

Report

HW#3 Pipelined MIPS

Computer Architecture

Dpt. of Mobile System Engineering
32192530 양윤성

2023.06.18.

● Contents

1. Introduction	1p
2. Background	2p
2-1. Pipelining	2p
2-2. Performance Analysis	3p
2-3. Pipeline Design	4p
2-4. Hazards	5p
2-5. Data Dependency Handling	6p
2-6. Branch Prediction	8p
3. Implementation	12p
3-1. Data Path	12p
3-2. Definitions	14p
3-3. Latch (LAT.c)	15p
3-4. Hazard Detection Unit (HU.c)	17p
3-5. Forward Unit (FW.c)	19p
3-6. Branch Prediction (PRE.c)	21p
3-7. main.c	24p
4. Environment	28p
4-1. Build Environment	28p
4-2. Compile	29p
4-3. Working Proofs	32p
5. Lesson	40p

1. Introduction

Single Cycle은 한 Clock Cycle 에서 하나의 명령어를 처리하는 방식이다. 그렇기에 명령어가 Fetch부터 Write Back까지 모두 진행된 뒤에 다음 명령어가 실행된다. 그러나 Pipelining을 사용하면 여러 명령어가 동시에 처리가 가능하다. 이는 명령어들을 처리하는 작업을 여러 단계로 나누고 병렬적으로 진행하는 것을 의미하며 각 단계별로 서로 다른 명령어를 맡아 작업한다. 기존의 Single Cycle보다 명령어 처리 속도가 뛰어나고 이는 전체 시스템 성능을 향상에도 영향을 미치기 때문에 현대에는 거의 Pipelining을 사용하여 CPU를 디자인한다.

바로 이전 프로젝트에서 사용한 Single Cycle구조를 Pipelining하는 것이 Computer Architecture 수업의 마지막 프로젝트이다. 입력은 이전과 같은 방식으로, MIPS binary 파일을 읽어서 실행한 뒤 binary 파일의 정보를 메모리에 저장한다. 32비트의 명령어들이 Big Endian으로 저장되고, r0부터 ra(r31)까지 총 32개의 레지스터를 지원하는 점도 같다. 다만 명령어의 5가지 실행 단계(Fetch, Decode, Execution, Memory Access, Write Back)가 진행되는 동안 다른 명령어들도 실행된다. 따라서 각 Cycle별로 Fetch 단계부터 Write Back 단계까지 어떤 명령어들이 어떻게 처리되는지를 각각 확인할 수 있다. 이러한 특징이 이전 프로젝트와 가장 큰 차이이다. 명령어 처리는 PC값이 0xFFFF:FFFF가 되기 전까지 반복하며 최종 Return Value는 v0(r2) 레지스터에 저장된다.

이번 레포트에서는 Pipelining의 의미와 같이 Single-Cycle와 비교하여 얼마나 효율적인지를 소개하고, 이를 위해 필요한 Hazard와 Hazard 해결법 등과 같은 여러 배경지식들을 설명할 것이다. 이후 에뮬레이터의 전체 Data Path를 소개하고 본격적으로 Pipelining한 구현 코드를 결과 사진과 함께 안내한다. 또한 여러 Hazard를 감지하고 해결하는 방식의 성능을 직접 비교 및 분석해 본다. 마지막으로 개발 환경과 컴파일 과정에 대해 설명하고 총 3가지의 프로젝트를 진행하며 개인적으로 배웠던 점에 대해 말할 것이다.

2. Background

2-1. Pipelining

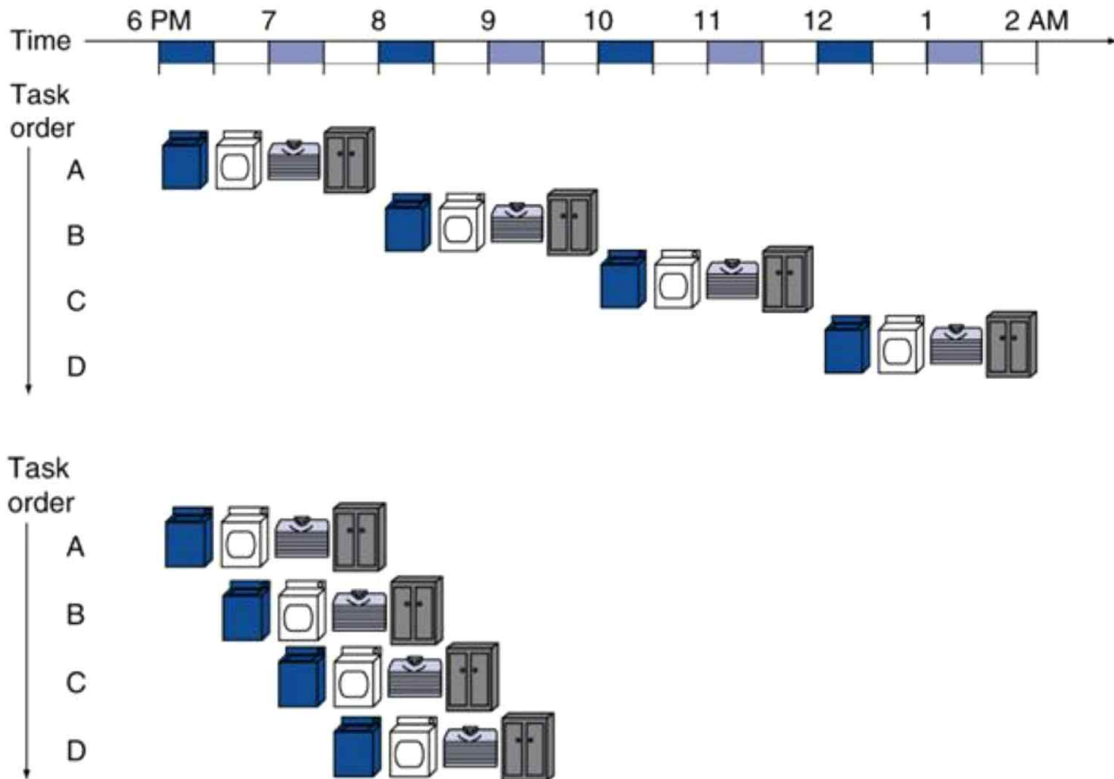


그림1. 예시로 보는 Pipelining 전과 후

위 그림은 세탁의 과정으로 보는 Pipelining 비교이다. 위의 상황이 Single-Cycle이다. A가 세탁, 건조, 정리를 모두 마쳐야 B가 세탁, 건조, 정리를 한다. 이런 방식이라면 A부터 D까지 모두 세탁을 완료하는데에 많은 시간이 소모되고, 비어있는 시간 동안 세탁기와 건조기 사용이 없어 낭비된다. 반면 아래의 경우 병렬적으로 세탁을 진행하여 A가 옷을 정리할 동안 B는 옷을 개고, C는 건조기를 사용하고, D는 세탁을 하고 있는 모습을 확인할 수 있다. 이런 방식이라면 A부터 D까지 세탁 완료에 소모되는 시간이 획기적으로 줄어들게 된다. CPU에서의 파이프라이닝도 이와 같다. CPU가 명령어를 처리하는 단계는 Fetch(IF) - Decode(ID) - Execute(EX) - Memory Access(MEM) - Write Back(WB) 순인데 위의 예시와 같이 비어있는 상태 없이 모든 단계가 작업하게 설계하면 된다. 다시 말해, 최대 5개의 명령어가 동시에 처리가 가능한 구조이다. 각 단계별로 하나의 Cycle이 제공되므로 5번째 Cycle부터 최대의 효율을 낼 수 있다.

2-2. Performance Analysis

Ex) $n=100, T=10$

(a) $T_s = 100 \times 10 = 1000\text{sec}$

(b) $k=5, T_p = (100+5-1) \times 10/5 = 208 \text{ seconds}$

(c) $k=10, T_p = (100+10-1) \times 10/10 = 109 \text{ seconds}$

그림2. Single-Cycle(T_s)와 Pipelined(T_p)의 처리시간 비교

파이프라이닝을 하지 않은 Single-Cycle 소요 시간을 T_s , 파이프라이닝을 했을 때의 소요시간을 T_p 라고 하자. 이때 n 개의 명령어를 처리하고 각 명령어는 T 만큼 걸린다고 가정하면 $T_s = n \times T$ 이고, $T_p = (n+k-1) \times T/k$ 이다. 이때 T_p 에서 k 는 스테이지(Latch)의 수, $k-1$ 은 파이프라인을 채우는데에 걸리는 소요 시간이라 가정한다. 그러면 둘의 Speed를 비교한다면 $S_p = T_s/T_p = (n \times k) / (n+k-1)$ 이고, 여기서 n 에 매우 커질수록 분모의 값의 의미없어질 정도로 작아지기에, 사실상 k 만 의미있는 값이 된다. 따라서 S_p 는 곧 k 가 됨을 생각할 수 있고 명령어 처리 속도는 k (스테이지 수)에 비례함을 알 수 있다. 가령 그림2에서 기존 Single-Cycle의 T_s 는 1000sec이지만, 5스테이지로 파이프라이닝을 진행하면 208sec가 되어 약 5배가 빨라지고, 10스테이지로 파이프라이닝을 진행하면 109sec가 되어 약 10배 빨라진다.

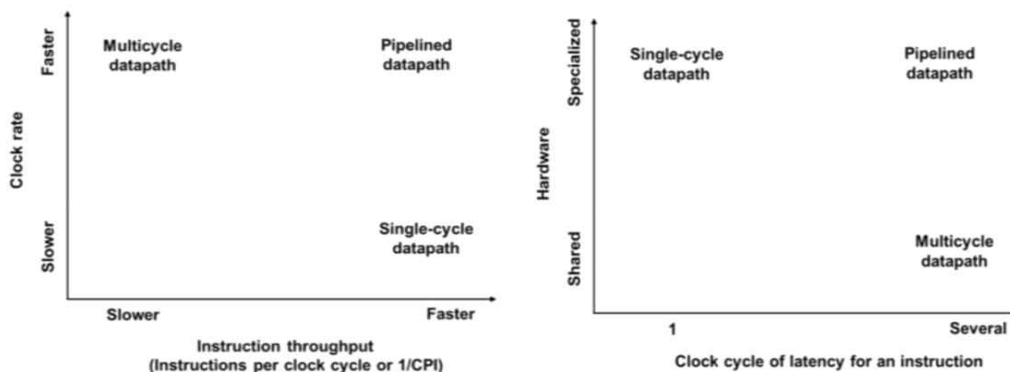


그림3. Single-Cycle과 Pipelined의 성능 비교

위의 그림3의 왼쪽과 같이, Pipelined Datapath는 Clock Rate와 CPI가 둘다 빠르다. 다른 DataPath들은 한 쪽은 특출나지만 한 쪽은 부족하다는 점과 대조되는 부분이다. 그러나 파이프라이닝이 장점만 있는 것은 아니다. 그림3의 오른쪽과 같이 Pipelined Datapath는 하드웨어적으로 더 복잡한 구조를 띠고 있으며 명령어 처리를 위해 요구하는 Cycle의 수가 많다는 단점이 있다.

2-3. Pipeline Design

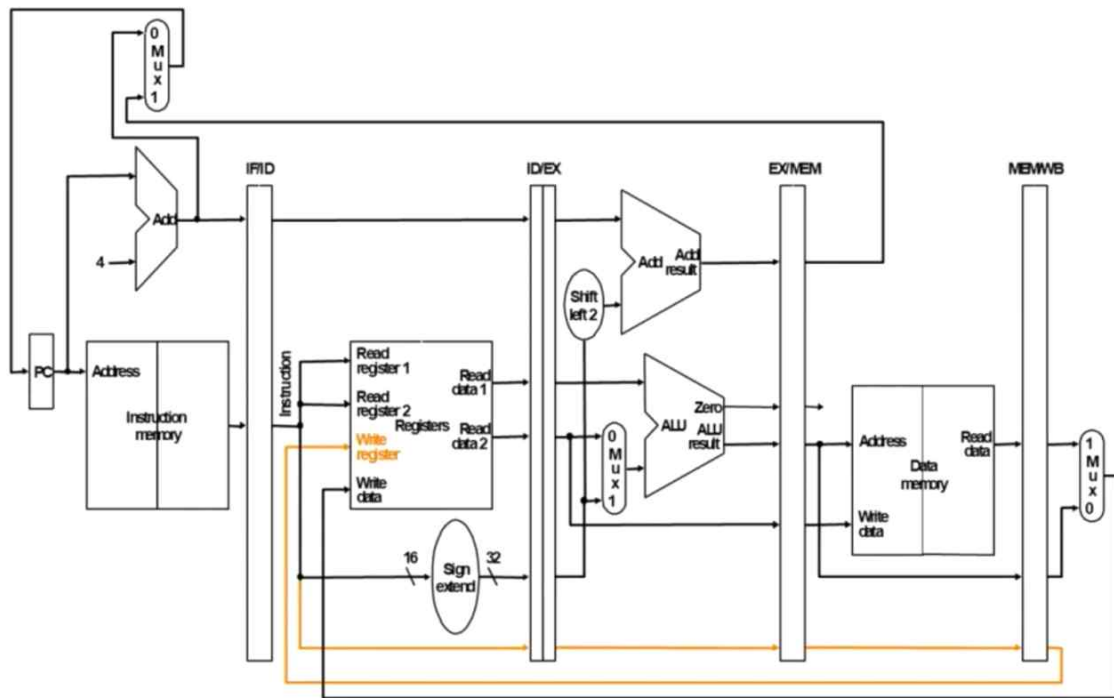


그림4. Pipeline 기본 Design

파이프라이닝의 기본 아이디어는 명령어의 실행을 작은 단계로 나누는 것이다. 단계를 나누면 각각 하나의 Cycle이 배정된다. 이러면 2-2에서 말했던것 처럼 Cycle의 수는 많아질 수 있지만 Clock Rate를 높이며 Cycle의 시간을 줄일 수 있다. 그러나 이 방법을 구현하기 위해서는 각 단계의 실행 상태를 기억해야만 한다. 매 Clock마다 Instruction Memory에서 새로운 명령어를 가져오기 때문에 기존에 처리하고 있던 명령어의 상태를 저장하는 레지스터가 필요하다. 이를 Latch라고 하며, MIPS Pipelining에서는 IF/ID, ID/EX, EX/MEM, MEM/WB 총 4가지의 스테이지로 파이프라이닝을 진행한다. 그리고 기존 Single-Cycle에서는 Decode단계에서 Write Register를 미리 정해줬지만, Pipelined Datapath에서는 WB 단계까지 가면 ID 단계에서 이미 다른 명령어를 처리하고 있다. 그러므로 위 그림처럼 Write Register가 정해지면 Latch를 통해 WB까지 전달해줘야 한다. 이렇게 하면 완벽할 것 같지만, 사실 파이프라이닝이 무조건적인 성능 향상을 보장해 주진 않는다. 일부 경우에 따라 종속성이나 실행 순서에 관련하여 명령어 처리에 문제가 생길 수 있다. 따라서 파이프라이닝은 설계와 구현에 상당한 고려가 필요하며, 효과적인 성능을 위해서는 후술할 2-4의 Hazard를 해결할 최적의 기법들을 적절히 사용해야 한다.

2-4. Hazards

파이프라이닝에서 문제가 발생하여 명령어들이 서로 간섭하거나 제대로 처리가 되지 않는 문제를 Hazards라고 하며, 이로 인해 파이프라인의 효율성을 감소시키거나 잘못된 결과를 초래할 수 있다. 파이프라이닝에서 주로 문제가 되는 Hazards는 아래와 같이 3가지가 있다.

① Structural Hazard

Instruction Fetch 단계에서도, Memory Access 단계에서도 각각 명령어의 Fetch와 메모리의 Load/Store을 위해 메모리 접근이 필요하다. 여기서 메모리가 하나라면 파이프라이닝 시 두 단계에서 동시에 메모리 접근을 하여 Memory Access 충돌이 발생할 수 있다. 이로 인해 파이프라인이 일시적으로 멈추거나 제대로 동작하지 않을 수 있다. 이것을 Structural Hazard라고 한다.

② Data Hazard

Data Hazard는 파이프라인 스테이지 간의 데이터 종속성 때문에 발생한다. 가령 현재 명령어를 처리하고 있는데 바로 이전의 명령어 실행 결과가 필요한 경우이다. 이렇게 현재 명령어가 이전 명령어를 의존하고 있는 상황을 의미한다.

③ Control Hazard

Control Hazard는 예외 처리나 분기 명령어, Jump같은 상황에서 발생한다. 우선 예외가 발생하면 현재 파이프라인에서 실행 중인 명령어를 취소하고 예외 처리 코드로 분기해야 하므로 제어 흐름이 바뀐다. 분기 명령어도 Branch Taken을 성공했을 경우 제어 흐름이 변경되고 Jump는 무조건 다른 주소로 이동한다. 이 3가지 경우 명령어 처리가 꼬일 수 있어 파이프라인이 깨지게 된다.

Structural Hazard는 IF와 MEM에서의 메모리를 분리하면 해결되지만, 나머지 두 경우는 처리하기 어렵다. 이들을 해결하기 위해 현재는 하드웨어나 소프트웨어에서의 다양한 기법과 최적화가 개발되었다. 이를 통해 Hazard를 최소화하고 파이프라인의 성능을 향상시킬 수 있다. 이제 2-5부터 Hazard를 어떻게 해결하는지 소개하겠다.

2-5. Data Dependency Handling

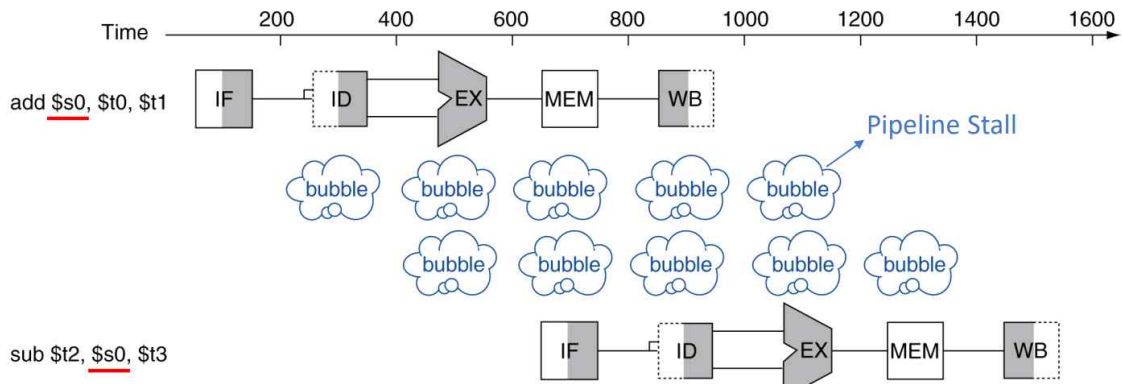
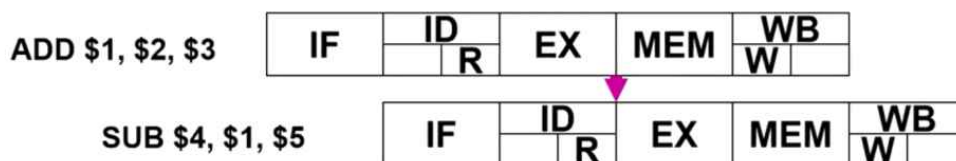


그림5. Freezing Pipeline

Data Hazard에서 데이터 의존성을 줄이기 위한 기법이 총 3가지가 있는데, 우선 Freezing Pipeline이다. Stalling이라고도 하는데, 말 그대로 뒤 명령어의 실행을 늦추는 것이다. 그림5와 같이 2번째 명령어를 실행하기 위해서는 1번째 명령어의 EX단계가 끝나야 하므로 기다린다. 이를 통해 다음 스테이지에서 데이터를 사용할 수 있도록 충분한 시간을 확보할 수 있다. 그러나 이 방법은 파이프라인의 성능을 크게 저하시킬 수 있다. 기존의 CPI를 1이라 하고, 위의 예시에서 Clock이 2만큼 지연됐는데 절반이 Freezing됐다고 가정하자. 그럼 변경된 CPI는 $1 + 0.5 \times 2 = 2$ 가 되어 버리므로 CPU 실행시간이 증가하도록 영향을 끼치게 된다. 따라서 다른 방법을 사용하는 것이 좋다.

• ALU result to next instruction (Stall X)



• Load result to next instruction (Stall 1)



그림6. Forwarding

2번째 방법인 Forwarding은 이전 스테이지에서 계산된 결과를 사용하여 다음 스테이지로 전달하는 방법이다. 파이프라인 구조에서 Forwarding을 진행해주는 장치를 Forward Unit이라고 한다. 여기서 Hazard를 감지하면 실행 중인 명령어의 결과값을 임시저장 후 다음 명령어가 필요할 시에 제공한다. 그림6처럼 명령어에 따라 EX의 결과값이나 MEM의 결과값 중 선택하여 재사용하게 되고, MEM의 결과값을 사용하게 되면 EX보다 한 Clock더 기다려야 하므로 1번 stall을 해준다. Forwarding은 대기 시간을 최소화하여 프로세서의 이용률을 높이고 실행 속도를 향상시킨다.

C code for

$A = B + E; C = B + F;$

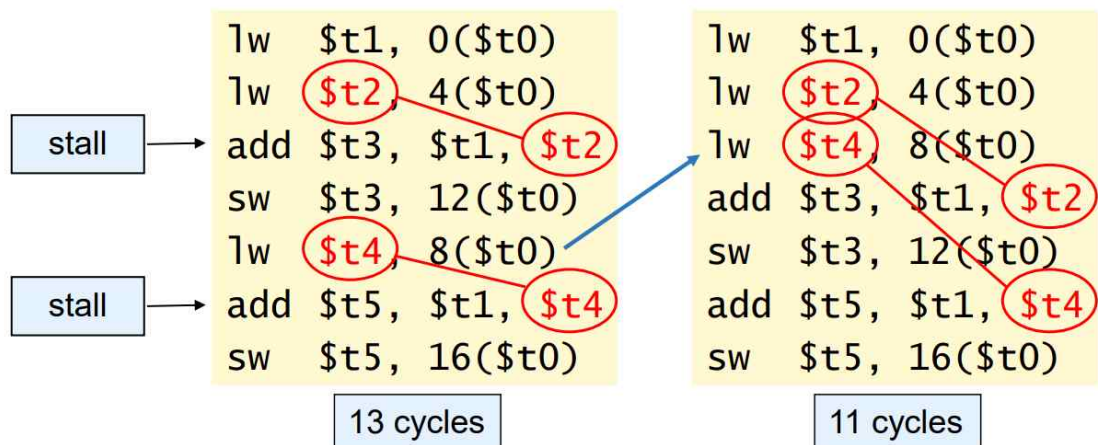


그림7. Compiler Scheduling

마지막 방법은 Compiler Scheduling으로 어셈블리 코드 자체를 수정하여 Hazard가 아예 발생하지 않도록 한다. 컴파일러가 코드를 수정하는 방법은 2가지가 있는데, 우선 코드에 nop을 삽입하는 방법이다. 단순한 수정이라 구현이 쉽지만 Cycle수가 늘어나기 때문에 실질적으로 많이 사용하는 방법은 그림7과 같이 명령어의 순서를 재배치하는 것이다. 컴파일러가 프로그램을 파악하여 데이터 종속성을 이해하고 분석하여 최적의 실행 순서를 만들게 된다. 순서의 기준은 데이터 종속성이 적은 명령어가 앞으로 온다. 그러나 이 방법은 하드웨어적인 구현 기술이 필요하기 때문에 이번 프로젝트에서는 데이터 종속성 해결을 위해 2번째 방법인 Forwarding을 사용하였다.

2-6. Branch Prediction

Control Hazard를 해결하는 방법은 Stall on Branch와 Branch Prediction이 있다. 가장 쉬운 방법인 Stall on Branch은 Data Hazard의 Freezing Pipeline과 유사하다. Branch할 주소가 결정되기 전까지 쭉 Stalling을 하는 것이지만 역시 CPI가 높아지고 CPU 실행속도에도 영향을 주기 때문에 추천하지 않는 방법이다. 사실상 Branch Prediction이 Control Hazard를 해결하는 주된 방법이다. Branch Prediction은 분기 명령어의 Taken유무를 예측하는 기술이다. 분기 명령어에 따라 제어 흐름이 계속해서 바뀌기 때문에 분기가 발생하는 목적지 주소를 미리 예측하여 대비한다. 이렇게 하면 예측된 목적지로 이동했더라도 파이프라인을 유지할 수 있고 실행 지연을 최소화한다. 얼마나 정확하게 예측하냐에 따라 프로세서의 성능이 크게 좌우되기에, 상황에 맞는 예측기법을 적절히 사용하는 것과 더 정확한 Branch Prediction 알고리즘을 고안하는 것이 중요하다.

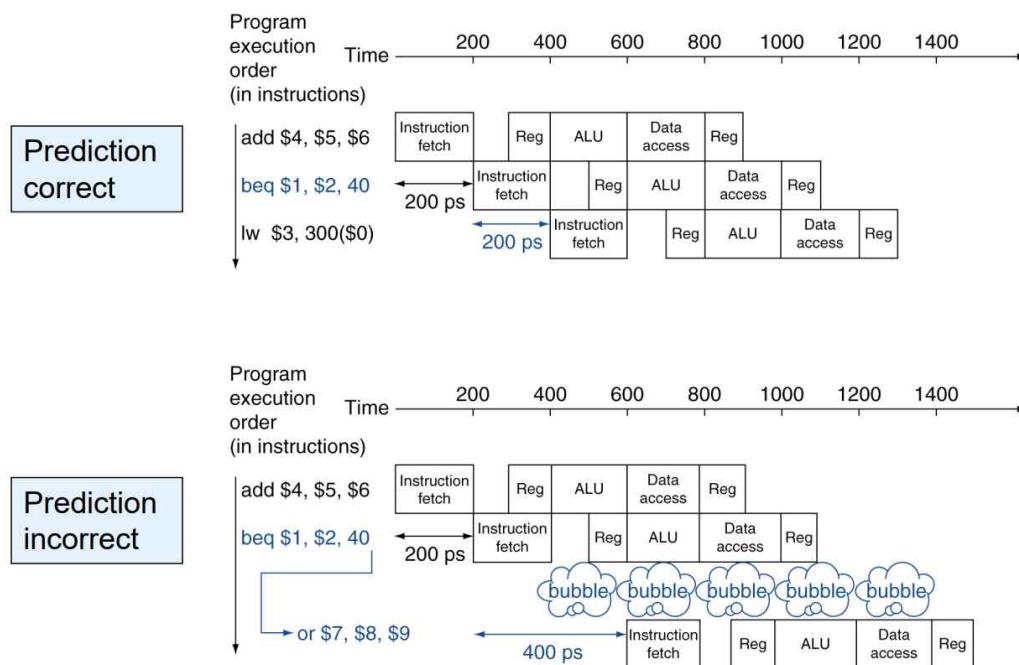


그림8. ANT에서의 예측 성공 시와 예측 실패 시 비교

가장 쉬운 예측 기법은 Static Branch Prediction 중 하나인 Always Not Taken(ANT)이다. 이는 분기가 항상 발생하지 않다고 가정하는 방법이다. 예측이 만약 틀렸다면 그림8 아래쪽처럼 다음번 명령어 처리를 1번 stall한다. 이와 반대되는 예측 기법은 Always Taken(ALT)로 분기가 항상 발생한다고 가정하고 예측하는 방법이다.

그러나 이 2가지 방법은 분기가 거의 발생하거나 발생하지 않는 극단적인 상황에 취약하다. 그러므로 Branch의 실행 패턴과 특성을 고려하여 적절한 분기예측 기법을 선택해야 한다. 그래서 프로그램의 실행 흐름을 분석하여 분기를 예측하는 Dynamic Branch Prediction이 유용하다. 이전에 처리된 분기 명령어의 실행 결과를 History에 저장하고, 이를 기반으로 예측하는 방식이다. 대표적으로 아래 4가지의 기법을 소개 하겠다.

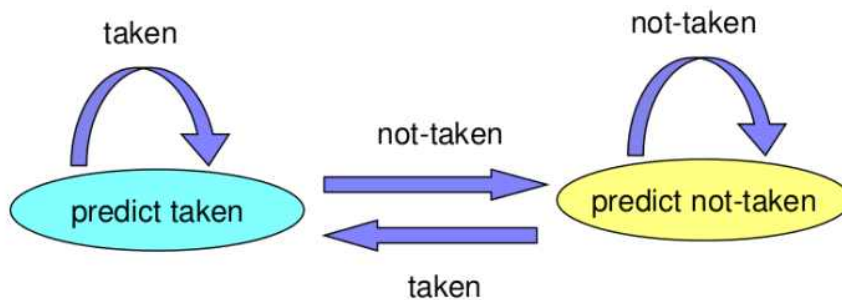


그림9. 1-bit Last Time Prediction 상태 변환

Dynamic Branch Prediction중 가장 간단한 방법으로 직전 명령어 주소에서 branch 여부 정보를 1-bit로만 저장한다. 그리고 다음번 Branch를 예측할 때 이전의 정보를 토대로 동일한 예측을 진행한다. 반복문 상황에서 Branch가 계속 발생하거나 발생 하지 않을 경우에 높은 정확도를 갖게 된다. 반대로 branch taken과 not taken이 반복되는 경우에는 정확도가 매우 떨어진다는 단점이 있다.

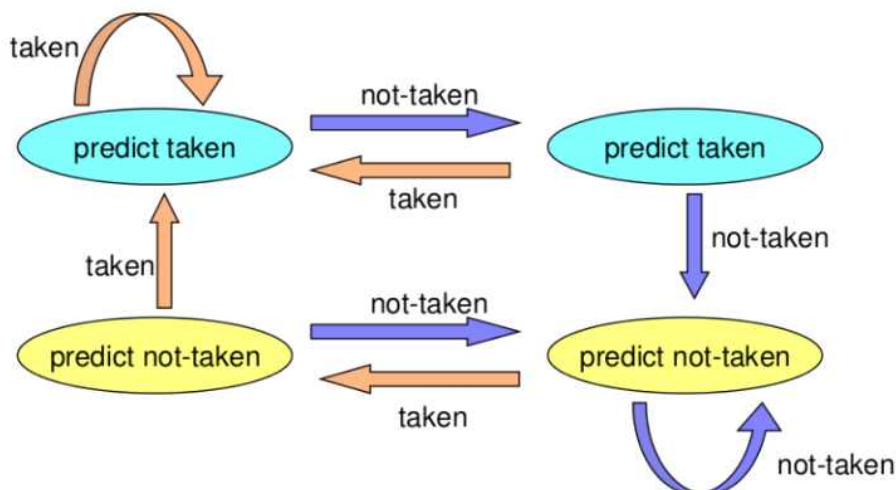


그림10. 2-bit Counter Prediction 상태 변환

이전의 1-bit 방법은 예측 정보를 하나의 비트에만 저장하므로 한계가 있었다. 이를 보완한 방법이 2-bit Counter Prediction이다. 예측 정보의 bit를 2bit로 늘리고, 각 비트는 Branch의 예측 상태를 나타낸다. state는 그림10에서 predict taken부터 시계방향으로 strongly taken, weakly taken, strongly not taken, weakly not taken 이다. 일반적인 초기값은 weakly not taken이고 Branch의 결과에 따라 state가 업데이트된다. 이렇게 단일 비트를 사용하는 경우에 비해 더 많은 예측 상태를 나타낼 수 있어 정확도가 올라간다.

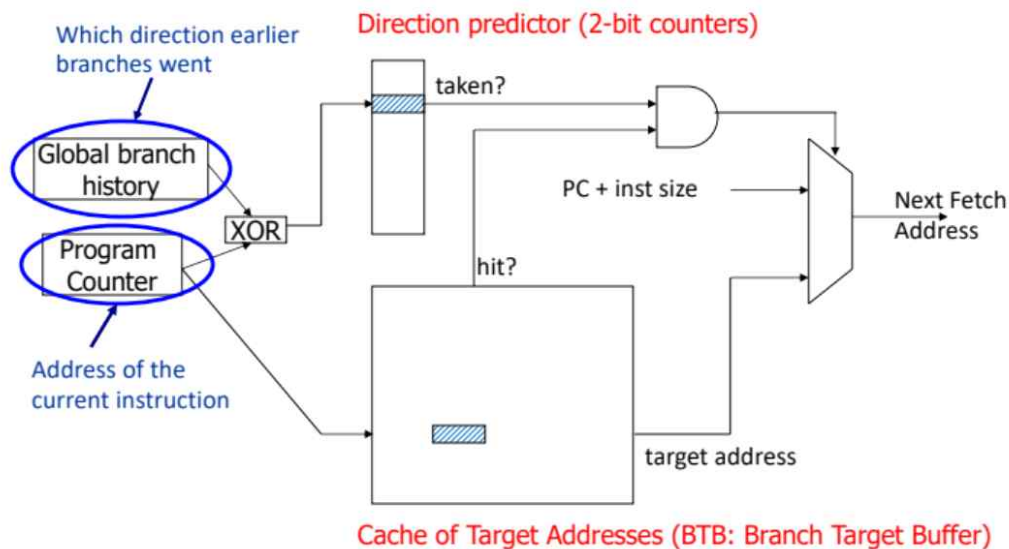


그림11. 2-Level Global Share Prediction

2-bit 방법에서 예측 정확도를 개선했지만, 여러 Branch의 결과들이 서로 상관관계를 가져 영향을 미치는 경우 성능이 떨어지게 된다. 이런 경우 전역 분기 예측을 사용한다. 우선 Global History Register(GHR)에 모든 Branch의 taken과 not taken 이력을 저장한 후 현재 Branch의 PC와 XOR연산을 진행한다. 이 연산 결과를 Pattern History Table(PHT)의 인덱스로 사용한다. XOR연산을 했기에 Branch끼리 인덱스가 겹치는 상황을 피할 수 있어 각 Branch들은 고유한 Index를 가지게 된다. 이 Index 값을 사용하여 PHT에 내장된 2-bit Counter Predictor로 Branch를 예측한다. 이런 방식을 2-Level Global Share Prediction(GShare)이라 한다.

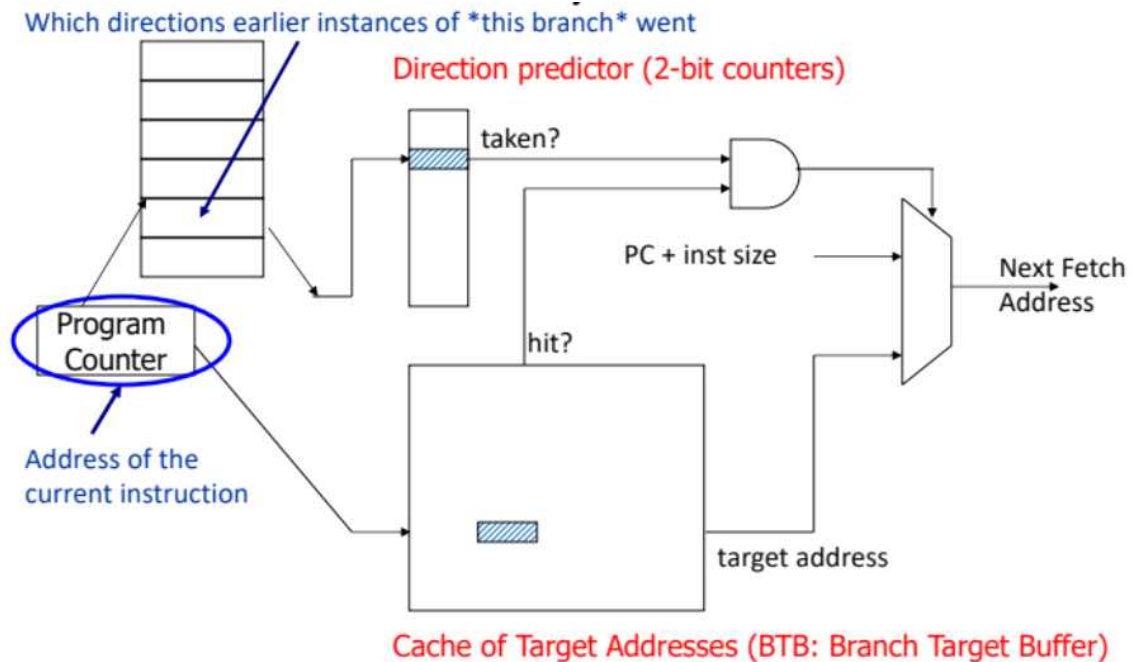


그림12. 2-Level Local Branch Prediction

또 다른 2-bit Counter Prediction의 보완 방법으로 2-Level Local Branch Prediction이 있다. 이 방식은 각 Branch마다 별도의 History 레지스터를 가져 독립적인 History를 관리할 수 있는 특징이 있다. 그래서 이 방식은 동일한 Local Branch History가 나타난 마지막 시점에서의 Branch 결과를 기반으로 예측을 한다. 이를 구현하기 위해서는 두 단계의 History를 사용한다. 첫번째는 Local History 레지스터 세트이다. 이 세트는 Branch의 PC값을 기반으로 해당 Branch와 관련된 History 레지스터를 선택한다. 두번째는 각 History별 Saturating Table이다. 이 테이블은 History마다 Branch의 예측 결과를 저장한다. 각 History들은 2-bit counter를 가지며 이 counter는 Branch의 예측 결과에 따라 업데이트된다.

이처럼 여러 Branch Prediction 기법들이 있고 이들은 여러 기준들을 고려한 후 예측 성능 요구사항에 가장 잘 맞는 방법을 선택한다. 비용, 전력소모, 복잡성 등의 기본적인 상황에 맞추는 것은 물론이고 프로그램의 분기 패턴 특성을 잘 파악해야 한다. 가령 일부 프로그램은 지역성(Locality)이 높아 GShare보단 Local Branch Prediction이 더 나은 성능을 제공할 수도 있는 것이다. 이번 프로젝트에서는 가장 기본적인 예측 방법인 Static Branch Prediction을 사용하여 구현하였다.

3. Implementation

3-1. Data Path

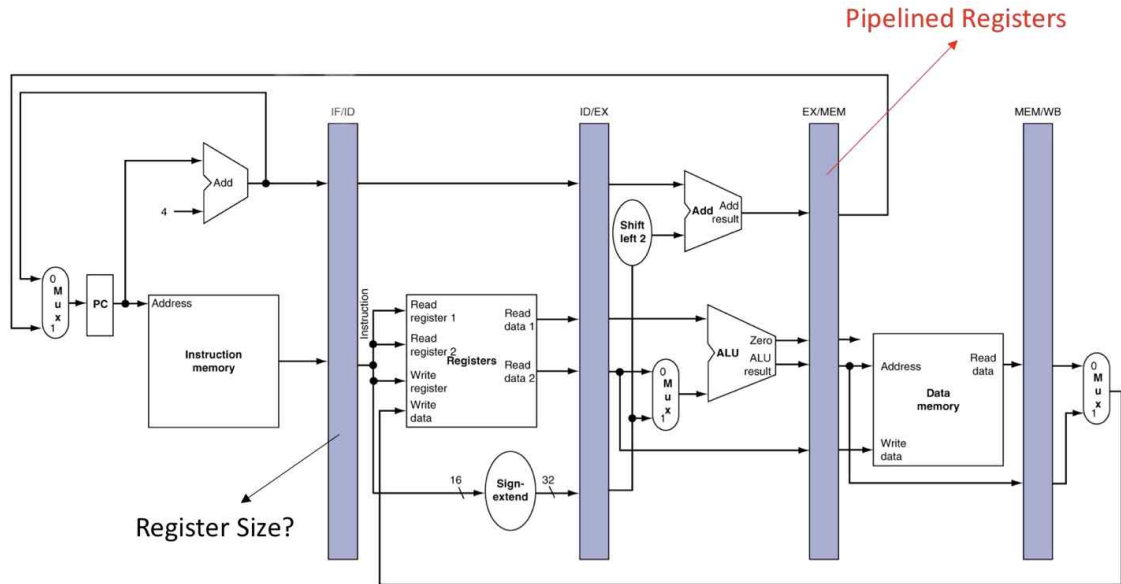


그림13. Simple Pipelined Datapath (04-4. Processor 3p)

Single-Cycle 구조를 파이프라이닝하기 위해서는 각 단계를 독립적으로 구분하여 관리할 일종의 레지스터가 필요하다. 이 레지스터로는 앞서 말한 Latch를 사용하며 이를 통해 명령어가 병렬적으로 처리된다. 그러나 위의 기본적인 파이프라인 구조에서 발생할 수 있는 대표적인 2가지의 문제가 있다. 우선 write back은 가장 마지막 단계인데, 이전 단계에서 이미 새로운 rd 레지스터의 정보가 갱신되어 register file에 저장된다. 이런 경우 올바르지 않은 write register에 write data가 작성된다. 따라서 IF/ID 단계에서 추출한 rd 레지스터의 정보를 바로 register file에 연결하는 것이 아닌, MEM/WB Latch까지 전송한 다음 register file에 연결한다. 2번째 문제는 Control signal이다. 본래 control signal은 ID단계에 위치해 있어 명령어가 다음 단계로 넘어가 버리면 올바른 control signal 정보를 받지 못한다. 그래서 ID단계에서 만든 signal들을 Latch들이 각자의 단계에 맞는 control signal만 사용하고 남은 신호들을 다음 Latch로 전달한다. 아래 13쪽에서 두 문제를 해결한 Data Path와 프로젝트를 위해 직접 설계한 파이프라인 Data Path를 확인할 수 있다.

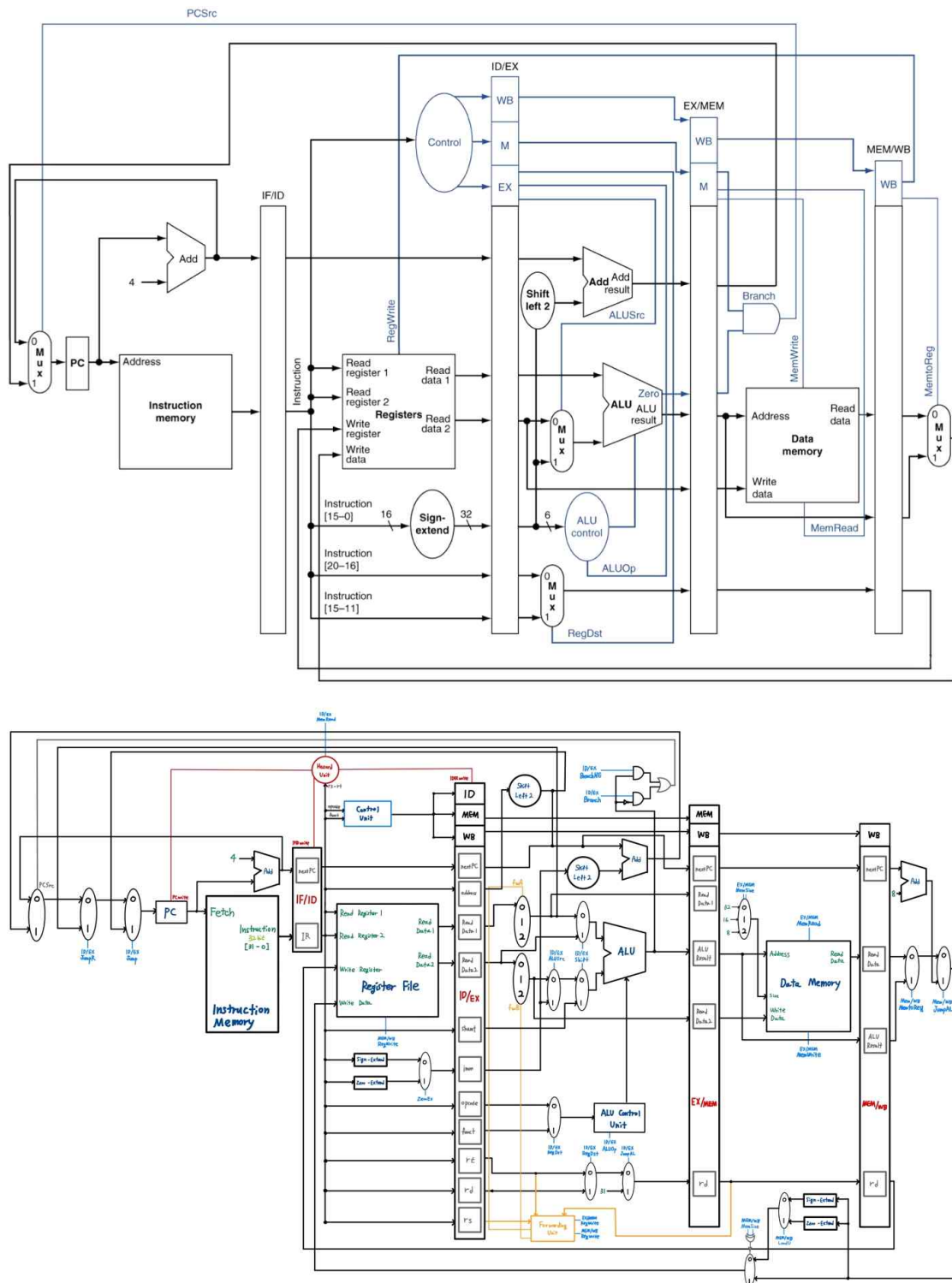


그림14. Pipelined 전체 Data Path (위는 기존, 아래는 직접 설계)

직접 설계한 Data path는 초안 설계이며, 프로젝트 코드와 달라진 점은 ALT 구현을 위해 IF/ID에 명령어 포맷을 저장할 공간을 추가한 점이다. 또한 branch predictor은 회로도 상에서 작성하지 않았다.

3-2. Definitions

레지스터 번호나 명령어의 매크로 지정, 변수의 unsigned_int 타입 사용, 변수 크기 지정 등은 Single-Cycle MIPS Emulator를 구현했을 때와 같으므로 설명을 생략한다. 다만 레지스터 번호와 명령어 매크로 지정 헤더파일을 따로 구분하였다. (instruction.h, register.h) 마찬가지로 이번 페이지에서는 뒷 내용을 이해하기 위한 주요 전역변수만을 설명하고, 구조체, Signal 관련 변수들과 function은 파트별로 자세히 설명할 것이다. Single-Cycle 프로젝트보다 사용하는 전역변수가 훨씬 줄어든 모습을 확인할 수 있는데, 이는 후술할 Latch에 모두 저장하는 방식이라 Latch 구조체 변수로 모두 이동하였다.

변수 이름	데이터 타입	설명
DH_DW_counter	int	Detect&wait로 Data Hazard를 해결하기 위한 stalled turn counter 변수
CH_DW_counter	int	Detect&wait로 Control Hazard를 해결하기 위한 stalled turn counter 변수
cycle	int	에뮬레이터에서 실행한 총 Cycle의 수
R_cnt	int	R Format 명령어 수
I_cnt	int	I Format 명령어 수
J_cnt	int	J Format 명령어 수
brc_cnt	int	Branch taken 횟수
success	int	Branch prediction을 성공한 횟수 (분석용)
fail	int	Branch prediction을 실패한 횟수 (분석용)
mem_cnt	int	메모리에 접근한 횟수
InsMemory	uint8_t[MAX_SIZE]	명령어들이 저장되는 배열
DataMemory	uint8_t[MAX_SIZE]	데이터가 저장되는 배열
Register	uint32_t[32]	r0~r31의 상태를 저장하는 배열
PC	uint32_t	Program Counter
IR	uint32_t	Instruction Register

3-3. Latch (LAT.c)

```
/* ID/EX Latch Formats */
typedef struct Latch_IDEX{
    bool valid; // 접근 허용/거절 bit
    uint32_t nextPC;
    uint32_t ReadData1;
    uint32_t ReadData2;
    uint8_t opcode;
    uint8_t rs;
    uint8_t rt;
    uint8_t rd;
    uint8_t shamt;
    uint8_t funct;
    uint32_t imm;
    uint32_t address;
    Signal_IDEX sig;
}Latch_IDEX;

/* EX/MEM Latch Formats */
typedef struct Latch_EXMEM{
    bool valid; // 접근 허용/거절 bit
    uint32_t nextPC;
    uint32_t ReadData1;
    uint32_t ReadData2;
    uint32_t ALUResult;
    uint8_t rd;
    Signal_EXMEM sig;
}Latch_EXMEM;

/* EX Control Signals */
typedef struct Signal_IDEX{
    bool Jump;
    bool JumpR;
    bool JumpAL;
    bool Branch;
    bool BranchNE;
    bool ZeroEx;
    bool Shift;
    bool RegDst;
    bool RegWrite;
    bool MemWrite;
    bool MemRead;
    bool MemtoReg;
    bool LoadU;
    bool ALUSrc;
    uint8_t ALUOp;
    uint8_t MemSize;
}Signal_IDEX;

/* MEM Control Signals */
typedef struct Signal_EXMEM{
    bool JumpAL;
    bool Branch; // 출력용으로만 필요
    bool BranchNE; // 출력용으로만 필요
    bool RegWrite;
    bool MemWrite;
    bool MemRead;
    bool MemtoReg;
    bool LoadU;
    uint8_t MemSize;
}Signal_EXMEM;
```

그림15. Latch 구조체와 Latch 별 Control Signal 구조체 (in base.h)

파이프라인 구조에서 핵심이 되는 Latch이다. 위 사진에서는 ID/EX, EX/MEM Latch 만 대표적으로 보여주고 있고, IF/ID와 MEM/WB Latch와 Control signal 구조체도 존재한다. 이들은 각자의 명령어 단계의 처리를 위해 일종의 저장 공간에서 데이터를 사용한다. 이 데이터들은 이전 Latch로부터 받아오거나, 몇몇 signal들로 인해 특정 값으로 업데이트 된다. 이 Latch들의 동작을 관리하는 함수들을 LAT.c에 작성했으며 다음 쪽에서 후술한다. 또한 Latch는 저장공간이 2개가 필요한데, 이는 각 단계의 결과를 앞쪽 공간에 임시로 저장한 뒤에 다음 단계로 전달해야 하기 때문이다. 따라서 각 Latch의 구조체 타입의 변수 4개를 일반적으로 선언한 것이 아닌, 아래와 같이 index가 2인 구조체 배열로 선언해 주었다.

```
Latch_IDEX IDEX[2];
Latch_EXMEM EXMEM[2];
```

```

void LAT_init(){
    // Latch valid를 일단 모두 false
    IFID[0].valid = false;
    IDEX[0].valid = false;
    EXMEM[0].valid = false;
    MEMWB[0].valid = false;

    // 레지스터 내의 요소들을 memset을 사용하여 모두 0으로 초기화
    memset(&IFID[0], 0, sizeof(Latch_IFID));
    memset(&IFID[1], 0, sizeof(Latch_IFID));
    memset(&IDEX[0], 0, sizeof(Latch_IDEX));
    memset(&IDEX[1], 0, sizeof(Latch_IDEX));
    memset(&EXMEM[0], 0, sizeof(Latch_EXMEM));
    memset(&EXMEM[1], 0, sizeof(Latch_EXMEM));
    memset(&MEMWB[0], 0, sizeof(Latch_MEMWB));
    memset(&MEMWB[1], 0, sizeof(Latch_MEMWB));
}

void LAT_update(){
    if (IFIDwrite == true){
        memcpy(&IFID[1], &IFID[0], sizeof(Latch_IFID));
        memset(&IFID[0], 0, sizeof(Latch_IFID));
    }
    else{ // IFIDwrite == false면 같은 명령어를 한번 더 decode한다.
        memcpy(&IFID[0], &IFID[1], sizeof(Latch_IFID));
    }
    if (IDEXwrite == true){
        memcpy(&IDEX[1], &IDEX[0], sizeof(Latch_IDEX));
        memset(&IDEX[0], 0, sizeof(Latch_IDEX));
    }
    else{ // IDEXwrite == false면 같은 명령어를 한번 더 execute하게 하고, IDEX Latch를 잠근다.
        memcpy(&IDEX[0], &IDEX[1], sizeof(Latch_IDEX));
        IDEX[1].valid = false;
    }
    memcpy(&EXMEM[1], &EXMEM[0], sizeof(Latch_EXMEM));
    memset(&EXMEM[0], 0, sizeof(Latch_EXMEM));
    memcpy(&MEMWB[1], &MEMWB[0], sizeof(Latch_MEMWB));
    memset(&MEMWB[0], 0, sizeof(Latch_MEMWB));
}

```

그림16. LAT_init, LAT_update (in LAT.c)

LAT.c에는 총 4가지 종류의 함수가 존재하는데, LAT_init()은 Latch의 valid값과 Latch의 저장 공간들을 초기화해준다. 여기서 valid가 true여야만 Latch가 작동하며, false일 경우 작동하지 않고 stalled 시킨다. 이는 Hazard를 해결하기 위한 중요한 요소이다. LAT_update()는 한 Cycle이 종료될때마다 Latch의 값을 갱신시킨다. 앞서 말한 Latch의 0번째 index값을 1번째 index로 복사해옴으로써 업데이트 한다. 여기서도 Hazard가 발생한 경우에 따라 업데이트의 방법이 달라지는데, 특정 상황에서 ID와 EX단계를 stalled하기 위해 똑같은 내용을 0번째 index에 재복사 하는 방식으로 명령어의 decode나 execute를 한번 더 진행하도록 한다. MEM이나 WB의 경우 이전 단계의 valid값에 따라 자연스럽게 stalled된다.

```

void set_val_IDEX(){
    IDEX[0].nextPC = IFID[1].nextPC;
    IDEX[0].opcode = IFID[1].opcode;
    IDEX[0].rs = IFID[1].rs;
    IDEX[0].rt = IFID[1].rt;
    IDEX[0].rd = IFID[1].rd;
    IDEX[0].shamt = IFID[1].shamt;
    IDEX[0].funct = IFID[1].funct;
    IDEX[0].address = IFID[1].address;
    IDEX[0].imm = MUX(IDEX[0].sig.ZeroEx, signExtend(IFID[1].imm), zeroExtend(IFID[1].imm));
}

void set_sig_EXMEM(){
    EXMEM[0].sig.JumpAL = IDEX[1].sig.JumpAL;
    EXMEM[0].sig.Branch = IDEX[1].sig.Branch; // 출력용으로만 필요, 이론적으로는 전달X
    EXMEM[0].sig.BranchNE = IDEX[1].sig.BranchNE; // 출력용으로만 필요, 이론적으로는 전달X
    EXMEM[0].sig.RegWrite = IDEX[1].sig.RegWrite;
    EXMEM[0].sig.MemWrite = IDEX[1].sig.MemWrite;
    EXMEM[0].sig.MemRead = IDEX[1].sig.MemRead;
    EXMEM[0].sig.MemtoReg = IDEX[1].sig.MemtoReg;
    EXMEM[0].sig.LoadU = IDEX[1].sig.LoadU;
    EXMEM[0].sig.MemSize = IDEX[1].sig.MemSize;
}

```

그림17. set_val, set_sig (in LAT.c)

set_val() 함수는 모든 Latch에 존재하며 외부에서 들어온 데이터들을 각각 저장하고 특정 상황에서는 가공하여 저장한다. 가령 위 사진에서 IDEX Latch의 immediate 값이 그 예시이다. set_sig() 함수는 Control signal이 필요한 EX, MEM, WB 단계와 관련된 Latch에만 존재하며 이전 단계의 특정 Control signal들만 받아온다. 이를 통해 단계별로 명령어 처리가 가능하다.

3-4. Hazard Detection Unit (HU.c)

```

void HU_init(){
    PCwrite = true;
    IFIDwrite = true;
    IDEXwrite = true;
    DH_DW_counter = 0;
    CH_DW_counter = 0;
}

```

그림18. HU_init

Hazard Detection Unit은 특정 상황에서 발생하는 Data hazard와 jump관련 명령어로 발생하는 hazard를 감지하고 처리하는 장치이다.

이 unit은 Definition에서 언급한 2가지의 counter 변수와 3가지의 control signal을 사용한다. 표는 3가지의 신호 값이 false일 때 발생하는 변화를 정리한 것이다.

PCwrite	이전 PC값으로 한번 더 Fetch하게 하여 다음 명령어의 Fetch를 막는다.
IFIDwrite	IF/ID Latch를 같은 내용을 한번 더 decode하게 업데이트 한다.
IDEXwrite	ID/EX Latch를 같은 내용을 한번 더 execute하게 업데이트 한다. 또한 ID/EX Latch의 valid값을 바꿔 실행하지 못하도록 한다.

```
void HU_operation_opt0(){
    // Detect Control Hazard (case J, JR, JAL, JALR)
    if (IDEX[1].sig.Jump || IDEX[1].sig.JumpR){
        IFID[0].valid = false;
    }

    if (DH_DW_counter > 0){
        DH_DW_counter--;
        if (DH_DW_counter == 0){
            PCwrite = true;
            IFIDwrite = true;
            IDEXwrite = true;
        }
    }
    // Data Hazard
    else if (((EXMEM[0].rd != 0) && (EXMEM[0].sig.RegWrite) && ((EXMEM[0].rd == IDEX[0].rs || (EXMEM[0].rd == IDEX[0].rt))) ||
             ((MEMWB[0].rd != 0) && (MEMWB[0].sig.RegWrite) && ((EXMEM[0].rd != IDEX[0].rs && (MEMWB[0].rd == IDEX[0].rs)) ||
              ((EXMEM[0].rd != IDEX[0].rt) && (MEMWB[0].rd == IDEX[0].rt))))){
        printf("\nData Hazard Detected");
        DH_DW_counter = 2;
        PCwrite = false;
        IFIDwrite = false;
        IDEXwrite = false;
    }
    else{
        PCwrite = true;
        IFIDwrite = true;
        IDEXwrite = true;
    }
}
```

그림19. HU_operation_opt0

HU.c에는 Data Hazard를 고치기 위해 2가지의 operation이 존재한다. 이들은 forwarding 옵션 유무에 따라 다르게 실행된다. 먼저 forwarding 옵션이 없을 시 실행하는 HU_operation_opt0()이다. 이 함수에서는 Data Hazard를 detect & wait 방법으로 처리한다. 가장 위의 if문에서는 jump 관련 명령어일 시 이전에 온 명령어를 덮어쓰우게끔 stalled하도록 한다. 이때 2 stalled를 하지 않는 이유는 명령어 2개 중 하나는 컴파일러가 scheduling하여 수정된 상태이기 때문이다. 아래쪽의 else-if 문의 조건이 Data Hazard가 발생하는 모든 condition들을 넣은 것으로 저런 상황이 발생한다면 DH_DW_counter값을 2로 만든다. 이 카운터 값이 0이 되기 전까지 PCwrite, IFIDwrite, IDEXwrite값은 false이므로 정상적으로 Latch가 작동하는 것을 방지한다.

```

void HU_operation_opt1(){
    // Detect Control Hazard (case J, JR, JAL, JALR)
    if (IDEX[1].sig.Jump || IDEX[1].sig.JumpR){
        IFID[0].valid = false;
    }

    // Detect Load-Use Hazard
    if (IDEX[1].sig.MemRead && (IDEX[1].rt == IFID[1].rs || IDEX[1].rt == IFID[1].rt)){
        printf("\nLoad-Use Hazard Deected");
        PCwrite = false;
        IFIDwrite = false;
        IDEXwrite = false;
    }

    else{
        PCwrite = true;
        IFIDwrite = true;
        IDEXwrite = true;
    }
}

```

그림20. HU_operation_opt1

다음은 forwarding 옵션이 있을 경우 작동하는 HU_operation_opt1() 함수이다. opt0 함수와 다르게 일일이 2 stalled를 할 필요가 없으므로 jump 관련 명령어와 Load-Use Hazard일 경우만을 감지하고 처리한다. opt0 함수에서 보았던 Data Hazard 조건은 forwarding unit에서 처리한다.

3-5. Forward Unit (FW.c)

```

void FW_init(){
    fwA = 0b00;
    fwB = 0b00;
}

void FW_operation(){
    fwA = 0b00;
    fwB = 0b00;
    if ((EXMEM[0].rd != 0) && (EXMEM[0].sig.RegWrite)){ // EXMEM forwarding (add 등)
        if (EXMEM[0].rd == IDEX[0].rs){
            fwA = 0b10;
        }
        if (EXMEM[0].rd == IDEX[0].rt){
            fwB = 0b10;
        }
    }
    if ((MEMWB[0].rd != 0) && (MEMWB[0].sig.RegWrite)){ // MEMWB forwarding (LW 등)
        if ((EXMEM[0].rd != IDEX[0].rs) && (MEMWB[0].rd == IDEX[0].rs)){
            fwA = 0b01;
        }
        if ((EXMEM[0].rd != IDEX[0].rt) && (MEMWB[0].rd == IDEX[0].rt)){
            fwB = 0b01;
        }
    }
}

```

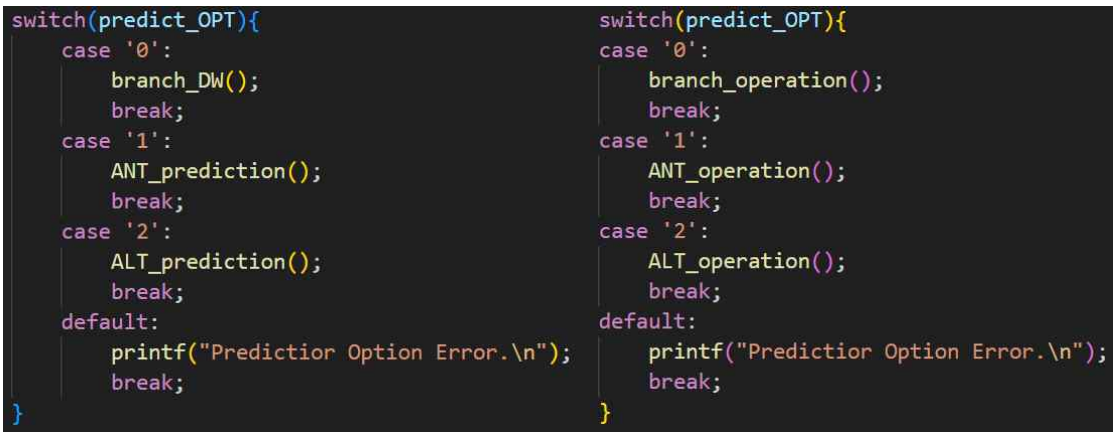
그림21. HU_operation_opt1

Forward Unit은 Hazard 감지 시 연산 결과값을 임시로 저장한 후 다음 명령어가 필요할 때 제공한다. 그래서 명령어가 write back 단계까지 진행하여 불필요한 stalled이 되는 경우가 줄어든다. 이를 통해 파이프라인 구조에서 필요한 cycle의 수가 크게 감소한다. Forwarding bit는 ALU의 2가지 input과 대응하는 fwA, fwB 2개가 존재하고 이들은 2개의 bit로 구성된다. add와 같이 EXMEM에서 forwarding이 필요한 경우에는 비트가 10이 되고, lw와 같이 MEMWB에서 forwarding이 필요한 경우에는 비트가 01이 된다. Hazard가 감지가 되지 않았으면 비트는 그대로 00이다. 비트의 케이스에 따른 forwarding condition은 아래와 같다.

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The 1st ALU operand comes from the register file
ForwardA = 10	EX/MEM	The 1st ALU operand is forwarded from the prior ALU result
ForwardA = 01	MEM/WB	The 1st ALU operand is forwarded from data memory or from an earlier ALU result
ForwardB = 00	ID/EX	The 2nd ALU operand comes from the register file
ForwardB = 10	EX/MEM	The 2nd ALU operand is forwarded from the prior ALU result
ForwardB = 01	MEM/WB	The 2nd ALU operand is forwarded from data memory or from an earlier ALU result

그림22. forwarding conditions (04-5. Processor 7p)

3-6. Branch Prediction (PRE.c)



```
switch(predict_OPT){
    case '0':
        branch_DW();
        break;
    case '1':
        ANT_prediction();
        break;
    case '2':
        ALT_prediction();
        break;
    default:
        printf("Prediction Option Error.\n");
        break;
}

switch(predict_OPT){
    case '0':
        branch_operation();
        break;
    case '1':
        ANT_operation();
        break;
    case '2':
        ALT_operation();
        break;
    default:
        printf("Prediction Option Error.\n");
        break;
}
```

그림23. main에서 사용되는 prediction part (왼쪽은 예측, 오른쪽은 실행)

이번 프로젝트에서 가장 핵심인 부분이다. Branch 예측이 파이프라인의 효율성 향상에 큰 영향을 미치고, 다양한 상황에 따라 효율적인 예측 알고리즘이 모두 다르기 때문이다. 현재 구현한 파이프라인은 3가지의 Control hazard 해결 옵션을 제시하는데, 각각 detect & wait, always not taken prediction(ANT), always taken prediction(ALT)이다. 첫 번째인 detect & wait은 단순히 branch 관련 명령어가 나오면 2 stalled를 진해하고, 두 번째와 세 번째 방법은 static branch prediction 방법이다. 이들은 IF단계에서 분기를 예측하여 다음 PC값을 결정하고, EX단계에서 branch가 실제로 taken인지 아닌지를 판단한다. 처음에는 branch taken 여부를 ID 단계로 당겨 구현하려 했다. Branch 예측 실패 패널티인 2 stalled를 1 stalled로 줄일 수 있기 때문이다. 그러나 branch에서 발생하는 data hazard의 문제, Equalator 구현 문제, forwarding 문제 등 여러 문제점들을 해결하는 시행착오 끝에 결국 구현에는 실패하고 EX단계로 구현했다. 그래도 static prediction까지는 모두 구현을 완료하여 detect & wait와 prediction 방법의 비교, 코드에 따른 ANT prediction과 ALT prediction의 정확도와 cycle 수에 대한 유의미한 분석은 가능했다.

```

void branch_DW(){
    PC = PCAdder(PC);
    IFID[0].nextPC = PC;
    if (CH_DW_counter > 0){
        CH_DW_counter--;
        PC -= 4;
        if (CH_DW_counter == 0){
            PC += 4;
        }
    }
    else if ((IFID[0].opcode == BEQ || IFID[0].opcode == BNE)){
        printf("\nBranch Detected");
        CH_DW_counter = 2;
    }
}

void branch_operation(){
    if (PCSrc) brc_cnt++;
    if (PCwrite){
        PC = MUX(PCSrc, PC, branchAdder(IDEX[1].nextPC, IDEX[1].imm));
    }
}

```

그림24. Control Hazard 해결법 – detect & wait (in PRE.c)

detect & wait는 branch를 예측하는 방법이 아닌, 명령어가 BEQ나 BNE일 시 무조건 stalled하여 잘못된 코드의 제어 흐름을 방지한다. 이전 Data hazard에서 detect & wait를 한 방법과 거의 유사하게 CH_DW_counter 변수를 이용하여 hazard를 해결한다. branch 명령어임을 감지하며 카운터값을 2로 설정하고, 0이 되기 전까지 PC가 다음 명령어를 fetch하지 못하도록 PC값 자체를 control한다. 이때 PCwrite 신호는 Hazard detect unit 전용이기도 하고 중복해서 쓰면 신호가 충돌하는 경우가 발생하기 때문에 PC값을 직접 설정하는 방식을 선택했다. branch_operation()은 평범한 branch 실행으로 main에서 아래와 같이 결정한 PCSrc 값에 따라 target address를 구한다.

```

PCSrc =
(IDEX[1].sig.Branch &&!EXMEM[0].ALUResult) ||
(IDEX[1].sig.BranchNE &&EXMEM[0].ALUResult);

```



```

void ANT_prediction(){
    PC = PCAdder(PC);
    IFID[0].nextPC = PC;
}

void ANT_operation(){
    if (PCSrc){
        // Branch Taken -> Prediction Fail
        brc_cnt++;
        fail++;
        IFID[0].valid = false;
        IDEX[0].valid = false;
    }
    else{
        // Branch Not Taken -> Prediction Success
        success++;
    }
    // PC update
    if (PCwrite){
        PC = MUX(PCSrc, PC, branchAdder(IDEX[1].nextPC, IDEX[1].imm));
    }
}

```

그림25. Control Hazard 해결법 – ANT prediction (static)

```

void ALT_prediction(){
    uint32_t tmp_imm;
    PC = PCAdder(PC);
    if (IFID[0].imm & 0x8000){
        tmp_imm = 0xffff0000 | IFID[0].imm;
    }
    else{
        tmp_imm = 0x0000ffff & IFID[0].imm;
    }
    if ((IFID[0].opcode == BEQ || IFID[0].opcode == BNE)){
        ALT_backupPC = PC;
        PC = branchAdder(PC, tmp_imm);
    }
    IFID[0].nextPC = PC;
}

void ALT_operation(){
    if (PCSrc){
        // Branch Taken -> Prediction Success
        success++;
    }
    else{
        // Branch Not Taken -> Prediction Fail
        brc_cnt++;
        fail++;
        IFID[0].valid = false;
        IDEX[0].valid = false;
        PC = ALT_backupPC;
    }
}

```

그림26. Control Hazard 해결법 – ALT prediction (static)

ANT는 always not taken 예측 방법으로 말 그대로 branch가 항상 일어나지 않는다고 예측한다. 그래서 ANT_prediction() 함수를 보면 다음 PC값을 무조건 PC+4로 설정함을 알 수 있다. ANT_operation()에서는 PCSrc값이 1이라면, 즉 branch가 taken이 되었다면 이전 두 명령어가 덮어 씌워져 없어지게끔 두 Latch의 valid값을 바꿔 2 stalled를 발생시킨다. branch실행이 완료되면 detect & wait와 같이 PCSrc 값에 따라 target address를 결정해준다. ALT(always taken) 예측 방법은 ANT와 정확히 반대로 진행하는데, branch 관련 명령어일 시 다음 PC값을 항상 branch target address로 설정해준다. 또한 예측 오답을 대비하기 위해 ALT_backupPC라는 변수에 현재 PC값을 미리 백업해두고, Hazard 발생 시 백업한 값을 불러온다.

3-7. main.c

```
int main(int argc, char* argv[]){
    forward_OPT = argv[1][0];
    predict_OPT = argv[2][0];
    start();
    do{
        cycle++;
        /* Sequential Execution */
        MEM();
        WB();
        IF();
        ID();
        EX();
        /* Show Cycle State */
        print_cycle();
        /* Update latches */
        LAT_update();
    } while (PC != 0xffffffff);
    print_result();
    return 0;
}
```

그림27. main

main 함수의 인자로 argv 배열을 받는 것을 확인할 수 있는데, 이를 통해 forwarding 옵션과 branch prediction 옵션을 받아와 IF와 EX단계의 switch문에 사용한다. 옵션을 선택하여 컴파일하는 방법을 4-2에서 후술한다.

main에서 눈여겨볼 점은 Single-Cycle과 while문의 구조가 바뀌었다는 점이다. PC값을 while문의 마지막에 갱신하고 체크해야 하고, PC값이 0xffffffff라는 것을 마지막에 감지해야 하므로, while문을 do-while문으로 바꿔주고 EX()단계를 마지막에 배치 주었다. 어차피 모든 단계가 독립적이고 valid 값으로 관리되기 때문에 순서를 바꿔도 지장이 없다. MEM()부터 EX()까지 각 단계는 이전 Single-Cycle 프로젝트와 큰 차이가 없지만, 그래도 변화한 점이 있으므로 짚고 넘어갈 것이다.

```
void IF(){
    if (PC != 0xffffffff){
        IFID[0].valid = true;
        if (PCwrite == false) PC = IFID[1].nextPC;
        IR = IM_fetch(PC);
        set_val_IFID();
        // 다음 명령어 Fetch를 위해 branch prediction을 진행
        switch(predict_OPT){
            case '0':
                branch_DW();
                break;
            case '1':
                ANT_prediction();
                break;
            case '2':
                ALT_prediction();
                break;
            default:
                printf("Prediction Option Error.\n");
                break;
        }
    }
}

void ID(){
    if (IFID[1].valid == true){
        // Execute를 위한 Control Signal 준비, 값 설정 및 다음 latch로 전송
        CU_setting(IFID[1].opcode, IFID[1].funct);
        RF_read(IFID[1].rs, IFID[1].rt);
        IDEX[0].valid = true;
        set_sig_EXMEM();
        set_val_IDEX();
    }
}
```

그림28. IF, ID (in main.c)

IF에서 명령어를 Fetch해오는 것은 이전과 동일하지만 PC값을 결정해주는 메커니즘이 다르다. 만약 forwarding 옵션이 0인 상태에서 detect & wait 기법으로 Data Hazard를 감지했을 경우 PCwrite값이 false가 되고, 이럴 경우 PC+4가 아닌 다음 단계의 PC값인 nextPC를 당겨와서 사용한다. 그리고 set_val_IFID()함수는 이전 프로젝트에서 decode 단계에 있던 것으로, 원래는 32비트의 명령어를 형식에 맞게 쪼개서 하나의 명령어 구조체를 만들었었다. 그러나 파이프라인 구조에서는 Latch의 활용으로 인해 명령어 구조체가 필요 없어져 IF/ID Latch에 명령어를 쪼개서 opcode부터 address까지 저장한다. Single-cycle과 다르게 IF에서 32비트를 나누는 이유는 아래의 predict_OPT가 2인 경우, 즉 ALT prediction을 진행할 경우 fetch단계에서 target address를 구하기 위해 opcode, immediate값이 미리 필요하기 때문이다. ID 단계는 Latch 업데이트 부분을 제외하고 이전 프로젝트와 동일하다.

```
if (IDEX[1].valid == true){
    uint32_t ALUControl = ALUCU_control(IDEX[1].sig.ALUOp, MUX(IDEX[1].sig.RegDst, IDEX[1].opcode, IDEX[1].funct));
    EXMEM[0].ReadData1 = MUX3(fwA, IDEX[1].ReadData1,
                              MUX(MEMWB[1].sig.MemtoReg, MEMWB[1].ALUResult, MEMWB[1].ReadData), EXMEM[1].ALUResult);
    EXMEM[0].ReadData2 = MUX3(fwB, IDEX[1].ReadData2,
                              MUX(MEMWB[1].sig.MemtoReg, MEMWB[1].ALUResult, MEMWB[1].ReadData), EXMEM[1].ALUResult);
    EXMEM[0].ALUResult = ALU_calculate(ALUControl, MUX(IDEX[1].sig.Shift, EXMEM[0].ReadData1, EXMEM[0].ReadData2),
                                       MUX(IDEX[1].sig.Shift, MUX(IDEX[1].sig.ALSrc, EXMEM[0].ReadData2, IDEX[1].imm), IDEX[1].shamt));
    EXMEM[0].valid = true;
    set_val_EXMEM();

    PCSrc = (IDEX[1].sig.Branch && !EXMEM[0].ALUResult) || (IDEX[1].sig.BranchNE && EXMEM[0].ALUResult);
    if (IDEX[1].sig.Branch || IDEX[1].sig.BranchNE){
        switch(predict_OPT){
            case '0':
                branch_operation();
                break;
            case '1':
                ANT_operation();
                break;
            case '2':
                ALT_operation();
                break;
            default:
                printf("Prediction Option Error.\n");
                break;
        }
    }
    // PC Update
    if (PCwrite){
        PC = MUX(IDEX[1].sig.Jump, MUX(IDEX[1].sig.JumpR, PC, EXMEM[0].ReadData1), jumpMaker(IDEX[1].nextPC, IDEX[1].address));
    }
}

set_sig_MEMWB();
// Hazard Detection, Forwarding
switch(forward_OPT){
    case '0':
        HU_operation_opt0();
        break;
    case '1':
        HU_operation_opt1();
        FW_operation();
        break;
    default:
        printf("Forwarding Option Error.\n");
        break;
}
```

그림29. EX (in main.c)

기존 EX와 동일하게 ReadData1, 2와 ALU 연산 결과 값을 도출하는 것은 같지만, 여기서 forwarding 메커니즘이 적용된다. forwarding 비트가 01이면 MEMWB의 연산 결과를 가져오고, forwarding 비트가 10이면 EXMEM의 연산 결과를 가져온다. forward 옵션이 0이면 어차피 fwA와 fwB가 00으로 고정이므로 영향이 없다. 3가지 값을 결정해 주었다면 PCSrc 값을 계산하고 명령어가 BEQ이거나 BNE일 경우에만 branch prediction 관련 작업을 수행한다. prediction이 끝나면 PC값 관련 신호들로 최종 PC값을 결정해준다. 마지막으로 현재 단계에서 Data Hazard가 발생했는지를 체크하기 위해 forward 옵션에 따라 Data Hazard를 감지하고 해결한다. MEM 단계와 WB 단계는 아래와 같이 Latch의 값을 활용한다는 점을 제외하고는 이전 프로젝트와 차이점이 거의 없다. 1가지 다른 점은, 원래 Single-Cycle에서는 JAL과 JALR을 위해 PC+8값을 앞쪽에서 계산한 후에 당겨왔는데, 이번 프로젝트는 파이프라인 구조를 지켜야 하므로 PC값을 각 Latch의 nextPC에 넣어 전달한 다음 마지막에 nextPC에 +4를 하여 해당 단계에 맞는 PC+8을 만들어준다.

```
void MEM(){
    if (EXMEM[1].valid == true){
        DM_operation(EXMEM[1].sig.MemRead, EXMEM[1].sig.MemWrite, EXMEM[1].ALUResult, EXMEM[1].ReadData2,
                     MUX3(EXMEM[1].sig.MemSize, 32, 16, 8));
        MEMWB[0].valid = true;
        set_val_MEMWB();
    }
}

void WB(){
    if (MEMWB[1].valid == true){
        uint32_t outData = MUX(MEMWB[1].sig.JumpAL,
                               MUX(MEMWB[1].sig.MemtoReg, MEMWB[1].ALUResult, MEMWB[1].ReadData), MEMWB[1].nextPC + 4);
        DataSrc = (MEMWB[1].sig.MemSize == 0b00) || (MEMWB[1].sig.MemSize == 0b11);
        RF_write(MEMWB[1].sig.RegWrite, MEMWB[1].rd,
                 MUX(DataSrc, MUX(MEMWB[1].sig.LoadU, signExtend(outData), zeroExtend(outData)), outData));
    }
}
```

그림30. MEM, WB (in main.c)

4. Environment

4-1. Build Environment

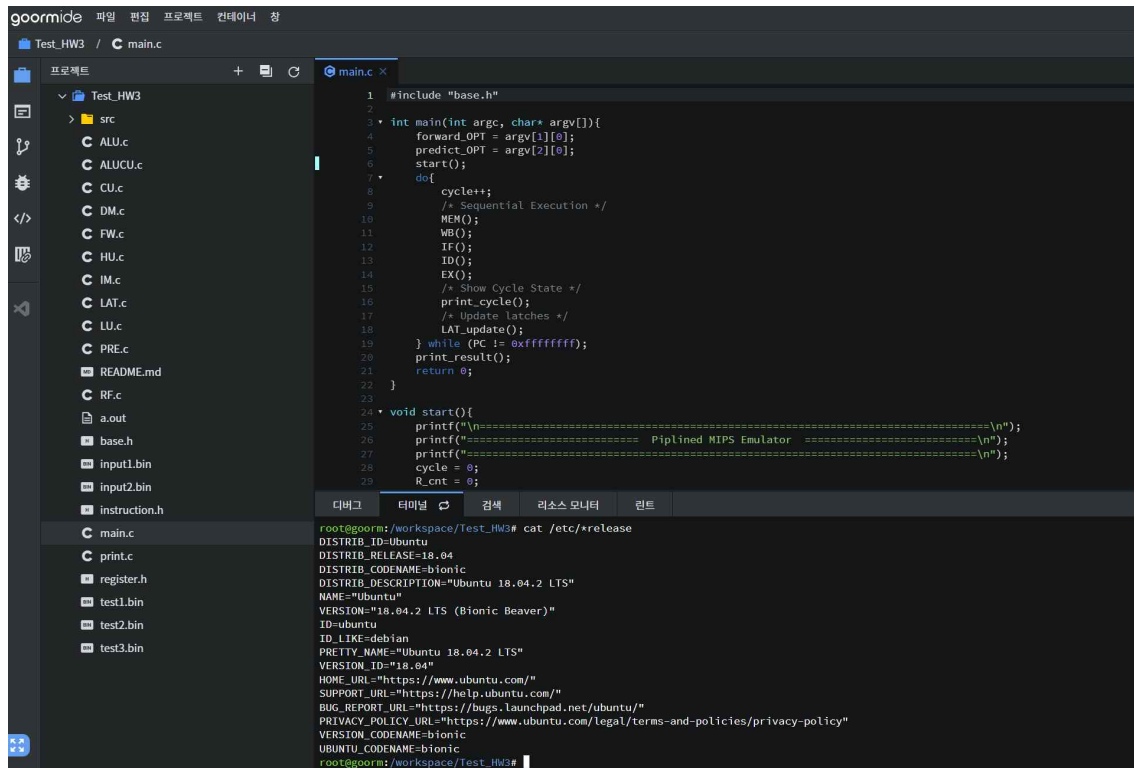


그림31. goormIDE 컨테이너 내의 Ubuntu(Linux) Environment

IDE : goorm Cloud IDE Service (<https://www.goorm.io/dashboard>)

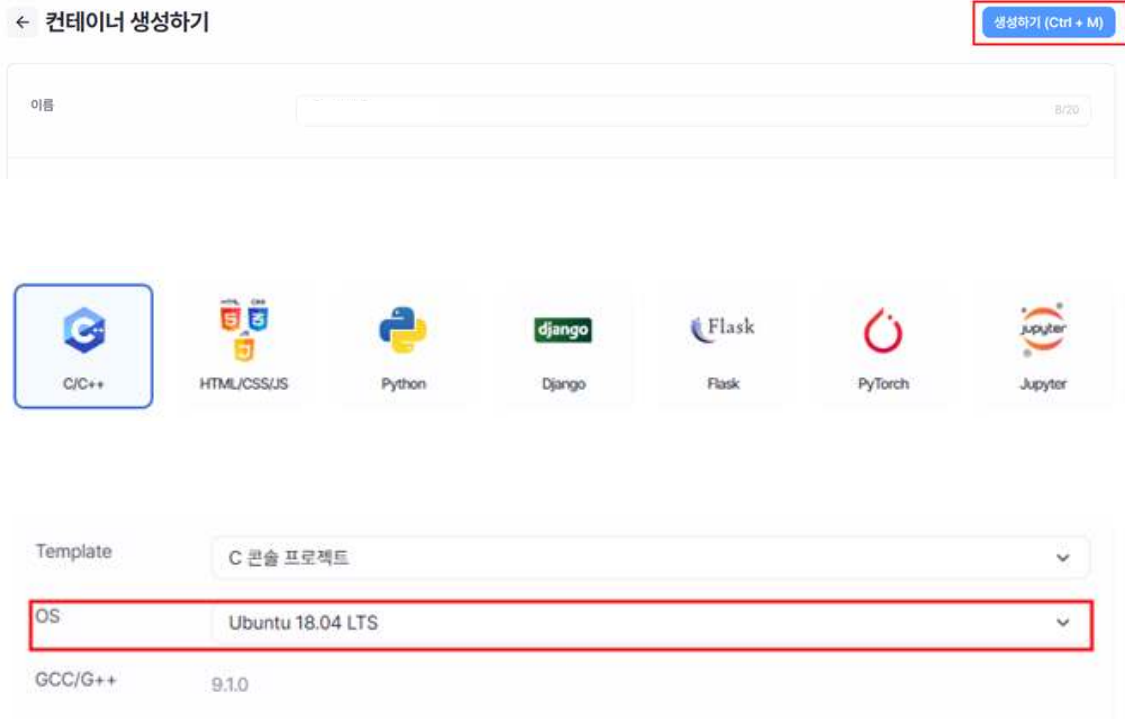
OS : Ubuntu 18.04.6 LTS

GCC Compiler (Ubuntu 9.1.0-2ubuntu2~18.04) 9.1.0

2번째 프로젝트에서 사용한 GoormIDE 개발 환경을 그대로 사용하였다. 이번 프로젝트 역시 리눅스 환경에서의 MIPS 크로스 컴파일러가 필요하기 때문이다. 이전에 설명한듯 GoormIDE는 컨테이너 기반의 work space를 제공하며, 각각의 컨테이너는 독립적인 가상 리눅스 환경 위에서 작동한다는 점이 큰 메리트이다. 이를 활용하여 필요한 c파일을 언제든지 Assembly 코드로도 확인하거나, object 파일과 binary 파일을 명령어만을 입력하여 생성한다.

4-2. Compile

GoormIDE 홈페이지 (<https://ide.goorm.io/>) 에서 회원가입을 하고 로그인하면 나의 대쉬보드로 접속된다. 오른쪽 위의 "새 컨테이너" 를 클릭하면 아래와 같이 이동한다.



The image shows the 'Container Creation' page in GoormIDE. At the top left is a back arrow and the text '컨테이너 생성하기'. At the top right is a blue button labeled '생성하기 (Ctrl + M)'. Below this is a form with a '이름' (Name) field. Underneath the name field is a row of software stack icons: C/C++ (selected with a blue border), HTML/CSS/JS, Python, Django, Flask, PyTorch, and Jupyter. Below the icons is a 'Template' dropdown menu set to 'C 콘솔 프로젝트'. Below that is an 'OS' dropdown menu set to 'Ubuntu 18.04 LTS', which is highlighted with a red rectangle. At the bottom, 'GCC/G++' is set to '9.1.0'.

컨테이너 생성하기 페이지가 나오면 프로젝트 이름을 입력한다. 생성하기 전에 스크롤을 아래로 내려서 소프트웨어 스택을 C/C++로, OS를 Ubuntu 18.04 LTS로 설정한 후에 생성하기를 누른다. 다른 옵션들은 기본 설정으로 해도 무방하다.

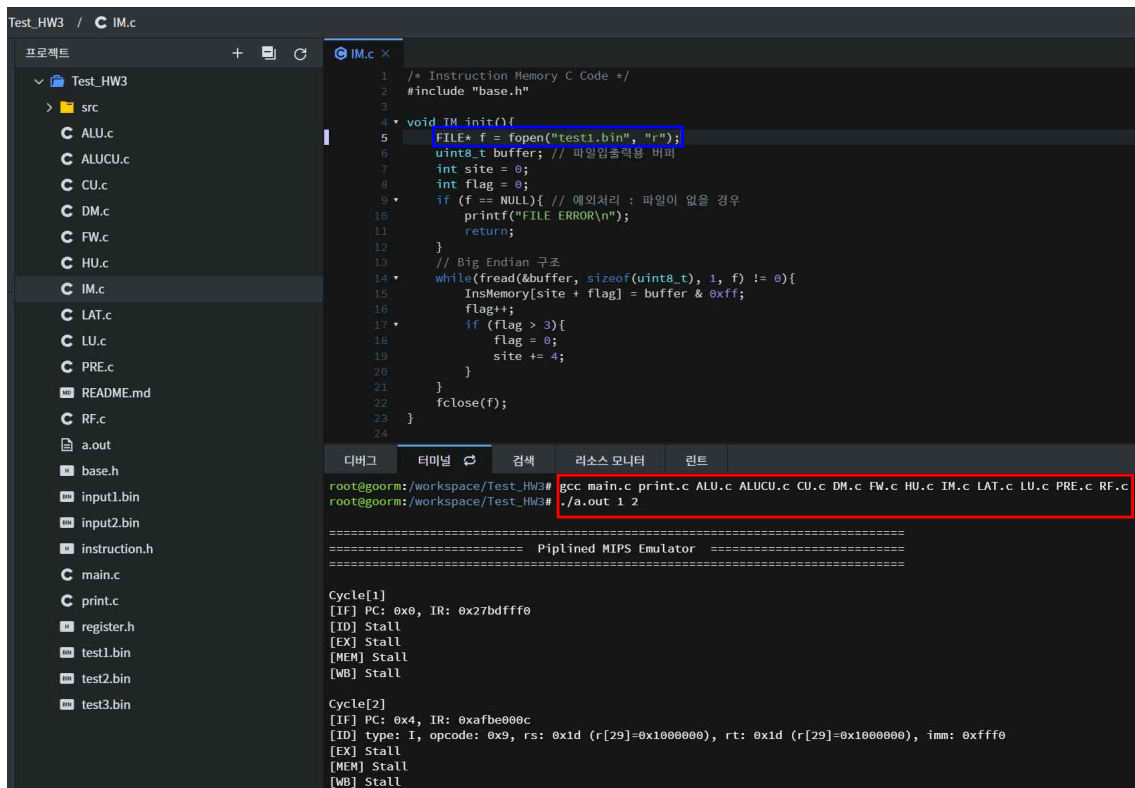
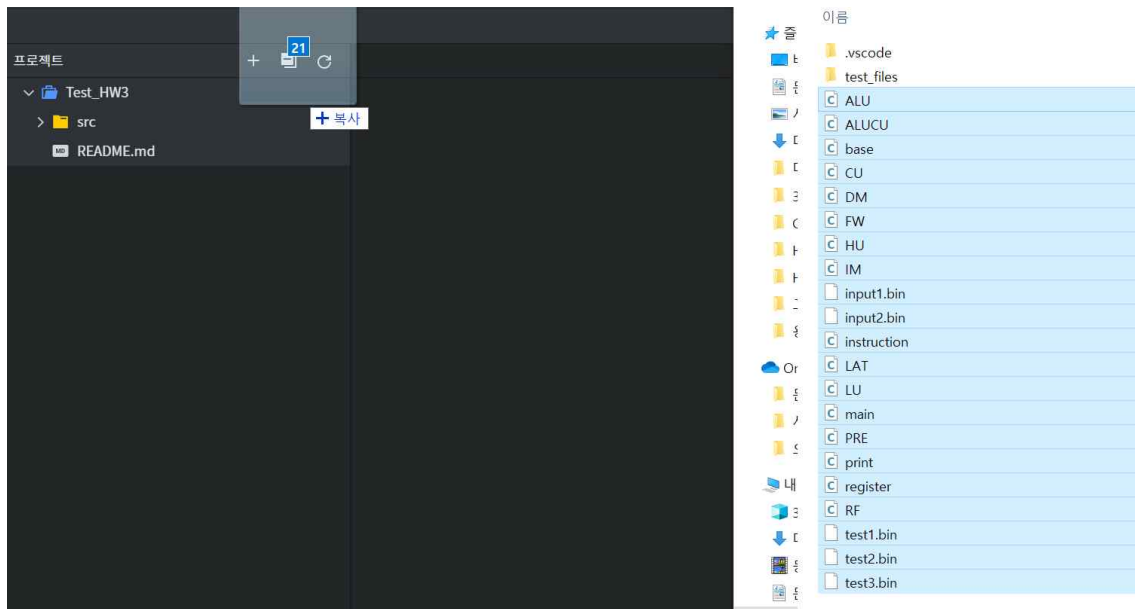
```
occurred during the signature verification. The repository is not updated and the previous index files will be used. GPG error: https://cli-assets.h
2에 인증 할 수 없습니다: NO_PUBKEY 536F8F1DE80F6A35
occurred during the signature verification. The repository is not updated and the previous index files will be used. GPG error: https://cf-cli-debian
트지 않습니다: EXPKEYSIG 172B5989FCD21EF8 CF CLI Team <cf-cli-eng@pivotal.io>
ackages.cloudfoundry.org/debian/dists/stable/InRelease 파일을 받는데 실패했습니다. 다음 서명이 올바르지 않습니다: EXPKEYSIG 172B5989FCD21EF8 CF CLI T
cli-assets.heroku.com/apt/./InRelease 파일을 받는데 실패했습니다. 다음 서명들은 공개키가 없기 때문에 인증할 수 없습니다: NO_PUBKEY 536F8F1DE80F6A35
ex files failed to download. They have been ignored, or old ones used instead.
/workspace/Test_HW2# sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 536F8F1DE80F6A35
```

컨테이너가 생성되면 터미널을 열어서 **sudo apt-get update** 명령어를 입력한다. 그러나 위의 사진처럼 key 관련한 오류가 표시될 수 있다. 이는 사진 속 빨간색으로 강조한 부분처럼 공개키에 에러가 있는 경우이다. 아래의 명령어로 서버에서 2개의 key를 다시 가져오고 다시 update를 시도하면 정상적으로 실행된다.

sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys (key번호)

update 이후에는 아래의 명령어로 MIPS 크로스 컴파일러를 설치한다.

sudo apt-get -y install gcc-mips-linux-gnu



셋팅이 모두 완료되면 **HW3_32192530_양윤성_code** 파일을 압축해제한 뒤 모든 파일들을 프로젝트 내에 드래그하여 추가한다. 복사 뒤에는 기존 컨테이너에 있던 main.c는 지워줘야 한다. 파일에는 총 5가지의 기본 테스트 코드 (input1, input2, test1, test2, test3)가 존재한다. 이 코드들 중 bin 파일로 하나를 선택하여 사진 속 파란색 부분처럼 IM.c의 5번째 줄에 fopen의 인수로 작성해준다.

이 5가지의 bin 파일들은 HxD로 Jump Address를 수정한 상태이므로 Jump 명령어가 정상 실행된다. 이제 다음 2줄의 명령어를 입력하면 에뮬레이터가 실행된다. 필요한 테스트 파일의 c코드와 object 파일은 test_file 폴더에서 찾아 사용하면 된다.

**gcc main.c print.c ALU.c ALUCU.c CU.c DM.c FW.c HU.c IM.c LAT.c LU.c PRE.c RF.c
./a.out (forwarding 옵션) (branch prediction 옵션)**

컴파일을 한 뒤 실행파일을 실행할 때 작성해야할 옵션 설명은 다음과 같다. forwarding 옵션 중 1개, prediction 옵션 중 1개를 선택하여 실행하면 된다. 범위 밖의 숫자를 입력할 경우 에러 메시지가 출력되며 파이프라인 에뮬레이터가 정상적으로 작동하지 않는다.

forwarding 옵션	설명	prediction 옵션	설명
0	Detect & Wait	0	Detect & Wait
1	Forwarding	1	Static Prediction (ANT)
-	-	2	Static Prediction (ALT)

저번 프로젝트에서는 c코드를 MIPS 크로스 컴파일러로 binary 파일을 만들고 assembly 코드를 확인하기 위해 구름이 필수였지만, 이번 프로젝트에서는 이미 만들어진 binary 파일이 있다면 visual studio code에서 프로젝트를 구동해도 상관없다. 전체적인 방법은 구름과 같으며, 파일을 압축 해제해서 visual studio code로 폴더를 불러온 다음 터미널을 열어서 똑같이 컴파일하고 실행 파일을 실행하면 된다.

4-3. Working Proofs

테스트 파일은 이전 프로젝트와 같다. 이번 프로젝트에서는 각 테스트 코드별로 6가지 옵션 경우의 수에 따른 출력 결과를 위주로 보여줄 것이다.

파일 이름	원본 코드	테스트 주 목적
test1.bin	test1.c	간단한 char 자료형 사용으로 load/store byte를 테스트
test2.bin	test2.c	다른 함수로 jump 여부 테스트
test3.bin	test3.c	fibonacci 수열 재귀 테스트
input1.bin	input1.c	0부터 9까지 더하는 for 반복문 테스트 (과제 테스트 코드)
input2.bin	input2.c	다른 함수로 jump 여부 테스트 (과제 테스트 코드)

```

Cycle[70]
[IF] PC: 0x48, IR: 0x03a0f025
[ID] type: I, opcode: 0x2b, rs: 0x1d (r[29]=0xfffff78), rt: 0x1e (r[30]=0xfffff98), imm: 0x18
[EX] Store Address : 0x00fffff94 = r[29] + 0x0000001c
[MEM] Stall
[WB] Stall

Cycle[71]
[IF] PC: 0x4c, IR: 0xafc40020
[ID] type: R, opcode: 0x0, rs: 0x1d (r[29]=0xfffff78), rt: 0x0 (r[0]=0x0), rd: 0x1e (r[30]=0xfffff98), shamt: 0x0, funct: 0x25
[EX] Store Address : 0x00fffff90 = r[29] + 0x00000018
[MEM] Store Data : M[0x00fffff94] = 0x84
[WB] Stall

Data Hazard Detected
Cycle[72]
[IF] PC: 0x50, IR: 0x8fc30020
[ID] type: I, opcode: 0x2b, rs: 0x1e (r[30]=0xfffff98), rt: 0x4 (r[4]=0x2), imm: 0x20
[EX] r[30] <- 0xfffff78 = r[29] | r[0]
[MEM] Store Data : M[0x00fffff90] = 0xfffff98
[WB] No write back

Cycle[73]
[IF] PC: 0x50, IR: 0x8fc30020
[ID] type: I, opcode: 0x2b, rs: 0x1e (r[30]=0xfffff98), rt: 0x4 (r[4]=0x2), imm: 0x20
[EX] Stall
[MEM] No memory access
[WB] No write back

Cycle[74]
[IF] PC: 0x50, IR: 0x8fc30020
[ID] type: I, opcode: 0x2b, rs: 0x1e (r[30]=0xfffff78), rt: 0x4 (r[4]=0x2), imm: 0x20
[EX] Stall
[MEM] Stall
[WB] R[30] = 0xfffff78

```

그림32. Cycle별 상태 출력

Cycle에서는 각 단계별로 어떤 명령어가 처리되고 있는지를 볼 수 있다. 여기서 Stalled는 2가지로 표시된다. 먼저 Data Hazard Detect로 인해 2 Stalled가 발생한 경우 Hazard가 발생한 구간에 "Data Hazard Detected"가 표시되며 이후 IF와 ID를 같은 내용을 2번 반복하며 stalled임을 알 수 있다. 또 다른 경우는 Branch 예측 실패나 점프 등으로 발생한 1~2 Stalled로, Latch의 valid값을 변경하여 기존 명령어를 덮어 씌우는 방식이기 때문에 Stall로 표시된다.

※ 33 ~ 37p : test1.bin ~ input2.bin 테스트 결과 사진

```

***** Result *****
Data Hazards : Detect and wait
Control Hazards : Detect and wait
Return Value (r2): 98 (0x62)
Total Cycle: 18
Executed 'R' Instruction: 3
Executed 'I' Instruction: 7
Executed 'J' Instruction: 0
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 4

***** End of Pipelined Emulator! *****
***** Result *****
Data Hazards : Detect and wait
Control Hazards : Static branch prediction - ANT
Return Value (r2): 98 (0x62)
Total Cycle: 18
Executed 'R' Instruction: 3
Executed 'I' Instruction: 7
Executed 'J' Instruction: 0
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 4

***** End of Pipelined Emulator! *****
***** Result *****
Data Hazards : Detect and wait
Control Hazards : Static branch prediction - ALT
Return Value (r2): 98 (0x62)
Total Cycle: 18
Executed 'R' Instruction: 3
Executed 'I' Instruction: 7
Executed 'J' Instruction: 0
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 4

***** End of Pipelined Emulator! *****
***** Result *****
Data Hazards : Forwarding
Control Hazards : Detect and wait
Return Value (r2): 98 (0x62)
Total Cycle: 12
Executed 'R' Instruction: 3
Executed 'I' Instruction: 7
Executed 'J' Instruction: 0
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 4

***** End of Pipelined Emulator! *****
***** Result *****
Data Hazards : Forwarding
Control Hazards : Static branch prediction - ANT
Return Value (r2): 98 (0x62)
Total Cycle: 12
Executed 'R' Instruction: 3
Executed 'I' Instruction: 7
Executed 'J' Instruction: 0
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 4

***** End of Pipelined Emulator! *****
***** Result *****
Data Hazards : Forwarding
Control Hazards : Static branch prediction - ALT
Return Value (r2): 98 (0x62)
Total Cycle: 12
Executed 'R' Instruction: 3
Executed 'I' Instruction: 7
Executed 'J' Instruction: 0
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 4

***** End of Pipelined Emulator! *****

```

그림33. test1.bin 테스트 결과

```

***** Result *****
Data Hazards : Detect and wait
Control Hazards : Detect and wait
Return Value (r2): 201 (0xc9)
Total Cycle: 59
Executed 'R' Instruction: 7
Executed 'I' Instruction: 23
Executed 'J' Instruction: 1
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 17
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Detect and wait
Control Hazards : Static branch prediction - ANT
Return Value (r2): 201 (0xc9)
Total Cycle: 59
Executed 'R' Instruction: 7
Executed 'I' Instruction: 23
Executed 'J' Instruction: 1
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 17
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Detect and wait
Control Hazards : Static branch prediction - ALT
Return Value (r2): 201 (0xc9)
Total Cycle: 59
Executed 'R' Instruction: 7
Executed 'I' Instruction: 23
Executed 'J' Instruction: 1
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 17
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Forwarding
Control Hazards : Detect and wait
Return Value (r2): 201 (0xc9)
Total Cycle: 39
Executed 'R' Instruction: 7
Executed 'I' Instruction: 23
Executed 'J' Instruction: 1
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 17
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Forwarding
Control Hazards : Static branch prediction - ANT
Return Value (r2): 201 (0xc9)
Total Cycle: 39
Executed 'R' Instruction: 7
Executed 'I' Instruction: 23
Executed 'J' Instruction: 1
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 17
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Forwarding
Control Hazards : Static branch prediction - ALT
Return Value (r2): 201 (0xc9)
Total Cycle: 39
Executed 'R' Instruction: 7
Executed 'I' Instruction: 23
Executed 'J' Instruction: 1
Number of Branch Taken: 0
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 17
***** End of Pipelined Emulator! *****

```

그림34. test2.bin 테스트 결과

```

***** Result *****
Data Hazards : Detect and wait
Control Hazards : Detect and wait
Return Value (r2): 55 (0x37)
Total Cycle: 4864
Executed 'R' Instruction: 546
Executed 'I' Instruction: 1699
Executed 'J' Instruction: 109
Number of Branch Taken: 109
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 988
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Detect and wait
Control Hazards : Static branch prediction - ANT
Return Value (r2): 55 (0x37)
Total Cycle: 4756
Executed 'R' Instruction: 546
Executed 'I' Instruction: 1645
Executed 'J' Instruction: 109
Number of Branch Taken: 109
Number of Predict Success: 55
Number of Predict Fail: 109
Number of Memory Access Instruction: 988
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Detect and wait
Control Hazards : Static branch prediction - ALT
Return Value (r2): 55 (0x37)
Total Cycle: 4648
Executed 'R' Instruction: 546
Executed 'I' Instruction: 1700
Executed 'J' Instruction: 109
Number of Branch Taken: 55
Number of Predict Success: 109
Number of Predict Fail: 55
Number of Memory Access Instruction: 988
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Forwarding
Control Hazards : Detect and wait
Return Value (r2): 55 (0x37)
Total Cycle: 3283
Executed 'R' Instruction: 546
Executed 'I' Instruction: 1645
Executed 'J' Instruction: 109
Number of Branch Taken: 109
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 988
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Forwarding
Control Hazards : Static branch prediction - ANT
Return Value (r2): 55 (0x37)
Total Cycle: 3228
Executed 'R' Instruction: 546
Executed 'I' Instruction: 1645
Executed 'J' Instruction: 109
Number of Branch Taken: 109
Number of Predict Success: 55
Number of Predict Fail: 109
Number of Memory Access Instruction: 988
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Forwarding
Control Hazards : Static branch prediction - ALT
Return Value (r2): 55 (0x37)
Total Cycle: 3120
Executed 'R' Instruction: 546
Executed 'I' Instruction: 1700
Executed 'J' Instruction: 109
Number of Branch Taken: 55
Number of Predict Success: 109
Number of Predict Fail: 55
Number of Memory Access Instruction: 988
***** End of Pipelined Emulator! *****

```

그림35. test3.bin 테스트 결과


```

***** Result *****
Data Hazards : Detect and wait
Control Hazards : Detect and wait
Return Value (r2): 45 (0x2d)
Total Cycle: 320
Executed 'R' Instruction: 13
Executed 'I' Instruction: 111
Executed 'J' Instruction: 0
Number of Branch Taken: 11
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 76
***** End of Pipelined Emulator! *****
***** Result *****

Data Hazards : Detect and wait
Control Hazards : Static branch prediction - ANT
Return Value (r2): 45 (0x2d)
Total Cycle: 300
Executed 'R' Instruction: 13
Executed 'I' Instruction: 101
Executed 'J' Instruction: 0
Number of Branch Taken: 11
Number of Predict Success: 1
Number of Predict Fail: 11
Number of Memory Access Instruction: 66
***** End of Pipelined Emulator! *****
***** Result *****

Data Hazards : Detect and wait
Control Hazards : Static branch prediction - ALT
Return Value (r2): 45 (0x2d)
Total Cycle: 280
Executed 'R' Instruction: 13
Executed 'I' Instruction: 102
Executed 'J' Instruction: 0
Number of Branch Taken: 1
Number of Predict Success: 11
Number of Predict Fail: 1
Number of Memory Access Instruction: 66
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Forwarding
Control Hazards : Detect and wait
Return Value (r2): 45 (0x2d)
Total Cycle: 171
Executed 'R' Instruction: 13
Executed 'I' Instruction: 101
Executed 'J' Instruction: 0
Number of Branch Taken: 11
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 66
***** End of Pipelined Emulator! *****
***** Result *****

Data Hazards : Forwarding
Control Hazards : Static branch prediction - ANT
Return Value (r2): 45 (0x2d)
Total Cycle: 170
Executed 'R' Instruction: 13
Executed 'I' Instruction: 101
Executed 'J' Instruction: 0
Number of Branch Taken: 11
Number of Predict Success: 1
Number of Predict Fail: 11
Number of Memory Access Instruction: 66
***** End of Pipelined Emulator! *****
***** Result *****

Data Hazards : Forwarding
Control Hazards : Static branch prediction - ALT
Return Value (r2): 45 (0x2d)
Total Cycle: 150
Executed 'R' Instruction: 13
Executed 'I' Instruction: 102
Executed 'J' Instruction: 0
Number of Branch Taken: 1
Number of Predict Success: 11
Number of Predict Fail: 1
Number of Memory Access Instruction: 66
***** End of Pipelined Emulator! *****

```

그림36. input1.bin 테스트 결과

```

***** Result *****
Data Hazards : Detect and wait
Control Hazards : Detect and wait
Return Value (r2): 10 (0xa)
Total Cycle: 183
Executed 'R' Instruction: 24
Executed 'I' Instruction: 63
Executed 'J' Instruction: 4
Number of Branch Taken: 4
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 36
***** End of Pipelined Emulator! *****
***** Result *****

Data Hazards : Detect and wait
Control Hazards : Static branch prediction - ANT
Return Value (r2): 10 (0xa)
Total Cycle: 177
Executed 'R' Instruction: 24
Executed 'I' Instruction: 60
Executed 'J' Instruction: 4
Number of Branch Taken: 4
Number of Predict Success: 1
Number of Predict Fail: 4
Number of Memory Access Instruction: 36
***** End of Pipelined Emulator! *****
***** Result *****

Data Hazards : Detect and wait
Control Hazards : Static branch prediction - ALT
Return Value (r2): 10 (0xa)
Total Cycle: 171
Executed 'R' Instruction: 24
Executed 'I' Instruction: 61
Executed 'J' Instruction: 4
Number of Branch Taken: 1
Number of Predict Success: 4
Number of Predict Fail: 1
Number of Memory Access Instruction: 36
***** End of Pipelined Emulator! *****

***** Result *****
Data Hazards : Forwarding
Control Hazards : Detect and wait
Return Value (r2): 10 (0xa)
Total Cycle: 122
Executed 'R' Instruction: 24
Executed 'I' Instruction: 60
Executed 'J' Instruction: 4
Number of Branch Taken: 4
Number of Predict Success: 0
Number of Predict Fail: 0
Number of Memory Access Instruction: 36
***** End of Pipelined Emulator! *****
***** Result *****

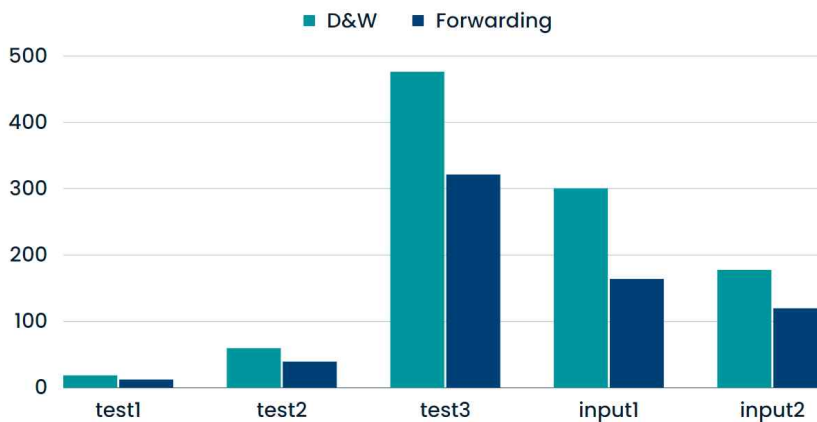
Data Hazards : Forwarding
Control Hazards : Static branch prediction - ANT
Return Value (r2): 10 (0xa)
Total Cycle: 121
Executed 'R' Instruction: 24
Executed 'I' Instruction: 60
Executed 'J' Instruction: 4
Number of Branch Taken: 4
Number of Predict Success: 1
Number of Predict Fail: 4
Number of Memory Access Instruction: 36
***** End of Pipelined Emulator! *****
***** Result *****

Data Hazards : Forwarding
Control Hazards : Static branch prediction - ALT
Return Value (r2): 10 (0xa)
Total Cycle: 115
Executed 'R' Instruction: 24
Executed 'I' Instruction: 61
Executed 'J' Instruction: 4
Number of Branch Taken: 1
Number of Predict Success: 4
Number of Predict Fail: 1
Number of Memory Access Instruction: 36
***** End of Pipelined Emulator! *****

```

그림37. input2.bin 테스트 결과

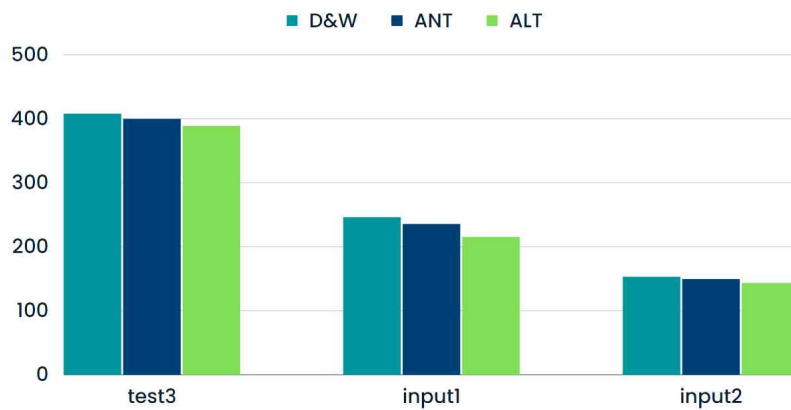
● Data Hazard Cycle Count Analysis : Detect & Wait vs Forwarding



총 Cycle 수	test1.bin	test2.bin	test3.bin	input1.bin	input2.bin
Detect&Wait	18	59	4756	300	177
Forwarding	12	39	3210.33	163.67	119.33

- test3은 값이 커서 그래프에서만 10으로 나눈 값을 비교
- Cycle 수는 해당 항목 별로 3가지 branch prediction 케이스의 Cycle 수 평균값

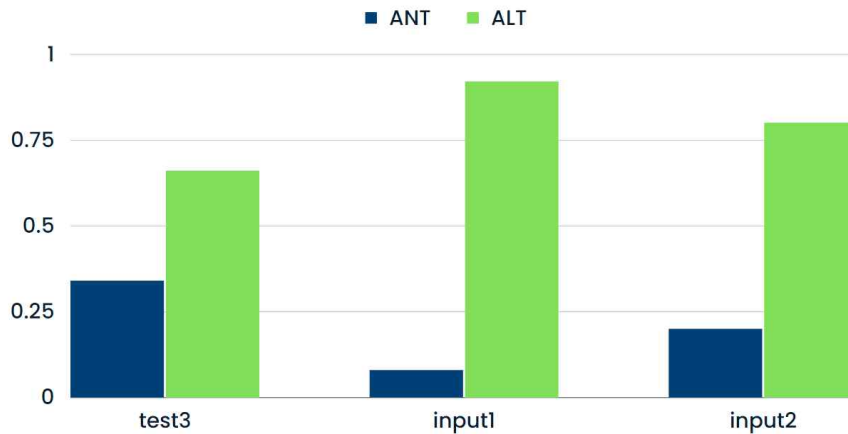
● Control Hazard Cycle Count Analysis : Detect & Wait vs ANT vs ALT



총 Cycle 수	test3.bin	input1.bin	input2.bin
Detect&Wait	4073.5	245.5	152.5
ANT	3992	235	149
ALT	3884	215	143

- test1, test2는 차이가 없어 비교가 무의미하므로 분석 X
- test3은 값이 커서 그래프에서만 10으로 나눈 값을 비교
- Cycle 수는 해당 항목 별로 2가지 data hazard detection 케이스의 Cycle 수 평균값

● Branch Prediction Hit Analysis : Detect & Wait vs ANT vs ALT



예측 성공	test3.bin	input1.bin	input2.bin
ANT	55/164	1/12	1/5
ALT	109/164	11/12	4/5

- test1, test2는 차이가 없어 비교가 무의미하므로 분석 X
- 그래프와 표의 통계치는 Hit(예측성공)한 수 / 총 branch 수

기존에 사용하던 5개의 테스트 코드로 cycle 수와 예측 적중률을 비교해 보았다. 비록 표본의 수가 적어서 정확한 비교는 어렵지만, 직접 구현한 코드로 파이프라인의 성능을 분석해보는 것에 의의를 두었다. 우선 Data Hazard를 해결할 때 detect & wait보다 forwarding 방식이 예상대로 총 필요한 cycle 수가 적다는 결론이 나왔다. test1과 같이 명령어가 적은 경우에는 큰 차이가 없지만, 특히 test3처럼 필요한 명령어 수가 많아질수록 detect & wait와 forwarding의 요구 cycle 차이가 훨씬 증가했다. 다음으로 Control Hazard를 해결할 때 총 cycle 수는 ALT가 가장 적게 나왔고, ANT와 ALT만 비교한 예측 적중률 역시 ALT가 훨씬 높게 나왔다. 이를 보아 detect & wait보다는 static branch prediction의 성능이 더 뛰어나다는 사실을 알 수 있다. 그러나 ALT가 ANT보다 cycle 측면에서도, 적중률 측면에서도 훨씬 뛰어난 모습을 보여줬음에도 ALT가 더 좋다고 확답을 내리기는 어렵다. 프로그램마다 branch prediction의 성능이 다르고 이번 분석에서 표본이 너무 적었기 때문이다. 기회가 된다면 Dynamic branch prediction까지 구현한 다음 코드의 표본 수를 늘려서 더욱 정확한 분석을 시도해 볼 것이다.

5. Lesson

저번 프로젝트에서 명령어 처리를 어떻게 병렬적으로 진행하는지에 대해 궁금했는데, 이번 프로젝트를 진행하며 제대로 공부할 수 있었다. 먼저 Pipelining에 관련한 여러 배경 식들을 공부하며 Pipelined Data Path를 직접 설계해 보았다. 혼자서, 혹은 수업 시간에 습득한 지식을 이용해 이전 Single-Cycle 코드 기반으로 스스로 파이프라이닝 시도를 했다. 마지막으로 이론적으로만 배운 Forwarding, Branch Prediction, Data Dependency Handling을 실제 구현해보며 Hazard들을 해결해 나갔다. 이러한 최적화를 통해 Single-Cycle에서 성능을 직접 향상시켰고, 같은 Pipelined 구조에서도 Hazard를 처리하는 방식을 다르게 하여 성능을 분석해보는 값진 경험이었다. 다만 구현 전에 여러 Branch Prediction들을 공부하며 프로젝트에 모두 구현을 해보겠다는 목표가 있었지만 결국 ANT, ALT로만 Control Hazard를 해결했다. 이 목표를 달성했다면 어떤 Predictor가 어느 상황에서 더 효과적인지를 상세히 분석할 수 있었겠지만 그러지 못해 큰 아쉬움이 남는다.

지난 3달간 3가지의 프로젝트를 모두 완성해 보았는데, 순차적으로 새로운 개념들을 깊이 이해했다. Simple Calculator에서는 대략적인 폰 노이만 구조와 명령어의 처리 과정을, Single-Cycle에서는 하나의 명령어가 어떤 방식으로 처리되어 결과가 나오는지, Pipelined에서는 명령어 처리의 병렬화를 통해 어떻게 성능을 끌어 올리는지를 손수 배울 수 있었다. 또한 코드 구현 관련해서도 많은 교훈을 얻었다. 첫 설계를 탄탄히 해야 이후 코드를 수정하거나 추가할 때 확장성도 좋고 코드 복잡성이 크지 않았었다. 그리고 printf로 잘못된 부분의 디버깅을 수행하거나, 결과를 출력해서 어디에서 코드 진행이 막혔는지를 관찰하고 해결해 보며 코드를 수정하는 테크닉을 얻었다. 이 모든 경험들로 컴퓨터 시스템의 구조, CPU가 명령어와 데이터를 처리하는 방식을 완전히 이해한다는 최종 목표에 도달하였고 절대 잊지 못할 것 같다. 부족한 부분도 많았지만 스스로 크게 성장할 수 있었던 뜻깊은 시간이었다.