

컴퓨터구조 프로젝트 보고서

Project 3

Pipeline Architecture

수업 명: 컴퓨터구조

담당 교수: 이성원 교수님

학과: 컴퓨터정보공학부

학번: 2023202070

이름: 최현진

제출일: 2025.05.24

1. Introduction

본 프로젝트는 MIPS 파이프라인 구조에서 주어진 shell_sort 어셈블리 코드를 기반으로, data 및 control 의존성을 고려한 명령어 흐름을 구현하는 데 목적이 있다. 명령어 간의 RAW(RAW: Read After Write) hazard과 control hazard를 파악하여, NOP 삽입 또는 ALU Forwarding을 통해 pipeline stall을 해소한다. 모든 명령어는 32비트 바이너리 형태로 변환되어 M_TEXT_SEG.txt에 저장되며, 포워딩 제어 신호는 A/B 포트 기준으로 M_TEXT_FWD.txt에 기록된다. 프로젝트는 shell_sort.asm → M_TEXT_SEG.txt 변환, M_TEXT_FWD.txt 작성, 시뮬레이션 검증 단계로 구성되며, 명령어 실행 결과는 reg_dump.txt와 mem_dump.txt를 통해 확인된다.

2. Assignment

2-1. 프로젝트 이론 설명

Structural Hazards: 명령어 수행 중 메모리 자원을 동시에 접근하여 발생하는 hazard이다. 본 구현에서는 Instruction Memory(IM)과 Data Memory(DM)가 분리되어 있기 때문에 발생하지 않는다.

Data Hazards:

RAW (Read After Write)

이전 명령어가 특정 레지스터를 write한 후, 그 다음 명령어가 같은 레지스터를 read하려는 경우 발생한다. 본 프로젝트의 파이프라인 구조에서 유일하게 발생 가능한 데이터 의존성이다.

add \$2, \$1, \$3
sub \$4, \$2, \$5
EX 단계에서 \$2가 필요하므로, 이전 명령어 결과가 WB 단계에 도달하기 전까지는 Forwarding 또는 NOP 삽입을 통해 해결해야 한다.

WAW (Write After Write) 및 WAR (Write After Read)

본 구현에서는 명령어의 write-back이 고정된 WB 단계에서만 이루어지므로, 동일한 레지스터를 두 명령어가 동시에 쓰거나(WAW), 후속 명령어가 먼저 읽는 경우(WAR)는 존재하지 않는다. 본 구현에 사용된 파이프라인에서는 RAW data hazards만 발생한다.

Control Hazards

Control Hazard는 분기(branch)나 점프(jump) 명령어의 실행 결과에 따라 프로그램 흐름이 바뀌기 때문에, 분기 여부가 확정되기 전까지 다음 명령어를 fetch하면 발생한다.

beq \$1, \$2, LABEL
add \$3, \$4, \$5 # 이미 fetch
본 프로젝트의 MIPS 파이프라인에서는 branch 조건 판단은 ID stage에서 수행된다. 하지만 다음 명령어는 이미 IF 단계에서 fetch된 상태이기 때문에, 이 명령어가 잘못 실행될 가능성이 있다.

H/W적으로 hazard를 피하는 방법으로 Forwarding Unit의 추가가 있다. MM 단계(ALU 결과)나 WB 단계의 데이터를 EX 단계의 ALU 입력으로 전달하는 구조이다.

본 프로젝트에서는 M_TEXT_FWD.txt파일을 통해 다음과 같은 제어 신호 조합을 사용하여 forwarding을 구현하였다:

00: From Register file of ID stage to EX stage

01: From ALU output of MM stage

10: From Writeback data of WB stage

branch 조건 판단은 ID 단계에서 이루어지므로, 피연산자로 사용되는 레지스터(\$rs, \$rt)의 값이 이전 명령어에서 write되는 경우, ID 단계에서 필요한 값을 사용할 수 없어 data hazard가 발생한다. 본 구현에서는 ID 단계로의 포워딩 경로가 존재하지 않기 때문에, nop을 삽입하여 branch 명령어가 올바른 값을 참조하도록 해야 한다.

load 명령어는 데이터가 MEM 완료 후에서야 도출되므로, 바로 다음 명령어가 EX 단계에서 그 값을 사용하는 경우에는 Forwarding만으로 해결이 불가능하다. 이 경우에는 Forwarding + 1 cycle NOP 조합으로 해결했다.

S/W적으로 hazard를 피하는 방법은 stall이 있다. 예를 들어,

<pre>sub \$2, \$1, \$3 and \$12, \$2, \$5 or \$13, \$6, \$2 add \$14, \$2, \$2</pre> <p>sub 명령어의 \$2와 파란색으로 표시한 부분 간에 data 의존성이 존재한다.</p>	<pre>sub \$2, \$1, \$3 nop nop nop and \$12, \$2, \$5 or \$13, \$6, \$2 add \$14, \$2, \$2</pre> <p>nop 3개를 추가하여 sub 명령어의 \$2와 나머지 세 명령어 간의 data hazards를 해결할 수 있다.</p>
---	---

하지만, stall은 프로그램 실행 시간을 늦추는 치명적인 단점이 존재한다. S/W적으로 stall 및 hazard를 피하는 또다른 방법은, code scheduling이다. 예를 들어,

<pre>lw \$t1, 0(\$t0) lw \$t2, 4(\$t0) nop add \$t3, \$t1, \$t2 sw \$t3, 12(\$t0) lw \$t4, 8(\$t0) nop add \$t5, \$t1, \$t4 sw \$t5, 16(\$t0)</pre>	<pre>lw \$t1, 0(\$t0) lw \$t2, 4(\$t0) lw \$t4, 8(\$t0) add \$t3, \$t1, \$t2 sw \$t3, 12(\$t0) add \$t5, \$t1, \$t4 sw \$t5, 16(\$t0)</pre>
---	---

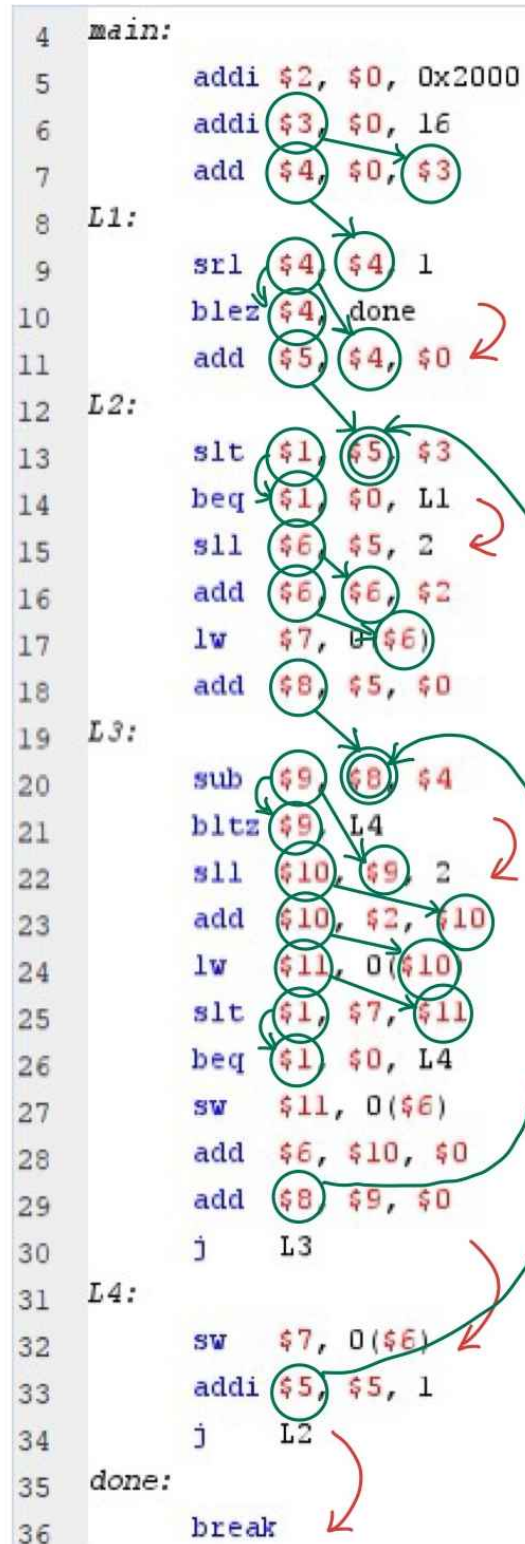
해당 경우 발생한 data hazard를 해결하기 위해서는 2개의 nop가 추가되고, 전체 사이클 수는 13이다. (Load-Use Hazard + forwarding 상정)	code scheduling을 통해 lw 명령어 1 또는 2 cycle 이후에 add 명령어를 배치하여, stall 없이도 hazard를 해결하게 된다. 전체 사이클 수는 11로 감소한다.
---	---

control hazards를 해결하는 방법으로 prediction이 있다. 기본적으로 branch가 실제로는 not taken 이라고 가정하여 fetch를 계속 진행하고, 실제 taken일 경우 rollback 처리하는 방식이다. branch target buffer를 사용하는 2-bit prediction 기법이 존재한다.

또다른 방법으로는 delayed branch가 있다. branch 또는 jump 명령어 바로 이후에 무조건 fetch되는 명령어를 배치한다. 이 명령어를 "branch slot"이라 하며 해당 프로젝트 구현에는 nop를 사용했다.

2-2. Find All Dependencies and Simulate the provided sort code

먼저, 제공된 Radix Sort 어셈블리 코드에서 발생한 data hazards를 초록색으로, control hazards를 빨간색으로 표시했다.

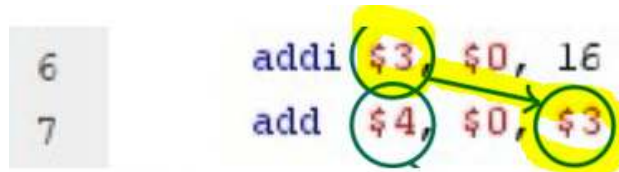


이렇게 발생한 hazards를 각각 설명한 이후,

- 기존 어셈블리코드에서 Forwarding 제어 없이 필요 없는 NOP만 제거한 시뮬레이션
- 기존 어셈블리코드에서 Forward 제어신호를 추가하여 더 많은 NOP를 제거하여, 재구성된 어셈블리 코드 시뮬레이션

으로 시나리오를 나누어 어셈블리 코드를 재구성하고 시뮬레이션을 수행하겠다.

1. `addi $3, $0, 16` <-> `add $4, $0, $3`



addi	IF	ID	EX	MEM	WB				
add		IF	ID	EX	MEM	WB			

addi가 \$3 레지스터에 값을 쓰는 시점은 WB 단계이고, add는 해당 값을 ID 단계에서 읽으려 하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

6      addi $3, $0, 16
7      nop
8      nop
9      nop
10     add  $4, $0, $3

```

addi	IF	ID	EX	MEM	WB				
add		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 add의 실행을 stall시킴으로써, addi가 WB에서 값을 쓴 이후에 add가 EX에서 \$3을 사용할 수 있게 한다.

b.

```

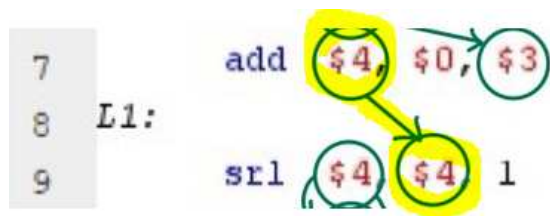
addi $3, $0, 16 00_00 // 0x004 2 addi $3, $0, 16
add  $4, $0, $3 00_01 // 0x008 3 add  $4, $0, $3, b port forwarding

```

addi	IF	ID	EX	MEM	WB				
add		IF	ID	EX	MEM	WB			

forwarding 유닛을 사용하여 addi의 EX 결과를 add의 EX 단계로 직접 전달해 더 많은 nop를 제거하고 hazard를 제거한다. M_TEXT_FWD.txt에는 b 포트에 01을 설정하여 포워딩 경로를 지정한다.

2. add \$4, \$0, \$3 <-> srl \$4, \$4, 1



add	IF	ID	EX	MEM	WB				
srl		IF	ID	EX	MEM	WB			

add가 \$4 레지스터에 값을 쓰는 시점은 WB 단계이고, srl은 해당 값을 ID 단계에서 읽으려 하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

10      add $4, $0, $3
11      nop
12      nop
13      nop
14  L1:
15      srl $4, $4, 1

```

add	IF	ID	EX	MEM	WB				
srl		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 srl의 실행을 지연시킴으로써, add가 WB에서 값을 쓴 이후에 srl이 EX 단계에서 \$4를 사용할 수 있게 한다.

b.

```

7      add $4, $0, $3
8      nop
9      nop
10     nop
11  L1:
12     srl $4, $4, 1
23     slt $1, $5, $3
24     nop
25     nop
26     nop
27     beq $1, $0, L1

```

slt <-> beq 간에는 \$1에 대한 data hazard가 존재하며, 본 구현에서는 ID stage로의 포워딩 신호가 없기 때문에 nop을 제거할 수 없다. 만약 add <-> srl 간 nop를 제거하고 포워딩한다고 하면, beq에서 L1로 분기 시에 srl 명령어는 \$5 값을 beq 명령어 이후에 포워딩하려 시도하게 된다. add가 아닌 잘못된 명령어로부터 값을 전달받게 되므로 add <-> srl 간의 hazard는 nop을 통해서만 해결 가능하며 제거할 수 없다.

3. srl \$4, \$4, 1 <-> blez \$4, done



srl	IF	ID	EX	MEM	WB				
blez		IF	ID	EX	MEM	WB			

srl이 \$4 레지스터에 값을 쓰는 시점은 WB 단계이고, blez는 해당 값을 ID 단계에서 조건 판단에 사용하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

14 L1:
15     srl $4, $4, 1
16     nop
17     nop
18     nop
19     blez $4, done

```

srl	IF	ID	EX	MEM	WB				
blez		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 blez의 실행을 지연시킴으로써, srl이 WB에서 값을 쓴 이후에 blez가 ID에서 \$4를 사용할 수 있게 한다.



해당 nop 삽입으로 srl와 blez 바로 아래 명령어인 add \$5, \$4, \$0와의 \$4 hazard도 해결됐다.

b.

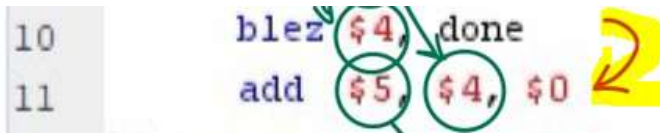
```

23     slt $1, $5, $3
24     nop
25     nop
26     nop
27     beq $1, $0, L1

```

srl <-> beq 간에는 \$4에 대한 data hazard가 존재하며, 본 구현에서는 ID stage로의 포워딩 신호가 없기 때문에 nop을 더 이상 제거할 수 없다.

4. blez \$4, done <-> add \$5, \$4, 0



blez	IF	ID	EX	MEM	WB				
add		IF	ID	EX	MEM	WB			

blez는 ID 단계에서 분기 여부를 판단하지만, 다음 명령어인 add는 이미 IF 단계에서 fetch된 상태이다. 분기 결과에 따라 add 명령어가 실행되지 않아야 할 수도 있기 때문에, 잘못된 명령어가 실행되는 control hazard가 발생한다.

a.

```

19      blez $4, done
20      nop
21      add $5, $4, $0
  
```

해결을 위해 blez 다음에 무조건 실행되는 delay slot으로 nop을 하나만 삽입하여 분기 결과가 확정되기 전 잘못된 명령어가 실행되지 않도록 한다.

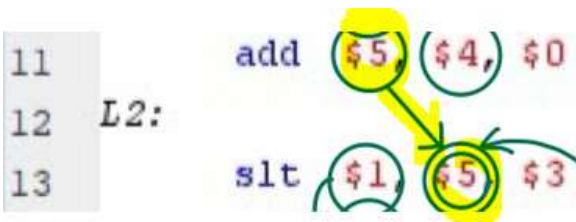
b.

```

16      blez $4, done
17      nop
18      add $5, $4, $0
  
```

control hazard는 forwarding으로 해결할 수 없으므로, 해결 방식은 a와 동일하며 더 이상 nop를 제거할 수 없다.

5. add \$5, \$4, 0 <-> slt \$1, \$5, \$3



add	IF	ID	EX	MEM	WB				
slt		IF	ID	EX	MEM	WB			

add가 \$5 레지스터에 값을 쓰는 시점은 WB 단계이고, slt는 해당 값을 ID 단계에서 읽으려 하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

21      add $5, $4, $0
22      nop
23      nop
24      nop
25  L2:
26      slt $1, $5, $3

```

add	IF	ID	EX	MEM	WB				
slt		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 slt 명령어의 실행을 지연시킴으로써, add가 WB에서 \$5 값을 쓴 이후에 slt가 ID에서 이를 읽을 수 있도록 한다.

b.

```

18      add $5, $4, $0
19      nop
20      nop
21      nop
22  L2:
23      slt $1, $5, $3
61      addi $5, $5, 1
62      nop
63      j    L2

```

만약 add ↔ slt 간의 nop을 더 제거하고 포워딩으로 해결하려 한다면, j 명령어가 L2로 분기할 경우 slt 명령어는 add의 결과가 WB에 도달하기 전에 실행되며, \$5 값을 j 명령어 이후에 포워딩하려 시도하게 된다. 이로 인해 slt는 add가 아닌 잘못된 명령어로부터 값을 전달받게 되므로, add ↔ slt 간의 hazard는 반드시 nop을 통해 해결되어야 하며, 제거할 수 없다.

6. slt \$1, \$5, \$3 ↔ beq \$1, \$0, L1

```

13      slt $1, $5, $3
14      beq $1, $0, L1

```

slt	IF	ID	EX	MEM	WB				
beq		IF	ID	EX	MEM	WB			

slt가 \$1 레지스터에 값을 쓰는 시점은 WB 단계이고, beq는 해당 \$1 값을 ID 단계에서 분기 조건 판단에 사용한다. 따라서 WB에서 쓰기 완료되기 전에 ID에서 read가 발생하여 raw data hazard가 발생한다.

a.

```

25  L2:
26      slt $1, $5, $3
27      nop
28      nop
29      nop
30      beq $1, $0, L1

```

slt	IF	ID	EX	MEM	WB				
beq		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 beq의 실행을 지연시킴으로써, slt이 WB에서 값을 쓴 이후에 beq가 ID에서 \$1를 사용할 수 있게 한다.

b.

```

22 L2:
23     slt $1, $5, $3
24     nop
25     nop
26     nop
27     beq $1, $0, L1

```

slt ↔ beq 간에는 \$1에 대한 data hazard가 존재하며, beq는 ID 단계에서 \$1 값을 읽는다. 본 구현에서는 ID stage로의 포워딩 경로가 없기 때문에 nop을 제거할 수 없다. 해당 hazard는 반드시 nop을 통해 해결되어야 하며 nop를 더 이상 제거할 수 없다.

7. beq \$1, \$0, L1 ↔ sll \$6, \$5, 2

```

14     beq $1, $0, L1
15     sll $6, $5, 2

```

beq	IF	ID	EX	MEM	WB				
sll		IF	ID	EX	MEM	WB			

beq는 ID 단계에서 분기 조건을 판단하지만, 다음 명령어인 sll은 이미 IF 단계에서 fetch된 상태이다. 분기 결과에 따라 sll 명령어가 실행되지 않아야 할 수도 있음에도 불구하고, 해당 명령어가 실행되어 잘못된 동작이 발생할 수 있는 control hazard가 발생한다.

a.

```

30     beq $1, $0, L1
31     nop
32     sll $6, $5, 2

```

해결을 위해 beq 다음에 무조건 실행되는 delay slot으로 nop을 하나만 삽입하여 분기 결과가 확정되기 전 잘못된 명령어가 실행되지 않도록 한다.

b.

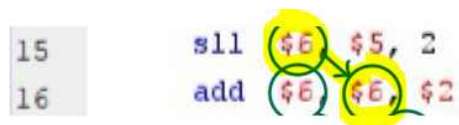
```

27     beq $1, $0, L1
28     nop
29     sll $6, $5, 2

```

control hazard는 forwarding으로 해결할 수 없으므로, 해결 방식은 a와 동일하며 더 이상 nop를 제거할 수 없다.

8. sll \$6, \$5, 2<-> add \$6, \$6, \$2



sll	IF	ID	EX	MEM	WB				
add		IF	ID	EX	MEM	WB			

sll가 \$6 레지스터에 값을 쓰는 시점은 WB 단계이고, add는 해당 값을 ID 단계에서 읽으려 하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

32      sll $6, $5, 2
33      nop
34      nop
35      nop
36      add $6, $6, $2

```

sll	IF	ID	EX	MEM	WB				
add		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 add의 실행을 stall시킴으로써, sll가 WB에서 값을 쓴 이후에 add가 EX에서 \$6을 사용할 수 있게 한다.

b.

```

29      sll $6, $5, 2      00_00 // 0x058 23 sll $6, $5, 2
30      add $6, $6, $2      01_00 // 0x05C 24 add $6, $6, $2, a port forwarding

```

sll	IF	ID	EX	MEM	WB				
add		IF	ID	EX	MEM	WB			

forwarding 유닛을 사용하여 sll의 EX 결과를 add의 EX 단계로 직접 전달해 더 많은 nop를 제거하고 hazard를 제거한다. M_TEXT_FWD.txt에는 a 포트에 01을 설정하여 포워딩 경로를 지정한다.

9. add \$6, \$6, \$2<-> lw \$7, 0(\$6)



add	IF	ID	EX	MEM	WB				
lw		IF	ID	EX	MEM	WB			

add는 WB 단계에서 \$6에 값을 쓰고, lw는 EX 단계에서 \$6을 주소 계산에 사용한다. 따라서 WB 전에 read가 발생하여 RAW data hazard가 발생한다.

a.

```

36      add $6, $6, $2
37      nop
38      nop
39      nop
40      lw  $7, 0($6)

```

add	IF	ID	EX	MEM	WB				
lw		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 lw의 실행을 stall시킴으로써, add가 WB에서 값을 쓴 이후에 lw가 EX에서 \$6을 사용할 수 있게 한다.

b.

```

30      add $6, $6, $2 01_00 // 0x05C 24 add $6, $6, $2, a port forwarding
31      lw  $7, 0($6) 01_00 // 0x060 25 lw $7, 0($6), a port forwarding

```

add	IF	ID	EX	MEM	WB				
lw		IF	ID	EX	MEM	WB			

forwarding 유닛을 사용하여 add의 EX 결과를 lw의 EX 단계로 직접 전달해 더 많은 nop를 제거하고 hazard를 제거한다. M_TEXT_FWD.txt에는 a 포트에 01을 설정하여 포워딩 경로를 지정한다.

10. add \$8, \$5, \$0<-> sub \$9, \$8, \$4



add	IF	ID	EX	MEM	WB				
sub		IF	ID	EX	MEM	WB			

add가 \$8 레지스터에 값을 쓰는 시점은 WB 단계이고, sub는 해당 값을 ID 단계에서 읽으려 하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

41      add $8, $5, $0
42      nop
43      nop
44      nop
45      L3:
46      sub $9, $8, $4

```

add	IF	ID	EX	MEM	WB				
sub		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 sub 명령어의 실행을 지연시킴으로써, add가 WB에서 \$8 값을 쓴 이후에 sub가 ID에서 이를 읽을 수 있도록 한다.

b.

```

32      add $8, $5, $0
33      nop
34      nop
35      nop
36 L3:
37      sub $9, $8, $4
55      add $8, $9, $0
56      nop
57      j    L3
  
```

만약 add ↔ sub 간의 nop을 제거하고 포워딩으로 해결하려 한다면, j 명령어가 L3로 분기할 경우 sub 명령어는 add의 결과가 WB에 도달하기 전에 실행되며, \$8 값을 j 명령어 이후에 포워딩하려 시도하게 된다. 이로 인해 sub는 add가 아닌 잘못된 명령어로부터 값을 전달받게 되므로, add ↔ sub 간의 hazard는 반드시 nop을 통해 해결되어야 하며, 더 이상 제거할 수 없다.

11. sub \$9, \$8, \$4 ↔ bltz \$9, L4

```

20      sub $9, $8, $4
21      bltz $9, L4
  
```

sub	IF	ID	EX	MEM	WB				
bltz		IF	ID	EX	MEM	WB			

sub이 \$9 레지스터에 값을 쓰는 시점은 WB 단계이고, bltz는 해당 값을 ID 단계에서 조건 판단에 사용하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

46      sub $9, $8, $4
47      nop
48      nop
49      nop
50      bltz $9, L4
  
```

sub	IF	ID	EX	MEM	WB				
bltz		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 bltz의 실행을 지연시킴으로써, sub이 WB에서 값을 쓴 이후에 bltz가 ID에서 \$9를 사용할 수 있게 한다.

```

20      sub $9, $8, $4
21      bltz $9, L4
22      sll $10, $9, 2
  
```

해당 nop 삽입으로 sub와 bltz 바로 아래 명령어인 sll \$10, \$9, 2와의 \$9 hazard도 해결됐다.

b.

```

37      sub  $9, $8, $4
38      nop
39      nop
40      nop
41      bltz $9, L4

```

sub <-> bltz 간에는 \$9에 대한 data hazard가 존재하며, 본 구현에서는 ID stage로의 포워딩 신호가 없기 때문에 nop을 더 이상 제거할 수 없다.

12. bltz \$9, L4 <-> sll \$10, \$9, 2

```

21      bltz $9, L4
22      sll  $10, $9, 2

```

bltz	IF	ID	EX	MEM	WB				
sll		IF	ID	EX	MEM	WB			

bltz는 ID 단계에서 분기 조건을 판단하지만, 다음 명령어인 sll은 이미 IF 단계에서 fetch된 상태이다. 분기 결과에 따라 sll 명령어가 실행되지 않아야 할 수도 있음에도 불구하고, 해당 명령어가 실행되어 잘못된 동작이 발생할 수 있는 control hazard가 발생한다.

a.

```

50      bltz $9, L4
51      nop
52      sll  $10, $9, 2

```

해결을 위해 bltz 다음에 무조건 실행되는 delay slot으로 nop을 한 개만 삽입하여 분기 결과가 확정되기 전 잘못된 명령어가 실행되지 않도록 한다.

b.

```

41      bltz $9, L4
42      nop
43      sll  $10, $9, 2

```

control hazard는 forwarding으로 해결할 수 없으므로, 해결 방식은 a와 동일하며 더 이상 nop를 제거할 수 없다.

13. sll \$10, \$9, 2 <-> add \$10, \$2, \$10

```

22      sll  $10, $9, 2
23      add  $10, $2, $10

```


sll	IF	ID	EX	MEM	WB				
add		IF	ID	EX	MEM	WB			

sll가 \$10 레지스터에 값을 쓰는 시점은 WB 단계이고, add는 해당 값을 ID 단계에서 읽으려 하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

52      sll $10, $9, 2
53      nop
54      nop
55      nop
56      add $10, $2, $10

```

sll	IF	ID	EX	MEM	WB				
add		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 add의 실행을 stall시킴으로써, sll가 WB에서 값을 쓴 이후에 add가 EX에서 \$10을 사용할 수 있게 한다.

b.

```

43      sll $10, $9, 2      00_00 // 0x08C 36 sll $10, $9, 2
44      add $10, $2, $10    00_01 // 0x090 37 add $10, $2, $10, b port forwarding

```

sll	IF	ID	EX	MEM	WB				
add		IF	ID	EX	MEM	WB			

forwarding 유닛을 사용하여 sll의 EX 결과를 add의 EX 단계로 직접 전달해 더 많은 nop를 제거하고 hazard를 제거한다. M_TEXT_FWD.txt에는 b 포트에 01을 설정하여 포워딩 경로를 지정한다.

14. add \$10, \$2, \$10 <-> lw \$11, 0(\$10)

```

23      add $10, $2, $10
24      lw  $11, 0($10)

```

add	IF	ID	EX	MEM	WB				
lw		IF	ID	EX	MEM	WB			

add는 WB 단계에서 \$10에 값을 쓰고, lw는 EX 단계에서 \$10을 주소 계산에 사용한다. 따라서 WB 전에 read가 발생하여 RAW data hazard가 발생한다.

a.

```

56      add $10, $2, $10
57      nop
58      nop
59      nop
60      lw  $11, 0($10)

```

add	IF	ID	EX	MEM	WB				
lw		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 lw의 실행을 stall시킴으로써, add가 WB에서 값을 쓴 이후에 lw가 EX에서 \$10을 사용할 수 있게 한다.

b.

```

44      add $10, $2, $10 00_01 // 0x090 37 add $10, $2, $10, b port forwarding
45      lw  $11, 0($10) 01_00 // 0x094 38 lw $11, 0($10), a port forwarding

```

add	IF	ID	EX	MEM	WB				
lw		IF	ID	EX	MEM	WB			

forwarding 유닛을 사용하여 add의 EX 결과를 lw의 EX 단계로 직접 전달해 더 많은 nop를 제거하고 hazard를 제거한다. M_TEXT_FWD.txt에는 a 포트에 01을 설정하여 포워딩 경로를 지정한다.

15. lw \$11, 0(\$10) <-> slt \$1, \$7, \$11

```

24      lw  $11, 0($10)
25      slt $1, $7, $11

```

lw	IF	ID	EX	MEM	WB				
slt		IF	ID	EX	MEM	WB			

lw는 WB 단계에서 \$11 값을 write하지만, 바로 다음 명령어인 slt는 ID 단계에서 \$11 값을 읽으려고 한다. 하지만 메모리로부터 값을 가져오는 lw의 특성상, 데이터는 MEM 단계 이후에야 유효하다. 이로 인해 forwarding만으로는 hazard를 완전히 해결할 수 없고, load-use hazard가 발생한다.

a.

```

60      lw  $11, 0($10)
61      nop
62      nop
63      nop
64      slt $1, $7, $11

```

lw	IF	ID	EX	MEM	WB				
slt		stall	stall	stall	IF	ID	EX	MEM	WB

lw와 slt 사이에 nop을 3개만 삽입하여 slt의 실행을 WB 이후로 지연시킨다. lw가 WB에서 값을 쓴 이후에 slt가 EX에서 \$11을 사용할 수 있게 한다.

b.

```

45      lw    $11, 0($10)
46      nop
47      slt   $1, $7, $11

```

01_00 // 0x094 38 lw \$11, 0(\$10), a port forwarding

00_00 // 0x098 39 nop

00_10 // 0x09C 40 slt \$1, \$7, \$11, b port forwarding(from lw WB)

lw	IF	ID	EX	MEM	WB				
slt		IF	ID	stall	EX	MEM	WB		

forwarding을 적용하고 slt의 실행을 1 사이클만 지연(nop 1개 삽입)함으로써, lw의 WB 값을 slt의 EX 단계로 포워딩하여 hazard를 해소한다. 이때 M_TEXT_FWD.txt에서는 slt의 b 포트에 10을 설정하여 WB에서 값을 전달받도록 구성한다.

16. slt \$1, \$7, \$11 <-> beq \$1, \$0, L4

```

25      slt   $1, $7, $11
26      beq   $1, $0, L4

```

slt	IF	ID	EX	MEM	WB				
beq		IF	ID	EX	MEM	WB			

slt이 \$1 레지스터에 값을 쓰는 시점은 WB 단계이고, beq는 해당 값을 ID 단계에서 조건 판단에 사용하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

64      slt   $1, $7, $11
65      nop
66      nop
67      nop
68      beq   $1, $0, L4

```

slt	IF	ID	EX	MEM	WB				
beq		stall	stall	stall	IF	ID	EX	MEM	WB

nop을 3개만 삽입하여 beq의 실행을 지연시킴으로써, slt이 WB에서 값을 쓴 이후에 beq가 ID에서 \$1를 사용할 수 있게 한다.

b.

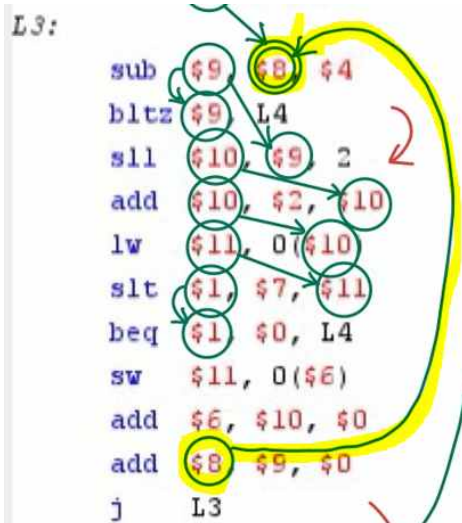
```

47      slt  $1, $7, $11
48      nop
49      nop
50      nop
51      beq  $1, $0, L4

```

slt <-> beq 간에는 \$1에 대한 data hazard가 존재하며, beq는 해당 값을 ID 단계에서 조건 판단에 사용하지만 본 구현에서는 ID stage로의 포워딩 신호가 없기 때문에 nop을 더 이상 제거할 수 없다.

17. add \$8, \$9, \$0 <-> sub \$9, \$8, \$4



add	IF	ID	EX	MEM	WB				
j		IF	ID	EX	MEM	WB			
sub			IF	ID	EX	MEM	WB		

add이 \$8 레지스터에 값을 쓰는 시점은 WB 단계이고, sub는 해당 값을 EX 단계에서 사용하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

72      add  $8, $9, $0
73      nop
74      j    L3
75      nop

```

add	IF	ID	EX	MEM	WB				
j		stall	stall	IF	ID	EX	MEM	WB	
sub					IF	ID	EX	MEM	WB

해결을 위해 먼저 add 다음에 nop을 하나 추가하고, j 다음에 무조건 실행되는 delay slot으로 nop을 한 개만 삽입하여 sub 실행 전 총 3 cycles이 소요되도록 구성했다. add가 WB에서 쓰기 완료 후 sub가 \$8를 read할 수 있다.

b.

```

32      add $8, $5, $0
33      nop
34      nop
35      nop
36 L3:
37      sub $9, $8, $4
55      add $8, $9, $0
56      nop
57      j   L3
58      nop

```

만약 add ↔ sub 간의 nop을 제거하고 sub에 포워딩 신호를 입력하면, j 명령어가 L3로 분기할 경우 sub 명령어는 add의 결과가 WB에 도달하기 전에 실행되며, \$8 값을 j 명령어 이후에 포워딩하려 시도하게 된다. 이로 인해 sub는 add가 아닌 잘못된 명령어로부터 값을 전달받게 되므로, add ↔ sub 간의 hazard는 반드시 nop을 통해 해결되어야 하며, 더 이상 제거할 수 없다.

18. j L3 ↔ sw \$7, 0(\$6)

```

30      j   L3
31 L4:
32      sw  $7, 0($6)

```



j	IF	ID	EX	MEM	WB				
sw		IF	ID	EX	MEM	WB			

j 명령어는 ID 단계에서 분기 대상 주소를 결정하지만, 다음 명령어인 sw는 이미 IF 단계에서 fetch되어 실행 대기 중인 상태이다. 만약 점프가 실제로 발생한다면, sw는 실행되어서는 안 되는 명령어가 되며 이로 인해 잘못된 동작이 발생할 수 있는 control hazard가 발생한다.

a.

```

74      j   L3
75      nop
76 L4:
77      sw  $7, 0($6)

```

j 명령어 다음에 무조건 실행되는 delay slot에 nop을 하나만 삽입함으로써, 분기 결과가 확정되기 전 잘못된 명령어가 실행되지 않도록 한다.

b.

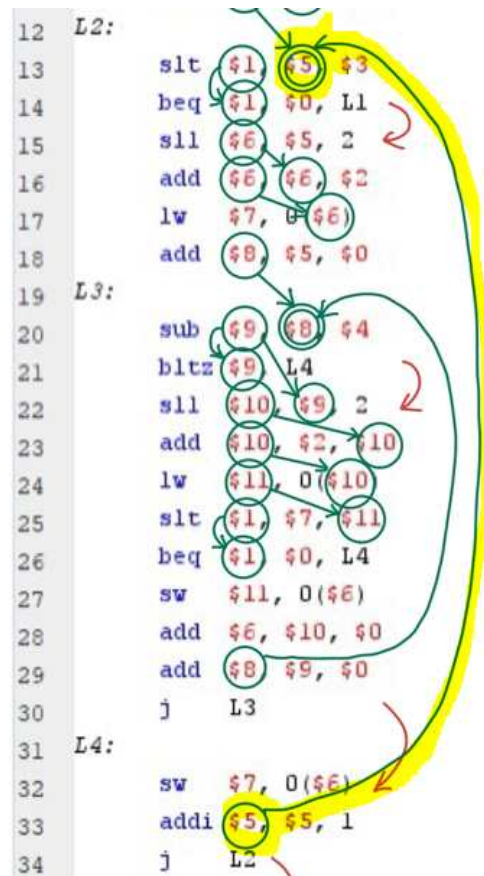
```

57      j   L3
58      nop
59 L4:
60      sw  $7, 0($6)

```

control hazard는 forwarding으로 해결할 수 없으므로, 해결 방식은 a와 동일하며 더 이상 nop를 제거할 수 없다.

19. addi \$5, \$5, 1 <-> slt \$1, \$5, \$3



addi	IF	ID	EX	MEM	WB				
j		IF	ID	EX	MEM	WB			
slt			IF	ID	EX	MEM	WB		

addi이 \$5 레지스터에 값을 쓰는 시점은 WB 단계이고, slt는 해당 값을 EX 단계에서 사용하기 때문에 WB에서 쓰기 완료되기 전에 ID에서 읽기가 발생해 raw data hazard가 발생한다.

a.

```

78   addi $5, $5, 1
79   nop
80   j    L2
81   nop
  
```

addi	IF	ID	EX	MEM	WB				
j		stall	stall	IF	ID	EX	MEM	WB	
slt					IF	ID	EX	MEM	WB

해결을 위해 먼저 addi 다음에 nop을 하나 추가하고, j 다음에 무조건 실행되는 delay slot으로 nop을 한 개만 삽입하여 slt 실행 전 총 3 cycles이 소요되도록 구성했다. addi에서 WB에서 쓰기 완료 후 slt가 \$5를 read할 수 있다.

b.

```

18      add $5, $4, $0
19      nop
20      nop
21      nop
22 L2:
23      slt $1, $5, $3
61      addi $5, $5, 1
62      nop
63      j    L2
64      nop

```


만약 add ↔ slt 간의 nop을 제거하고 slt에 포워딩 신호를 입력하면, j 명령어가 L2로 분기할 경우 slt 명령어는 add의 결과가 WB에 도달하기 전에 실행되며, \$5 값을 j 명령어 이후에 포워딩하려 시도하게 된다. 이로 인해 slt는 add가 아닌 잘못된 명령어로부터 값을 전달받게 되므로, add ↔ slt 간의 hazard는 반드시 nop을 통해 해결되어야 하며, 더 이상 제거할 수 없다.

20. j L2 ↔ break

```

34      j    L2
35 done:
36      break

```



j	IF	ID	EX	MEM	WB				
break		IF	ID	EX	MEM	WB			

j 명령어는 ID 단계에서 분기 대상 주소를 결정하지만, 다음 명령어인 break는 이미 IF 단계에서 fetch되어 실행 대기 중인 상태이다. 만약 점프가 실제로 발생한다면, break는 실행되어서는 안 되는 명령어가 되며 이로 인해 잘못된 동작이 발생할 수 있는 control hazard가 발생한다.

a.

```

80      j    L2
81      nop
82 done:
83      break

```

j 명령어 다음에 무조건 실행되는 delay slot에 nop을 하나만 삽입함으로써, 분기 결과가 확정되기 전 잘못된 명령어가 실행되지 않도록 한다.

b.

```

63      j    L2
64      nop
65 done:
66      break

```

control hazard는 forwarding으로 해결할 수 없으므로, 해결 방식은 a와 동일하며 더 이상 nop를 제거할 수 없다.

```

G:\Downloads\CA-project\project3\prj3_PCPU_2025>run.bat

G:\Downloads\CA-project\project3\prj3_PCPU_2025>vvp tb_PC.o -fst
WARNING: Memory_F.v:42: $readmemb(M_TEXT_SEG.txt): Not enough words in the file for the requested range [0:1023].
WARNING: Memory_F.v:43: $readmemb(M_TEXT_FWD.txt): Not enough words in the file for the requested range [0:1023].
WARNING: Memory_F.v:121: $readmemb(M_DATA_SEG.txt): Not enough words in the file for the requested range [0:1023].

-----
| H020-3-1647-01: Computer Architecture |
| CE.KW.AC.KR |
-----

FST info: dumpfile tb_PC.vcd opened for output.

-----
Break signal: 1, # of Cycles: 5227
-----

tb_PipelinedCPU_P.v:85: $finish called at 52385000 (1ps)

G:\Downloads\CA-project\project3\prj3_PCPU_2025>FC /L mem_dump_SS.txt mem_dump.txt
파일을 비교합니다: mem_dump_SS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.

G:\Downloads\CA-project\project3\prj3_PCPU_2025>FC /L reg_dump_SS.txt reg_dump.txt
파일을 비교합니다: reg_dump_SS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.

```

각 명령어 사이에 4 nops가 추가된 주어진 radix sort 시뮬레이션 결과, 사이클 수는 5227 이다.

a. 기존 어셈블리코드에서 Forwarding 제어 없이 필요 없는 NOP만 제거한 시뮬레이션을 위한 radix sort 어셈블리 코드를 변환한 M_TEXT_SEG.txt에 주석을 작성한 결과이다.

```

00100000_00000010_00100000_00000000 // 00_00    main: addi $2, $0, 0x2000
00100000_00000011_00000000_00010000 // 00_00          addi $3, $0, 16
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_00000011_00100000_00100000 // 00_00          add  $4, $0, $3
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_00000100_00100000_01000010 // 00_00    L1:   srl  $4, $4, 1
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_00000000_00000000_00000000 // 00_00          nop
00011000_10000000_00000000_00111101 // 00_00          blez $4, done
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_10000000_00101000_00100000 // 00_00          add  $5, $4, $0
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_00000000_00000000_00000000 // 00_00          nop
00000000_10100011_00001000_00101010 // 00_00    L2:   slt  $1, $5, $3
00000000_00000000_00000000_00000000 // 00_00          nop

```


00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00010000_00100000_11111111_11110001	// 00_00	beq \$1, \$0, L1
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000101_00110000_10000000	// 00_00	sll \$6, \$5, 2
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_11000010_00110000_00100000	// 00_00	add \$6, \$6, \$2
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
10001100_11000111_00000000_00000000	// 00_00	lw \$7, 0(\$6)
00000000_10100000_01000000_00100000	// 00_00	add \$8, \$5, \$0
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000001_00000100_01001000_00100010	// 00_00	L3: sub \$9, \$8, \$4
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000101_00100000_00000000_00011010	// 00_00	bltz \$9, L4
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00001001_01010000_10000000	// 00_00	sll \$10, \$9, 2
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_01001010_01010000_00100000	// 00_00	add \$10, \$10, \$2
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
10001101_01001011_00000000_00000000	// 00_00	lw \$11, 0(\$10)
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_00000000_00000000_00000000	// 00_00	nop
00000000_11101011_00001000_00101010	// 00_00	slt \$1, \$7, \$11
00000000_00000000_00000000_00000000	// 00_00	nop

```

00000000_00000000_00000000_00000000 // 00_00      nop
00000000_00000000_00000000_00000000 // 00_00      nop
00010000_00100000_00000000_00001000 // 00_00      beq  $1, $0, L4
00000000_00000000_00000000_00000000 // 00_00      nop
10101100_11001011_00000000_00000000 // 00_00      sw   $11, 0($6)
00000001_01000000_00110000_00100000 // 00_00      add  $6, $10, $0
00000001_00100000_01000000_00100000 // 00_00      add  $8, $9, $0
00000000_00000000_00000000_00000000 // 00_00      nop
00001000_00000000_00000000_00100110 // 00_00      j    L3
00000000_00000000_00000000_00000000 // 00_00      nop
10101100_11000111_00000000_00000000 // 00_00  L4:  sw   $7, 0($6)
00100000_10100101_00000000_00000001 // 00_00      addi $5, $5, 1
00000000_00000000_00000000_00000000 // 00_00      nop
00001000_00000000_00000000_00010011 // 00_00      j    L2
00000000_00000000_00000000_00000000 // 00_00      nop
00000000_00000000_00000000_00001101 // 00_00  done: break

```

```

G:\Downloads\WCA-project\project3\prj3_PCPU_2025_nop>run.bat
G:\Downloads\WCA-project\project3\prj3_PCPU_2025_nop>vvp tb_PC.o -fst
WARNING: Memory_F.v:42: $readmemb(M_TEXT_SEG.txt): Not enough words in the file for the requested range [0:1023]
WARNING: Memory_F.v:43: $readmemb(M_TEXT_FWD.txt): Not enough words in the file for the requested range [0:1023]
WARNING: Memory_F.v:121: $readmemb(M_DATA_SEG.txt): Not enough words in the file for the requested range [0:1023]
-----
| H020-3-1647-01: Computer Architecture |
|                               CE.KW.AC.KR                               |
-----
FST info: dumpfile tb_PC.vcd opened for output.
-----
Break signal: 1, # of Cycles:      3275
-----
tb_PipelinedCPU_P.v:85: $finish called at 32865000 (1ps)
G:\Downloads\WCA-project\project3\prj3_PCPU_2025_nop>FC /L mem_dump_SS.txt mem_dump.txt
파일을 비교합니다: mem_dump_SS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.
G:\Downloads\WCA-project\project3\prj3_PCPU_2025_nop>FC /L reg_dump_SS.txt reg_dump.txt
파일을 비교합니다: reg_dump_SS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.

```

a 시뮬레이션 결과 위와 같이 사이클 수가 5227(4 nops)->3275로 줄어들었다.

b. 기존 어셈블리코드에서 Forward 제어 신호를 추가하여 더 많은 NOP를 제거하여, 재구성된 어셈블리 코드 시뮬레이션위한 radix sort 어셈블리 코드를 변환한 M_TEXT_SEG.txt에 주석을 작성한 결과이다.

```

00100000_00000010_00100000_00000000 // 00_00      main: addi $2, $0, 0x2000
00100000_00000011_00000000_00010000 // 00_00      addi $3, $0, 16
00000000_00000011_00100000_00100000 // 00_00      add  $4, $0, $3
00000000_00000000_00000000_00000000 // 00_00      nop

```

00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000100_00100000_01000010	//	00_00	L1:	srl \$4, \$4, 1
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00011000_10000000_00000000_00111101	//	00_00		blez \$4, done
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_10000000_00101000_00100000	//	00_00		add \$5, \$4, \$0
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_10100011_00001000_00101010	//	00_00	L2:	slt \$1, \$5, \$3
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00010000_00100000_11111111_11110001	//	00_00		beq \$1, \$0, L1
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000101_00110000_10000000	//	00_00		sll \$6, \$5, 2
00000000_11000010_00110000_00100000	//	00_00		add \$6, \$6, \$2
10001100_11000111_00000000_00000000	//	00_00		lw \$7, 0(\$6)
00000000_10100000_01000000_00100000	//	00_00		add \$8, \$5, \$0
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00000001_00000100_01001000_00100010	//	00_00	L3:	sub \$9, \$8, \$4
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00000000_00000000_00000000	//	00_00		nop
00000101_00100000_00000000_00011010	//	00_00		bltz \$9, L4
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_00001001_01010000_10000000	//	00_00		sll \$10, \$9, 2
00000000_01001010_01010000_00100000	//	00_00		add \$10, \$2, \$10
10001101_01001011_00000000_00000000	//	00_00		lw \$11, 0(\$10)
00000000_00000000_00000000_00000000	//	00_00		nop
00000000_11101011_00001000_00101010	//	00_00		slt \$1, \$7, \$11
00000000_00000000_00000000_00000000	//	00_00		nop

```

00000000_00000000_00000000_00000000 // 00_00      nop
00000000_00000000_00000000_00000000 // 00_00      nop
00010000_00100000_00000000_00001000 // 00_00      beq  $1, $0, L4
00000000_00000000_00000000_00000000 // 00_00      nop
10101100_11001011_00000000_00000000 // 00_00      sw   $11, 0($6)
00000001_01000000_00110000_00100000 // 00_00      add  $6, $10, $0
00000001_00100000_01000000_00100000 // 00_00      add  $8, $9, $0
00000000_00000000_00000000_00000000 // 00_00      nop
00001000_00000000_00000000_00100110 // 00_00      j    L3
00000000_00000000_00000000_00000000 // 00_00      nop
10101100_11000111_00000000_00000000 // 00_00      L4:  sw   $7, 0($6)
00100000_10100101_00000000_00000001 // 00_00      addi $5, $5, 1
00000000_00000000_00000000_00000000 // 00_00      nop
00001000_00000000_00000000_00010011 // 00_00      j    L2
00000000_00000000_00000000_00000000 // 00_00      nop
00000000_00000000_00000000_00001101 // 00_00      done: break

```

```

G:\Downloads\WCA-project\project3\prj3_PCPU_2025_forwarding>run.bat
G:\Downloads\WCA-project\project3\prj3_PCPU_2025_forwarding>vvp tb_PC.o -fst
WARNING: Memory_F.v:42: $readmemb(M_TEXT_SEG.txt): Not enough words in the file for the requested range [0:1023].
WARNING: Memory_F.v:43: $readmemb(M_TEXT_FWD.txt): Not enough words in the file for the requested range [0:1023].
WARNING: Memory_F.v:121: $readmemb(M_DATA_SEG.txt): Not enough words in the file for the requested range [0:1023].
-----
| H020-3-1647-01: Computer Architecture |
|                               CE.KW.AC.KR |
-----
FST info: dumpfile tb_PC.vcd opened for output.
-----
Break signal: 1, # of Cycles:      2378
-----
tb_PipelinedCPU_P.v:85: $finish called at 23895000 (1ps)
G:\Downloads\WCA-project\project3\prj3_PCPU_2025_forwarding>FC /L mem_dump_SS.txt mem_dump.txt
파일을 비교합니다: mem_dump_SS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.
G:\Downloads\WCA-project\project3\prj3_PCPU_2025_forwarding>FC /L reg_dump_SS.txt reg_dump.txt
파일을 비교합니다: reg_dump_SS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.

```

b 시뮬레이션 결과, 위와 같이 사이클 수가 5227(4 nops)->3275(최소 nops) -> 2378(forwarding 추가)로 줄어 들었다

3. 문제점 및 고찰

본 프로젝트는 MIPS 파이프라인 구조를 기반으로, 주어진 shell_sort 어셈블리 코드에서 발생 가능한 data 및 control hazard를 식별하고, 이를 해결하기 위한 nop 삽입 및 ALU 포워딩 적용 방식을 설계하고 구현하는 것을 목표로 하였다. 초기에는 각 명령어 사이에 4개의 nop이 삽입되어 모든 하자드를 제거한 상태의 시뮬레이션을 진행하였고, 이 경우 총 사이클 수는 5227이었다. 이는 파이프라인에 항상 하나의 명령어만 존재하도록 하여 하자드 발생을 원천적으로 차단한 것이다.

이후 a 시뮬레이션에서는 명령어 사이에 3개의 nop만 삽입해도 hazard가 발생하지 않았다. 이는 앞선 명령어가 WB 단계에서 레지스터에 값을 write할 때, 다음 명령어는 아직 IF 단계에 있기 때문이다. 실제로 레지스터 값을 읽는 시점은 ID 단계이므로, WB가 완료된 직후 ID가 시작되도록 nop을 3개만 삽입하면 데이터가 정상적으로 전달된다. 이와 같이 파이프라인의 단계 차이를 고려한 최소한의 stall만으로도 hazard를 안정적으로 회피할 수 있다.

b 시뮬레이션에서는 포워딩 제어 신호를 활용해 더 많은 nop을 제거하였다. 그 과정에서 hazard의 종류(raw, load-use, control)를 정확히 구분하고, 포워딩이 가능한 상황과 그렇지 않은 상황(branch ID stage에서 read가 필요한 명령어 등)을 구별하는 훈련이 되었다. 특히 load-use hazard의 경우, lw 명령어가 MEM 단계가 완료되어야 레지스터 값이 준비되기 때문에 forwarding으로만 해결할 수 없어 최소 1개의 nop이 반드시 필요하다는 점을 여러 번 시뮬레이션을 통해 체감하며 이해하게 되었다.

또한, 데이터 하자드가 단순히 연속된 명령어 사이(1↔2)뿐 아니라, 비연속적인 경우(1↔3)에서도 발생할 수 있으며, 중간 명령어(2)가 동일한 레지스터를 write하는 경우에는 1↔3 간 의존성은 무시 가능하다는 점도 확인하였다.

branch 또는 jump 명령어를 사이에 두고 발생하는 data hazard의 처리는 단순한 연속 명령어 간 hazard보다 훨씬 복잡하게 느껴졌다. 포워딩은 일반적으로 직전 명령어의 결과를 기준으로 동작하기 때문에, 만약 이전 명령어가 분기(branch)나 점프(jump) 명령어라면 포워딩 경로가 끊기거나 올바른 값이 전달되지 않는 경우가 발생한다. 예를 들어, add 명령어에서 계산된 값을 이후 srl 명령어가 참조해야 할 때, 중간에 beq가 위치하면 포워딩이 add가 아닌 beq로부터 시도되어 잘못된 결과를 얻게 되는 경우가 있었다. 이러한 문제는 시뮬레이션을 여러 번 수행하면서 직접 확인하게 되었으며, 포워딩이 어떤 명령어 기준으로 동작하는지, 그리고 중간에 control flow 명령어가 개입될 경우 어떻게 동작을 정의할지는 구현 방식에 따라 달라질 수 있음을 깨달았다.

본 프로젝트에서 문제점으로 거론할 수 있는 점은, control hazard의 경우, 분기 대상이 ID 단계에서 결정됨에도 불구하고, 본 구현에서는 ID stage로의 포워딩 경로가 제공되지 않아 stall을 제거할 수 없는 구조라는 점이 가장 큰 제한 요소였다. 만약 이 포워딩이 구현되어 있다면, branch 조건 판단에 필요한 데이터 의존성을 해결할 수 있어 전체 nop 삽입 수를 크게 줄이고 실행 시간을 단축할 수 있었을 것이다.

한편, 어셈블리 코드를 M_TEXT_SEG.txt 형태의 머신 코드로 변환하는 과정에서 mars를 사용하지 않고 메모장으로 직접 작성할 경우 시뮬레이션이 정상 작동하지 않았다. 정확한 원인은 확인되지 않았지만, 포맷 오류나 인코딩 문제로 인해 시뮬레이터가 바이너리 코드를 인식하지 못했을 가능성이 있다. 이러한 시행착오를 통해 명령어 작성, 포워딩 제어 신호 지정, 시뮬레이션 결과 해석에 이르기까지 파이프라인 구조의 복잡성과 민감성을 체계적으로 체험할 수 있었다.