Multicycle & Pipeline

MIPS Simulator

Lab Work #3

32181928 박찬호

Dankook University

Mobile Systems Engineering

2022 Spring

Contents 목차

Ⅰ 서론 01

오늘날 Single Cycle은 왜 사용되지 않는가? 01

구현 목표 및 계획 01

실행 방법 02

Ⅱ 본론 03

Multi-Cycle & Pipeline 아이디어 03

효율성 향상 정도 예측 05

데이터패스 07

파이프라인 해저드 08

Branch Prediction 기법 10

디버깅 방법 12

소스파일 구성 13

소스 코드 설명 14

Ⅲ 결론 27

결과 분석 27

회고 29

서론

지난 Lab Work 2 를 통해 Single Cycle MIPS Simulator 를 구현한 바 있다. 그러나, 현대적 CPU 설계에서는 Single Cycle 을 더는 사용하지 않는다.

오늘날 Single Cycle 은 왜 사용되지 않는가?

분명 Single Cycle 구현은 그 자체로 올바르게 작동하고, Multi-Cycle & Pipeline 설계에 비해 이해하기 쉬운 구조를 가지고 있다. 그러나, Single Cycle 은 Clock Cycle 이 모든 명령어에 대해 같은 길이를 가져야 한다는 단점을 가지고 있다. 즉, 최장 길이의 경로를 갖는 명령어에 의해 Clock Cycle 이 결정되고 이로 인해 더 짧은 경로의 명령어는 지연되는 시간만큼 낭비가 생기게 된다.

예를 들어, LW 명령어와 JUMP 타입 명령어를 비교하여 생각할 수 있다. LW 명령어는 Instruction Memory, Register, ALU, Data Memory, Register 를 차례로 사용하는 긴 경로를 갖는데 반해, JUMP 타입 명령어는 (JR 을 제외하면) Data Memory 와 Register 에 접근할 필요가 없는 짧은 경로를 갖는다. 그러나 LW 의 경로를 기준으로 Clock Cycle 이 결정되고, 이는 JUMP 명령어가 한 Clock Cycle 을 다 사용하지 않는 지연의 이유가 된다. 이처럼 Single Cycle 구현은 모든 명령어에 대해 최악의 지연을 가정하는 설계라는 단점을 가진다.

따라서, 현대의 CPU 는 Pipelining 이라는 기술을 이용하여, Multi-Cycle 구현을 완성한다. 본 보고서에서는 이러한 설게 방식을 자세히 살펴보고, 시뮬레이터로 어떻게 코드 구현을 완성하였는지 설명하도록 하겠다

구현 목표 및 계획

Multi-Cycle MIPS Datapath 를 바탕으로 각 component 를 코드로 구현하고 알맞게 연결하여 Multi-Cycle Simulator 를 완성한다.

1. Multi-Cycle Datapath 를 바탕으로 component 를 코드로 구현한다.
2. 각 Stage 의 latch 를 구현하고, stage 를 구현한다.
3. Stage 의 input, output 으로 latch 를 붙여 Single Cycle 로 동작 시킨다.
4. Latch update 를 별도 구현한다.
5. 한 사이클에 각 stage 가 동시에 돌도록 pipeline 을 구현한다.
6. Pipeline 이 올바르게 동작하도록, Data Forwarding 과 Branch Not Taken 을 구현한다.
7. (Optional) Branch Predictor 를 구현하고, Predictor 를 다양하게 변경해가며 구현한다.

실행 방법

첨부된 파일을 Visual Studio 로 실행하면 실행 가능하다. 테스트 환경은 Windows 10 64bit & Visual Studio 2019 이다.

본론

Multi-Cycle & Pipeline 아이디어

앞서 밝힌 것과 같이, Multi-Cycle 과 Pipeline 은 Single Cycle 의 문제를 해결하기 위해서 고안되었다. 이 챕터에서는 Multi-Cycle 과 Pipeline 아이디어를 각각 설명하고, 두 아이디어가 어떻게 Single Cycle 의 문제를 해결하고 성능을 향상시킬 수 있는지 설명한다

1. Multi-Cycle

Multi-Cycle 은 명령어 실행을 여러 단계로 나누어, 여러 Cycle 에 걸쳐 실행하는 것을 말한다. 이 경우 나눈 단계만큼 CPI 는 증가할 수 있는데, 이는 명령어에 따라 사용하는 단계가 다를 수 있으므로 결과적으로 명령어마다 다른 CPI 를 갖는다. 즉, 명령어마다 필요한 Clock Cycle 수를 달리하고, 이를 통해 지연 문제를 해결할 수 있다.

2. Pipelining

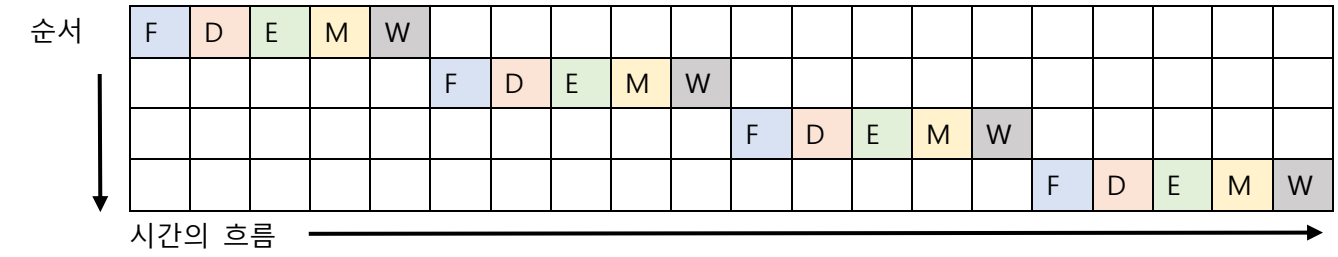
Pipeline 은 멀티 사이클의 낭비를 해결한다. 한 명령어가 Multi-Cycle 설계에서 매 Cycle 마다 여러 단계를 거쳐 실행될 때, non-pipelined 설계에서는 현재 실행 중인 단계를 제외한 나머지 단계들은 사용되지 않아 component 가 낭비된다. Pipelining 은 이러한 낭비를 해결한다.

좀 더 자세히 설명하기 위해, MIPS Pipeline 의 명령어 처리 단계를 먼저 제시한다.

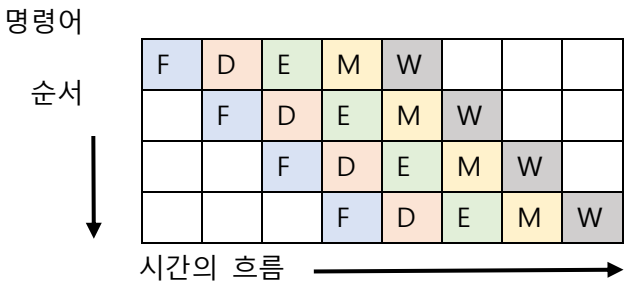
MIPS Pipeline 설계는 아래의 다섯 단계로 구성되어 있다.

1. Instruction Fetch (F)
2. Instruction Decode (D)
3. Execute (E)
4. Memory (M)
5. Write Back (W)

각 단계는 1. 명령어를 가져오고, 2. 명령어를 해독하고, 3. 연산을 수행하고, 4. 데이터 메모리에 접근하고, 5. 결과를 레지스터에 쓰는, 단계 이름 그대로의 기능을 수행한다. 만약, 다섯 단계를 모두 사용하는 연산을 네 번 연달아 실행한다면, 이러한 단계가 Non-Pipelined 설계에서는 아래와 같이 동작한다



그러나 Pipelined 설계에서는 단계들이 다음과 같이 동작한다.



F, D, E, M, W 각 단계에서 사용하는 component 들을 단계에 맞게 묶어 각각 F, D, E, M, W 라고 하자. 그리고 이 component 들이 중복 사용되지 않는다고 가정하자

(실제 MIPS 에서는 register 와 같이 여러 단계에서 공유하는 하드웨어 자원이 있고, 이는 멀티 사이클 설계의 장점으로 작용한다)

Non-Pipelined 설계에서는 F 단계에서 F 가 사용 중일 때, D, E, M, W 가 유휴자원이 된다.

그러나, Pipelined 설계에서는 F 단계에서 F 가 사용 중일 때, D, E, M, W 가 모두 상황에 따라 사용될 수 있다. 이는 한 Clock Cycle 에서 명령어의 병렬성을 통해 처리량을 증가시키게 되고, 결과적으로 Non-Pipelined 설계에서 명령어를 처리할 때 보다 적은 Cycle 만에 명령어를 처리하게 된다. 따라서, 전체 시간이 단축된다

효율성 향상 정도 예측

Multi-Cycle Pipelined 설계는 1) Multi-Cycle 로 Single Cycle 의 한 Cycle 안에서 낭비되는 시간, 즉 '지연'을 없애고, 2) Pipeline 으로 Multi-Cycle 의 낭비를 없앤다. 이러한 과정을 통해 Multi-Cycle Pipelined 설계는 효율성의 향상을 이뤄내는데, 과연 어느 정도의 효율성이 향상될까?

가상의 실험을 구성하고, 결과를 계산하여 효율성 향상 정도를 예측해보자

먼저 MIPS Pipeline 의 각 단계에서 필요한 시간을 가정한다.

1. IF : Instruction Fetch : 200ps

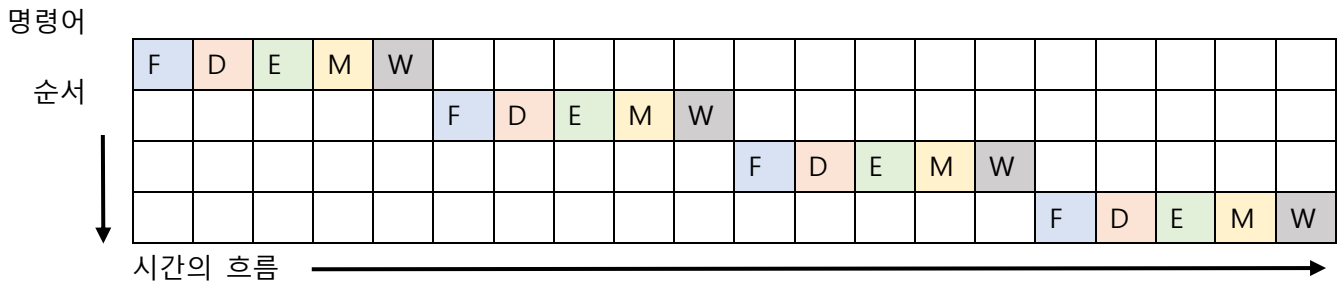
2. ID : Instruction Decode : 100ps

3. EX : Execution : 200ps

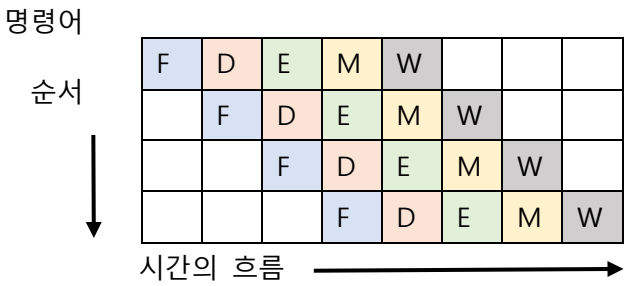
4. MEM : Memory Access : 200ps

5. WB : Write Back : 100ps

이 경우 Single Cycle 에서는 아래와 같이 명령어가 순차적으로 실행되고, 4 개의 명령어가 모두 실행되는 데에는 (200 + 100 + 200 + 200 + 100) * 4 = 3200 (ps) 만큼의 시간이 필요하다.



그러나 Pipelined 설계에서는 단계들이 아래와 같이 동작한다.



여기서 유의할 점은 ID 와 같이 조금 더 짧은 시간(100ps)이 요구되는 단계도 가장 긴 시간인 200ps 에 맞춰서 동작하게 된다. 또한, ID 와 WB 는 같은 register file 을 공유하므로, 한 Cycle 안에 두 Stage 가 다 실행될 수 있도록 한 Cycle 의 길이를 정해야 하는 것 역시 유의할 점이다. 다행히도, 본 실험에서는 register file 을 사용하는 두 단계 모두 100ps 로, 최장 시간인 200ps 의 절반이 되어, 낭비 없이 한 cycle 의 길이를 200ps 로 정할 수 있다.

따라서, Pipelined 구현은 200 * (5 + (4 - 1)) = 1600 (ps) 만큼의 시간이 소모된다.

위에서 사용한 식을 간단하게 정리해보면

- (Single Cycle 의 실행 시간) = (한 cycle 의 시간) * (명령어의 개수)
- (Pipeline 의 실행 시간) = (한 cycle 의 시간) * { (단계 수 - 1) + (명령어의 개수) }

  = (한 cycle 의 시간) * (명령어의 개수) + (한 cycle 의 시간) * 4

와 같다.

여기서 인할 수 있는 것은 명령어의 개수가 늘어날수록 '(한 cycle 의 시간) * 4' 부분은 무시할 수 있을 만큼 작아지고, pipeline 의 효율은 '한 cycle 의 시간 비율'에 가까워진다는 것이다. (4 : 1)

예를 들어, 10 억개와 같은 아주 많은 명령어를 실행한다고 가정하면, 전체 시간은 아래와 같이 변하게 된다.

Single Cycle = 800 * 1,000,000,000 = 800 billion (ps)

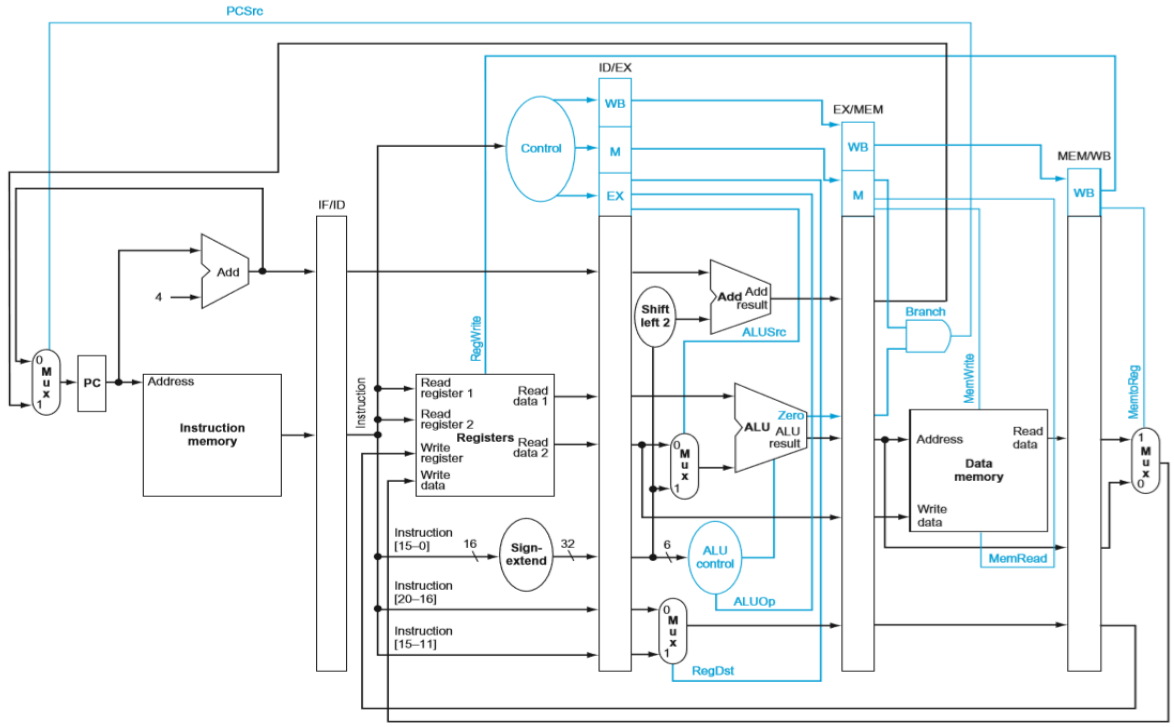
Pipeline = 200 * 1,000,000,000 + 200 * 4 ≒ 200 billion (ps)

이 값은 앞서 예측한 것과 같이 한 cycle 의 시간 비율에 가까워짐을 확인할 수 있다.

본 보고서의 결론에서는 이상적인 상황에서 효율이 얼마나 향상될지 예측한 위 결과와 최종 cycle 수에서의 결과를 비교할 것이다.

데이터패스

Multi-Cycle Pipelined 설계는 기본적으로 아래와 같은 Datapath 를 기준으로 한다.



그러나 위의 데이터 패스는 그대로 구현하기에는 몇 가지 문제를 가지고 있다.

1) Jump 등의 명령어를 어떻게 처리할지가 생략되어 있다.

2) Control Signal 을 어떻게 구성할지 추가적인 정보가 필요하다

3) 해저드 문제를 해결할 수 없다.

1)의 문제는 코드 구현을 통해 자세히 살펴보도록 하겠다. 2)의 문제는 아래의 signal 표를 통해 해결 가능하다.

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

가장 문제가 되는 것은 3)의 해저드 문제이다. 파이프라인은 다음 명령어가 다음 클럭 사이클에 실행될 수 없는 상황으로 세 가지 종류가 존재한다

파이프라인 해저드

1. 구조적 해저드

구조적 해저드는 하드웨어가 한 사이클 내에 실행되는 여러 명령어를 지원할 수 없을 때에 발생한다. 예컨대 위의 예제에서 Register에 읽고 쓰는 데에 걸리는 시간이 150ps라고 하고, clock cycle은 여전히 200ps로 유지한다고 생각하자. 이 경우, ID와 WB에서 동시에 register에 접근할 경우, 먼저 시작한 쪽이 끝나고 다른 쪽이 시작한 경우 50ps의 시간 밖에 남지 않는다. 이때는 register에 접근하는 과정을 마무리할 수 없다. 즉, 구조적 해저드가 발생한다. 또한, 메모리가 Instruction Memory와 Data Memory로 나누어 지지 않고 하나라는 가정을 해 보자. 이 경우, 동일한 Bus로 Memory에 접근하게 되므로 역시 구조적 해저드가 발생한다. 다행히도, MIPS 설계와 명령어는 구조적 해저드를 피하는 것이 용이하다. 본 보고서에서도 구조적 해저드를 해결하는 것에 초점을 두지 않는다.

2. 데이터 해저드

데이터 해저드는 어떤 명령어가 아직 실행 중인 앞선 명령어에 종속성을 가질 때 발생한다. 예를 들어, simple2.mips.asm의 pc가 0xc, 0x10일 때의 명령어를 확인해보면

c: 24020064 li v0,100 |||| 10: afc20008 sw v0,8(s8)

와 같이 0xc에서 v0에 100이라는 값을 넣고, 0x10에서 이 값을 메모리에 저장한다. 이 과정에서, 0x10의 명령어는 0xc의 결과를 필요로 하는데, 이는 0xc 명령어의 WB 단계 이후에 register file 읽기를 통해 확인할 수 있다. 즉, 데이터의 종속성으로 인해, 0x10의 명령어는 0xc의 결과가 register에 write back 될 때까지 기다려야 하는 것이다.

이와 같이 결과가 읽기 가능할 때까지 기다릴 경우, 종속성 문제를 피할 수 있다. 그러나 이러한 종속성은 자주 발생하게 되고, 이는 상당한 낭비로 이어진다. 성능을 향상시키기 위해, 데이터 해저드는 데이터 포워딩을 통해 해결할 수 있다.

데이터 포워딩은 어떻게 작동하는가? 먼저 우리가 필요한, 종속성 있는 데이터는 EX 단계에서 사용된다는 점에 착안한다. 물론, Jump 명령어를 Decode 단계에서 수행하는 경우, JR은 Decode

단계에서 종속성이 있는 데이터를 필요로 하기도 한다. 그러나 이와 같이 예외적인 상황은 실제 구현에서 해결하는 것으로 생각한다.

다시 돌아가서, EX 단계에서 종속성이 있는 데이터가 필요하다면 이전의 두 명령어, 즉, MEM 단계와 WB 단계에 있는 데이터에 종속성을 가진다. 이미 파이프라인 실행을 마친 데이터는 register 에 값을 업데이트한 후이므로, 데이터 해저드와는 연관이 없다.

이전의 두 명령어에서 값을 포워딩 하는 경우라면, 어떤 조건을 만족해야 할까? 먼저 이전의 두 명령어가 WB 단계에서 값을 저장하는지, Control Signal 을 확인해야 한다. 이전의 두 명령어가 WB 단계에서 register 에 쓰기를 하지 않는다면 값을 포워딩 할 필요가 없다. 이전의 두 명령어가 register 에 쓰기를 진행한다면, 쓰는 register file 의 인덱스가 현재의 명령어에서 필요한지 확인해야 한다. 이런 두 조건을 만족한다면, 이전의 두 명령어에서 register 에 쓰여질 값을 포워딩으로 현재의 EX 단계에 사용하면 된다.

이때 중요하게 고려할 부분이 있다. 첫째, 쓰여질 register 가 $0 이면 안된다. $0 는 항상 0 을 값으로 저장하고 있어야하기 때문이다. 둘째, 이전의 두 명령어 모두 register 에 값을 쓰고 현재의 명령어가 필요한 register 인덱스라면, 이 경우 더 가까운 명령어의 값을 받아와야 한다. 즉, MEM 단계에 있는 명령어가 필요하게 된다.

위의 조건에 맞게 데이터 포워딩 유닛을 추가하면 데이터 해저드는 해결된다. (lw 의 경우 포워딩과 함께 버블이 필수적으로 필요하나, MIPS 컴파일러가 버블이 필요한 경우 자동으로 nop 을 추가하기 때문에 별도의 조건을 추가할 필요가 없다.)

3. 제어 해저드

제어 해저드는 branch 명령어 이후 어떤 명령어를 실행하는지가 늦게 결정되고, 이로 인해 자원이 낭비가 생기는 경우를 말한다. 이는 branch 명령어가 실행될 경우, branch 가 taken 되는지 여부가 EX 단계의 zero 에서 결정되고, 그 주소 역시 EX 단계에서 결정되기 때문에 발생한다.

가장 단순한 해결책은 branch 명령어의 결과가 나올 때까지 지연하는 것이다. 그러나, 이 방법은 상당한 성능 하락을 초래한다. 만약, 현재의 구조에 다른 하드웨어를 추가하여 ID 단계에서 branch 의 결과를 알 수 있다고 가정하자. 이때, branch 명령어는 다른 명령어에 비해 하나의 clock cycle 을 더 필요로 한다. SPEC CPU2006 벤치마크에서 Integer 연산의 branch 명령어는 전체의 17%를 차지한다. 다른 명령어의 CPI 가 모두 1 이라고 가정하면, branch 명령어는 2 의 CPI 를 갖게 된다. 따라서, 전체 CPI 값은 1.17 이 되고 이상적인 경우와 비교했을 때 1.17 배의 속도 저하가 생긴다.

이러한 이유로 branch 명령어를 만났을 때 지연을 시키는 것은 좋은 해결책이 아니다. 더 나은 해결책은 branch 를 만났을 경우 1) Fetch 단계에서 빠르게 branch 인지 확인하고, 2) branch 가 taken 인지 예측하여, 3) taken 시 이동할 pc 를 빨리 아는 것이다.

1) 과 3) 의 문제는 BTB (Branch Target Buffer)라는 버퍼를 추가함으로써 해결할 수 있다. Branch 가 발생할 경우 해당 branch 의 pc 와 taken 시 업데이트 될 pc 를 buffer 에 저장하면, 다음 번에 동일한 pc 값에서 명령어를 fetch 할 경우 빠르게 branch 임을 알고 taken 시 이동할 pc 역시 빠르게 알 수 있다.

2) 의 문제가 바로 branch prediction 이라고 불리는 것으로, branch 가 taken 일지 not taken 일지를 예측하는 기법이다. 이 기법은 always taken, always not taken, backward taken forward not taken 과 같이 static 한 기법과 last-time prediction (one-bit), two-bit counter prediction, Gshare, Local, Hybrid 와 같은 dynamic 한 기법으로 나뉜다.

이 부분은 최초에 branch always not taken 으로 구현을 마친 후, optional 로 one-bit predictor 를 사용한 branch prediction 을 구현하고 predictor 를 교체하며 실험해보는 것으로 계획한다.

Branch Prediction 기법

이 장에서는 Branch Prediction 기법들에 대해 이해한 것을 정리하고 설명하고자 한다. 결과적으로 본 프로젝트에서는 always not taken 으로 구현하여 다양한 Branch Prediction 구현에 실패하였으나, 구현하고자 목표로 삼았던 기법들을 정리하면서 그 아쉬움을 달래기로 하였다.

1. Always Taken, Always Not Taken

   Branch 를 만날 경우 일관되게 Taken 또는 Not Taken 으로 direction 을 정하는 기법이다. Branch Stalling 보다 훨씬 더 나은 성능을 가지지만, Branch Prediction 이 틀렸을 경우 Branch 이후 실행되고 있는 명령어들, 즉, 이전 단계에서 실행중인 명령어를 flush 해야 한다. 가장 쉬운 Branch prediction 기법이다.

2. backward taken forward not taken

   loop 문이 있는 경우, looping 중에는 backward 방향의 taken 이 계속 일어나므로 backward 의 경우에는 taken 이라고 가정한다. 이 경우 높은 정확도를 갖는다.

3. last-time prediction (one-bit)

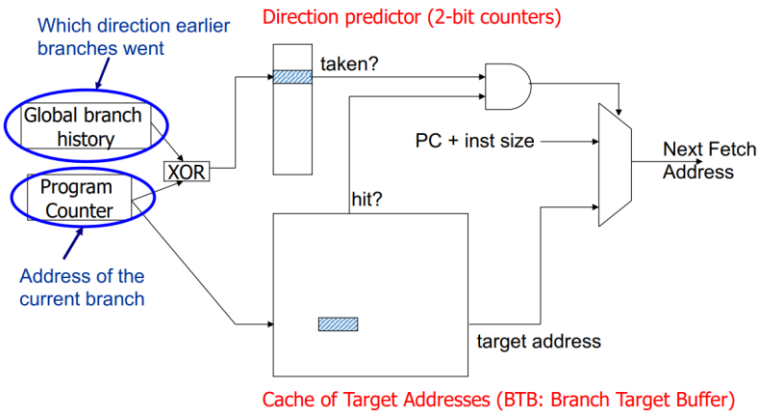
1, 2 의 방법 들은 branch 의 일반적 상황을 고려하여 static 하게 예측하는 기법이다. 3~7 의 방법은 각 branch 의 특성과 행동을 고려하여 dynamic 하게 예측할 수 있다. Last-time prediction(one-bit)은 branch 의 taken 과 not taken 을 one-bit 로 저장하여 다음 번 branch prediction 시 이전과 동일한 예측을 수행하는 기법이다. 이 방법은 순환문과 같은 경우 높은 정확도를 가지지만, 순환문이 끝나는 경우 한 번의 틀린 예측을 수행하게 된다. 또한 taken-not taken 이 반복적으로 발생하는 경우 계속 틀린 예측을 수행한다.

4. two-bit counter prediction

one-bit prediction 의 문제를 개선하기 위해 저장하는 bit 를 2-bits 로 늘린다. 이 경우 one-bit 의 장점은 가져오면서, one-bit 가 branch 의 taken-not taken 이 변화하는 상황에 대응하지 못하는 단점을 개선한다.

5. Gshare

여러 branch 간에 상호 연관성이 있을 경우, 한 branch 의 taken-not taken 결과가 이전의 branch 들에 영향을 미치는 경우가 있다. 이러한 Global 한 경우의 branch prediction 문제는 GBH(Global Branch History)를 통해 해결할 수 있다. GBH 는 GHR(Global History Register)에 global 하게 여러 branch 의 결과를 임의의 n 비트만큼 저장하고, n 비트의 인덱스를 갖는 PHT(Pattern History Table)에서 2 bit counter 로 예측한다. 이때, GHR 의 결과가 특정 패턴에 몰려 있거나, 동일한 패턴인데 branch 에 따라 결과가 다른 경우를 위해 XOR 을 진행한다. 이러한 global branch prediction 기법을 Gshare 라고 한다.



6. Local

   Gshare 와 같이 branch history 를 저장하고, patter table 에서 찾는 방식을 locally 하게도 적용 가능하다. 한 branch 의 결과를 계속 패턴으로 저장하고 동일하게 패턴별로 다른 2 bit counter 를 통해 예측하면 된다.

7. Hybrid

   Gshare 와 같은 global branch prediction 기법과 local branch prediction 기법을 다 사용하면서, 각 branch 마다 가장 높은 결과를 만드는 기법을 골라서 사용한다.

디버깅 방법

실제 구현 결과를 설명하기에 앞서, 본 구현에서 사용했던 다양한 디버깅 방법을 먼저 소개한다. 이는 현재의 프로젝트에서 얻었던 가장 값진 결과물들 중 하나로, 추후 다른 개발 과정에서도 도움이 될 자산이다.

1. Memory 를 관찰하는 디버깅

   1 의 방법은 본래 자주 사용하던 방법이다. 그러나, 이번 구현에서는 기존의 디버깅 시 메모리의 정적인 위치 (배열 등) 를 관찰했던 것과 달리, 실행에 따라 관찰해야 하는 메모리의 위치가 바뀌는 문제가 있었다. 이를 Visual Studio 의 Watch 부분에서 관찰해야 하는 메모리의 위치를 매 실행마다 동적으로 계산하고, 해당 위치를 관찰하는 것으로 문제를 해결할 수 있었다.

2. Output file 출력하여 스크립트로 비교하기

   현재의 프로그램에서 인풋 파일을 테스트하는 동안 register 등의 값이 빈번하게 변하고, 이를 사람의 눈으로 직접 추적하기는 상당한 끈기를 요구했다. 그렇다고 최종 결과만 비교하기에는 중간에 문제가 발생할 경우 정확한 문제점을 찾기가 어려웠다. 따라서 정확히 구현된 single cycle MIPS simulator 의 결과를 기준으로, branch 에 의해 pc 값이 변할 것 같은 경우에는 PC 값을 파일로 출력하여 비교하고, data dependency 에 의해 register 가 변할 것 같은 경우에는 register 를 파일로 출력하여 비교했다. 비교의 과정은 python 등의 스크립트를 이용하여 빠르게 문제가 발생하기 시작하는 지점을 찾을 수 있었다.

3. 인풋 파일 분석하기

   인풋 파일에 따라 결과가 어느 쪽은 맞고 어느 쪽은 틀리게 나오는 경우가 더러 있었다. 이때는 각 input 파일에서 쓰이는 명령어를 출력하여, 결과가 틀린 쪽에서 추가로 문제가 생길 수 있는 부분이 있는지 확인하였다. 또한, 데이터 플로우 구현 시에는 register 간에 종속성이 있는 부분이 pc 값을, branch prediction 구현 시에는 branch 명령의 pc 값을 미리 검출하여 디버깅 시에 활용했다.

4. 메모리 점유율 등 분석 도구 사용

   Visual Studio 에서 Diagnostic tool 를 다양하게 제공한다. 이 중 메모리 점유율 등의 분석 도구를 사용하면, 예상치 못하게 로직이 진행되고 있는지를 미리 확인할 수 있다. 예컨대, 로직이 알맞은 방향으로 흐르지 않아 동적 할당된 메모리가 해제 되지 않고 누적되는 경우 메모리 점유율이 비정상적으로 빠르게 증가하는 것을 관찰하고 문제가 있음을 알 수 있다.

5. 최종 결과 비교

   최종 결과를 single cycle MIPS simulator 의 결과와 비교하여 결과가 다른 테스트 파일을 디버깅하였다

소스파일 구성

파일의 구성과 구현 기능은 다음과 같다.

<alloc.h> <alloc.c> : 메모리 할당을 안전하게 할 수 있도록 malloc_s, multiple_malloc_s 함수를 만들어서 사용했다.

<alu.c> <alu.h> : alu 가 지원하는 연산을 함수와 매크로 함수로 구현했다.

<base_header.h> : 모든 영역에서 필요로 하는 기본 헤더를 묶은 헤더이다.

<components.c> <components.h> : adder, set control line, latch update, mux 등 데이터 패스에서 필요로 하는 기본 component 와 그 기능을 구현한 영역이다.

<controle_line.h> : EX, M, WB 등의 control line 을 구조체로 정의한 헤더이다.

<latch.h> : stage 간의 latch 를 구조체로 정의한 헤더이다.

<main.c> : 프로그램이 실행되는 main 함수이다.

<mips.c> <mips.h> : MIPS Pipeline 의 단계들과 Forwarding 등의 기능을 구현한 영역이다.

<mode.h> : Debug mode 를 define 으로 끄고 켤 수 있는 헤더이다. 기본 값은 debug mode 를 define 한 후 undefine 함으로써, debug mode 가 꺼져 있는 상태이다. Debug mode 가 켜질 경우 전체 프로그램에서 register 등의 값을 추적할 수 있다.

<predictor.c> <predictor.h> : branch prediction 을 구현한 영역이다. 결과적으로 branch prediction 구현에 실패하였다.

소스 코드 설명

소스 코드 설명은 전체 프로그램의 핵심적인 로직을 따라가며, 어떻게 각 단계가 실행되는지를 프로그램 로직 순서대로 설명한다.

```
int main(int argc, char* argv[]) {

    char* file_name_list[7] = {
        "./test_prog/simple.bin",
        "./test_prog/simple2.bin",
        "./test_prog/simple3.bin",
        "./test_prog/simple4.bin",
        "./test_prog/gcd.bin",
        "./test_prog/fib.bin",
        "./test_prog/input4.bin"
    };
```

프로그램의 실행과 함께, input file 의 배열이 생성된다. 전체 테스트 파일을 모두 순차적으로 테스트하기 위함이다. 각 파일 별로 for loop 를 돈다.

```
        unsigned int* const registers = initialize_registers();
        int num_of_instructions = 0, num_of_file_lines = 0;
        int* const memory = initialize_memory();
        FILE* fp = initialize_file(file_name, memory, &num_of_file_lines);
```

For loop 의 시작에 registers, memory, file pointer 등을 초기화한다.

Register 는 조건에 맞게 값을 채우고, memory 는 모두 -1 로 초기화하며, file pointer 에는 input file 을 읽어 들인다.

```
int count = 0, pc = 0, temp_pc = 0, add_result = 0,
    num_I = 0, num_R = 0, num_J = 0, num_branch = 0,
    num_register = 0, num_memory = 0;

bool PCSrc = false;
```

이후 실행에 필요한 값을 모두 초기화한다.

```
ForwardLine* line;

ForwardInput* f_input;

IF_ID* initial_if_id;

ID_EX* initial_id_ex;

EX_MEM* initial_ex_mem;

MEM_WB* initial_mem_wb;

multiple_malloc_s(6,
    &line, sizeof(*line),
    &f_input, sizeof(*f_input),
    &initial_if_id, sizeof(*initial_if_id),
    &initial_id_ex, sizeof(*initial_id_ex),
    &initial_ex_mem, sizeof(*initial_ex_mem),
    &initial_mem_wb, sizeof(*initial_mem_wb));

initial_if_id->valid =
    initial_id_ex->valid =
    initial_ex_mem->valid =
    initial_mem_wb->valid = false;
```

이어서, Data Forwarding 에 필요한 line, f_input, 그리고 최초 실행에 필요한 latch 등을 malloc_s 로 할당하고, valid bit 를 모두 false 로 설정해준다. 본 구현에서는 모든 latch 에 별도의 valid bit 를 두어, flush 나 nop 처리를 valid bit 를 이용하여 단순화했다.

```
IF_ID* if_id_latch[2] = { NULL, initial_if_id };
ID_EX* id_ex_latch[2] = { NULL, initial_id_ex };
EX_MEM* ex_mem_latch[2] = { NULL, initial_ex_mem };
MEM_WB* mem_wb_latch[2] = { NULL, initial_mem_wb };
```

초기 latch 값을 각 latch[1]에 대입한다. Latch 의 1 번 인덱스의 값을 각 단계 실행에 input 으로 사용하고, output 으로 다음 latch 의 0 번 인덱스에 저장하는 것으로 구상하였으며, latch update 는 모든 단계가 끝난 후 실행된다.

이후 본격적인 cycle 실행이 진행된다. 본격적인 설명에 앞서, 각 latch 의 형태가 어떻게 정의되었는지 확인하도록 한다.

```
typedef struct {
    int pc;
    int pc_plus_four;
    int instruction;
    bool valid;
} IF_ID;

typedef struct {
    WB_control_line* wb;
    M_control_line* m;
    EX_control_line* ex;

    int pc_plus_four;
    int* read_data;
    int sign_ext_imm;
    int inst_20_16;
    int inst_15_11;
    int opcode;
    int rs;
    int rt;
    int instruction;
    int pc;
    int funct;
    bool valid;
} ID_EX;

typedef struct {
    WB_control_line* wb;
    M_control_line* m;

    int add_result;
    bool zero;
    int alu_result;
    int read_data_two;
    int reg_dst_result;
    int pc;

    bool valid;
} EX_MEM;
```

```
typedef struct {

    WB_control_line* wb;

    int read_data;
    int alu_result;
    int reg_dst_result;
    bool valid;
} MEM_WB;
```

각 latch 는 기본적으로 데이터패스를 기준으로 필드를 구성하였으며, valid bit 를 가진다. 구현의 필요에 따라, 추가적으로 rs, rt, pc, funct 등의 값을 추가하였으며, 각 단계에서 사용을 마친 control line 은 데이터패스와 동일하게 다음 단계로 전달하지 않았다.

Control line 은 아래와 같이 정의되어 있다.

```
typedef struct {
    bool reg_write;
    bool mem_to_reg;
} WB_control_line;

typedef struct {
    bool branch;
    bool mem_read;
    bool mem_write;
} M_control_line;

typedef struct {
    bool reg_dst;
    bool alu_src;
} EX_control_line;
```

이는 미리 제시한 control line table 에 기초하여 작성하였다.

다시 main 의 로직으로 돌아가서, while 문 안에서 본격적인 실행이 시작된다. 매 사이클의 시작에서, data forwarding unit 에서 사용할 파라미터를 초기화한다.

```
            ForwardParameter forward_param = {
                .id_ex_rs = -1,
                .id_ex_rt = -1,
                .ex_mem_rd = -1,
                .ex_mem_regwrite = false,
                .mem_wb_rd = -1,
                .ex_mem_regwrite = false
            };
```

이후 각 단계를 실행하면서, 결과값을 forward parameter 와 latch, pc 등에 update 한다.

이때, 각 단계의 실행은 write back 부터 역순으로 진행한다. 이는 구현 상의 편의를 위해서이다.

실제 하드웨어에서 pipelined 설계는 같은 시간 안에 각각의 단계가 실행되지만, 이를 simulator 에서 구현하기 위해서는 multi thread 와 같은 비동기 처리를 필요로 한다. 이 경우 모든 단계에서 공통으로 접근할 수 있는 값을 다 차단하고, 오직 latch 만을 사용해야 하며, 먼저 종료된 단계가 다른 단계의 종료를 대기한 후 다음 사이클로 넘어가야하는 등의 여러 구현상 어려움이 존재한다. 따라서, 본 프로그램에서는 simulator 로서 제대로 동작하는 경우, 구현상의 편의를 추구하는 것으로 결정했다.

다시 역순 문제로 돌아와서, write back 부터 역순으로 단계를 진행시키면, data forwarding 문제에서 이점이 생긴다. 이는 write back 이 실행되면서 register 에 update 가 진행되어 write back 에서의 data forwarding 문제를 고려하지 않아도 되기 때문이다. (물론 실제 로직에서는 해당 부분의 구현을 추가했다).

또한 branch 나 jump 등 다음 pc 값이 pc+4 가 아닌 값으로 변하는 경우, 이후 단계에서 결정된 pc 값을 Instruction Fetch 에 바로 적용할 수 있어 한 cycle 을 아낄 수 있는 장점이 있다. 그러나 이는 소프트웨어 상에서 한 사이클을 아끼는 것으로 실제 MIPS 설계와는 무관한 부분이기 때문에 큰 의미를 갖진 않는다.

이제 각 단계의 실행 부분을 설명하겠다.

먼저 write back 단계이다.

```
f_input->write_data = write_back(mem_wb_latch[1], registers, &forward_param);
```

write back 의 실행을 위해서는 update 된 mem_wb latch 값이 들어가고, 추가로 register 와 forward param 을 인자로 추가했다. 이는 각각 register write 와 forward parameter 할당을 위함이다.

Write back 함수는 아래와 같이 구현되었다.

```
int write_back(MEM_WB* mem_wb,int* registers,ForwardParameter *forward_param){
    if ((mem_wb == NULL) || (mem_wb->valid == false)) return NULL;

    forward_param->mem_wb_rd = mem_wb->reg_dst_result;
    forward_param->mem_wb_regwrite = mem_wb->wb->reg_write;

    int write_data = mux(mem_wb->wb->mem_to_reg, mem_wb->alu_result, mem_wb-
>read_data);

    int write_register = mem_wb->reg_dst_result;

    operate_registers(registers, mem_wb->wb->reg_write, -1, -1,
write_register, write_data);

    return write_data;
}
```

먼저 모든 단계에서 공통적으로 latch 가 null 이거나 valid bit 가 false 인 경우를 검사한다. 이는 Latch 가 null 일 경우 null pointer 를 access 하는 경우나, valid 하지 않은 경우를 (flush 혹 nop 등) 확인하기 위함이다.

이후 forward parameter 를 update 하고, mux 로 register 에 쓰여질 write data 를 선택한다. 이후 해당 위치의 register 에 값을 write 한다

여기서 사용된 mux 와 operate register 는 아래와 같이 구현되었다

```
int mux(bool signal, int if_false, int if_true) {
    int return_value = signal ? if_true : if_false;
    return return_value;
}
```

Mux 는 signal, if_false, if_true 를 인자로 받아, signal 값에 따라 알맞은 값을 return 하는 간단한 함수이다.

```
int* operate_registers(int* registers, bool RegWrite, int read_register_1,
int read_register_2, int write_register, int write_data) {
    if (RegWrite) {

        registers[write_register] = write_data;

        return NULL;
    } else {

        int* read_data = malloc_s(sizeof(*read_data) * 2);

        read_data[0] = registers[read_register_1];
        read_data[1] = registers[read_register_2];

        return read_data;
    }
}
```

Operate registers 함수는 read register 와 write register 모두에 쓰이는 함수이다. RegWrite 일 경우 register 에 쓰기를 진행하고, 아니라면 register 에서 읽기를 진행한다. 이때, register 로부터 읽는 과정은 decode 단계에서 무조건 두 개의 register 읽고 두 개의 data 를 반환하므로, read data 라는 배열을 할당하여 사용한다.

```
f_input->write_data = write_back(mem_wb_latch[1], registers, &forward_param);
```

이렇게 write back 단계를 마치게 되면, 그 값은 f_input 의 write data 에 저장된다. f_input 은 forwarding unit 에서 도출된 forwarding control line 과 함께 execution 단계의 input 으로 들어가고, forwarding control line 의 값에 따라 data forwarding 의 source 로 사용된다.

이후 mem write 단계가 실행된다.

```
mem_wb_latch[0] = mem_write(ex_mem_latch[1], &PCSrc, memory, &add_result,
 &(f_input->pre_alu_result), &forward_param, NULL);
```

Mem write 단계에서는 ex_mem latch 를 받아 값을 업데이트 한다. 이때 PCSrc, Memory, add result 등의 변수 역시 받아 업데이트 한다.

```
MEM_WB* mem_write(EX_MEM* ex_mem, bool* PCSrc, int* memory, int* add_result,
int* pre_alu_result, ForwardParameter* forward_param, int* btb) {

    // valid 부분, 변수 update 부분 생략

    bool branch = and_gate(ex_mem->m->branch, ex_mem->zero);
    *PCSrc = branch;

    int read_data = data_memory(ex_mem->alu_result, ex_mem->read_data_two,
ex_mem->m->mem_write, ex_mem->m->mem_read, (char*)memory);

    MEM_WB* mem_wb = malloc_s(sizeof(*mem_wb));

    mem_wb->wb = ex_mem->wb;
    mem_wb->read_data = read_data;
    mem_wb->alu_result = ex_mem->alu_result;
    mem_wb->reg_dst_result = ex_mem->reg_dst_result;

    mem_wb->valid = true;


    return mem_wb;
}
```

Mem Write 함수 내에서는 위와 같이 Memory Access 기능과 branch 확인을 수행한다. Memory Access 는 data_memory 함수를 통해 이뤄지며, 이 함수는 ex_mem latch 로부터 전달받은 control signal 을 확인하여 memory 에서 값을 읽거나 쓰는 것을 수행한다.

Branch 확인은 branch 시그널과 zero signal 을 and 연산을 한 값을 통해 확인하고, 이 값이 true 라면 branch 연산이 taken 된 것이다. 현재의 구현은 always not taken 이므로 Taken 이라면 아래와 같이 flush 를 진행한다.

```
            if (PCSrc) {
                id_ex_latch[1]->valid = false;
                if_id_latch[1]->valid = false;
            }
```

최초 구현 시 위와 같이 always not taken 으로 구현하고, 추후 last time one bit prediction 을 predictor 와 btb 등이 서로 종속성이 없게 구현에 성공하면 predictor 만 다양한 기법의 것으로 교체해볼 생각이었다. 그러나 이 부분에서 여러 문제가 발생했고, 결과적으로 구현에 실패했다. 자세한 내용은 execution 부분에서 제시하겠다.

이후 Data Forwarding 에 필요한 값을 추가로 채운 후, Data Forwarding 을 수행한다.

```
            if ((id_ex_latch[1] != NULL) && id_ex_latch[1]->valid) {
                forward_param.id_ex_rs = id_ex_latch[1]->rs;
                forward_param.id_ex_rt = id_ex_latch[1]->rt;
            }

            line = forward(&forward_param, line);
```

forward 함수는 아래의 테이블을 그대로 구현한 것으로, forwarding parameter 의 값들을 비교하여, forwarding 의 조건에 맞을 경우 forwarding line 의 bit 를 변경한다. 이렇게 설정된 forwarding line 은 바로 직후의 execution 단계에서 바로 사용된다.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

```
ex_mem_latch[0] = execute(id_ex_latch[1], line, f_input, NULL);
```

execution 단계에서는 id_ex latch 와 forwarding line, f_input 을 받아 forwarding line 값에 따라 f_input 을 포워딩 받아 사용한다.

```
EX_MEM* execute(ID_EX* id_ex, ForwardLine* line, ForwardInput* input,
 int* const btb) {
    // validation 생략

    int sign_ext_imm_sl2 = shitf_left_two(id_ex->sign_ext_imm);

    int add_result = adder(id_ex->pc_plus_four, sign_ext_imm_sl2);

    bool zero = false;

    int funct = id_ex->sign_ext_imm & mask[5];

    int alu_input_a = id_ex->read_data[0];

    int b_mux_input = id_ex->read_data[1];

    int read_data_two = id_ex->read_data[1];
```

ex_mem 의 앞부분에서는 데이터 패스에서 제시된 것과 같이 전달받은 값을 변수에 할당하고, adder 를 통해 branch 시 이동할 주소를 미리 계산한다.

이후 forwarding line 을 확인하며, alu 연산에 input 으로 들어갈 값을 결정한다.

```
    if (line->ForwardA_bit_1 && !line->ForwardA_bit_0) { //10
        alu_input_a = input->pre_alu_result;
    } else if (!line->ForwardA_bit_1 && line->ForwardA_bit_0) { //01
        alu_input_a = input->write_data;
    }

    if (line->ForwardB_bit_1 && !line->ForwardB_bit_0) { //10
        b_mux_input = input->pre_alu_result;
        read_data_two = b_mux_input;
    } else if (!line->ForwardB_bit_1 && line->ForwardB_bit_0) { //01
        b_mux_input = input->write_data;
        read_data_two = b_mux_input;
    }
```

이어서 shift 연산의 경우 별도로 shamt 를 읽어내는 과정이 필요하게 되어, 이 부분에서 추가하였다.

```
    if ((id_ex->opcode == 0) && (funct == 0)) {
        int shamt = (id_ex->sign_ext_imm & mask[4]) >> shift_length[4];
        alu_input_a = shamt;
    }
```

앞서 변수 할당과 data forwarding 을 통해 결정된 input 값을 원래의 데이터 패스와 합쳐, control signal 에 따라 mux 로 input 을 결정한다.

```
    int alu_input_b = mux(id_ex->ex->alu_src, b_mux_input, id_ex-
>sign_ext_imm);

    int alu_result = alu(alu_input_a, alu_input_b, &zero, id_ex->opcode,
funct);

    int RegDst = mux(id_ex->ex->reg_dst, id_ex->inst_20_16, id_ex-
>inst_15_11);
```

이후 JAL 과 JR 연산을 지원하기 위해 별도로 예외적인 처리를 해주었는데, 이 부분에서 branch prediction 구현과 관련된 문제가 잇달아 발생했다.

```
    if (id_ex->opcode == 3) {
        RegDst = 31;
        alu_result = id_ex->pc_plus_four + 4;
    } else if ((id_ex->opcode == 0) && (funct == 8)) {
        add_result = id_ex->read_data[0];
        zero = true;
        id_ex->m->branch = true;
    }
```

최초 참고했던 데이터 패스에는 Jump 연산을 위한 부분이 생략되어 있었다. Latch 로 단계가 나눠져 있는 구조를 참고하기 쉽다고 생각하여, 수업 자료보다 우선적으로 해당 데이터패스를 사용하게 되었는데, 이때 jump 연산을 decode 영역에서 구현하고자 최초 시도했다. 이 경우, JR 과 JAL 구현이 어렵게 되었고, 이를 위해 Execute 영역에서 별도의 로직을 추가했다. 문제는 JR 에서 zero 와 branch 를 true 바꿔, 일종의 branch 처럼 구현하게 되었던 것에서 발생했다. 별도의 btb 나 prediction 없이 always not taken 으로 다음의 pc + 4 를 우선 실행하고, flush 하는

현재의 구조에서는 JR 이 문제가 없었으나, btb 를 사용하고 동적으로 branch prediction 을 진행하기 시작하자 JR 과 JAL 등의 수정이 불가피했고, 이후 수정을 거듭하며 점차 문제가 악화되었다. 결국 branch prediction 을 지우고 always not taken 으로 roll-back 하게 되었다.

이 내용은 회고에서 더 자세히 다루도록 하겠다.

다시 execution 으로 돌아가서, execution 을 위와 같이 마치고, 값을 latch 에 채워 반환한다.

이후 decode 를 수행한다.

```
id_ex_latch[0] = decode(if_id_latch[1], registers);
```

decode 에서는 opcode 를 찾고 opcode 에 맞는 control line 을 설정한다. 이후 HW2 에서와 같이, mask 와 shift length 를 사용하여 instruction 을 쪼개어 읽어내고, sing_ext_imm 과 zero_ext_imm 등의 값을 계산한다.

```
ID_EX* decode(IF_ID* if_id, int* registers) {
    // validation 생략

    int opcode = find_opcode(if_id->instruction);
    void** control_lines = control_unit(opcode);

    EX_control_line* ex = (EX_control_line*)control_lines[0];
    M_control_line* m = (M_control_line*)control_lines[1];
    WB_control_line* wb = (WB_control_line*)control_lines[2];

    int read_register_1 = (if_id->instruction & mask[1]) >> shift_length[1];
    int read_register_2 = (if_id->instruction & mask[2]) >> shift_length[2];
    int* read_data = operate_registers(registers, false, read_register_1,
read_register_2, -1, -1);

    short imm = (short)(if_id->instruction & (mask[3] + mask[4] + mask[5]));
    int sign_ext_imm = sign_extend(imm);
    int zero_ext_imm = ZERO_IMMIDIATE_FLAG & sign_ext_imm;

    if ((opcode == 0xc) || (opcode == 0xd)) sign_ext_imm = zero_ext_imm;
    if (opcode == 0xf) sign_ext_imm = (sign_ext_imm << 16);

    int inst_20_16 = (if_id->instruction & mask[2]) >> shift_length[2];
    int inst_15_11 = (if_id->instruction & mask[3]) >> shift_length[3];
    int funct = sign_ext_imm & mask[5];
// latch 할당 생략
```

이 영역에서 decode 를 통해 얻어진 결과를 받아 Jump 연산을 수행한다.

```
            if ((id_ex_latch[0] != NULL) && id_ex_latch[0]->valid) {

                // EXECUTE JUMP
                if ((id_ex_latch[0]->opcode == 2) || (id_ex_latch[0]->opcode
== 3)) {
                    int address = jump_address(if_id_latch[0]->instruction);
                    temp_pc = ((JUMP_ADDRESS_FLAG) & (pc)) + (address << 2);
                    id_ex_latch[0]->valid = false;

                    if (id_ex_latch[0]->opcode == 3) registers[31] = pc + 4;
                }
            }
```

Jump 연산에 맞는 opcode 를 확인하고, jump 라면 temp_pc 를 이용하여 다음 번 cycle 때 pc 가 바뀌도록 수정한 후, 현재 decode 에서 출력된 latch 를 invalid 로 바꾸어 이후 단계들이 실행되지 않도록 한다. 이때, temp_pc 를 pc 로 바꾸게 되면, jump 의 pc 값이 바로 다음의 fetch 에서 실행되게 되지만, 꼭 필요한 부분이 아니고 MIPS simulator 의 본래 동작에 맞게 temp_pc 로 구현하였다.

이후 fetch 와 latch update 를 수행한다. Fetch 는 nop 를 검출하여 valid 를 false 하는 부분을 제외하면, memory 배열에서 pc 위치의 값을 읽어들이는 방식이라서 설명을 생략한다. Latch update 역시 latch 0 번 인덱스의 값을 1 번 인덱스로 옮기는 코드이기 때문에 설명을 생략한다.

```
            if_id_latch[0] = fetch(&pc, add_result, &PCSrc, memory,
&count, NULL);
```

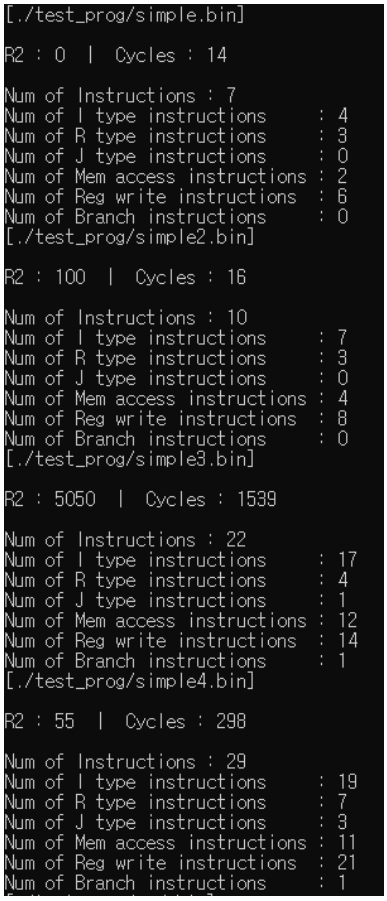
```
latch_update(if_id_latch, id_ex_latch, ex_mem_latch, mem_wb_latch);
```

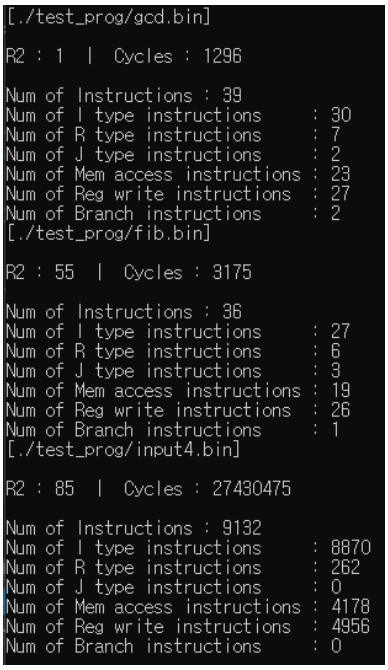
이로써 전체 코드에 대한 설명은 완료되었다.

결론

결과 분석

본 프로그램의 결과를 소개하고 분석하겠다.





결과는 위와 같이 출력 되었는데, 이를 Single Cycle 의 결과와 비교하도록 하겠다.

INPUT FILE	SAME RESULT	CYCLE-SINGLE	CYCLE-PIPELINE
SIMPLE	Yes	8	14
SIMPLE2	Yes	10	16
SIMPLE3	Yes	1330	1539
SIMPLE4	Yes	243	298
GCD	Yes	1061	1296
FIB	Yes	2679	3175
INPUT4	Yes	23372706	27430475

이 값을 이용하여 앞서 제시한 single cycle 과 pipeline 의 실행 시간 식에 대입하도록 하겠다. 다시금 그 식을 제시하면

- (Single Cycle 의 실행 시간) = (한 cycle 의 시간) * (명령어의 개수)
- (Pipeline 의 실행 시간) = (한 cycle 의 시간) * { (단계 수 - 1) + (명령어의 개수) }

  = (한 cycle 의 시간) * (명령어의 개수) + (한 cycle 의 시간) * 4

위와 같으며, 이때 single cycle 의 한 cycle 시간은 800ps, pipeline 의 한 cycle 시간은 200ps 라고 위의 실험과 동일한 가정을 유지하도록 하겠다. 또한, 명령어의 개수는 실행되는 명령어의 누적된 개수이므로 여기서 cycle 수와 같다고 생각한다. 이를 이용하여 위의 표를 실행 시간에 대한 표로 바꾸면 아래와 같다. 시간의 단위는 모두 ps 이다.

INPUT FILE	SAME RESULT	CYCLE-SINGLE	TIME-SINGLE	CYCLE-PIPE	TIME-PIPE
SIMPLE	Yes	8	6,400	14	3,600
SIMPLE2	Yes	10	8,000	16	4,000
SIMPLE3	Yes	1,330	1,064,000	1,539	308,600
SIMPLE4	Yes	243	194,400	298	60,400
GCD	Yes	1,061	848,800	1,296	260,000
FIB	Yes	2,679	2,143,200	3,175	635,800
INPUT4	Yes	23,372,706	18,698,164,800	27,430,475	5,486,095,800

그리고 각 시간을 비교하면, cycle 의 수가 늘어날수록 점차 한 cycle 의 시간 비율인 4 : 1 에 가까워 지는 것을 볼 수 있다. 이로써 최초 생각했던 실험에서의 예측이 옳았음을 확인할 수 있었다.

회고

이번 프로젝트를 수행하며 많은 어려움을 겪었고 몇 가지 중요한 교훈을 얻을 수 있었다.

1. 시간은 여유롭게 계획하고 시작하자

   HW1, HW2 에 비해 구현의 난이도가 올라감과 동시에 여러 가지 일이 겹치면서 마지막에 구현하는 데에 상당한 노력과 시간이 들었다. 조금 미리 설계하고 구현을 시작했으면, 다양한 branch prediction 실험이라는 궁극적 목표에 도달했거나 조금 더 일찍 동일한 결과로 마무리할 수 있었을 것이다. 이 부분이 가장 아쉽다.

2. 세세한 구조를 설계하고 구현에 뛰어들자

   최초 파이프라인 구현에 앞서 latch 를 포함한 single cycle 을 먼저 구현했고, 이 과정에서 참고한 데이터 패스의 여러 부분을 임기응변으로 해결했다. 즉, 생략된 부분이 있는 불완전한 데이터 패스를 참고하고, 이를 구조적인 설계로 메우지 않고 구현 상으로 해결했는데 그 결과로 새로운 기능 추가에 쉽게 흔들리는 구조를 만들게 되었다. 불안정한 구조는 계속 코드를 덧대며 해결할 수밖에 없었고, 그러한 코드가 누적된 후에는 다시 안정된 구조로 돌아가기 어려운 상황이 되었다. 이는 always not taken 까지는 잘 동작했으나, 결국 branch prediction 을 추가하면서 제대로 돌지 않는 코드가 되었다. 복잡한 프로젝트이거나 기능 추가에 안정적인 코드를 구현하려면 세세한 구조를 안정적으로 설계하고 뛰어드는 것이 필요하다는 것을 다시금 깨닫게 되었다.

3. 다양한 디버깅을 시도하자.

   프로젝트 초반에는 결과값 비교, 메모리 확인 등 손에 익은 방식으로만 디버깅을 시도했는데, 이는 파이프라인 이후 점차 동작이 복잡해지자 충분치 못한 방법이 되었다. 다양한 방식으로 결과를 출력하고 관찰할 수 있도록 디버깅에 있어서도 새로운 시도를 할 필요가 있음을 알게 되었다.

본 프로젝트를 진행하는 과정은 여태껏 수행했던 다양한 과제, 프로젝트 중 가장 힘들고 지난한 시간이었다. 스스로 용납되지 않는, 바람직하지 못한 얼기설기 코드를 몇 차례 뜯어 고쳐가며 원상복구 시켰고, 이따금 잘못된 구조에서 비롯된 문제를 확인하고 그 해결책을 깨달을 때마다 python 으로 다시금 구현하고 싶은 욕구를 억누를 수 없었다. 그러나 버티고 버텨, 부족하지만 결과물을 완성해내며 한 계단 더 성장하고 교훈을 얻어갈 수 있었다.

감사합니다