컴퓨터 구조 프로젝트3 보고서

학번 : 2018202046

이름 : 이준휘

교수님 : 이성원

분반 : 월3, 수4

1. Introduction

해당 과제에서는 pipeline 방식으로 설계된 processor가 주어진다. 해당 processor를 가지고 주어진 코드(Bubble sort)에서 nop을 줄임으로써 cycle을 줄이는 방법을 시뮬레이션한다. 또한 기존 코드에 ALU forwarding 신호를 추가적으로 사용하여 더욱 많은 NOP을 제거하고, 재구성하여 만든 코드를 시뮬레이션한다.

1. Theorem
2. Hazard

Pipeline은 Multi-cycle processor에서 나눈 stage를 이용하여 각 stage별로 다른 명령을 수행함으로써 throughput을 높이는 방법이다. 하지만 해당 방식은 기존 Multi-cycle processor에서 나오지 않던 문제가 발생하게 된다. 크게 Structural Hazard, Data Hazard, Control Hazard로 나눌 수 있다.

Structural Hazard는 Hardware의 설계상에서 일어난다. 가장 우선적으로 만약 기존의 multi-cycle processor의 명령을 그대로 가져와서 설계하였다 해보자. 그렇다면 R-type은 4-cycle, branch는 3-cycle, read는 5-cycle, write는 4-cycle로 각각 다를 것이다. 이 때 명령어들이 수행되는 과정에서 중복되는 unit을 동시에 사용하는 경우 이에 대한 Conflict가 발생할 수밖에 없다. 이를 방지하기 위해 pipeline processor에서는 모든 명령들의 stage를 5 cycle에 수행되도록 고정해두는 방식(branch 제외)을 통해 각 명령들에 대하여 하나의 unit에만 접근할 수 있도록 설계된다.

Data Hazard는 Data의 접근에서 나타나게 된다. 쉽게 말해서는 Data에 대한 Read/Write Sequence가 달라지면서 발생하는 문제로 RAW(Read After Write), WAW(Write After Write), WAR(Write After Read)로 나눌 수 있다. 이러한 문제들의 큰 원인은 이전 명령을 통해 register에 값이 쓰이기 이전에 다른 명령을 통해 해당 register에 접근을 하는 경우 쓰이려는 값이 아닌 이전 값이 접근되면서 나타난다. 이를 해결하는 방법은 우선 기본적인 방법인 stall과 forwarding unit을 통해 해결하는 방법이 있다. 이는 아래에서 설명할 예정이다.

마지막으로 남은 Hazard는 Control Hazard는 pipeline에서 jump나 branch가 발생할 경우 나타나는 문제다. 해당 문제의 경우 Branch의 상태에 따라 다음 명령어들이 삭제되거나 유지되야 한다. 물론 이를 해결하기 위해서는 기본적인 방법인 stall을 사용할 수 있다. 하지만 stall를 사용할 경우 2 cycle의 손실이 나오게 된다. 2 cycle의 손실이 있지만 만약 Branch를 EX stage가 아닌 ID에서 판단하여 jump할 경우 1cycle의 손실만을 가질 수 있다. 하지만 이러한 손실도 크기 때문에 현대의 processor에서는 prediction을 통해 branch를 하는 경우와 안 하는 경우를 판단하여 stall을 더욱 줄이게 된다.

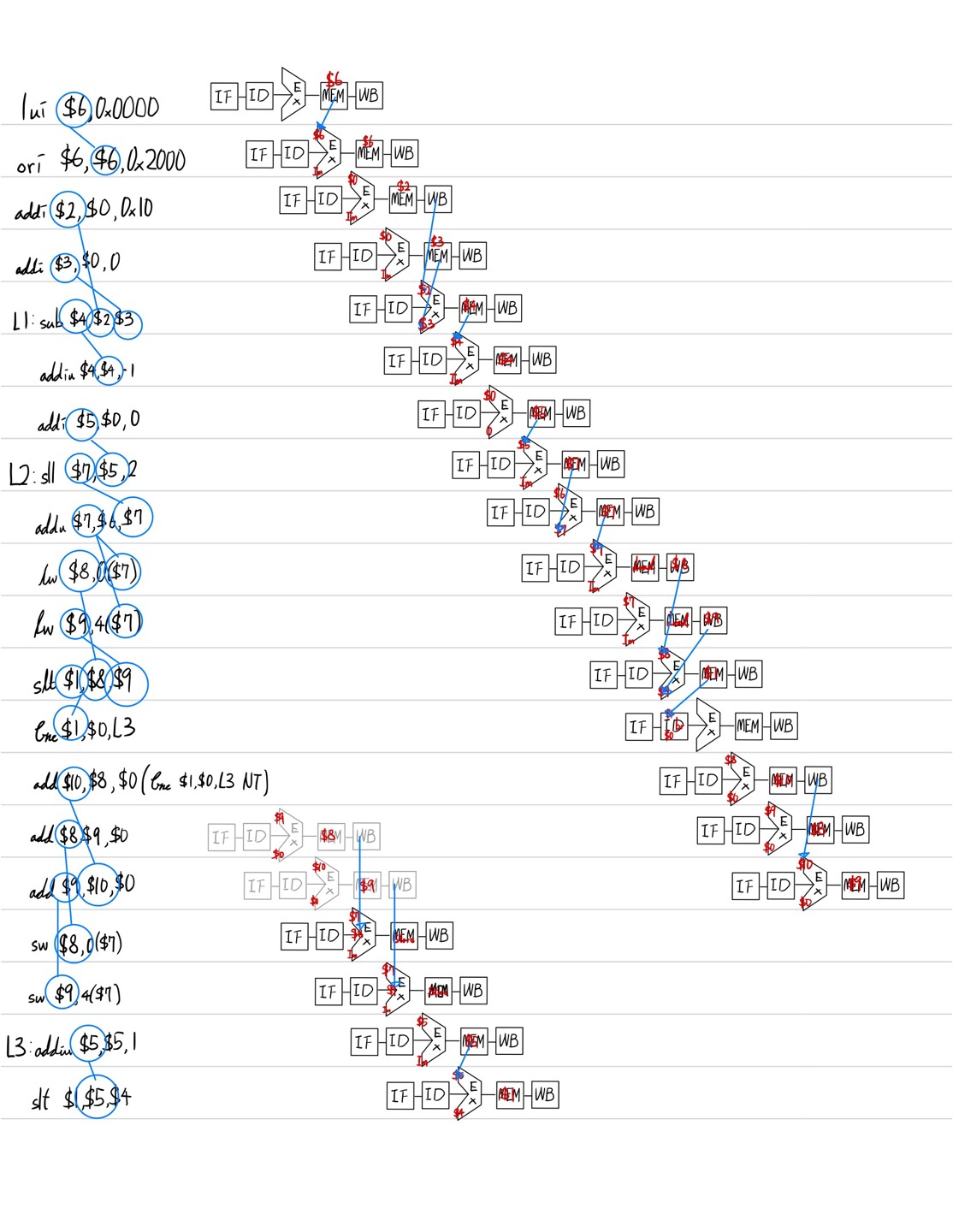
1. S/W-forwarding

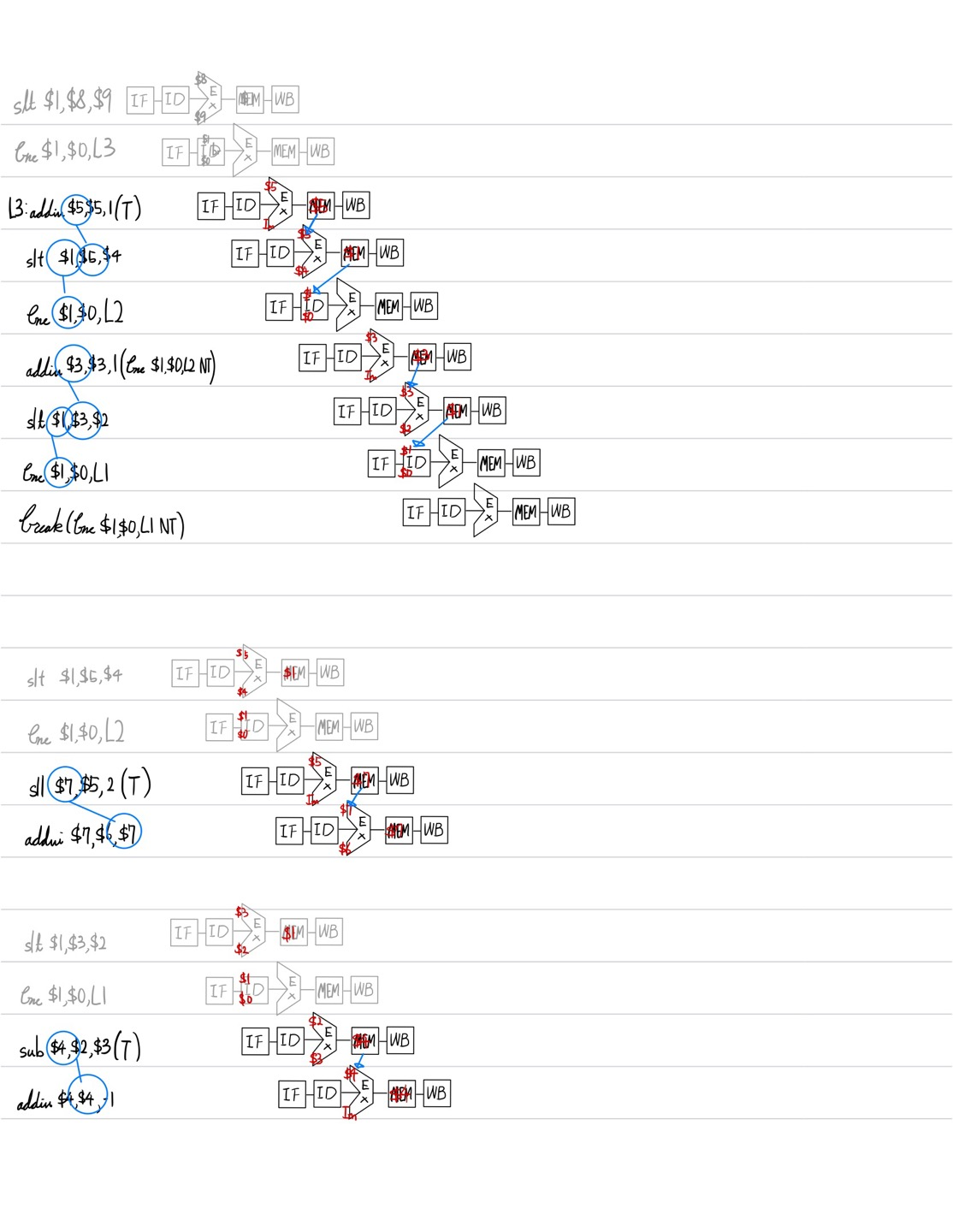
S/W-forwarding은 stall(nop)을 삽입하여 Hazard를 해결하는 가장 단순한 방법이다. 해당 방식은 가장 단순하지만 stall을 사용하는 만큼 throughput이 줄어드는 단점이 있다. 때문에 이러한 방식을 사용할 때에는 stall의 개수를 최대한 줄이는 것이 중요하다. 가장 우선으로 processor가 어떠한 환경에서도 돌아가기 위해서 4개의 stall을 넣는 방법이 있다. 하지만 해당 방식은 비효율적이다. 때문에 pipeline 상에 데이터가 존재할 경우 pipeline을 빠져나갈 때까지 stall의 개수를 조절하여 이를 해결하는 방법을 사용한다. 또는 명령어들의 순서를 바꾸어 nop을 줄이는 방법 또한 존재한다.

1. H/W-forwarding

H/W-forwarding이란 만약 접근하는 register의 데이터가 pipeline 안에 존재하는 경우 해당 데이터를 mux와 forwarding unit을 통해 끌어와서 사용함으로써 이를 해결하는 것이다. 해당 방식은 Hardware의 size가 커진다는 단점이 있지만, stall의 양이 현저하게 줄일 수 있다는 장점이 있기 때문에 해당 방식이 많이 사용된다.

1. Result
2. Code Dependency





위의 코드와 그림은 각 명령어별로의 Code Dependency를 표시한 것이다. 그림으로 확인할 수 있다시피 코드 상에서는 이전 명령에서 write한 명령어를 사용하려는 경우가 많기 때문에 이러한 과정에서 stall이나 forwarding 없이 코드를 돌릴 경우 hazard로 인한 원하지 않은 결과가 나올 확률이 높다.

1. S/W Forwarding

다음 코드는 4-stall code에서 stall를 줄임으로써 cycle을 줄인 것이다. software forwarding.txt에 저장되어 있으며 각 stall이 해당 개수인 이유는 다음 주석을 통해 확인할 수 있다.

main:

lui $6,0x0000

nop

nop

nop

ori $6,$6,0x2000

/\*ori에서 $6을 접근하지만 lui에서 사용했던 부분을 접근하는 것이다. 때문에 ori가 ID 상태일 때 lui는 pipeline을 통과해야 한다. 때문에 3번의 stall이 사용되었다. \*/

addi $2,$0,0x10

addi $3,$0,0

/\*addi $2, $0, 0x10과 addi $3, $0, 0 사이에는 code dependency가 없기 때문에 stall이 없어도 상관이 없다. 이후에 나오는 sub의 경우 $2와 $3가 사용되기 때문에 아래 3개의 stall이 있다. \*/

nop

nop

nop

L1:

sub $4,$2,$3

nop

nop

nop

addiu $4,$4,-1

/\*addiu는 sub에서 write한 $4를 사용하기 때문에 3 stall이 추가되어 있다. \*/

addi $5,$0,0

/\*addi와 addiu는 code dependency가 없다. \*/

nop

nop

nop

L2:

sll $7,$5,2

/\*addi의 $5를 sll에서 사용하기 때문에 3 stall이 필요하다. \*/

nop

nop

nop

addu $7,$6,$7

/\* addu에서는 sll에서 write한 $7을 사용하기 때문에 3stall이 필요하다. \*/

nop

nop

nop

lw $8,0($7)

/\* lw는 addu에서 write한 $7 값을 사용하기 때문에 3 stall이 필요하다. \*/

lw $9,4($7)

nop

nop

nop

slt $1,$8,$9

/\* lw 또한 pipeline을 빠져나와 다음 명령에서 사용되기 위해서는 3 stall이 필요하다. 때문에 slt사이에 3 stall이 삽입되어 있다. \*/

nop

nop

nop

bne $1,$0,L3

/\* bne는 ID에서 바로 수행되는 명령어다. 때문에 ID상태일 때 pipeline 안에 같은

Data hazard가 발생하면 안 되기 때문에 3 stall이 필요하다. 또한 이후에 나오는 명령의 경우 not taken일 때 IF의 명령어가 지워지기 때문에 nop이 필요하다. \*/

nop

add $10,$8,$0

add $8,$9,$0

/\* add 2개간의 code dependency가 없기 때문에 붙어서 사용할 수 있다. \*/

nop

nop

add $9,$10,$0

sw $8,0($7)

/\*add와 sw의 $9, $8은 add로부터 이미 1cycle 멀기 때문에 2 stall을 넣었다. \*/

nop

nop

sw $9,4($7)

/\*sw의 $9도 마찬가지로 add로부터 1cycle 멀기 때문에 2 stall을 넣었다. \*/

L3:

addiu $5,$5,1

/\*sw와 addiu는 code dependency가 없기 때문에 stall이 없다. \*/

nop

nop

nop

slt $1,$5,$4

/\*slt의 경우 $5가 pipeline 상에 있기 때문에 3개의 stall이 필요하다. \*/

nop

nop

nop

bne $1,$0,L2

/\* bne는 ID에서 바로 수행되는 명령어다. 때문에 ID상태일 때 pipeline 안에 같은

Data hazard가 발생하면 안 되기 때문에 3 stall이 필요하다. 또한 이후에 나오는 명령의 경우 not taken일 때 IF의 명령어가 지워지기 때문에 nop이 필요하다. \*/

nop

addiu $3,$3,1

nop

nop

nop

slt $1,$3,$2

/\*slt의 경우 $3가 pipeline 상에 있기 때문에 3개의 stall이 필요하다. \*/

nop

nop

nop

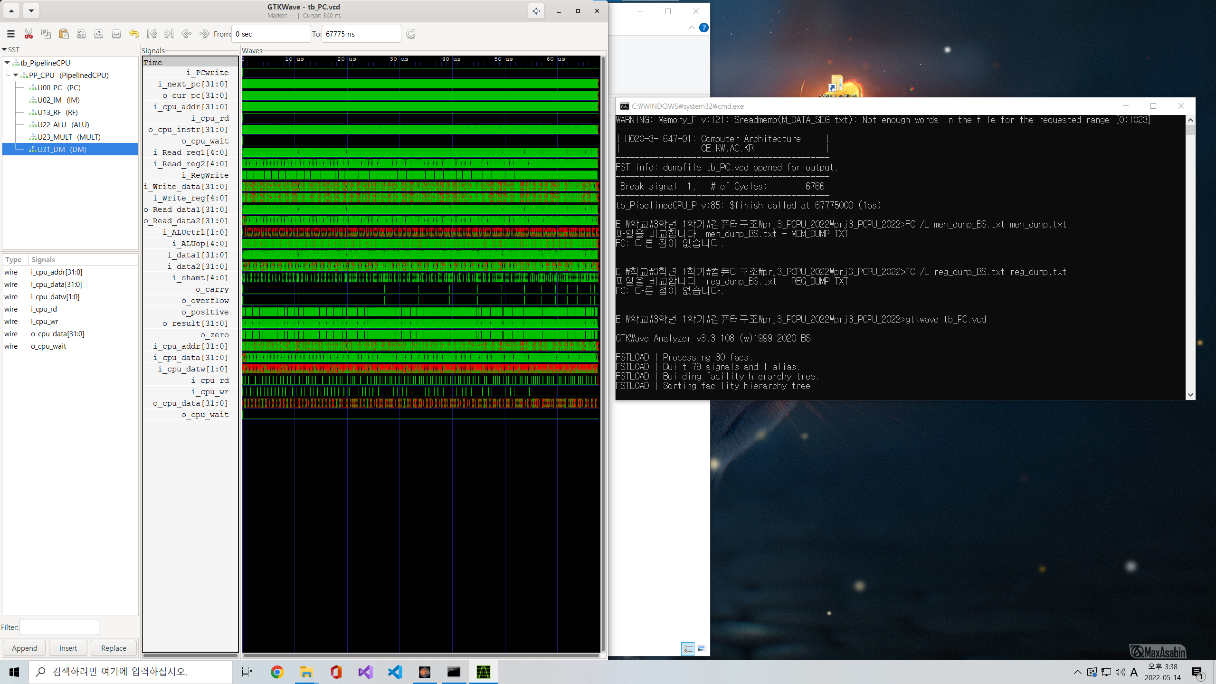
bne $1,$0,L1

/\* bne는 ID에서 바로 수행되는 명령어다. 때문에 ID상태일 때 pipeline 안에 같은

Data hazard가 발생하면 안 되기 때문에 3 stall이 필요하다. 또한 이후에 나오는 명령의 경우 not taken일 때 IF의 명령어가 지워지기 때문에 nop이 필요하다. \*/

nop

break



해당 사진은 4번의 stall만 있을 때의 사진이다. 보다시피 6766회의 cycle을 가지고 있는 것을 확인할 수 있다.

텍스트, 스크린샷, 전자기기, 컴퓨터이(가) 표시된 사진

자동 생성된 설명

해당 사진은 S/W Forwarding을 통해 최대한 stall을 줄인 코드의 실행결과다. 다음에서 볼 수 있다시피 결과는 기존의 코드와 동일하게 돌아가지만 Cycle은 4322로 배의 성능 향상이 이루어졌다. 이를 통해 stall을 줄이는 첫번째 과제가 정상적으로 동작하는 것을 확인하였다.

1. H/W Forwarding

다음 코드는 M\_TXT\_FWD.txt에 저장된 H/W forwarding을 해당 코드와 같이 주석으로 기록한 것이다. 또한 아래에서 주석을 통해 해당 코드를 작성한 이유를 서술할 것이다.

00\_00 // 0x000 addi $2, $0, 0x10

00\_00 // 0x004 addi $3, $0, 0

//기존에 lui, ori를 addi보다 먼저 사용하였지만, 순서를 바꿈으로써 0x14에 나오는 sub와의 code dependency를 없앴다.

00\_00 // 0x008 lui $6,0x0000

01\_00 // 0x00C ori $6, $6, 0x2000

//EX 단계일 때 lui의 MEM 단계의 $6을 사용한다.

00\_00 // 0x010 L1: addi $5, $0, 0

//addi는 아래 줄의 sub와 addiu과 상관이 없다. sub와 0x04번지와 3개의 cycle이 차이가 나면 nop이 필요가 없기 때문에 순서를 바꾸었다.

00\_00 // 0x014 sub $4, $2, $3

01\_00 // 0x018 addiu $4, $4, -1

// sub 명령의 MEM 단계의 $4을 사용한다.

00\_00 // 0x01C nop

//0x20번지는 L2로 label이 있기 때문에 해당 위치에 forwarding을 지정할 수 없다. 때문에 nop을 통해 0x10번지의 addi와의 사이의 cycle을 별려주었다.

00\_00 // 0x020 L2: sll $7, $5, 2

00\_01 // 0x024 addu $7, $6, $7

//sll 명령에서 MEM 단계의 $7을 사용한다.

01\_00 // 0x028 lw $8, 0($7)

//addu 명령에서 MEM 단계의 $7을 사용한다.

10\_00 // 0x02C lw $9, 4($7)

//addu 명령에서 WB 단계의 $7을 사용한다.

10\_00 // 0x030 add $8, $8, $0

//해당 명령은 기존에 없던 무의미한 동작을 하는 명령이다. 하지만 해당 명령이 필요한 이유는 다음과 같다. 이후에 slt에서 $8과 $9을 사용해야 하는데 이 때 $8이 pipeline 내부에 존재하도록 만들어 추가적인 stall 한 개를 덜 사용할 수 있기 때문이다.

01\_10 // 0x034 slt $1, $8, $9

//pipeline 상에서 add(MEM) lw(WB)의 데이터를 이용한다.

00\_00 // 0x038 nop

00\_00 // 0x03C nop

00\_00 // 0x040 nop

00\_00 // 0x044 bne $1, $0, L3

//bne의 경우 ID stage에서 일어난다. 해당 stage에서는 forwarding unit이 없기 때문에 3번의 stall이 필수적으로 필요하다.

00\_00 // 0x048 nop

//해당 하드웨어는 bne가 not taken이어도 이전의 명령을 지우기 때문에 nop을 사용해야 한다.

00\_00 // 0x04C add $10, $8, $0

00\_00 // 0x050 add $8, $9, $0

10\_00 // 0x054 add $9, $10, $0

//add 명령에서 WB 단계의 $10을 사용한다.

00\_10 // 0x058 sw $8, 0($7)

//add 명령에서 WB 단계의 $8을 사용한다.

00\_10 // 0x05C sw $9, 4($7)

//add 명령에서 WB 단계의 $9을 사용한다.

00\_00 // 0x060 L3: addiu $5,$5,1

//label의 경우 하드웨어 스팩 상 3번의 stall 후에 branch를 한다. 때문에 별도의 forwarding이 필요 없다.

01\_00 // 0x064 slt $1,$5,$4

//addiu의 MEM 단계의 $5을 사용한다.

00\_00 // 0x068 nop

00\_00 // 0x06C nop

00\_00 // 0x070 nop

00\_00 // 0x074 bne $1, $0, L2

//bne의 경우 ID stage에서 일어난다. 해당 stage에서는 forwarding unit이 없기 때문에 3번의 stall이 필수적으로 필요하다.

00\_00 // 0x078 nop

//해당 하드웨어는 bne가 not taken이어도 이전의 명령을 지우기 때문에 nop을 사용해야 한다.

00\_00 // 0x07C addiu $3, $3, 1

01\_00 // 0x080 slt $1, $3, $2

//addiu의 MEM 단계의 $3을 사용한다.

00\_00 // 0x084 nop

00\_00 // 0x088 nop

00\_00 // 0x08C nop

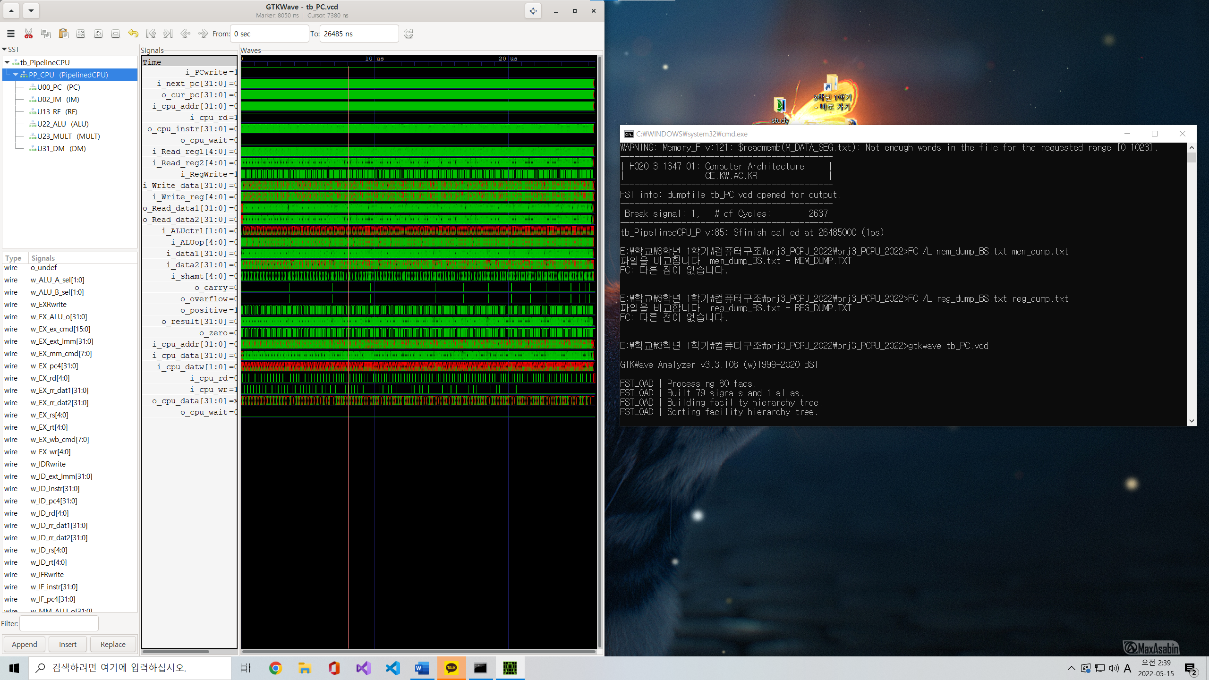
00\_00 // 0x090 bne $1, $0, L1

//bne의 경우 ID stage에서 일어난다. 해당 stage에서는 forwarding unit이 없기 때문에 3번의 stall이 필수적으로 필요하다.

00\_00 // 0x094 nop

//해당 하드웨어는 bne가 not taken이어도 이전의 명령을 지우기 때문에 nop을 사용해야 한다.

00\_00 // 0x098 break



해당 결과를 살펴보면 Memory와 register의 결과가 다르지 않기 때문에 동일한 결과를 보여준다는 것을 알 수 있다. 하지만 성능은 forwarding을 통해 줄인 cycle은 6676 -> 2637로 배의 성능 향상으로 결과가 개선되었다. 단순 S/W forwarding만 한 code와 비교해보아도 배의 성능 향상으로 결과가 나왔다. 이를 통해 성공적으로 forwarding이 수행되었다는 것을 알 수 있다.

1. Consideration

해당 과제를 통해 Pipeline을 통해 processor를 설계할 경우 위와 같이 여러가지 Hazard, 즉 문제가 발생할 수 있다는 것을 직접 확인할 수 있었다. 또한 이러한 문제가 결과에도 심각한 영향을 끼치는 것 또한 알 수 있었다. S/W의 forwarding을 통해 code 사이의 dependency를 파악하여 stall을 줄이는 방법을 고민할 수 있었고 이 과정에서 pipeline에 대한 이해도를 높일 수 있었다. 또한 forwarding 과정에서 forwarding에 넘어갈 신호를 고려하고 scheduling을 통해 순서를 바꾸어 stall을 줄이면서 더욱 높은 성능의 processor를 만들었다는 사실에 성취감을 느꼈다.

1. Reference

* 강의자료만을 참고