REPORT

HW 3

과목명 : 컴퓨터구조론 (2분반)
교 수 : 남재현 교수님
소속학과: 소프트웨어학과
학 번 : 32190192
이 름 : 구영민
마 감 : 2024.06.18 (화요일까지)

단국대학교 DANKOOK UNIVERSITY

목 차

단국대학교 DANKOOK UNIVERSITY

사진 목차

1. Instruction

a) 과제 목표

이번 과제는 MIPS 아키텍처 기반의 파이프라인 프로세서를 시뮬레이션하는 프로그램을 구현하는 것입니다. 주요 목표는 파이프라인의 각 단계를 처리하고, 데이터 해저드와 분기 예측 문제를 해결하는 메커니즘을 구현하여 파이프라인의 성능을 최적화하는 것입니다.

포워딩 로직 구현

데이터 해저드 발생 시 필요한 데이터를 즉시 사용할 수 있도록 포워딩 메커니즘을 구현합니다.

Static Prediction 추가

분기 예측을 통해 파이프라인의 Always Taken 방식의 분기 예측을 구현합니다.

레지스터 버퍼 추가

파이프라인 각 단계 사이의 데이터를 저장하는 레지스터 버퍼를 정의하고 사용합니다.

버블 처리 로직 구현

데이터 해저드가 발생했을 때, 이를 해결하기 위해 파이프라인에 버블을 삽입하는 로직을 구현합니다.

2. Background

a) 요구사항 정리

1. 포워딩 로직 구현

데이터 해저드를 최소화하기 위해 포워딩 메커니즘을 구현합니다.

- EX 단계에서의 포워딩: ALU 연산 결과를 다음 명령어의 입력으로 즉시 사용할 수 있도록 합니다.
- MEM 단계에서의 포워딩: 메모리 접근 결과를 다음 명령어의 입력으로 즉시 사용할 수 있도록 합니다.

2. Static Prediction 추가

Always Taken 방식의 분기 예측을 통해 파이프라인의 성능 저하를 줄임

- 분기 명령어(beq, bne 등)를 감지하여 Always Taken 방식으로 분기를 예측합니다.
- 분기 예측 결과에 따라 프로그램 카운터(PC)를 업데이트 합니다.
-

3. 레지스터 버퍼 추가

파이프라인 각 단계 사이의 데이터를 저장하는 레지스터 버퍼를 정의하고 사용합니다.

- IF/ID 레지스터: Instruction Fetch와 Instruction Decode 사이의 데이터 저장합니다.
- ID/EX 레지스터: Instruction Decode와 Execute 사이의 데이터 저장합니다.
- EX/MEM 레지스터: Execute와 Memory Access 사이의 데이터 저장합니다.
- MEM/WB 레지스터: Memory Access와 Write Back 사이의 데이터 저장합니다.

4. 버블 처리 로직 구현

데이터 해저드 발생 시 파이프라인의 일부 단계를 정지시키고, NOP(No Operation)를 삽입하여 해결합니다.

- 데이터 해저드를 감지하고, 필요한 경우 파이프라인에 버블을 삽입합니다.
- IF/ID, ID/EX, EX/MEM, MEM/WB 등의 파이프라인 레지스터의 유효성(valid)을 설정하여 버블을 처리합니다.

5. 전체 파이프라인 설계

파이프라인의 각 단계를 처리하는 로직을 구현하여 명령어가 올바르게 실행되도록 합니다.

- Instruction Fetch 단계: 메모리에서 명령어를 가져와 IF/ID 레지스터에 저장합니다.
- Instruction Decode 단계: 명령어를 해독하고, 필요한 레지스터 값을 읽어 ID/EX 레지스터에 저장합니다.
- Execute 단계: ALU 연산을 수행하고 결과를 EX/MEM 레지스터에 저장합니다.
- Memory Access 단계: 메모리 접근(읽기/쓰기) 작업을 수행하고 결과를 MEM/WB 레지스터에 저장합니다.
- Write Back 단계: 연산 결과를 레지스터 파일에 저장합니다.

b) Testcase에서 hazard 발생 위치 조사

구현 전 테스트를 위해 헤저드 발생 위치를 모두 파악해보았습니다

Testcase1

```
1c: 8fc30000    lw v1,0(s8)
20: 8fc20004    lw v0,4(s8)
```

[사진 1] Testcase 1-1

- 8fc20004 lw v0, 4(s8) 명령어는 lw v1, 0(s8) 명령어의 결과를 사용하지 않지만, 이후에 사용될 수 있으므로 Load-use Hazard가 발생할 가능성이 있습니다.

```
1c: 8fc30000    lw v1,0(s8)
20: 8fc20004    lw v0,4(s8)
24: 00000000    nop
28: 00621021    addu  v0,v1,v0
```

[사진 2] Testcase 1-2

- addu v0, v1, v0 명령어는 lw v1, 0(s8) 명령어의 결과를 사용합니다. 이 경우 v1 레지

스터에 로드된 값이 addu 명령어에서 사용되기 전에 메모리 접근이 완료되어야 합니다.

- Load-use Hazard가 발생할 가능성이 있습니다.

```
28: 00621021    addu  v0,v1,v0
2c: afc20000    sw v0,0(s8)
```

[사진 3] Testcase 1-3

- sw v0, 0(s8) 명령어는 addu v0, v1, v0 명령어의 결과를 메모리에 저장합니다.
- v0 레지스터에 저장된 값이 sw 명령어에서 사용되기 전에 연산이 완료되어야 합니다.
- Load-write Hazard가 발생할 가능성이 있습니다.

```
54: 8fc20000    lw v0,0(s8)
58: 03c0e825    move  sp,s8
```

[사진 4] Testcase 1-4

- move sp, s8 명령어는 lw v0, 0(s8) 명령어의 결과와 관련된 데이터 해저드가 발생할 수 있습니다.

```
48: 2842000a    slti  v0,v0,10
4c: 1440fff3    bnez  v0,1c <main+0x1c>
```

[사진 5] Testcase 1-5

- bnez v0, 1c <main+0x1c> 명령어는 slti v0, v0, 10 명령어의 결과를 사용합니다. 이 경우 분기 명령어가 실행될 때까지 v0 레지스터의 값이 유효해야 합니다.

Testcase2

```
10: 24020004  li  v0,4
14: afc2001c  sw  v0,28(s8)
18: 8fc4001c  lw  a0,28(s8)
```

[사진 6] Testcase 2-1

- sw v0, 28(s8) 명령어는 v0 레지스터에 저장된 값을 메모리에 씁니다.
- lw a0, 28(s8) 명령어는 동일한 메모리 위치에서 값을 읽습니다.
- v0 레지스터의 값이 메모리에 저장된 후에 메모리에서 읽어야 하므로 sw와 lw 명령어 사이에 데이터 해저드가 발생할 수 있습니다.

```
50: 8fc30020  lw  v1,32(s8)
54: 24020001  li  v0,1
58: 14620004  bne v1,v0,6c <foo+0x30>
```

[사진 7] Testcase 2-2

- lw v1, 32(s8) 명령어는 v1 레지스터에 값을 로드합니다.

- bne v1, v0, 6c <foo+0x30> 명령어는 v1 레지스터의 값을 사용하여 분기 여부를 결정합니다.
- v1 레지스터에 값이 로드되기 전에 bne 명령어가 실행되면 데이터 해저드가 발생할 수 있습니다.

```
74: 2442ffff  addiu v0,v0,-1
78: 00402025  move  a0,v0
```

[사진 8] Testcase 2-3

- addiu v0, v0, -1 명령어는 v0 레지스터의 값을 갱신합니다.
- move a0, v0 명령어는 갱신된 v0 레지스터의 값을 사용합니다.
- v0 레지스터의 값이 갱신되기 전에 move 명령어가 실행되면 데이터 해저드가 발생할 수 있습니다.

```
7c: 0c000000  jal 0 <main>
80: 00000000  nop
```

[사진 9] Testcase 2-4

- jal 0 <main> 명령어는 함수 호출을 수행하고 ra 레지스터를 갱신합니다.
- move v1, v0 명령어는 함수 호출 후의 v0 레지스터 값을 v1 레지스터로 이동합니다.
- 함수 호출 후에 v0 레지스터의 값이 유효하지 않을 수 있으므로 데이터 해저드가 발생할 수 있습니다.

Testcase3

```
10: 2404000f  li  a0,15
14: 0c000000  jal 0 <main>
```

[사진 10] Testcase 3-1

- li a0, 15 명령어는 a0 레지스터에 값을 로드합니다.
- jal 0 <main> 명령어는 함수 호출을 수행하고 ra 레지스터를 갱신합니다.
- 함수 호출 후에 a0 레지스터의 값이 유효하지 않을 수 있으므로 데이터 해저드가 발생할 수 있습니다.

```
4c: 8fc20028  lw  v0,40(s8)
50: 00000000  nop
54: 28420002  slti  v0,v0,2
58: 10400004  beqz  v0,6c <fibonacci+0x38>
```

[사진 11] Testcase 3-2

- lw v0, 40(s8) 명령어는 v0 레지스터에 값을 로드합니다.
- slti v0, v0, 2 명령어는 v0 레지스터 값을 사용하여 비교합니다.
- beqz v0, 6c <fibonacci+0x38> 명령어는 v0 레지스터 값을 사용하여 조건 분기를 수행합니다.
- v0 레지스터의 값이 로드되기 전에 slti와 beqz 명령어가 실행되면 해저드가 발생할

수 있습니다.

```
78: 00402025  move  a0,v0
7c: 0c000000  jal 0 <main>
```

[사진 12] Testcase 3-3

- move a0, v0 명령어는 v0 레지스터의 값을 a0 레지스터로 이동합니다.
- jal 0 <main> 명령어는 함수 호출을 수행하고 ra 레지스터를 갱신합니다.
- v0 레지스터의 값이 a0로 이동되기 전에 jal 명령어가 실행되면 해저드가 발생할 수 있습니다.

Testcase4

```
10: 24040005  li  a0,5
14: 0c000000  jal 0 <main>
```

[사진 13] Testcase 4-1

- li a0, 5 명령어는 a0 레지스터에 값을 로드합니다.
- jal 0 <main> 명령어는 함수 호출을 수행하고 ra 레지스터를 갱신합니다.
- 함수 호출 후에 a0 레지스터의 값이 유효하지 않을 수 있으므로 해저드가 발생할 수 있습니다.

```
58: 8fc30020  lw  v1,32(s8)
5c: 24020001  li  v0,1
60: 14620004  bne v1,v0,74 <factorial+0x40>
```

[사진 14] Testcase 4-2

- lw v0, 32(s8) 명령어는 v0 레지스터에 값을 로드합니다.
- beqz v0, 68 <factorial+0x34> 명령어는 v0 레지스터 값을 사용하여 조건 분기를 수행합니다.
- v0 레지스터의 값이 로드되기 전에 beqz 명령어가 실행되면 해저드가 발생할 수 있습니다.

```
74: 8fc20020  lw  v0,32(s8)
78: 00000000  nop
7c: 2442ffff  addiu v0,v0,-1
```

[사진 15] Testcase 4-3

- lw v1, 32(s8) 명령어는 v1 레지스터에 값을 로드합니다.
- bne v1, v0, 74 <factorial+0x40> 명령어는 v1 레지스터 값을 사용하여 조건 분기를 수행합니다.
- v1 레지스터의 값이 로드되기 전에 bne 명령어가 실행되면 해저드가 발생할 수 있습니다.

Testcase5

```
50: 8fc20024  lw  v0,36(s8)
54: 00000000  nop
58: 14400004  bnez  v0,6c <power+0x34>
```

[사진 16] Testcase 5-1

- lw v0, 36(s8) 명령어는 v0 레지스터에 값을 로드합니다.
- bnez v0, 6c <power+0x34> 명령어는 v0 레지스터 값을 사용하여 조건 분기를 수행합니다.
- v0 레지스터의 값이 로드되기 전에 bnez 명령어가 실행되면 해저드가 발생할 수 있습니다.

```
6c: 8fc20024  lw  v0,36(s8)
70: 00000000  nop
74: 2442ffff  addiu v0,v0,-1
```

[사진 17] Testcase 5-2

- lw v0, 36(s8) 명령어는 v0 레지스터에 값을 로드합니다.
- addiu v0, v0, -1 명령어는 v0 레지스터의 값을 갱신합니다.
- v0 레지스터의 값이 로드되기 전에 addiu 명령어가 실행되면 해저드가 발생할 수 있습니다.

```
78: 00402825  move  a1,v0
7c: 8fc40020  lw  a0,32(s8)
```

[사진 18] Testcase 5-3

- move a1, v0 명령어는 v0 레지스터의 값을 a1 레지스터로 이동합니다.
- jal 0 <main> 명령어는 함수 호출을 수행하고 ra 레지스터를 갱신합니다.
- v0 레지스터의 값이 a1로 이동되기 전에 jal 명령어가 실행되면 데이터 해저드가 발생할 수 있습니다.

```
8c: 8fc20020  lw  v0,32(s8)
90: 00000000  nop
94: 00620018  mult  v1,v0
98: 00001012  mflo  v0
```

[사진 19] Testcase 5-4

- lw v0, 32(s8) 명령어는 v0 레지스터에 값을 로드합니다.
- mult v1, v0 명령어는 v1와 v0 레지스터의 값을 곱합니다.
- mflo v0 명령어는 곱셈의 하위 32비트 결과를 v0 레지스터에 저장합니다.
- v0 레지스터의 값이 로드되기 전에 mult 명령어가 실행되면 데이터 해저드가 발생할 수 있습니다.
- mult 명령어의 결과가 mflo 명령어에서 사용되기 전에 완료되어야 합니다.

3. Design

a) 프로그램 디자인

이 프로그램은 MIPS 프로세서의 파이프라인 시뮬레이터를 구현합니다. 기본적으로 명령어를 인출(Fetch), 디코드(Decode), 실행(Execute), 메모리 접근(Memory Access), 결과 쓰기(Write Back) 단계를 통해 처리하며, 이를 위해 각 단계마다 파이프라인 레지스터를 사용합니다. 또한 데이터 위험(Data Hazard)을 처리하기 위해 포워딩과 버블 삽입, 그리고 분기 예측을 구현했습니다.

b) 주요 기능 및 구조

포워딩 (Forwarding)

포워딩은 데이터 위험을 최소화하기 위해 필요한 기능입니다. 데이터 위험은 이전 명령어의 결과를 후속 명령어가 필요로 할 때 발생합니다. 포워딩은 필요한 데이터를 파이프라인의 이전 단계에서 바로 가져올 수 있도록 합니다. 이를 통해 명령어가 ALU에 도달하기 전에 필요한 데이터를 확보할 수 있습니다.

포워딩 로직은 forwarding 함수에서 구현됩니다. 이 함수는 EX 단계와 MEM 단계에서 포워딩을 처리합니다. EX 단계에서는 EX_MEM 레지스터가 ID_EX 레지스터와 동일한 레지스터를 참조하는 경우 ALU 결과를 포워딩합니다. MEM 단계에서는 MEM_WB 레지스터가 ID_EX 레지스터와 동일한 레지스터를 참조하는 경우 메모리에서 읽은 데이터를 포워딩합니다.

Static Prediction

스태틱 분기 예측은 항상 분기를 수행하는 것으로 가정합니다. 이는 단순한 예측 방식으로, 항상 분기 조건이 참이라고 가정하여 분기를 예측합니다. 분기 예측 로직은 branchPrediction 함수에서 구현됩니다. 이 함수는 opcode가 분기 명령어인 경우 항상 분기를 수행하도록 PCSrc를 설정합니다.

레지스터 버퍼 추가

파이프라인의 각 단계 사이에 데이터를 저장하는 파이프라인 레지스터를 사용합니다. 이는 각 단계별로 필요한 데이터를 저장하고 전달하는 역할을 합니다. 파이프라인 레지스터는 다음과 같습니다

- IF_ID_Reg: Instruction Fetch 단계와 Instruction Decode 단계 사이
- ID_EX_Reg: Instruction Decode 단계와 Execute 단계 사이
- EX_MEM_Reg: Execute 단계와 Memory Access 단계 사이
- MEM_WB_Reg: Memory Access 단계와 Write Back 단계 사이

데이터 위험 및 버블 처리

데이터 위험(Data Hazard) 처리 중 하나는 Load-Use 위험으로, 로드 명령어 이후에 사용 명령어가 바로 올 때 발생합니다. 이를 감지하고 버블을 삽입하여 해결합니다. 위험 감지 및 버블 삽입 로직은 hazardDetection 함수에서 구현됩니다. 이 함수는 ID_EX 단계의 메모리 읽기 신호가 활성화되어 있고, ID_EX 레지스터가 IF_ID 레지스터와 동일한 레지스터를 참조하는 경우 버블을 삽입합니다.

c) 실행 단계

Instruction Fetch (IF)

명령어를 메모리에서 가져와 파이프라인 레지스터 IF_ID에 저장합니다. 현재 PC의 명령어를 인출하고, PC를 다음 명령어로 이동합니다. 이 단계에서는 명령어의 주소와 명령어 자체를 IF_ID 레지스터에 저장합니다.

Instruction Decode (ID)

명령어를 디코드하고 필요한 데이터를 읽어옵니다. 파이프라인 레지스터 ID_EX에 저장합니다. 이 단계에서는 명령어의 타입을 판별하고, 명령어에 따라 제어 신호를 설정합니다. 또한, 소스 레지스터와 타겟 레지스터의 값을 읽어옵니다.

Execute (EX)

ALU 연산을 수행하고 결과를 파이프라인 레지스터 EX_MEM에 저장합니다. 이 단계에서는 ALU 연산을 수행하여 결과를 계산하고, 메모리 접근과 관련된 제어 신호를 설정합니다. 또한, 필요한 경우 포워딩을 처리합니다.

Memory Access (MEM)

메모리 접근 작업을 수행하고 결과를 파이프라인 레지스터 MEM_WB에 저장합니다. 이 단계에서는 메모리에서 데이터를 읽거나, 메모리에 데이터를 쓰는 작업을 수행합니다. 메모리 접근 결과는 MEM_WB 레지스터에 저장됩니다.

Write Back (WB)

연산 결과를 레지스터에 쓰는 작업을 수행합니다. 이 단계에서는 MEM_WB 레지스터의 데이터를 참조하여 결과를 레지스터 파일에 기록합니다. 메모리에서 읽은 데이터나 ALU 결과를 목적 레지스터에 씁니다.

Possible Jump

프로그램 카운터(PC)를 업데이트하여 가능한 점프를 처리합니다. 이 단계에서는 분기 명령어의 실행 결과에 따라 PC를 업데이트합니다. 분기 조건이 참이면 PC를 점프 주소로 설정합니다.

Branch Prediction

분기 예측을 기반으로 PC를 업데이트합니다. 분기 예측 로직은 branchPrediction 함수에서 구현되며, opcode가 분기 명령어인 경우 항상 분기를 수행하도록 설정합니다.

d) 전체적인 동작 흐름

1. 프로그램 실행 전에 레지스터와 메모리를 초기화합니다.
2. 명령어를 인출하고, 디코드하고, 실행하고, 메모리 접근을 수행하고, 결과를 쓰는 단계를 반복합니다.
3. 각 사이클마다 파이프라인 레지스터를 업데이트하여 다음 사이클에 사용할 데이터를 준비합니다.
4. 분기 명령어가 있는 경우, 분기 예측 로직을 통해 PC를 업데이트합니다.
5. 모든 명령어 실행이 완료되면 최종 결과를 출력합니다.

4. Implementation

a) 포워딩 함수

```
// 포워딩 로직을 구현하는 함수
void forwarding() {
    if (ENABLE_FORWARD) {
        // EX 단계에서 포워딩
        if (EX_MEM.RegWrite && (EX_MEM.writeReg != 0)) {
            if (EX_MEM.writeReg == ID_EX.rs) {
                ID_EX.readData1 = EX_MEM.ALUResult;
            }
            if (EX_MEM.writeReg == ID_EX.rt) {
                ID_EX.readData2 = EX_MEM.ALUResult;
            }
        }

        // MEM 단계에서 포워딩
        if (MEM_WB.RegWrite && (MEM_WB.writeReg != 0)) {
            if (MEM_WB.writeReg == ID_EX.rs) {
                ID_EX.readData1 = MEM_WB.MemtoReg ? MEM_WB.readData : MEM_WB.ALUResult;
            }
            if (MEM_WB.writeReg == ID_EX.rt) {
                ID_EX.readData2 = MEM_WB.MemtoReg ? MEM_WB.readData : MEM_WB.ALUResult;
            }
        }
    }
}
```

[사진 20] 포워딩 함수 구현

함수 선언 및 포워딩 활성화 조건

- forwarding() 함수는 데이터 포워딩을 처리합니다.
- 포워딩이 활성화된 경우에만 함수가 실행됩니다.

EX 단계에서 포워딩

- EX 단계에서는 ALU 연산 결과를 다음 단계로 전달합니다.
- EX 단계의 레지스터 쓰기 신호가 활성화되고, 쓰기 레지스터가 0이 아닌 경우 포워딩이 필요합니다.
- 쓰기 레지스터가 현재 EX 단계에서 읽어오는 소스 레지스터(rs)와 같다면, EX 단계의 첫 번째 읽기 데이터(readData1)를 ALU 결과로 업데이트합니다.
- 쓰기 레지스터가 현재 EX 단계에서 읽어오는 두 번째 소스 레지스터(rt)와 같다면, EX 단계의 두 번째 읽기 데이터(readData2)를 ALU 결과로 업데이트합니다.

MEM 단계에서 포워딩

- MEM 단계에서는 메모리 접근 결과를 다음 단계로 전달합니다.
- MEM 단계의 레지스터 쓰기 신호가 활성화되고, 쓰기 레지스터가 0이 아닌 경우 포워딩이 필요합니다.
- 쓰기 레지스터가 현재 EX 단계에서 읽어오는 소스 레지스터(rs)와 같다면, EX 단계의 첫 번째 읽기 데이터(readData1)를 MEM 단계의 결과로 업데이트합니다. 여기서 결과는 메모리에서 읽어온 데이터이거나 ALU 결과일 수 있습니다.
- 쓰기 레지스터가 현재 EX 단계에서 읽어오는 두 번째 소스 레지스터(rt)와 같다면, EX 단계의 두 번째 읽기 데이터(readData2)를 MEM 단계의 결과로 업데이트합니다. 여기서 결과는 메모리에서 읽어온 데이터이거나 ALU 결과일 수 있습니다.

b) 분기 예측 함수

```
// 분기 예측을 기반으로 PC를 업데이트하는 함수
void branchPrediction() {
    if (ENABLE_BRANCH_PREDICTION) {
        int predictedPC = pc + 4; // 기본적으로 다음 명령어로 이동

        if (opcode == 4) { // beqz
            predictedPC = pc + 4 * immediate;
        } else if (opcode == 5) { // bnez
            predictedPC = pc + 4 * immediate;
        } else if (opcode == 0 && funct == 8) { // jr
            predictedPC = Register[31];
        } else if (opcode == 3) { // jal
            predictedPC = 4 * J_address;
        }

        printf(" Predicted newPC: 0x%08x\n", predictedPC); // 예측된 PC 값 출력

        // 실제 점프 수행
        PCSrc = 1;
        int actualPC = possibleJump();

        // 예측이 맞았는지 확인하고 결과 출력
        if (predictedPC == actualPC) {
            printf("Branch prediction successful.\n");
        } else {
            printf("Branch prediction failed. Inserting bubble.\n");
            // 예측 실패 시 파이프라인 초기화 (버블 삽입)
            IF_ID.valid = 0;
            ID_EX.valid = 0;
            EX_MEM.valid = 0;
            MEM_WB.valid = 0;
            pc = actualPC; // 실제 PC로 수정
        }
    }
}
```

[사진 21] 분기 예측 함수 구현

함수 선언 및 분기 예측 활성화 조건

- ENABLE_BRANCH_PREDICTION가 참이면 분기 예측 로직이 실행됩니다.
- branchPrediction() 함수는 분기 예측을 통해 프로그램 카운터(PC)를 업데이트하는 역할을 합니다.
- 분기 예측이 활성화된 경우에만 함수가 실행됩니다.
- 분기 예측 후 버블처리를 추가하니 다른 함수가 잘 돌지 않아 코드제출할 때 제외하였습니다.

분기 명령어 조건 확인

- 분기 명령어를 확인하기 위해 명령어의 opcode를 검사합니다.
- opcode가 4 (beq, beqz) 또는 5 (bne, bnez)인 경우 분기 명령어로 간주합니다.

항상 분기하는 예측 (Static Prediction)

- ALWAYS_TAKEN 조건이 참인 경우 항상 분기한다고 가정합니다.
- 분기 예측이 항상 분기로 설정된 경우, PC 소스 신호(PCSrc)를 1로 설정합니다.
- PCSrc가 1로 설정되면, possibleJump() 함수가 호출되어 실제로 분기를 수행합니다.

c) 파이프라인 레지스터 구조체 정의

```
// 파이프라인 레지스터 구조체 정의
typedef struct {
    uint32_t pc;
    uint32_t instruction;
    int valid;
    uint32_t rs, rt, rd;
    int16_t immediate;
} IF_ID_Reg;

typedef struct {
    uint32_t pc;
    uint32_t instruction;
    int valid;
    uint32_t rs, rt, rd;
    uint32_t ALUResult, readData1, readData2;
    int16_t immediate;
    int RegDst, RegWrite, ALUSrc, PCSrc, MemRead, MemWrite, MemtoReg, ALUOp;
} ID_EX_Reg;

typedef struct {
    uint32_t pc;
    uint32_t instruction;
    int valid;
    uint32_t ALUResult, writeData;
    uint32_t writeReg;
    int MemRead, MemWrite, MemtoReg, RegWrite;
} EX_MEM_Reg;

typedef struct {
    uint32_t pc;
    uint32_t instruction;
    int valid;
    uint32_t readData;
    uint32_t ALUResult;
    uint32_t writeReg;
    int MemtoReg, RegWrite;
} MEM_WB_Reg;

// 파이프라인 레지스터 초기화
IF_ID_Reg IF_ID;
ID_EX_Reg ID_EX;
EX_MEM_Reg EX_MEM;
MEM_WB_Reg MEM_WB;
```

[사진 22] 파이브라인 레지스터 구조체 정의

IF_ID 레지스터

IF(Instruction Fetch) 단계에서 ID(Instruction Decode) 단계로 넘어가는 명령어와 관련된 정보를 저장합니다.

- pc: 프로그램 카운터 값
- instruction: 인출된 명령어
- valid: 레지스터의 유효성을 나타내는 플래그
- rs, rt, rd: 명령어의 소스 및 대상 레지스터
- immediate: 즉시 값

ID_EX 레지스터

ID(Instruction Decode) 단계에서 EX(Execute) 단계로 넘어가는 명령어와 관련된 정보를 저장합니다.

- pc: 프로그램 카운터 값
- instruction: 디코딩된 명령어
- valid: 레지스터의 유효성을 나타내는 플래그
- rs, rt, rd: 명령어의 소스 및 대상 레지스터
- ALUResult, readData1, readData2: ALU 결과와 레지스터에서 읽은 데이터
- immediate: 즉시 값
- 제어 신호: RegDst, RegWrite, ALUSrc, PCSrc, MemRead, MemWrite, MemtoReg, ALUOp

EX_MEM 레지스터

EX(Execute) 단계에서 MEM(Memory Access) 단계로 넘어가는 명령어와 관련된 정보를 저장합니다.

- pc: 프로그램 카운터 값
- instruction: 실행된 명령어
- valid: 레지스터의 유효성을 나타내는 플래그
- ALUResult, writeData: ALU 결과와 메모리에 쓰일 데이터
- writeReg: 쓰기 대상 레지스터
- 제어 신호: int MemRead, MemWrite, MemtoReg, RegWrite

MEM_WB 레지스터

MEM(Memory Access) 단계에서 WB(Write Back) 단계로 넘어가는 명령어와 관련된 정보를 저장합니다.

- pc: 프로그램 카운터 값
- instruction: 메모리 접근이 완료된 명령어
- valid: 레지스터의 유효성을 나타내는 플래그
- readData: 메모리에서 읽은 데이터
- ALUResult: ALU 결과
- writeReg: 쓰기 대상 레지스터
- 제어 신호: int MemtoReg, RegWrite

4) 버블 처리 함수

```
// 버블 감지 및 처리
if (hazardDetection()) {
    printf("\t[Hazard Detection] Bubble Inserted\n");
    EX_MEM.valid = 0;
    ID_EX.valid = 0;
    continue;
}
```

[사진 23] 버블 감지 함수

```
// 버블 처리 함수 (데이터 위험)
int hazardDetection() {
    // Load-Use 데이터 위험 감지
    if (ID_EX.MemRead && ((ID_EX.rt == IF_ID.rs) || (ID_EX.rt == IF_ID.rt))) {
        // 버블 삽입: IF/ID 파이프라인 레지스터를 정지하고, ID/EX 파이프라인 레지스터를 초기화
        ID_EX.valid = 0;
        return 1;
    }
    return 0;
}
```

[사진 24] 버블 처리 함수

버블 처리 함수는 데이터 해저드를 감지하고 처리하는 역할을 합니다.

Load-Use 데이터 위험 감지

- ID_EX 단계에서 메모리를 읽는 명령어(MemRead)가 있으며, 이 명령어의 rt 레지스터가 현재 IF_ID 단계의 rs 또는 rt 레지스터와 동일한 경우, 데이터 해저드가 발생한다고 판단합니다.

버블 삽입

- 데이터 해저드가 감지되면, ID_EX 파이프라인 레지스터를 유효하지 않음(valid = 0)으로 설정하여 버블을 삽입합니다.
- 이를 통해 ID_EX 단계의 명령어가 다음 사이클로 넘어가지 않고 대기하게 됩니다.
- 동시에 IF_ID 파이프라인 레지스터도 정지시켜 다음 명령어의 인출을 지연시킵니다.

5. Test

a) 파이프라인 레지스터 버퍼 추가

```
koicloud@ca-32190192:~/vscode/hw3$ gcc -o 32190192 32190192.c
koicloud@ca-32190192:~/vscode/hw3$ ./32190192 input3/fibonacci.bin


32190192> Cycle: 1
        [Instruction Fetch] 0x27bdffe0 (PC=0x00000000)
        [Instruction Decode] Type: I, Inst: addiu sp sp -32
                opcode: 9, rt: 29 (16777216), rs: 29 (16777216), imm: -32
                RegDst: 0, RegWrite: 1, ALUSrc: 1, PCSrc: 0, MemRead: 0, MemWrite: 0, MemtoReg: 0, ALUOp: 2
                IF/ID: PC = 0x00000000, Instruction = 0x27bdffe0, Valid = 1
        [Execute] ALU = 16777184
                ID/EX: PC = 0x00000000, Instruction = 0x27bdffe0, Valid = 1, ALUOp = 2, RegDst = 0, ALUSrc = 1, MemRead = 0, MemWrite = 0, MemtoReg = 0, RegWrite = 1
        [Memory Access] Pass
                EX/MEM: PC = 0x00000000, Instruction = 0x27bdffe0, Valid = 1, ALUResult = 16777184, MemRead = 0, MemWrite = 0, MemtoReg = 0, RegWrite = 1
        [Write Back] Target: sp, Value: 16777184 /
                MEM/WB: PC = 0x00000000, Instruction = 0x27bdffe0, Valid = 1, ALUResult = 16777184, ReadData = 0, MemtoReg = 0, RegWrite = 1
 newPC: 0x00000004
```

[사진 25] 레지스터 버퍼 출력 테스트

각 파이브라인 사이에 레지스터 버퍼를 놓아 출력한 모습입니다.

b) 헤저드 탐지 후 버블 처리

```
4c: 8fc20028  lw  v0,40(s8)
50: 00000000  nop
54: 28420002  slti  v0,v0,2
58: 10400004  beqz  v0,6c <fibonacci+0x38>
```

[사진 26] 버블처리 예시 어셈블리어

- lw v0, 40(s8) 명령어는 v0 레지스터에 값을 로드합니다.
- slti v0, v0, 2 명령어는 v0 레지스터 값을 사용하여 비교합니다.
- beqz v0, 6c <fibonacci+0x38> 명령어는 v0 레지스터 값을 사용하여 조건 분기를 수행합니다.
- v0 레지스터의 값이 로드되기 전에 slti와 beqz 명령어가 실행되면 해저드가 발생할 수 있습니다.

```
32190192> Cycle: 13
        [Instruction Fetch] 0x8fc20028 (PC=0x0000004c)
        [Instruction Decode] Type: I, Inst: lw v0 40(s8)
                opcode: 35, rt: 2 (0), rs: 30 (16777144), imm: 40
                RegDst: 0, RegWrite: 1, ALUSrc: 1, PCSrc: 0, MemRead: 1, MemWrite: 0, MemtoReg: 1, ALUOp: 2
                IF/ID: PC = 0x0000004c, Instruction = 0x8fc20028, Valid = 1
        [Execute] ALU = 16777184
                ID/EX: PC = 0x0000004c, Instruction = 0x8fc20028, Valid = 1, ALUOp = 2, RegDst = 0, ALUSrc = 1, MemRead = 1, MemWrite = 0, MemtoReg = 1, RegWrite = 1
        [Memory Access] Load, Address: 0x00ffffe0, Value: 15
                EX/MEM: PC = 0x0000004c, Instruction = 0x8fc20028, Valid = 1, ALUResult = 16777184, MemRead = 1, MemWrite = 0, MemtoReg = 1, RegWrite = 1
        [Write Back] target: v0, Value: 15 /
                MEM/WB: PC = 0x0000004c, Instruction = 0x8fc20028, Valid = 1, ALUResult = 16777184, ReadData = 15, MemtoReg = 1, RegWrite = 1
 newPC: 0x00000050



32190192> Cycle: 14
        [Instruction Fetch] 0x00000000 (PC=0x00000050)
        [Instruction Decode] NOP!!!


32190192> Cycle: 15
        [Instruction Fetch] 0x28420002 (PC=0x00000054)
        [Hazard Detection] Bubble Inserted


32190192> Cycle: 16
        [Instruction Fetch] 0x10400004 (PC=0x00000058)
        [Hazard Detection] Bubble Inserted
```

[사진 27] 버블처리 테스트

실행결과

V0 레지스터의 값이 로드되기전에 slti, beqz명령어가 실행 돼서 버블처리가 되는 것을 보실 수 있습니다.

c) 포워딩

포워딩을 해결해보려 했지만 성공하지 못하였습니다.

6. Lesson

a) 느낀 점

구현을 완료하지 못해 테스트 케이스가 돌아가지 않는 것이 매우 아쉽지만, 테스트 케이스를 돌아가게 하기 위해 데이터 해저드가 발생하는 지점을 모두 찾아보며 헤저드를 해결하려고 노력하며 과제를 진행했습니다. 하지만 이를 해결하는 것은 쉽지 않았습니다.

컴퓨터구조론 수업을 들으며 많은 성장을 할 수 있었고, 한계에 도전한다는 마음으로 과제를 진행했습니다. 과제 2를 마무리 지은 것도 지금도 신기하게 느껴지지만, 과제 3을 하며 컴퓨터의 복잡성에 대해 한 번 더 느끼게 되었습니다. 성능을 올리기 위해 포워딩, 분기 예측 등 간단한 로직인 것 같아도 그 로직을 관리하기 위해서는 매우 복잡한 로직들이 들어간다고 느꼈습니다.

특히, 데이터 해저드를 해결하기 위해 발생 지점을 모두 찾아내고 이를 처리하는 과정이 어려웠습니다. 포워딩과 분기 예측 로직을 구현하면서, 단순해 보이는 로직도 실제로는 매우 복잡하다는 것을 깨달았습니다. 이러한 복잡성을 다루기 위해 다양한 조건과 상황을 고려해야 했고, 이를 통해 더욱 정교한 로직을 구현하는 방법을 배울 수 있었습니다.

b) 데이터 해저드 감지

Load-Use 해저드가 발생했을 때 이를 적절히 감지하고 처리하는 것은 매우 까다로웠습니다. 처음에는 해저드를 감지하는 로직을 구현했지만, 특정 상황에서 해저드를 제대로 감지하지 못하거나, 불필요한 버블을 삽입하는 경우가 발생했습니다.
초기에는 단순히 레지스터 비교를 통해 해저드를 감지했으나, 이를 보다 정교하게 하기 위해 MemRead 신호와 레지스터 간의 관계를 명확히 정의했습니다.

c) 포워딩 로직 구현

포워딩을 구현하는 과정에서 여러 어려움이 있었습니다. 특히, EX 단계와 MEM 단계에서의 포워딩 로직을 정확하게 구현하는 것이 쉽지 않았습니다. 여러 논문과 참고 자료를 통해 포워딩 로직의 원리를 다시 한 번 학습하고, 이를 구현에 반영했습니다.
복잡한 로직을 구현할 때는 기존의 연구 자료나 논문을 참조하는 것이 큰 도움이 되는 것을 깨달았습니다.

d) 분기 예측 로직 구현

분기 예측을 구현하는 과정에서 다양한 분기 상황을 고려해야 했습니다. 특히, 항상 분기가 발생한다고 가정하는 단순한 예측 방식(Static Prediction)을 구현하는 데에도 여러 어려움이 있었습니다.

이번 과제를 통해 파이프라인 설계에서 발생할 수 있는 다양한 문제를 해결하는 방법을 배웠습니다. 특히, 데이터 해저드와 포워딩, 분기 예측 등 복잡한 로직을 구현하면서 많은 교훈을 얻을 수 있었습니다. 이를 통해 실제 시스템 설계에서 발생할 수 있는 문제를 보다 효율적으로 해결할 수 있는 능력을 키울 수 있었습니다.