컴퓨터 구조 실험

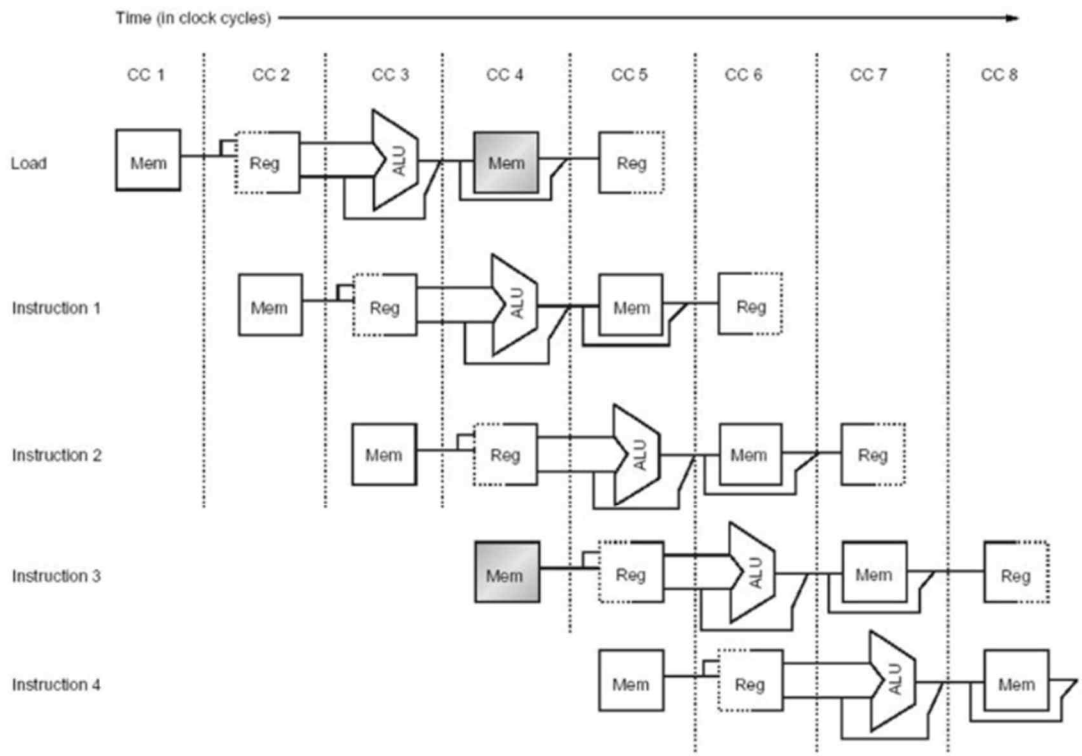
Project 3 과제

수업 명 : 컴퓨터구조실험
과제 이름 : project3
담당 교수님 : 이성원 교수님
학 번 : 2019202005
이 름 : 남종식

■ 프로젝트 이론 내용에 대한 설명

harzard설명 hazard를 피하기 위한 방법

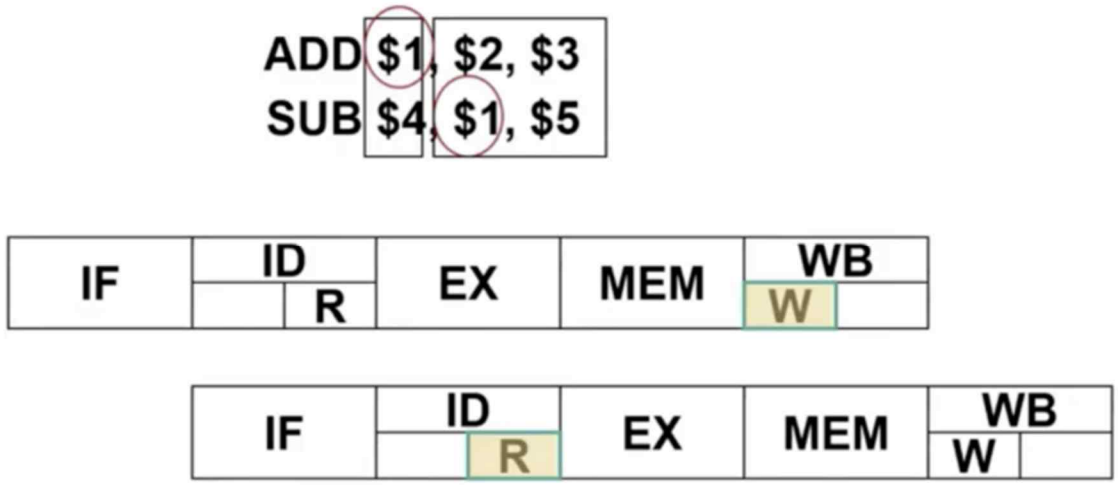
먼저 pipeline hazard란 pipeline의 방식이 적용된 프로세서에서 발생할 수 있는 잠재적인 문제입니다. 여기서 pipeline방식은 연속적으로 명령어를 실행하는 과정이며 이때 명령어 실행 순서에 의해 발생하며 이는 성능을 저하시키거나 잘못된 결과를 야기할 수 있습니다. 이 pipeline hazard는 총 3가지가 있습니다. 첫째 structural hazard, 둘째 data hazard, 셋째 control hazard가 있습니다.
Structural hazard란 간단히 말해서 프로세서의 자원이 부족해서 발생하는 문제입니다. 다시 말해서 파이프라인에서 사용되는 하드웨어 컴포넌트가 한 번에 하나의 명령어만을 처리할 수 있는 경우에 주로 발생합니다. 예를 들어, 파이프라인의 한 단계에서 여러 명령어가 같은 memory를 동시에 접근하려고 할 때 structural hazard가 발생합니다. 이로 인해 해당 단계에서는 하나의 명령어만을 처리할 수 있고, 다른 명령어는 대기 상태에 머무르게 됩니다.



위 사진을 봤을 때 load명령어에서는 data fetch를 위해 memory에 접근하는 모습을 볼 수 있고, instruction3에서는 instruction fetch를 위해 memory에 접근하는 모습을 확인할 수 있습니다. 바로 이때, structural hazard가 발생합니다.

이를 해결하기 위해서는, hardware resource를 추가하는 해결책이 있습니다. 위 사진의 경우에는 instruction memory와 data memory를 나누어 datapath에 추가하면 됩니다.

다음은 **data hazard**입니다. Data Hazard는 프로세서 파이프라인에서 발생하는 명령어 실행 순서에 의한 의존성으로 인해 발생하는 위험입니다. Data Hazard는 한 명령어가 필요한 데이터를 다른 명령어가 아직 생성하지 않은 상태에서 요구할 때 발생합니다. 프로그램의 실행 시간과 성능에 부정적인 영향을 미칠 수 있으며, 제대로 처리하지 않을 경우 오류가 발생할 수도 있습니다.



위 사진을 보면 add명령어에서 $2와 $3의 덧셈 결과를 $1에 저장하기 전에 sub명령어가 $1값을 사용해 버리는 문제가 발생합니다. 이는 결과 값이 저장되기 전에 레지스터를 읽는 경우로 이를 data hazard라고 부릅니다.
이 data hazard는 3가지 경우가 존재합니다.
Read After Write (RAW): 한 명령어가 데이터를 쓰고, 그 데이터가 아직 다른 명령어에 의해 읽히기 전에 다른 명령어가 해당 데이터를 읽는 경우 발생합니다. 이는 한 명령어의 결과를 다른 명령어가 올바르게 사용할 수 없는 상황입니다.

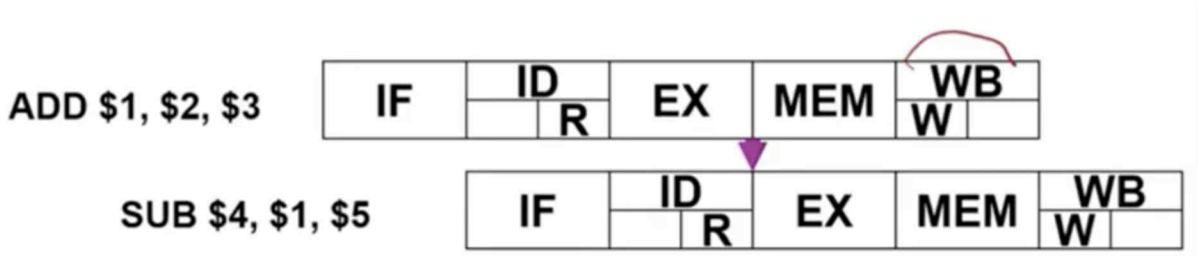
Write After Read (WAR): 한 명령어가 데이터를 읽고, 그 데이터가 아직 다른 명령어에 의해 쓰이기 전에 다른 명령어가 해당 데이터를 쓰는 경우 발생합니다. 이는 명령어가 올바른 데이터를 읽을 수 없는 상황입니다.

Write After Write (WAW): 한 명령어가 데이터를 쓰고, 그 데이터가 아직 다른 명령어에 의해 쓰이기 전에 다른 명령어가 해당 데이터를 쓰는 경우 발생합니다. 이는 명령어가 원하지 않는 데이터를 사용하거나, 데이터 불일치가 발생할 수 있는 상황입니다.

다음은 data hazard를 해결할 수 있는 방법입니다.
먼저 stall을 이용해 다음 명령어를 freeze시키는 방법입니다.

Add 명령어와 load 명령어가 $1레지스터에 값을 write하기 전까지 sub 명령어에는 stall을 넣어 아무것도 하지 않은 채 clock을 보냅니다. write과정이 끝나면 그때 $1레지스터의 값을 읽어 명령어를 수행합니다.

다음은 forwarding을 이용하는 방법입니다.



add명령어가 $1레지스터에 값을 write하기 위해서는 EX단계에서 값을 계산하는데 이때, write단계까지 기다리지 않고 EX단계에서 계산한 값을 다음

명령어 sub의 EX단계의 $1 input값으로 가져오면 됩니다. 이는 mux와 datapath의 수정이 필요하지만 stall이 필요 없으므로 시간적으로 이득을 볼 수 있습니다.

load명령어가 $1레지스터에 값을 write하기 위해서는 MEM단계에서 값을 가져오는데 이때, write단계까지 기다리지 않고 MEM단계에서 가져온 값을 다음 명령어 sub의 EX단계의 $1 input값으로 가져오면 됩니다. 이는 mux와 datapath의 수정이 필요하지만 stall이 한 번만 필요 하므로 시간적으로 1 clock 이득을 볼 수 있습니다.

다음은 compiler scheduling방법입니다. Hardware를 수정하지 않고 compiler와 assembler만을 이용해서 software를 수정하는 방법입니다.
명령어 사이에 NOP를 추가해 줌으로써 add 명령어가 WB단계일 때 sub 명령어가 ID단계일 수 있습니다.

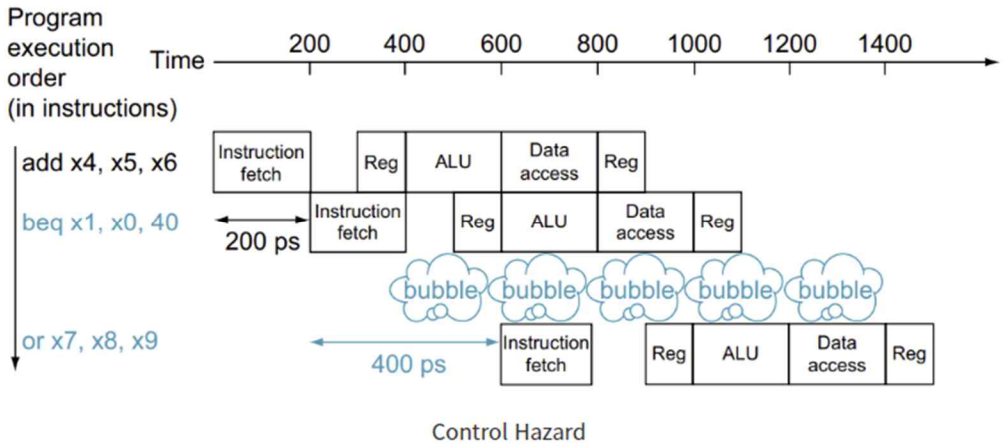
혹은 명령어의 순서를 바꿔서 해결할 수도 있습니다.

ADD $1, $2, $3
lw $6, 100($7)
AND $8,$8,$10
SUB $4, $1, $5

add명령어와 sub명령어가 사용하지 않는 register를 이용하는 명령어 2개 이상을 중간에 삽입하여 시간적 이득을 볼 수 있습니다.
마지막으로 control hazard입니다. Control Hazard는 conditional branch를 통해 PC 값을 바꿀 때, 이미 pipeline에 들어와 있는 명령어가 flush 되는 현상입니다. 즉 pipeline에 이미 반입된 다음 명령어들을 버려야 한다는 뜻입니다.

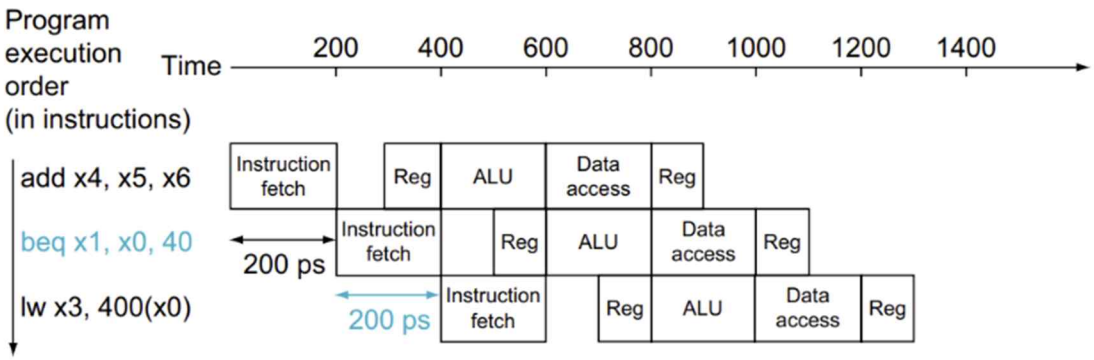
기본적으로 IF는 다음 명령어를 반입하고, branch는 MEM 단계에서 결정되어서 PC를 바꾸기 때문에, 최소 3 cycle bubble이 발생합니다.

다음은 **control hazard**의 해결법입니다. Branch를 진행할 지 안할 지에 대해 test하는 register를 추가해 명령어가 branch라는 조건과 register 두 개의 값이 같은 지의 조건이 맞는지 확인합니다. ID단계에서 register를 추가로 사용하여 문제를 해결합니다.

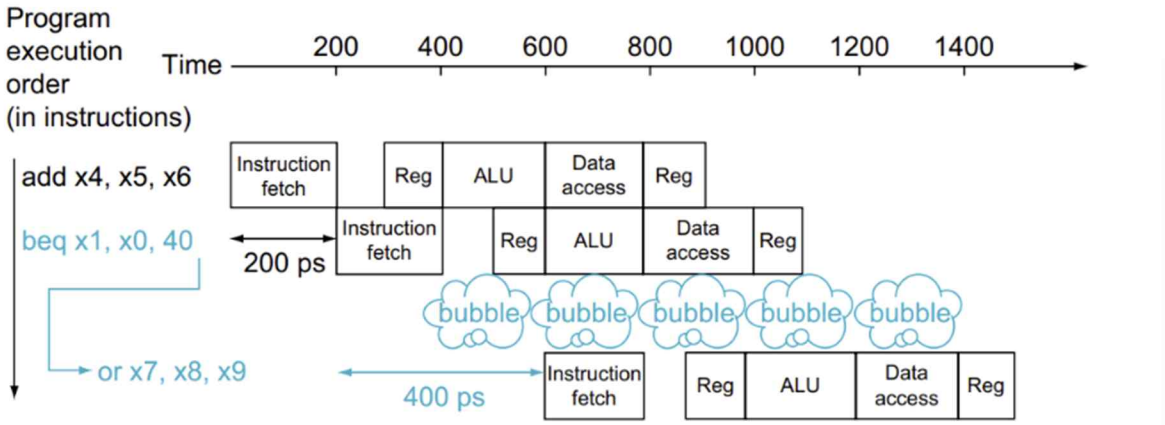


Control Hazard

이처럼 MEM 단계에서 branch가 결정되던 기존의 datapath와는 달리 ID 단계에서 결정이 일어나서 1 cycle만 손해를 봅니다. beq가 False라면 그냥 실행하지만, beq가 True 라면 branch를 다시 계산해서 멀리 있는 명령어를 가져옵니다. 하지만 명령어가 길면 register를 추가한다 해도 stall로 인한 손해는 너무 크므로 CPU는 prediction을 합니다. 이를 통해 conditional branch가 taken될 지, untaken될 지에 대해 예측합니다. 그리고 prediction이 틀린다면 stall이 발생합니다.



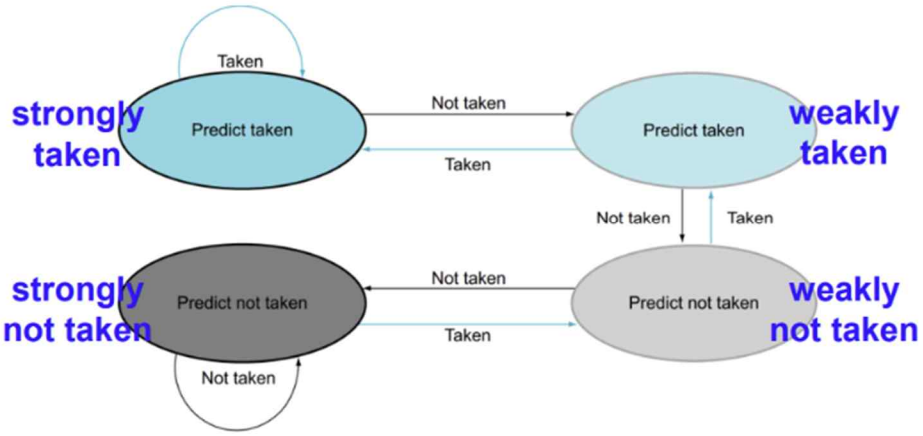
다음은 beq가 false일 때 결과입니다.



다음은 beq가 true일 때 결과입니다.

이때, prediction은 Static Branch Prediction과 Dynamic Branch Prediction으로 나뉩니다. 여기서 Dynamic Branch Prediction에 대해서 알아보겠습니다.
예측을 하기 위해선, 하드웨어에 또 다른 register를 추가해야 합니다. 이를 branch prediction buffer 혹은 branch history table이라고 합니다. branch prediction buffer는 IF 단계에 있으며, 해당 buffer는 branch가 최근에 taken 되었는지 아닌지에 대한 bit 정보를 저장하고 있습니다. 따라서 저장되어 있는 bit에 따라 예측을 수행하고, 만약에 실패한다면 업데이트를 합니다. 두 가지 종류가 있는데, 1-bit predictor와 2-bit predictor가 있습니다. 1-bit predictor에는 문제가 많이 발생할 수 있어 이를 해결하기 위한 2-bit predictor가 있습니다.

2-bit predictor에 대해서 알아보겠습니다.



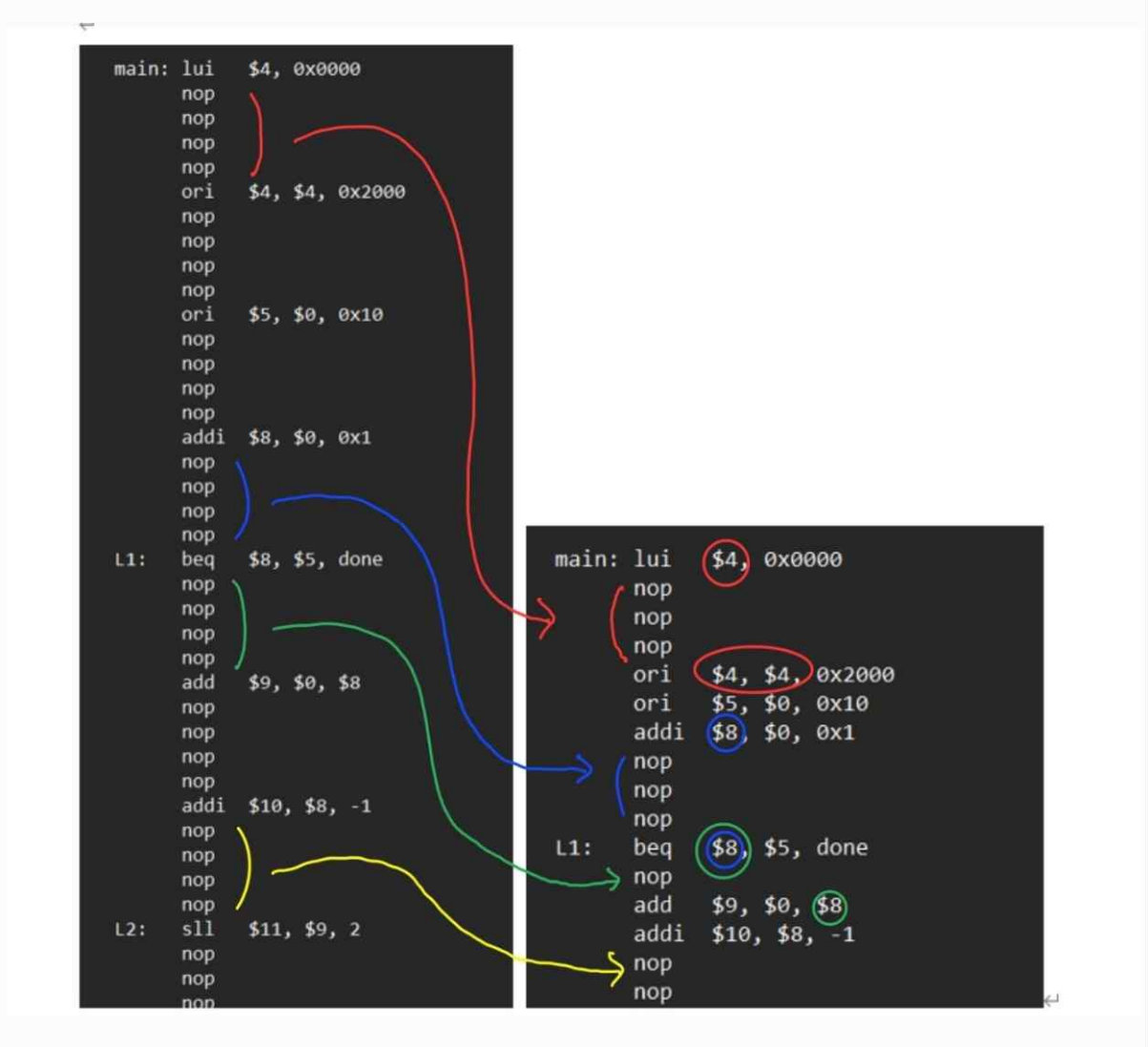
Taken 예측이 틀렸을 경우:
Strongly Taken (11) 상태는 Weakly Taken (10)로 변경됩니다.
Weakly Taken (10) 상태는 Weakly Not Taken (01)로 변경됩니다.
Weakly Not Taken (01) 상태는 Strongly Not Taken (00)로 변경됩니다.
Strongly Not Taken (00) 상태는 그대로 유지됩니다.

Not Taken 예측이 틀렸을 경우:
Strongly Taken (11) 상태는 Weakly Taken (10)로 변경됩니다.
Weakly Taken (10) 상태는 Strongly Taken (11)로 변경됩니다.
Weakly Not Taken (01) 상태는 Strongly Not Taken (00)로 변경됩니다.
Strongly Not Taken (00) 상태는 Weakly Not Taken (01)로 변경됩니다.

2-bit predictor는 상대적으로 단순한 구조를 가지고 있으면서도 분기 예측의 정확성을 높일 수 있는 기법입니다.

■ Assembly code 설명

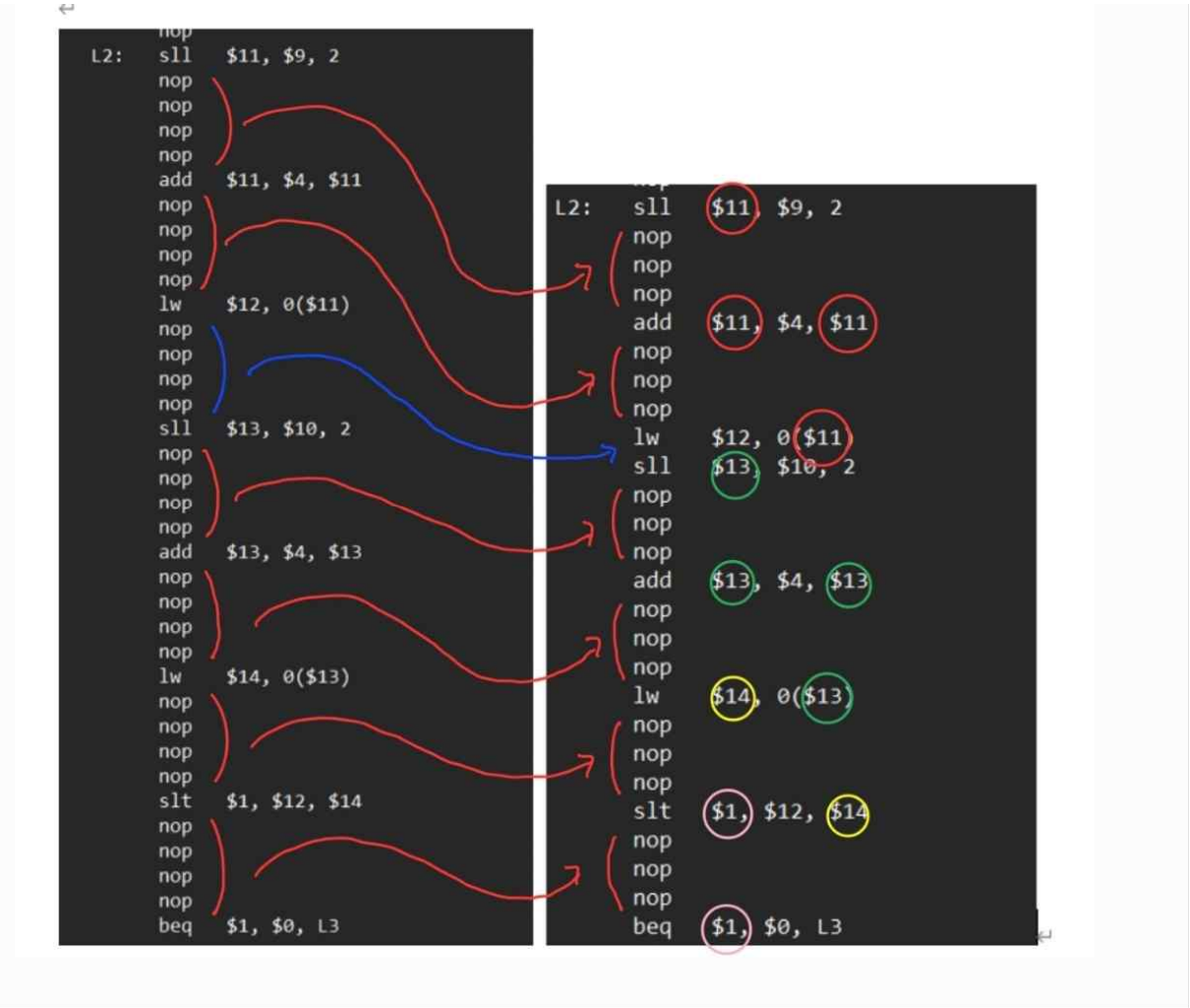
먼저 기존의 코드에서 NOP를 제거한 경우입니다.



[기존 코드] [nop제거 후 코드]

명령어의 진행단계는 IF -> ID -> EX -> MEM -> WB로 구성되어 있는데, 이전의 명령어에서 사용한 레지스터들이 바로 뒤의 명령어에서 사용되기 때문에, 이전 명령어의 WB단계 이후에 뒤의 명령어의 ID단계가 진행되어야 data hazard를 피할 수 있습니다. 빨간색 화살표와, 파란색 화살표를 보면 이전의 명령어의 레지스터에 값이 저장된 후 레지스터를 사용해야 하기 때문에 nop를 3개 넣어주어야 합니다. 기존의 코드에서 4개였던 nop를 3개로 줄인 모습을 확인할

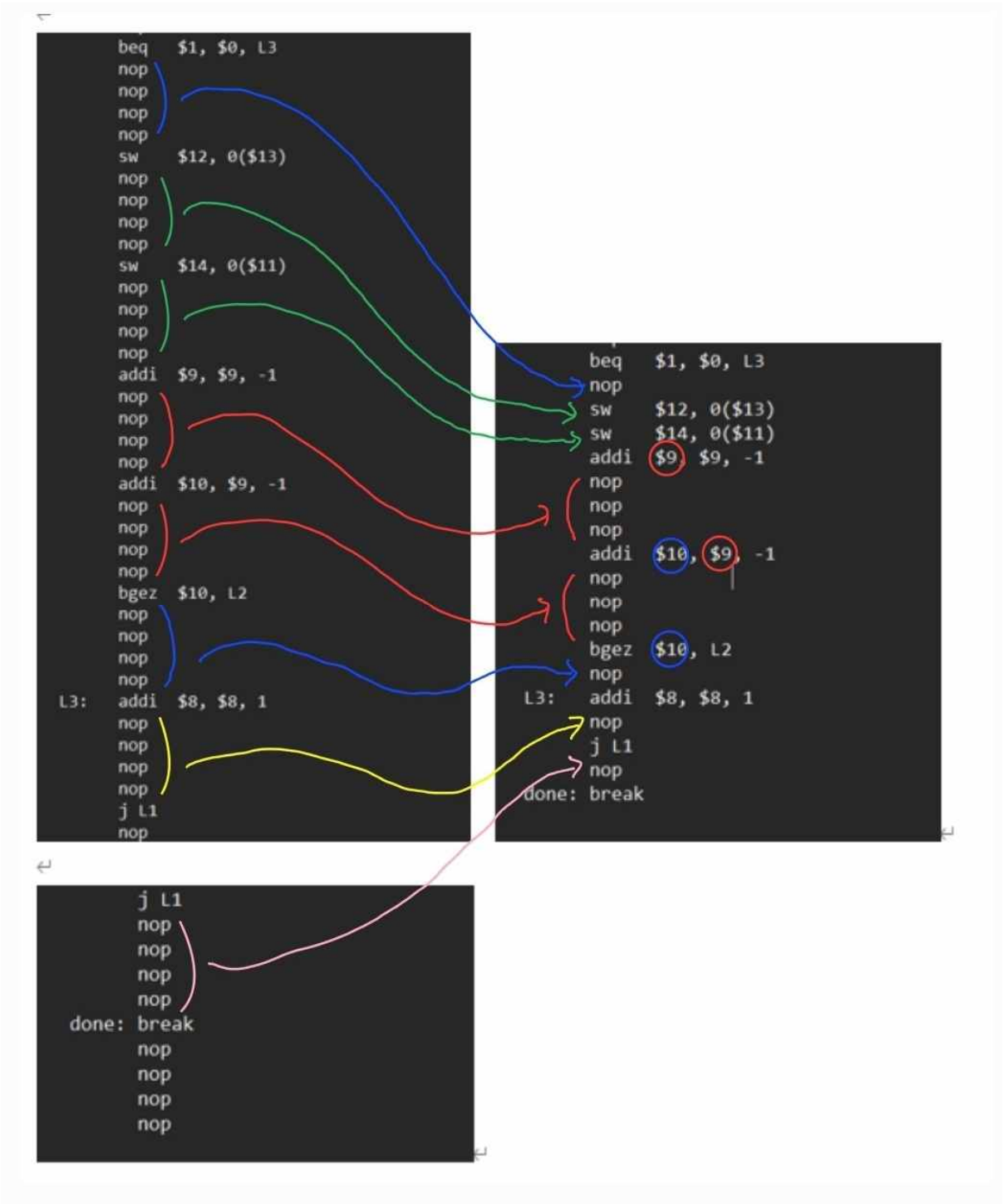
수 있습니다.
ori $4, $4, 0x2000, ori $5, $0, 0x10, addi $8, $0, 0x1 이 명령어들 사이에는 사용하는 레지스터가 겹치지 않아 nop가 없어도 data hazard가 발생하지 않으므로 nop를 모두 제거해준 모습을 확인할 수 있습니다.
초록색 화살표를 봤을 때 beq명령어의 분기 결과를 ID단계에서 가져오므로 뒤에 nop를 1개 넣어준 모습을 확인할 수 있습니다. 마지막으로 노란색 화살표를 보면 add명령어에서 $9레지스터에 값이 저장된 이후에 $9레지스터의 값을 읽어와야 하므로 nop를 2개 넣어주어 data hazard를 해결한 모습을 확인할 수 있습니다.



[기존 코드] [nop 제거 후 코드]

빨간색 화살표로 된 부분을 확인해보면 모두 이전에 레지스터의 값이 저장되기 전에 레지스터에 접근하는 모습을 확인할 수 있습니다. 이러한 경우에는 data hazard가 발생하므로 이 사이에 nop 3개를 넣어줌으로써 해결할 수 있습니다.

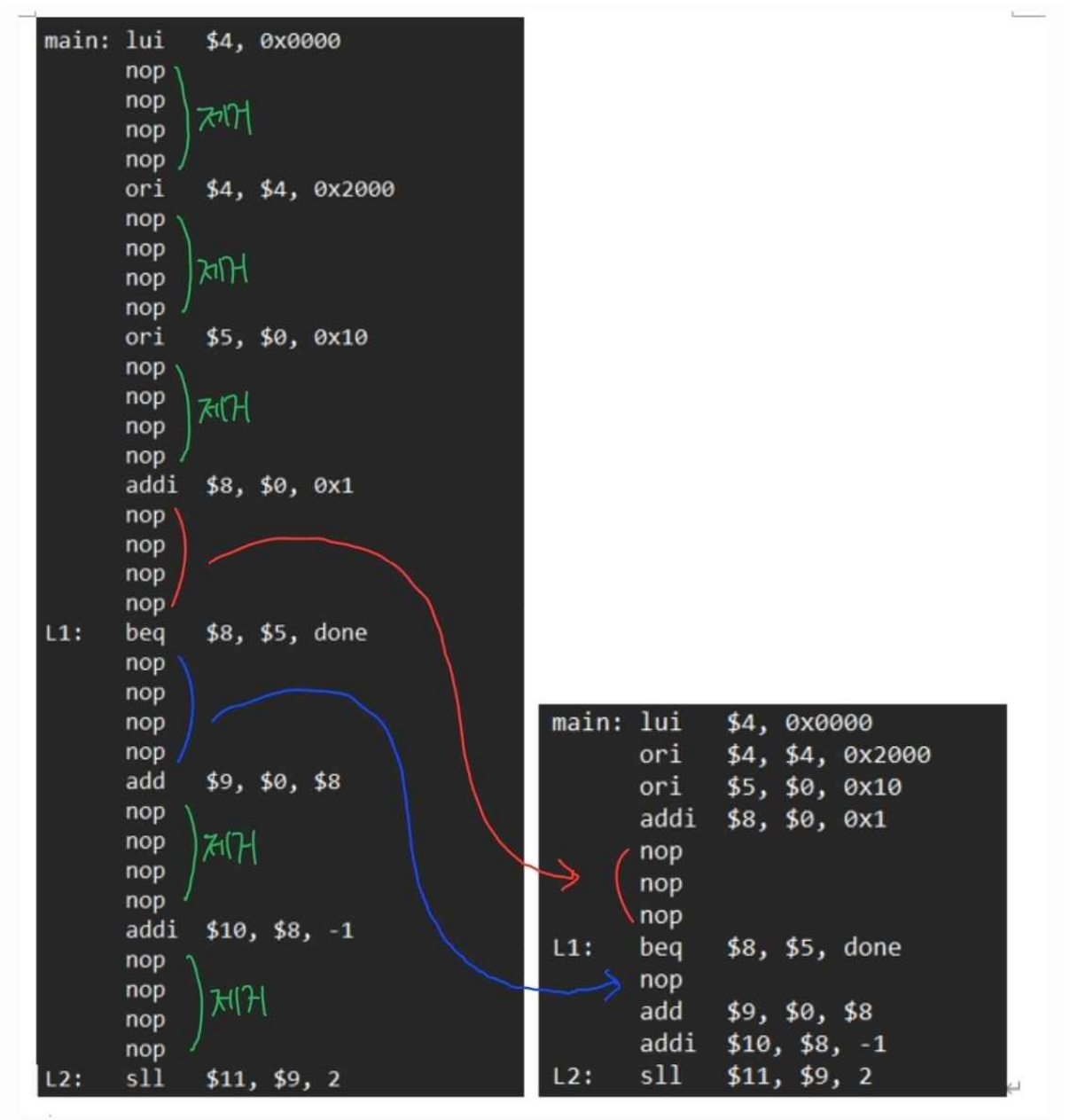
빨간색 화살표 부분 모두 기존의 nop4개에서 3개로 줄였습니다.
다음은 파란색 화살표 부분입니다. 이 경우에는 이전의 명령어에서 사용한 레지스터를 다음 명령어에서 사용하지 않으므로 nop가 없어도 data hazard가 발생하지 않습니다. 그러므로 사이에 nop를 모두 제거해 주었습니다.



[기존 코드] [nop 제거 후 코드]

먼저 빨간색 화살표 부분을 보겠습니다. 이전 명령어에서 레지스터의 값이 저장되기 전에 다음 명령어에서 레지스터에 접근하기 때문에 이때 data hazard가 발생합니다. 이를 해결하기 위해서 명령어 사이에 nop 3개를 넣어줬습니다. 기존의 코드에서 4개였던 nop를 3개로 줄인 모습을 확인할 수 있습니다.
다음으로 파란색 화살표를 보겠습니다. branch관련 명령어를 확인할 수 있는데 branch명령어는 분기 조건을 판단 후 분기가 수행되거나 수행되지 않을 수 있습니다. 만약, 분기가 수행되지 않을 경우에는 분기 이후의 명령어가 정상적으로 실행되어야 합니다. 하지만 분기가 수행된다면 분기 이후의 명령어는 필요하지 않은 명령어가 되며 분기 이후에 실행되어야 하는 명령어로 넘어가야 합니다. 이를 해결하기 위해 branch관련 명령어 뒤에 nop 하나를 추가하는 모습을 확인할 수 있습니다. 이렇게 함으로써, 분기가 수행되지 않을 경우에는 nop가 실행되어 아무런 영향도 주지 않게 됩니다. 분기가 수행될 경우에는 분기 이후의 명령어를 건너 뛰고 다음 명령어로 진행하게 됩니다.
다음으로 초록색 화살표 부분을 보겠습니다. 이 부분에서는 이전에 명령어에서 사용하는 레지스터와 다음 명령어에서 사용하는 레지스터가 서로 겹치지 않아 nop가 없어도 data hazard가 발생하지 않습니다. 그래서 기존의 코드에서 4개의 nop를 모두 제거해준 모습을 확인할 수 있습니다.
노란색 화살표 부분과 분홍색 화살표 부분을 보겠습니다. j명령어가 있는 모습을 확인할 수 있는데 j명령어 앞뒤로 nop를 넣어 주었습니다. j명령어 앞뒤에 nop를 추가함으로써 파이프라인 구조에서 분기 이후의 명령어들이 실행되지 않도록 했습니다. j명령어 앞에 nop를 추가함으로써 jump명령어가 파이프라인의 모든 스테이지를 통과할 때까지 기다리며, 뒤에 nop를 추가함으로써 jump명령어가 실행되지 않을 경우를 대비하여 분기 이후의 명령어를 제거합니다.

다음은 기존의 코드에서 forward 제어신호를 추가하여 더 많은 NOP를 제거한 경우입니다.



[기존 코드] [forward 제어신호 추가 후 코드]

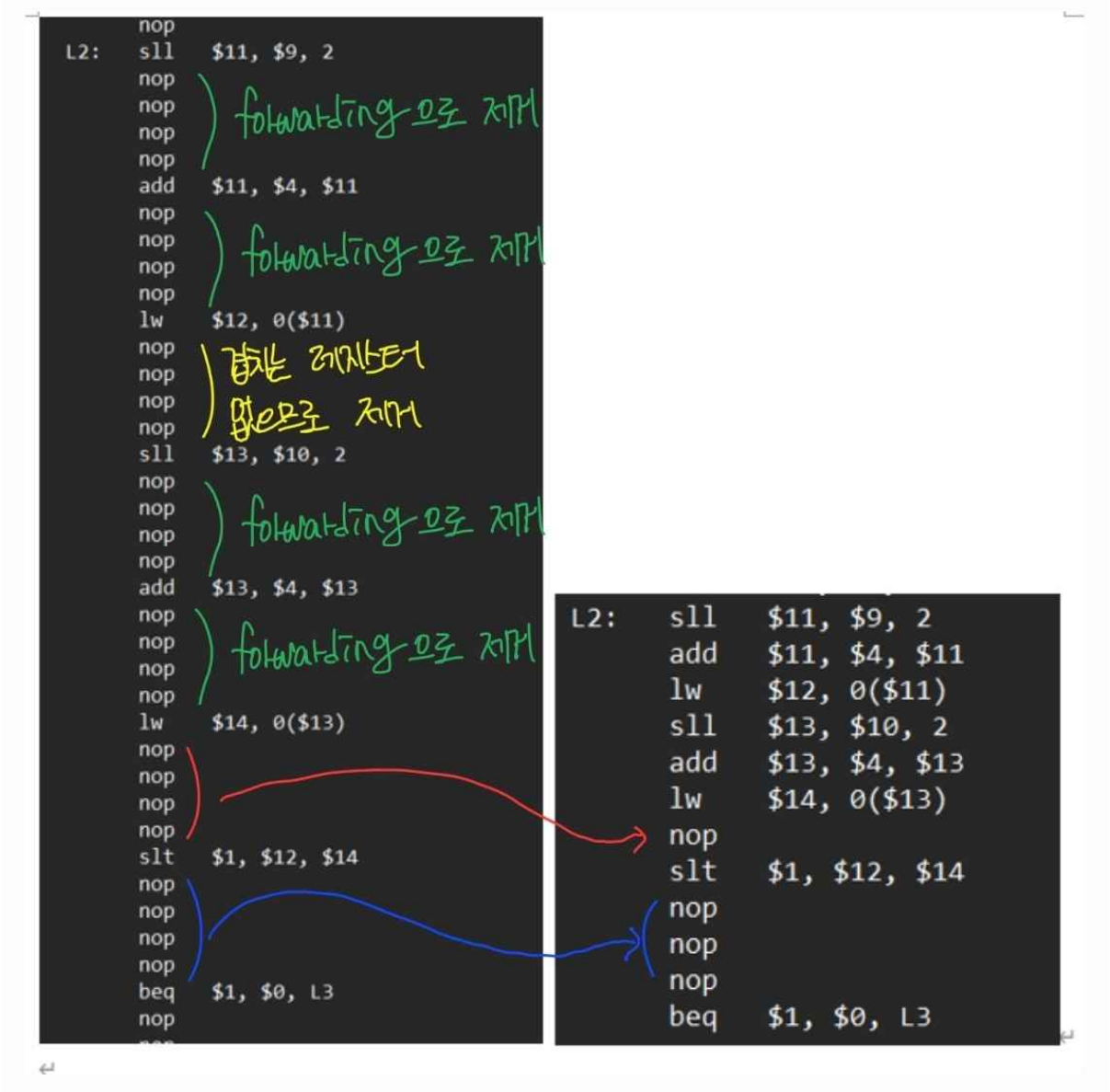
첫번째 NOP를 제거하는 부분에서 lui명령어가 $4레지스터에 값을 아직 저장하지 않았는데 다음 명령어 ori가 $4레지스터의 값을 읽어올 때 data hazard가 발생합니다. 이때 lui 명령어의 EX단계에서 저장될 값의 결과를 미리 가져와 ori 명령어에서 계산을 clock손해 없이 진행합니다. 이 방식이 forwading입니다.

```
01_00  //  0x000  //lui   $4, 0x0000
00_00  //  0x004  //ori   $4, $4, 0x2000
00_00  //  0x008  //ori   $5, $0, 0x10
00_00  //  0x00C  //addi  $8, $0, 0x1
00_00  //  0x010
```

ALU의 결과 값을 가져왔으므로 signal신호로 input A부분에 01을 넣어준 모습입니다.
그리고 다음 초록색 두 부분의 명령어는 겹치는 레지스터를 사용하지 않아 모두 NOP를 제거했습니다.
빨간색 화살표와 파란색 화살표를 살펴보면 위에서 단순히 NOP를 제거했을 경우와 같은 이유로 NOP를 제거할 수 있으면 이때 forwarding은 사용되지 않습니다. 마지막으로 add 명령어부터 보겠습니다. Add 명령어에서 사용하는 $9레지스터를 sll 명령어에서 사용하려 하기 때문에 이때 data hazard가 발생합니다. 이를 해결하기 위해서 write back할 값을 미리 가져와 forwarding을 통해 NOP를 제거할 수 있습니다.

```
00_00  //  0x024  //add   $9, $0, $8
00_10  //  0x028  //addi  $10, $8, -1
```

write back할 값을 가져왔으므로 Input B의 signal신호로 10을 넣어준 모습입니다.



[기존 코드] [forward 제어신호 추가 후 코드]

먼저 sll명령어와 add명령어 사이에서 $11레지스터에 있어서 data hazard가 발생하는 모습을 확인할 수 있는데 이는 이전 명령어의 EX단계의 결과 값을 다음 명령어의 ALU의 input값으로 가져오는 forwarding을 통해 해결할 수 있습니다. Input B에서 $11레지스터를 사용하고 있으므로 Input B의 signal 신호로 01을 넣어준 모습입니다.

```
00_01  //  0x02C  //sll   $11, $9, 2
```

다음 명령어에서는 input A의 signal 신호로 01을 넣어주어 forwarding을 이용해 NOP를 제거했습니다.

```
01_00  //  0x030  //add   $11, $4, $11
```

다음 노란색 부분에서는 명령어 사이에 겹치는 레지스터가 없으므로 NOP를 모두 제거해주는 모습입니다.

다음 초록색 부분에서는 $13레지스터에 있어서 data hazard가 발생하는 모습을 확인할 수 있는데 이전 명령어의 EX단계의 결과 값을 다음 명령어의 ALU의 input값으로 가져오는 forwarding을 통해 해결할 수 있습니다. Input B에서 $13레지스터를 사용하고 있으므로 signal 신호로 01을 넣어줬습니다.

```
00_01  //  0x038  //sll   $13, $10, 2
```

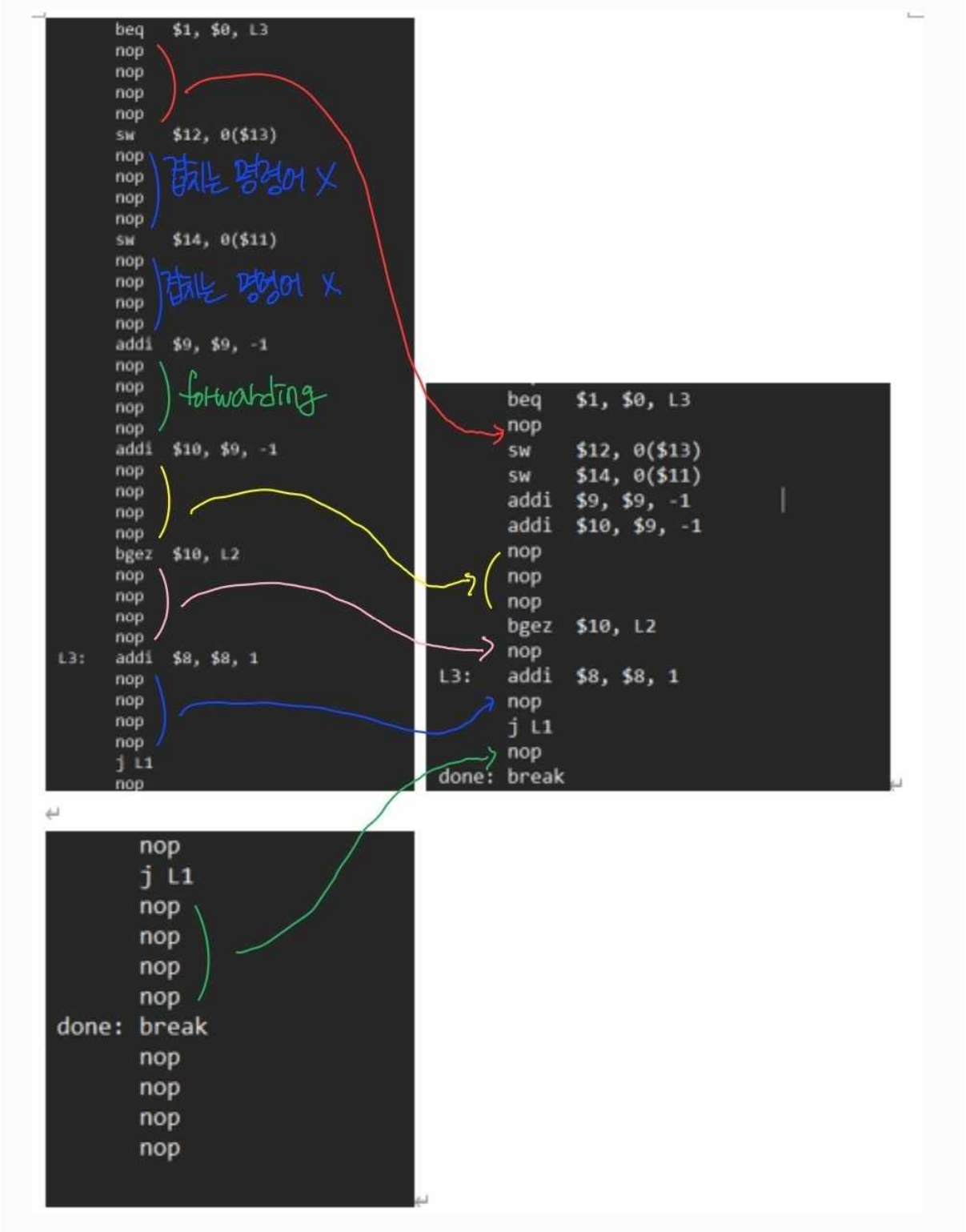
그 다음 초록색 부분에서는 또 $13레지스터에 있어서 data hazard가 발생하는 모습을 확인할 수 있는데 이전 명령어의 EX단계의 결과 값을 다음 명령어의 ALU의 input값으로 가져오는 forwarding을 통해 해결할 수 있습니다. Input A에서 $13레지스터를 사용하고 있으므로 signal 신호로 01을 넣어줬습니다.

```
01_00  //  0x03C  //add   $13, $4, $13
```

다음으로 빨간색 부분에서는 $14에 있어서 data hazard가 발생합니다. load명령어 뒤에 NOP를 하나 넣어주고 forwarding을 진행하면 load명령어에서 write back하는 값을 다음 명령어의 EX단계에서 가져올 수 있습니다. 이를 통해 data hazard를 해결할 수 있습니다.
Input B값으로 signal 신호 10을 넣어주었습니다.

```
00_00  //  0x040  //lw    $14, 0($13)
00_10  //  0x044
```

마지막으로 파란색 부분에서는 위에서 NOP만을 제거하는 방식을 이용했을 때와 같은 방식으로 기존의 코드에서 NOP를 하나 제거해주었습니다.



[기존 코드] [forward 제어신호 추가 후 코드]

먼저 첫번째 빨간색 화살표에서는 위에서 NOP를 제거하는 방식과 동일하며 이때 forwarding은 사용하지 않습니다. NOP를 하나로 줄여줬습니다.

다음 파란색 부분에서는 명령어 사이에 서로 겹치는 명령어가 없으므로 NOP를 모두 제거해준 모습입니다.

다음 초록색 부분에서는 $9레지스터에 있어서 data hazard가 발생합니다. 이때 이전 명령어의 EX단계의 결과 값을 다음 명령어의 ALU의 input값으로 가져오는 forwarding을 통해 해결할 수 있습니다. Input A에서 $9레지스터를 사용하고 있으므로 signal 신호로 01을 넣어줬습니다

```
01_00  //  0x068  //addi  $9, $9, -1
```

다음으로 노란색 부분과 핑크색 부분 그리고 파란색 화살표와 초록색 화살표 부부에서는 위에서 단순히 NOP만을 제거했을 경우와 같은 방식으로 NOP를 제거했으며 이때 forwarding은 사용하지 않습니다. 마지막의 break문 이후의 NOP는 모두 제거했습니다.

■ 명령 수행에 걸린 총 cycle수

```
-------------------------------------------
| H020-3-1647-01: Computer Architecture    |
|                 CE.KW.AC.KR              |
-------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-------------------------------------------
 Break signal: 1,   # of Cycles:       4185
-------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 41965000 (1ps)
```

먼저 기본 어셈블리 코드의 cycle수 입니다.

1. 기존의 어셈블리 코드에서 nop를 제거한 경우

```
-------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-------------------------------------------
 Break signal: 1,   # of Cycles:       2608
-------------------------------------------
```

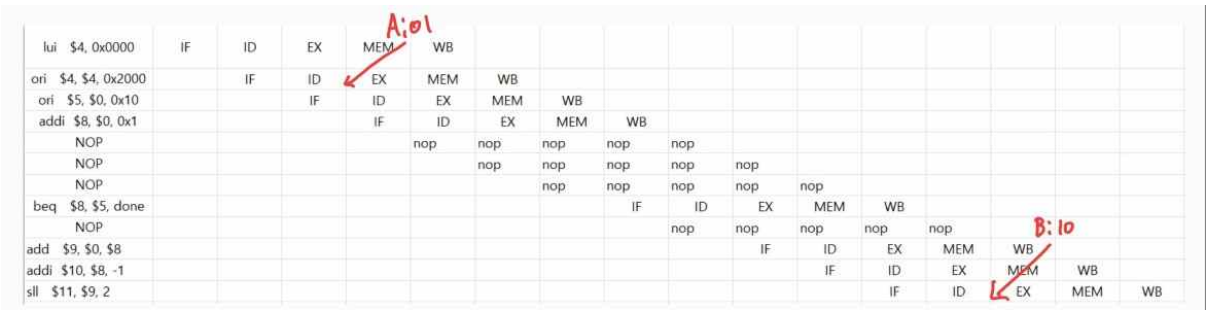
기존의 어셈블리 코드 보다 NOP제거를 통해 cycle수가 감소한 것을 확인할 수 있습니다.

2. 기존의 어셈블리 코드에서 forward제어신호를 추가한 경우

```
FST info: dumpfile tb_PC.vcd opened for output.
-----------------------------------------
 Break signal: 1,    # of Cycles:        1486
-----------------------------------------
```

다음은 기존의 어셈블리 코드에서 NOP를 제거한 경우와 forward제어신호를 추가한 경우의 총 cycle수입니다.
단순히 NOP를 제거한 경우 cycle값은 2608이며 forward제어신호를 추가해 NO를 제거한 경우에는 1486의 cycle값을 가지고 있는 것을 확인할 수 있습니다.
forward제어신호를 추가해 NOP를 제거한 경우에 cycle수가 훨씬 더 감소한 것을 확인할 수 있습니다.

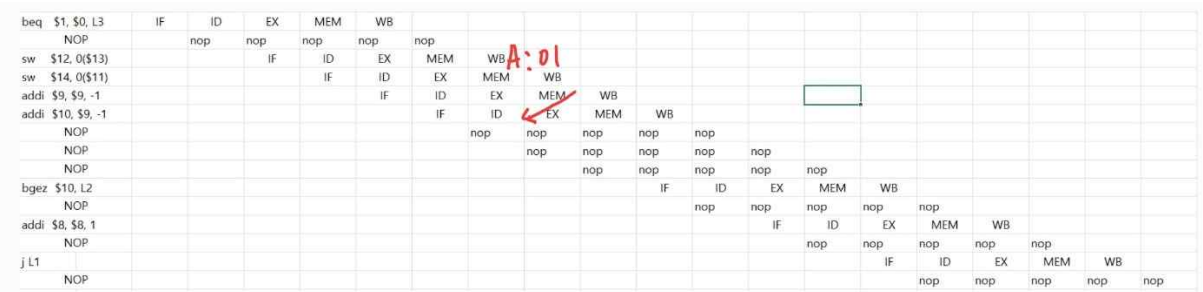
- **제시된 insertion sort의 dependency들을 코드가 적힌 종이에 표시 및 설명**



lui에서 $4에 저장된 값을 write back하기 전 forwarding을 통해 ori의 EX단계로 가져온 모습입니다. 이때 ori에서 $4는 input A이므로 A의 signal을 01로 해주었습니다.

add에서 $9에 write back한 값을 sll에서 input B의 값으로 forwarding을 통해 사용하는 모습입니다. B의 signal을 10으로 설정했습니다.

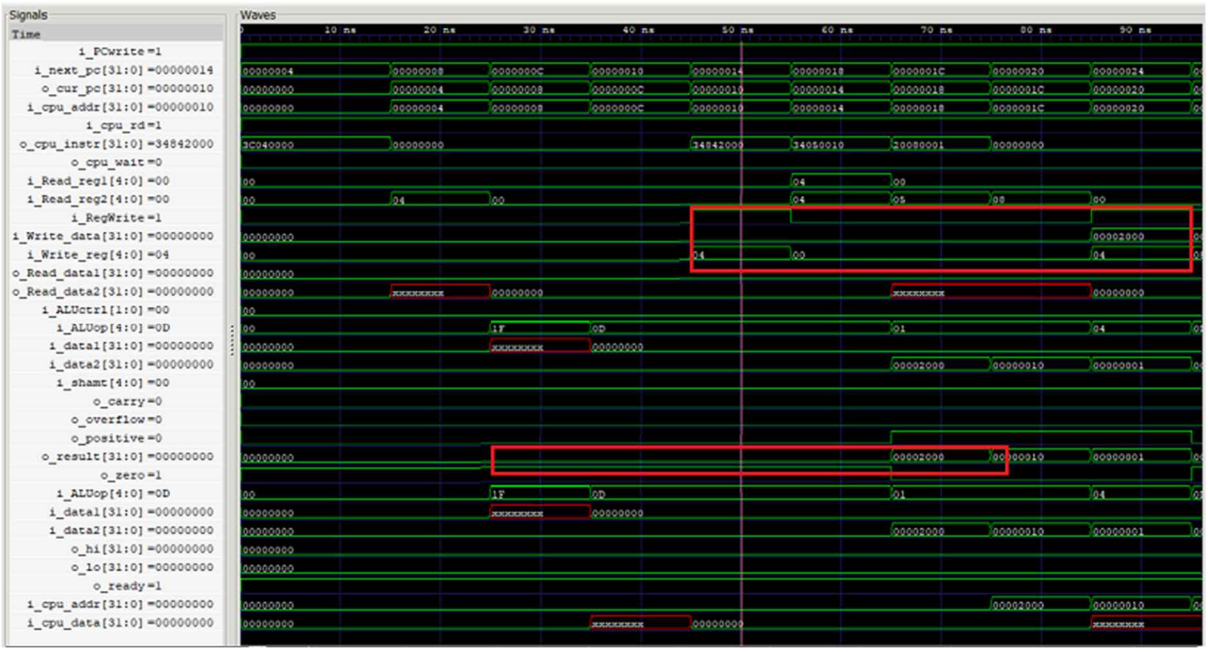
sll에서 $11의 값은 write back하기 전에 계산되는데 이때 forwarding을 통해 add의 EX단계에서 $11의 값을 사용합니다. 이는 add에서 input B의 값이므로 B의 signal을 01로 설정했습니다. 다음 명령어인 add와 lw사이에서도 같은 이유로 $11의 값을 forwarding합니다. 이때는 input A의 값으로 사용하고 있으므로 signal을 01로 설정했습니다.
그 다음 sll와 add사이에서는 $13에 대해서 같은 상황이 발생하는데 이때 또한 forwarding을 통해 값을 가져왔으며 input B에서 사용 중이므로 B의 signal을 01로 설정했습니다.
다음 명령어인 add와 lw사이에서도 같은 이유로 $13의 값을 forwarding합니다. 이때는 input A의 값으로 사용하고 있으므로 signal을 01로 설정했습니다.
마지막으로 add에서 $14에 write back한 값을 slt에서 input B의 값으로 forwarding을 통해 사용하는 모습입니다. B의 signal을 10으로 설정했습니다.



마지막 forwarding은 두개의 addi명령어 사이에서 진행합니다. 첫번째 addi에서 $9에 저장된 값을 write back하기 전 forwarding을 통해 두번째 addi의 EX단계로 가져온 모습입니다. 이때 두번째 addi에서 $9는 input A이므로 A의 signal을 01로 해주었습니다.

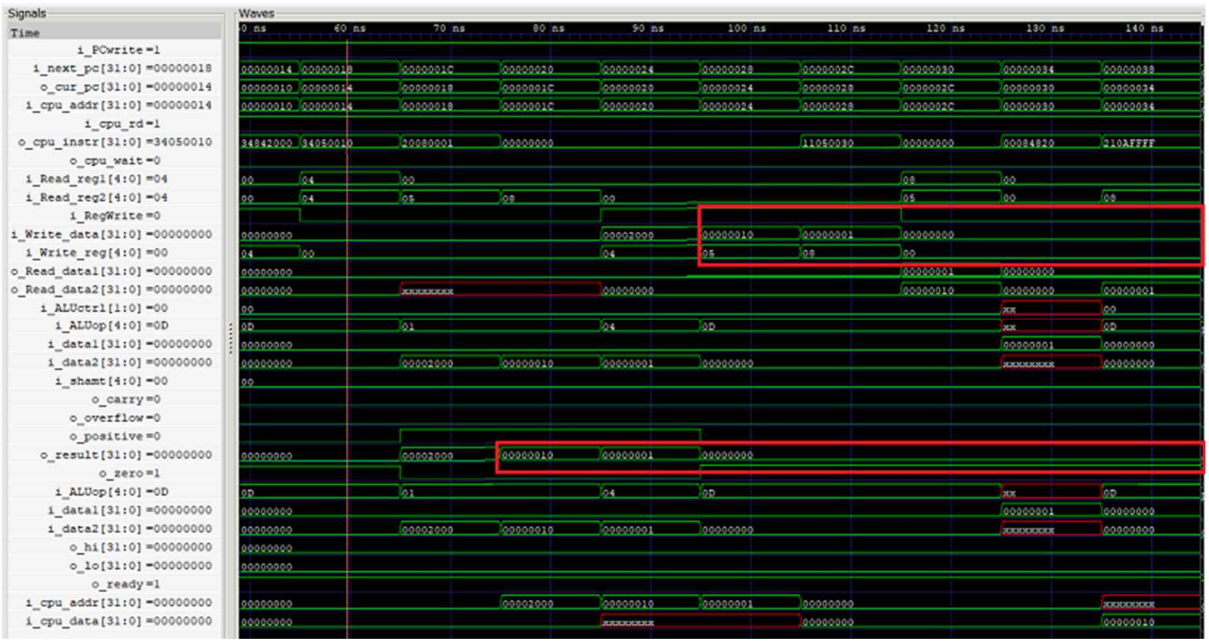
■ insertion sort의 2가지 시뮬레이션 진행 및 설명

다음은 기존의 코드에서 NOP를 제거한 경우의 시뮬레이션 결과입니다.



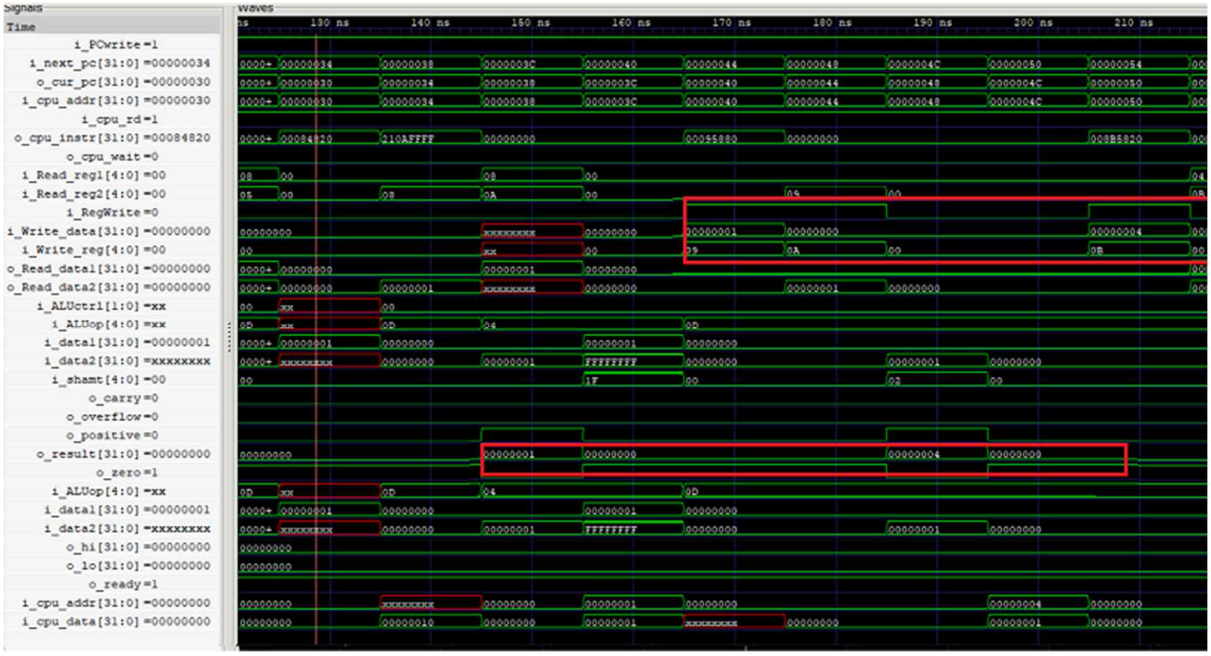

```
main: lui   $4, 0x0000
      nop
      nop
      nop
      ori   $4, $4, 0x2000
```

위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. lui명령어와 ori명령어의 EX단계에서의 결과 값이 result에 잘 출력되는 모습을 확인할 수 있습니다.
그리고 WB단계에서 이 값을 해당하는 register에 잘 저장하는 모습 또한 확인할 수 있습니다.




```
      ori   $5, $0, 0x10
      addi  $8, $0, 0x1
      nop
      nop
      nop
L1:   beq   $8, $5, done
```

위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. NOP가 시작하는 명령어에서는 이전 명령어의 해당하는 단계만 진행되고 NOP는 아무 작업도 하지 않는 모습을 확인할 수 있으며 ori, addi, beq명령어 모두 값이 잘 들어가며 제대로 수행되고 있는 모습을 확인할 수 있습니다. beq명령어에서 분기는 일어나지 않는 모습도 볼 수 있습니다.

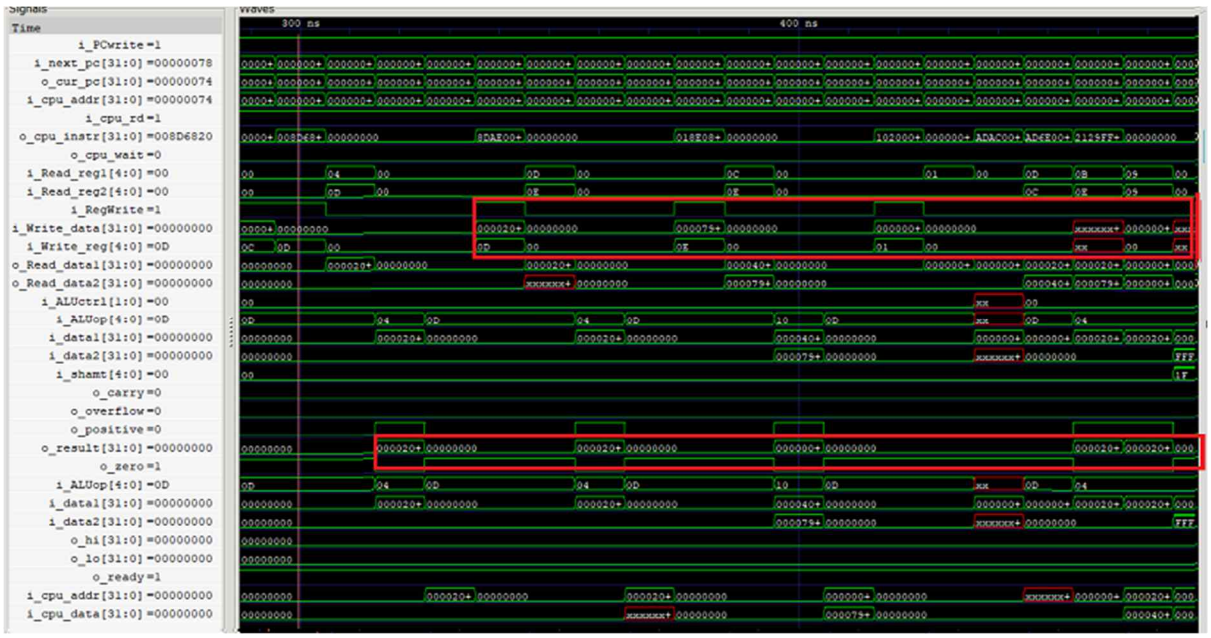



```
        nop
        add   $9, $0, $8
        addi  $10, $8, -1
        nop
        nop
L2:     sll   $11, $9, 2
        nop
        nop
        nop
```

위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. NOP에서는 이전 명령어를 제외하고 아무 작업도 하지 않으면 add, addi,sll명령어 모두 EX단계에서 값을 잘 처리하고 WB단계에서 잘 저장하는 모습을 확인할 수 있습니다. 모두 제대로 수행되는 모습입니다.

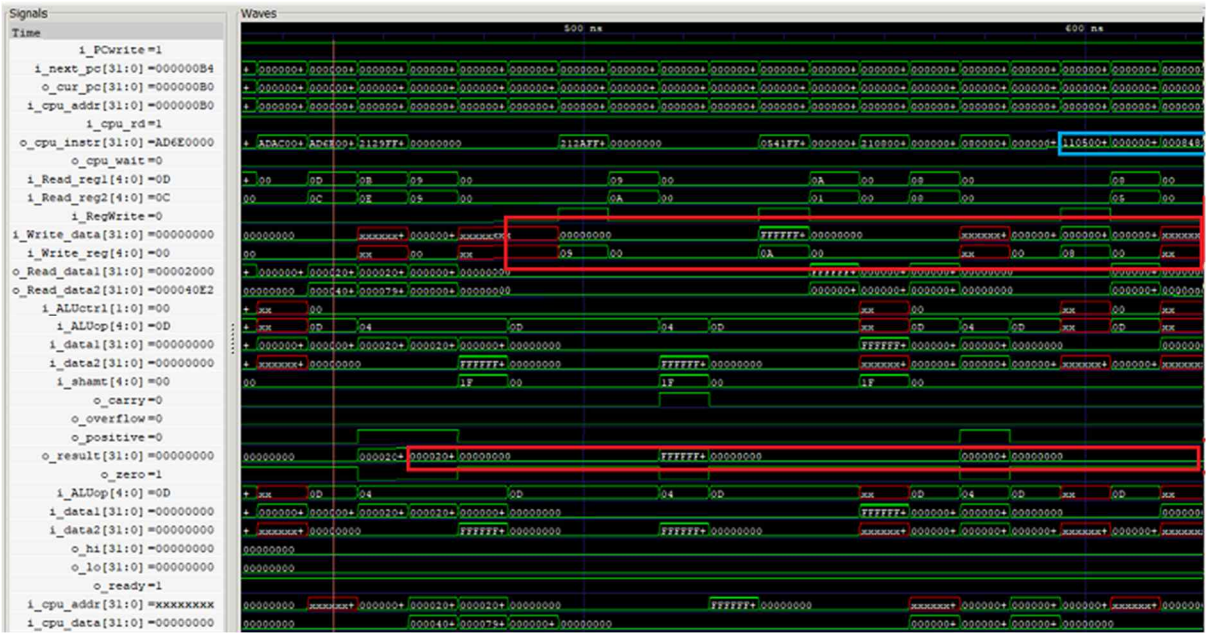

```
add   $11, $4, $11
nop
nop
nop
lw    $12, 0($11)
sll   $13, $10, 2
nop
nop
nop
```

위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. 모든 명령어 모두 값이 잘 출력되고 저장되는 모습을 확인할 수 있습니다.



```
    add   $13, $4, $13
    nop
    nop
    nop
    lw    $14, 0($13)
    nop
    nop
    nop
    slt   $1, $12, $14
    nop
    nop
    nop
    beq   $1, $0, L3
    nop
    sw    $12, 0($13)
```

위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. 명령어들이 제대로 수행되는 모습을 볼 수 있으며 beq명령어에서는 분기하지 않고 다음 명령어를 수행하는 모습 또한 확인할 수 있습니다.



```
      sw    $14, 0($11)
      addi  $9, $9, -1
      nop
      nop
      nop
      addi  $10, $9, -1
      nop
      nop
      nop
      bgez  $10, L2
      nop
L3:   addi  $8, $8, 1
      nop
      j L1
      nop
done: break
```

위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. 모든 명령어들이 잘 수행되며 L3에서 NOP이후 j명령어에서 L1로 NOP이후에 jump하는 모습을 볼 수 있습니다. 이후 L1에서 beq명령어를 fetch에 이후 과정을 진행하는 모습 또한 확인할 수 있습니다.

이후 프로그램은 2608 cycle동안 진행되고 break하는 모습을 확인할 수 있습니다.

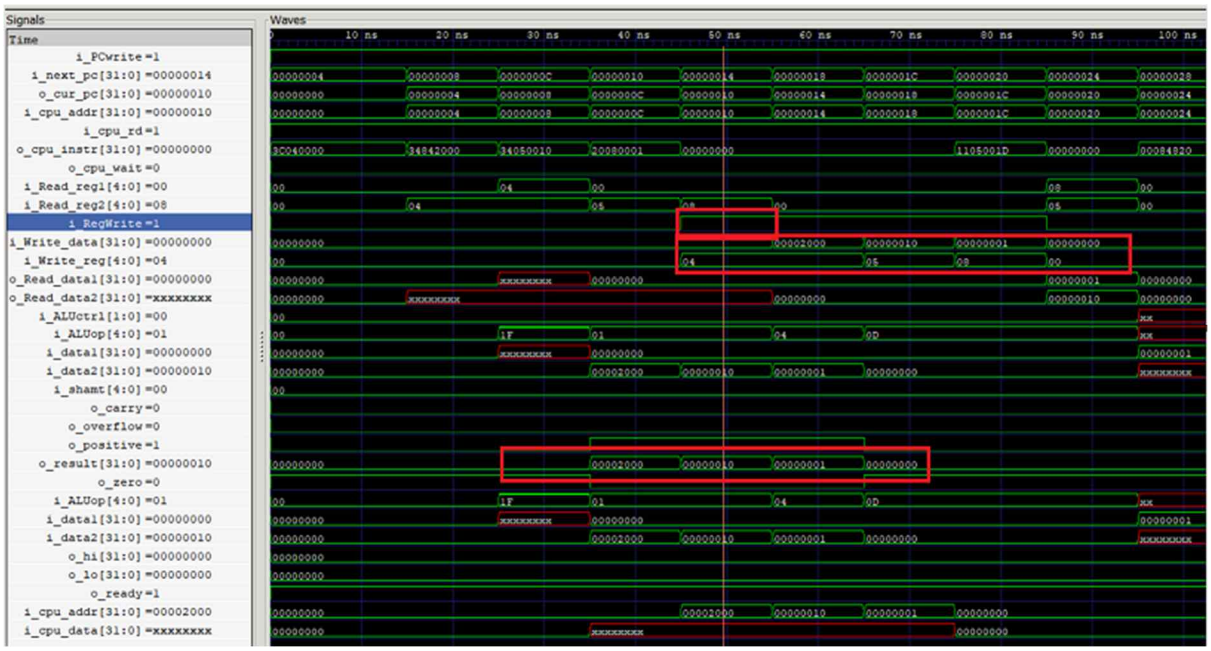
다음은 기존의 코드에서 forward 제어신호를 추가하여 더 많은 NOP를 제거한 경우의 시뮬레이션 결과입니다.

```
-----------------------------------------------
| 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\PC\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\PC\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.
```

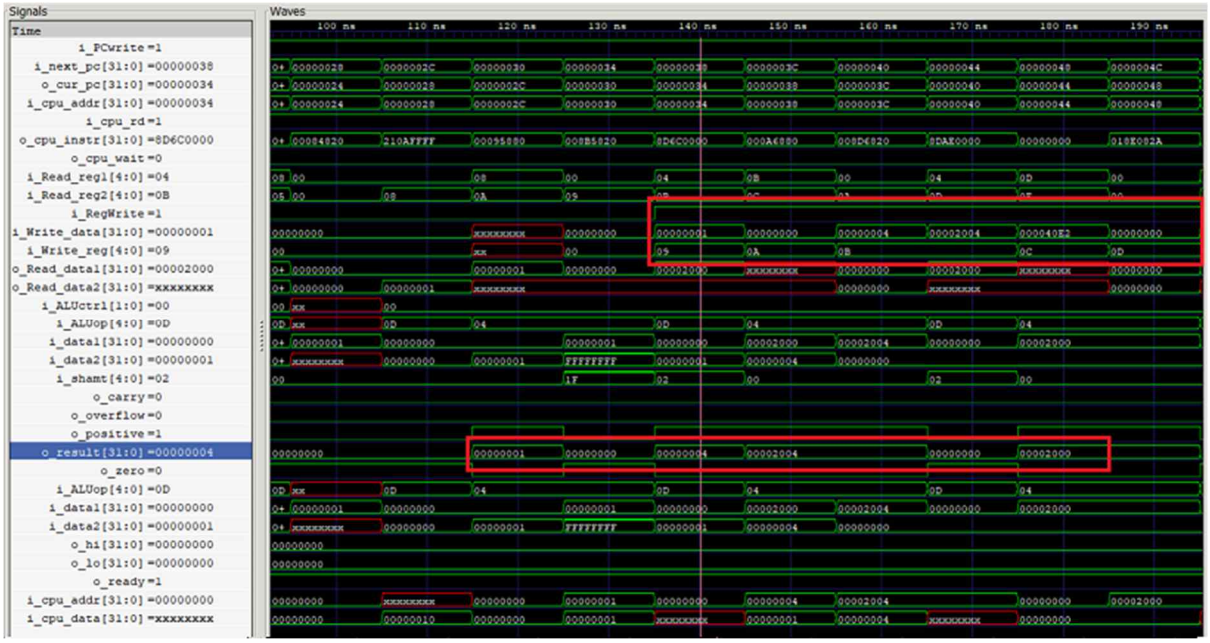
먼저 총 1486의 cycle이 소요된 것을 확인할 수 있고 파일들 모두 다른 점 없이 오류 발생하지 않고 잘 실행되는 모습을 확인할 수 있습니다.




```
main: lui  $4, 0x0000
      ori  $4, $4, 0x2000
      ori  $5, $0, 0x10
      addi $8, $0, 0x1
      nop
      nop
      nop
L1:   beq  $8, $5, done
      nop
```

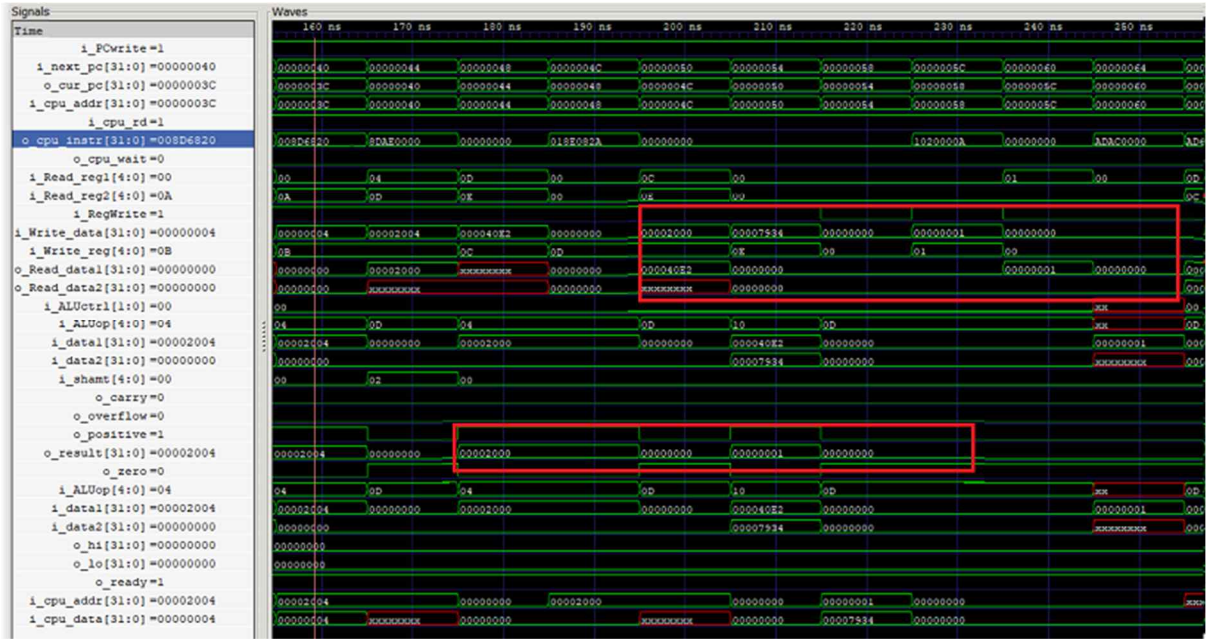
위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. 명령어와 레지스터를 fetch를 하는 모습을 read_reg부분과 cpu_instr에서 확인할 수 있고 result값을

통해 EX단계의 결과 값을 확인할 수 있습니다. 첫 번째 명령어의 WB단계부터 reg_write값이 1이 되어 레지스터에 result 값을 저장하는 모습을 확인할 수 있습니다. NOP가 나오기 전까지 값을 저장합니다. beq명령어에서 $8과 $5의 값이 서로 달라 분기하지 않는 모습을 확인할 수 있습니다.




```
    add   $9, $0, $8
    addi  $10, $8, -1
L2: sll   $11, $9, 2
    add   $11, $4, $11
    lw    $12, 0($11)
    sll   $13, $10, 2
```

위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. 위에서 확인했듯이 명령어와 레지스터를 fetch를 하는 모습을 read_reg부분과 cpu_instr에서 확인할 수 있고 result값을 통해 EX단계의 결과 값을 확인할 수 있습니다. 첫 번째 명령어의 WB단계부터 reg_write값이 1이 되어 레지스터에 result 값을 저장하는 모습을 확인할 수 있습니다. result값이 모두 잘 들어가고 코드에서 구현한 레지스터에 결과값이 잘 저장되는 모습을 확인할 수 있습니다.

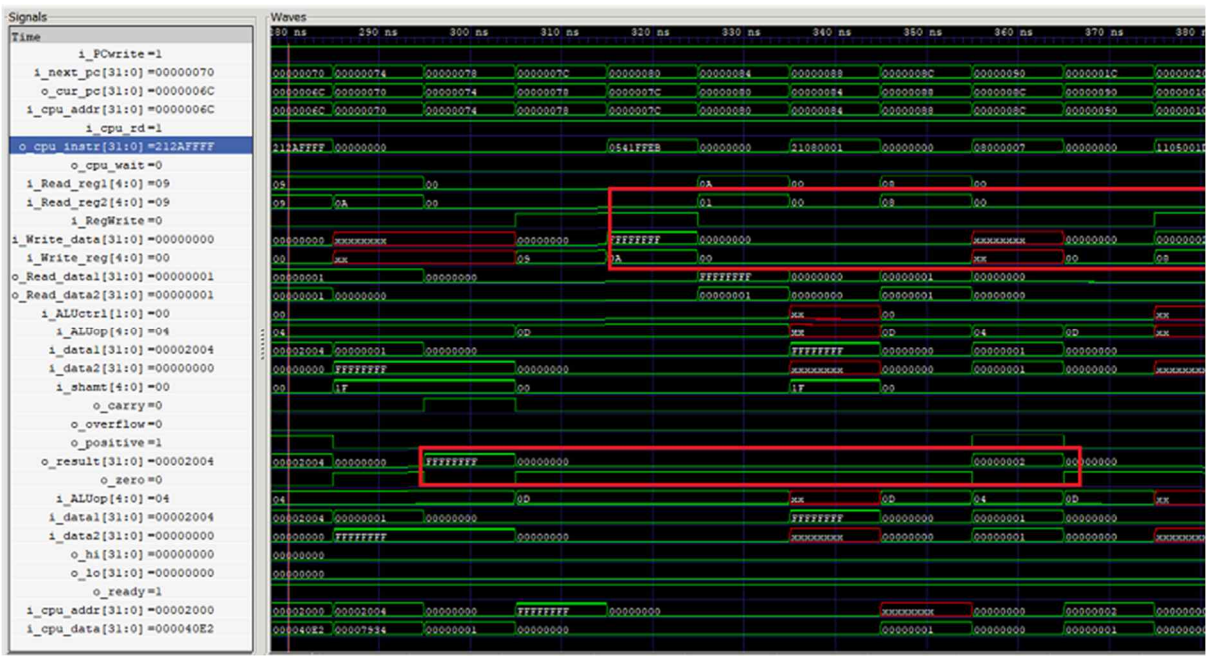



```
add   $13, $4, $13
lw    $14, 0($13)
nop
slt   $1, $12, $14
nop
nop
nop
```

위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. 이 부분 또한 result값이 모두 잘 들어가고 코드에서 구현한 레지스터에 결과값이 잘 저장되는 모습을 확인할 수 있습니다. 그리고 NOP구간에서는 이전 명령어의 단계에 해당하는 작업만 진행하고 NOP에 해당하는 단계에서는 아무 작업도 하지 않는 모습을 볼 수 있습니다.

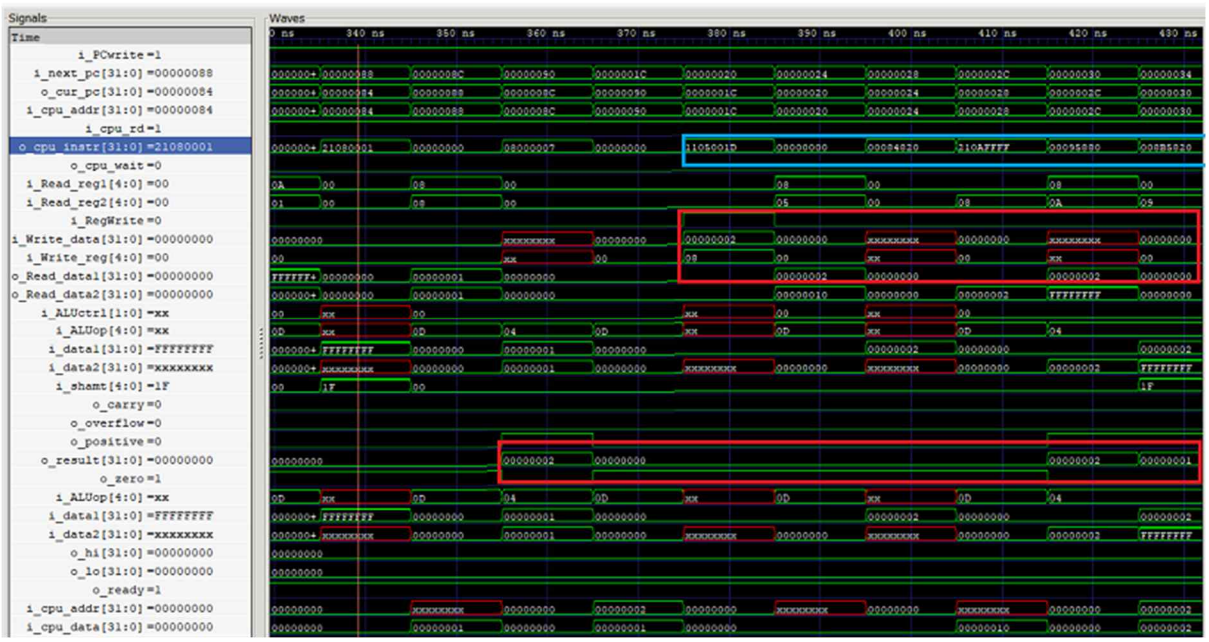

```
beq   $1, $0, L3
nop
sw    $12, 0($13)
sw    $14, 0($11)
addi  $9, $9, -1
```

위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. 첫번째 명령어 beq에서 $1의 값과 $0의 값이 서로 달라 분기하지 않는 모습을 확인할 수 있으며 다음 store명령어에서는 register에 write하지 않는 모습을 확인할 수 있습니다.



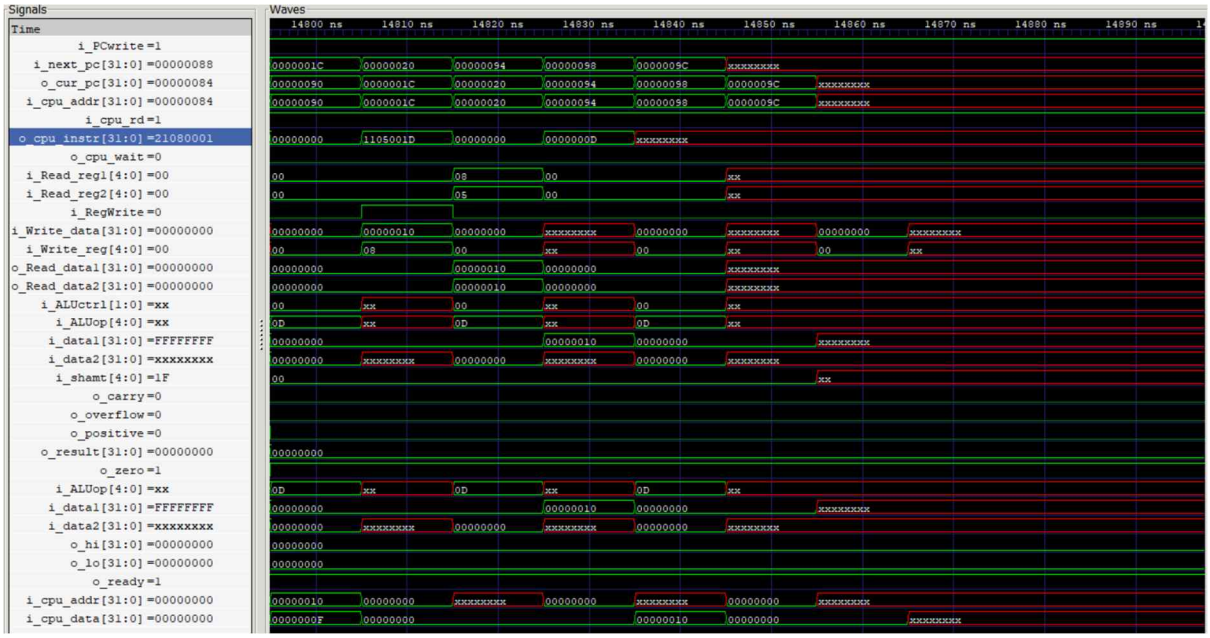
```
    addi  $10, $9, -1
    nop
    nop
    nop
    bgez  $10, L2
    nop
L3: addi  $8, $8, 1
```

위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. 일단 EX단계에서 result값을 잘 계산하고 WB단계에서 레지스터에 잘 저장하는 모습을 확인할 수 있으며 NOP단계에서는 이전 명령어의 작업을 제외하고는 아무 작업도 하지 않는 모습을 볼 수 있습니다. 그리고 bqez명령어에서는 분기하지 않고 다음 명령어가 수행되는 모습을 확인할 수 있습니다.




```
L3: addi  $8, $8, 1
    nop
    j L1
    nop
```

마지막으로 위 코드를 진행한 시뮬레이션 결과를 살펴보겠습니다. addi명령어가 각 단계에 따라 잘 진행되고 NOP이후 J명령어에서 L1으로 jump하는 모습을 확인할 수 있습니다. 마지막 NOP이후에 L1에 있는 beq명령어의 instruction을 fetch하는 모습을 확인할 수 있습니다.



이후 프로그램은 1486 cycle동안 진행되고 break하는 모습을 확인할 수 있습니다.

■ 프로젝트 내용 전체 정리 및 고찰

프로젝트의 전체 내용은 크게 두가지로 나누어집니다. 기존의 어셈블리 코드에서 NOP를 제거한 경우와 forwarding방식을 이용해 더 많은 NOP를 제거하는 경우에 대해 구현하고 시뮬레이션 결과를 분석하는 것입니다. 이를 진행하기 위해서는 mars에서 코드를 다시 구현하고 이를 M_TEXT_SEG파일에 덮어써서 gtkwave를 실행해 결과를 확인하면 됩니다. Forwarding 방식을 진행하기 위해서는 똑같이 코드를 구현하고 이를 위 파일에 덮어쓴 다음 M_TEXT_FWD 파일에 signal신호를 추가한 후 gtkwave를 통해 결과를 확인하면 됩니다.
프로젝트를 진행하는 하기 위해서는 우선 Pipeline에 대한 완벽한 이해와 명령어들이 수행되는 단계를 확실히 이해해야 했습니다. 전반적인 내용이 수업 및 실습 시간에 배운 내용이었으며 심지어 컴퓨터 구조 HW과제에서도 공부한 내용이었습니다. 하지만 직접 코드에 대해서 NOP를 추가 및 제거하고 forwarding도 직접 수행해보니까 처음에는 살짝 헷갈렸습니다. 일단 forwarding방식에 대해 살짝 헷갈려 중점적으로 공부했습니다.
Forwarding을 통해 데이터 위험을 해결하기 위해서는 데이터 의존성을 정확하게 파악하고 필요한 데이터를 올바르게 전달해야 합니다. 이를 위해 파이프라인의 각 단계에서 데이터 의존성을 분석하고 필요한 경우 Forwarding 경로를 설정해야

합니다. 또한, Forwarding을 적용할 때에는 적절한 회로 설계와 제어 신호 처리가 필요하며, 명령어의 종류와 파이프라인 구조에 따라 구현 방법이 달라질 수 있다는 점을 알았습니다. 데이터 위험을 제거하고 NOP를 제거하는 데에는 파이프라인 구조의 분석과 설계, 적절한 Forwarding 기법의 필요하다는 점과 이를 통해 파이프라인의 성능을 최적화하고 data hazard를 효과적으로 제거할 수 있다는 점 또한 알게 되었습니다.
그 후 프로젝트를 진행하면서 처음에는 NOP를 얼마나 제거해야 하는지, Hazard가 생기지 않게 Cycle을 줄이는 것이 고민이었습니다. 하지만 초반 명령어들의 동작을 시뮬레이터를 사용하여 확인하고 나서는, 레지스터가 겹치는지 여부를 고려해주어 쉽게 해결할 수 있었습니다. 일단 NOP를 제거하는 과정에서 NOP를 제거하고 시뮬레이션을 통해 결과를 확인하려 했는데 계속 오류가 나는 상황이 발생했습니다. 알고 보니, 기존의 코드에 대해 오타가 있어서 결과가 제대로 나오지 않았던 것이었습니다. 오타를 발견하고 나서는 결과 값을 제대로 확인할 수 있었으며 NOP제거를 통해 감소한 총 cycle수도 확인할 수 있었습니다.
두번째, forwarding방식을 이용하여 기존의 코드를 수정할 때 코드에서 forwarding을 통해 제거되는 NOP를 모두 지워주고 M_TEXT_FWD파일에서 signal을 수정하는 방식으로 진행했습니다. 하지만 이렇게 진행하니 오류는 발생하는데 어떤 명령어의 어떤 signal에서 오류가 발생하는 지에 대해서 확인할 수 없어 모두 지우고 처음부터 진행했습니다. 처음 코드에서 NOP를 명령어마다 지워 나갔고 M_TEXT_FWD파일에서 해당하는 명령어에 대한 forwarding signal을 수정했습니다. 이렇게 진행하니 어떤 signal에서 오류가 발생하는 지 확인할 수 있었고 어떤 명령어에서 NOP를 잘못 제거했는지에 대해서도 확인할 수 있었습니다.
마지막으로 hazard에 대해서 자세히 찾아보면서 수업시간에 배운 내용에 대해 다시 짚고 넘어갈 수 있어서 의미 있는 시간이었으며 과제를 진행하면서 시험공부를 병행한 것 같아 좋았습니다.

■ 문제점

먼저 코드에서 ori $5, $0, 100 이 부분에서 오타가 있어서 NOP를 맞게 제거하였는데도 오류가 계속 발생하는 경우가 있었습니다. 이 부분을 insert_sort 파일 확인 후 ori $5, $0, 0x10으로 수정하여 오류를 해결하였습니다.

다음으로 NOP를 제거하는 경우에서 발생하는 문제점입니다.

```
add     $9, $0, $8
        addi   $10, $8, -1
        nop
        nop
L2:     sll     $11, $9, 2
```

이 코드에서 add명령어와 sll명령어 사이에 $9 때문에 data hazard가 발생하는데 이를 해결하기 위해 NOP 2개를 addi명령어와 sll명령어 사이에 넣어 해결하였습니다. 하지만 좀 더 생각해봤을 때 add명령어와 addi명령어 사이에 NOP 2개를 넣어주거나 명령어 사이사이에 NOP 1개씩 넣어줘도 data hazard가 해결되어야 할 것 같은데 오류가 발생해서 이를 문제점으로 적어보았습니다.