# 컴퓨터구조
# Computer Architecture Assignment #3
# Pipeline Architecture

담당교수 : 이성원 교수님 (월3 수4)

2021202003 강준우

## Introduction

파이프라인 아키텍처에서는 이전에 멀티 프로세서에서 이어져서 이번에는 특정 단계를 겹쳐서 수행하여 효율을 높이는 과정을 수행합니다. 이때 각각의 프로세스를 겹치다보니 문제가 발생합니다. 이를 hazards라고 정의합니다.

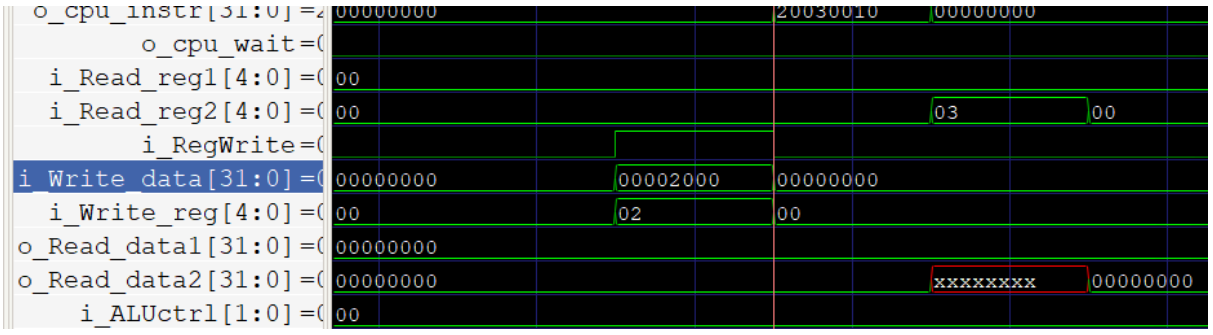
파이프라인 해저드 에서는 데이터 해저드/ 제어 해저드/ 구조 해저드가 있습니다.

데이터 해저드: 명령어간의 데이터의 의존성으로 인해 발생합니다. 즉 이후의 명령어가 이전의 명령어에서 결과값을 필요로 할 때 발생하게 됩니다.
이는 nop라는 명령어를 통해서 의도적으로 딜레이를 걸어주어 해결할 수 있습니다. 혹은 코드 스케줄링을 사용하여, 이미 준비가 완료되어있는 명령어를 먼저 수행하는 방법이 있읍니다. 또는 포워딩이라는 개념을 통해서 미리 해당 값을 받아와 다음 명령어에게 빠른 전달을 수행하여 문제를 해결 할 수 있습니다.

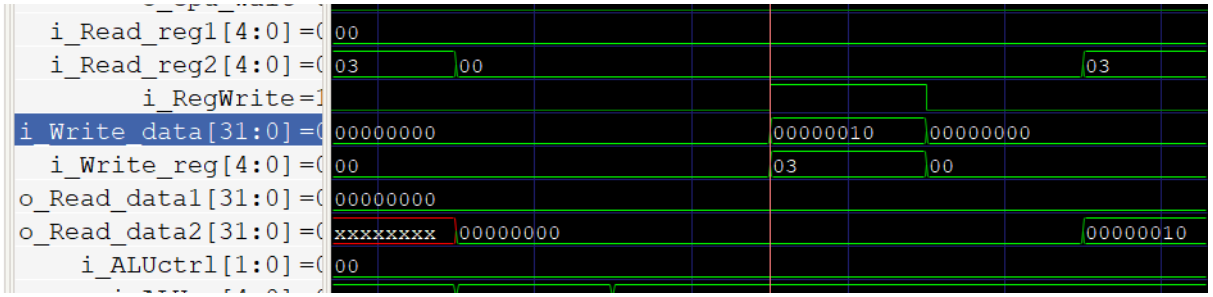
컨트롤 해저드: 명령어 중에는 조건부 분기를 통해서 결과가 바뀌는 명령어가 있습니다. 이때 조건은 명령어가 수행될 때 판단하게 됩니다. 하지만 조건을 판단할 수 없는 등의 문제로 인해서 분기의 결정을 할 수없을때 컨트롤 해저드가 발생하게 됩니다. 이때는 분기를 예측하여 수행하거나 분기할 수 있을 때까지 지연하는 방법들을 수행합니다.

구조적 해저드: 명령어에 하드웨어 자원은 한정되어 있지만 이를 다수의 명령어에서 하드웨어로 접근하여 모두에게 자원을 제공할 수 없을 때 발생하게 됩니다. 이를 해겨랗기 위해서 자원을 유하게 제공하거나, 스케줄링과 지연등을 통해서 데이터에 동시 접근률을 낮추어 해결할 수 있습니다.

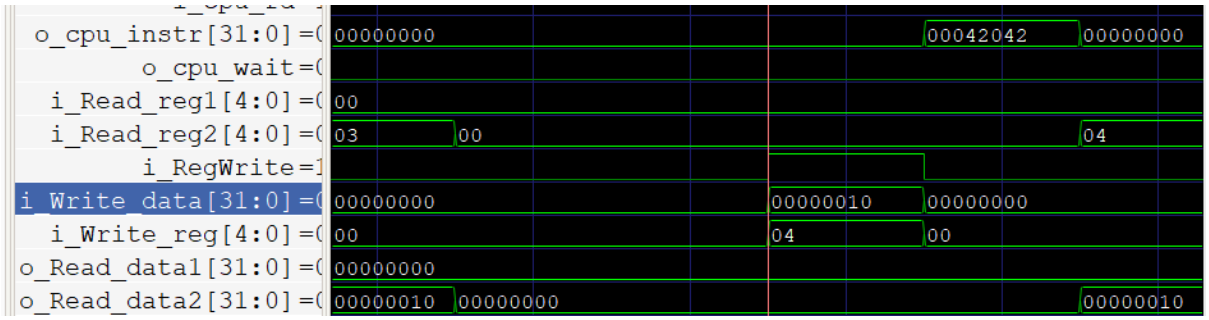
먼저 해당 명령어의 수행을 비교하기 위해서 모든 과정에서 nop를 4번씩 사용하여 웨이브폼을 확인했습니다.



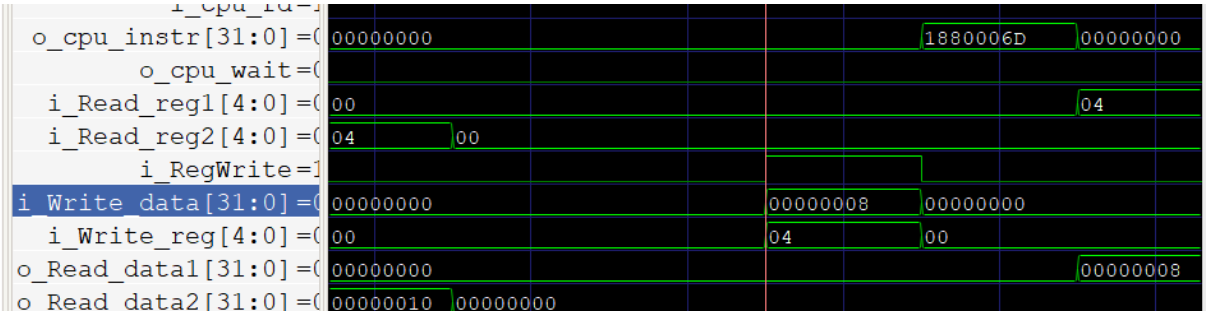

addi $2, $0, 0x2000 일때 write data에 2000확인



addi $3, $0, 16 16은 16진수로 10입니다.

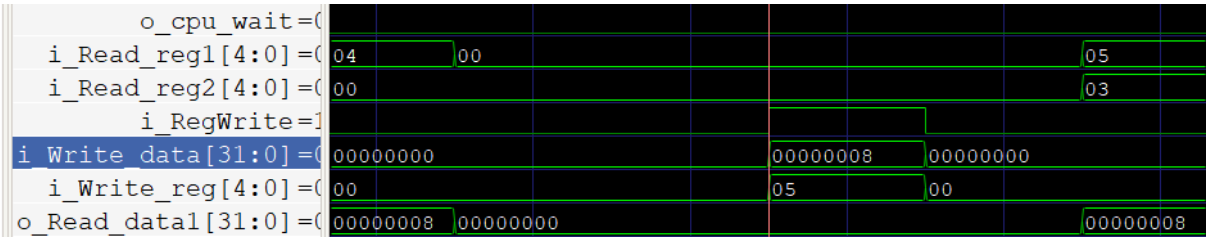



add $4, $0, $3, $4에 16 저장이어서 동일하게 10입니다.

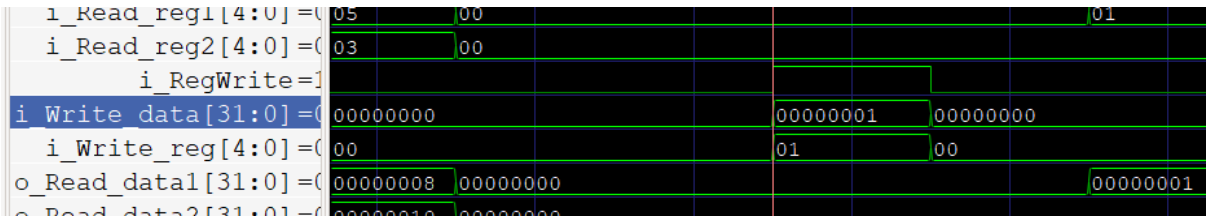



srl $4, $4, 1 ,$4 = $4 >> 1 = 8이고 루프를 절반으로 줄인 모습입니다.

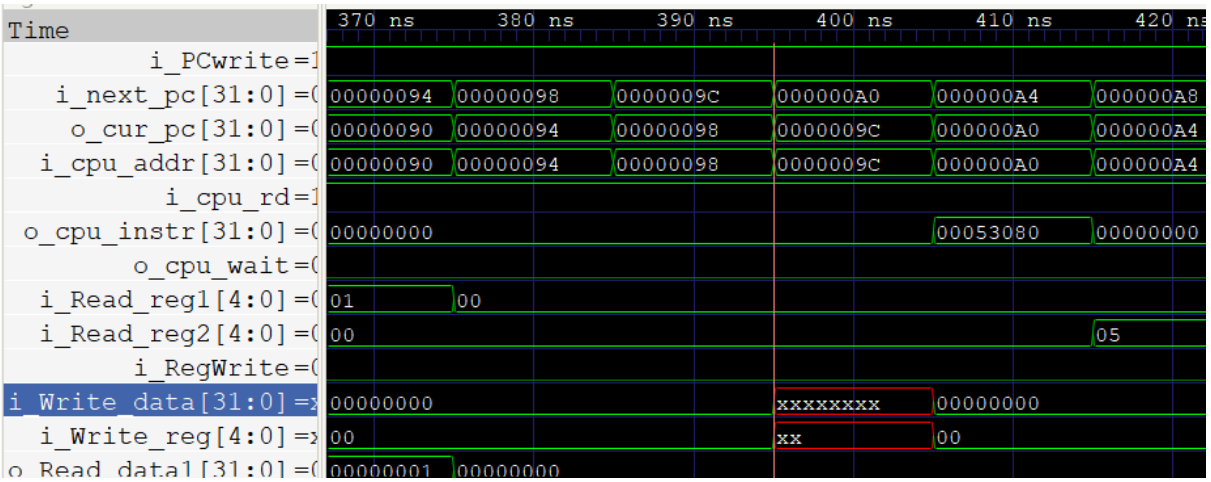

blez $4, done 조건불충족으로 점프하지 않습니다, 다음 pc값이 이어지는 모습입니다.



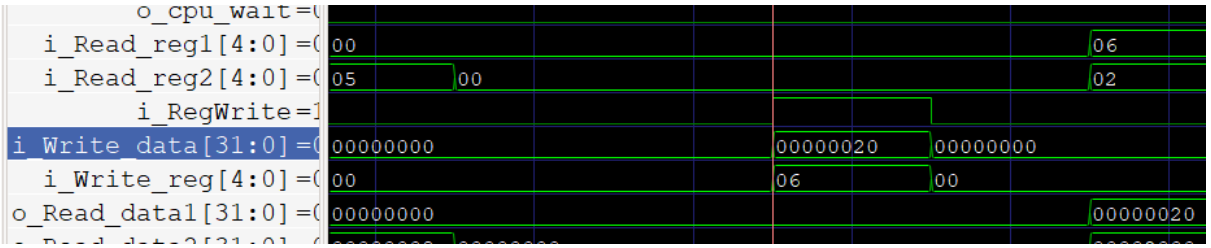

add $5, $4, $0 덧셈계산으로 8이 나옵니다. $5 = $4 = 8



slt $1, $5, $3 $5=8, $3=16 이어서 참(1)을 반환합니다.

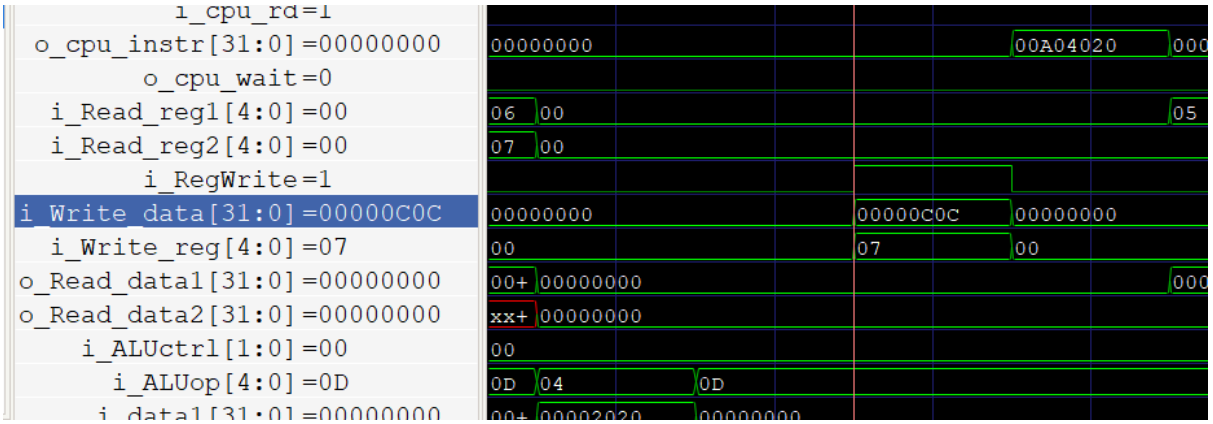



beq $1, $0, L1 $1이 1이어서 점프하지 않습니다.

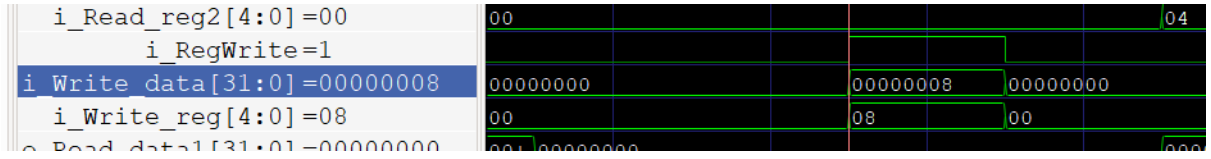



sll $6, $5, 2 2칸 왼쪽은 4를 곱하는것과 같고, 8*4를 통해서 0x20이 저장됩니다.

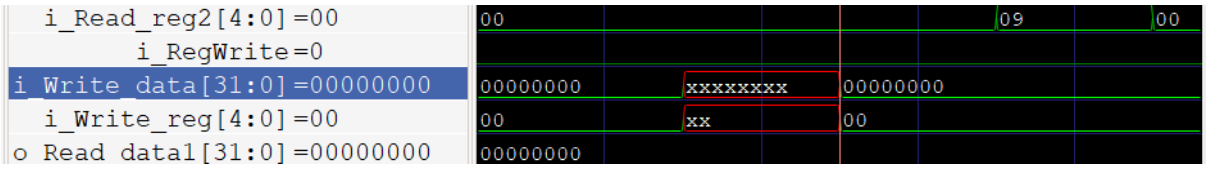

add $6, $6, $2 2000+0020 = 2020이 나옵니다.



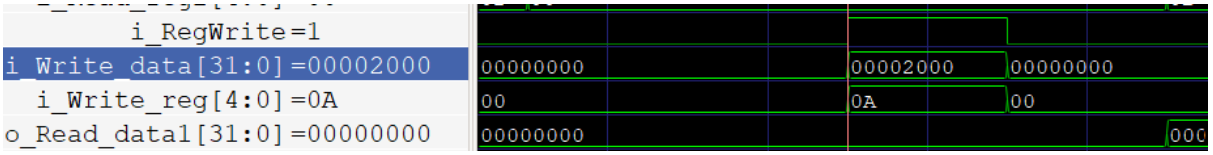

lw $7, 0($6) 2020주소로 접근합니다. 2000번주소를 기준으로 20은 32를 의미하고 32/4 = 8로 자료에 있는 LC0: 중에서 0부터 시작해서 8번인 값 3084가 7번 레지스터에 저장되고 해당 값은 c0c로 저장됩니다.



add $8, $5, $0 5번 레지스터에 값이 0x8이어서 8이 됩니다.

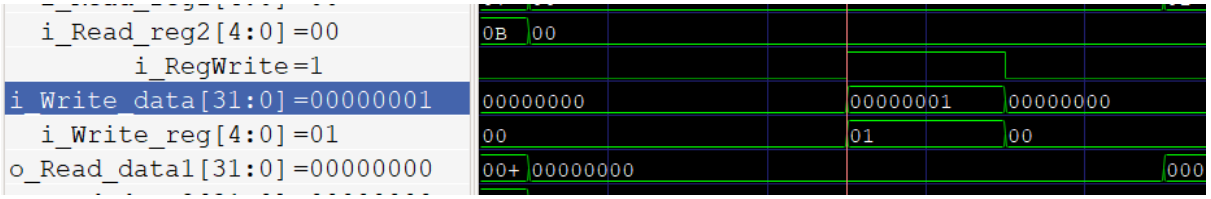



sub $9, $8, $4 둘다 값이 같아서 0으로 결과가 나오는 모습입니다.
bltz $9, L4 이전 계산에서 0으로 결과가 나와서 음수가아니므로 점프하지 않고 계속 이어집니다.
sll $10, $9, 2 또한 0을 시프트해도 0이어서 결과가 0인 모습입니다.

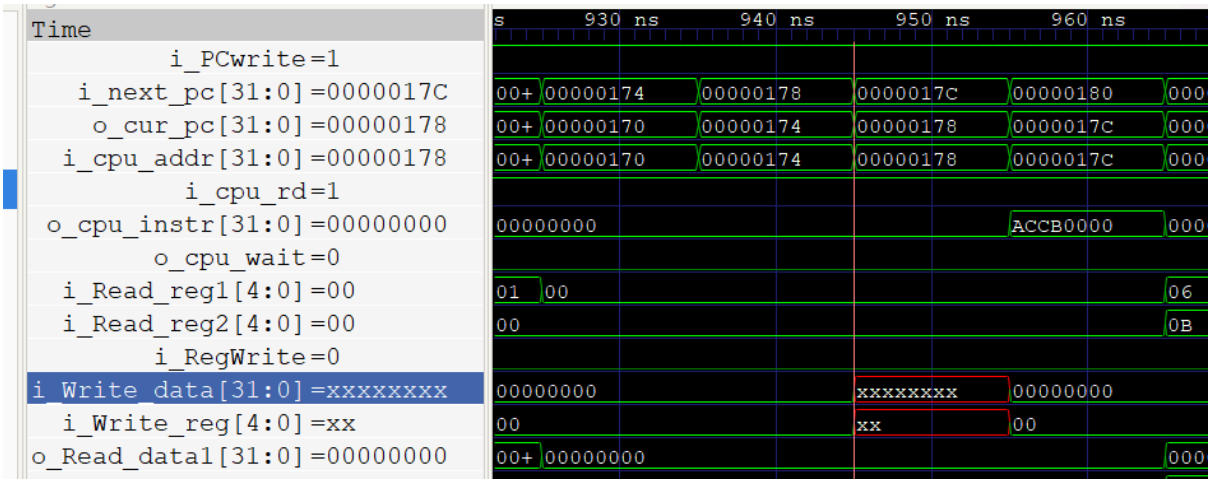



add $10, $2, $10 $2 = 0x2000, $10 = 0 이어서 2000값이 저장됩니다.

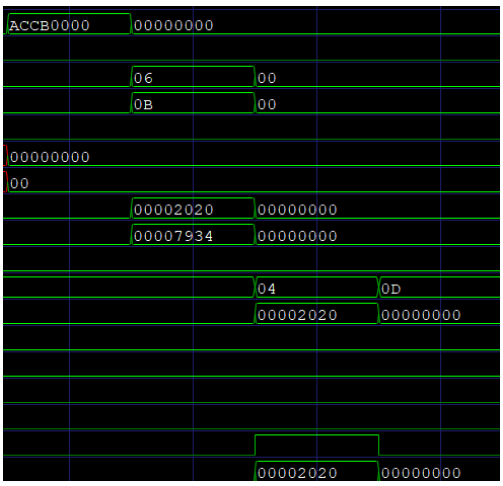

lw $11, 0($10) 이전에 10번 레지스터가 2000레지스터와 0을 더해서 2000번째 메모리에 접근하지만 해당 메모리는 .word 31028 값을 가지고 있고 이를 16진수로 변환하여 7934로 저장됩니다.




slt $1, $7, $11 $7 < $11이므로 1이 반환됩니다.



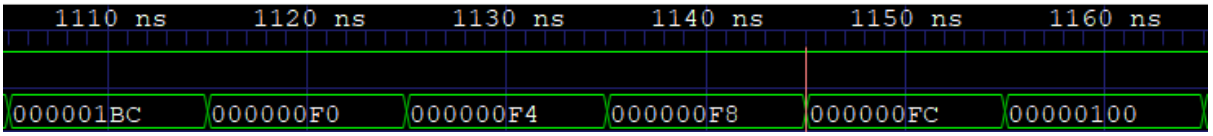

beq $1, $0, L4 1번레지스터가 값이 1이므로 점프없이 넘어갑니다,



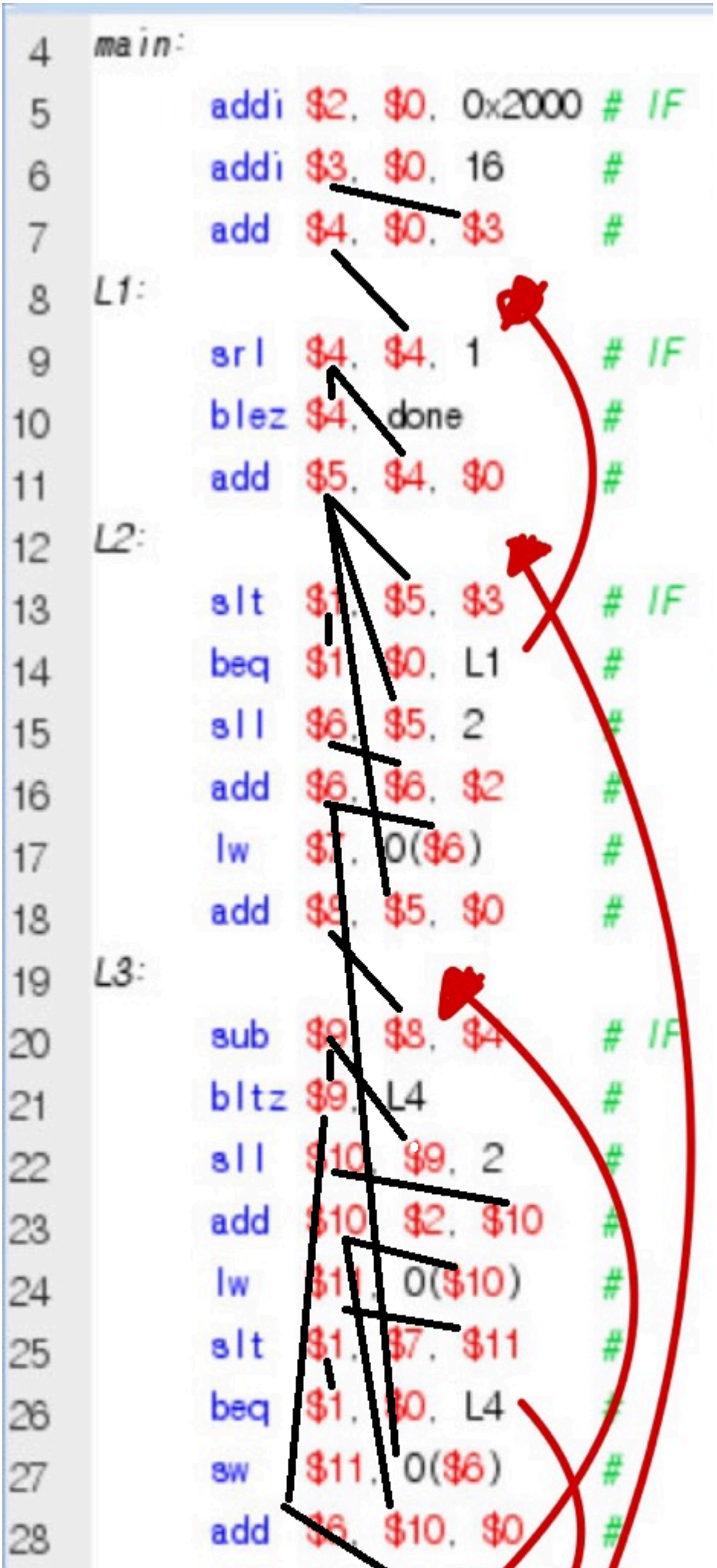

sw $11, 0($6) 저장되어 있는 모습 입니다.


add $6, $10, $0 10번 레지스터의 2000값이 들어갑니다.
add $8, $9, $0 두 레지스터 값 다 0이어서 이후 0도 확인할 수 있습니다.

이후 blez에서 부합하기 전까지 점프를 하기전까지 L3를 반복하지만, 2번째 사이클 후 점프하여 L4로 이동한 모습입니다. 이후 L4에서는




Sw와 Addi를 통해 7번 레지스터오 5번 레지스터를 초기화 시키고 L2로 이동합니다.
위와 같은 반복이 반복되는 모습입니다.


4 main:
5 addi $2, $0, 0x2000 # IF
6 addi $3, $0, 16 #
7 add $4, $0, $3 #
8 L1:
9 srl $4, $4, 1 # IF
10 blez $4, done #
11 add $5, $4, $0 #
12 L2:
13 slt $1, $5, $3 # IF
14 beq $1, $0, L1 #
15 sll $6, $5, 2 #
16 add $6, $6, $2 #
17 lw $7, 0($6) #
18 add $8, $5, $0 #
19 L3:
20 sub $9, $8, $4 # IF
21 bltz $9, L4 #
22 sll $10, $9, 2 #
23 add $10, $2, $10 #
24 lw $11, 0($10) #
25 slt $1, $7, $11 #
26 beq $1, $0, L4 #
27 sw $11, 0($6) #
28 add $6, $10, $0 #

다음은 명령어에 따른 레지스터의 연결성 및 main과 L1~4까지의 이동성을 보여주는 그림입니다.

기존의 모든 명령어에 4개의 NOP를 넣었을 때는 5227개의 사이클이 발생합니다.

```
------------------------------------------
| H020-3-1647-01: Computer Architecture    |
|                 CE.KW.AC.KR              |
------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
------------------------------------------
 Break signal: 1,   # of Cycles:        5227
------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 52385000 (1ps)

C:\Users\user\OneDrive\Desktop\prj3_PCPU_2025>FC /L mem_dump_SS.txt mem_dump.txt
파일을 비교합니다: mem_dump_SS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.


C:\Users\user\OneDrive\Desktop\prj3_PCPU_2025>FC /L reg_dump_SS.txt reg_dump.txt
파일을 비교합니다: reg_dump_SS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.
```

이제부터 해당 사이클을 줄여보겠습니다. 이를 위해서 꼭 필요한 nop를 정의하고 필요없는 nop를 지워보겠습니다.

main:
    addi $2, $0, 0x2000
독립적이라 필요 없습니다다
    addi $3, $0, 16
    nop
    nop
    nop
 add $4, $0, $3는 바로 위의 addi $3, $0, 16이 생성한 $3 값을 사용합니다. 해당 부분에서는 NOP 3개가 필요합니다. 그래야 WB를 마친 이후에, add가 $3을 읽어야 정상 동작합니다.
    add  $4, $0, $3
    nop
    nop
    nop
srl에서 레지스터 4는 add가 마무리된 뒤에 수행해야합니다.
srl  $4, $4, 1
    nop
    nop
    nop
blez또한 srl에서 wb가 마무리된 뒤에 수행해야합니다.
    blez $4, done
    nop
점프가 아닐 때 해저드 방지를 위해서 nop하나가 필요합니다.
    add  $5, $4, $0
    nop
    nop
    nop

```
L2:
slt에서 5번 레지스터를 위한 add를 기다리기 위한 3개의 nop 필요
    slt  $1, $5, $3
    nop
    nop
    nop
1번 레지스터 사용하기 위해서 slt를 기다리 위한 3개의 nop 필요
    beq  $1, $0, L1
    nop
점프가 아닐 때 해저드 방지를 위해서 nop하나가 필요합니다.
    sll  $6, $5, 2
    nop
    nop
    nop
    nop
6번 레지스터를 사용하기 위해 sll를 기다리는 3개의 nop필요
    add  $6, $6, $2
    nop
    nop
    nop
6번 레지스터를 사용하기 위한 add 을 기다리는 3개의 nop 필요
    lw   $7, 0($6)
    nop
lw에서의 딜레이 걸어줍니다.
    add  $8, $5, $0
    nop
    nop
    nop
L3:
8번 레지스터를 사용하기 위해 add의 를 기다리는 3개의 nop 필요
    sub  $9, $8, $4
    nop
    nop
    nop
9번 레지스터를 사용하기 위해 sub의 wb를 기다리는 3개의 nop 필요
    bltz $9, L4
nop는 고려하지 않아서 필요없음
    sll  $10, $9, 2
    nop
    nop
    nop
10번 레지스터를 사용하기 위해 sll의 를 기다리는 3개의 nop 필요
    add  $10, $2, $10
    nop
    nop
    nop
10번 레지스터를 사용하기 위해 add의 를 기다리는 3개의 nop 필요
    lw   $11, 0($10)
```

```
nop
nop
```

```
    nop
```
11번 레지스터를 사용하기 위해 slt를 기다리는 3개의 nop 필요
```
    slt  $1, $7, $11
    nop
    nop
    nop
```
1 번 레지스터를 사용하기 위해 slt를 기다리는 3개의 nop 필요
```
    beq  $1, $0, L4
```
분기 이후 값 사용을 하지 않아 nop없이도 작동합니다.
```
    sw   $11, 0($6)
```
sw에서 6번 레지스터는 변하지 않으니 nop없이도 작동합니다.
```
    add  $6, $10, $0
```
add에서 9번 0번 레지스터 변경이 없어 nop 없이도 작동합니다.
```
    add  $8, $9, $0
    nop
    j    L3
    nop
```
다시 L3로 돌아갈 때 8번 레지스터의 값이 저장되어야 합니다. 하지만 j L3명령어가 있으니 앞뒤의 nop만으로도 대기가 충분합니다.
```
L4:
    sw   $7, 0($6)
```
두 명령어 사이의 연관성이 없기에 무시됩니다.
```
    addi $5, $5, 1
    nop
    j    L2
    nop
```
다시 L2로 돌아갈 때 5번 레지스터의 값이 저장되어야 합니다. 하지만 j L2명령어가 있으니 앞뒤의 nop만으로도 대기가 충분합니다.

```
done:
    break
```

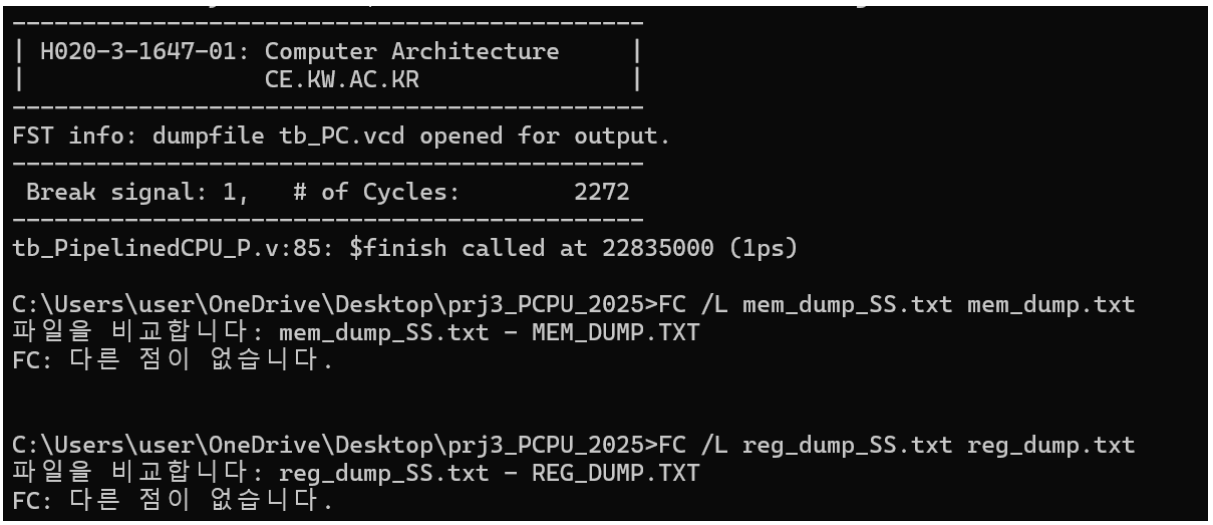
```
-------------------------------------------
| H020-3-1647-01: Computer Architecture     |
|                 CE.KW.AC.KR               |
-------------------------------------------
FST info: dumpfile tb_PC.vcd opened for output.
-------------------------------------------
 Break signal: 1,   # of Cycles:        3169
-------------------------------------------
tb_PipelinedCPU_P.v:85: $finish called at 31805000 (1ps)

C:\Users\user\OneDrive\Desktop\prj3_PCPU_2025>FC /L mem_dump_SS.txt mem_dump.txt
파일을 비교합니다: mem_dump_SS.txt - MEM_DUMP.TXT
FC: 다른 점이 없습니다.


C:\Users\user\OneDrive\Desktop\prj3_PCPU_2025>FC /L reg_dump_SS.txt reg_dump.txt
파일을 비교합니다: reg_dump_SS.txt - REG_DUMP.TXT
FC: 다른 점이 없습니다.
```

결과는 다음과 같이 이전의 사이클보다 줄어든3169를 확인할 수 있었습니다.

이번에는 포워딩을 통해서 nop를 더 줄여보겠습니다.
addi $3, $0, 16
add  $4, $0, $3 #FWD
00_01  //  0x008
add는 첫 번째 명령어 addi의 결과 $3를 바로 사용해야 하므로,
포워딩을 통해 WB 단계의 값을 EX 단계로 전달받을 수 있습니다.

sll  $6, $5, 2
add  $6, $6, $2 #FWD
lw   $7, 0($6)  #FWD
00_01  //  0x090
01_00  //  0x094
add $6, $6, $2는 바로 앞 sll 명령에서 만들어진 6번 레지스터 값을 사용하기 위해서
EX과MEM 사이 일때 값을 포워딩(01)받습니다
lw $7, 0($6)는 add 명령의 결과 6번 레지스터를 사용하기 위해서  EX과 MEM 사이 값을
포워딩하여 사용합니다

add  $10, $2, $10  #FWD
lw   $11, 0($10)   #FWD
nop
slt  $1, $7, $11   #FWD
00_01  //  0x08C
01_00  //  0x090
00_00  //  0x094
00_10  //  0x098
add $10, $2, $10의 10번 레지스터를 사용하기위해서 nop대신 포워딩을 사용합니다.
lw $11, 0($10)도 10번 레지스터를 사용하기위해서 nop대신 포워딩을 사용합니다.
slt $1, $7, $11에서는 $11의 값이 아직 WB단계이므로 MEM과WB사이 단계의 값을
포워딩하여 사용합니다.



# 고찰

이번 프로젝트에서는 이전과 또 다르게 시나리오에 따른 명령어를 수행해보았습니다. 사이클을 최대한 낮추는것이 목표다보니, 해당 시나리오상의 명령어에 한하는 최소한의 사이클을 구현하려는 모습이 느껴졌습니다. 만약에 코드가 유동적으로 바뀐다면 또 다른 결과를 유도할 수 있지 않을까 라는 생각을 하게 되었습니다.

명령어를 구현하는 과정에서 결과적으로 모든줄을 한번씩 하고 종료되는 프로그램이 아니라, 반복되는 명령어다보니 웨이브 폼을 확인하는 과정에서 결과를 모두 확인하기 힘든 경험을 했습니다. 수정 후 결과를 비교하는 명령어 덕분에 좀더 편하게 해당 코드가 정상 작동하는지 판단할 수 있었습니다.

또한 nop만을 판단했을 때는 nop를 최대한 줄여보면서 확인했지만 결국 파이프라인을 구현하는 과정에서 정확한 근거를 기반으로한 nop 제거가 필요함을 확인했습니다.

포워딩을 구현할 때 사실 nop3개가 있는 곳 모두가 가능하지 않을까?라는 생각을하고 구현했는데 예상보다 제한적으로 포워딩이 가능했습니다.
주석을 달면서 nop를 줄여보았는데 너무 여백 공간이 늘어나서 좋은 방법이 아니라고 생각했습니다.

## Reference

프로젝트2 제안서 2025.pdf