컴퓨터구조 Report

Project #3 – Pipeline Architecture

담당교수: 이성원 교수님

실습분반: 수 7교시

학번: 2019202050

이름: 이강현

[Introduction]

이번 프로젝트는 pipeline architecture에서 생길 수 있는 여러가지 hazard에 대해 알고 이를 찾아내어 hazard가 발생하지 않게끔 한다. hazard의 종류는 structural hazard, data hazard, control hazard 세가지가 있다. 먼저 structural hazard는 같은 cycle에서 두개 이상의 명령어가 서로 같은 resource를 사용하려고 하는 것이다. data hazard는 여러가지가 있지만 해당 architecture에서는 RAW(read after write)에 관한 hazard를 다루고 RAW는 이전 명령어에서 아직 write를 마치지 않았지만 현재 명령어에서 해당 register를 read하려고 할 때 나타난다. control hazard는 branch나 jump와 같은 명령어 이후에 오는 명령어들이 분기의 유무를 알지 못한 시점에 실행되어 버린다면 실행하지 않아야 할 명령어가 실행된다. 이와 같은 상황을 control hazard로 다룰 것이다. 1차적으로 명령어 간 dependency를 모두 찾아내고 이를 nop(no operation)명령어를 통해 해결할 것이다. 추가적으로 forwarding을 사용하여 nop의 사용을 최대한 줄이고 throughput을 올려 performance를 향상시킨다.

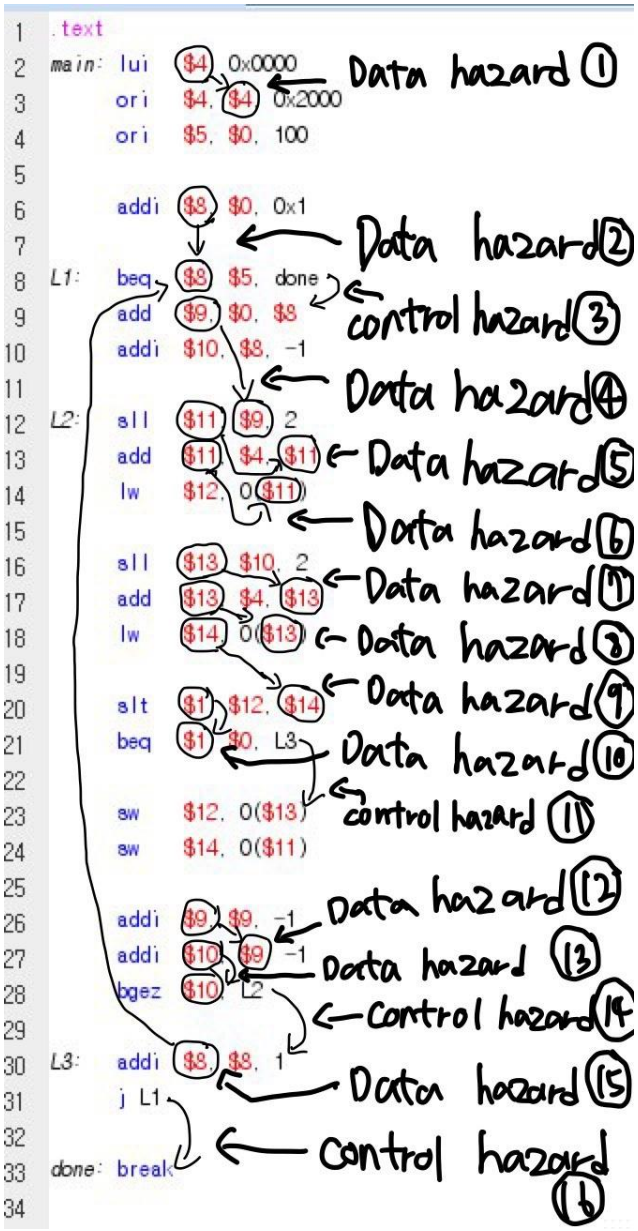
[Assignment]

hazard를 피하기 위한 방법은 먼저 stall을 사용하거나 forwarding을 사용하는 방법이 있다. stall을 사용하여 data dependency가 있는 부분에서 data가 준비될 때까지 기다리거나(S/W-coding) 전 명령어의 부분에서 하드웨어적으로 데이터를 직접 ALU로 보내주는 방법이다.(H/W-Forwarding)

data hazard에서의 S/W-coding은 컴파일러가 명령어를 재정렬하거나 nop를 넣어주어 진행할 수 있다. H/W-Forwarding은 이전 명령어의 EX단계를 마무리하고 MEM단계나 혹은 WB단계에서 이후 연산된 값과 dependency를 나타내는 부분이 있다면 이 값을 EX단계의 ALU로 보내준다. 따라서 레지스터에 쓰여짐을 기다리고 이후 ID단계에서 register를 fetch하여 얻는 경우와는 성능면에서 더 효율적으로 작동한다.

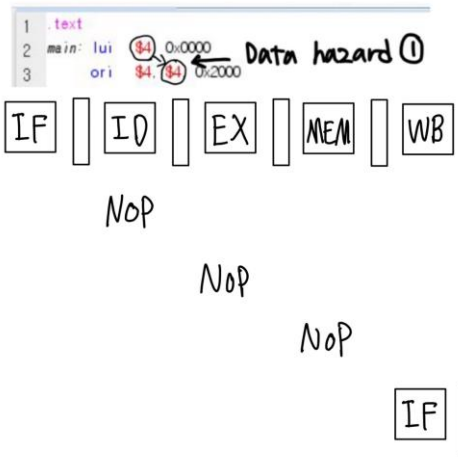
control hazard에서의 S/W-coding은 branch prediction이 틀려 이후 명령어가 실행되는것을 미리 방지하기 위해 nop를 추가하거나 branch명령어의 delay slot을 이용하여 분기와 별개로 실행되어야 하는 코드를 넣어두거나 하는 방식도 가능하다. H/W forwarding은 branch의 결과를 미리 예측할 수 있게끔 하여 지연을 최소화할 수 있다.

아래는 주어진 코드에서 hazard가 발생하는 부분을 화살표로 표기하였고 해당 부분에서 어떤 종류의 hazard가 발생하는지 표기한 부분이다.



먼저 nop만 사용하여 hazard를 해결하는 것을 보인다.

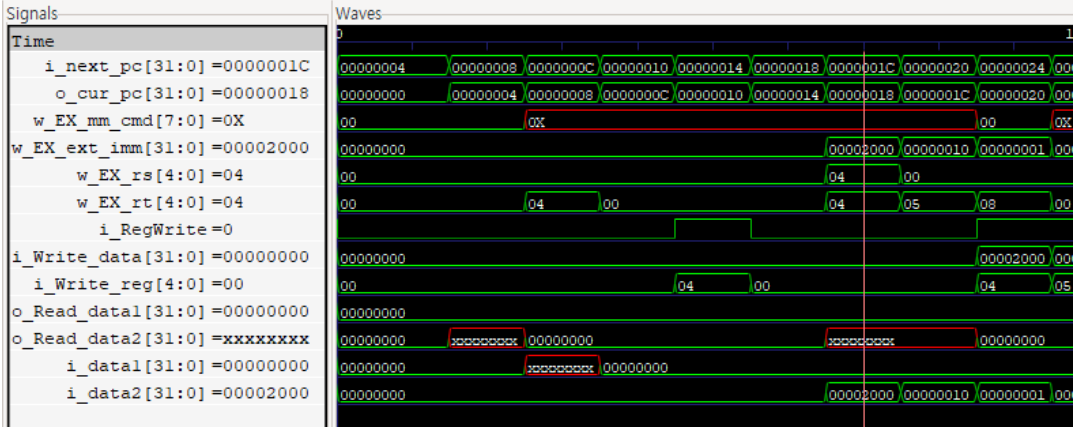
1.




```
main: lui   $4, 0x0000
      nop
      nop
      nop
      ori   $4, $4, 0x2000
```

$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000

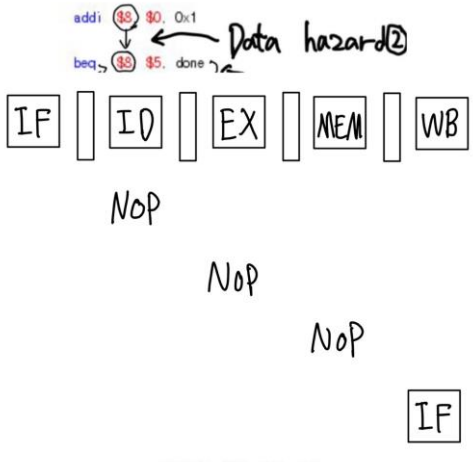
Address	Code	Basic			
0x00000000	0x3c040000	lui $4,0x00000000	2:	main: lui	$4, 0x0000
0x00000004	0x00000000	nop	3:	nop	
0x00000008	0x00000000	nop	4:	nop	
0x0000000c	0x00000000	nop	5:	nop	
0x00000010	0x34842000	ori $4,$4,0x00002000	6:	ori	$4, $4, 0x2000



$4에 lui명령의 결과가 저장되고 바로 다음 명령어에서 $4를 이용해 ori명령을 진행하는 부분이다. 전 명령어에서 레지스터에 결과값을 온전히 저장한 후 다음 명령어에서 register fetch가 일어나야 올바른 $4값을 참조할 수 있으므로 nop를 3개 사용하여 위와 같이 코드를 구현하였다.

simulation결과를 보면 0x00은 lui의 IF,0x04,0x08,0x0c는 nop,0x10은 lui명령의 write back단계이므로 $4에 0x0000값을 쓰고있는 것을 알 수 있고 이 cycle에서 ori의 IF가 일어난다. 그렇기에 data hazard를 해결할 수 있고 0x18 ori의 EX단계에서 ALU에 $4와 상수 0x2000이 올바르게 대기하고 있음을 알 수 있다.

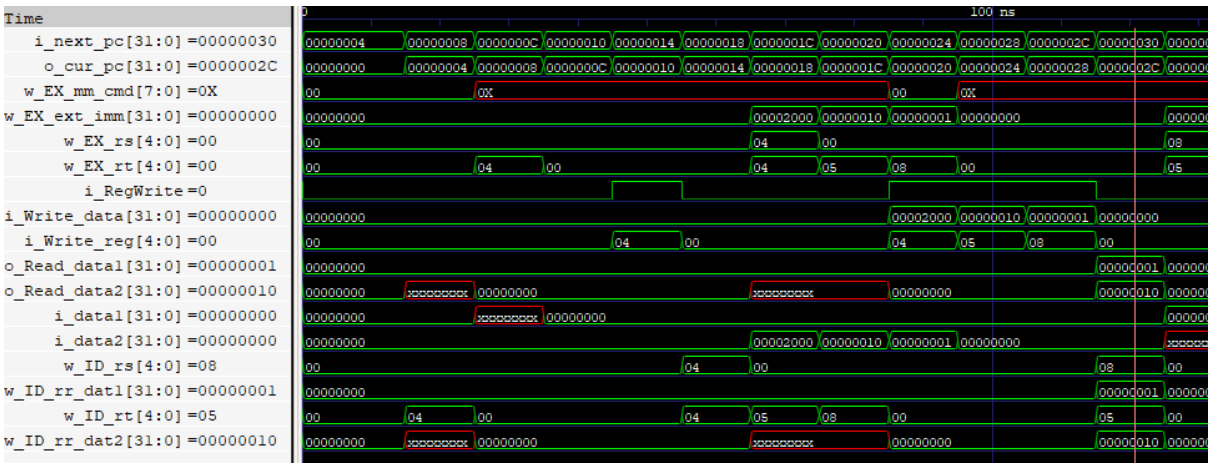
2.




```
addi  $8, $0, 0x1
nop
nop
nop

beq   $8, $5, done
```

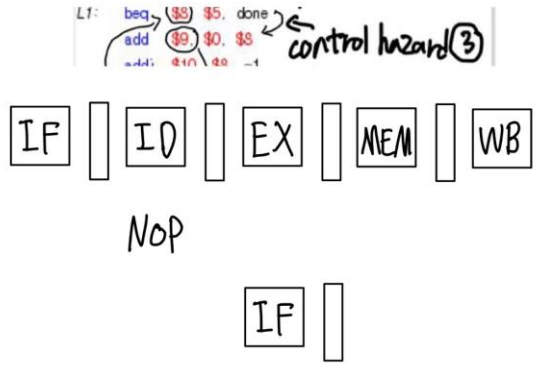
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000000

0x00000018	0x20080001	addi $8,$0,0x00000001	9:	addi $8, $0, 0x1
0x0000001c	0x00000000	nop	10:	nop
0x00000020	0x00000000	nop	11:	nop
0x00000024	0x00000000	nop	12:	nop
0x00000028	0x11050030	beq $8,$5,0x00000030	14: L1:	beq $8, $5, done



2번 hazard도 1번 hazard와 마찬가지로 이전 명령어의 WB단계가 끝난 후 레지스터 값을 참조해야 하므로 nop를 3개 넣어준다. 0x28에서는 beq명령어의 IF가 진행될 것이고 그렇다면 0x2c에서 ID가 진행될 것이다. 그렇다면 branch의 조건 계산은 register 뒤 비교기에서 진행되므로 0x2c에서 올바른 값을 참조하는지 확인하면 된다. 이전 단계에서 $8에는 0x1이 들어갔고 $5에는 0x10이 있으므로 ID단계에서 올바른 데이터가 대기하고 있음을 알 수 있다.

3.

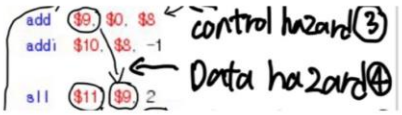


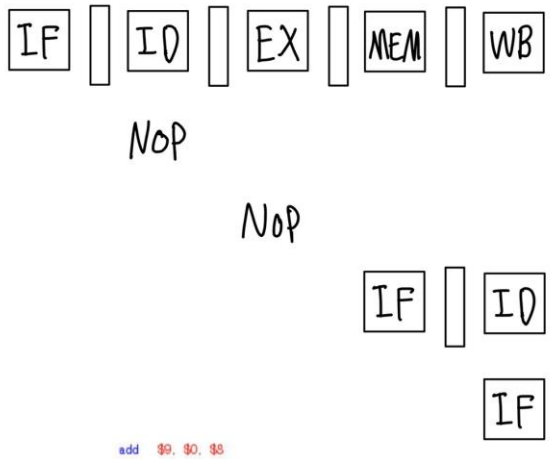

```
L1:   beq   $8, $5, done
      nop
      add   $9, $0, $8
```

0x00000028	0x11050030	beq $8,$5,0x00000030	14: L1:	beq	$8, $5, done
0x0000002c	0x00000000	nop	15:	nop	

3번에서의 nop는 필요할 수도 있고 아닐 수도 있다. 만약에 beq명령이 done으로 가게끔 흐름을 바꾼다면 바로 밑 add가 실행되어선 안되지만 실행될 것이다. 분기 조건을 연산하는 과정이 ID에서 진행되고 그 다음 cycle에서 pc가 update되기 때문에 새로운 실행흐름은 beq명령의 EX단계 cycle에서 진행될 것이다. 의도치 않은 명령어의 실행을 막기 위해서 넣은 nop이다.

4.

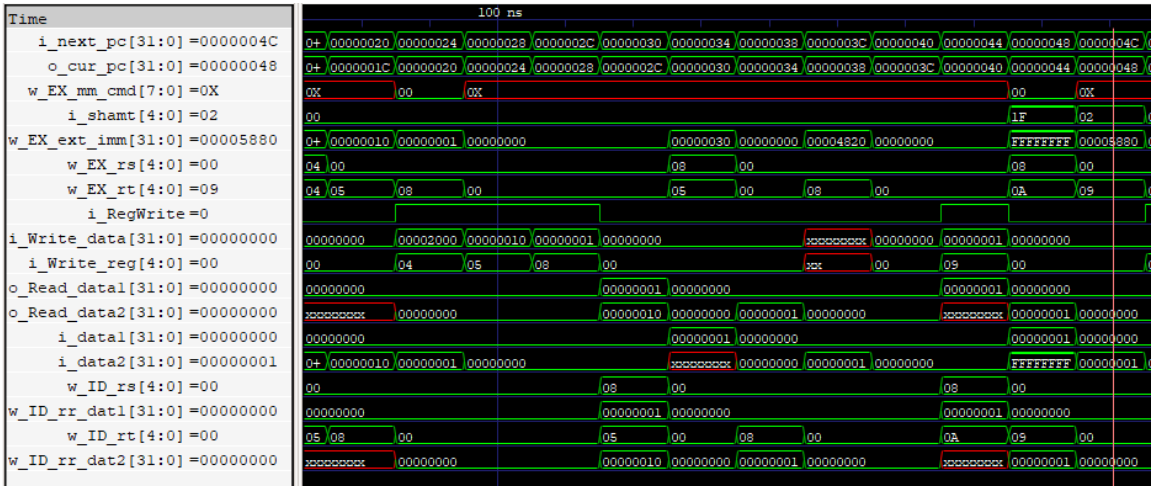





```
add  $9, $0, $8
nop
nop
addi $10, $8, -1

sll  $11, $9, 2
```

$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00000000

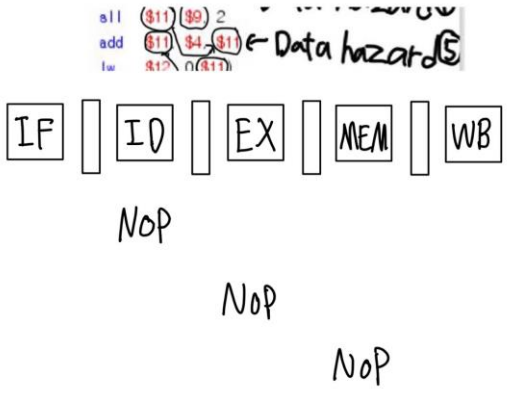
0x00000030	0x00084820	add $9,$0,$8	16:	add	$9, $0, $8
0x00000034	0x00000000	nop	17:	nop	
0x00000038	0x00000000	nop	18:	nop	
0x0000003c	0x210affff	addi $10,$8,0xffffffff	19:	addi	$10, $8, -1
0x00000040	0x00095880	sll $11,$9,0x00000002	21: L2:	sll	$11, $9, 2



4번 hazard는 add명령의 $9와 sll명령의 $9간의 dependency이다. 따라서 $9에 값을 온전히 쓴 후 sll명령어가 ID단계를 진행해야 한다. 0x30에서 add의 IF가 이루어지고 0x34,0x38에는 nop,0x3C에서는 addi명령의 IF가 이루어질 것이다. 0x40에서 sll의 IF와 add의 WB이 이루어지고 0x48에서 sll의 EX이 이루어진다. $9의 값은 add이후 0x1이 저장되었고 이는 sll의 ID단계에서 잘 읽히고 ALU의 input으로 들어가는 것을 확인할

수 있다.

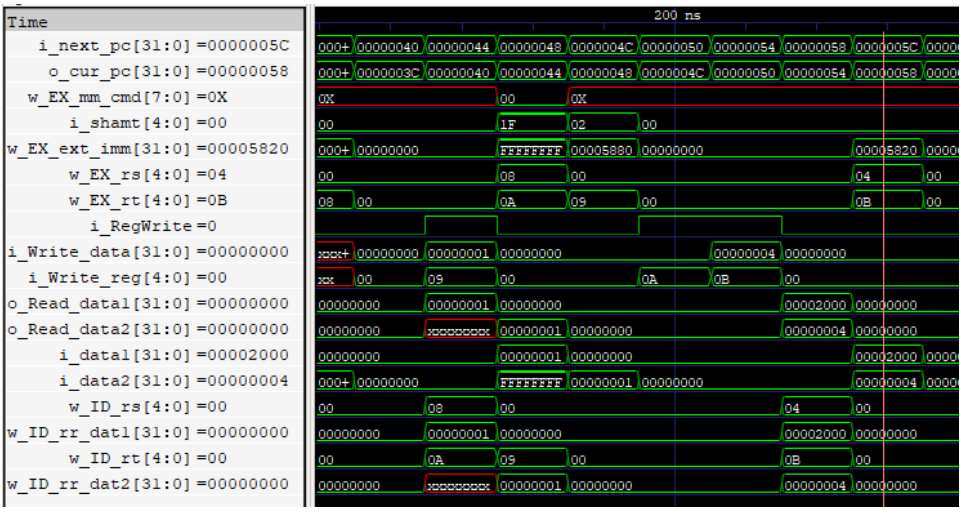
5.



4	0x00002000
5	0x00000010
6	0x00000000
7	0x00000000
8	0x00000001
9	0x00000001
10	0x00000000
11	0x00000004

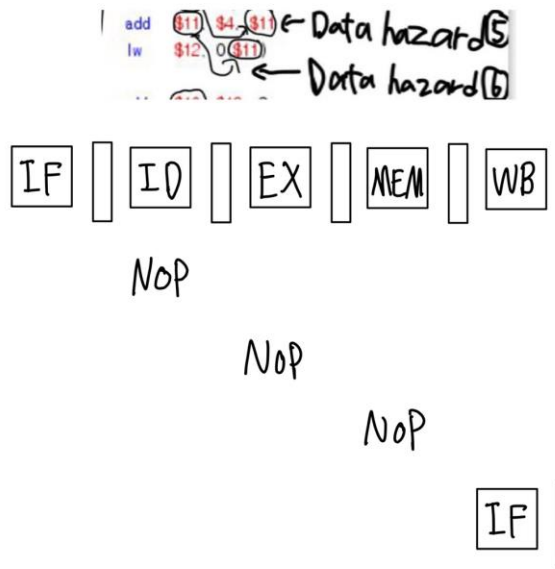
```
sll  $11, $9, 2
nop
nop
nop
add  $11, $4, $11
```

0x00000040	0x00095880	sll $11,$9,0x00000002	21: L2:	sll	$11, $9, 2
0x00000044	0x00000000	nop	22:	nop	
0x00000048	0x00000000	nop	23:	nop	
0x0000004c	0x00000000	nop	24:	nop	
0x00000050	0x008b5820	add $11,$4,$11	25:	add	$11, $4, $11



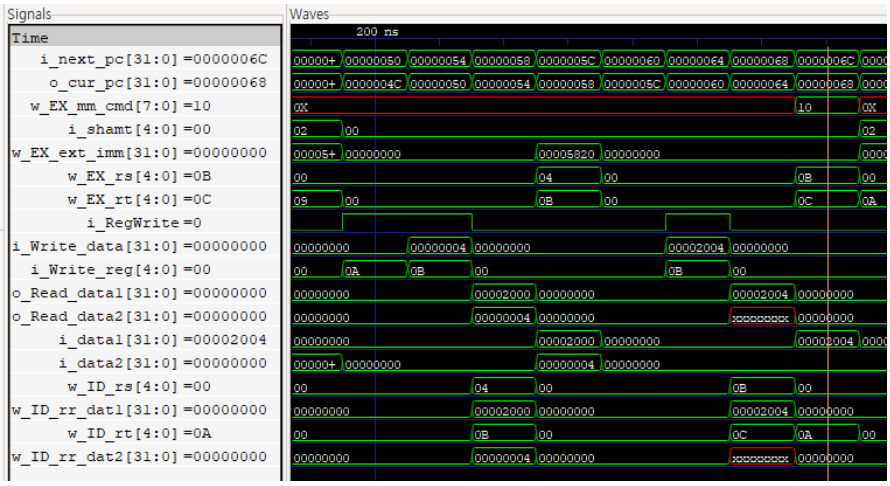
5번 hazard는 sll $11과 add $11간의 dependency이다. 0x40에서 sll명령의 IF, 0x44,0x48,0x4C에 nop, 0x50에 add명령의 IF와 sll의 WB이 진행된다. 따라서 0x54에서 $11에 sll의 연산 결과가 잘 읽히고 0x58에서 add의 ALU로 잘 들어감을 알 수 있다.

6.



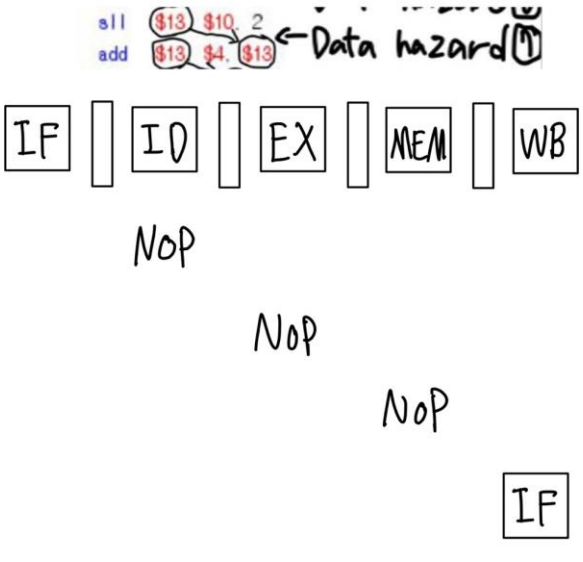
$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00002004
$t4	12	0x00000000

0x00000050	0x008b5820	add $11,$4,$11	25:	add	$11, $4, $11
0x00000054	0x00000000	nop	26:	nop	
0x00000058	0x00000000	nop	27:	nop	
0x0000005c	0x00000000	nop	28:	nop	
0x00000060	0x8d6c0000	lw $12,0x00000000($11)	29:	lw	$12, 0($11)



6번 hazard에서는 add명령의 $11과 lw의 $11과의 dependency가 있다. 0x50에서는 add의 IF, 0x54,0x58,0x5C는 nop, 0x60에서는 lw의 IF, add의 WB가 진행된다. 따라서 add명령의 WB가 마무리된 후 lw에서 $11값을 사용함을 알 수 있다.

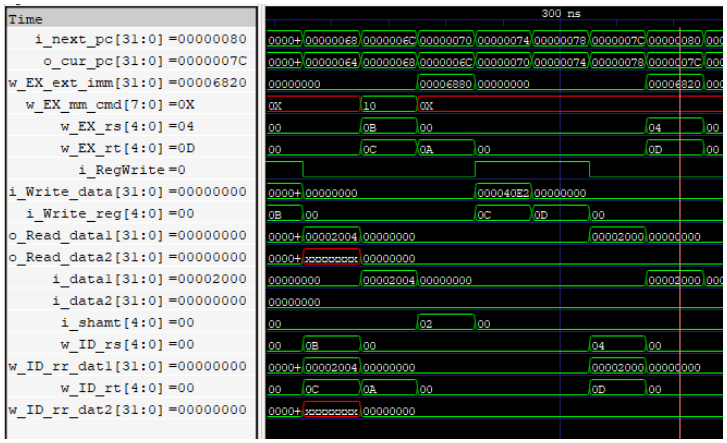
7.




```
sll   $13, $10, 2
nop
nop
nop
add   $13, $4, $13
```

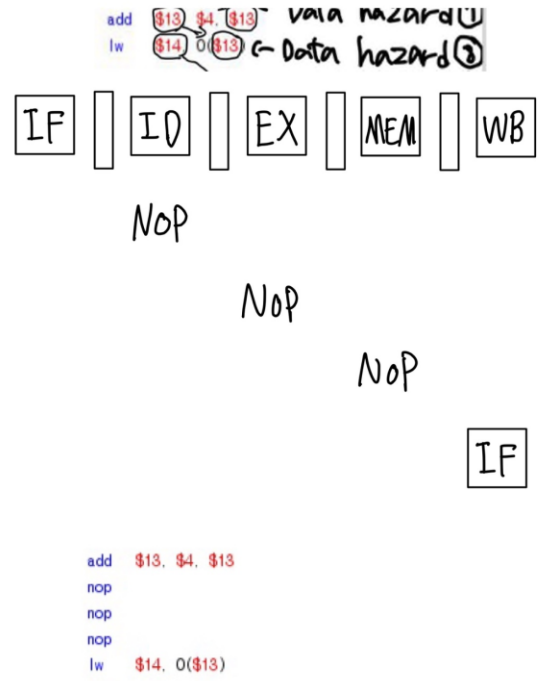
$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00002004
$t4	12	0x000040e2
$t5	13	0x00000000

0x00000064	0x000a6880	sll $13,$10,0x00000002	31:	sll	$13, $10, 2
0x00000068	0x00000000	nop	32:	nop	
0x0000006c	0x00000000	nop	33:	nop	
0x00000070	0x00000000	nop	34:	nop	
0x00000074	0x008d6820	add $13,$4,$13	35:	add	$13, $4, $13



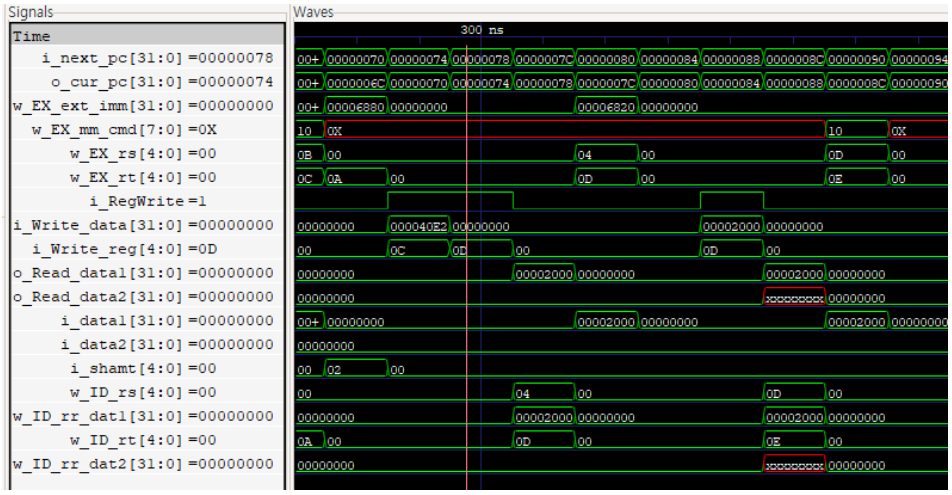
6번 hazard와 동일하게 nop를 3번 사용한 경우이며, sll의 결과값이 0x74에서 WB을 지나고 0x78에서 저장되므로 이때 add의 ID가 진행되어 결과값을 올바르게 참조하는 것을 알 수 있다.

8.



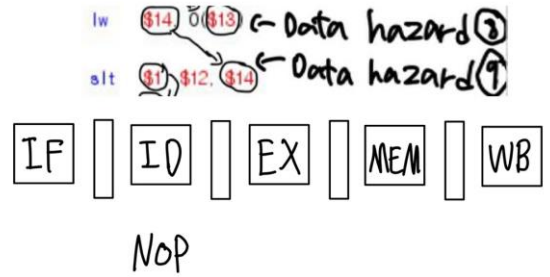
$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00002004
$t4	12	0x000040e2
$t5	13	0x00002000
$t6	14	0x00000000

0x00000074	0x008d6820	add $13,$4,$13	35:	add	$13, $4, $13
0x00000078	0x00000000	nop	36:	nop	
0x0000007c	0x00000000	nop	37:	nop	
0x00000080	0x00000000	nop	38:	nop	
0x00000084	0x8dae0000	lw $14,0x00000000($13)	39:	lw	$14, 0($13)



마찬가지로 7번 hazard와 같이 nop를 3개 사용하여 add의 $13과 lw의 $13 사이에서의 dependency를 해결한다. add의 결과를 0x88 즉 lw의 ID단계에서 올바르게 사용하는 것을 알 수 있다.

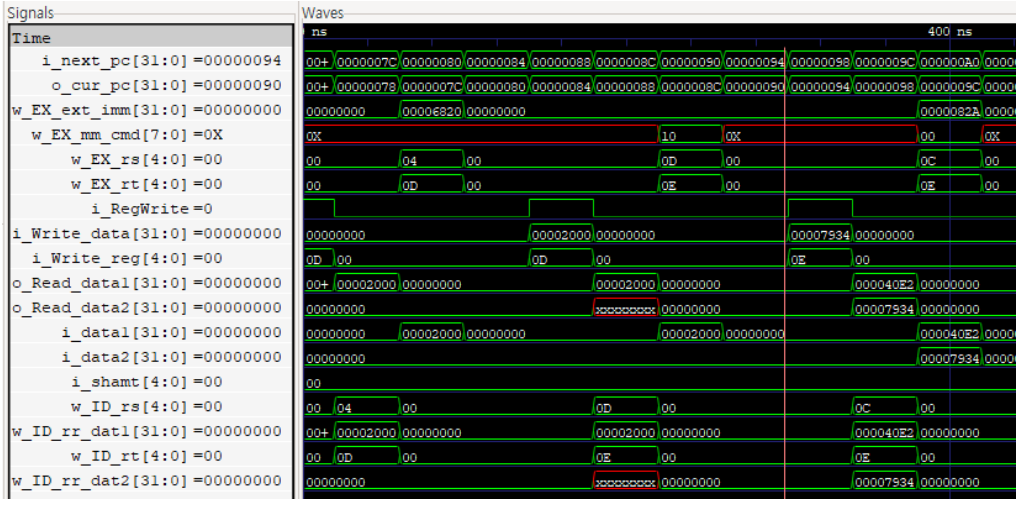
9.




```
lw    $14, 0($13)
nop
nop
nop

slt   $1, $12, $14
```

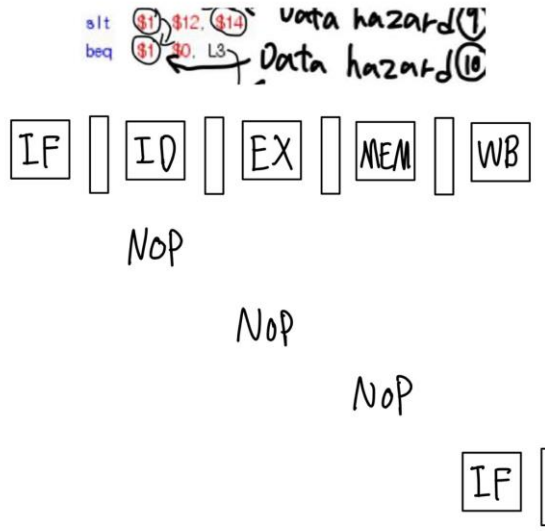
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00002004
$t4	12	0x000040e2
$t5	13	0x00002000
$t6	14	0x00007934

0x00000084	0x8dae0000	lw $14,0x00000000($13)	39:	lw	$14, 0($13)
0x00000088	0x00000000	nop	40:	nop	
0x0000008c	0x00000000	nop	41:	nop	
0x00000090	0x00000000	nop	42:	nop	
0x00000094	0x018e082a	slt $1,$12,$14	44:	slt	$1, $12, $14



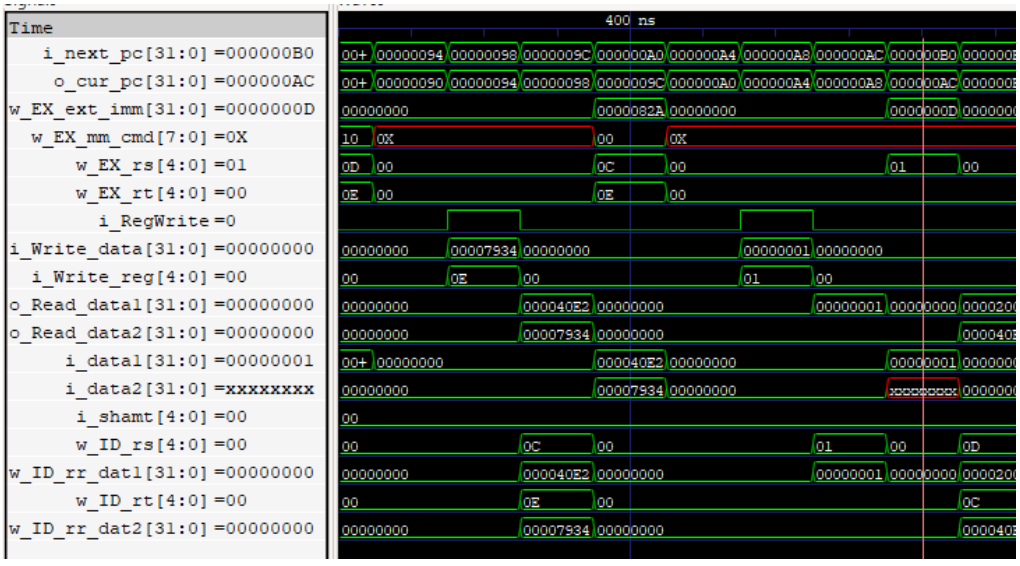
8번 hazard와 같이 nop를 3개 주었다. lw의 $14에 값이 저장될 때까지 nop 후 0x94에서 WB을 진행해 0x98에서 쓰여지면 slt의 ID단계에서 $14값을 올바르게 참조할 수 있다.

10.



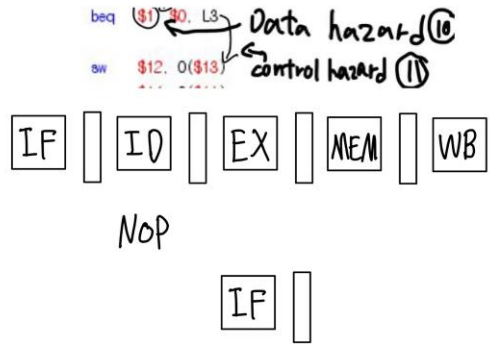
$at	1	0x00000001
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00002000

0x00000094	0x018e082a	slt $1,$12,$14	44:	slt	$1, $12, $14
0x00000098	0x00000000	nop	45:	nop	
0x0000009c	0x00000000	nop	46:	nop	
0x000000a0	0x00000000	nop	47:	nop	
0x000000a4	0x1020000d	beq $1,$0,0x0000000d	48:	beq	$1, $0, L3



9번 hazard와 마찬가지로 nop를 3번 넣어 0xa4 slt의 WB단계에서 $1에 값을 쓰고 이를 0xa8에서 올바른 값을 읽는다.

11.

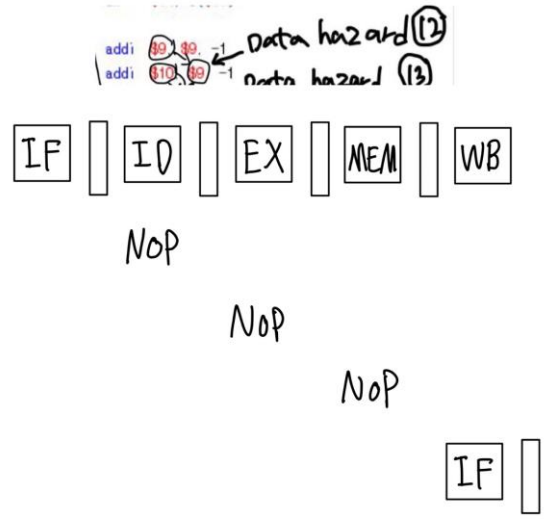



```
beq   $1, $0, L3
nop

sw    $12, 0($13)
```

0x000000a4	0x1020000d	beq $1,$0,0x0000000d	48:	beq	$1, $0, L3
0x000000a8	0x00000000	nop	49:	nop	
0x000000ac	0xadac0000	sw $12,0x00000000($13)	51:	sw	$12, 0($13)

11번 hazard 또한 3번 hazard와 같이 control hazard를 막기위한 것으로 분기를 하지 않는다면 필요없지만 분기를 할 경우 밑의 sw을 실행하지 않게 하기 위함이다.

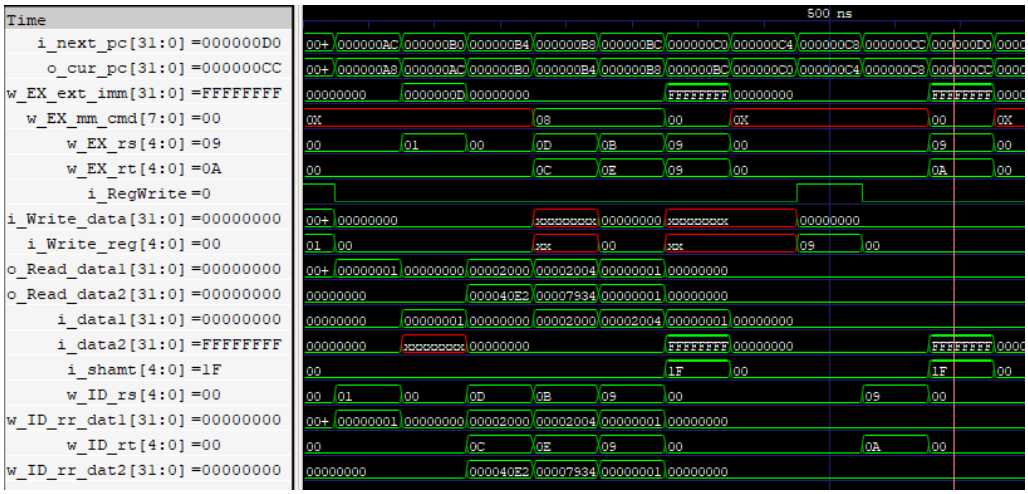
12.




```
addi  $9, $9, -1
nop
nop
nop
addi  $10, $9, -1
```

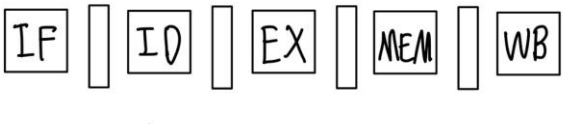
$t1	9	0x00000000
$t2	10	0x00000000

0x000000b4	0x2129ffff	addi $9,$9,0xffffffff	54:	addi	$9, $9, -1
0x000000b8	0x00000000	nop	55:	nop	
0x000000bc	0x00000000	nop	56:	nop	
0x000000c0	0x00000000	nop	57:	nop	
0x000000c4	0x212affff	addi $10,$9,0xffffffff	58:	addi	$10, $9, -1



12번 hazard 또한 $9의 dependency를 해결하기 위해 nop를 3번 넣어주었고 0xc8에서 이전 명령어의 결과가 $9에 쓰여진 후 다음 명령어의 ID에서 register fetch되어 0xcc에서 연산에 사용되는 것을 알 수 있다.

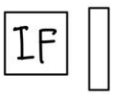
13.



Nop

Nop

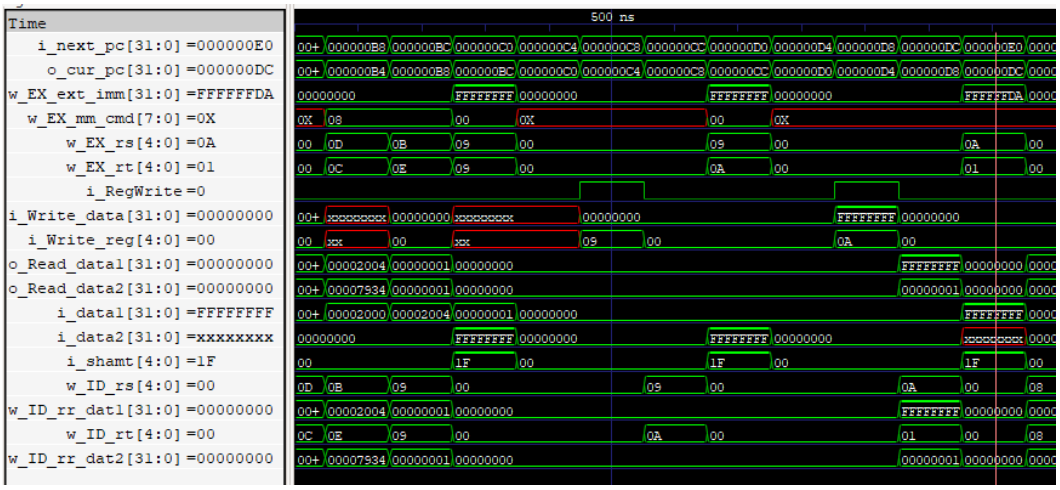
Nop




```
addi  $10, $9, -1
nop
nop
nop
bgez  $10, L2
```

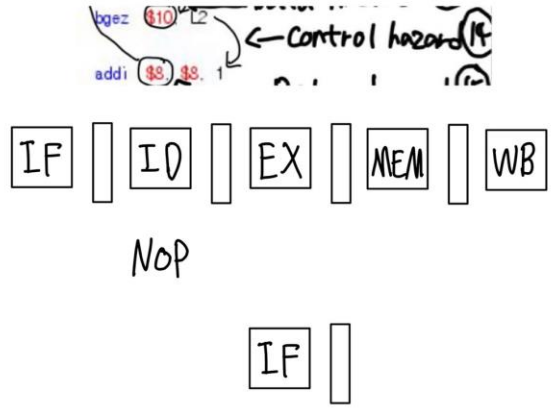
$t1	9	0x00000000
$t2	10	0xffffffff

0x000000c4	0x212affff	addi $10,$9,0xffffffff	58:	addi $10, $9, -1
0x000000c8	0x00000000	nop	59:	nop
0x000000cc	0x00000000	nop	60:	nop
0x000000d0	0x00000000	nop	61:	nop
0x000000d4	0x0541ffda	bgez $10,0xffffffda	62:	bgez $10, L2



13번 hazard도 nop를 3개 사용하였고 이 또한 12번 hazard와 동일하게 $10의 dependency를 해결하기 위함이다.

14.

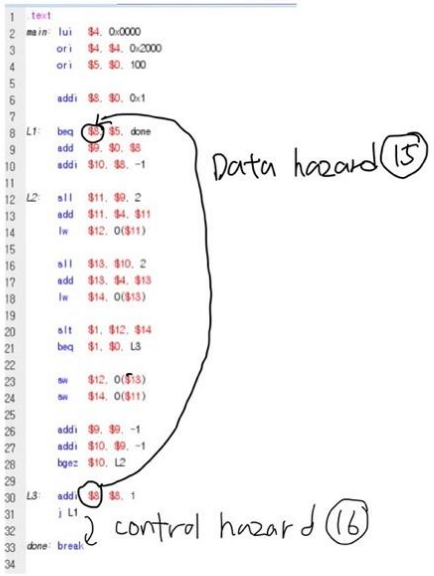


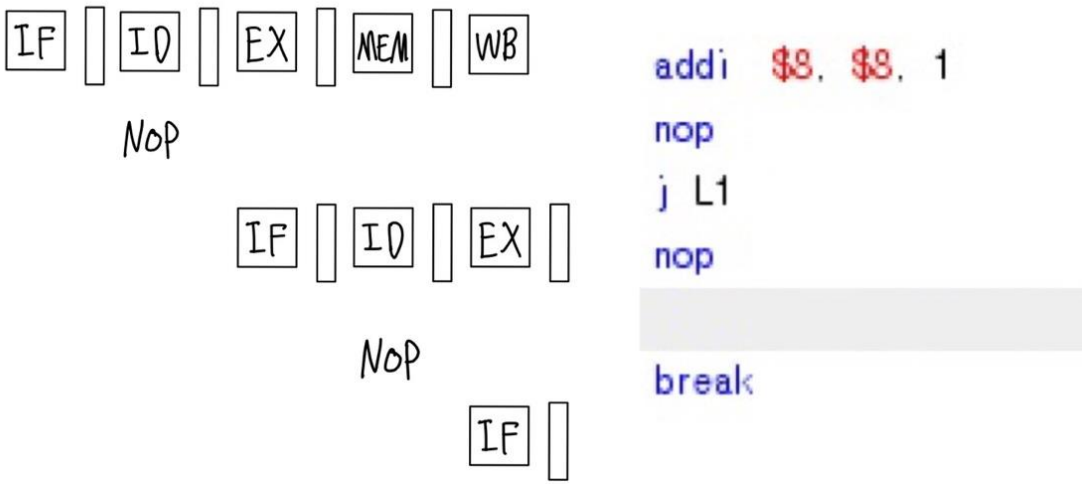

```
bgez  $10, L2
nop

addi  $8, $8, 1
```

0x000000d4	0x0541ffda	bgez $10,0xffffffda	62:	bgez $10, L2
0x000000d8	0x00000000	nop	63:	nop
0x000000dc	0x21080001	addi $8,$8,0x00000001	65: L3:	addi $8, $8, 1

14번 hazard도 3번,11번 hazard와 같은 control hazard로 흐름이 분기된다면 addi가 실행되지 않아야 하므로 nop를 추가해주었다.

15,16.

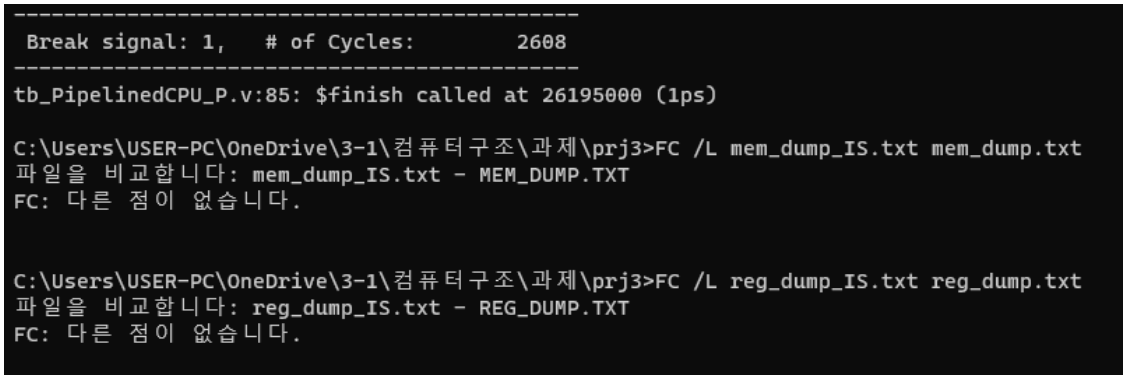




0x00000084	0x21080001	addi $8,$8,0x00000001	43: L3: addi $8, $8, 1
0x00000088	0x00000000	nop	44: nop
0x0000008c	0x08000007	j 0x0000001c	45: j L1
0x00000090	0x00000000	nop	46: nop
0x00000094	0x0000000d	break	49: done: break

마지막으로 15,16번 hazard는 addi의 $8과 L1으로 jump하여 분기하였을 때 L1의 beq가 사용하는 $8간의 data hazard가 존재하고 j명령어와 break문에 해당하는 부분에서 j명령어가 실행되고 있을 때 break문이 실행된다면 분기를 할 수 없기 때문에 나타나는 control hazard를 막기위해서 nop를 넣어주었다.

이렇게 nop만을 사용하여 hazard를 해결한 결과 총 cycle은 다음과 같다.



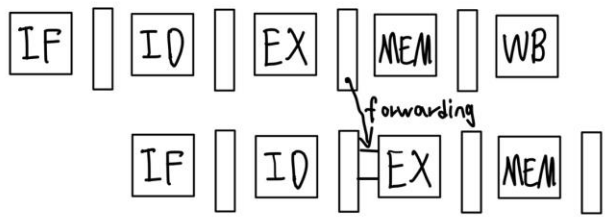
이번엔 nop와 forwarding을 사용하여 nop를 더 줄여 performance의 향상을 기대한다.

FWD_ALU_Ai 와 FWD_ALU_Bi는 파이프라인 stage간 데이터 forwarding을 제어하는 부분이다. FWD_ALU_Ai는 register fetch이후 rs에서 읽은 데이터가 ALU쪽으로 들어갈 때 해당하는 ALU input 부분의 데이터를 선택하기 위한 signal로 작동하고 FWD_ALU_Bi는 rt나 immediate값이 들어가는 ALU input 부분의 데이터를 선택하기 위한 signal로 작동한다. 따라서 00일때는 register에서 읽혀진 값이 그대로 ALU input으로 들어가고 01일때는 MEM 단계에서 ALU 출력값에 해당하는 값을 다시 ALU input으로 들어간다. 10일때는 WB단계에서 지니는 데이터를 ALU input으로 사용할 때 쓴다. 11은 reserved이다.

1.

```
1  .text
2  main: lui  $4, 0x0000
3        ori  $4, $4, 0x2000
```

Data hazard ①

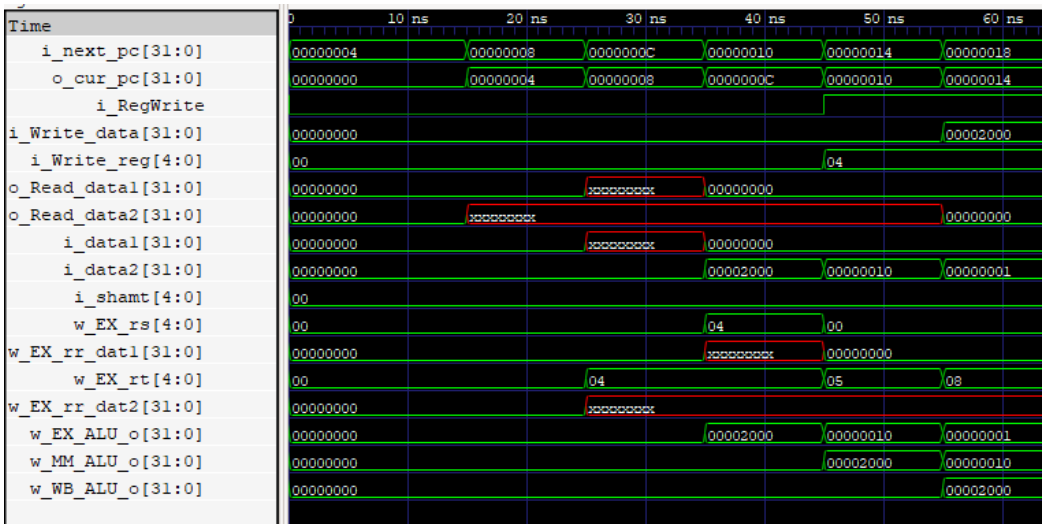



```
t
: lui   $4, 0x0000
  ori   $4, $4, 0x2000
```

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

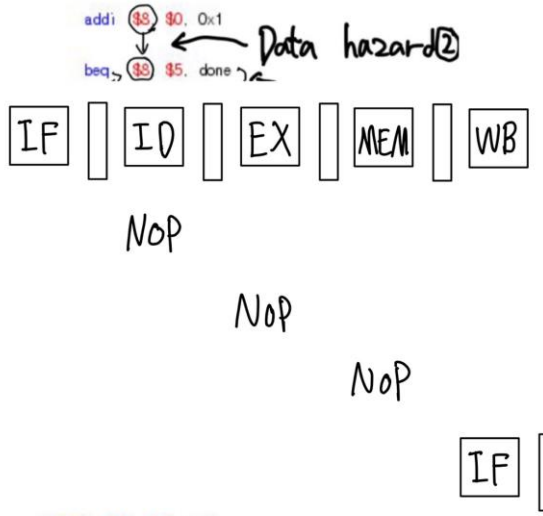
$a0	4	0x00000000
$a1	5	0x00000000

0x00000000	0x3c040000	lui $4,0x00000000	2: main: lui $4, 0x0000
0x00000004	0x34842000	ori $4,$4,0x00002000	3: ori $4, $4, 0x2000



lui의 $4결과가 ori의 rs를 연산하는 ALU입력 자리로 forwarding이 된 모습이다. 0x08에서는 ori명령어의 register fetch가 일어나는 부분이다. 값이 없기 때문에 $4를 읽어도 x를 나타내지만 forwarding이 진행되어 0x0c단계 즉 EX단계의 ALU input에 $4의 값이 들어가는 것을 알 수 있다.

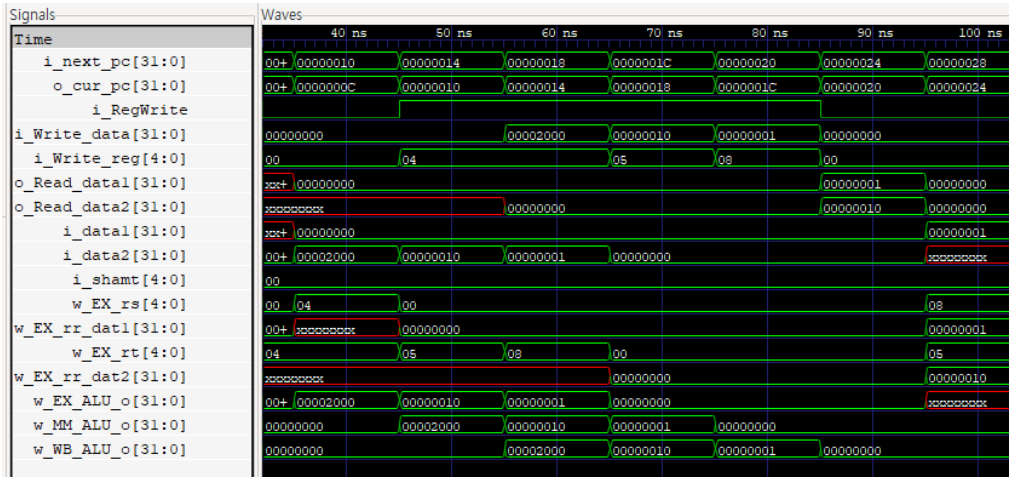
2.




```
addi  $8, $0, 0x1
nop
nop
nop

beq   $8, $5, done
```

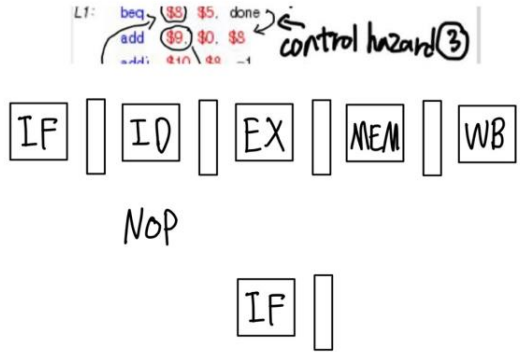
$zero	0	0x00000000
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000000

0x0000000c	0x20080001	addi $8,$0,0x00000001	6:	addi	$8, $0, 0x1
0x00000010	0x00000000	nop	7:	nop	
0x00000014	0x00000000	nop	8:	nop	
0x00000018	0x00000000	nop	9:	nop	
0x0000001c	0x1105001d	beq $8,$5,0x0000001d	11: L1:	beq	$8, $5, done



2번 hazard같은 경우는 nop를 3번 넣을 수 밖에 없다. 해당 과제 회로에서는 branch의 조건을 연산하기 위해 ALU를 사용하지 않고 ID단계에서 register값을 읽자마자 비교기를 통해 분기 여부를 결정한다. 또한 ID단계로의 forwarding을 지원하지 않기 때문에 이전 명령어에서 register에 값을 쓰는 WB단계를 거친 후 beq명령의 ID단계를 진행해야 하므로 nop만을 사용하였다.

3.

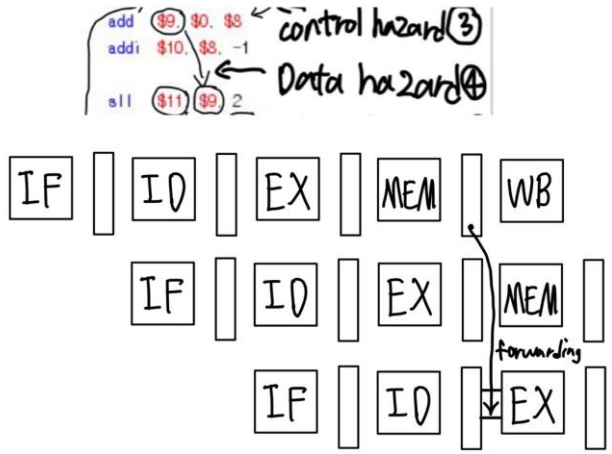



```
L1:   beq   $8, $5, done
      nop
      add   $9, $0, $8
```

0x0000001c	0x1105001d	beq $8,$5,0x0000001d	11: L1:	beq	$8, $5, done
0x00000020	0x00000000	nop	12:	nop	
0x00000024	0x00084820	add $9,$0,$8	13:	add	$9, $0, $8

3번 hazard도 nop만을 사용한다. 이 경우는 nop만을 사용하여 hazard를 해결했던 경우와 같은 경우이다. branch를 하지 않으면 상관없지만 branch를 하게될 때 하지 않아도 되는 add 명령을 하지 않기 위해 nop를 삽입하였다.

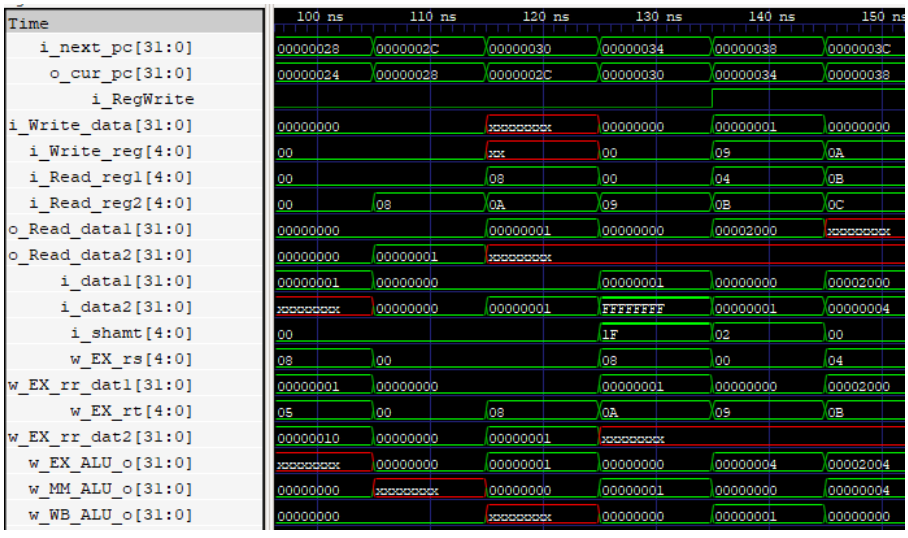
4.




```
add  $9, $0, $8
addi $10, $8, -1        00_10  //  0x028

sll  $11, $9, 2
```

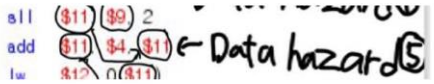
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00000000

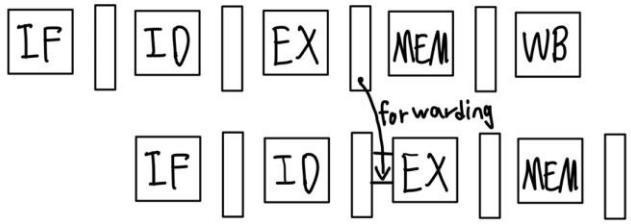
0x00000024	0x00084820	add $9,$0,$8	13:	add	$9, $0, $8
0x00000028	0x210affff	addi $10,$8,0xffffffff	14:	addi	$10, $8, -1
0x0000002c	0x00095880	sll $11,$9,0x00000002	16: L2:	sll	$11, $9, 2



4번 hazard는 add명령의 결과값을 담을 $9와 sll명령어의 rt에 해당하는 $9와의 dependency에서 일어난다. 따라서 add의 WB단계에서 가지고 있고 $9에 아직 쓰지 못한 값을 forwarding한다. 0x30에서는 sll의 ID단계가 진행되고 있는데 rt값으로 $9를 읽고 있는데 해당 값은 아무 값이 없는 상태이므로 x를 나타낸다. 하지만 바로 다음 단계의 ALU의 rt 입력부분으로 forwarding되어 0x01을 넣어주는 것을 확인한다.

5.



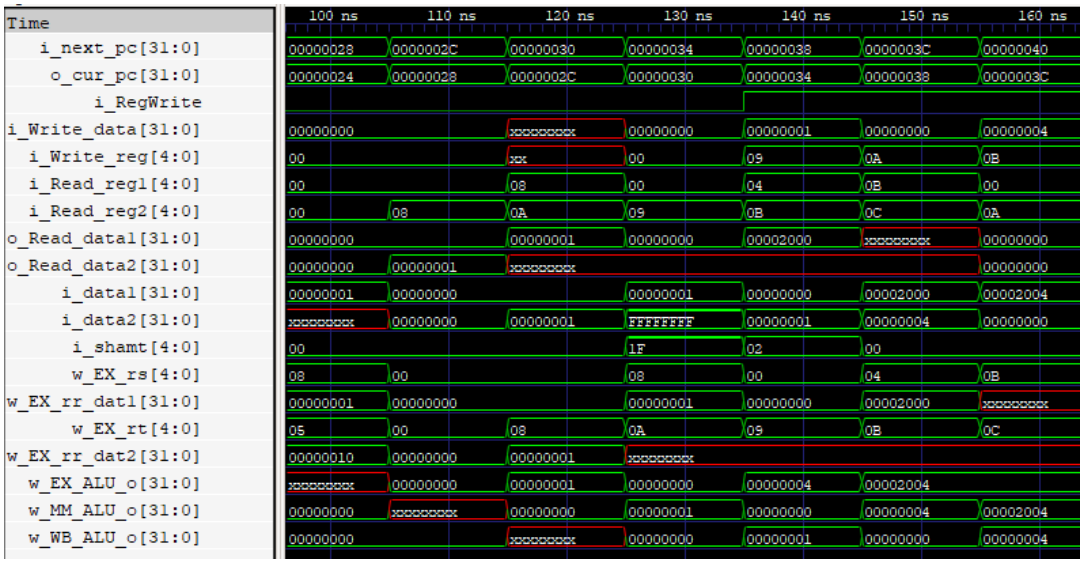



```
sll  $11, $9, 2
add  $11, $4, $11
```

```
00_01  //  0x02C
```

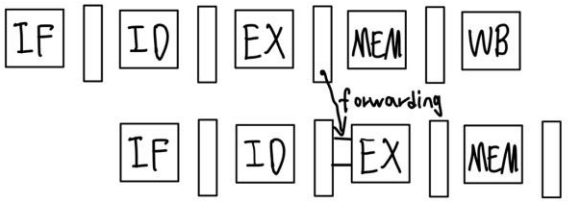
$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00000004

0x0000002c	0x00095880	sll $11,$9,0x00000002	16: L2:	sll	$11, $9, 2
0x00000030	0x008b5820	add $11,$4,$11	17:	add	$11, $4, $11



5번 hazard도 sll의 $11의 값이 add의 rt값 쪽으로 forwarding 되어야 한다. 따라서 add의 ID단계인 0x34에서 rs는 $4,rt는 $11을 읽지만 $11의 값이 x로 읽힌다. 하지만 다음 단계에서 sll의 결과값이 forwarding되어 ALU의 입력으로 들어감을 확인한다.

6.

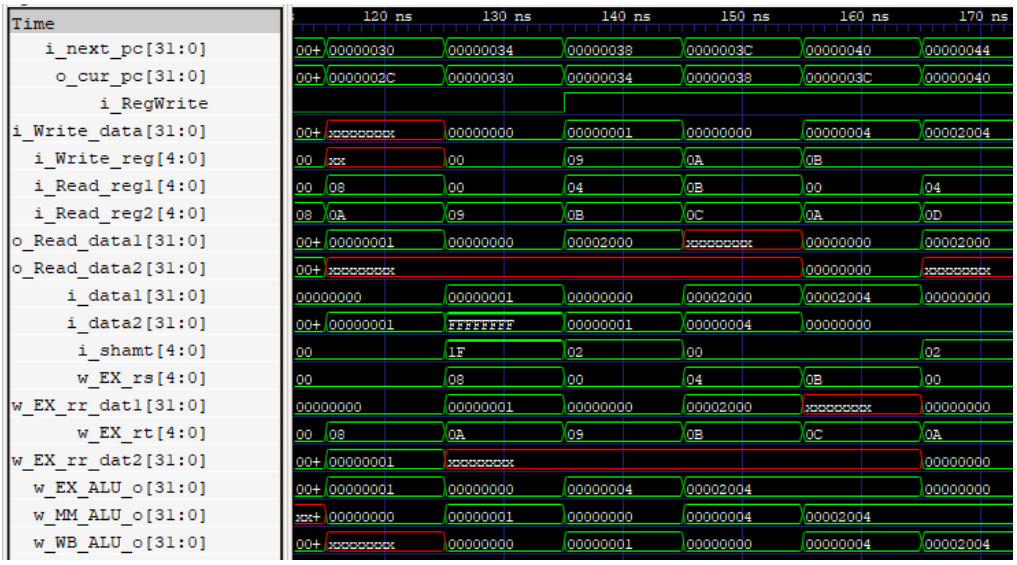



```
add   $11, $4, $11
lw    $12, 0($11)
```

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

$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00002004
$t4	12	0x00000000

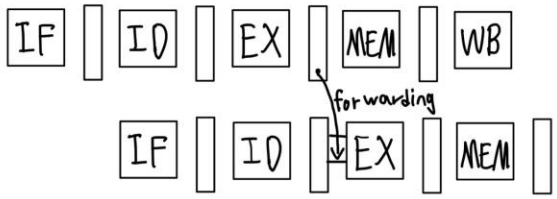
0x00000030	0x008b5820	add $11,$4,$11	17:	add	$11, $4, $11
0x00000034	0x8d6c0000	lw $12,0x00000000($11)	18:	lw	$12, 0($11)



6번 hazard는 add의 $11과 lw의 $11이 서로 dependency를 가지는 경우이다. 따라서 add의 MEM단계에서 lw의 ALU rs쪽 input으로의 forwarding이 진행되어야한다.

0x38에서 lw의 ID단계가 진행되는데 $11과 $12를 읽지만 값이 x임을 알 수 있다. 하지만 ALU에서는 forwarding이 되었으므로 $11의 값 0x2004가 들어가는 것을 알 수 있다.

7.

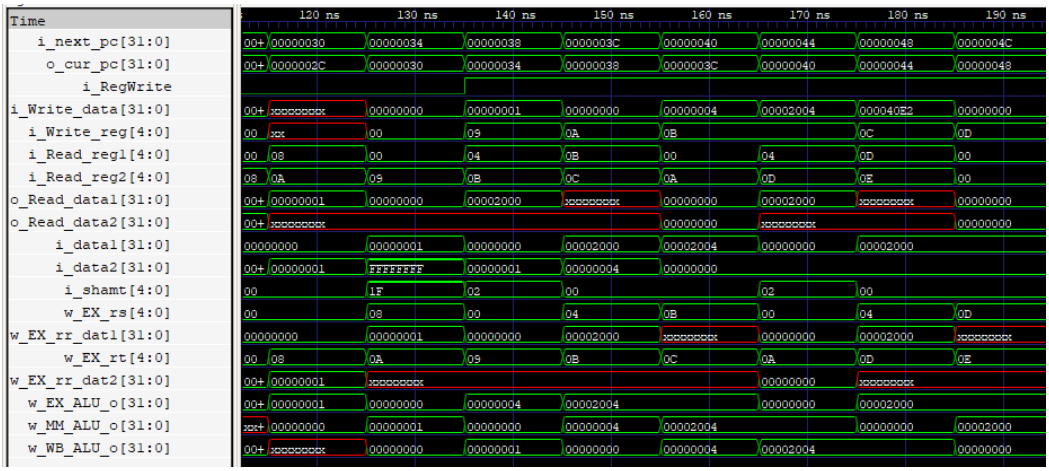



```
sll   $13, $10, 2
add   $13, $4, $13
```

00_01 // 0x038

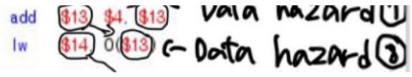
$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00002004
$t4	12	0x000040e2
$t5	13	0x00000000

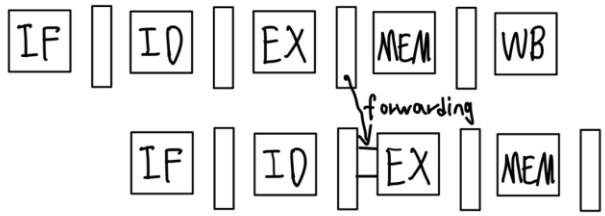
0x00000038	0x000a6880	sll $13,$10,0x00000002	20:	sll	$13, $10, 2
0x0000003c	0x008d6820	add $13,$4,$13	21:	add	$13, $4, $13



7번 hazard에서는 sll의 $13과 add의 rt쪽 $13과의 dependency을 확인한다. 따라서 add명령의 ALU rt쪽 입력에 sll에서 연산된 $13값이 forwarding되어야 한다. 0x40을 보면 register에서 $4와 $13을 읽지만 $13에서는 값이 x로 읽힌다. 하지만 다음 단계의 ALU에서는 rt쪽 값이 0x0으로 넣어진 것을 알 수 있다.

8.



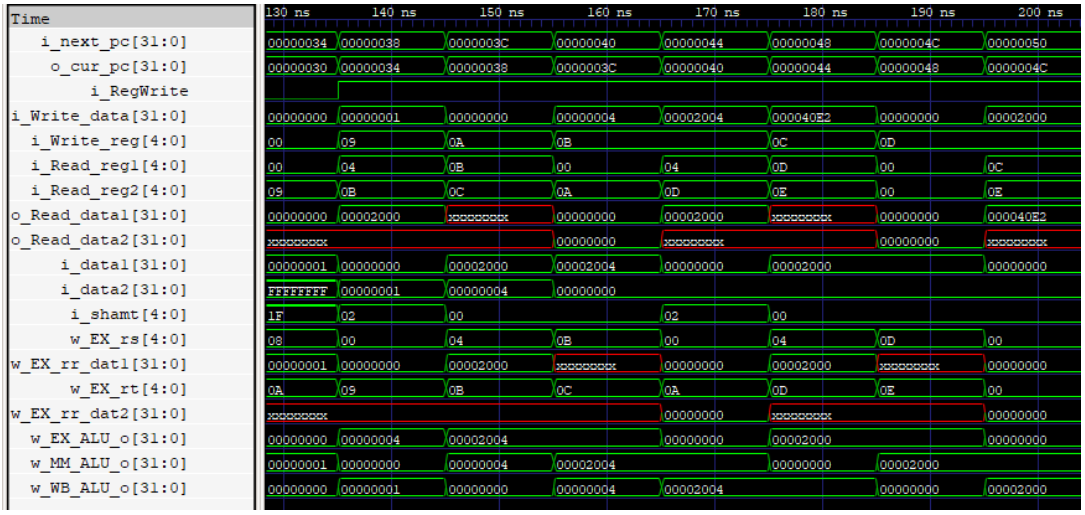


add $13, $4, $13
lw $14, 0($13)

01_00 // 0x03C

$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00002004
$t4	12	0x000040e2
$t5	13	0x00002000
$t6	14	0x00000000

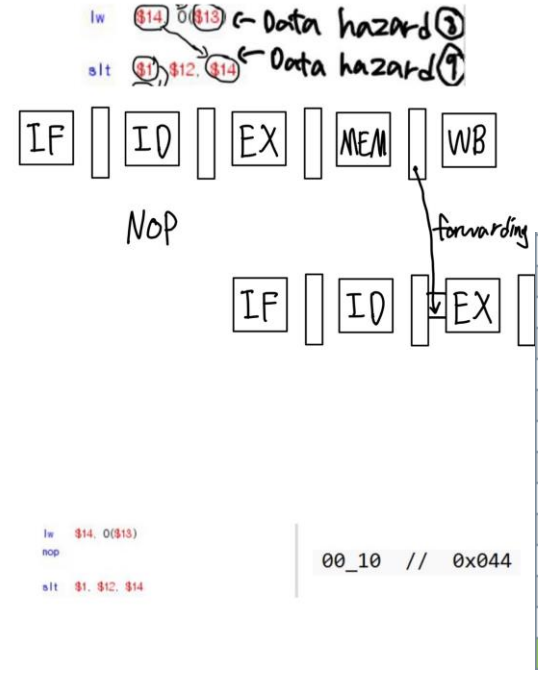
0x0000003c	0x008d6820	add $13,$4,$13	21:	add	$13, $4, $13
0x00000040	0x8dae0000	lw $14,0x00000000($13)	22:	lw	$14, 0($13)



8번 hazard에서는 add의 결과값이 저장되는 $13과 lw의 $13간의 dependency이다.

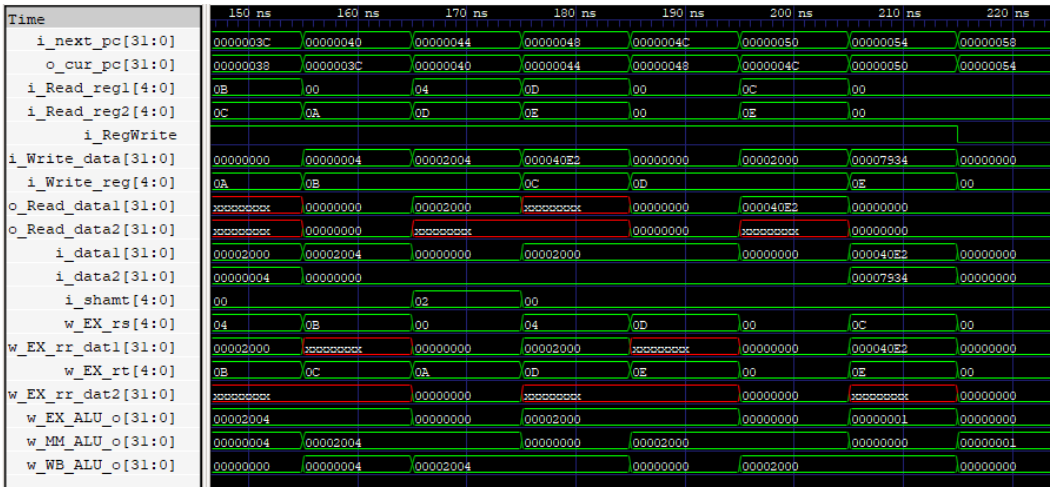
lw의 EX단계에서 ALU의 rs쪽 input으로 add의 결과값이 forwarding되어야하며 이는 0x44에서는 lw의 register fetch값이 x이지만 0x48에서는 ALU의 $13으로서의 값이 add의 결과값으로서 제대로 forwarding된 것을 알 수 있다.

9.



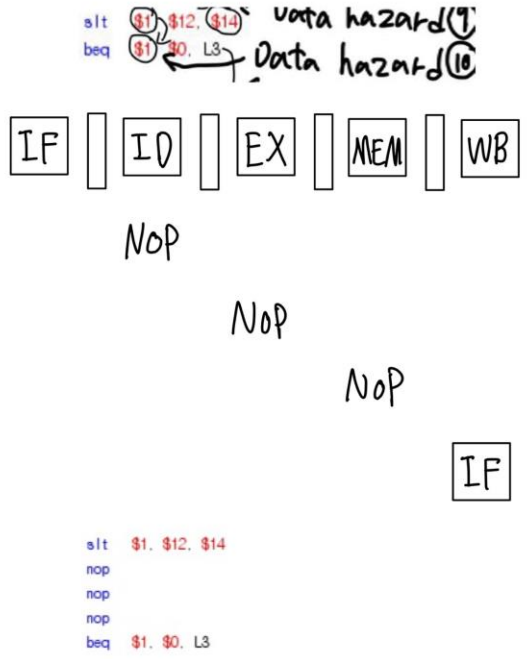
$at	1	0x00000000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00002004
$t4	12	0x000040e2
$t5	13	0x00002000
$t6	14	0x00007934

0x00000040	0x8dae0000	lw $14,0x00000000($13)	22:	lw	$14, 0($13)
0x00000044	0x00000000	nop	23:	nop	
0x00000048	0x018e082a	slt $1,$12,$14	25:	slt	$1, $12, $14



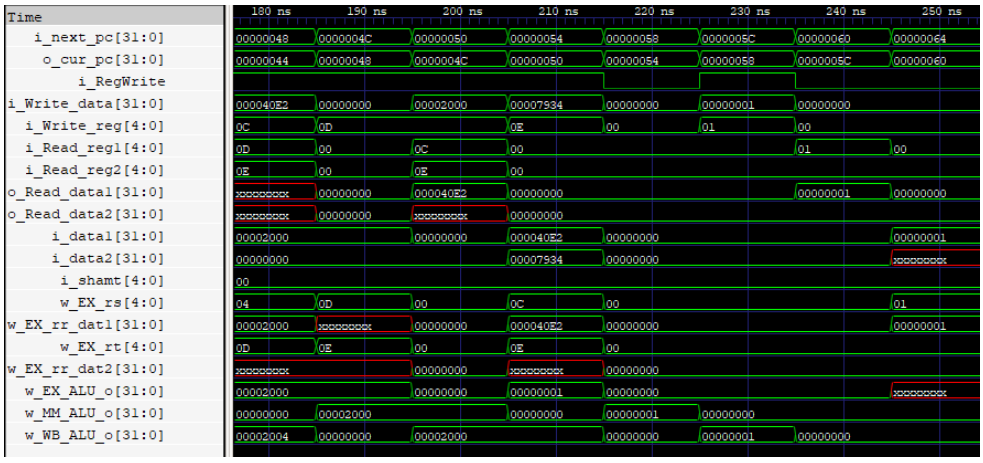
9번 hazard에서 lw명령어의 $14와 slt명령어의 $14간의 dependency를 나타낸다. lw는 $14값을 mem에서 얻어오기 때문에 WB단계에서 slt의 ALU input으로 forwarding 해야한다. 따라서 slt명령 전 부분에 lw명령어에서 값을 준비할 동안 EX단계를 늦추기 위해 nop를 하나 넣어 주었고 따라서 0x50에서 ALU의 rt쪽 input으로 값이 올바르게 forwarding되는 것을 확인할 수 있다.

10.



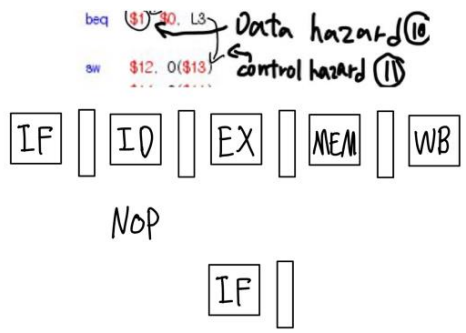
$at	1	0x00000001
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00002000
$a1	5	0x00000010
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x00000001
$t1	9	0x00000001
$t2	10	0x00000000
$t3	11	0x00002004
$t4	12	0x000040e2
$t5	13	0x00002000
$t6	14	0x00007934

0x00000048	0x018e082a	slt $1,$12,$14	25:	slt	$1, $12, $14
0x0000004c	0x00000000	nop	26:	nop	
0x00000050	0x00000000	nop	27:	nop	
0x00000054	0x00000000	nop	28:	nop	
0x00000058	0x1020000a	beq $1,$0,0x0000000a	29:	beq	$1, $0, L3



10번 hazard는 2번 hazard와 마찬가지로 다음 명령어가 branch 명령어이고 조건 연산을 ID에서 하기 때문에 ALU쪽으로의 forwarding이 불가능하다. 또한 ID쪽으로의 forwarding을 지원하지 않기에 nop를 3개 넣어주어 slt의 결과값이 register에 온전히 쓰여질때까지 beq의 ID단계를 미뤄야 hazard를 해결할 수 있다.

11.

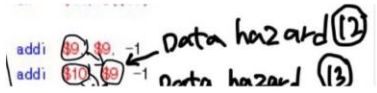


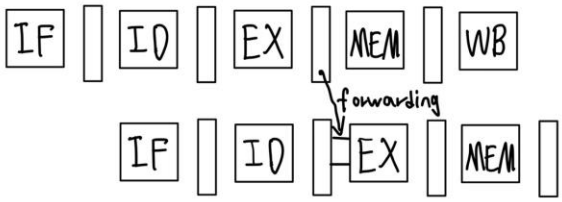

```
beq   $1, $0, L3
nop

sw    $12, 0($13)
```

0x00000058	0x1020000a	beq $1,$0,0x0000000a	29:	beq	$1, $0, L3
0x0000005c	0x00000000	nop	30:	nop	
0x00000060	0xadac0000	sw $12,0x00000000($13)	32:	sw	$12, 0($13)

11번 hazard는 흐름이 분기될 때 sw를 실행해버리는 경우를 막기위해 nop를 추가하여 새로운 흐름의 명령이 준비될때까지 기다린다.

12.



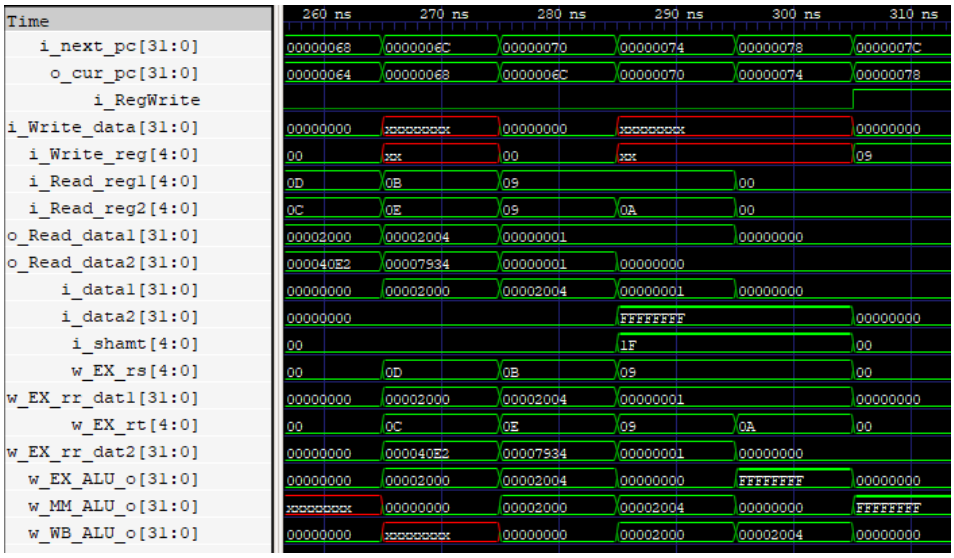



```
addi $9, $9, -1
addi $10, $9, -1
```

01_00 // 0x068

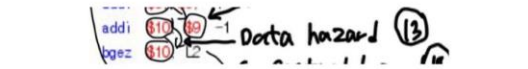
$t1	9	0x00000000
$t2	10	0x00000000

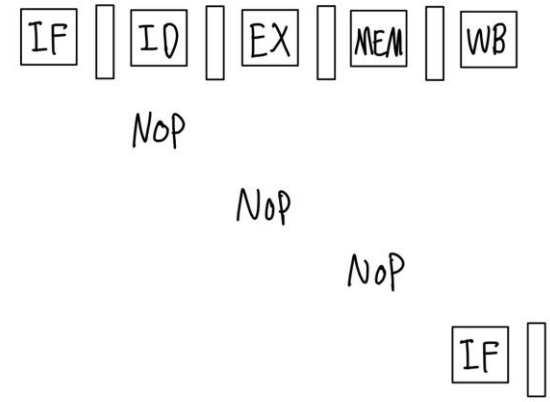
0x00000068	0x2129ffff	addi $9,$9,0xffffffff	35:	addi	$9, $9, -1
0x0000006c	0x212affff	addi $10,$9,0xffffffff	36:	addi	$10, $9, -1



12번 hazard는 이전 명령어 addi의 결과값이 저장될 $9와 다음 명령어 addi의 rs쪽 $9간의 dependency이다. 다음 명령어의 ID단계에 해당하는 0x70에서 $9값이 0x01로 읽히지만 이전 명령 addi의 결과값의 반영으로 다음명령 addi의 EX단계 ALU rs쪽 input으로는 0x0으로 forwarding되어 들어간 것을 알 수 있다.

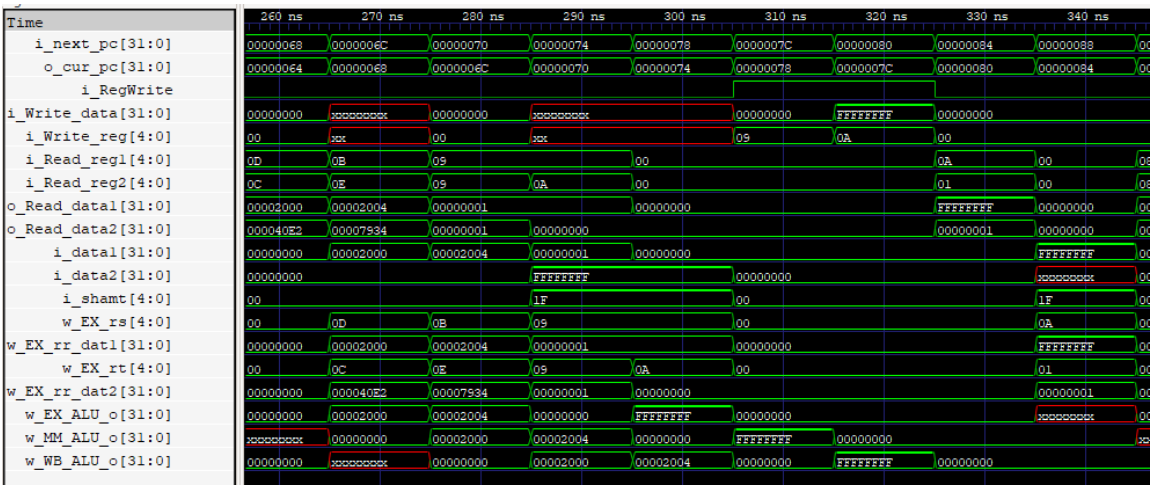
13.





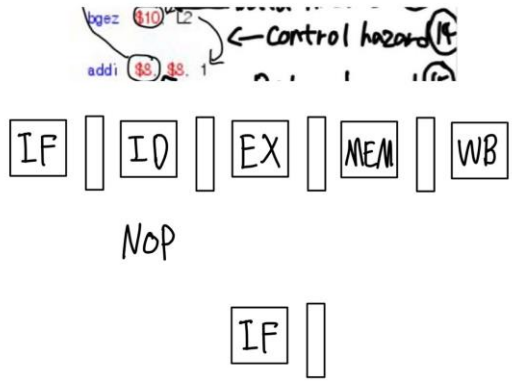
$t1	9	0x00000000
$t2	10	0xffffffff

0x0000006c	0x212affff	addi $10,$9,0xffffffff	36:	addi $10, $9, -1
0x00000070	0x00000000	nop	37:	nop
0x00000074	0x00000000	nop	38:	nop
0x00000078	0x00000000	nop	39:	nop
0x0000007c	0x0541ffeb	bgez $10,0xffffffeb	40:	bgez $10, L2



13번 hazard도 다음 명령어가 branch명령어이기 때문에 forwarding을 진행할 수 없다. 따라서 nop를 3개 넣어주었다.

14.

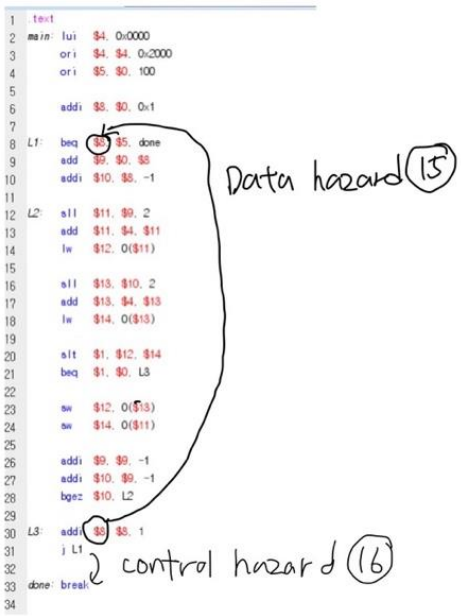


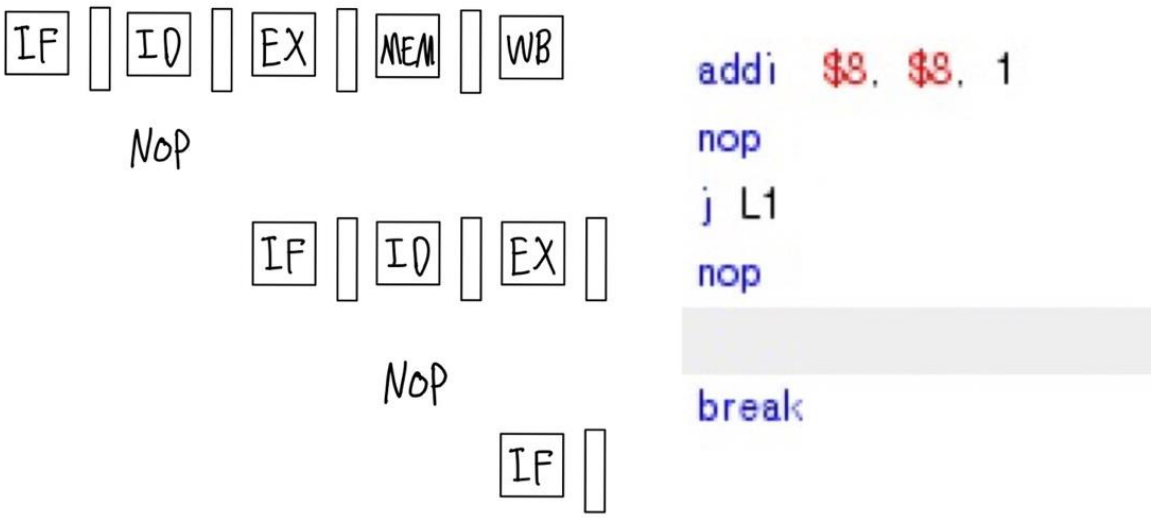

```
bgez  $10, L2
nop

addi  $8, $8, 1
```

0x0000007c	0x0541ffeb	bgez $10,0xffffffeb	40:	bgez $10, L2
0x00000080	0x00000000	nop	41:	nop
0x00000084	0x21080001	addi $8,$8,0x00000001	43: L3:	addi $8, $8, 1

14번 hazard또한 bgez가 실행흐름을 바꾼다면 addi의 IF단계가 실행되어 버리므로 이를 방지하기 위해 nop를 추가해주어 새로운 실행 흐름이 준비될때까지 addi를 진행하지 않도록 한다.

15,16.





0x00000084	0x21080001	addi $8,$8,0x00000001	43: L3: addi $8, $8, 1
0x00000088	0x00000000	nop	44: nop
0x0000008c	0x08000007	j 0x0000001c	45: j L1
0x00000090	0x00000000	nop	46: nop
0x00000094	0x0000000d	break	49: done: break

15번 hazard는 아래 j명령어에 의해 흐름이 L1쪽으로 바뀔 때 바뀌자마자 오는 beq 명령어와의 data hazard이므로 이는 forwarding을 진행할 수도 없기에 nop를 넣어주었다. 16번 hazard는 j명령이 온전하게 일을 마치기 전에 break문이 실행된다면 L1으로 jump할 수 없기에 nop를 추가해주었다.

모든 설명을 마무리하고 forwarding과 nop를 적절하게 사용하였을 때의 총 cycle 수이다. nop만 사용했을 때보다 cycle수가 현저히 줄었다. forwarding을 사용하면 nop를 사용하는 것보다 데이터가 준비될때까지 기다려야 하는 delay를 줄일 수 있기 때문에 pipeline의 사용 목적에 더 상응하는 결과를 나타낸다. 데이터를 넘겨주면 된다는 특징 때문에 nop를 사용하지 않고 바로 한 클록 마다 IF가 일어나고 명령어가 마무리되기 때문에 cpi를 1로 맞출 수 있다. 반복문이 아닌 프로그램이었다면 그 효과가 크진 않았겠지만 nop만을 사용한 cycle수와 forwarding을 최대한 활용한 아래 결과의 cycle수의 비교로 forwarding을 사용하면 performance의 큰 향상을 기대할 수 있다는 것이 입증된다.

```
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\USER-PC\OneDrive\3-1\컴퓨터구조\과제\prj3>FC /L mem_dump_IS.txt mem_dump.txt
파일을 비교합니다: mem_dump_IS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.


C:\Users\USER-PC\OneDrive\3-1\컴퓨터구조\과제\prj3>FC /L reg_dump_IS.txt reg_dump.txt
파일을 비교합니다: reg_dump_IS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.
```

<Conclusion>

이번 프로젝트에서는 data hazard와 control hazard가 언제 일어나는지 또 어떻게 해결하는지에 대해 실제 assembly code를 통해 살펴보았다. 또한 단순히 nop만으로 해결해보고 추가적으로 forwarding 기법을 사용해보면서 둘의 cycle 수를 비교해보았다. 이번 회로에서는 ALU로의 forwarding만을 진행한다는 점이 아쉬웠고 ID단계로의 forwarding이 가능했다면 더 performance를 높일 수 있었을 것이라고 생각한다. 추가적으로 performance를 높이는 방법에 대해 생각을 해보면 수업시간에 배운 branch prediction을 더욱 정확하게 하는 2-bit prediction같은 것을 사용하거나 분기와 별개로 꼭 실행해야하는 명령어라인이 있다면 재정렬하고 delay slot같은 것을 사용하여 보는 방법도 있을 것 같다. pipeline architecture는 기존의 single cycle과 multi cycle보다 각 단계로 나누어서 명령어를 겹쳐서 진행하니 더 높은 throughput을 보여주었다. 어려웠던 점은 branch관련한 부분이 좀 까다로웠던 것 같다. 특히 마지막 j명령어 이후 L1쪽으로 분기하자마자 생기는 hazard를 찾는 것이 어려웠다. pipeline특성상 연속적으로 명령어를 실행하면 총 5개의 명령어가 맞물리는데 첫번째와 두번째가 생길지 세번째가 생길지 네번째가 생길지 직접 하나하나 봐야 알 수 있기 때문에 어려웠던 것 같다.

하지만 pipeline을 하나하나 그려서 확인하니 흐름을 파악할 수 있었고 이를 통해 hazard를 해결하고 cycle을 줄이는 optimization을 진행할 수 있었던 것 같다.