컴퓨터구조실험 보고서

Project #3 - Pipeline Architecture

과 목 컴퓨터구조실험

담당교수 이성원교수님

학 과 컴퓨터정보공학부

학 번 2021202058

이 름 송채영

1. Introduction

pipelined computer Architectures 는 명령어 실행을 여러 단계로 나누어 성능을 향상시키도록 설계되어 있다. 하지만 이 단계가 중복될 경우 파이프라인 단계에서 hazard 가 발생할 수 있다. 이번 프로젝트는 pipelined computer Architectures 에서 발생할 수 있는 hazard 에 대해 알아보고, hazard 가 발생한 경우 이를 피하기 위한 방법을 찾아 프로그램의 performance 를 향상시키는 것이다.

Hazard 는 크게 세 가지 유형으로 나눌 수 있다. 먼저 Structural Hazard, 구조적 Hazard 는 요구되는 하드웨어 resource 가 동시에 사용될 수 없는 상황에 발생한다. 이에 대한 예시로 두 개의 명령어가 동일한 장치에 동시에 access 하려고 할 때 발생할 수 있다. 두 번째로 Data Hazard, 데이터 Hazard 가 있다. Data Hazard 는 명령어 간의 데이터 의존성으로 인해 발생하며, Data hazard 는 Read-after-write (RAW Hazard), Write-after-read (WAR hazard), Write-after-write (WAW Hazard)로 나뉠 수 있다. 먼저 RAW Hazard 는 이전 명령어의 결과 데이터를 다음 명령어에서 read 하려고 할 때 이전 명령어의 write 가 완료되지 않았기 때문에 필요한 데이터를 사용할 수 없어 hazard 가 발생한다. WAR hazard 는 이전 명령어가 read 하는 동안 다음 명령어가 해당 데이터를 write 하려고 할 때 이전 명령어의 데이터를 사용하는 동안 해당 데이터를 변경하려고 하기 때문에 hazard 가 발생한다. WAW hazard 는 두개의 연속적인 명령어가 동일한 레지스터나 메모리 위치에 쓰려고 할 때 발생한다. 마지막으로 Control Hazard, 제어 Hazard 가 있다. Control Hazard 는 branch instruction(분기 명령어), branch 나 jump 명령어를 처리하는 동안 분기 명령어에서 다음 명령어의 흐름이 두 가지로 나뉘기 때문에 발생한다.

Hazard 로 인한 지연을 최소화하기 위한 방법에는 크게 H/W Forwarding (Hardware Forwarding)과 S/W Coding (Software Coding)으로 나눌 수 있다. H/W Forwarding 은 하드웨어 수준에서 데이터 Hazard 를 해결하기 위한 방법이며, S/W Coding 은 소프트웨어 수준에서 Hazard 를 해결하는 방법을 말한다. 이번 프로젝트에서는 data Hazard 와 control Hazard 만 발생하므로 두 가지 경우로 나누어 설명하겠다.

먼저 데이터 Hazard 의 경우이다. H/W forwarding 의 경우 한 명령어의 실행이 이전 명령어의 결과에 의존할 때 hazard 가 발생하며 이전 명령어의 결과를 다음 명령어로 전달해 데이터 Hazard 를 해결할 수 있다. 동작방식은 다음과 같다. 먼저 Execution stage 에서 Memory stage 로의 forwarding 의 경우 Execution stage 에서 계산된 결과 데이터를 Memory stage 로 바로 전달한다. 이를 통해 Memory stage 에서 이전 명령어의 결과를 사용하는 다음 명령어가 데이터를 지연 없이 사용할 수 있다. Memory stage 에서 Execution stage 로의 forwarding 의 경우 Memory stage 에서 메모리 접근이 필요한 명령어의 결과 데이터를 Execution stage 로 전달한다. 이를 통해 Execution stage 에서 이전 명령어의 결과를 사용하는 다음 명령어가 데이터를 지연 없이 사용할 수 있다. Memory/Write Back stage 에서 Execution stage 로 forwarding 할 경우 Memory/Write

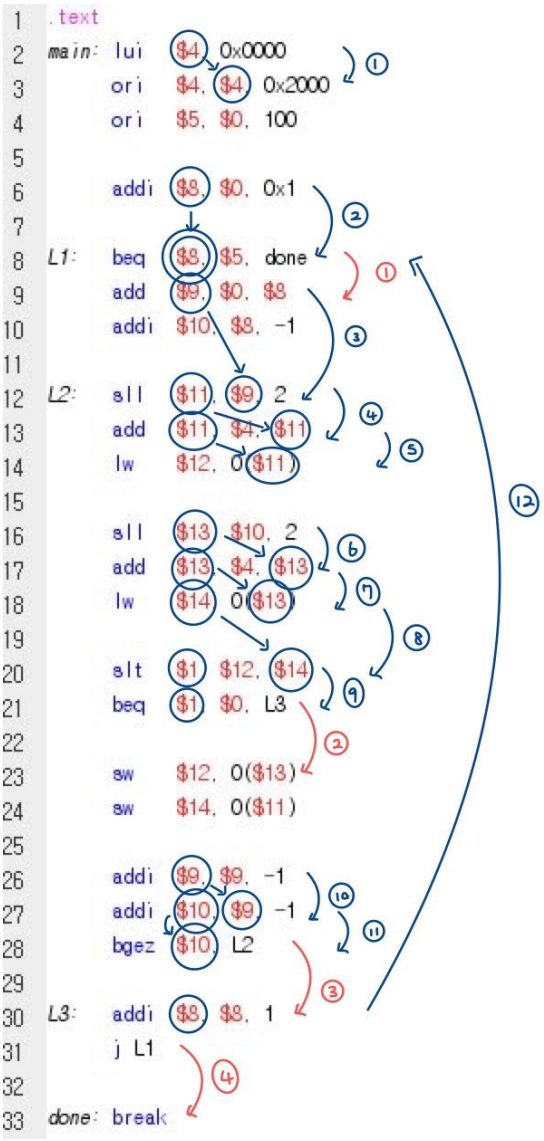
Back stage 에서 메모리 접근이 필요하지 않는 명령어의 결과 데이터를 Execution stage 로 전달한다. 이를 통해 Execution stage 에서 이전 명령어의 결과를 사용하는 다음 명령어가 데이터를 지연 없이 사용할 수 있다. S/W coding 의 경우 컴파일러가 명령어를 재배치하거나 RAW Hazard 가 발생하는 경우는 NOP 명령어를 넣어주어 Hazard 를 해결할 수 있다. NOP 는 아무 작업도 수행하지 않는다는 의미의 명령어로 해당 stage 에서 아무 동작도 하지 않고 대기 상태로 만든다. NOP 를 추가하는 위치는 Hazard 가 발생한 지점이며, 주로 RAW Hazard 가 발생한 경우에 추가해주어 이전 명령어의 실행이 완료될 때까지 대기하도록 하여 해결한다.

다음으로 control Hazard 의 경우이다. H/W forwarding 의 경우 branch 명령어에 의해 발생하며 branch 결과를 미리 알 수 있도록 한다. branch 명령어 이후의 명령어들이 branch 결과를 기다리는 동안, branch prediction 과 함께 branch 결과를 파이프라인 stage 간에 전달하여 지연을 최소화할 수 있다. Branch Prediction 은 control Hazard 를 해결하기 위해 사용되는 기법으로 branch 명령어가 실행될 때 해당 분기가 어느 방향으로 진행될지를 사전에 예측하는 과정을 말한다. S/W forwarding 의 경우 branch 명령어의 실행 순서를 조정하거나 branch prediction 이 틀렸을 경우를 대비하여 NOP 를 추가하여 실제 branch 결과가 도착할 때까지 대기해 올바른 branch 명령어가 실행되도록 한다.

하지만 NOP 를 삽입하는 것은 파이프라인의 성능을 저하시킬 수 있으므로 NOP 를 최대한 줄여 Hazard 를 처리하여야 한다.

2. 결과 화면

과제로 주어진 기존 insertion_sort.asm 코드에서 hazard 가 발생한 부분을 표시하였다. 파란색은 Data Hazard 가 발생한 부분을 나타냈으며, 빨간색은 Control Hazard 가 발생한 부분을 나타내었다.



기존 어셈블리코드에서 NOP 를 제거한 시뮬레이션과 기존 어셈블리코드에서 Forward 제어 신호를 추가하여 더 많은 NOP 를 제거하여, 재구성된 어셈블리 코드 시뮬레이션을 살펴보면 다음과 같다. 기존 어셈블리코드에서 NOP 를 제거한 시뮬레이션을 a, 기존 어셈블리코드에서 Forward 제어신호를 추가해 더 많은 NOP 를 제거하여, 재구성된 어셈블리 코드 시뮬레이션을 b 라고 하겠다.

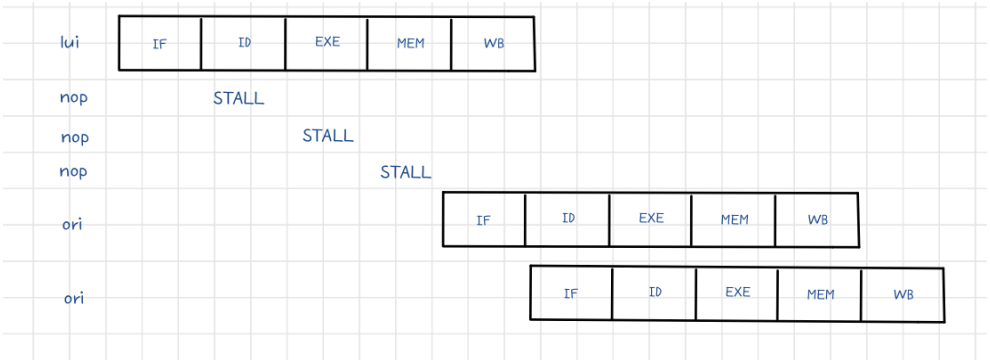
FWD_ALU_Ai 와 FWD_ALU_Bi 는 파이프라인 stage 간 데이터 forwarding 을 제어하는 bit field 를 의미한다. 데이터가 다른 stage 로 전달되는 경로를 지정하는데 사용하며 각각에 대해 설명해보겠다. 우선 FWD_ALU_Ai 는 ALU A 입력에 대한 데이터 forwarding 을 의미한다. 00 일 때는 ID stage 의 register file 에서 EX stage 로 데이터를 전달한다. ID stage 에서 read 한 register 값을 EX stage 의 ALU 의 A 의 입력으로 전달한다. 01 일 때는 MEM stage 의 ALU 출력에서 EX stage 로 데이터를 전달한다. MEM stage 에서 수행된 연산 결과를

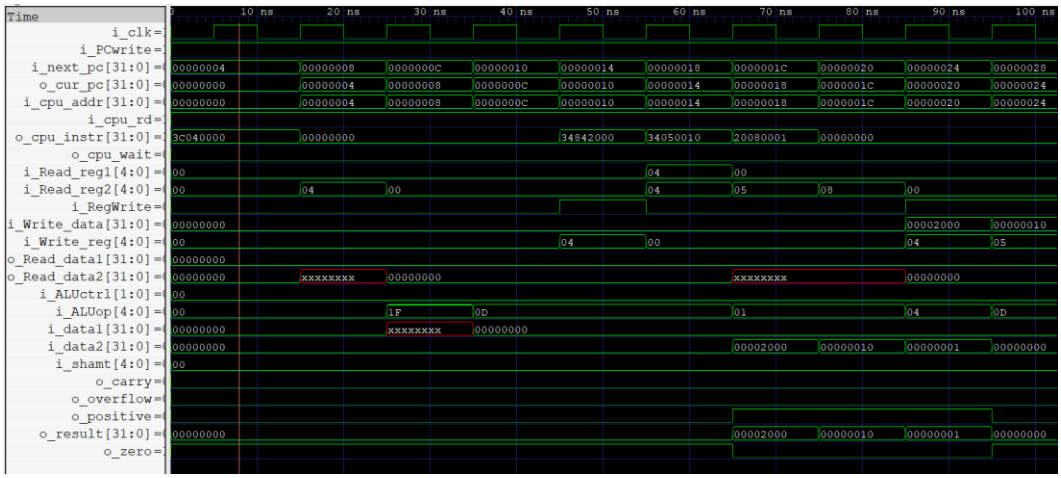
EX stage 의 ALU A 의 입력으로 전달한다. 10 일 때는 WB stage 의 Writeback 데이터에서 EX stage 로 데이터를 전달한다. WB stage 에서 수행된 명령어의 결과를 EX stage 의 ALU A 의 입력으로 전달한다. 다음으로 FWD_ALU_Bi 는 ALU B 입력에 대한 데이터 forwarding 을 의미한다. 00, 01, 10 일 때 FWD_ALU_Ai 와 동일한 내용을 가지며 ALU 의 B 의 입력으로 전달한다. 두 개의 signal 모두 11 일 때는 없다. (아래에서 M_TEXT_FWD 내용에 해당함)

①

우선 같은 레지스터에 대해 연속적으로 write 하기 때문에 data hazard 가 발생한다. lui 명령어는 레지스터 $4 에 상위 16 비트를 0x000 으로 load 한다. 그 다음 ori 명령어에서는 $4 의 값을 읽고, 하위 16 비트를 0x2000 으로 load 하며, 결과를 다시 $4 에 저장한다. ori 명령어에서 $4 의 값을 읽기 위해서는 lui 명령어에서 $4 에 쓰여진 값을 가져와야 하지만, lui 명령어의 결과가 아직 $4 에 쓰여지기 전에 ori 명령어에서 $4 를 읽으려고 하기 때문에 data hazard 가 발생한다.

a.

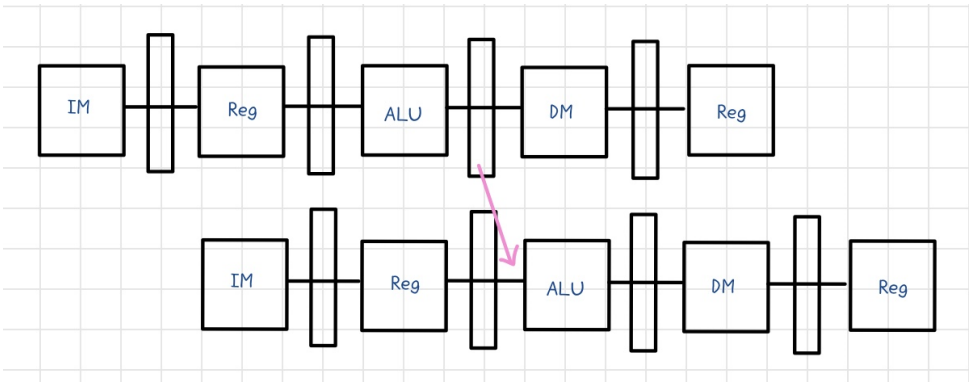




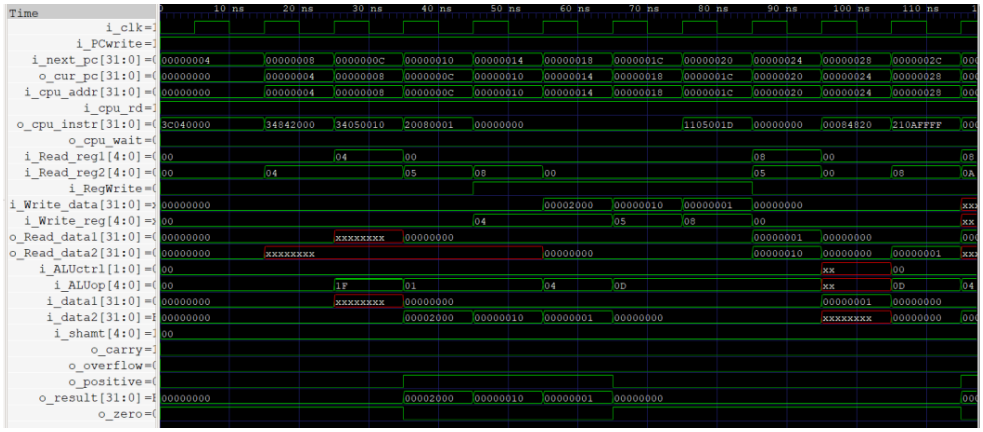
lui 명령어의 결과를 $4 에 쓴 후 ori 명령어에서 $4 를 읽어야 올바른 값이 나오므로, NOP 3 개를 추가해서 해결해주었다. 0x00, 0x04, 0x08, 0xc0, 0x10 은 차례대로 IF, ID(NOP), EX(NOP), MEM(NOP), WB 에 해당한다. lui 명령어의 WB 단계에서 i_RegWrite 가 1 이므로 write 하는 것을 알 수 있다. ori 명령어의 ID 단계인 0x14 에서 $4 에 load 되고, EX 단계인 0x18 에서 0x2000 이 올바르게 들어간 것을 확인할 수 있다.

b.

```
2  main: lui   $4, 0x0000
3        ori   $4, $4, 0x2000
4        ori   $5, $0, 0x10
```

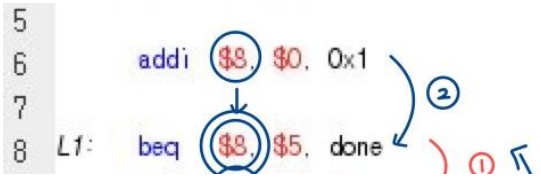



```
01_00  //  0x000
```



lui 명령어의 결과를 $4 에 쓴 후 ori 명령어에서 $4 를 읽어야 올바른 값이 나오므로, forwarding 을 01_00 으로 설정하였다. ALU A 입력에 대해서는 MEM sgate 의 ALU 출력을, ALU B 입력에 대해서는 ID stage 의 레지스터 파일을 사용한다는 의미이다. lui 명령어의 MEM stage 에서 o_result, 즉 output 은 lui 의 연산 값을 나타내며 ori 명령어의 EX stage 에서 input 은 연산을 위해 필요한 데이터를 나타낸다. 이때 input 과 output 이 동일하므로 forwarding 이 잘 된 것을 알 수 있다. (아래의 b 들은 모두 1-b 와 같은 맥락으로 설명하므로 아래에서 자세한 설명은 생략함)

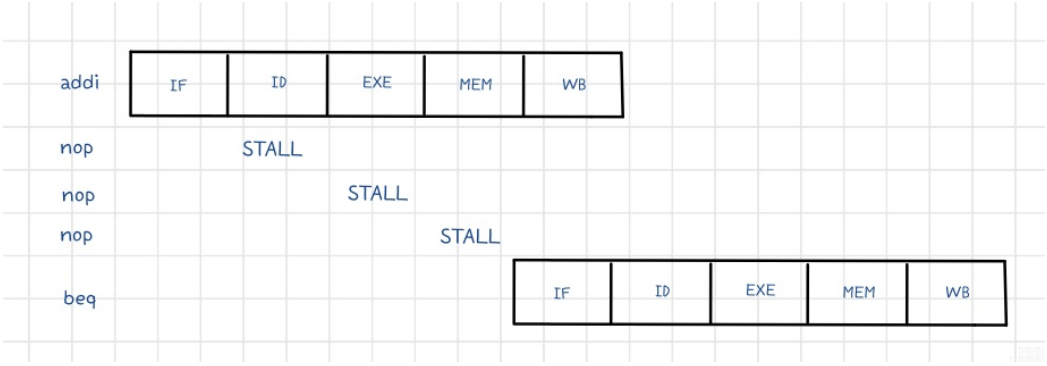
②

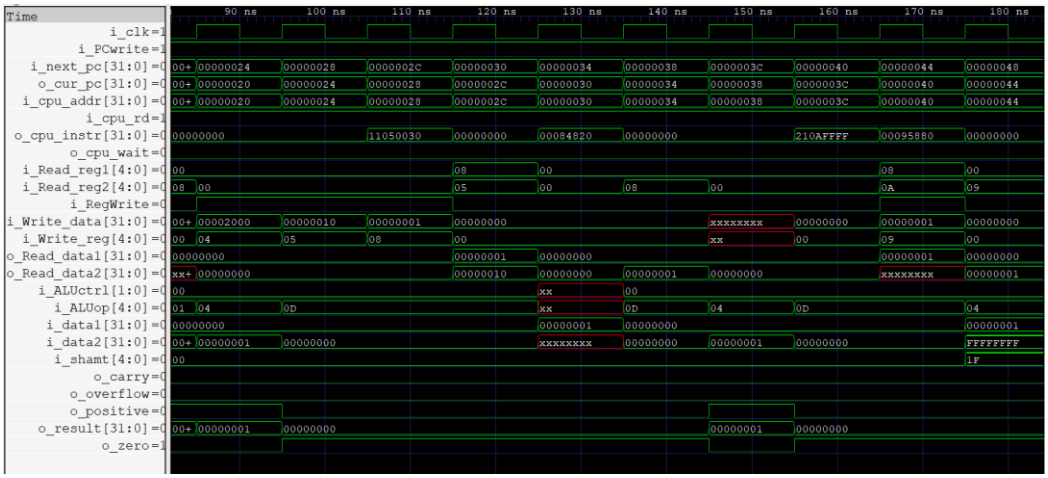


beq 명령어가 이전에 실행된 addi 명령어의 결과를 사용하기 때문에 data hazard 가 발생한다. addi 명령어는 레지스터 $8 에 값 1 을 더한 결과를 저장한다. beq 명령어는 $8 과 $5 의 값을 비교한 후 비교 결과에 따라 분기를 수행하는데 이때 addi 명령어의 실행 결과가 필요하기 때문에 hazard 가 발생한다.

a.

```
9        addi  $8, $0, 0x1
10       nop
11       nop
12       nop
13
14  L1:  beq   $8, $5, done
```

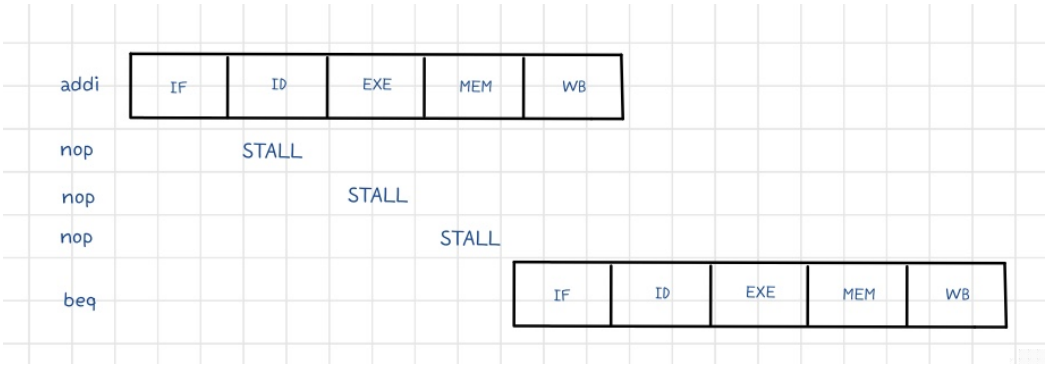




addi 명령어의 실행 결과가 WB 된 후 beq 명령어에서 실행 결과를 사용해야 올바른 값이 나오므로, NOP 3 개를 추가해 해결하였다. beq 명령어의 IF, 0x28 에서 $8 에 0x01 이 들어갔으며 ID, 0x2c 에서 $5 와 $8 에 올바른 값이 들어간 것을 확인할 수 있다.

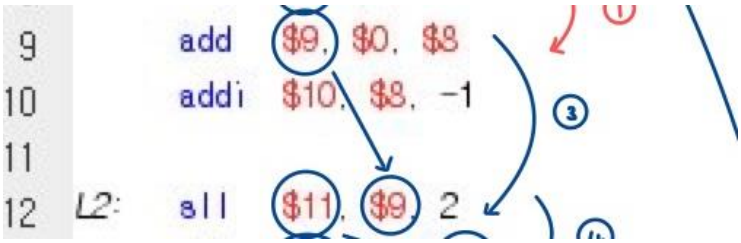
b.

```
6        addi  $8, $0, 0x1
7        nop
8        nop
9        nop
10
11   L1: beq   $8, $5, done
```



forwarding 은 이전에 실행된 명령어의 결과를 바로 다음 명령어로 전달해 데이터 dependency 를 해결하지만, addi 명령어와 beq 명령어 사이에는 한 개의 파이프라인 stage 가 더 있어 데이터를 바로 전달할 수 있는 경로가 없어 NOP 만을 사용하여 hazard 를 해결해야 한다.

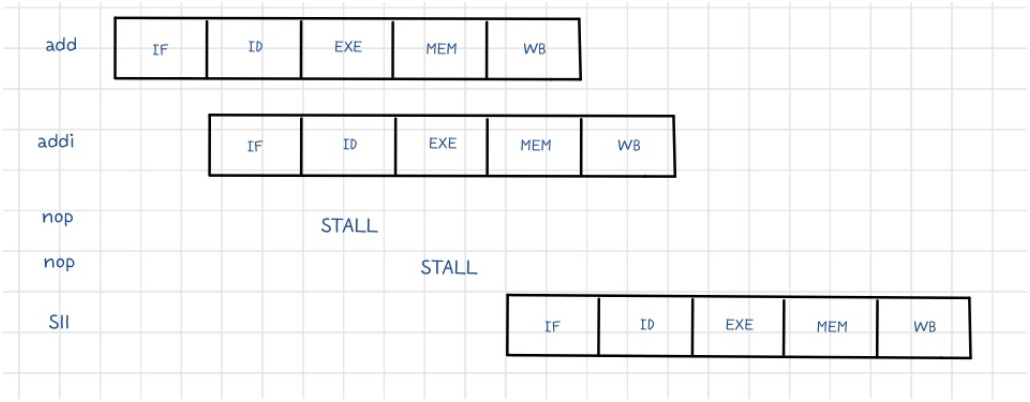
③

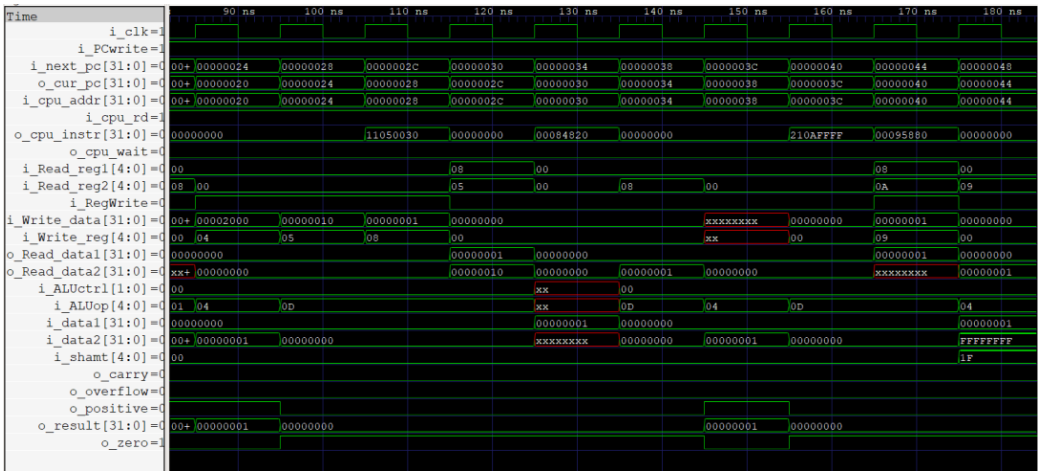


add 명령어의 결과를 사용하는 addi 와 sll 명령어 사이에 데이터 dependency 가 발생하기 때문이다. add 명령어는 $9 에 $0 과 $8 의 값을 더한 결과를 저장한다. addi 명령어는 $10 에 $8 과 -1 을 더한 결과를 저장한다. 이때 addi 명령어는 add 명령어의 실행 결과를 사용하는데 이 부분은 12 번에서 설명하도록 하겠다. sll 명령어는 $11 에 $9 의 값을 왼쪽으로 shift 2 번 한 값을 저장한다. 이때 sll 명령어는 add 명령어의 실행 결과인 $9 의 값을 사용하는데 이때 add 명령어의 결과가 아직 전달되지 않았기 때문에 data hazard 가 발생한다.

a.

```
16          add   $9, $0, $8
17          addi  $10, $8, -1
18          nop
19          nop
20
21    L2:   sll   $11, $9, 2
22          nop
23          nop
24          nop
```

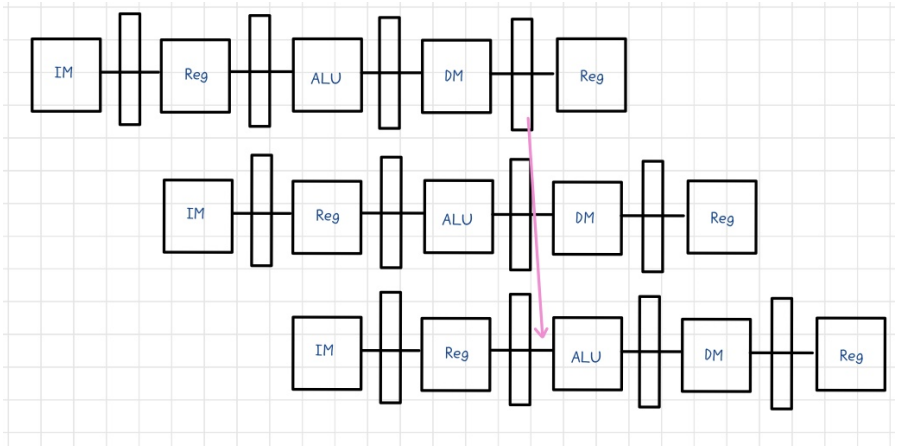




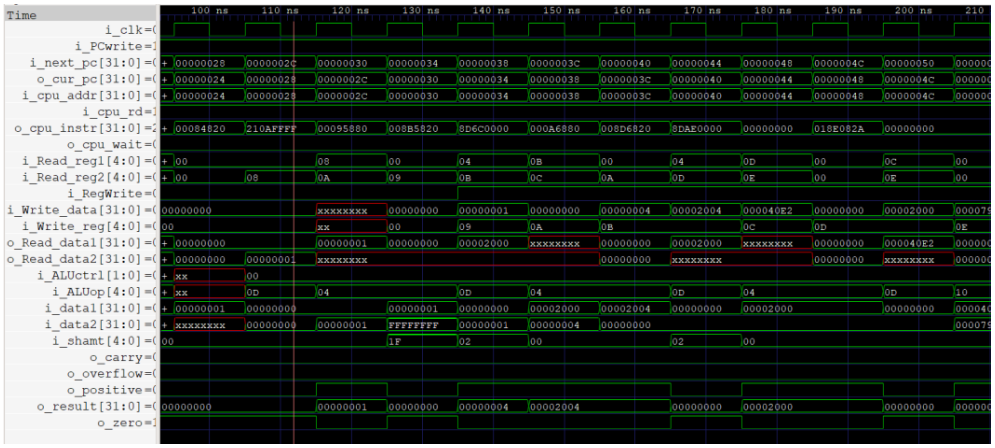
add 명령어의 결과가 WB 된 후 SLL 명령어에서 실행 결과를 사용해야 올바른 값이 나오기 때문에 NOP 2 개를 추가해 해결하였다. add 명령어의 IF 는 0x30 에서, addi 명령어의 IF 는 0x34 에서 이루어진다. SLL 의 IF 는 0x40 에서 이루어지므로 $9 에 0x01 이 잘 들어간 것과 그림에선 잘렸지만 0x48 에서 SLL 의 EX 가 이루어져 값이 잘 들어가는 것을 확인할 수 있다. (4 번 설명에 test 사진 참고)

b.

```
13          add   $9, $0, $8
14          addi  $10, $8, -1
15
16    L2:   sll   $11, $9, 2
```




```
00 10  //  0x028
```



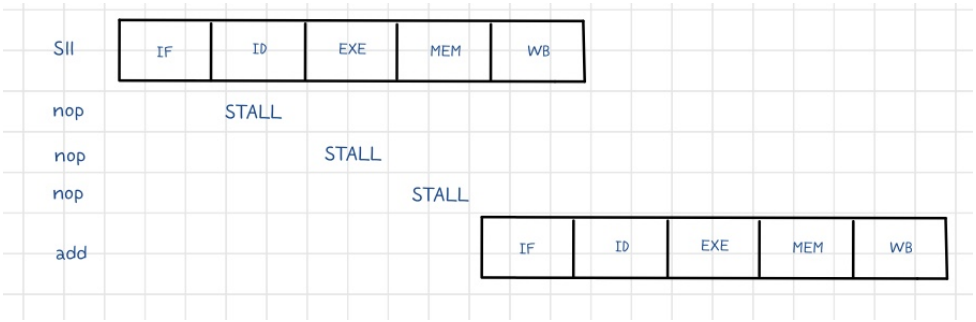
forwarding 을 00_10 으로 설정해 add 명령어의 결과가 addi 명령어와 SLL 명령어에 바로 전달하여 hazard 를 해결하였다. add 명령어의 결과가 이후 명령어로 전달되기 때문에 NOP 명령어를 추가하지 않았다. 이전 명령어의 output 과 sll 명령어의 input 이 동일하므로 forwarding 이 잘 된 것을 알 수 있다.

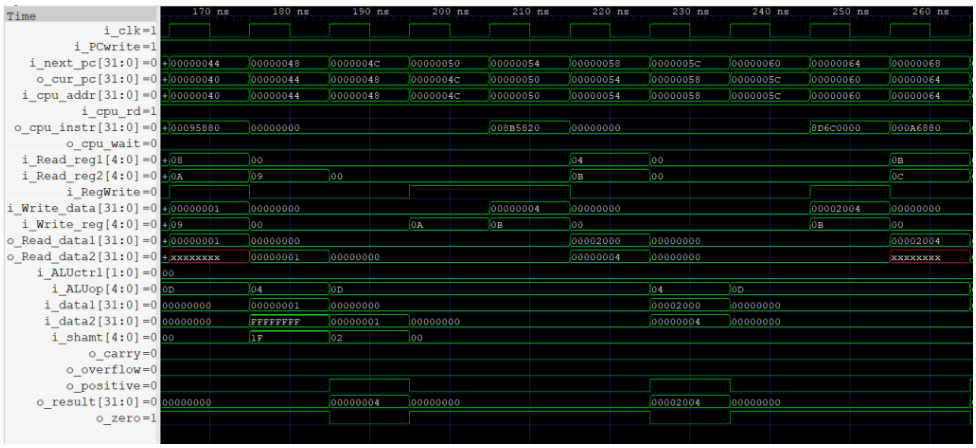
④

SLL 명령어의 결과를 add 명령어에서 사용하기 때문에 data hazard 가 발생한다. sll 명령어는 $11 에 $9 의 값을 왼쪽으로 2 bit shift 한 결과를 저장한다. 이후 add 명령어는 $11 에 $4 와 $11 을 더한 결과를 저장하는데 이때 이전에 실행된 sll 명령어의 결과 $11 을 사용하기 때문이다.

a.

```
21  L2:   sll   $11, $9, 2
22        nop
23        nop
24        nop
25        add   $11, $4, $11
```

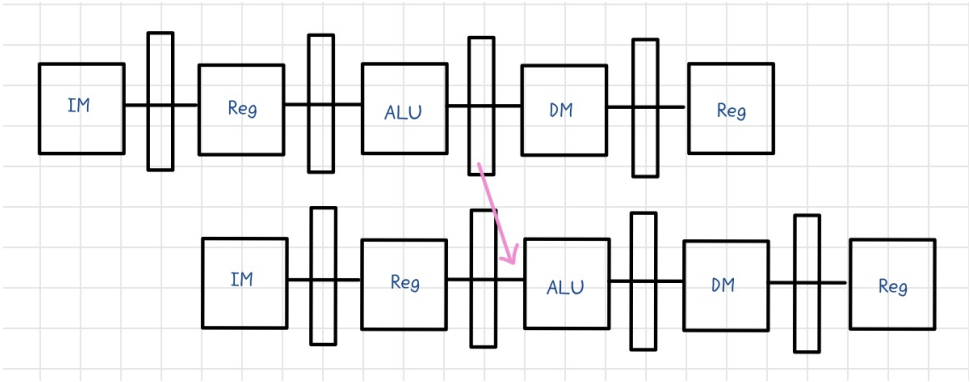




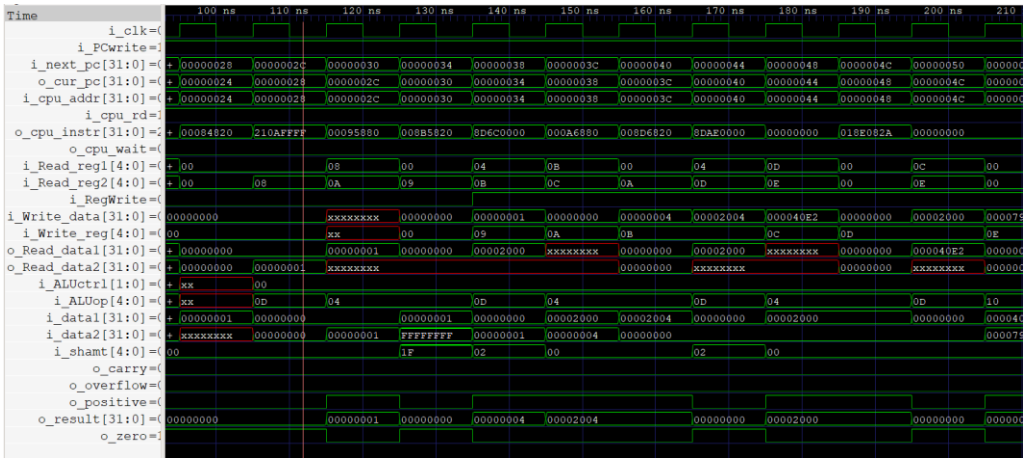
sll 명령어의 결과가 WB 된 후 add 명령어에서 실행 결과를 사용해야 올바른 값이 나오기 때문에 NOP 3 개를 추가해 해결하였다. sll 명령어의 IF 는 0x40 에서, add 명령어의 IF 는 0x50 에서 이루어진다. 0x54, add 의 ID 에서 sll 의 연산 결과가 $11 에 잘 저장되어 add 명령어의 ALU 로 알맞은 값이 들어가는 것을 확인할 수 있다.

b.

```
16  L2:   sll   $11, $9, 2
17        add   $11, $4, $11
```

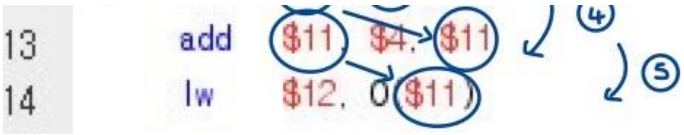


00_01 // 0x02C



forwarding 을 00_01 로 설정해 sll 명령어의 결과가 add 명령어에 바로 전달하게끔 하여 hazard 를 해결하였다. 이전 명령어의 output 과 add 명령어의 input 이 동일하므로 forwarding 이 잘 된 것을 알 수 있다.

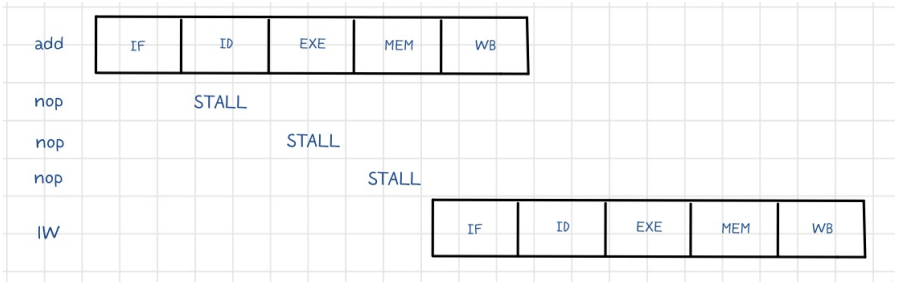
⑤

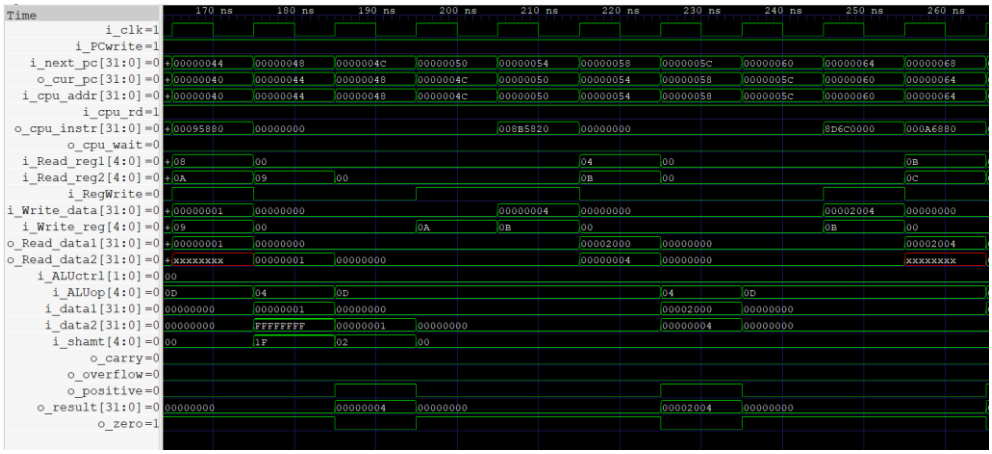


add 명령어의 결과를 lw 명령어가 사용하기 때문에 data hazard 가 발생한다. add 명령어는 $11 에 $4 와 $11 을 더한 결과를 저장한다. lw 명령어는 주어진 메모리 주소에서 데이터를 load 해 $12 에 저장하기 때문에 lw 명령어는 이전에 실행된 add 명령어의 결과 $11 을 사용하기 때문이다.

a.

```
25      add   $11, $4, $11
26      nop
27      nop
28      nop
29      lw    $12, 0($11)
```

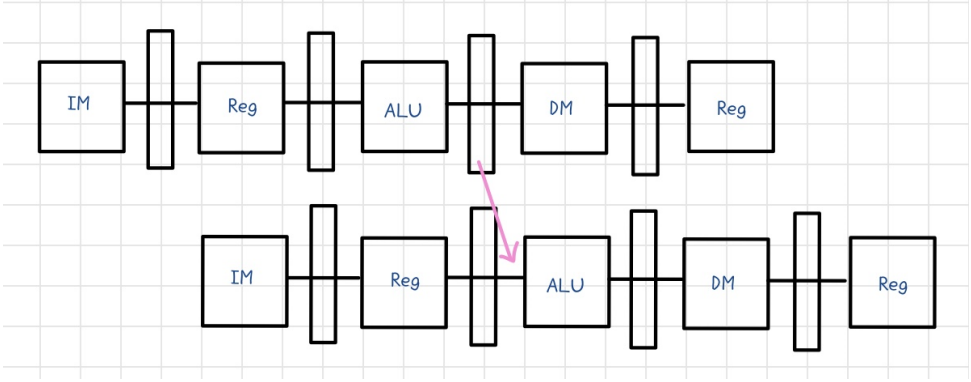




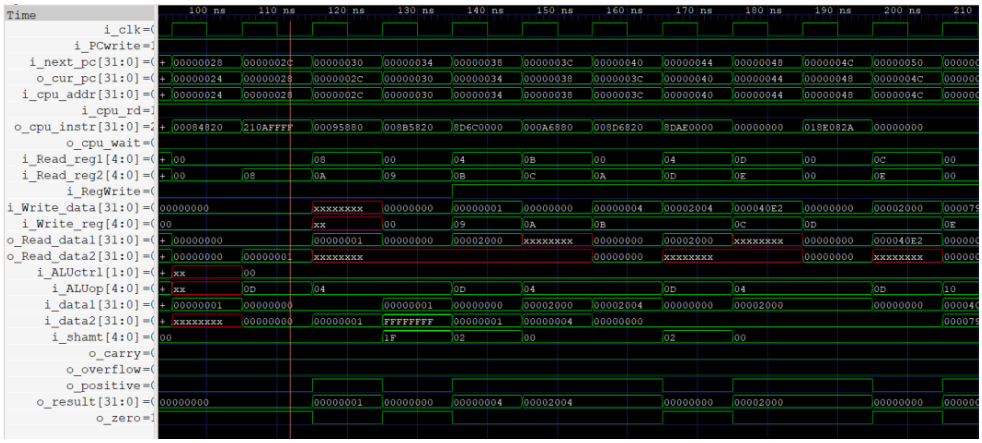
add 명령어의 결과가 WB 된 후 lw 명령어에서 실행 결과를 사용해야 올바른 값이 나오기 때문에 NOP 를 3 개를 추가해 해결하였다. add 명령어의 IF 는 0x50 에서, lw 명령어의 IF 는 0x60 에서 이루어진다. NOP 를 통해 hazard 를 해결한 것을 add 명령어의 WB 되고 $11 을 사용하는 것을 통해 확인할 수 있다.

b.

```
17      add   $11, $4, $11
18      lw    $12, 0($11)
```



```
01_00  //  0x030
```



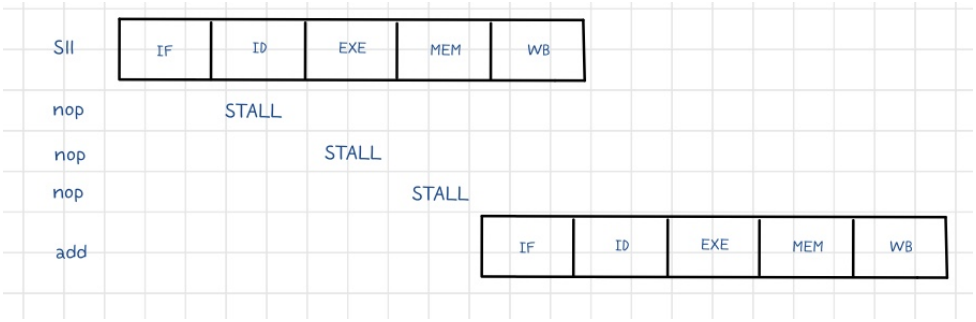
forwarding 을 01_00 으로 설정해 add 명령어의 결과가 lw 명령어에 바로 전달하게끔 하여 hazard 를 해결하였다. 이전 명령어의 output 과 lw 명령어의 input 이 동일하므로 forwarding 이 잘 된 것을 알 수 있다.

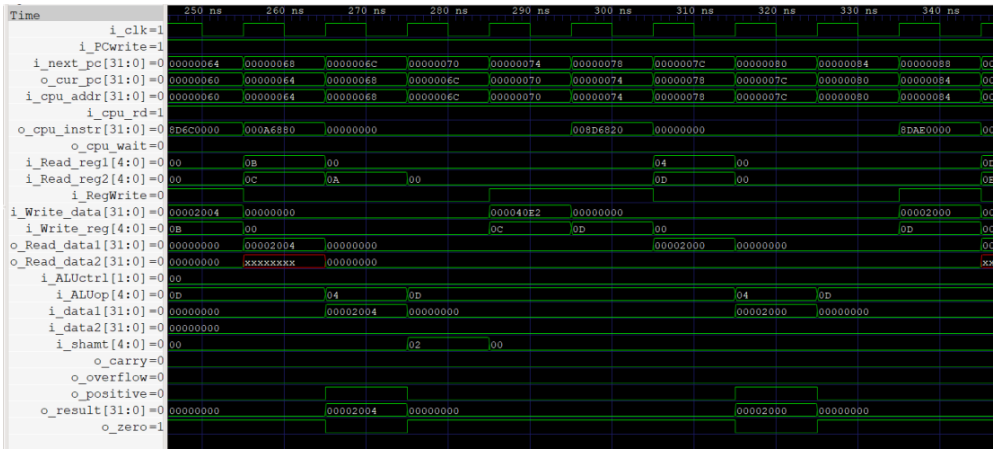
⑥

sll 명령어의 결과를 add 명령어가 사용하기 때문에 data hazard 가 발생한다. sll 명령어는 $13 에 $10 의 값을 왼쪽으로 2bit shift 한 결과를 저장한다. 이후 add 명령어는 $13 에 $4 와 $13 을 더한 값을 저장하는데 이때 add 명령어는 이전에 실행된 sll 명령어의 결과인 $13 을 사용하기 때문이다.

a.

```
31  sll  $13, $10, 2
32  nop
33  nop
34  nop
35  add  $13, $4, $13
36  nop
37  nop
38  nop
```

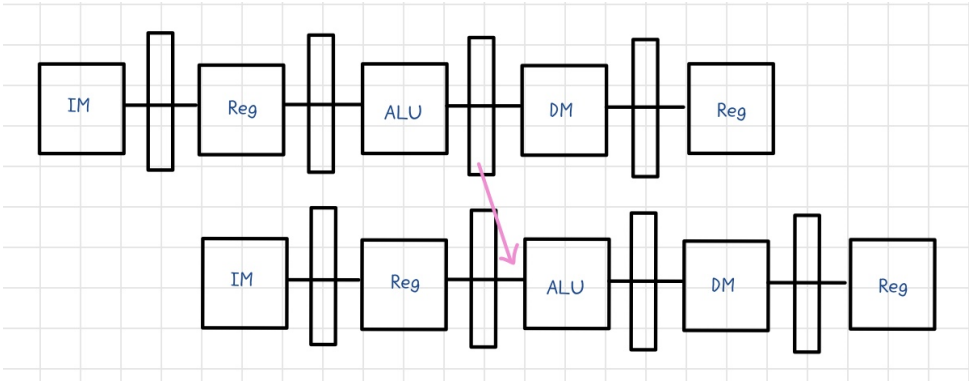




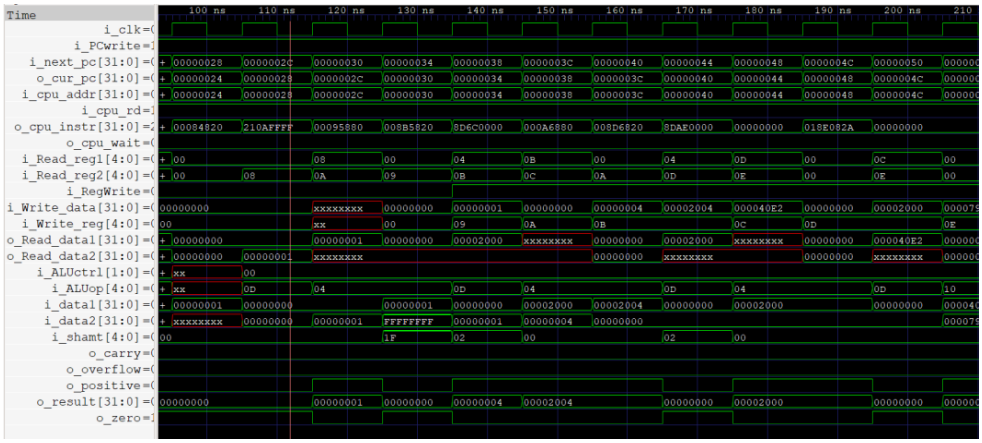
sll 명령어의 결과가 WB 된 후 add 명령어에서 실행 결과를 사용해야 올바른 값이 나오기 때문에 NOP 를 3 개 추가해 해결하였다. sll 명령어의 IF 는 0x64, add 명령어의 IF 는 0x74 에서 이루어진다. NOP 를 통해 hazard 를 해결한 것을 sll 명령어의 WB 되고 $13 을 사용하는 것을 통해 확인할 수 있다.

b.

```
20      sll   $13, $10, 2
21      add   $13, $4, $13
```




```
00_01  //  0x038
```

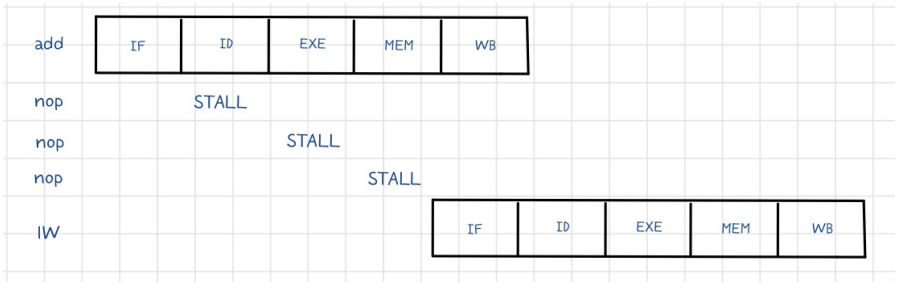


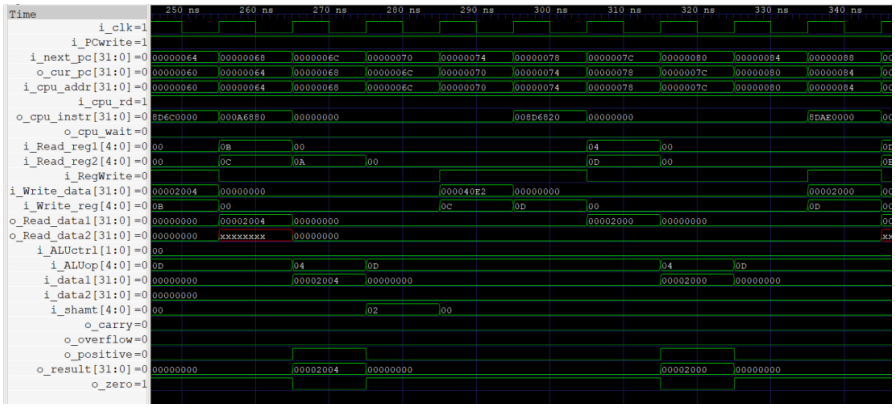
forwarding 을 00_01 로 설정해 sll 명령어의 결과가 add 명령어에 바로 전달하게끔 하여 hazard 를 해결하였다. 이전 명령어의 output 과 add 명령어의 input 이 동일하므로 forwarding 이 잘 된 것을 알 수 있다.

⑦

add 명령어의 결과를 lw 명령어가 사용하기 때문에 data hazard 가 발생한다. add 명령어는 $13 에 $4 와 $13 을 더한 결과를 저장한다. lw 명령어는 주어진 메모리 주소에서 데이터를 load 해 $14 에 저장한다. 이때 lw 명령어는 이전에 실행된 add 명령어의 결과인 $13 을 사용해야 하기 때문이다.

a.

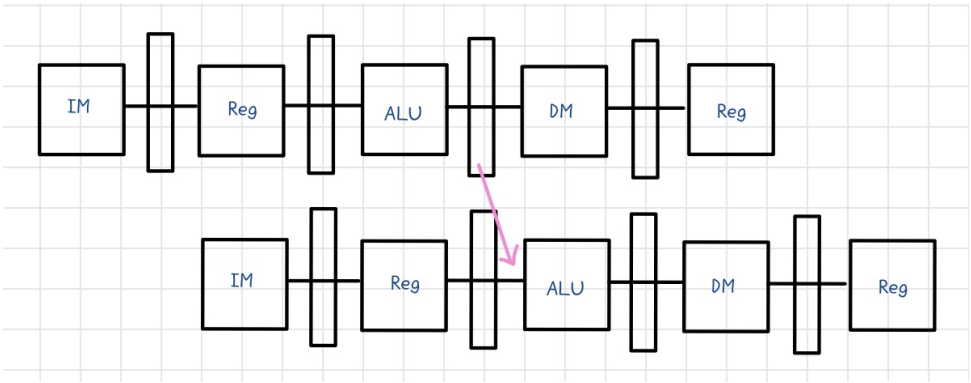




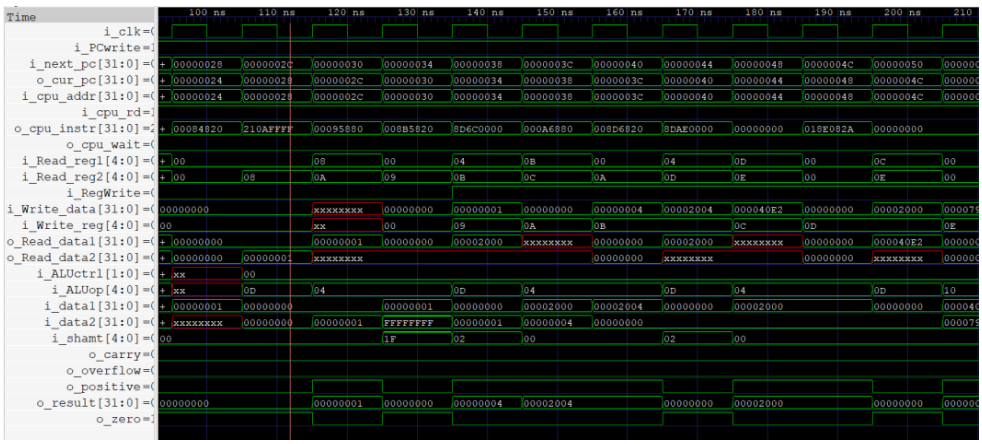
add 명령어의 결과가 WB 된 후 lw 명령어에서 실행 결과를 사용해야 올바른 값이 나오기 때문에 NOP 를 3 개 추가해 해결하였다. add 명령어의 IF 는 0x74, add 명령어의 IF 는 0x84 에서 이루어진다. NOP 를 통해 hazard 를 해결한 것을 add 명령어의 WB 되고 $13 을 사용하는 것을 통해 확인할 수 있다.

b.

```
21      add   $13, $4, $13
22      lw    $14, 0($13)
```

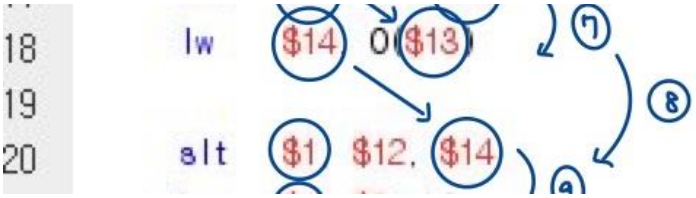



```
01_00  //  0x03C
```



forwarding 을 01_00 으로 설정해 add 명령어의 결과가 lw 명령어에 바로 전달하게끔 하여 hazard 를 해결하였다. 이전 명령어의 output 과 lw 명령어의 input 이 동일하므로 forwarding 이 잘 된 것을 알 수 있다.

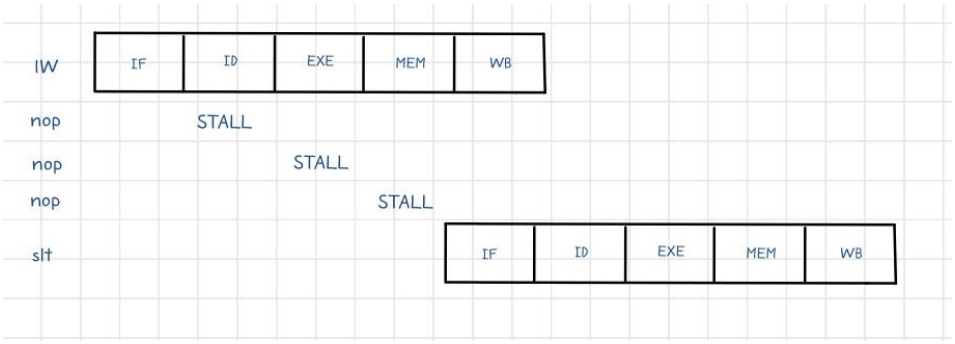
⑧

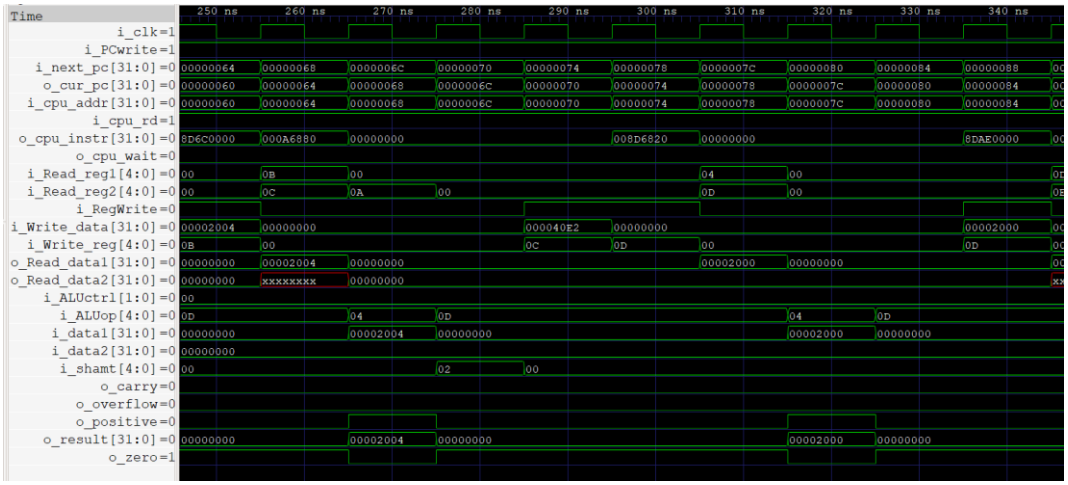


lw 명령어의 결과를 slt 명령어가 사용하기 때문에 data hazard 가 발생한다. lw 명령어는 메모리 주소 $13 에서 데이터를 load 해 $14 에 저장한다. slt 명령어는 $12 와 $14 를 비교해 결과를 $1 에 저장한다. 이때 slt 명령어는 이전에 실행된 lw 명령어의 결과인 $14 를 사용하기 때문이다.

a.

```
39      lw    $14, 0($13)
40      nop
41      nop
42      nop
43
44      slt   $1, $12, $14
```

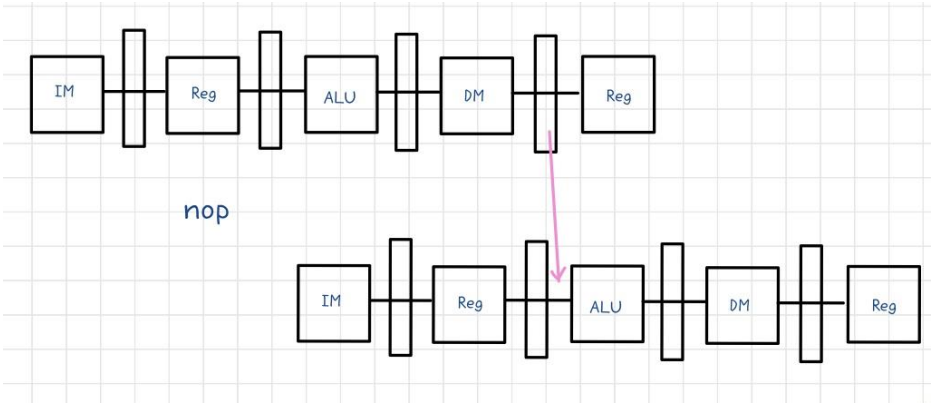




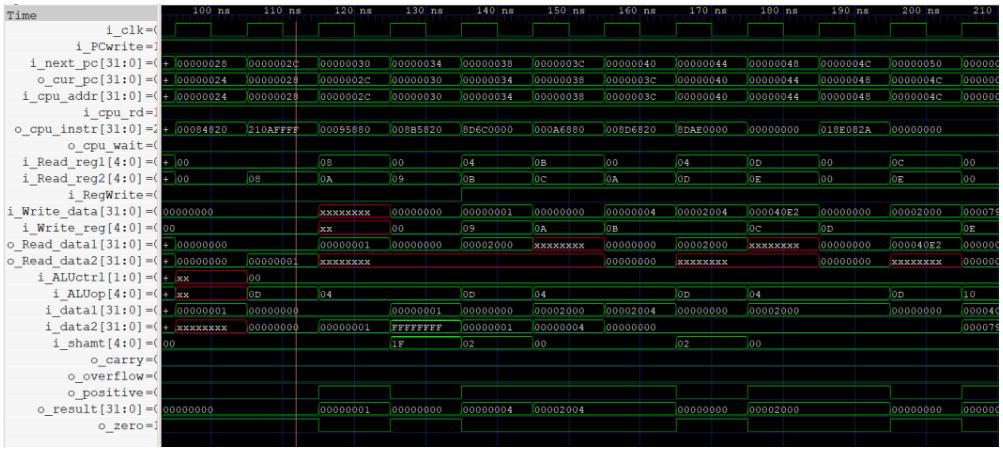
lw 명령어의 결과가 WB 된 후 slt 명령어에서 실행 결과를 사용해야 올바른 값이 나오기 때문에 NOP 를 3 개 추가해 해결하였다. lw 명령어의 IF 는 0x84, slt 명령어의 IF 는 0x94 에서 이루어진다. NOP 를 통해 hazard 를 해결한 것을 lw 명령어의 WB 되고 $14 를 사용하는 것을 통해 확인할 수 있다.

b.

```
22      lw    $14, 0($13)
23      nop
24
25      slt   $1, $12, $14
```




```
00_10  //  0x044
```

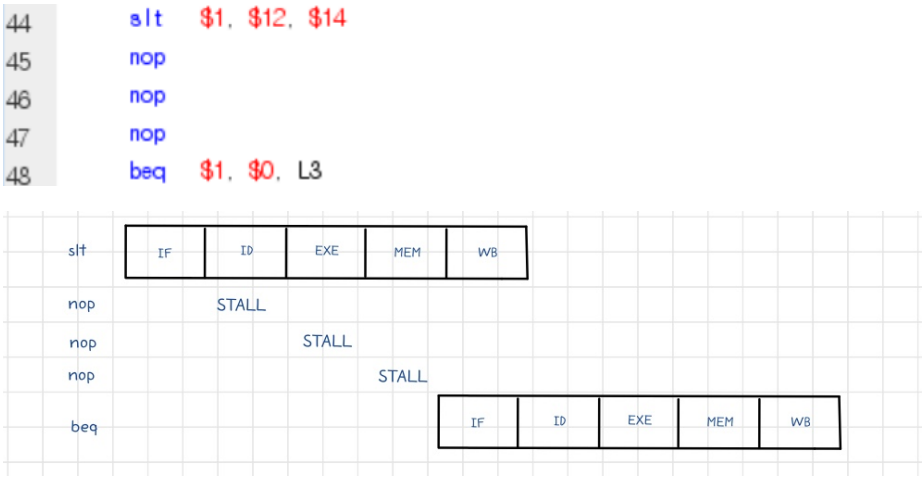


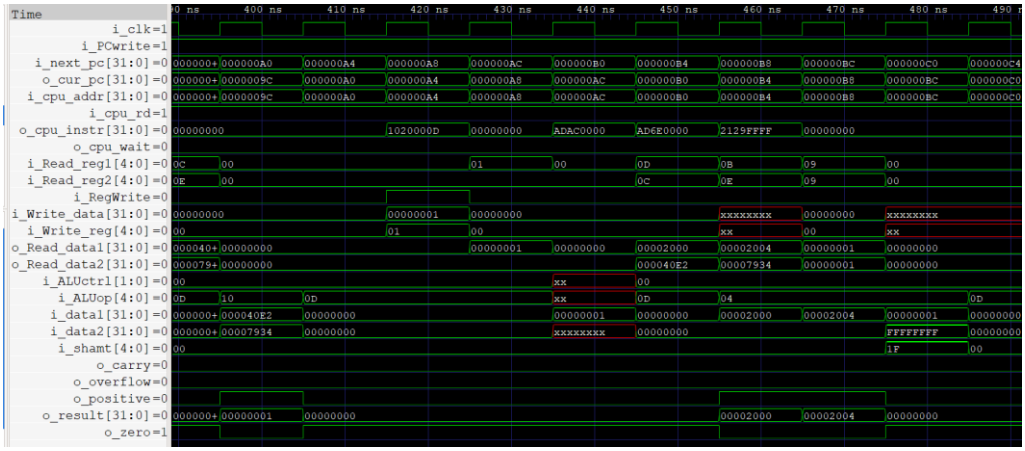
forwarding 을 00_10 으로 설정해 lw 명령어의 결과가 slt 명령어에 바로 전달하게끔 하여 hazard 를 해결하였다. 이 부분은 forwarding 만으로 hazard 를 해결할 수 없어 NOP 를 넣어주어 파이프라인을 정지시키는 동안에 slt 명령어가 수행되지 않고 lw 명령어의 결과를 기다리도록 하였다. 이전 명령어의 output 과 slt 명령어의 input 이 동일하므로 forwarding 이 잘 된 것을 알 수 있다.

⑨

slt 명령어의 결과를 beq 명령어가 사용하기 때문에 data hazard 가 발생한다. slt 명령어는 $12 와 $14 의 값을 비교하여 결과를 $1 에 저장한다. 이후 beq 명령어는 $1 의 값을 사용하여 분기 조건을 확인하는데 이때 이전 명령어가 실행이 완료 되기 전에 beq 명령어에서 $1 값을 사용하기 때문이다.

a.

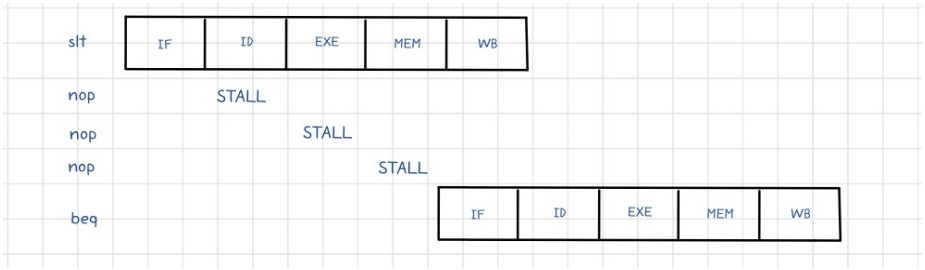


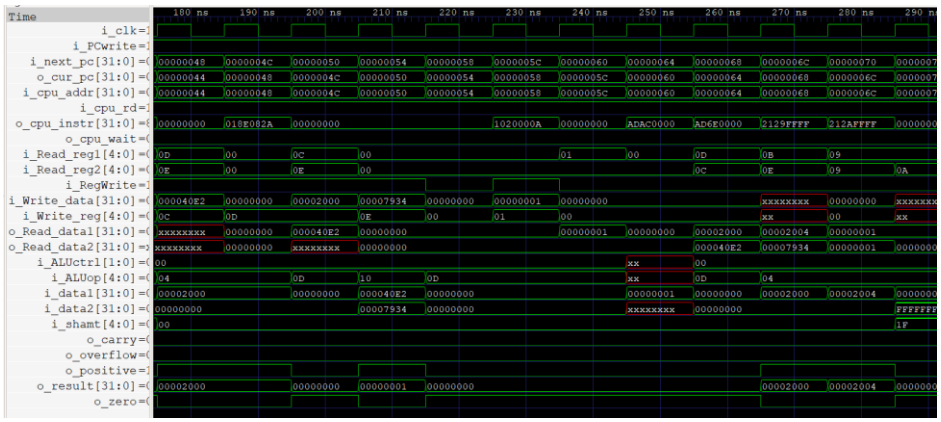


slt 명령어의 결과가 WB 된 후 beq 명령어에서 실행 결과를 사용해야 올바른 값이 나오기 때문에 NOP 를 3 개 추가해 해결하였다. slt 명령어의 IF 는 0x94, beq 명령어의 IF 는 0xa4 에서 이루어진다. NOP 를 통해 hazard 를 해결한 것을 slt 명령어의 WB 되고 $1 을 사용하는 것을 통해 확인할 수 있다.

b.

```
25      slt   $1, $12, $14
26      nop
27      nop
28      nop
29      beq   $1, $0, L3
```



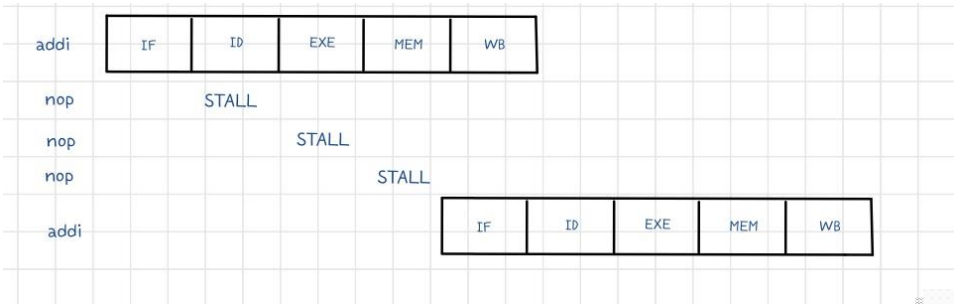


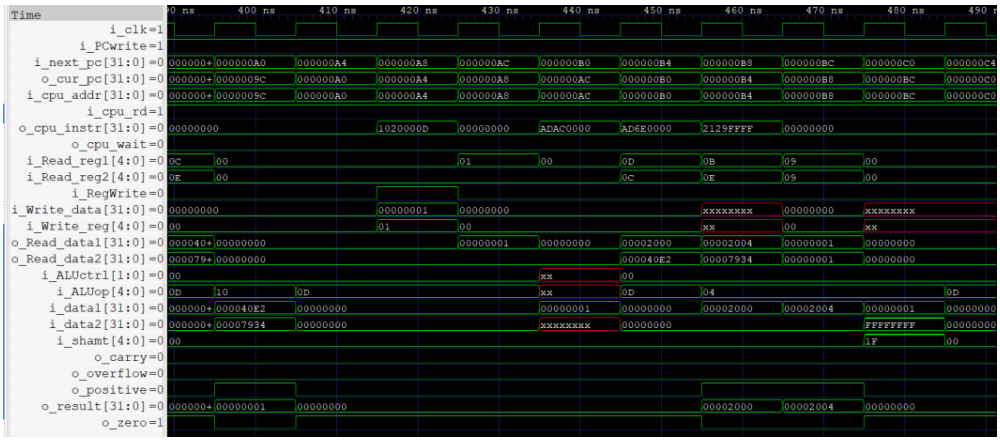
slt 명령어와 beq 명령어 사이에는 한 개의 파이프라인 stage 가 더 있어 데이터를 바로 전달할 수 있는 경로가 없어 NOP 만을 사용하여 hazard 를 해결해야 한다.

⑩

addi 명령어의 결과를 27 번째 줄 addi 명령어가 사용하기 때문에 data hazard 가 발생한다. 첫 번째 addi 명령어는 $9 에 $9 와 -1 을 더한 계산 결과를 저장한다. 두 번째 addi 명령어는 $10 에 $9 와 -1 을 더하는 연산을 저장한다. 이때 이전 명령어가 실행이 완료 되기 전에 addi 두번째 명령어에서 $9 값을 사용하기 때문이다.

a.

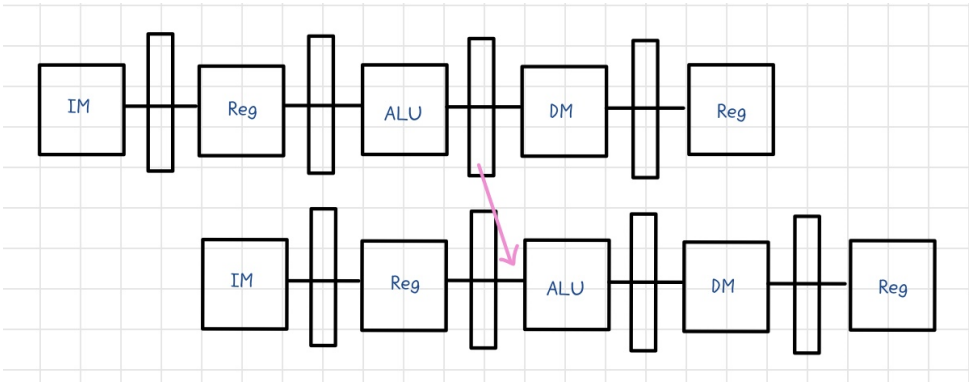




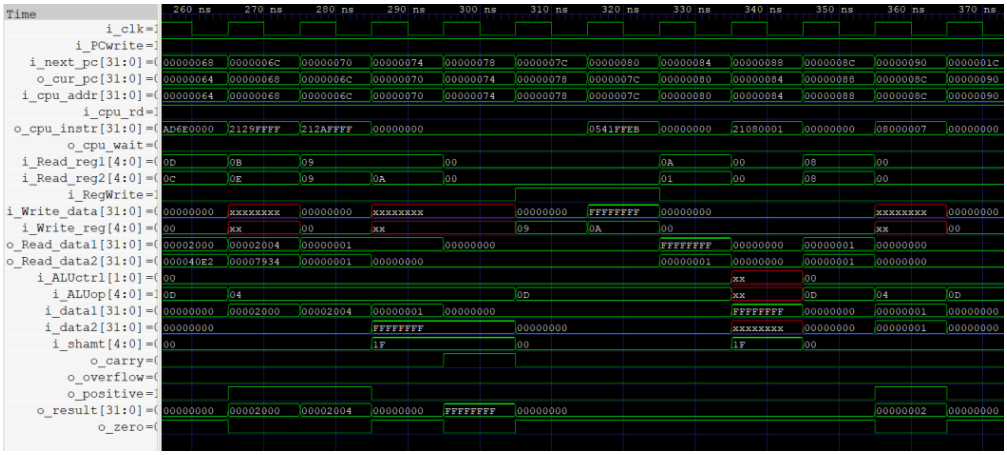
첫번째 addi 명령어의 결과가 WB 된 후 두번째 addi 명령어에서 실행 결과를 사용해야 올바른 값이 나오기 때문에 NOP 를 3 개 추가해 해결하였다. 첫번째 addi 명령어의 IF 는 0xb8, 두번째 addi 명령어의 IF 는 0xc8 에서 이루어진다. NOP 를 통해 hazard 를 해결한 것을 첫번째 addi 명령어의 WB 되고 $9 를 사용하는 것을 통해 확인할 수 있다.

b.

```
35      addi  $9, $9, -1
36      addi  $10, $9, -1
```




```
01_00  //  0x068
```



forwarding 을 01_00 으로 설정해 첫 번째 addi 명령어의 결과가 두번째 addi 명령어에 바로 전달하게끔 하여 hazard 를 해결하였다. 이전 명령어의 output 과 두번째 addi 명령어의 input 이 동일하므로 forwarding 이 잘 된 것을 알 수 있다.

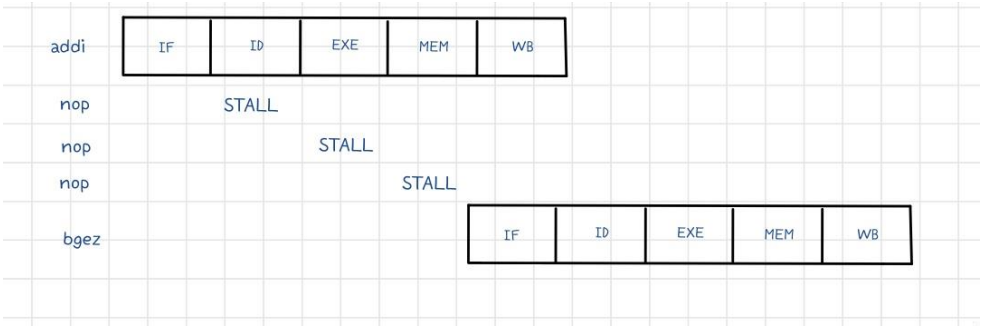
⑪

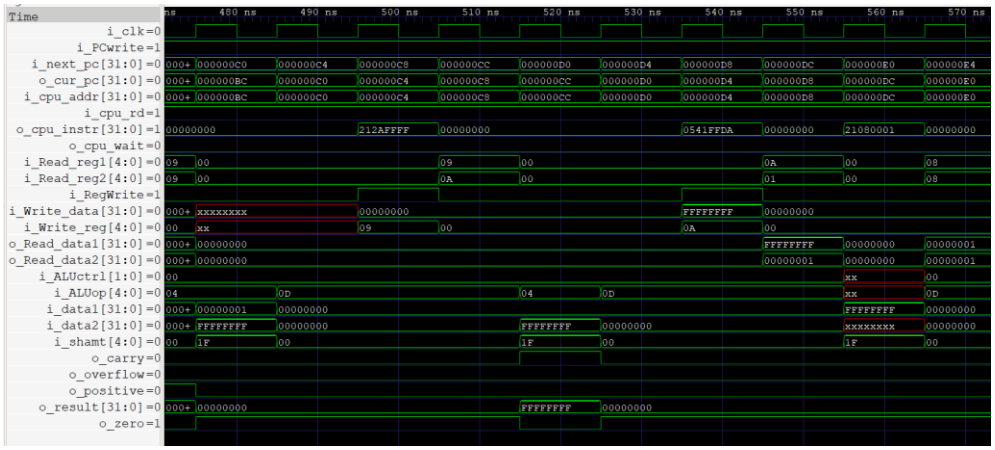
```
27      addi  $10, $9, -1
28      bgez  $10, L2
```

addi 명령어의 결과를 bgez 명령어가 사용하기 때문에 data hazard 가 발생한다. $10 에 $9 와 -1 의 덧셈 결과를 저장한다. 이어서 bgez 명령어는 $10 의 값을 사용해 조건 분기를 수행하는데 이때 $10 에 변경된 값을 사용하기 때문이다.

a.

```
58      addi  $10, $9, -1
59      nop
60      nop
61      nop
62      bgez  $10, L2
```

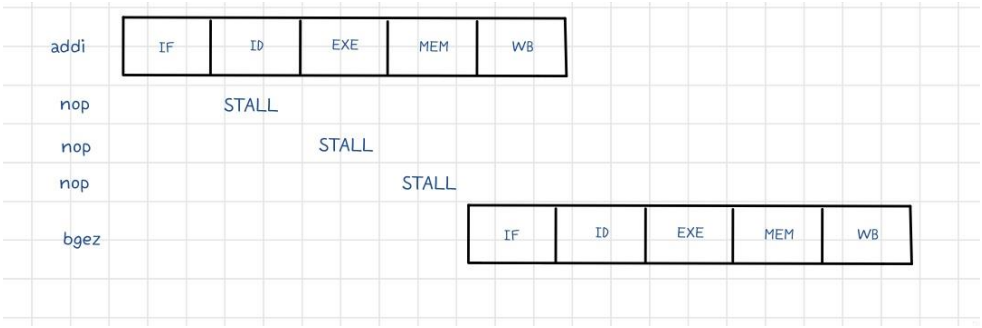


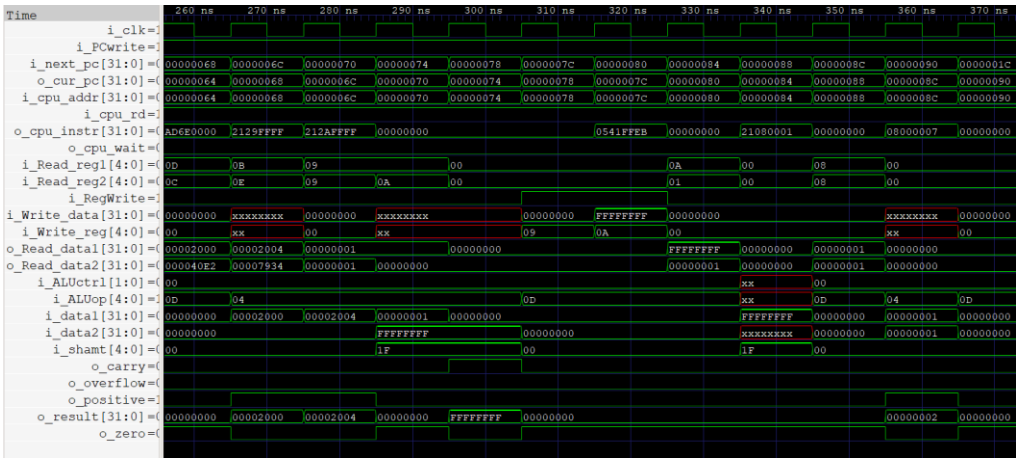


addi 명령어의 결과가 WB 된 후 bgez 명령어에서 실행 결과를 사용해야 올바른 값이 나오기 때문에 NOP 를 3 개 추가해 해결하였다. addi 명령어의 IF 는 0xc8, bgez 명령어의 IF 는 0xd8 에서 이루어진다. NOP 를 통해 hazard 를 해결한 것을 addi 명령어의 WB 되고 $10 을 사용하는 것을 통해 확인할 수 있다.

b.

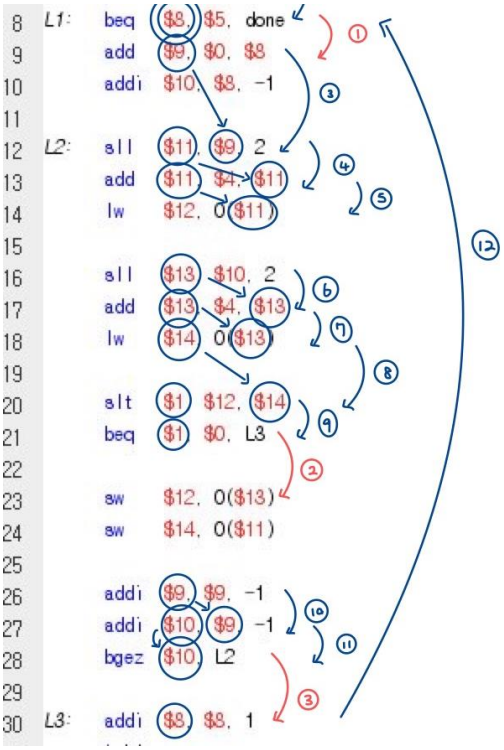
```
36      addi  $10, $9, -1
37      nop
38      nop
39      nop
40      bgez  $10, L2
```





addi 명령어와 bgez 명령어 사이에는 한 개의 파이프라인 stage 가 더 있어 데이터를 바로 전달할 수 있는 경로가 없어 NOP 만을 사용하여 hazard 를 해결해야 한다.

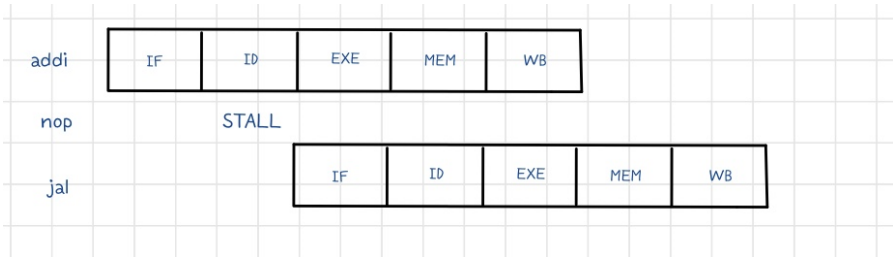
⑫



addi 명령어의 결과를 beq 명령어가 사용하기 때문에 data hazard 가 발생한다. $8 에 $8 과 1 의 덧셈 결과를 저장한다. 이어서 beq 명령어는 $8 의 값을 사용해 조건 분기를 수행하는데 이때 $8 에 변경된 값을 사용하기 때문이다. 이부분에서 data hazard 를 해결하면 그 다음줄에서 $8 을 사용하는 것도 함께 해결된다.

a.

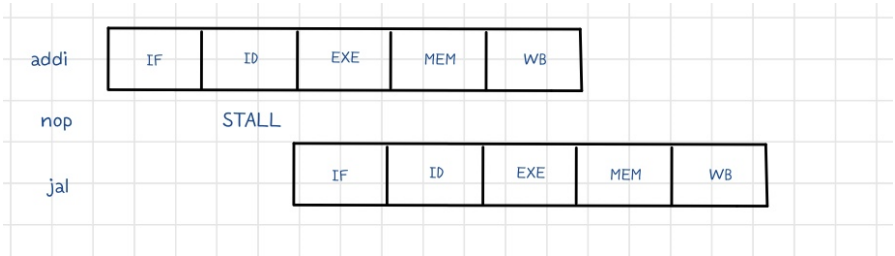
```
65  L3:   addi  $8, $8, 1
66        nop
67        j L1
```



addi 명령어가 L1 으로 jump 하였을 때 $8 의 값을 사용해야 하는 data hazard 를 해결하기 위해 NOP 를 추가해주어 해결하였다.

b.

```
43  L3:   addi  $8, $8, 1
44        nop
45        j L1
```



addi 명령어와 j 명령어 사이에는 한 개의 파이프라인 stage 가 더 있어 데이터를 바로 전달할 수 있는 경로가 없어 NOP 만을 사용하여 hazard 를 해결해야 한다.

이어서 control hazard 에 대해서도 설명해보겠다.

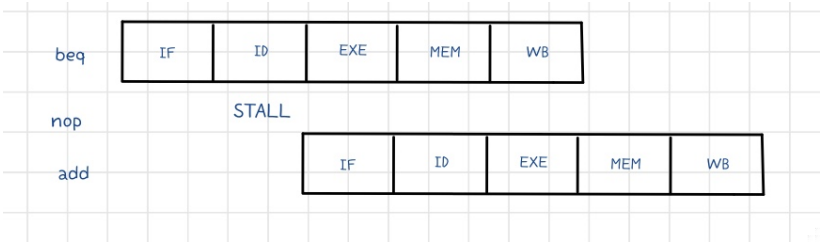
①

beq 명령어는 분기 조건을 평가한 후 분기를 수행하는데 분기 결과에 따라 다음 실행될 명령어가 결정되기 때문에 control hazard 가 발생한다. beq 명령어는 분기를

수행하기 전에는 다음에 실행될 명령어가 불확실해 add $9, $0, $8 명령어는 분기되거나 실행되지 않을 수도 있기 때문이다.

a.

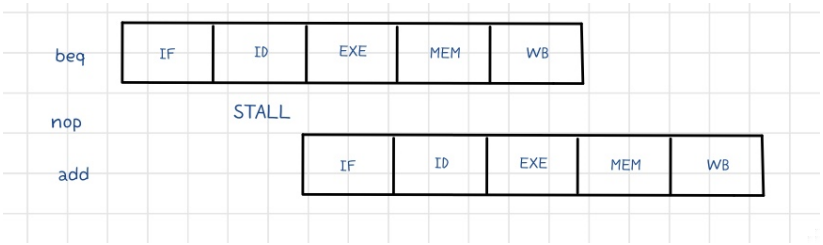
```
14  L1:   beq   $8, $5, done
15        nop
16        add   $9, $0, $8
```



따라서 hazard 를 해결하기 위해 beq 명령어의 분기 결과가 확실해질 때까지 add $9, $0, $8 명령어를 지연시켜야 하므로, NOP 를 추가해주어 hazard 를 해결해주었다.

b.

```
11  L1:   beq   $8, $5, done
12        nop
13        add   $9, $0, $8
```



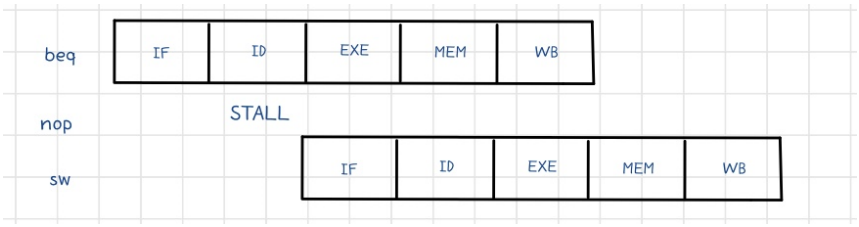
beq 명령어와 add 명령어 사이에는 한 개의 파이프라인 stage 가 더 있어 데이터를 바로 전달할 수 있는 경로가 없어 NOP 만을 사용하여 hazard 를 해결해야 한다.

②

beq 명령어는 분기 조건을 평가한 후 분기를 수행하는데 분기 결과에 따라 다음 실행될 명령어가 결정되기 때문에 control hazard 가 발생한다. beq 명령어는 분기를 수행하기 전에는 다음에 실행될 명령어가 불확실해 sw $12, 0($13) 명령어는 분기되거나 실행되지 않을 수도 있기 때문이다.

a.

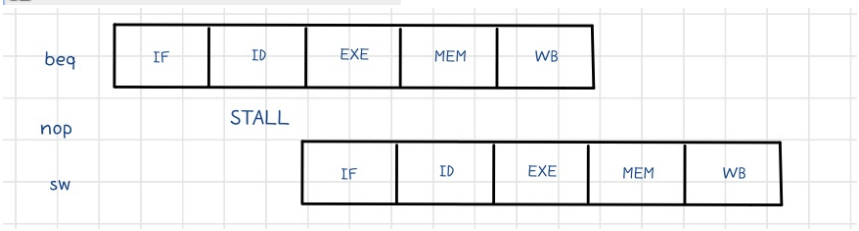
```
48        beq   $1, $0, L3
49        nop
50
51        sw    $12, 0($13)
```



hazard 를 해결하기 위해 beq 명령어의 분기 결과가 확실해질 때까지 sw $12, 0($13) 명령어를 지연시켜야 하므로, NOP 를 추가해주어 hazard 를 해결해주었다.

b.

```
29        beq   $1, $0, L3
30        nop
31
32        sw    $12, 0($13)
```



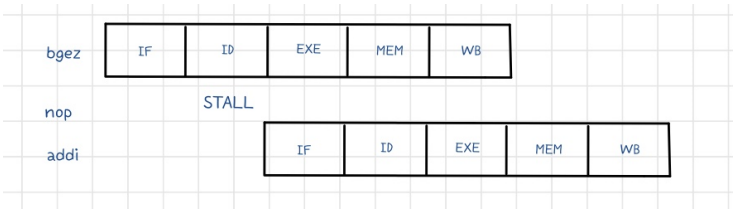
beq 명령어와 sw 명령어 사이에는 한 개의 파이프라인 stage 가 더 있어 데이터를 바로 전달할 수 있는 경로가 없어 NOP 만을 사용하여 hazard 를 해결해야 한다.

③

bgez 명령어는 분기 조건을 평가한 후 분기를 수행하는데 분기 조건을 만족할 경우에만 분기가 수행되므로, 분기 결과에 따라 실행될 명령어가 결정되기 때문에 control hazard 가 발생한다. bqez 명령어는 분기를 수행하기 전에는 다음에 실행될 명령어가 불확실해 addi $8, $8, 1 명령어는 분기되거나 실행되지 않을 수도 있기 때문이다.

a.

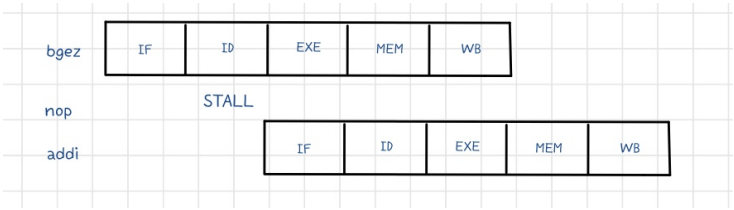
```
62        bgez  $10, L2
63        nop
64
65  L3:   addi  $8, $8, 1
```



hazard 를 해결하기 위해 bqez 명령어의 분기 결과가 확실해질 때까지 addi $8, $8, 1 명령어를 지연시켜야 하므로, NOP 를 추가해주어 hazard 를 해결해주었다.

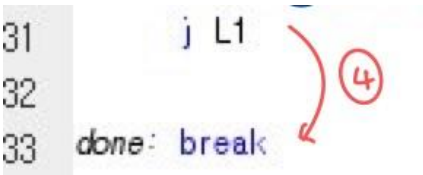
b.

```
40        bgez  $10, L2
41        nop
42
43  L3:   addi  $8, $8, 1
```



bqez 명령어와 addi 명령어 사이에는 한 개의 파이프라인 stage 가 더 있어 데이터를 바로 전달할 수 있는 경로가 없어 NOP 만을 사용하여 hazard 를 해결해야 한다.

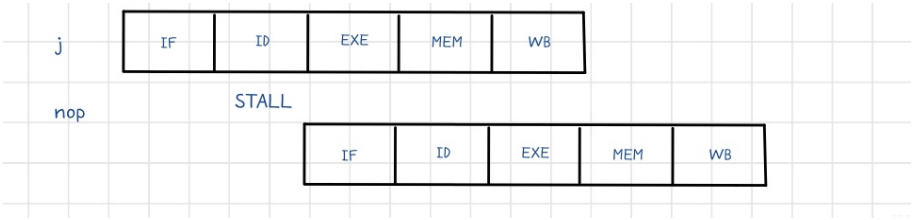
④



j 명령어는 주어진 주소로 jump 해 해당 주소에서 실행을 계속한다. j 명령어는 분기 조건 평가 없이 무조건 분기를 수행하기 때문에 control hazard 가 발생하지 않는다. 하지만 위 코드는 break 명령어를 실행하면 break 이후에 L1 으로 jump 할 수 없기 때문에 이 부분이 불확실해 control hazard 가 발생할 수 있다.

a.

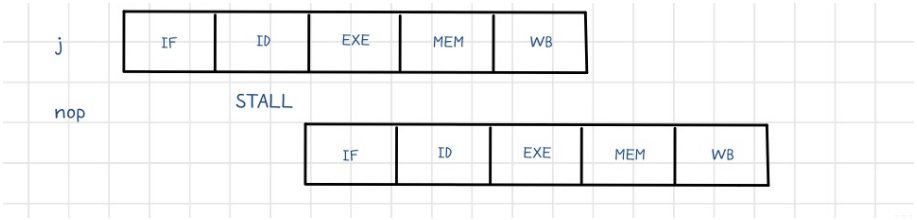
```
67          j L1
68          nop
69
70  done:   break
```



break 문에 의해 발생하는 control hazard 를 막기 위해 NOP 를 추가해주어 hazard 를 해결해주었다.

b.

```
45          j L1
46          nop
47
48  done:   break
```



break 문에 의해 발생하는 control hazard 를 막기 위해 NOP 를 추가해주어 hazard 를 해결해주었다.

명령어 수행에 걸린 a 와 b 의 총 cycle 수이다. a 보다 b 의 경우가 cycle 수와 performance 면에서 더 나은 것을 볼 수 있다. 한 연산의 결과가 다른 연산의 입력으로 사용되는 경우(데이터 종속성), forwarding 을 활용해 메모리 접근을 최소화하고 성능을 향상시킬 수 있다. 또한 여러 단계로 구성된 명령어 실행 과정을 겹치게 함으로써 성능을 향상시키는 파이프라인에서 파이프라인이 멈추게 될 수 있어 이를 forwarding 으로 해결할 수 있다. 이처럼 forwarding 을 사용함으로써 데이터의 종속성을 제거하고 파이프라인의 효율을 향상시켜 cycle 수가 줄어들며 performance 면에서도 향상되었다고 생각한다.

```
-------------------------------------------
| H020-3-1647-01: Computer Architecture     |
|                 CE.KW.AC.KR               |
-------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-------------------------------------------
 Break signal: 1,   # of Cycles:        2608
-------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 26195000 (1ps)

C:\Users\a0109\OneDrive\바탕 화면\prj3_PCPU_2023>FC /L mem_dump_IS.txt mem_dump.txt
파일을 비교합니다: mem_dump_IS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.


C:\Users\a0109\OneDrive\바탕 화면\prj3_PCPU_2023>FC /L reg_dump_IS.txt reg_dump.txt
파일을 비교합니다: reg_dump_IS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.
```

```
-------------------------------------------
| H020-3-1647-01: Computer Architecture     |
|                 CE.KW.AC.KR               |
-------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-------------------------------------------
 Break signal: 1,   # of Cycles:        1486
-------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 14975000 (1ps)

C:\Users\a0109\OneDrive\바탕 화면\prj3_PCPU_2023>FC /L mem_dump_IS.txt mem_dump.txt
파일을 비교합니다: mem_dump_IS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.


C:\Users\a0109\OneDrive\바탕 화면\prj3_PCPU_2023>FC /L reg_dump_IS.txt reg_dump.txt
파일을 비교합니다: reg_dump_IS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.
```

3. 고찰

이번 프로젝트를 진행하면서 컴퓨터구조 수업 때 다뤘던 hazard 에 대해 더 알아보고 이해할 수 있었다. 이번 프로젝트에는 정확한 답은 없으나 전체 cycle 수를 최소화하는 것이 목적이었다. 처음 a 의 경우를 test 하면서 마지막 L3 부분에서 두 번의 nop 를 주어 전체 사이클 수가 2623 cycle 이 나왔다. 하지만 파이프라인을 그려보고 결과를 파악하며 nop 를 하나만 주어도 hazard 도 해결하고 cycle 수도 줄일 수 있음을 알게 되었다. 따라서 총 cycle 수가 2608 cycle 이 될 수 있었다. 또한 j 명령어 부분에서 jump 를 L1 으로 한 후 beq 명령어에서 done 으로 갈지 말지 branch 의 여부를 결정하는 순서로 진행되는데, 이때 break 를 하는 이유를 정확히 이해하지 못했었다. 하지만 제안서 그림을 참고했을 때 jump 도 stage 에 맞게 진행하기 때문에 jump 의 instruction decode 단계까지 break 가 실행되지 않도록 해줘야 하므로 control hazard 가 발생하지 않게 하기 위해 NOP 를 넣어주어야 한다는 것을 알게 되었다. b 의 경우를 test 하면서 branch 의 경우 forwarding 을 할 수 없다고 이해했는데, 이 역시 한 개의 파이프라인 stage 가 더 있어 데이터를 바로 전달할 수 있는 경로가 없기 때문에 hazard 를 해결하기 위해 NOP 를 사용해야 한다고 생각했다.

4. Reference

컴퓨터구조실험 강의자료

컴퓨터구조 강의자료