

# Computer Architecture

## Project 3

### (Pipeline Architecture)

학 과: 컴퓨터정보공학부

담당교수: 이성원 교수님

실습분반: 금34

학 번: 2020202037

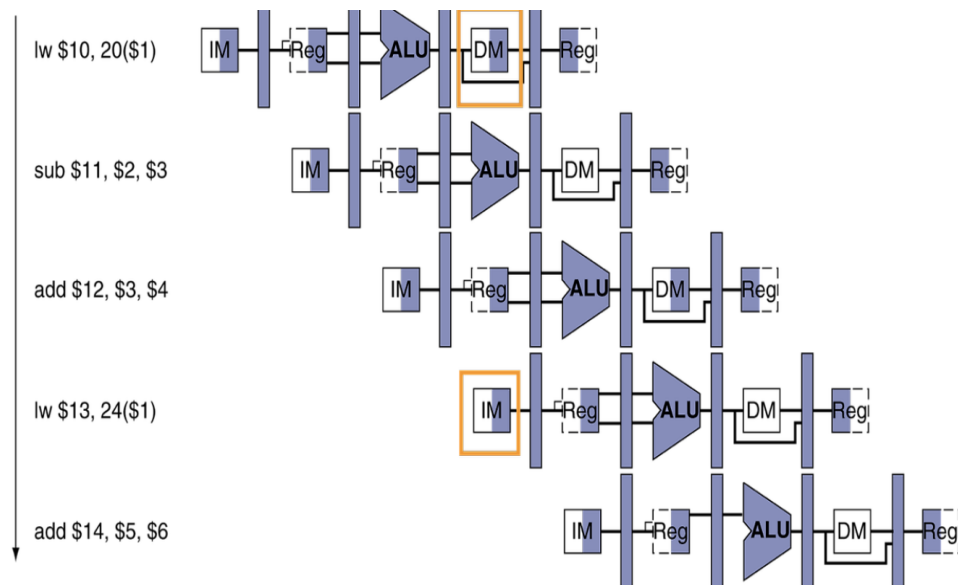
성 명: 엄정호

## 1. 프로젝트 개요

본 프로젝트는 MIPS아키텍처의 파이프라인 아키텍처에서 발생하는 Structural hazards와 Data hazards에 대해 알아보고 주어진 어셈블리코드에 NOP와 forwarding을 이용해 hazard를 해결 하는 방법을 구현한다. 주어진 어셈블리 코드에는 명령어마다 4개의 nop가 할당되어있다. 이 명령어를 그대로 동작 시킬 경우 해당 아키텍처는 single cycle처럼 동작해 hazard를 방지 할 수 있지만 하자드가 발생하지 않는 부분에 nop삽입으로 인해 동작 시 많은 cycle이 소모된다. 이러한 불필요한 NOP를 제거하고, 필요한 경우에만 데이터를 포워딩하여 하자드를 해결하는 것입니다. Structural hazards해결하기 위해 하드웨어 자원 충돌을 방지하고, Data hazards해결하기 위해 필요한 데이터를 미리 전달하여 명령어 실행을 연속적으로 수행할 수 있도록 명령어를 변경한다.

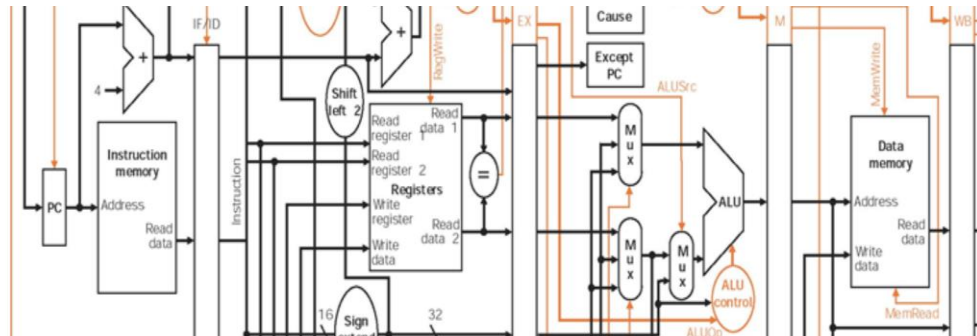
hazard : hazard란 매 사이클마다 새로운 명령어가 실행되는 파이프라인 아키텍처에서 발생하는 dependency에 의해 발생하는 문제로 Structural hazards, Data hazards, control hazards로 나눌 수 있다. 모든 hazard는 각 명령어에 stall을 파이프 라인의 개수만큼 추가함으로써 해결 가능하지만 그렇게 된다면 pipeline의 장점은 사라지고 single cycle처럼 동작하기 때문에 명령어 배치를 바꿔 hazard가 발생하는 부분에 최소한의 stall을 넣고 forward를 이용해 해결하는 것이 바람직하다.

Structural hazards : 파이프라인에서 한 사이클 동안 여러 개의 insturction이 수행되기 때문에 생기는 구조적인 문제이다. 즉 2개의 명령어가 하나의 자원에 동시에 접근 하려고 할 때 발생하게된다.



위 그림은 Structural hazard의 예시이다

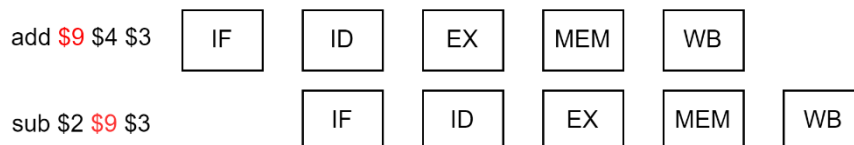
4번째 cycle에서 첫번째 instruction은 data memory를 read 하고 있고, 4번째 instruction은 instruction fetch를 하고 있다. Instruction memory와 data memory가 하나로 이루어져 있다면 첫번째 insturction이 종료된 후에 네번째 instruction을 수행할 수 있다. 즉, structure hazard가 발생한다. 따라서 4번째 명령어가 삽입되기 전 stall을 삽입하거나 아래 그림처럼 instruction memory와 data memory를 구분함으로써 하자드를 해결 할 수 있다.



첫번째 명령어의 register write back stage와 네번째 명령어의 ID stage가 겹치는 것을 관찰할 수 있는데, 이는 두 명령어가 동일한 하드웨어 유닛에 동시에 접근하고 있어서 구조적 하자드가 발생합니다. 이러한 상황에서는 파이프라인을 개선하여 하나의 사이클을 두 개의 사이클로 분할함으로써 하자드를 방지할 수 있습니다. 예를 들어, 첫 번째 명령어의 register write back stage를 한 사이클에서 실행하고, 네 번째 명령어의 ID stage를 해당 사이클의 뒷부분에서 실행하도록 변경할 수 있습니다. 이를 통해 두 명령어가 동시에 하드웨어 자원에 접근하는 것을 피하고, 구조적 하자드를 방지할 수 있습니다. (단, 이러한 변경이 본 프로젝트의 아키텍처에서는 구현되어 있지 않습니다.)

Data hazards:

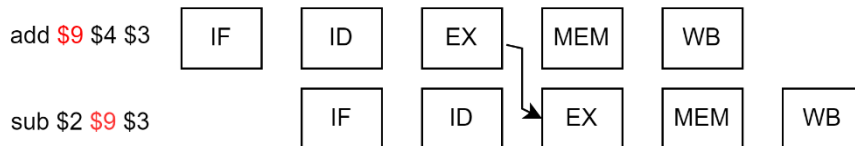
데이터 하자드는 명령어의 실행 순서에 의해 발생하는 문제로 이전 명령어의 결과가 다음 명령어에서 사용될 때 발생하게 된다. 예를 들어



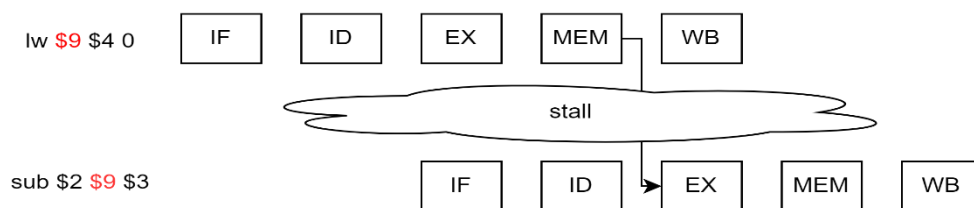
위 명령의 경우 ADD연산의 결과값이 다음 명령어의 SUB명령에 사용되기 때문에 해당 연산값을 사용하기 위해선 위 명령어의 WB stage이후에 sub 명령의 ID stage가

실행되어야 한다. 즉 총 3개의 nop가 삽입 되어야 정상적인 실행이 가능하다.

이를 hardware적으로 해결하기 위해 우리는 forwarding을 사용해야 한다. forwarding이란 이전단계에서 연산 결과가 나오는 단계의 데이터를 미리 가져와 사용하는 것으로 위 예시에서는 EX stage에서 mem 단계로 넘어 갈 때 연산결과를 제공함으로 사용되는 stall의 수를 줄일 수 있다.



forward의 경우에도 연산이 된 이후의 값이 사용되어야 하기 때문에 mem stage의 값이 필요한 load명령어의 경우 1cycle stall이 요구된다.



hard ware로 해결 불가능한 load명령어의 stall은 software적으로 처리 가능하다.

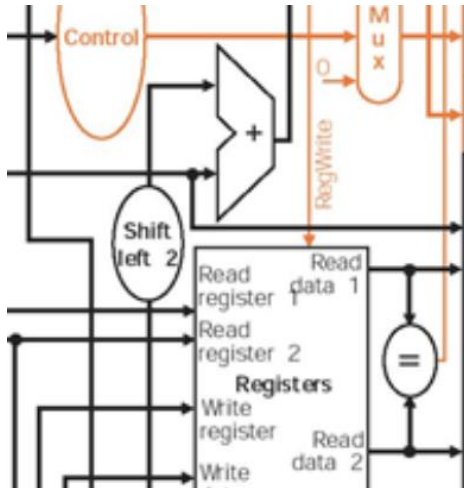
컴파일러가 load와 다음 명령어 사이에 의존성 문제가 발생하지 않는 명령어를 하나 넣어주는 것이다.

예를 들어 위의 동작 수행 과정에서 sub명령어 아래에 `lw $8, $3, 0`이 존재한다고 가정해 보자. 해당 명령어의 경우 위의 두 명령어에 대해 의존성이 발생하지 않음으로 해당 명령어를 stall이 발생하는 위치에 삽입할 경우 불필요한 stall을 줄이는 것이 가능하다.

## Control hazard

Control hazard란 conditional branch를 통해 pc주소가 바뀔 때 이미 파이프라인 안에 들어와 있는 명령어를 버려야 할 때 발생한다. 보통의 파이프라인의 경우 branch여부를 mem단계에서 결정되어 pc를 바꾸기 때문에 branch가 발생할 경우 이전에 들어온 3개의 명령어를 버려야 한다.

이에 대한 해결 방법으로 branch 여부를 미리 확인하는 레지스터를 추가함으로써 ID단계에서 문제 해결이 가능하다

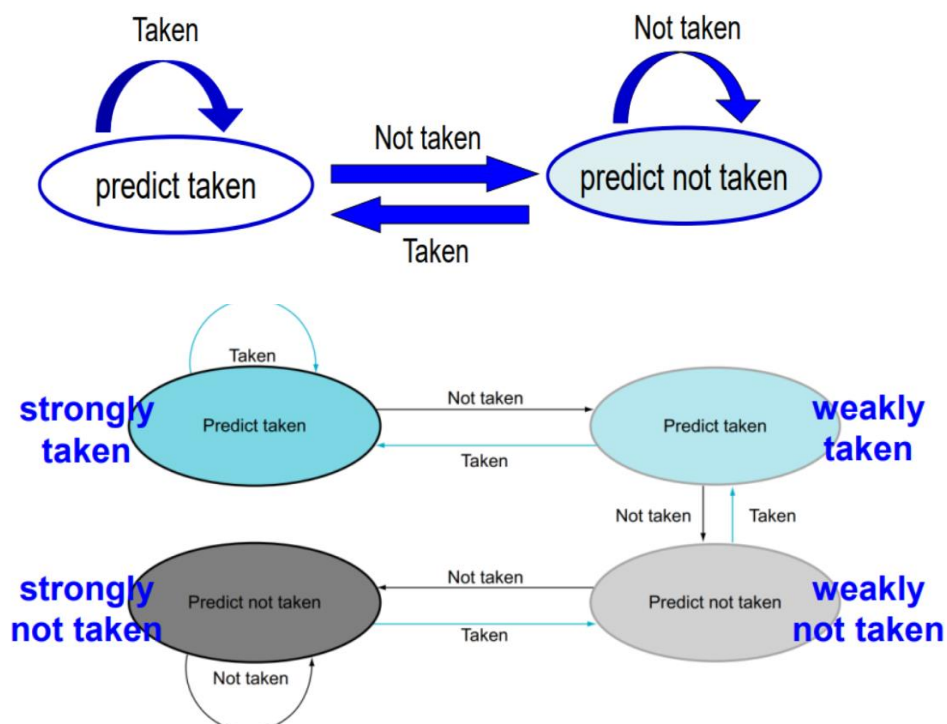


adder와 comparator를 하나씩 추가하여 branch 명령어 실행시 id stage address의 주소를 결정하고 beq/bne명령어를 실행함으로써 다음 분기의 발생 여부를 판단한다. 이 연산 결정 진행동안 명령어가 들어오면 안됨으로 하나의 nop 삽입이 요구된다.

하지만 이러한 레지스터 추가에도 여전히 stall은 존재하게 된다. 이 stall로 인한 지연을 해결하기 위해 예측 분기를 사용할 필요가 있다. 예측 분기란 예측을 통해 분기가 일어날지 일어나지 않을지 판단하는 것으로 기본적으로 untaken단계에서 시작한다.

prediction을 untaken으로 설정해 두고 다음 명령을 계속 받을 때 분기가 발생하지 않는다면 stall을 줄일 수 있고 분기가 발생했다면 stall이 발생하게 된다. 이러한 분기 예측은 정적 분기 예측과 동적 분기예측으로 나눌 수 있는데 정적 분기 예측의 경우 branch의 일반적인 패턴을 통해 예측을 수행한다. loop와 같은 반복문의 경우 여러 번의 taken과 한번의 not taken으로 해당 반복문을 빠져나오기 때문에 100번의 loop를 반복할동안 99번의 stall을 줄이고 1번의 stall만을 발생하도록 할 수 있다.

동적 분기예측은 이전의 분기 결과를 이용해 다음 분기를 예측하는 것이다. 1bit 예측과 2bit 예측 방법이 있는데



1bit 예측의 경우 예측이 틀릴 때 마다 다음 예측 방향을 바꾸기 때문에 50% 정도의 성공 확률을 보이지만 2bit 예측의 경우 두 번의 동일한 예측을 수행하기 때문에 75%의 성공 확률을 보여준다. 분기 예측으로 stall을 줄이기 위해선 fetch단계에서 다음 branch가 어디로 이동할지 알아야 하는 문제점이 발생하는데 이를 해결하기 위한 것이 BTB(branch target buffer)이다 BTB란 이전 분기 명령에서의 분기 위치를 저장하고 이후 동일한 분기 명령을 만났을 때 빠르게 타겟 주소를 제공하는데 사용된다.

## 2. 검증 전략 및 분석 결과

어셈블리코드 & dependency

main:

```
ori $a0, $zero, 0x2000
ori $a1, $zero, 0x2280
ori $a2, $zero, 0x2200
ori $a3, $zero, 0x0010
ori $s0, $zero, 0x0005
and $t5, $zero, $zero
and $t0, $zero, $zero
```

L0:

```
and $t1, $zero, $zero
add $t8, $a2, $zero
```

L1:

```
sw $zero, 0($t8)
addi $t8, $t8, 0x04
addi $t1, $t1, 0x01
slti $at, $t1, 0x08
bne $at, $zero, L1
add $t6, $a0, $zero
and $t1, $zero, $zero
```

L2:

```
lw $v0, 0($t6)
srlv $v1, $v0, $t5
andi $t4, $v1, 0x7
sll $v1, $t4, 0x02
mult $v1, $a3
mflo $t9
add $t9, $a1, $t9
add $t8, $a2, $v1
lw $v1, 0($t8)
sll $t7, $v1, 0x02
```

```

add $t7, $t9, $t7
sw $v0, 0($t7)
addi $v1, $v1, 0x01
sw $v1, 0($t8)
addi $t6, $t6, 0x04
addi $t1, $t1, 0x01
slt $at, $t1, $a3
bne $at, $zero, L2
add $t6, $a0, $zero
and $t1, $zero, $zero

```

L3:

```

and $t2, $zero, $zero
sll $t8, $t1, 0x02
mult $t8, $a3
mflo $t7
add $t8, $a2, $t8
lw $v1, 0($t8)
beq $v1, $zero, L5
add $t7, $a1, $t7

```

L4:

```

lw $v0, 0($t7)
sw $v0, 0($t6)
addi $t6, $t6, 0x04
addi $t7, $t7, 0x04
addi $t2, $t2, 0x01
slt $at, $t2, $v1
bne $at, $zero, L4

```

L5:

```

add $t1, $t1, 0x01
slti $at, $t1, 0x08

```



```

bne $at, $zero, L3
add $t5, $t5, 0x03
add $t0, $t0, 0x01
slt $at, $t0, $s0
bne $at, $zero, L0

```

*data ha zero di compare karna*  
*data ha zero*  
*data ha zero*  
*e compare karna*

## NO Forwarding

```

main:
nop
ori $a0, $zero, 0x2000
ori $a1, $zero, 0x2280
ori $a2, $zero, 0x2200
ori $a3, $zero, 0x0010
ori $a0, $zero, 0x0005
and $t5, $zero, $zero
and $t0, $zero, $zero

```

main -> 정렬에 이용할 값들을 각 레지스터에 할당한다.

모두 다른 레지스터에 값을 할당하므로 dependency가 발생하지 않아 nop가 필요하지 않다.

```

L0:
and $t1, $zero, $zero
add $t8, $a2, $zero
nop
nop
nop
L1:
sw $zero, 0($t8)

```

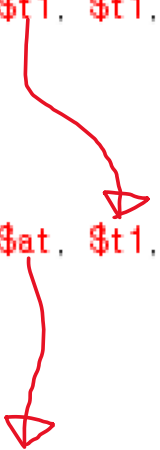
L0,L1

L0의 마지막 add의 연산 결과를 L1이 SW명령에 이용하게 된다.

add연산의 값이 store의 ID단계에서 필요하기 때문에 해당 단계 이전에 \$8에 연산 결과가 존재하면 된다. 따라서 add의 WB단계에 sw의 Fetch가 시작 하면 WB이후 id stage가 실행되기 때문에 총 3개의 nop가 필요하다.

L1:

```
sw    $zero, 0($t8)
addi  $t8, $t8, 0x04
addi  $t1, $t1, 0x01
#nop
#nop
#nop
slti  $at, $t1, 0x08
nop
nop
nop
bne   $at, $zero, L1
nop
```



addi \$t1 \$t1 0x01연산 값이 slti의 연산에 사용된다. 따라서 addi값이 WB이후 slti의 id단계가 수행되어야 함으로 3개의 nop가 요구된다.

```

add    $t6, $a0, $zero
and    $t1, $zero, $zero
#nop
#nop
L2:
lw     $v0, 0($t6)

```

addi \$t6 \$a0 \$zero 연산의 수행 결과 L2 lw명령어의 alu연산에 필요하다. 아래 and \$t1 \$zero, \$zero명령어가 없었다면 3개의 nop가 필요하지만 의존성이 없는 해당 명령어가 하나의 nop를 대신하고 있기 때문에 해당 명령어 이후 2개의 nop가 요구된다.

```

L2:
lw     $v0, 0($t6)
#nop
#nop
nop
srlv   $v1, $v0, $t5
#nop
#nop
#nop
andi   $t4, $v1, 0x7
#nop
#nop
#nop
sll    $v1, $t4, 0x02

```

세 명령어 모두 이전 연산의 결과값이 다음 alu연산에 사용된다. 따라서 모두 3개의 nop가 필요하다.

```

mult $v1, $a3
mflo $t9
#nop
#nop
#nop
add $t9, $a1, $t9
add $t8, $a2, $v1
#nop
#nop
#nop
lw $v1, 0($t8)
#nop
#nop
nop
sll $t7, $v1, 0x02
#nop
#nop
#nop

```

mult는 alu연산 후 해당 연산 값을 내부 레지스터에 저장하기 때문에 뒤 명령어는 대기 없이 alu에 존재하는 이전 곱 연산의 low값을 가져올 수 있어 nop가 필요하지 않다.

add \$8 \$a2 \$v1

명령어의 연산 결과가 저장되는 레지스터의 값을 다음 lw명령어의 alu 연산에서 사용되기 때문에 3개의 nop가 들어가 WB stage이후에 lw의 ID stage가 나오게 된다.

lw \$1 \$v0(\$8)

lw값이 저장되는 레지스터 값을 sll명령어에서 사용하기 때문에 lw명령어의 wb 단계 이후에 다음 명령어의 ID단계가 실행되어야 한다 때문에 3개의 nop가 필요하다.

```

sll    $t7, $v1, 0x02
#nop
#nop
#nop
add    $t7, $t9, $t7
#nop
#nop
#nop
sw     $v0, 0($t7)
addi   $v1, $v1, 0x01
#nop
#nop
#nop
sw     $v1, 0($t8)

```

sll / add / addi

세 명령어 모두 다음 명령어에서 이전 연산의 결과를 가지고 있는 레지스터가 필요하기 때문에 3개의 nop가 필요하다.

```

addi   $t6, $t6, 0x04
addi   $t1, $t1, 0x01
#nop
#nop
#nop
slt    $at, $t1, $a3
nop
nop
nop
bne    $at, $zero, L2
nop
add    $t6, $a0, $zero
and    $t1, $zero, $zero

```

addi / slt

명령의 결과값이 다음 명령어의 연산에 이용되고 있기 때문에 해당 명령어의 WB stage이후 ID stage의 시작을 위해 각각 3개의 nop가 요구된다.

bne 명령어의 경우 branch 여부를 decode단계에서 확인 하기 때문에

decode 단계 이후에 다음 명령어를 받아야 함으로 1개의 nop가 필요하다.

```
L3:
    and    $t1, $zero, $zero
    and    $t2, $zero, $zero
    nop
    nop
    sll    $t8, $t4, 0x02
    nop
    #nop
    #nop
    mult   $t8, $a3
    mflo   $t7
    add    $t8, $a2, $t8
    #nop
    #nop
    #nop
    lw     $v1, 0($t8)
```

and \$t1, \$zero, \$zero연산의 결과를 sll명령어에서 사용하기 위해 and 연산의 WB stage 이후 sll의 ID단계가 실행되어야 한다 이를 위해 3개의 nop명령어가 필요하지만 명령어 사이에 and가 nop를 대신할 수 있기 때문에 2개의 nop를 사용해 하자드 해결이 가능하다.

sll과 add \$t8 \$a2 \$t8명령어도 마찬가지로 총 3개의 nop가 필요하며 mflo의 경우 mult에서 alu연산 후 저장된 값을 사용하기 때문에 nop가 필요하지 않다.

```

lw    $v1, 0($t8)
nop
nop
nop
beq   $v1, $zero, L5
nop
add   $t7, $a1, $t7
#nop
#nop
#nop
L4:
lw    $v0, 0($t7)
nop
#nop
#nop
sw    $v0, 0($t6)

```

```
lw    $v1, 0($t8)
```

```
add   $t7, $a1, $t7
```

```
lw    $v0, 0($t7)
```

세 명령어의 경우 해당 명령어의 연산 결과가 다음 명령어에서 이용되기 때문에 3개의 nop가 필요하며

```
beq   $v1, $zero, L5
```

명령어의 경우 branch여부를 판단하기 전까지 새로운 명령어를 패치하지 않아야 함으로 하나의 nop가 필요하다.

```

addi $t6, $t6, 0x04
addi $t7, $t7, 0x04
addi $t2, $t2, 0x01
#nop
#nop
#nop
slt $at, $t2, $v1
nop
nop
nop
bne $at, $zero, L4
nop

```

L5:

```
addi $t2, $t2, 0x01
```

```
slt $at, $t2, $v1
```

위 두 명령어의 결과값을 다음 명령에서 사용하기 위해 3개의 nop가 요구된다.

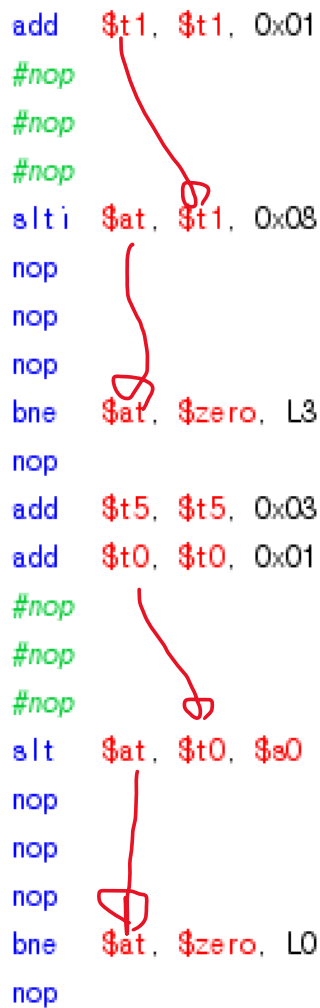
```
bne $at, $zero, L4
```

branch 여부가 id 단계에서 결정됨으로 1개의 nop가 필요하다.



L5:

```
add    $t1, $t1, 0x01
#nop
#nop
#nop
slti    $at, $t1, 0x08
nop
nop
nop
bne     $at, $zero, L3
nop
add     $t5, $t5, 0x03
add     $t0, $t0, 0x01
#nop
#nop
#nop
slt     $at, $t0, $s0
nop
nop
nop
bne     $at, $zero, L0
nop
```



```
add    $t1, $t1, 0x01
```

```
slti    $at, $t1, 0x08
```

```
add     $t0, $t0, 0x01
```

```
slt     $at, $t0, $s0
```

4개의 명령어의 결과 값을 다음 레지스터에서 사용하기 위해 WB이후에 다음 명령어의 ID단계가 실행되어야 함으로 최소 3개의 nop가 요구되며

```
bne     $at, $zero, L3
```

```
bne     $at, $zero, L0
```

branch명령의 경우 branch여부를 id단계에서 판단하기 때문에 1개의 nop 명령이 요구된다.

```

-----
| H020-3-1647-01: Computer Architecture |
| CE.KW.AC.KR |
-----
FST info: dumpfile tb_PC.vcd opened for output.
-----
Break signal: 1, # of Cycles: 7678
-----
tb_PipelinedCPU_P.v:85: $finish called at 76895000 (1ps)

```

single cycle    total cycle

## Forwarding

bne명령의 경우 본 프로젝트에서는 alu포워딩이 불가능하다. 이유는 분기 명령어가 디코딩 단계에서 이미 처리되기 때문에 alu값을 전달하는 방식인 alu 포워딩은 데이터 종속성 문제를 해결하지 못하기 때문이다

즉 branch이전에 발생한 nop와 branch연산을 기다리는 nop 1개는 지우지 못한다.

```

22      addi $t1, $t1, 0x01
23      #nop
24      #nop
25      #nop
26      slti $at, $t1, 0x08

```

```

00_00 // 0x03C addi $t1, $t1, 0x01
01_00 // 0x040 slti $at, $t1, 0x08

```

slti의 ALU 연산에 사용될 값을 addi명령어 수행 중 alu에서 MM stage로 이동 하는 값을 가져와 사용해 nop3개를 줄이는 것이 가능하다.

```

add    $t6, $a0, $zero
and     $t1, $zero, $zero
#nop
#nop

```

L2:

```

lw     $v0, 0($t6)

```

```

00_00 // 0x058  add  $t6, $a0, $zero
00_00 // 0x05C  and   $t1, $zero, $zero
10_00 // 0x060  lw   $v0, 0($t6)

```

연산에 사용할 \$6레지스터의 값을 add \$6 \$a0 \$zero 명령에서 가져와야 하기 때문에 MM->WB state로 넘어가는 값을 alu A값으로 가져오면 nop를 2개 줄일 수 있다.

이 부분의 경우 branch지점 사이에 명령어가 존재해 forwarding이 수행되지 않을 수 있으나 본 프로젝트에서는 문제없이 코드 수행이 가능했다.

```

lw     $v0, 0($t6)
#nop
#nop
nop
or lw  $v1, $v0, $t5

```

```

10_00 // 0x060 lw $v0, 0($t6)
00_00 // 0x06C
00_10 // 0x070 srlv $v1, $v0, $t5

```

load명령어의 경우 v1값이 MM ->WB stage에서 생성됨으로 해당 구간에서 forward를 진행 할 수 있다.

srlv에서 alu B값으로 v1레지스터를 사용하기 때문에 해당 구간에서 포워딩을 진행한다.

```
srlv $v1, $v0, $t5
```

```
#nop
```

```
#nop
```

```
#nop
```

```
andi $t4, $v1, 0x7
```

```
#nop
```

```
#nop
```

```
#nop
```

```
sll $v1, $t4, 0x02
```

```
#nop
```

```
#nop
```

```
#nop
```

```
mult $v1, $a3
```

```

00_10 // 0x070 srlv $v1, $v0, $t5
01_00 // 0x074 andi $t4, $v1, 0x7
00_01 // 0x078 sll $v1, $t4, 0x02
01_00 // 0x07C mult $v1, $a3

```

위 세가지 명령어 모두 forwarding을 이용해 nop를 3개씩 줄이는 것이 가능했다 각각 forward이 필요한 지점의 alu 값을(A or B) 이전 명령어의 alu -> MM 단계에서 값을 가져오는 것이 가능했다.

```
mflo $t9
```

```
#nop
```

```
#nop
```

```
#nop
```

```
add $t9, $a1, $t9
```

```
00_00 // 0x080 mflo $t9
```

```
00_01 // 0x084 add $t9, $a1, $t9
```

add명령어에서 사용할 레지스터 \$9 값이 이전 단계의 ALU->MM구간에서 생성되기 때문에 해당 구간에서 포워딩 가능했다.

```
add $t8, $a2, $v1
```

```
#nop
```

```
#nop
```

```
#nop
```

```
lw $v1, 0($t8)
```

```
#nop
```

```
#nop
```

```
nop
```

```
sll $t7, $v1, 0x02
```

```

00_00 // 0x088 add $t8, $a2, $v1
01_00 // 0x08C lw $v1, 0($t8)
00_00 // 0x098
00_10 // 0x09C sll $t7, $v1, 0x02

```

LW명령의 경우 이전 명령어의 ALU->MM stage값을 포워딩 하는 것으로 nop를 3개 줄일 수 있으며 sll의 경우 lw의 레지스터 값이 mm->wb stage에서 가져와야 함으로 1개의 nop가 필요했다.

```
sll $t7, $v1, 0x02
```

```
#nop
```

```
#nop
```

```
#nop
```

```
add $t7, $t9, $t7
```

```
#nop
```

```
#nop
```

```
#nop
```

```
sw $v0, 0($t7)
```

```

00_10 // 0x09C sll $t7, $v1, 0x02
00_01 // 0x0A0 add $t7, $t9, $t7
01_00 // 0x0A4 sw $v0, 0($t7)

```

add와 SW명령어 모두 이전 단계의 ALU->MM stage값을 포워딩 함으로써 nop를 줄이는 것이 가능하다.

```
addi $v1, $v1, 0x01
```

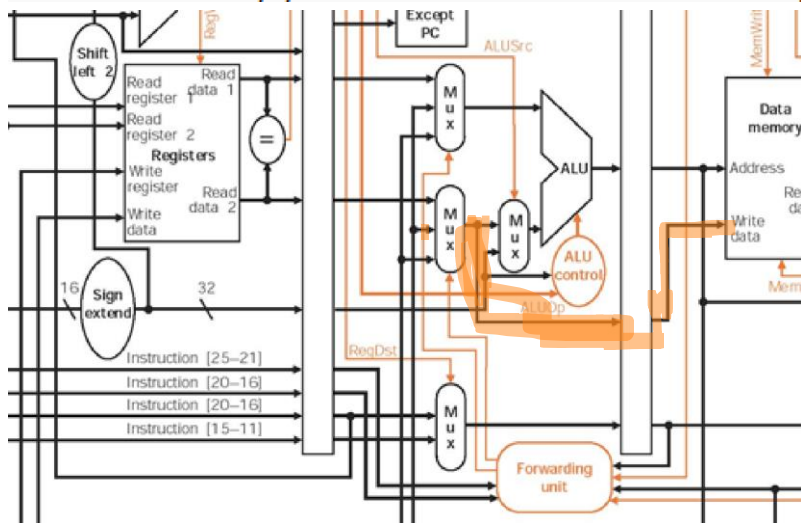
```
#nop
```

```
#nop
```

```
#nop
```

```
sw $v1, 0($t8)
```

00_00	//	0x0A8	addi \$v1, \$v1, 0x01
00_01	//	0x0B8	sw \$v1, 0(\$t8)



위 회로도를 보면 ALU B에서 나온 값이 write data로 이동하는 것을 확인 가능하다 즉 write back에 필요한 데이터를 포워딩 유닛을 통해 가져오고 다음 mux를 통해 IMM value를 사용한 연산을 수행한다면 이구간 또한 ALU->WBstage에서의 포워딩이 가능하다.

```

addi  $t1, $t1, 0x01
#nop
#nop
#nop
slt   $at, $t1, $a3
nop
nop
nop
bne   $at, $zero, L2
nop
add   $t6, $a0, $zero

```

00_00	//	0x0C0	addi \$t1, \$t1, 0x01
01_00	//	0x0C4	slt  \$at, \$t1, \$a3
00_00	//	0x0C8	
00_00	//	0x0CC	
00_00	//	0x0D0	
00_00	//	0x0D4	bne  \$at, \$zero, L2
00_00	//	0x0D8	
00_00	//	0x0DC	

slt 명령어에서 필요한 \$at값이 addi 명령의 alu->Mm stage에서 연산 결과값이 나오기 때문 해당 구간에서 포워딩을 수행함으로써 nop를 줄이는 것이 가능했다.



```
and $t1, $zero, $zero
```

L3:

```
and $t2, $zero, $zero
```

```
nop
```

```
nop
```

```
sll $t8, $t1, 0x02
```

```
00_00 // 0x0E4 and $t2, $zero, $zero
```

```
00_00 // 0x0E8
```

```
00_00 // 0x0EC
```

```
00_00 // 0x0F0 sll $t8, $t1, 0x02
```

and와 sll사이의 nop의 경우 해당 명령어 사이에 분기 지점이 있기 때문에 분기가 발생했을 때 분기지점 이전의 and연산 값을 가져올 경우 문제가 생기기 때문에 포워딩이 불가능하다.

```
sll $t8, $t1, 0x02
```

```
nop
```

```
#nop
```

```
#nop
```

```
mult $t8, $a3
```

```
00_00 // 0x0F0 sll $t8, $t1, 0x02
```

```
00_00 // 0x0FC
```

```
10_00 // 0x100 mult $t8, $a3
```

위 sll 명령의 경우 alu -> mem단계에서 포워딩을 수행 했으나 해당 구간에서 포워딩이 불가능했다. 때문에 1 cycle딜레이를 주고 그 다음 단계에서 포워딩을 진행해 2개의 nop를 지우는 것이 가능했다.

```

add    $t8, $a2, $t8
#nop
#nop
#nop
lw     $v1, 0($t8)
nop
nop
nop
beq    $v1, $zero, L5
nop
add    $t7, $a1, $t7

```

00_00	//	0x108	add	\$t8, \$a2, \$t8
01_00	//	0x10C	lw	\$v1, 0(\$t8)
00_00	//	0x110		
00_00	//	0x114		
00_00	//	0x118		
00_00	//	0x11C	beq	\$v1, \$zero, L5
00_00	//	0x120		
00_00	//	0x124	add	\$t7, \$a1, \$t7

lw \$t8.값을 add의 alu->MM stage에서 연산된 값을 가져오는 포워딩을 진행하는 것으로 해당 구간의 nop를 지울 수 있었다.  
위에서 설명했듯 beq양쪽의 nop는 지우는 것이 불가능하다.

```
add    $t7, $a1, $t7
```

```
#nop
```

```
#nop
```

```
#nop
```

L4:

```
lw     $v0, 0($t7)
```

```
nop
```

```
#nop
```

```
#nop
```

```
sw     $v0, 0($t6)
```

```
00_00 // 0x120
```

```
00_00 // 0x124    add    $t7, $a1, $t7
```

```
01_00 // 0x128    lw     $v0, 0($t7)
```

```
00_00 // 0x120
```

```
00_10 // 0x138    sw     $v0, 0($t6)
```

add명령의 경우 레지스터에 값을 쓰는 연산 결과가 alu -> mm stage에서 나오기 때문에 해당 구간에서 포워딩을 수행함으로써 nop를 3개 줄일 수 있었으며 sw명령의 경우 mm ->wb stage에서 레지스터에 쓸 값이 나오기 때문에 한 개의 nop가 사용되었다.

```

addi  $t2, $t2, 0x01
#nop
#nop
#nop
slt   $at, $t2, $v1
nop
nop
nop
bne   $at, $zero, L4
nop

```

00_00	//	0x144	addi \$t2, \$t2, 0x01
01_00	//	0x148	slt  \$at, \$t2, \$v1
00_00	//	0x14C	
00_00	//	0x150	
00_00	//	0x154	
00_00	//	0x158	bne  \$at, \$zero, L4
00_00	//	0x15C	

slt에 사용될 \$2 레지스터 값은 이전 addi 명령에서 alu ->mm stage에서 생성되기 때문에 해당 구간에서의 포워딩을 통해 nop를 줄이는 것이 가능하다.

```

add    $t1, $t1, 0x01
#nop
#nop
#nop
slti   $at, $t1, 0x08
nop
nop
nop
bne    $at, $zero, L3
nop

```

01_00	//	0x164	slti \$at, \$t1, 0x08
00_00	//	0x168	
00_00	//	0x16C	
00_00	//	0x170	
00_00	//	0x174	bne \$at, \$zero, L3
00_00	//	0x178	

slti에 사용될 \$t1 레지스터 값은 이전 add 명령에서 alu -> mm stage에서 생성되기 때문에 해당 구간에서의 포워딩을 통해 nop를 줄이는 것이 가능하다.

```

add    $t5, $t5, 0x03
add    $t0, $t0, 0x01
#nop
#nop
#nop
slt    $at, $t0, $s0
nop
nop
nop
bne    $at, $zero, L0
nop

```

slt에 사용될 \$t0 레지스터 값은 이전 add 명령에서 alu -> mm stage에서 생성되기 때문에 해당 구간에서의 포워딩을 통해 nop를 줄이는 것이 가능하다.

총 cycle 수

```

-----
| H020-3-1647-01: Computer Architecture |
|                               CE.KW.AC.KR |
-----
FST info: dumpfile tb_PC.vcd opened for output.
-----
Break signal: 1,   # of Cycles:          4216
-----
tb_PipelinedCPU_P.v:85: $finish called at 42275000 (1ps)

```

단순히 nop를 추가한 것에 비해 3000cycle정도 줄은 것을 확인 할 수 있었다. 이는 명령어 자체에서 줄인 nop개수는 적지만 해당 반복문을 수행하는 부분의 nop가 줄어 실제로는 해당 명령어가 반복되는 만큼의 사이클을 줄일 수 있었기 때문이라고 생각한다.

### 3. 문제점 및 고찰

본 프로젝트에서는 파이프라인 아키텍처에서 수행되는 어셈블리 명령

어에서 hazard가 발생하는 부분에 nop또는 forwarding을 통해 해당 hazard를 해결하는 과제를 수행했다. control hazard의 경우 레지스터의 읽기와 쓰기를 하는 부분을 한 클럭의 전과 후로 하드웨어 적으로 해결되었기에 고려할 필요가 없었으며 데이터의 의존성 사이에서 발생하는 데이터 하자드와 branch명령어 사용시 발생하는 컨트롤 하자드에 대해 고려해야 했다. 프로젝트를 진행하면서 내가 다루고 있는 cpu의 회로를 제대로 파악하는 것이 중요하다는 것을 느꼈다. 단순히 배웠던 내용을 생각해 WB단계와 ID단계가 한 클럭안에서 동시에 수행 될 수 있다고 생각하고 nop2개씩 넣어 무한 루프를 도는 상황이 발생했기 때문이다. forward의 경우 branch구간 사이에 포워딩이 필요할 경우 포워딩이 불가능 할 것이라고 생각 했는데 포워딩으로 해결 가능한 부분이 있었고 같은 명령어 라도 다르게 포워딩이 필요한 부분 또한 존재할 수 있음을 알게 되었다. 이번 프로젝트를 통해 mips의 파이프라인 아키텍처에서 발생할 수 있는 hazard를 다루는 방법을 배울 수 있었다.