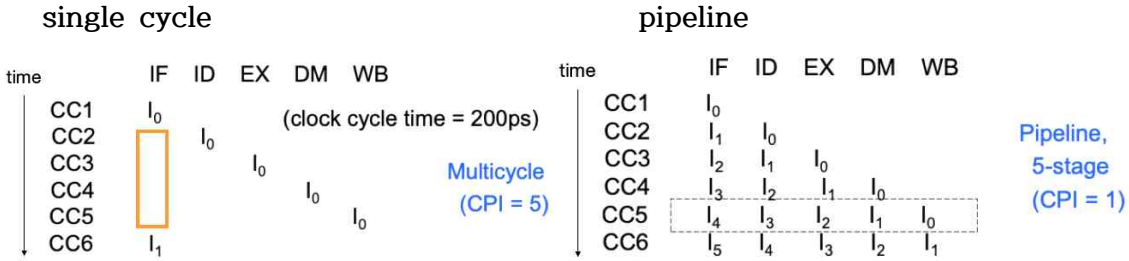
Freeday used 5
Freeday left 0

2024.06.01.

1. 프로젝트 목표

Pipeline에 대해

이전 과제의 single-cycle 설계와 pipeline구조로 설계 간의 성능 비교를 위한 프로젝트로, single-cycle 의 모든 명령어가 하나의 클럭 주기를 거치며, fetch, decode, execue, memory access, write back 총 5단계의 실행 단계를 거치도록 설계하는 방식과 달리, pipeline은 실행 단계를 분리하여 각 단계를 병렬구조로 실행하는 방식으로, 각각 새로운 명령어를 pipeline의 다른 단계에 들어가는 방식으로 대략 single-cycle 대비 5배에 가까운 효율을 기대할 수 있다.



pipeline 설계의 장단점

장점

1. 성능 향상 : 여러 명령어를 동시에 처리할 수 있으므로 처리량이 증가한다.
2. 효율적인 자원 사용 : 각 단계가 동시에 실행되어 자원을 더 효율적으로 사용할 수 있음
3. 확장성 : 각 단계가 독립적이므로 특정 단계를 개선하거나 확장하는 것이 비교적 간단하여, 새로운 명령어나 기능을 추가할 때 유연성이 높다.

단점

1. 복잡성 증가 : single-cycle과 비교하여 상대적으로 pipeline의 설계는 복잡하다.
2. 종속성 처리 : 구조적인 문제로 인한 명령어 간의 종속성을 해결하기 위해 추가적인 hardware와 control logic이 필요하다.

이 프로젝트의 목표는 MIPS pipeline을 이해하고 pipeline의 구현에 필요한 Structural Hazared, data Hazard, Control Hazard를 해결하고, Branch predicrion 방법을 통하여 single-cycle과 CPU의 clock cycle 수를 비교하여 성능비교를 하기 위함이다.

2. 프로젝트 조건

기본요구사항

1. Binary 파일을 입력파일로 받아 프로그램을 실행할 수 있어야 함
    I.실행 전 바이너리 파일의 모든 내용을 입력받아 데이터로 사용
    II. 메모리는 0x40000줄을 가지는 배열으로 각각의 명령어를 저장

2. 레지스터는 각 32bit의 32줄의 배열로 이루어져 있다.
    I. 레지스터는 각 32bit의 32줄의 배열로 이루어져 있으며, 초기 LR(r31)과 SP(r29)의 값은 0xffffffff, 0x10000000으로 세팅

3. MIPS Pipeline의 5 stage 및 Latch의 구현
    I. 각각의 cycle에서는 명령어의 5단계(Fetch, Decode, Execution, Memory Access, Write Back)가 각기 다른 명령어를 차례로 받아 동시에 실행
    II. 각 단계의 명령어 실행 상태를 Latch(임시 저장소)에 저장

4. 데이터 종속성 해결하기
    I. 같은 단계에서 동시에 메모리, 레지스터에 접근하여 발생하는 문제를 해결
    II. Forwarding, Stall의 구현

5. Control 종속성 해결하기 (BEQ, BNE, Jump, JAL, JR)
    I. Jump 또는 Branch 명령어의 실행 시 이전에 로드된 잘못된 명령어를 지우고, 알맞은 주소(pc)값을 Fetch명령어에 입력
    II. Branch의 실행 중 cycle소모를 줄이기 위한 Branch prediction의 구현
    III. 반복(loop)에서 발생하는 cycle소모를 줄이기 위한 이전에 발생한 Jump, Branch 명령어가 발생한 pc, 이동한 address의 기록을 가지는 BTB 생성

6. 프로그램의 종료 조건 설정
    I. PC(Program Counter)의 값이 0xffffffff이 되면 실행을 종료
    II. 최종 반환 값은 v0(또는 r2) 레지스터에 저장

7. 결과창 출력하기
    I. 총 명령어의 수(R_type, I_type, J_type) 출력
    II. Memory Access의 실행 수 출력
    III. Write Back의 실행 수 출력
    IV. Branch의 실행, Branch-taken의 실행 수 출력
    V. Jump의 실행 수 출력

3. 프로젝트에 사용된 개념

Pipeline register (Latch)

각 pipeline의 단계 사이에 데이터를 저장 및 전달하기 위한 역할을 한다.

A. IF/ID Register
- IF(Instruction Fetch) 단계에서 ID(Instruction Decode)단계로 정보를 전달
- 전달 요소: instruction, PC + 4

B. ID/EX Register
- ID (Instruction Decode) 단계에서 EX (Execute) 단계로 정보를 전달한다.
- 전달 요소: PC + 4, Registers, immediate(SignExtend), Control Signals

C. EX/MEM Register
- EX (Execute) 단계에서 MEM (Memory Access) 단계로 정보를 전달한다.
- Memory address, Control Signals, Data to Write

D. MEM/WB Register
- MEM (Memory Access) 단계에서 WB (Write Back) 단계로 정보를 전달한다.
- Data Read, Control Signals

Hazard (Data, Control, Structural)

다음 stage에서 instruction이 실행되는 것을 방해하는 상황으로, 구조적, data, Branch Hazard는 pipeline에서 발생하는 문제이다.

A. Structure Hazard

: 같은 stage에서 자원을 동시에 사용하여 발생하는 충돌으로 ,MIPS pipeline에 Laod Word, Store Word 명령어는 data access를 필요로 하는데, Fetch 단계에서도 명령어를 불러오기 위해 Memory에 접근하여 발생하는 문제이다.

해결방법
- Fetch의 실행을 한 단계 미루는 작업인 Stall으로 Hazard를 해결한다.

B. Data Hazard

: Fetch 되는 명령어가 사용하는 Register의 데이터가 이전 명령어에 의해 아직 계산중 일 경우 발생하는 문제이다.

해결방법
- Forwarding: Execution 단계의 ALU 연산의 결과값을 바로 Decode 단계에 전달하여 알맞은 data를 사용할 수 있도록 한다.

- Load-Use data Hazard: forwarding으로 해결 불가능한 경우인 Memory Access와 memory의 값을 이용한 연산이 필요할 경우에는 불가피 하게 Stall이 한 차례 진행되어야 한다.

C. Control Hazard

: Branch 명령어로 인해 발생하며, Execution 단계에서의 Branch의 여부가 결정되기 전에 잘못된 명령어를 Fetch하는 경우 발생하는 문제이다.

해결방법

- Branch Prediction: Branch 여부를 미리 예측하여 효율을 높인다. 다만, 잘못된 예측으로 stall을 하여 cycle의 소모가 생길 수 있다.

Branch Prediction (자세히)

: 로그램 실행 중의 분기(branch) 명령어가 taken 될지 not taken 될지를 예측하는 기술으로, 예측을 통해 pipeline의 효율을 높이고 불필요한 stall을 줄이는 역할을 한다.

A. Static Branch Prediction

: Branch의 실행 이전에 항상 Branch가 실행되거나, 항상 Branch가 실행되지 않는다고 가정하는 방법으로, 예측이 실패하면 cycle이 버려지는 경우가 발생한다.

- 장점: 구현이 간단하고, Hardware 비용이 낮다.
- 단점: 예측 정확도가 낮아 성능이 낮을 수 있다.

B. Dymamic Branch Prediction

: 실제 Branch 결과를 바탕으로 예측을 조정한다. static 방법보다 더 높은 예측 정확도를 제공한다.

1-Bit Prediction : Shortcoming

: 바로 이전의 결과(1-Bit)을 기반으로 예측하는 경우

-> Branch로 이루어진 다중 loop에서 2번의 예측 실패가 발생한다.

-> 내부 loop의 반복 중 마지막에서 예측실패(Branch taken)이 나타난다.

= 내부 loop는 항상 taken되어 앞으로도 taken 될 것으로 예측한다.

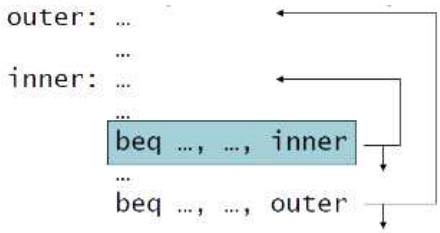
-> 내부 loop의 반복 중 첫 번째에서 예측실패(Branch not taken)이 나타난다.

= 내부 loop의 not taken으로 외부 loop가 실행되어 내부는 not taken된다 예측한다.

2-Bit Predictor

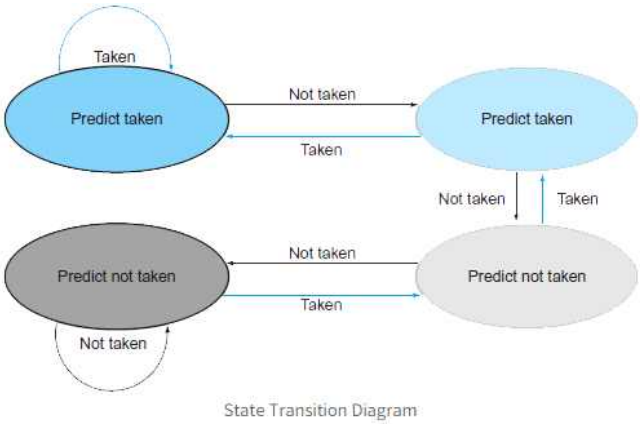
: 2-Bit를 두 번의 연속된 예측 실패가 일어나야만 예측 기준을 바꾸는 방식으로 1-bit predictor의 단점을 보완한다. 다음의 4가지 상태를 가진다.

- Strongly Taken (11): 매우 확실하게 taken으로 예측
- Weakly Taken (10): 약하게 taken으로 예측
- Weakly Not Taken (01): 약하게 not taken으로 예측
- Strongly Not Taken (00): 매우 확실하게 not taken으로 예측



1-bit prediction

2-bit prediction->



State Transition Diagram

BTB (Branch target Buffer)

: Branch와 prediction의 결과를 저장하는 History register를 만들어 명령어가 실행된 pc 주소값과 Branch의 실행여부, Branch가 실행되어 옮겨질 Branch Address 값을 저장하여, 후에 같은 pc주소의 명령어를 실행할 경우 추가적인 cycle의 소모가 없도록 branch prediction의 효율을 높여준다.

Branch Predictor에 따른 효율 비교

- single cycle = 1
- Always Taken Predictor : CPI = [1 + (0.20 * 0.3) * 2] = 1.12
- Always Not Taken Predictor : CPI = [1 + (0.20 * 0.7) * 2] = 1.28
- 1-Bit Counter Predictor : CPI = [1 + (0.20 * 0.15) * 2] = 1.06
- 2BC Predictor : CPI = [1 + (0.20 * 0.10) * 2] = 1.04

CPI(Cycle Per Instruction)= [1 *(branch frequency *misprediction rate) *2] 으로 낮을수록 명령어가 빠르게 처리되는 것을 의미한다.

4. 프로그램 빌드 환경

컴파일 환경: Microsoft Visual Studio
프로젝트 언어: C언어
실행 파일: pipeline.c / simple , 2, 3, 4, 5, gcd, fib, input.bin

5. 프로그램 구성

1) 글로벌 변수 선언 및 초기화

```
int Memory[0x400000];
int Regs[32] = { 0, };
unsigned int PC = 0;
unsigned int nPC = 0;

int Forward[3] = { 0 };
int Forward_val[3] = { 0 };

int ctrl_dep_Jump = 0;
int ctrl_dep_Branch = 0;
int BTB_ex = 0;
int BTB_I = 0;
int BTB[0x100000][3] = { 0 };
Latch latch[9] = { 0 };
Count_num count_num = { 0 };
```

- BTB_ex : 현 pc주소가 BTB기록에 있는지 판별
- BTB_I : 현 pc주소가 BTB의 몇 번째 index인지 확인하는 변수
- BTB : 이중 배열 구조로 각각 Banch, Jump가 실행되는 명령어의 pc주소값, 이동하는 pc Address값, Branch의 실행여부를 저장함

- Latch : 명령어의 단계별로 명령어 요소 contrl signal 요소 등등을 저장하는 구조체 배열
- Count_num: 명령어의 실행을 기록하는 구조체

2) 초기화 함수 선언

```
void Regs_Set(int Regs[])
{
    for (int n = 0; n < 32; n++)
        Regs[n] = 0;
    Regs[29] = 0x100000;
    Regs[31] = 0xffffffff;
    return;
}
```

void Regs_Set(int Regs[])
: 레지스터의 값을 0으로 초기화하고, Regs[29]는 0x100000, Regs[31]은 0xffffffff으로 초기화하는 함수

```
void Latch_Set(Latch* latch)
{
    latch->pc = 0;

    latch->instruction = 0;
    latch->opcode = 0;
```

void Latch_Set(Latch* latch)
: latch 구조체의 요소들을 전부 0으로 초기화 하는 함수

```
void Inst_Set() // &latch[1]로 함수 호출
{
    latch[3].opcode = (latch[3].instruction >> 26) & 0x3f;
    latch[3].rs = (latch[3].instruction >> 21) & 0x1f;
    latch[3].rt = (latch[3].instruction >> 15) & 0x1f;
    latch[3].rd = (latch[3].instruction >> 11) & 0x1f;
    latch[3].shamt = (latch[3].instruction >> 6) & 0x1f;
    latch[3].funct = latch[3].instruction & 0x0000003f;
    latch[3].immediate = latch[3].instruction & 0x0000ffff;
    latch[3].address = latch[3].instruction & 0x03ffffff;
    latch[3].SignExtImm = (latch[3].immediate & 0x8000) ? (latc
    latch[3].ZeroExtImm = latch[3].SignExtImm & 0x0000ffff;
    latch[3].BranchAddr = latch[3].SignExtImm * 4;
    latch[3].JumpAddr = ((latch[3].pc + 4) & 0xf0000000) | (lat
}
```

void Inst_Set()
: MIPS Green sheet를 기반으로 instruction을 해석하여 IF_ID output 단계의 latch의 요소에 opcode, rs, rt, rd, immediate 등의 알맞은 값을 입력해준다.

```
void Latch_copy(Latch* input, Latch* output)
{
    memcpy(output, input, sizeof(Latch));
}
```

void Latch_copy(Latch* input, Latch* output)
: Latch 간의 복사를 위한 함수로 input의 요소들을 ouput의 요소에 복사한다.

3) pipeline 구현 보조 함수 선언

```
int Mux(int input1, int input2, int signal)
{
    if (signal == 1)
        return input2;
    else
        return input1;
}
```

int Mux(int input1, int input2, int signal)
: signal값에 따라 input1 또는 inpu2를 반환한다.

```
void Regs_value(int Read_regs1, int Read_regs2, int Write_regs)
{
    latch[3].Read_register1 = Read_regs1;
    latch[3].Read_register2 = Read_regs2;
    latch[3].Write_register = Write_regs;   // 적을 Regs 주소 저장
    latch[3].Read_data1 = Regs[latch[3].Read_register1];
    latch[3].Read_data2 = Regs[latch[3].Read_register2];
    return;
}   // latch[3] == ID_output latch
```

void Regs_value(int Read_regs1, int Read_regs2, int Write_regs)
: 입력된 매개변수를 이용하여 ID_EX의 output latch의 Read data, write register 값들을 입력해준다.

```
void Count_RIJ_type()   // RIJ_type Count
{
    if (latch[3].instruction == 0)
    {
        count_num.Nop_num++;    // Nop count
    }
    else if (latch[3].opcode == 0)
    {
        count_num.R_type_num++; // R-type count
```

void Count_RIJ_type()
: instruction을 해석하여 각각 Nop, R_type, I_type, J_type 명령어인지 구분하여 Count_num구조체에 저장한다.

```
void Ctrl_signal_Set()
{
    if (latch[3].opcode == 0x0)
    {
        latch[3].RegDst = 1;
        latch[3].ALUop = 1;
    }   // R-type inst
```

void Ctrl_signal_Set()
: latch에 입력된 instruction에 따라 Contrl signal을 입력한다.

```
void ALU_Ctrl_signal(int opcode, int funct)
{
    if (latch[3].ALUop == 1)
    {
        switch (latch[3].funct)
        {
        case 0x20:  // Add
        case 0x21:  // AddU
            latch[3].ALU_ctrl_num = 1;  // +
            return;
```

void ALU_Ctrl_signal(int opcode, int funct)
: 매개변수에 입력된 opcode, funct 값에 따라 실행되는 ALU 연산을 구분하여 latch의 ALU_ctrl_num에 저장한다.

```
void ALU(int input1, int input2, int ALU_ctrl_num)
{
    switch (ALU_ctrl_num)
    {
    case 0:
        latch[3].ALU_result = 0;
        return;
    case 1:
        latch[3].ALU_result = input1 + input2;
        return;
```

void ALU(int input1, int input2, int ALU_ctrl_num)

: 매개변수에 입력된 input1, input2, ALU_ctrl_num에 따라 알맞은 연산을 실행하여 EX_MEM 단계의 ouput latch의 ALU_result 값을 입력한다.

+ 연산 중 Branch의 실행 여부에 따라 Count_num 구조체의 값을 수정한다.

4) pipeline Hazard 해결 함수 선언

void Forwarding_unit_in(Latch* Latch, int i)

: 매개변수에 입력된 I와 latch의 ctrl signal 요소인 RegWrite, Jump요소에 따라 Forwarding을 진행한다.

```
int F_mux(int r, int data)
{
    for (int i = 0; i < 3; i++) {
        if (Forward[i] == r) {
            return Forward_val[i];
        }
    }
    return data;
}
```

int F_mux(int r, int data)

: forwarding을 하는 과정 중 Execution 단계의 ALU연산에 input되는 값들을 선택하는 함수

```
int BTB_detection()
{
    int i = 1;
    BTB[1][0] = 1;
    for (i = 1; i < 0x100000; i++)
    {
        if (latch[1].pc == BTB[i][0])
        {
            latch[1].pc = BTB[i][1];
            nPC = latch[1].pc;

            BTB_ex = 1;
            return i;
        }
        else
        {
            BTB_ex = 0;
            return 0;
        }
    }
    return 0;
}
```

int BTB_detection()

: BTB에 기록된 pc값과 현재 실행할 pc값을 비교하여 같을 경우 latch의 pc값을 기록된 pc가 이동하는 주소값으로 변경하여 그에 알맞은 명령어를 불러오도록 한다.

```
void BTB_Insert(unsigned int pc, int Address, int taken)
{
    for (int i = 1; i < 0x100000; i++)
    {
        if (BTB[i][0] == pc)
        {
            return;
        }
        else if (BTB[i][0] == 0)
        {
            BTB[i][0] = pc;         // pc값 입력
            BTB[i][1] = Address;    // address값 입력
            BTB[i][2] = taken;      // 1=jump, 2=not taken, 3=taken
        }
    }
}
```

void BTB_Insert(unsigned int pc, int Address, int taken)

: 매개변수의 입력값에 따라 BTB배열에 Jump 또는 Branch가 실행될 경우 실행된 pc값, 이동한 주소값, Branch의 실행 여부를 저장하는 함수이다.

5) Instructino_level 구현

void Fetch()

```
void Fetch()    // latch[1] = IF_ouput_latch
{
    latch[1].pc = nPC;
    latch[1].instruction = Memory[nPC / 4];

    if ((latch[5].opcode == 0x0) && (latch[5].funct == 0x08))
    {
        nPC = latch[5].Read_data1;
    }   // JR은 rs를 기반으로 하니 BTB 기록X
    else
    {
        nPC = Mux(Mux(latch[1].pc + 4, (latch[5].BranchAddr + latch[5].pc + 4)
    }   // Jump, Branch에 따른 => 주소값, else => pc + 4값 반환

    // 기록에 있으면 nPC값 덮어씌우기
    BTB_I = BTB_detection();

    if (BTB_ex == 1);   // BTB에 기록이 있음
    else if (BTB_ex == 0)   // BTB에 기록이 없음 / 위의 nPC값에 따름
    {
        if ((ctrl_dep_Jump > 0) || (ctrl_dep_Branch > 0))
        {
            Latch_Set(&latch[1]);
            if (nPC == 0xffffffff) // 넘겨와 받은 값이 end일 경우 확인
            {
                PC = nPC;   // PC값을 WB에서 확인하고 while종료
                return;
            }
        }   // Jump = 1 or Branch = 1~3
        else; // Branch not taken & else
    }
    // 바뀐 pc값, 명령어 다시 입력해주기
    latch[1].pc = nPC;
    latch[1].instruction = Memory[nPC / 4];
}
```

- 현재 PC 값을 기반으로 명령어를 가져온다
- JR 명령어의 경우, 레지스터 값을 새로운 PC로 설정한다.
- Jump나 Branch 명령어에 따라 다음 명령어 주소를 계산한다.
- BTB를 사용해 기록된 분기 주소를 탐색하고, 필요시 새로운 PC를 설정한다.
- 계산된 PC 값과 명령어를 latch[1]에 저장하여 다음 사이클에서 사용하도록 한다.

void Decode()

```
void Decode()
{
    Latch_copy(&latch[2], &latch[3]);

    if (ctrl_dep_Jump > 0)  // Jump 시
    {
        Latch_Set(&latch[3]);
        ctrl_dep_Jump--;
    }
    if (ctrl_dep_Branch > 0)// Branch 시
    {
        Latch_Set(&latch[3]);
    }
    else;                   // 예외처리
    Inst_set();
    Count_RIJ_type();
    ALU_Ctrl_signal(latch[3].opcode, latch[3].funct);
    Ctrl_signal_Set();

    if (BTB_ex == 1)        // BTB 기록 O
    {
        // ID에서 실행해 Ex에 들어가 Branch, Jump를 실행방지
        if (BTB[BTB_i][2] == 1)         // jump의 기록
        {
            latch[3].Jump = 0;
        }
        else;
    }
    else if (BTB_ex == 0)   // BTB 기록 X
    {
        // Jump
        if (latch[3].Jump == 1)
        {
            ctrl_dep_Jump = 1;
            if (latch[3].RegDst == 1);
            else
            {
                BTB_Insert(latch[3].pc, latch[3].JumpAddr, 1);
            }
        }
    }
    // imm = signExtimm 처리
    if ((latch[3].opcode == 0xc) || (latch[3].opcode == 0xd) || (latch[3].opcode == 0xf));
    // Andi, LUI, ORI 제외
    else
    {
        latch[3].immediate = latch[3].SignExtImm;
    }

    // Reg_value 입력
    if (latch[3].opcode == 0x3)
    {
        Regs_value(latch[3].rs, latch[3].rt, 31);
    }   // JAL
    // imm~ 등은 Reg를 통하지 않아서 제외
    else
    {
        Regs_value(latch[3].rs, latch[3].rt, Mux(latch[3].rt, latch[3].rd, latch[3].RegDst));
    }   // else
}
```

- Latch의 값 이동간 소실을 방지한 복사
- Jump 및 Branch 명령어 처리

- ctrl_dep값에 따른 latch 초기화

- 명령어 설정
- R, I, J type별 카운트
- ALU 제어 신호 설정
- 기타 제어 신호 설정

- BTB기록이 있을 경우 jump ctrl_signal을 초기화하여 중복실행방지
- BTB기록이 없을 경우 중 jump의 실행에 따른 값을 BTB에 저장한다.

- Execution단계에 필요한 피연산자 값들을 Reg_value함수로 지정한다.

void Execution()

```
void Execution()
{
    Latch_copy(&latch[4], &latch[5]);

    // jump는 1번만 버블
    if (ctrl_dep_Branch > 0)// Branch 시
    {
        Latch_Set(&latch[5]);
        ctrl_dep_Branch--;
    }
    else;                   // 예외처리
// forwarding 해서 불러온 값 F_Mux로 재정의
latch[5].Read_data1 = F_mux(latch[5].Read_register1, latch[5].Read_data1);
latch[5].Read_data2 = F_mux(latch[5].Read_register2, latch[5].Read_data2);
```

- Latch의 값 이동간 소실을 방지한 복사한다.

- ctrl_dep값에 따른 Latch 초기화한다.

ALU 연산 실행

- rs와 ZeroExtimm을 사용하는 AndI, ORI를 구분하여 ALU 실행한다.
- rt와 shamt를 사용하는 SLL, SRL을 구분하여 ALU 실행한다.
- Mux에 ALUSrc control signal값을 기준으로 Read_data2, 또는 immediate값 중 선택하여 ALU를 실행한다.

```
if (BTB_ex == 1)        // 기존의 BTB 기록 O
{///
    if (BTB[BTB_I][2] == 2) // Branch not taken 기록
    {
        latch[5].Branch = 0;
        count_num.Branch_predict_O++;
        return;
    }
    else if (BTB[BTB_I][2] == 3)    // Branch taken 기록
    {
        latch[5].Branch = 0;
        count_num.Branch_predict_X++;
        return;
    }
}
else if (BTB_ex == 0)   // 기존의 BTB 기록 X
{
    // Branch
    if ((latch[5].Branch && latch[5].ALU_result) == 1)
    {
        ctrl_dep_Branch = 3;
        BTB_Insert(latch[5].pc, (latch[5].JumpAddr + latch[5].pc + 4), 3);
        count_num.Branch_predict_X++;
    }
    else if ((latch[5].Branch && latch[5].ALU_result) != 1)
    {
        ctrl_dep_Branch = 0;
        BTB_Insert(latch[5].pc, latch[5].pc + 4, 2);
        count_num.Branch_predict_O++;
    }
}
    }
   }
   return;
}
```

- BTB에 기록이 있을 경우 BTB기록에 따른 Count_num 구조체의 Branch predict 값을 수정한다.

- BTB에 기록이 없을 경우 Execution결과에 따라 ctrl_dep값의 설정과 Count_num 구조체의 Branch predict 값을 수정, Branch의 실행결과를 BTB에 저장한다.

void Memory_Access()

```
void Memory_Access()
{
    Latch_copy(&latch[6], &latch[7]);

    latch[7].Address = latch[7].ALU_result;
    latch[7].M_Write_data = latch[7].Read_data2;

    if ((latch[7].MemWrite == 1) || (latch[7].MemRead == 1))
    {
        count_num.MEM_Access_num++;;

        if ((latch[7].MemWrite == 1) && (latch[7].opcode == 0x2b))
        {
            latch[7].Read_data = 0;
            Memory[(latch[7].Address / 4) - 1] = latch[7].M_Write_data;
            return;
        }   // SW
        if ((latch[7].MemRead == 1) && (latch[7].opcode == 0x23))
        {
            latch[7].Read_data = Memory[(latch[7].Address / 4) - 1];
            return;
        }   // LW
    }
    else
    {
        latch[7].Read_data = 0;
    }
}
```

- Latch의 값 이동간 소실을 방지한 복사한다.

- latch의 요소인 address와 M_read_data요소에 ALU_result, Read_data2값을 입력한다.

- 명령어에 맞는 memory 수정을 실행한다.

void Write_Back()

```
void Write_Back()
{
    latch[8].R_Write_data = Mux(latch[8].ALU_result, latch[8].Read_data, latch[8].Mem2Regs);
    if (latch[8].RegWrite == 1) // Not SW, BOE, BNE, Jump, JR
    {
        if (latch[8].opcode == 0x3)
        {
            Regs[31] = latch[8].pc + 8;
        }   // JAL 조건
        else
        {
            Regs[latch[8].Write_register] = latch[8].R_Write_data;
        }   // Not SW, BOE, BNE, Jump, JR인 나머지
    }
}
```

- latch의 레지스터에 반환할 값인 R_Write_data의 값을 Mux에 Mem2Reg contrl signal 값에 따라 ALU_result 또는 Read_data의 값을 입력한다.
- 명령어가 jR일 경우 레지스터 31번의 값에 pc+8의 값을 업데이트 하고, 아닐 경우에는 레지스터의 Write_register번째 index에 R_Write_data를 입력한다.

6) main 보조 함수.

void Get_Instruction()

```
void Get_Instruction()
{
    FILE* fp = NULL;
    fp = fopen("simple.bin", "rb");
    if (fp == NULL)
    {
        perror("file open error");
    }

    int i = 0;
    int var = 0;

    while (1)
    {
        int ret = fread(&var, sizeof(var), 1, fp);
        if (ret == 0) break;

        unsigned int b1, b2, b3, b4;
        b1 = var & 0x000000ff;
        b2 = var & 0xff00;
        b3 = var & 0x00ff0000;
        b4 = (var >> 24) & 0xff;
        int res = (b1 << 24) | (b2 << 8) | (b3 >> 8) | b4;

        int instruction = res;
        Memory[i] = instruction;
        i++;
    }
    fclose(fp);
    return;
}
```

- 입력파일의 내용을 받아 Memory에 입력한다.

7) main 함수

```
int main(int argc, char* argv[])
{
    Get_Instruction();

    while (PC != 0xffffffff)
    {
        Fetch();
        Decode();
        Execution();
        Memory_Access();
        Write_Back();

        for (int i = 1; i < 5; i++)
        {
            Latch_copy(&latch[(2 * i) - 1], &latch[2 * i]);
        }

        count_num.cycle_num++;
    }
    Print_result();
}
```

- 입력파일을 내용을 받고, memory에 저장한다.

- PC가 0xffffffff가 되는 while으로 instrcution_level의 각 단계를 실행한다.

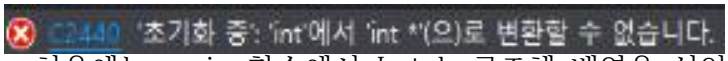
- 각 단계들의 실행이 끝나면, latch ouput의 요소 값들을 다음 stage의 latch input의 요소에 입력한다.

- cycle의 카운트
- count_num의 결과 출력

6. 문제점 및 해결방법

// 현재 작성한 코드는 작동하지 않습니다.

2중 포인터 문제



처음에는 main 함수에서 Latch 구조체 배열을 선언하고, 이를 각 함수에 매개변수로 전달하여 Latch의 요소 값을 수정하는 방식을 구상하였다. 그러나 함수에 Latch 구조체 배열을 이중 포인터로 전달하고, 배열의 요소 값을 수정하는 과정에서 이중 포인터 사용에 익숙하지 않아 함수 호출과 요소 값 수정이 원활하게 작동하지 않았다. 정확한 원인은 이중 포인터를 매개변수로 받아 배열의 요소를 수정하려 했으나, 포인터의 참조와 역참조 과정에서 예상치 못한 동작이 발생했다. 이러한 문제로 인해, 배열의 요소값을 원하는 대로 변경하는 데 어려움을 겪었다.

1차 대안으로 main 함수 내에 Latch 구조체 배열을 선언하는 대신, Latch 구조체 배열을 글로벌 변수로 선언하였다. 이를 통해 함수에서 Latch 구조체 배열을 매개변수로 전달하지 않고도 배열의 요소값에 접근하고 수정할 수 있게 되었다.

그리고 함수에 Latch 구조체 배열을 매개변수로 전달하지 않고, 글로벌 변수로 선언된 Latch 구조체 배열에 직접 접근하여 요소값을 변경하는 방식을 채택하였다. 이를 통해 이중 포인터 사용으로 인한 문제를 회피하고, 배열의 요소값을 쉽게 수정할 수 있게 되었다.

Control Dependency 문제

Jump와 Branch의 Control Dependency 문제를 해결하기 위해 이전에 실행된 기록이 있어야하는데, 이를 위해 PC 값의 재설정이나 latch의 초기화 등의 작업을 하나로 묶어 구현하려 했으나, 정확한 단계적인 흐름을 적절하게 분리하지 못하여 Control Dependency 문제를 해결하지 못했다.

해결방법

단계적인 흐름 분리:

Jump와 Branch의 실행에 따른 제어 흐름을 명확하게 분리하여 각각의 기능을 단계적으로 수행하였고, Jump와 Branch 명령어의 경우 조건을 평가하여 PC 값을 수정하는 과정을 각각 분리하여 구현하였다.

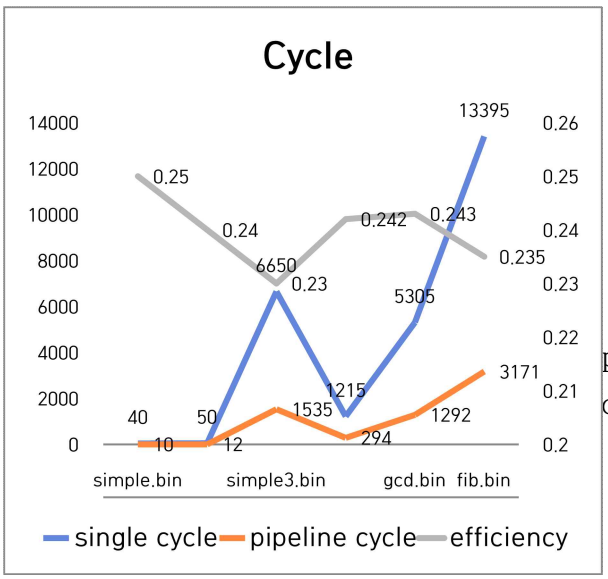
PC 값의 재설정과 Latch 초기화 등의 작업들 또한 별도의 단계로 분리하여 구현하였습니다. 이를 통해 Jump 명령어의 실행에 따른 PC 값의 재설정과 Jump 및 Branch 명령어의 실행에 따라 필요한 작업들을 명확하게 처리할 수 있었습니다. 이러한 변경으로 Jump와 Branch의 Control Dependency 문제를 효과적으로 해결하고, 올바른 실행 조건을 설정할 수 있었다.

// 실제로 올바른 해결을 하였는지는 확인하지 못했습니다.

7. 실행결과

// 작성한 코드가 작동하지 않아 아래의 **참고자료** 결과를 참고하여 작성하였습니다.

single-cycle과 pipeline의 cycle 효율 비교



	single	pipeline	efficiency
simple.bin	40	10	0.25%
simple2.bin	50	12	0.24%
simple3.bin	6650	1347	0.23%
simple4.bin	1215	290	0.24%
gcd.bin	5305	1269	0.24%
fib.bin	13395	3232	0.23%

이 자료는 single cycle과 pipeline을 구현하여 실행한 결과의 cycle 수를 비교한 그래프와 표이다.

앞에서 예상하였던 1/5배의 cycle 소비율을 기대하였지만, 구현된 pipeline의 cycle 소비는 약 24%으로 예상보다 조금 높았다. 이는 pipeline의 구현에서 발생하는 문제인 Hazard를 해결하기 위해 소모되는 cycle의 수가 영향을 미친 것이다. 이러한 추가적으로 소모되는 cycle을 감안하더라도 pipeline의 구조가 single cycle 대비 효율적이라는 것을 알 수 있다.

Branch Prediction의 구현에 따른 효율 비교

	instruction	Branch Inst	always not taken	always taken	1-bit counter	2-bit counter
simple3.bin	1330	102	101	1	2	2
simple4.bin	243	10	9	1	2	2
gcd.bin	1061	73	45	28	15	15
fib.bin	2679	109	54	55	42	54
		CPI	1.088	1.026	1.020	1.022

위 표는 파일별 Branch의 예측 실패한 횟수를 보여주는 표인다.

Branch의 Predictor에서 Always not taken이 상대적으로 가장 많은 예측 오류를 발생시켰으며, 가장 높은 CPI를 보여 가장 비효율적인 Predictor라는 것을 확인할 수 있었다. 그리고 위 프로그램들을 가장 효율적으로 작동시키는 Predictor는 1-bit counter이다.

//simple.bin과 simple2.bin의 경우는 Branch 명령어가 존재하지 않아 제외하였다.

8. 느낀점

과제 중 가장 이해하기 힘들었던 부분은 데이터 해저드(Data Hazard)의 해결을 위한 Forwarding과 Stall의 구현과 제어 해저드(Control Hazard)의 해결 방법을 구현하는 것이었다. 하지만, 과제 중 가장 당황스러운 부분은 2중 포인터를 활용하는 부분이었다. 함수에 latch를 매개변수로 받으면서 포인터로 불러와야 하는지, 직접적으로 불러와야 하는지에 대해 고민이 많았다. main 함수에서 1중으로 호출되었을 때는 큰 문제 없이 작동했지만, 1중으로 불러진 함수 내에서 latch를 2중 포인터로 불러오는 과정에서 함수가 제대로 작동하지 않는 문제가 발생했다. 포인터 활용에 대한 이해가 부족하여 생긴 문제라는 것을 알게 되어 더 답답했다. 인터넷으로 검색해 보아도 구조체 배열을 오류 없이 함수의 매개변수로 사용하는 방법에 대해 명확히 파악하지 못했다.

다른 대안으로 latch를 전역 변수로 선언하고 함수에 매개변수를 넣는 대신 글로벌 변수의 값을 직접 변경하는 방법을 택하여 코드를 다시 작성했다. 하지만 이 방법에서도 latch의 instruction 요소인 opcode, rs, rt, rd 등의 값이 제대로 업데이트되지 않아 결국 프로그램이 제대로 작동하지 않는 상태로 코딩이 마무리되었다.

실행 결과를 알 수 없어 보고서를 마무리하려 했으나, 선배들에게서 이전에 했던 자료를 받아 실행 결과를 가지고 single cycle과 pipeline의 효율 및 predictor별 CPI 효율을 비교해볼 수 있었다. 교수님의 자료에 있던 통상적인 predictor별 CPI 효율과 비교해 보니, 실제 결과에서 상당히 낮은 CPI 값이 나와 당황했다. 특히 1-bit Counter가 2-bit Counter보다 CPI가 낮은 것을 보고, 단순히 프로그램의 구조와 실행 흐름에 따라 다르게 나타날 수 있다는 것을 깨달았다.

9. 참고자료

- 프로그래밍으로 구현하지 못하여 실행 후 결과분석을 위해 실행결과를 참고하였습니다.
https://github.com/LeeChangYoon/Computer-Architecture/tree/main/Project3%20-%20Pipelined%20MIPS%20Simulator
- Dynamic Branch Prediction의 구조 및 설명사진을 보고서 작성에 사용하였습니다.
https://ydeer.tistory.com/143
- Hazard 구현에 필요한 이론을 참고하였습니다.
https://one2bla.me/cs6290/lesson4/2-bit-predictor.html
- https://luv-n-interest.tistory.com/995