<프로젝트3 보고서>

컴퓨터구조와 모바일프로세서

파이프라인 MIPS 리포트

학　　과 : 모바일시스템공학과
학　　번 : 32164809
이　　름 : 탁준석
담당교수 : 유시환
제출일자 : 21.06.03
남은 프리데이 : 0일

목 차

1. 개요

가. 실험 목적

우리는 저번 프로젝트2에서 싱글 사이클 MIPS 마이크로아키텍처를 설계하고 그 코드를 작성하여 추상적인 개념이었던 명령어 집합 구조가 물리적인 하드웨어로 어떻게 구현되는지 학습했다. 이번 프로젝트3에서는 싱글 사이클 MIPS와 비슷한 데이터패스를 사용하지만 처리량이 훨씬 높은 파이프라인 MIPS 마이크로아키텍처를 설계한다. 싱글 사이클의 한계와 파이프라인이 도입된 이유를 알아보고 파이프라인에서 MIPS 명령어 집합이 실행되는 과정을 학습한다. 또한 싱글 사이클에서 발생하지 않았던 세 가지 종속성 문제를 집중적으로 살펴보고, 이것을 해결하는 다양한 방법을 알아본다. 앞선 학습 내용을 충분히 숙지했다면 파이프라인 MIPS 프로그램을 완성해 나가면서 파이프라인을 더욱 심도 있게 이해해보자.

나. 실험 목표

- 싱글 사이클 MIPS 프로그램을 응용하여 파이프라인 MIPS 프로그램을 개발한다.
- 클럭 사이클마다 변화하는 데이터를 출력한다.
- 마지막 클럭 사이클에 최종 결과, 클럭 사이클 수, 명령어 수, 메모리 접근 횟수, 분기한 횟수, 분기하지 않은 횟수, 점프한 횟수를 출력한다.
- 싱글 사이클과 파이프라인의 클럭 사이클을 비교하여 파이프라인 마이크로프로세서의 성능 향상을 확인한다.

종속성	해결 방법
구조적 종속성(Structural dependency)	① Instruction memory/Data memory
데이터 종속성(Data dependency)	① Forwarding/Bypassing
제어 종속성(Control dependency)	① Always not taken ② One level branch prediction ③ Two level global branch prediction ④ Two level gshare branch prediction

<표 1.1> 프로젝트3 파이프라인 MIPS 프로그램에서 구현할 종속성 해결 방법.

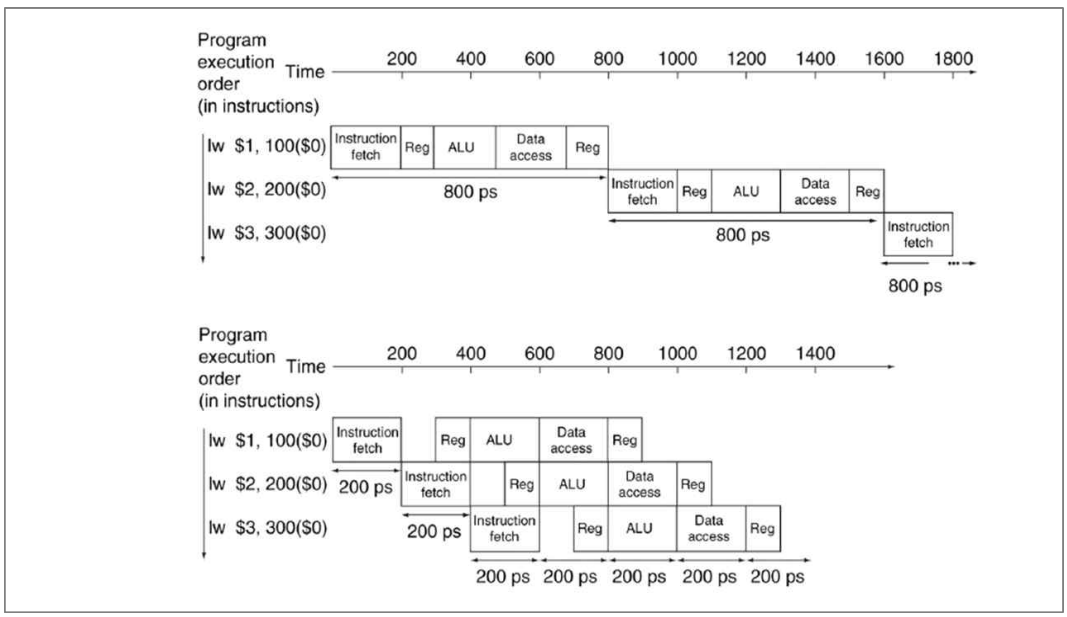
2. 이론

제 2장, 이론에서는 파이프라인에 대한 기본적인 개념을 학습한다. 싱글 사이클의 한계부터 시작하여 파이프라인 단계, 종속성과 그 해결 방법에 대해 알아보자. 학습한 내용을 바탕으로 파이프라인 데이터 패스를 직접 그려나가며 파이프라인 MIPS 프로그램의 초석을 다져보도록 하자.

가. 파이프라이닝(Pipelining)

1) 싱글 사이클의 한계(The Limitation of Single Cycle Design)

저번 프로젝트2에서 설계한 싱글 사이클을 다시 생각해보자. 싱글 사이클은 한 클럭 사이클에 하나의 명령어를 처리하고, 모든 명령어에 대해 클럭 사이클 시간이 같다는 특징을 가지고 있다. 다시 말해, 처리 시간이 서로 다른 명령어를 클럭 사이클이 모두 수용하기 위해서 시간이 가장 오래 걸리는 명령어로 그 길이가 맞춰져 있었다. 가령, 적재 명령어(lw)는 <그림 2.1>와 같이 명령어 메모리, 레지스터 파일, ALU, 데이터 메모리, 그리고 다시 레지스터 파일까지 다섯 개의 하드웨어 유닛에 모두 접근한다. 그러나 적재 명령어를 제외한 다른 명령어는 하드웨어 유닛에 부분적으로 접근하기 때문에 주어진 클럭 사이클 시간을 다 사용하지 않는다. 매우 간단한 명령어 집합을 사용했던 초창기 컴퓨터와 달리, 현대 컴퓨터는 매우 복잡한 명령어 집합을 사용한다. 이런 까다로운 명령어 집합을 고정된 클럭 사이클, 그것도 오직 직렬로 처리하는 것은 심각한 비효율을 초래한다. 이 분명한 한계를 극복하기 위해 '파이프라인'이라는 또 다른 구현 기술이 제시되었는데, 다음 '파이프라인 개관'에서 이것을 자세히 살펴보도록 하자.



<그림 2.1> 싱글 사이클과 파이프라인의 적재 명령어 처리 시간 비교. 클석 사이클 시간이 싱글 사이클은 800ps, 파이프라인은 200ps이다. 따라서 세 개의 적재 명령어를 처리하는데 싱글 사이클은 2400ps가 걸리고 파이프라인은 1400ps가 걸린다.

2) 파이프라인 개관(Pipleline Overview)

오늘날 파이프라인은 현대에 들어 매우 보편적인 기술로 여러 명령어가 중첩되어 실행되는 구현 기술을 말한다. 한 클럭 사이클에 하나의 명령을 실행하는 싱글 사이클과 달리, 파이프라인은 클럭 사이클에 여러 명령어를 병렬로 실행하여 처리량을 증가시킴으로써 성능을 향상한다. 단, 주의해야할 점이 있다. 파이프라인은 개별 명령어의 처리 시간을 줄이지 않는다. <그림 2.1>의 두 번째 그래프와 같이 파이프라인이 싱글 사이클보다 더 빠른 이유는 모든 명령어가 병렬로 동작하여 같은 시간에 더 많은 명령어를 처리할 수 있기 때문이다. 그렇다면 파이프라인은 싱글 사이클보다 얼마만큼 속도가 향상될까? 파이프라인 역시 싱글 사이클과 마찬가지로 한 명령어를 처리하는데 다섯 단계가 걸린다. 만약 처리할 명령어가 아주 적다면 그 성능 향상은 미미하겠지만, 실행할 명령어가 점점 많아질수록 성능은 올라갈 것이다. 이상적인 조건에 파이프라인에 의한 속도 향상은 명령어 처리 단계 수와 거의 같으므로, 다섯 단계를 사용하는 파이프라인은 약 5배 정도의 성능 향상을 기대할 수 있을 것이다.

3) 파이프라인 단계(Pipeline Stage)

파이프라인을 설계하기 위해서 우리는 파이프라인에서 MIPS 명령어가 어떻게 처리되는지 알아야 한다. 다행스럽게도 명령어를 처리하기 위한 파이프라인 단계는 싱글 사이클과 마찬가지로 다섯 단계이다. 한 클럭 사이클이 모든 단계를 커버하는 싱글 사이클과 달리, 파이프라인은 한 클럭 사이클이 한 단계만을 커버한다. 쉽게 말해 파이프라인에서는 명령어A가 파이프라인 1단계를 지나 2단계로 넘어갈 때, 다음 명령어B가 1단계로 들어온다. 마찬가지로 명령어 A가 2단계를 지나 3단계로 넘어갈 때, 다음 명령어B는 1단계에서 2단계로 넘어가고, 다다음 명령어C가 1단계로 들어온다.

1. Instruction fetch
   명령어를 가져오기 위해서 명령어 메모리(Instruction Memory)에 프로그램 카운터(PC)를 보낸다.

2. Instruction decode and register operand fetch
   PC가 가리키는 명령어 워드를 분석해 사용할 레지스터를 선택하고, 선택된 레지스터를 읽는다.

3. Execute and evaluate memory address
   명령어가 지시하는 명령을 ALU를 통해 계산한다. 메모리 참조 명령어는 유효주소를 계산하기 위해, 산술논리 명령어는 opcode의 실행을 위해, 분기 명령어는 비교를 위해 ALU를 사용한다.

4. Memory operand fetch
   메모리를 참조하는 명령어는 저장을 완결하기 위해 또는 로드될 워드를 읽기 위해 데이터를 가지고 있는 메모리(Data Memory)에 접근한다.

5. Store and write back result
   산술논리 명령어는 계산한 결과를 쓰기 레지스터에 저장한다. 메모리를 참조하는 명령어는 메모리에 계산한 결과를 저장한다.

나. 파이프라인 종속성(Pipeline Dependency)

파이프라인은 명령어를 병렬로 처리하면서 이전에는 나타나지 않았던 문제가 발생한다. 바로 다음 명령어가 다음 클럭 사이클에 실행될 수 없는 경우이다. 이러한 경우를 종속성(Dependency) 또는 해저드(Hazard)라고 부르고 크게 구조적 종속성, 데이터 종속성, 그리고 제어 종속성으로 구분할 수 있다.

1) 구조적 종속성과 해결(Structural Dependency and Solutions)

첫 번째 종속성은 구조적 종속성(Structural dependency)이다. 우리는 저번 프로젝트2에서 명령어 메모리와 데이터 메모리가 합쳐진 하나의 메모리를 사용한다고 간주하였다. 즉, 명령어 메모리와 데이터 메모리를 따로 구분하지 않았다. 그런데 메모리를 구분하지 않으면 같은 리소스를 여러 명령어가 사용하려고 하여, 같은 클럭 사이클에 실행해야 하는 명령어를 하드웨어가 지원하지 못하는 상황이 발생한다. 예를 들어, 첫 번째 명령어가 적재 명령어(lw)이고 네 번째 명령어가 덧셈 명령어(add)라고 가정해보자. 첫 번째 적재 명령어는 데이터를 저장하기 위해 데이터 메모리에 접근하고, 네 번째 명령어는 명령어를 인출하기 위해 명령어 메모리에 접근하는데, 메모리가 하나이기 때문에 동시에 접근하는 게 불가능한 상황을 마주하게 된다. 그렇다면 하드웨어 수준에서 구조적 종속성은 어떻게 해결할 수 있을까?

구조적 종속성을 해결하는 방법으로는 두 가지를 제시할 수 있다. 첫 번째, 지연(Stall)한다. 데이터 메모리를 사용할 때마다 명령어 처리를 한 단계 지연하여 메모리를 동시에 사용하는 상황을 피하는 것이다. 지연 방법은 적은 비용으로 구현하기가 쉽다는 장점이 있다. 그러나 수억 개의 명령어를 처리하는 현대 컴퓨터에서 CPI를 증가시켜 성능을 저하한다는 분명한 단점이다. 두 번째, 명령어 메모리와 데이터 메모리를 서로 분리한다. 해당 메모리에 각각 접근하도록 만드는 이 방법은 하나의 메모리에 비해 비용이 증가한다는 단점이 있지만, 클럭 사이클을 낭비하지 않아 성능을 증가시키는 압도적인 장점이 있다. 이번 프로젝트3에서는 두 번째 방법을 사용하여 구조적 종속성을 이론적으로 해결하고자 한다.

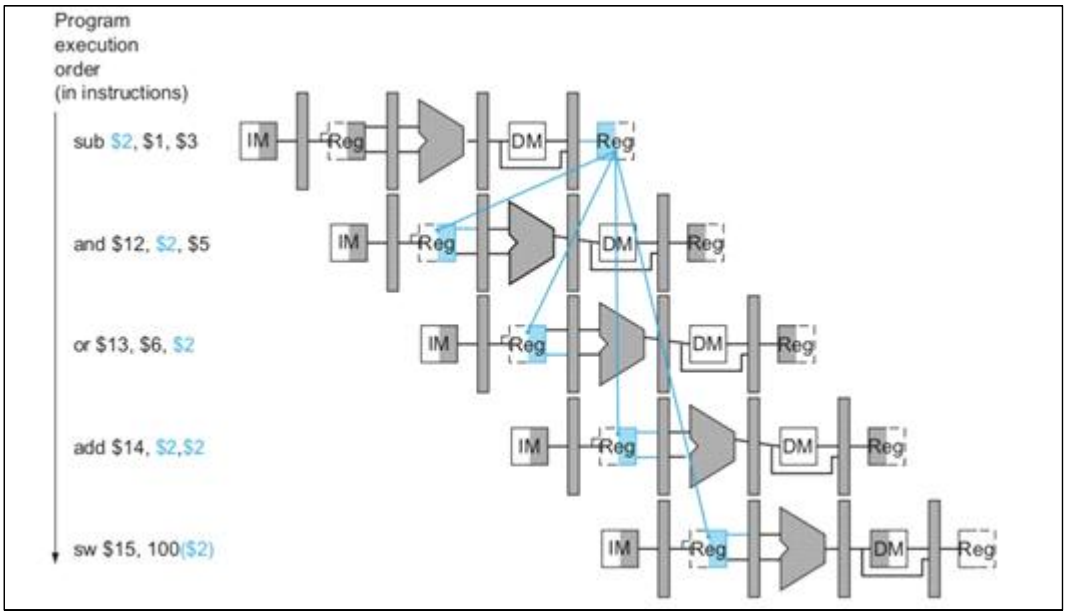
2) 데이터 종속성(Data dependency)

두 번째 종속성은 데이터 종속성(Data dependency)이다. 데이터 종속성은 명령어를 실행하는데 필요한 데이터가 아직 준비되지 않아서 명령어가 제때 실행될 수 없는 상황을 말한다. 데이터 종속성이 발생하는 경우는 다음 <표 2.1>과 같이 세 가지로 구분할 수 있다. 파이프라인에서 문제되는 데이터 종속성은 Read after Write로 어떤 명령어가 앞선 명령어로부터 적절한 데이터를 받지 못하는 경우에 발생한다.

데이터 종속성	예시	설명
Read after Write (RAW)	$R_3 \leftarrow R_1\ operation\ R_2$ $R_5 \leftarrow R_3\ operation\ R_4$	Flow dependence 상황으로 메모리에 쓰기를 하기 전에 다음 명령어가 읽기를 하려고 할 때 발생한다.
Write after Read (WAR)	$R_3 \leftarrow R_1\ operation\ R_2$ $R_1 \leftarrow R_4\ operation\ R_5$	Anti dependence 상황으로 메모리에서 읽기를 하기 전에 다음 명령어가 쓰기를 하려고 할 때 발생한다.
Write after Write (WAW)	$R_3 \leftarrow R_1\ operation\ R_2$ $R_5 \leftarrow R_3\ operation\ R_4$ $R_3 \leftarrow R_6\ operation\ R_7$	Output-dependence 상황으로 두 명령어가 같은 위치에 쓰기를 하려고 할 때 발생한다.

<표 2.1> 데이터 종속성의 종류와 그 예시. 파이프라인에서 문제가 발생하는 데이터 종속성은 Read after Write이다.

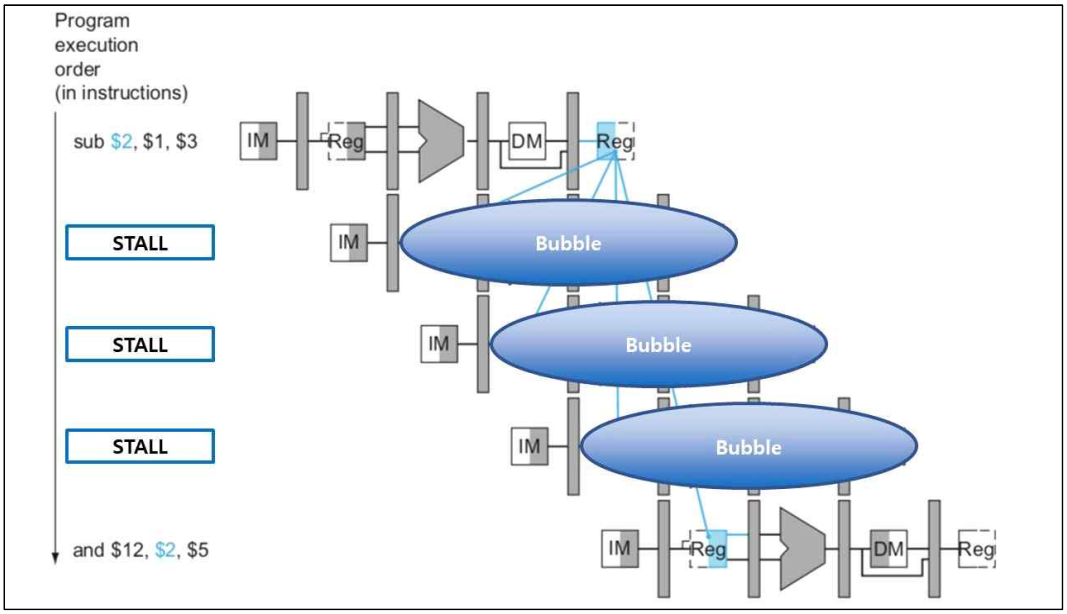
파이프라인 데이터패스에서 나타날 수 있는 RAW 데이터 종속성은 세 가지로 구분할 수 있는데 각각 EX 종속성, MEM 종속성, WB 종속성이다. <그림 2.2>는 파이프라인에서 발생할 수 있는 데이터 종속성을 나타낸 것으로 sub 명령어와 종속성을 가진 명령어를 파란 선으로 이어서 보여주고 있다. $2는 sub 명령어의 5단계(Write Back) 이후에 올바른 데이터가 입력되는데 and 명령어는 sub 명령어의 3단계(Execution)에서, or 명령어는 sub 명령어의 4단계(Memory Access)에서, 그리고 add 명령어는 sub 명령어의 5단계에서 피연산자 $2를 요구하는 종속성이 발생한다. 이렇게 한 단계 앞선 명령어의 3단계에서 발생하는 종속성을 EX 종속성, 두 단계 앞선 명령어의 4단계에서 발생하는 종속성을 MEM 종속성, 세 단계 앞선 명령어의 5단계에서 발생하는 종속성을 WB 종속성이라고 정의하겠다.



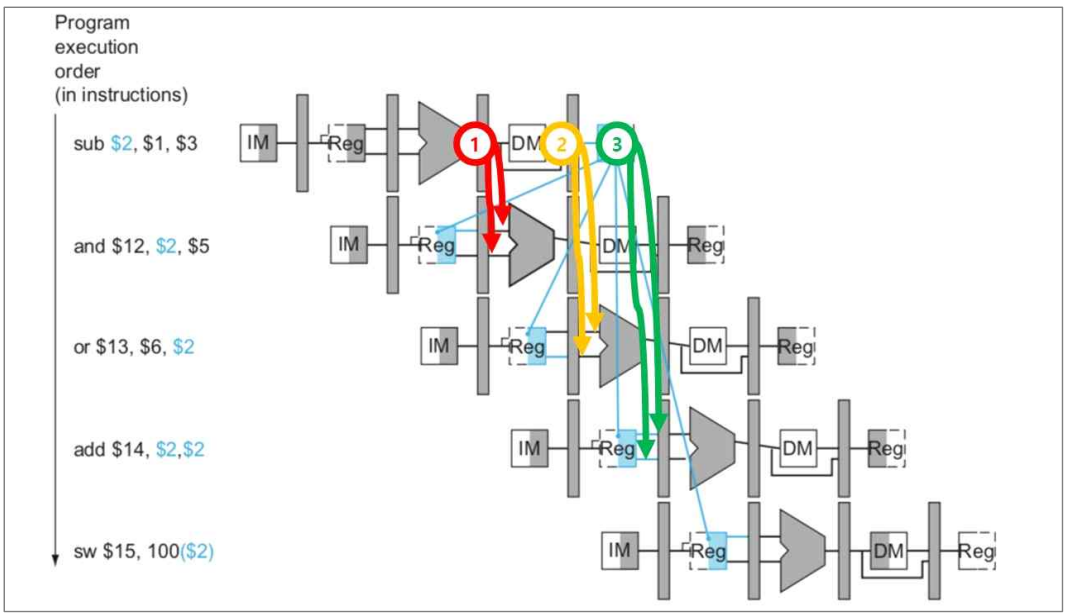
<그림 2.2> sub 명령어의 $2와 종속성을 가지고 있는 세 가지 명령어 and, or, add. sub 명령어가 $2에 쓰기 전에 뒤따라오는 명령어가 $2를 읽으면 올바르지 못한 $2가 전달된다. sw 명령어는 sub 명령어가 모두 처리되고 2단계를 실행하므로 $2와의 종속성이 발생하지 않는다.

그렇다면 하드웨어 수준에서 데이터 종속성은 어떻게 해결할 수 있을까? 데이터 종속성을 해결하는 방법으로는 두 가지를 제시할 수 있는데, 첫 번째는 지연(Stall)이다. 사실, 지연은 해결보다 회피에 더 가까운 방법으로 데이터 종속성이 발생할 때마다 다음 명령어의 실행을 모두 멈추고, 종속성을 가진 레지스터에 올바른 데이터가 입력될 때까지 기다리는 것이다. <그림 2.3>를 살펴보면 레지스터는 5단계(Write Back) 이후에 올바른 데이터가 입력되므로 EX 종속성은 세 사이클, MEM 종속성은 두 사이클, WB 종속성은 한 사이클을 지연해야 한다. 그러나 데이터 종속성은 너무 자주 일어나고 그때마다 파이프라인을 지연시키는 행위는 성능 향상을 위해 파이프라인을 도입한 우리의 의도와 반대된다. 지연을 하지 않고 데이터 종속성을 해결할 수 있는 방법은 없을까?

데이터 종속성을 해결하는 또 다른 매력적인 방법은 하드웨어를 추가해 데이터를 전방전달(Forwarding) 또는 우회전달(Bypassing)하는 것이다. 이 방법은 명령어 처리가 끝날 때까지 다음 명령어가 기다릴 필요가 없다는 사실에 기반을 두고 있다. <그림 2.4>를 살펴보면 sub 명령어에서 빨간색, 노란색, 그리고 초록색 화살표가 종속성이 있는 데이터를 다음 명령어에 미리 보내는 모습을 볼 수 있다. 이렇게 다음 명령어를 처리하는데 필요한 데이터를 이전 명령어가 파이프라인 단계 중간에 미리 보내줄 수 있다면 지연이라는 비효율적인 행위를 하지 않아도 된다. 이번 프로젝트3에서는 두 번째 방법을 사용하여 데이터 종속성을 해결하고자 한다.



<그림 2.3> 세 단계 지연을 통해 and, or, add 명령어와의 데이터 종속성을 회피. $2와 종속성이 있던 명령어가 지연되어 다섯 번째 명령어가 sw에서 and 명령어로 바뀌었다. 종속성을 피하기 위한 지연을 거품(Bubble)이라고 부른다.



<그림 2.4> 데이터 종속성을 해결하는 전방전달 연결. 빨간 동그라미와 화살표는 EX 종속성을 해결하는 전방전달, 노란 동그라미와 화살표는 MEM 종속성을 해결하는 전방전달, 그리고 초록 동그라미와 화살표는 WB 종속성을 해결하는 전방전달이다.

3) 제어 종속성(Control dependency)

세 번째 종속성은 제어 종속성(Control dependency)이다. 제어 종속성은 다른 명령어들이 실행 중에 어떤 명령어의 결과에 기반을 둔 결정을 할 필요가 있을 때 나타나는데, 이 파이프라인에서 결정 작업에 해당하는 것은 바로 분기 명령어이다. 파이프라인 마이크로프로세서가 분기 명령어를 처리하기 위해서 우리는 어떤 방법으로 접근해야 할까? 우선, 다음 세 가지 질문을 깊게 고민해보자.

1. 분기 명령어인가?
   명령어 메모리에서 인출된 명령어가 분기 명령어인지 아닌지 모른다.

2. 분기 명령어라면, 분기를 하는가?
   분기 명령어라면 분기를 하는지, 안 하는지 아직 모른다.

3. 분기를 한다면, 분기 주소는 어떻게 되는가?
   분기를 한다면 분기 주소가 어떻게 되는지 아직 모른다.

하드웨어 수준에서 제어 종속성은 어떻게 해결할 수 있을까? 물론 제어 종속성을 100% 해결하는 방법은 역시 지연(Stall)이다. 모든 분기 명령어에 대해 분기를 하는지 확인하고, 분기를 한다면 분기 주소를 계산할 때까지 클럭 사이클을 잠시 멈추는 것이다. 하지만 이 방법은 속도 저하라는 지불해야할 대가가 너무 크다. 위의 세 가지 질문을 고려했을 때, 데이터 종속성을 해결하는 두 번째 방법은 예측(Prediction)이다. 이것은 다음 명령어의 주소를 미리 예측하겠다는 의도에서 출발한다. 예측이 옳으면 정상적으로 진행하고, 예측이 틀리면 처리 중인 일부 명령어를 버린다. 분기 예측은 정적 분기 예측(Static branch prediction)부터 동적 분기 예측(Dynamic branch prediction)까지 그 확률을 올릴 수 있는 다양한 방법이 존재하지만 이번 프로젝트3에서는 Always not taken과 One level branch prediction, Two level Global branch prediction, Two level Gshare branch prediction 방법을 구현하고자 한다.

정적 분기 예측인 Always not taken은 항상 분기를 하지 않는다고 예측한다. 명령어가 1단계(Instruction Fetch)에 들어오면 분기를 무조건 하지 않는다고 예측하고, 3단계(Execution)에 가서 분기를 하지 않았는지 확인한다.

반면에 동적 분기 예측은 'Branch Target Buffer(BTB)', 'Pattern History Table(PHT)', 그리고 'Global History Register(GHR)'라는 세 가지 새로운 자료구조를 도입하여 분기를 예측한다. BTB는 분기 명령어의 PC와 분기할 주소(Branch Target)를 저장해 분기할 주소를 미리 예측할 수 있게 한다. PHT는 2비트 카운터로 0부터 3까지 해당하는 패턴을 저장한다. 0은 strongly not taken, 1은 weakly not taken, 2는 weakly taken, 3은 strongly taken을 의미하고 분기 명령어의 분기 여부에 따라 숫자를 1만큼 증가시키거나 감소시킨다. GHR은 어떤 분기 명령어의 분기 여부는 지난 분기 명령어의 분기 여부에 영향을 받는다는 사실을 바탕으로 제시된 레지스터로, 지난 분기 이력을 저장하여 PHT의 인덱스로 사용된다.

BTB

Index	PC	Branch Target
1	0x10	0x1234
2	0x11	0x5678
3	0x12	0x9abc
4	0x13	0xdef1
5	...	...

PHT

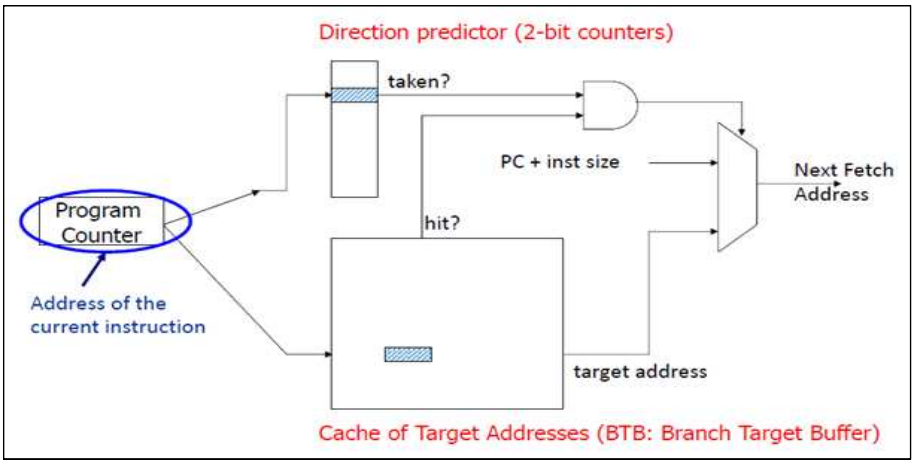
Index	2bit counter
1	0b00
2	0b01
3	0b11
4	0b10
...	...

<표 2.3> BTB와 PHT을 도식화. BTB는 3단계(Execution)에서 분기 명령어가 분기를 한 경우, 분기 명령어의 PC와 Branch Target을 저장한다. PHT는 3단계에서 분기 명령어의 분기 여부에 따라 2비트 카운터를 1만큼 증가시키거나 감소시킨다.

GHR	0	0	0	0	0	1
GHR	0	0	0	0	1	1

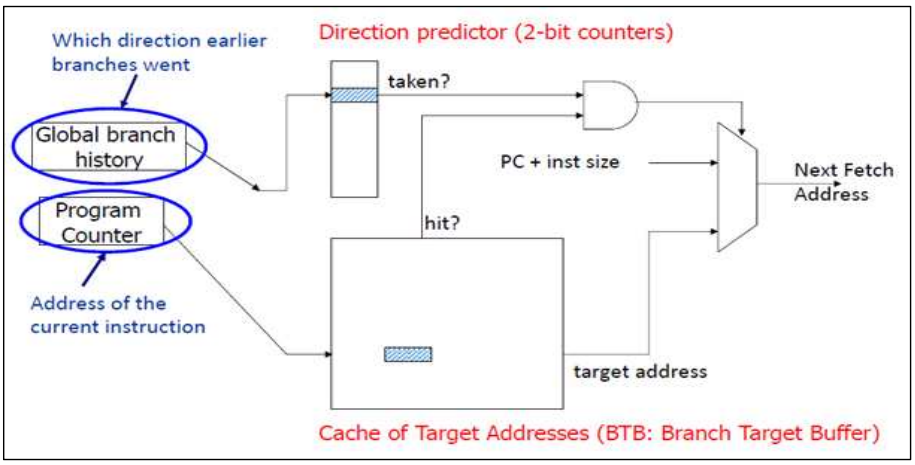
<표 2.4> GHR을 도식화. GHR은 오른쪽으로 1만큼 비트 연산하고 분기 명령어의 분기 여부에 따라 0을 더하거나 1을 더한다. 이 표는 6비트 GHR이 분기하는 분기 명령어를 만난 상황을 보여준다.

첫 번째 동적 분기 예측인 One level branch prediction은 프로그램 카운터로 분기 예측 버퍼와 패턴 이력표에 접근하여 분기 여부를 예측한다.



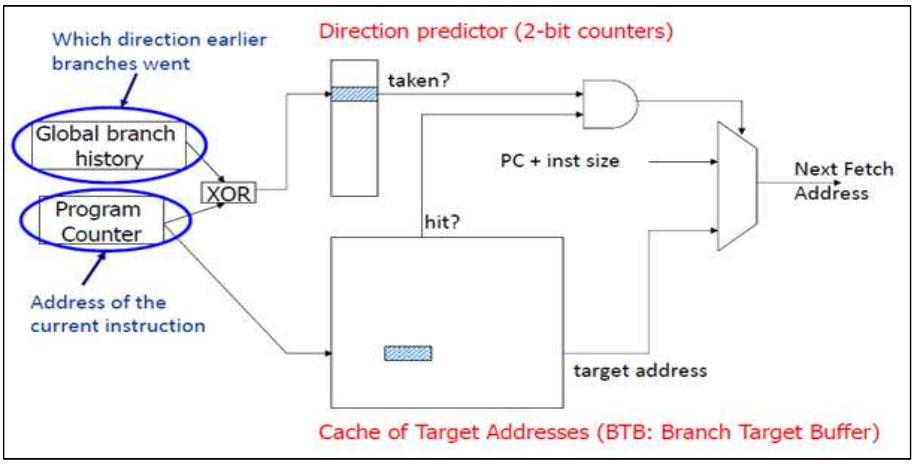
<그림 2.5> One level branch prediction을 도식화. 프로그램 카운터가 PHT, BTB의 인덱스로 사용된다.

두 번째 동적 분기 예측인 Two level Global branch prediction은 프로그램 카운터로 Branch Target Buffer에 접근하고, 최근 분기를 반영한 Global Branch History로 Pattern History Table에 접근하여 분기 여부를 예측한다.



<그림 2.6> Two level global branch prediction을 도식화. 프로그램 카운터가 BTB의 인덱스로, GHR가 PHT의 인덱스로 사용된다.

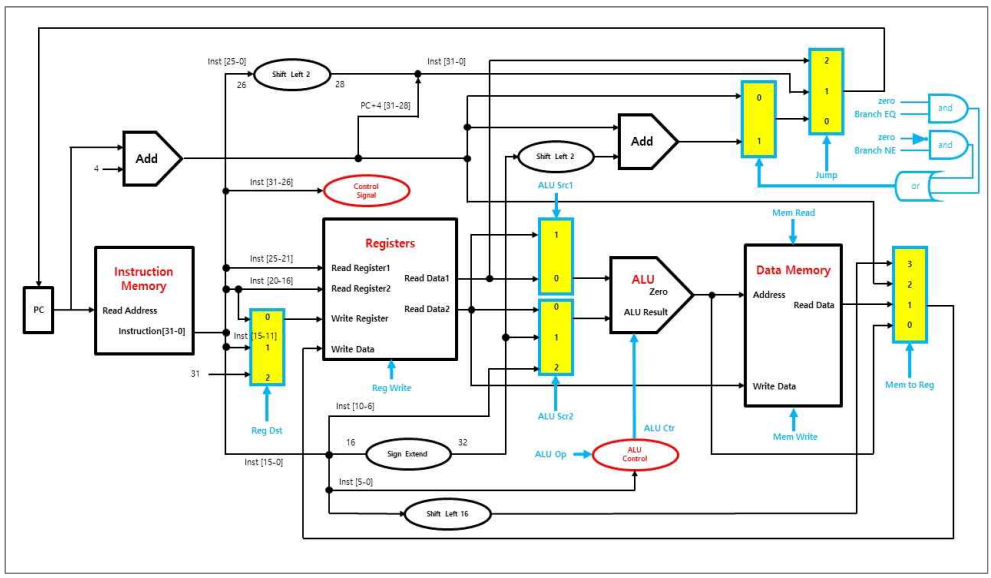
세 번째 동적 분기 예측인 Two level Gshare branch prediction은 프로그램 카운터로 Branch Target Buffer에 접근하고, 최근 분기를 반영한 Global Branch History와 프로그램 카운터를 XOR하여 Pattern History Table에 접근함으로써 분기 여부를 예측한다.



<그림 2.7> Two level gshare branch prediction을 도식화. 프로그램 카운터가 BTB의 인덱스로, GHR과 프로그램을 XOR 연산하여 PHT의 인덱스로 사용된다.

다. 파이프라인 설계(Pipeline Design)

지금까지 파이프라인과 종속성 개념들을 알아봤다면 이제는 파이프라인을 직접 설계할 차례이다. 파이프라인 설계를 시작하는 가장 합리적인 방법은 저번 싱글 사이클에서 MIPS 명령어를 처리하는데 사용했던 데이터패스를 바탕으로 시작하는 것이다. 파이프라인을 실행하는데 추가해야할 몇 가지 하드웨어에 대해 알아보고, 명령어 처리 중 발생하는 문제를 해결해보자. 다음으로 노트와 펜을 가지고 데이터패스를 알맞게 수정하면서 싱글 사이클 데이터패스가 파이프라인 데이터패스로 어떻게 바뀌는지 살펴보자. 본격적으로 파이프라인을 설계하기 전에 <그림 2.8>을 보며 프로젝트2에서 사용했던 싱글 사이클 데이터패스를 다시 한 번 상기하길 바란다.

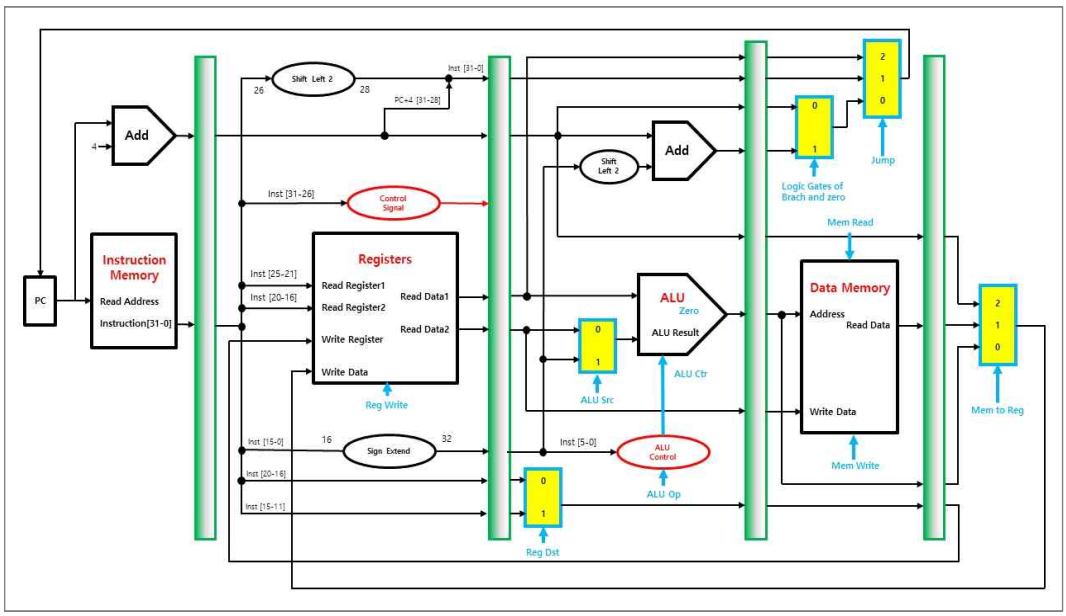


<그림 2.8> 프로젝트2에서 사용했던 싱글 사이클 데이터패스. 메모리, 레지스터 파일, ALU, 멀티플렉서, 가산기 등 하드웨어와 제어 신호를 확인할 수 있다.

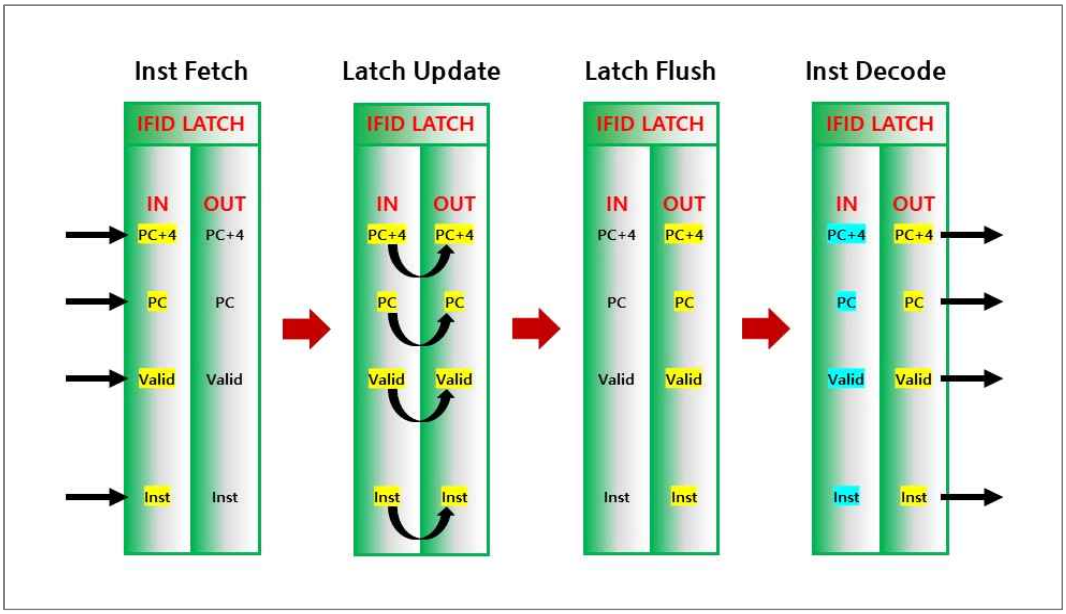
1) 레치(Latch)

한 클럭 사이클에 하나의 명령어를 실행하는 싱글 사이클은 다른 명령어의 간섭을 받는 경우가 없었다. 다음 명령어는 현재 실행되고 있는 명령어가 모두 처리되고 난 후에 실행되었기 때문이다. 하지만 파이프라인은 앞서 살펴봤던 파이프라인 단계에 따라 한 클럭 사이클에 최대 다섯 개의 명령어가 동시에 실행되기 때문에 각 단계를 명확하게 구분하지 않으면 명령어 간의 데이터가 서로 충돌할 가능성이 있다. 따라서 파이프라인은 실행 중인 명령어가 한 단계에서 다음 단계로 넘어가기 전, 각 단계를 구분하는 저장소에 해당 명령어의 데이터를 임시로 저장해야 한다. 그렇지 않으면 다음 명령어가 그 파이프라인 단계에 들어올 때 이전 데이터를 덮어버리기 때문이다.

우리는 파이프라인 단계에 따라 데이터패스를 다섯 부분으로 구분해 그 사이마다 레치를 삽입할 것이다. 레치(Latch) 또는 파이프라인 레지스터(Pipeline register)는 파이프라인 단계 사이에 존재하는 임시 저장소로, 다음 단계를 실행하기 위해 이전 명령어의 데이터를 저장하는 역할을 한다. 레치는 데이터를 받아들이는 부분과 데이터를 내보내는 부분으로 나누어지는데 편의상 전자를 IN 레치, 후자를 OUT 레치라고 정의하자. 한편, 레치는 해당 레치가 구분하고 있는 단계에 따라 이름이 붙여진다. 가령, 1단계(Insturuction Fetch)와 2단계(Instruction Decode) 사이의 레치는 IFID 레치, 2단계와 3단계(Execution) 사이의 레치는 IDEX 레치라고 부른다. <그림 2.9>은 레치가 초록색으로 강조된 파이프라인 데이터패스를 보여준다. 우리는 이 그림에서 IFID, IDEX, EXMEM, MEMWB에 해당하는 레치를 확인할 수 있다.



<그림 2.9> 명령어 처리 단계 사이마다 레치가 추가된 파이프라인 데이터패스. 왼쪽에서 오른쪽 순서대로 IFID, IDEX, EXMEM, MEMWB 레치가 파이프라인 단계를 구분하고 있다.



<그림 2.10> 클럭 사이클마다 IFID 레치에서 일어나는 변화. 명령어의 데이터가 레치 안에서 이동하는 과정은 노란색 배경으로 칠해져 있다. 하늘색 배경으로 칠해진 데이터는 새로운 명령어의 데이터이다.

파이프라인에서 실행 중인 모든 명령어는 다음 단계가 시작되어 새로운 명령어가 인출되기 전에 자신의 상태를 레치에 갱신해야 한다. <그림 2.10>는 MIPS 명령어가 1단계에서 2단계로 넘어가는 동안 IFID 레치에 일어나는 변화를 나타낸 것이다. 어떤 명령어가 1단계를 모두 실행하면, 다음 단계에 사용할 데이터를 IFID IN 레치에 저장한다. 데이터를 전달받은 IFID IN 레치는 IFID OUT 레치에 데이터를 전달하고, 다음 명령어가 레치에 데이터를 저장할 수 있도록 IFID IN 레치를 0으로 깔끔하게 비워둔다. IFID OUT 레치로 옮겨진 데이터는 2단계를 실행하는데 사용되고, 새로운 명령어의 데이터는 다시 IFID IN 레치에 저장된다. 다섯 단계 파이프라인 설계는 모두 네 개의 레치를 사용하는데, 방금 전 설명한 IFID 레치와 마찬가지로 다른 세 개 역시 동일한 과정을 거쳐 데이터를 다음 단계로 전달한다. 단, IN 레치의 데이터를 OUT 레치로 옮기는 레치 업데이트(Latch Update)와 IN 레치를 0으로 비우는 플러시(Latch Flush)는 클럭 사이클의 맨 마지막에 한꺼번에 수행하여 클럭 사이클 중간에 데이터가 전달되는 불상사가 없도록 만들 것이다.

2) 멀티플렉서(Multiplexer)

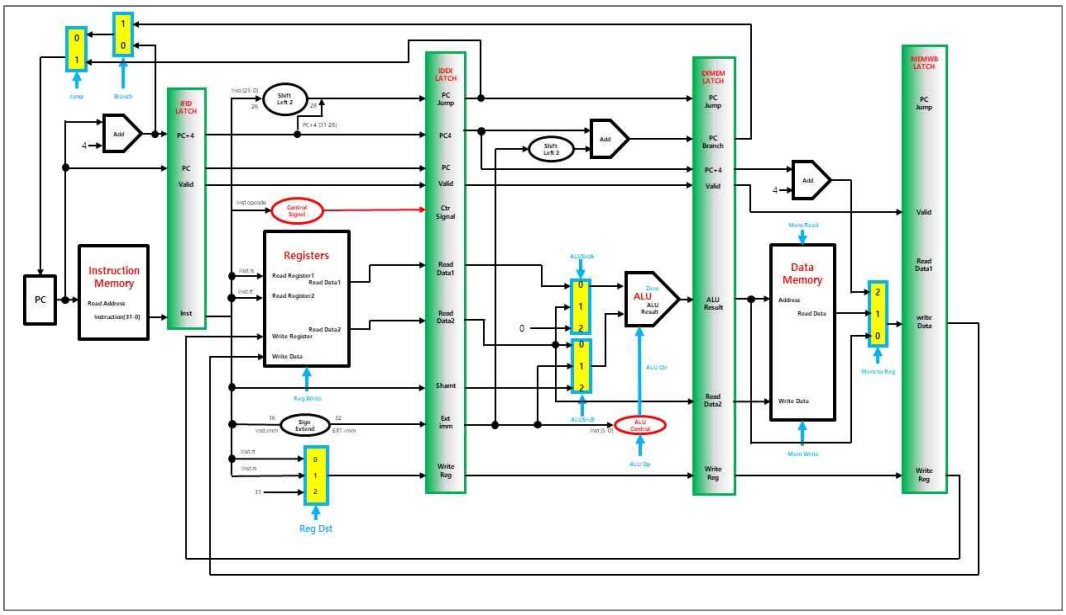
멀티플렉서(Multiplexer, MUX)는 여러 입력 신호들 가운데 하나의 신호를 선택해 출력 신호로 내보내는 논리 회로를 말한다. 우리는 싱글 사이클을 설계할 때부터 MIPS 명령어의 종류에 따라 데이터패스의 모양이 조금씩 다르다는 것을 확인했다. 그리고 데이터를 하나로 통합하는 과정에서 서로 다른 명령어가 데이터패스의 구성 요소를 공유하기 위해 여러 입력 데이터를 하나로 연결하는 멀티플렉서를 사용했다. 파이프라인 데이터패스 역시 싱글 사이클 데이터패스를 바탕으로 하고 있으므로 싱글 사이클 데이터패스에서 사용했던 멀티플렉서를 거의 그대로 가져와 사용할 것이다.

한편, 저번 프로젝트2에서 load upper immediate(lui) 명령어를 해결하기 위해 <그림 2.9>와 같이 ALU를 거치지 않고 { immediate << 16 } 하드웨어를 거쳐 MemtoReg 멀티플렉서로 결과를 내보냈었다. 이번 프로젝트3에서는 복잡한 데이터패스를 간소화하기 위해 lui 명령어를 위한 비트 연산 하드웨어 { immediate << 16 }를 Sign extend 하드웨어와 통합하고 ALUSrcA 멀티플렉서에 제어 신호 2번으로 주어 ALU에서 0과 add 연산한 결과를 내보내도록 바꾸었다. 따라서 ALUSrcA 멀티플렉서에 제어 신호 2

번으로 하는 입력 데이터 0을 추가하고, MemtoReg 멀티플렉서에 제어 신호를 3번으로 하는 입력 데이터 { immediate << 16 }를 삭제하였다. 이번 파이프라인 데이터패스에 새롭게 추가된 멀티플렉서는 전방전달과 관련있는ForwardA, ForwardB, ForwardC 그리고 ForwardD 멀티플렉서이다. 각 멀티플렉서의 입력 데이터와 제어 신호는 다음 <표 2.3>와 같다. 전방전달을 위한 멀티플렉서는 전방전달 유닛이 추가된 <그림 2.13>에서 확인할 수 있다.

멀티플렉서	제어 신호	입력 데이터	번호
MUX_RegDst	CTR.RegDst	INST.rt	0
		INST.rd	1
		31	2
MUX_ForwardA	Forward.SelectA	IDEX_OUT.readData1	0
		EXMEM_OUT.ALUresult	1
		MEMWB_OUT.writeData	2
MUX_ForwardB	Forward.SelectB	IDEX_OUT.readData2	0
		EXMEM_OUT.ALUresult	1
		MEMWB_OUT.writeData	2
MUX_ForwardC	Forward.SelectC	REG.readData1	0
		MEMWB_OUT.writeData	1
MUX_ForwardD	Forward.SelectD	REG.readData2	0
		MEMWB_OUT.writeData	1
MUX_ALUSrcA	IDEX_OUT.ALUSrcA	ALU.operand1	0
		ALU.operand2	1
		0	2
MUX_ALUSrcB	IDEX_OUT.ALUSrcB	ALU.operand2	0
		IDEX_OUT.extimm	1
		IDEX_OUT.shamt	2
MUX_MemtoReg	EXMEM_OUT.MemtoReg	EXMEM_OUT.ALUresult	0
		DataMem.readData	1
		EXMEM_OUT.PC4 + 4	2
MUX_Branch	EXMEM_OUT.Branch	PC	0
		EXMEM_OUT.PC_Branch	1
MUX_Jump	EXMEM_OUT.Jump	PC	0
		IDEX_OUT.PC_Jump	1

<표 2.3> 파이프라인 데이터패스에서 사용된 열 가지 멀티플렉서의 입력 데이터와 제어 신호. 싱글 사이클 데이터패스에서 사용되었던 여섯 가지 멀티플렉서에 전방전달을 위한 네 가지 멀티플렉서가 추가되었다.



<그림 2.11> 멀티플렉서와 제어 신호가 추가된 파이프라인 데이터패스. 싱글 사이클 데이터패스와 다르게 노란 배경으로 칠해진 멀티플렉서의 위치가 조금 바뀌었다. 이 그림에서는 전방전달과 관련된 멀티플렉서가 아직 포함되어 있지 않다.

3) 제어 논리(Control logic)

가) 제어 신호(Control signals)

파이프라인 데이터패스에서 사용된 멀티플렉서, 데이터 메모리, 그리고 레지스터 파일 쓰기는 명령어에 따라 조금씩 다른 제어 신호를 가진다. 따라서 명령어가 올바른 데이터패스에 따라 처리되도록 하기 위해 제어 논리를 적절하게 도입해 그 흐름을 통제해야 한다. 파이프라인 제어 신호는 제어부(Control Unit)가 통제하는 RegDst, ALUSrcA, ALUSrcB, MemRead, MemWrite, MemtoReg, RegWrite, Branch, Jump 제어 신호와 전방전달 유닛(Forwarding Unit)이 통제하는 SelectA, SelectB, SelectC, SelectB 제어 신호로 구성되어 있다. 다음 <표 2.4>는 파이프라인 데이터패스에서 사용된 제어 신호와 그 효과를 나타낸다.

신호	거짓 효과	참 효과	
	0	1	2
RegDst	레지스터 목적지 번호가 INST.rt로부터 주어진다.	레지스터 목적지 번호가 INST.rd로부터 주어진다.	레지스터 목적지 번호가 31이다.
SelectA	ALU의 첫 번째 중간 피연산자가 레지스터 파일의 첫 번째 출력으로부터 주어진다.	ALU의 첫 번째 중간 피연산자가 Execution 단계의 전방전달로부터 주어진다.	ALU의 첫 번째 중간 피연산자가 Memory Access 단계의 전방전달로부터 주어진다.
SelectB	ALU의 두 번째 중간 피연산자가 레지스터 파일의 두 번째 출력으로부터 주어진다.	ALU의 두 번째 중간 피연산자가 Execution 단계의 전방전달로부터 주어진다.	ALU의 두 번째 중간 피연산자가 Memory Access 단계의 전방전달로부터 주어진다.
SelectC	ALU의 첫 번째 피연산자가 레지스터 파일의 첫 번째 출력으로부터 주어진다.	ALU의 첫 번째 피연산자가 Write Back 단계의 전방전달로부터 주어진다.	

SelectD	ALU의 두 번째 피연산자가 레지스터 파일의 두 번째 출력으로부터 주어진다.	ALU의 두 번째 피연산자가 Write Back 단계의 전방전달로부터 주어진다.	
ALUSrcA	ALU의 첫 번째 피연산자가 SelectA MUX의 출력으로부터 주어진다.	ALU의 첫 번째 피연산자가 SelectB MUX의 출력으로부터 주어진다.	ALU의 첫 번째 피연산자가 0이다.
ALUSrcB	ALU의 두 번째 피연산자가 SelectB MUX의 출력으로부터 주어진다.	ALU의 두 번째 피연산자가 명령어의 부호가 확장될 하위 16비트로부터 주어진다.	ALU의 두 번째 피연산자가 명령어의 하위 15-10비트로부터 주어진다.
MemRead	없음	읽기 주소에 있는 데이터 메모리 내용이 읽기 데이터 출력으로 나온다.	
MemWrite	없음	쓰기 주소에 있는 데이터 메모리 내용이 쓰기 데이터로 바뀐다.	
MemtoReg	write Data로 보낼 입력을 ALU에서 전달받는다.	write Data로 보낼 입력을 데이터 메모리에서 전달받는다.	write Data로 보낼 입력을 PC+8을 계산하는 가산기에서 전달받는다.
RegWrite	없음	write Register 입력에 해당하는 레지스터에 write Data로 쓰기가 행해진다.	
Branch	PC+4를 계산하는 가산기의 출력이 PC에 전달된다.	PC+4+branchAddress가 PC에 전달된다.	
Jump	PC+4를 계산하는 가산기의 출력이 PC에 전달된다.	jumpAddress가 PC에 전달된다.	

<표 2.4> 파이프라인 데이터패스의 멀티플렉서, 데이터 메모리, 그리고 레지스터 파일 쓰기를 통제하는 제어 신호와 그 참, 거짓 효과. 제어부(Control Unit)는 명령어의 상위 6비트인 opcode를 분석하여 제어 신호를 결정하고, 전방전달 유닛(Forwarding Unit)은 상황에 맞는 RegWrite, writeReg, rs 등을 분석하여 제어 신호를 결정한다.

4) 데이터 종속성 해결

가) nop 명령어(nop Operation)

종속성을 해결하기 전, 우리가 먼저 살펴봐야할 것은 nop 명령어이다. <그림 2.12>는 fib.bin 파일의 일부를 가져온 것으로 nop 명령어가 load word, branch equal, branch not equal, jump, jump and link, jump register 명령어 다음에 자동으로 삽입되어 있다는 것을 알 수 있다. 아무 일도 하지 않는 nop 명령어를 고려하여 파이프라인을 설계하면 몇 가지 큰 이득을 볼 수 있다. 먼저, 데이터 종속성 중 load word 명령어는 전방전달을 도입하더라도 한 클럭 사이클을 지연해야 한다. lw 명령어의 4단계(Memory Access)이후 불러와야 할 데이터를 다음 명령어 3단계(Execution) ALU의 피연산자로 요구할 수 있기 때문이다. 그러나 nop 명령어가 lw 명령어 다음에 자동으로 삽입되어 있으면 클럭 사이클을 지연하지 않아도 된다. 다음으로 제어 종속성 중 beq, bne 명령어는 분기 예측에 실패하면 레치를 플러시(Flush)해야 한다. 분기 명령어가 분기를 안 한다고 예측했는데 3단계에서 확인한 결과 분기를 한 경우, 분기 명령어 다음에 nop 명령어가 삽입되어 있으므로 ID 레치를 제외하고 IF 레치만 플러시하면 된다. 마지막으로 jump 명령어는 2단계에서 점프 주소가 결정된다. 마찬가지로 jump 명령어 다음에 nop 명령어가 삽입되어 있으므로 이것을 고려하면 IF 레치를 플러시하지 않아도 된다.

```
00000040 <fib>:
  40:   27bdffd0        addiu   sp,sp,-48
  44:   afbf002c        sw      ra,44(sp)
  48:   afbe0028        sw      s8,40(sp)
  4c:   afb00024        sw      s0,36(sp)
  50:   03a0f021        move    s8,sp
  54:   afc40030        sw      a0,48(s8)
  58:   8fc20030        lw      v0,48(s8)
  5c:   00000000        nop
  60:   28420003        slti    v0,v0,3
  64:   10400004        beqz    v0,78 <fib+0x38>
  68:   00000000        nop
  6c:   24020001        li      v0,1
  70:   0800002e        j       b8 <fib+0x78>
  74:   00000000        nop
  78:   8fc20030        lw      v0,48(s8)
  7c:   00000000        nop
  (이하 생략)
```

```
00000000 <main>: fib
   0:   27bdffd8        addiu   sp,sp,-40
   4:   afbf0024        sw      ra,36(sp)
   8:   afbe0020        sw      s8,32(sp)
   c:   03a0f021        move    s8,sp
  10:   2402000a        li      v0,10
  14:   afc20018        sw      v0,24(s8)
  18:   8fc40018        lw      a0,24(s8)
  1c:   0c000000        jal     0 <main>
  20:   00000000        nop
  24:   afc2001c        sw      v0,28(s8)
  28:   03c0e821        move    sp,s8
  2c:   8fbf0024        lw      ra,36(sp)
  30:   8fbe0020        lw      s8,32(sp)
  34:   27bd0028        addiu   sp,sp,40
  38:   03e00008        jr      ra
  3c:   00000000        nop
```

<그림 2.12> lw, beq, bne, j, jal, jr 명령어 다음에 삽입된 nop 명령어. 자동으로 삽입된 nop 명령어를 이용하면 종속성을 더 쉽게 해결할 수 있다.

나) 전방전달 유닛(Forwarding unit)

파이프라인 종속성 중 데이터 종속성을 해결하는 방법으로 전방전달을 사용하겠다고 밝혔다. 이제 EX 종속성, MEM 종속성, 그리고 WB 종속성이 생기는 조건을 더 구체적으로 파악하여 종속성 해결의 실마리를 찾아보자.

첫 번째는 EX 종속성이다. EX 종속성은 한 단계 앞선 명령어가 Write Back 단계에서 레지스터 파일에 쓸 데이터를 현재 명령어가 ALU의 첫 번째 또는 두 번째 피연산자로 사용하려고 할 때 발생한다. 이것을 해결하기 위해 '전방전달 유닛'이라는 새로운 하드웨어를 추가해 EX 종속성이 발생했는지 확인하고, 종속성이 발생했다면 한 단계 앞선 명령어가 3단계(Execution)에서 내놓은 결과를 미리 가져와 ALU의 피연산자로 넣어줄 것이다. EX 종속성이 발생하는 조건은 다음과 같다. 현재 명령어의 피연산자로 사용될 레지스터가 앞선 명령어의 쓰기 레지스터와 같은지 확인한다. 이와 더불어서 어떤 명령어는 레지스터에 쓰기를 하지 않기 때문에 앞선 명령어의 RegWrite 제어 신호가 1로 인가되었는지 확인한다. 한편, MIPS의 레지스터 $0은 항상 상수 0을 가지고 있어야 하므로 해당 레지스터가 $0이면 그 값은 바꾸면 안 된다.

1. EX 종속성

if((EXMEM.RegWrite == 1)
and (EXMEM.writeReg != 0)
and (EXMEM.writeReg == IDEX.rs)) then ForwardA control signal is 0b01

if((EXMEM.RegWrite == 1)
and (EXMEM.writeReg != 0)
and (EXMEM.writeReg == IDEX.rt)) then ForwardB control signal is 0b01

두 번째는 MEM 종속성이다. MEM 종속성은 두 단계 앞선 명령어가 Write Back 단계에서 레지스터 파일에 쓸 데이터를 현재 명령어가 ALU의 첫 번째 또는 두 번째 피연산자로 사용하려고 할 때 발생한다. 이것을 해결하기 위해 추가한 전방전달 유닛에서 MEM 종속성이 발생했는지 확인하고, 종속성이 발생했다면 두 단계 앞선 명령어가 4단계(Memory Access)에서 내놓은 결과를 미리 가져와 ALU의 피연산자로 넣어줄 것이다. MEM 종속성이 발생하는 조건은 다음과 같다. 현재 명령어의 피연산자로 사용될 레지스터가 두 단계 앞선 명령어의 쓰기 레지스터와 같은지 확인한다. 어떤 명령어는 레지스터에 쓰기를 하지 않기 때문에 두 단계 앞선 명령어의 RegWrite 제어 신호가 1로 인가되었는지 확인한다. 한편, MIPS의 레지스터 $0은 항상 상수 0을 가지고 있어야 하므로 해당 레지스터가 $0이면 그 값은 바꾸면 안 된다.

2. MEM 종속성
if((MEMWB.RegWrite == 1)
and (MEMWB.writeReg != 0)
and not
and (MEMWB.writeReg == *IDEX.rs*)) then ForwardA control signal is 0b10

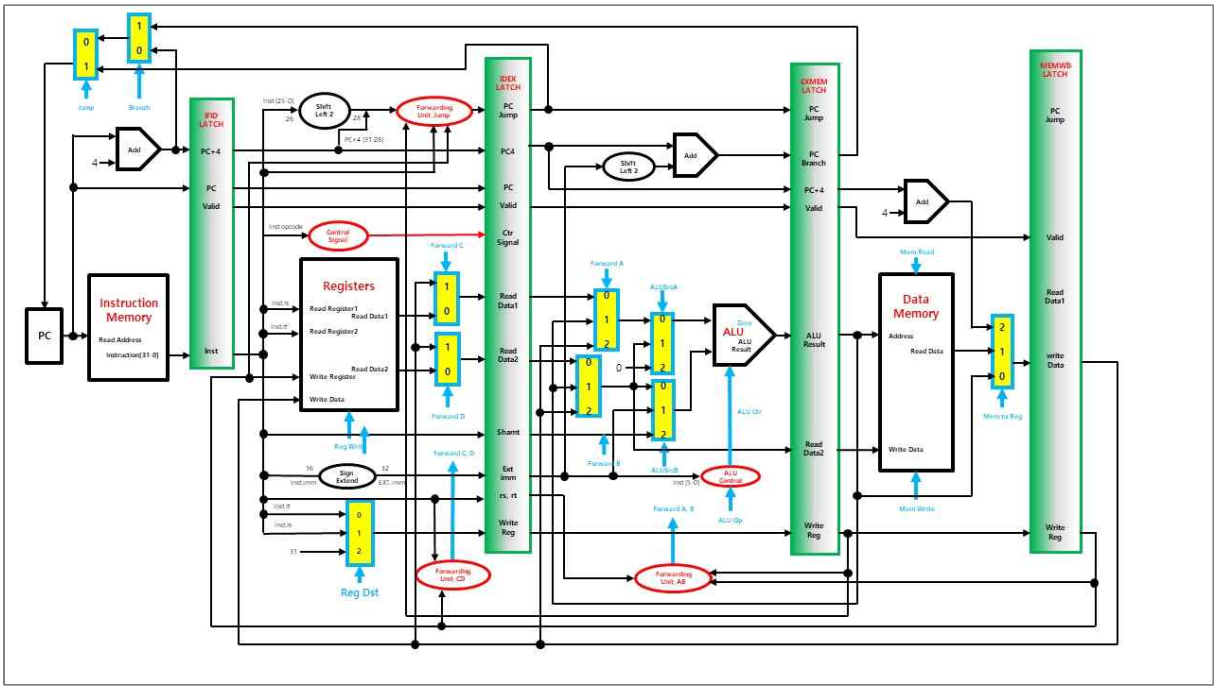
if((MEMWB.RegWrite == 1)
and (MEMWB.writeReg != 0)
and (MEMWB.writeReg == *IDEX.rt*)) then ForwardB control signal is 0b10

세 번째는 WB 종속성이다. WB 종속성은 세 단계 앞선 명령어가 Write Back 단계에서 레지스터 파일에 쓸 데이터를 다른 명령어가 ALU의 첫 번째 또는 두 번째 피연산자로 사용하려고 할 때 발생한다. WB 종속성은 EX, MEM 종속성과 조금 다른데 전방전달 유닛의 위치가 3단계(Execution)가 아닌 2단계(Decode)에 있다. 종속성 처리하기 위해 전방전달 유닛을 한 단계 앞당겨왔으므로 WB 종속성이 발생하는 조건은 MEM 종속성의 조건과 거의 비슷하다.

3. WB 종속성
if((MEMWB.RegWrite == 1)
and (MEMWB.writeReg != 0)
and (MEMWB.writeReg == *Inst.rs*)) then ForwardC control signal is 0b01

if((MEMWB.RegWrite == 1)
and (MEMWB.writeReg != 0)
and (MEMWB.writeReg == *Inst.rt*)) then ForwardD control signal is 0b01

한편, 데이터 종속성이 연속적으로 발생할 경우, 우리는 어느 데이터를 먼저 전방전달할지 우선순위를 고려해야한다. 현재 명령어가 피연산자로 사용해야할 데이터는 오래 전에 저장된 데이터가 아닌, 가장 최근에 저장된 데이터라는 것은 누구라도 직관적으로 알고 있을 것이다. 그렇다면 현재 명령어와 가장 가까운 명령어의 종속성은 무엇인가? EX 종속성이다. 즉, 가장 멀리 떨어진 명령어의 WB 종속성부터 먼저 확인하고, 다음으로 가까운 명령어의 종속성인 MEM 종속성을 확인하고, 마지막으로 가장 가까운 명령어의 EX 종속성을 확인해야 한다. <그림 2.13>에는 WB 종속성을 가장 먼저 확인하도록 전방전달 유닛이 2단계에 위치한 것을 확인할 수 있다.



<그림 2.13> 데이터 종속성을 해결하는 전방전달 유닛이 추가된 파이프라인 데이터패스. 시계방향 순으로 EX 종속성과 MEM 종속성을 해결하는 전방전달 유닛, WB 종속성을 해결하는 전방전달 유닛, 그리고 Jump 종속성을 해결하는 전방전달 유닛이 추가되어 있다.

5) 제어 종속성 해결

파이프라인 종속성 중 제어 종속성을 해결하는 방법으로 네 가지 분기 예측을 사용하겠다고 앞서 밝혔다. 이제 분기 예측 종류별로 예측을 실행하는 방법과 분기 예측이 실패한 경우, 이 예외를 어떻게 처리해야 하는지 알아보도록 하자.

가) 정적 분기 예측: Always not taken

첫 번째, Always not taken은 항상 분기를 하지 않는다고 예측하고 명령어들을 순서대로 실행한다. 만약 분기를 한다면 인출되고 해독되었던 명령어들은 버리고, PC를 분기 주소(Branch Target)로 이동하여 실행을 계속한다.

1. Static branch prediction; Always not taken
PC always becomes PC+4

if (Branch==1)
then IFID Latch is flushed, PC becomes Branch Address

나) 동적 분기 예측: One level branch prediction

두 번째, One level branch prediction은 1단계(Instruction Fetch)에서 분기를 예측하고 3단계(Execute)에서 분기 예측을 확인한다. 먼저 1단계에서는 현재 명령어의 PC가 BTB에 저장되어 있는지 확인한다. PC가 BTB에 저장되어 있으면 PC와 짝지어진 분기 주소(Branch Target)를 불러온다. 다음으로 PHT에서 PC를 인덱스로 하는 패턴을 불러온다. PC가 BTB에 저장되어 있고 패턴이 2 또는 3이면 분기를 한다고 예측하여, 다음 명령어의 PC에 불러온 분기 주소(Branch Target)를 저장한다. 다음 3단계에서는 예측한 분기를 확인한다. 분기를 하지 않는다고 예측했는데 만약 분기를 했다면, 1단계의 데이터를 저장한 IFID 레치를 플러시하고 PC를 분기 주소(Branch Target)로 바꾼다. 분기를 한다고 예측했는데 분기를 하지 않았다면 1, 2단계의 데이터를 저장한 IFID, IDEX 레치를 플러시하고 PC를 PC+4로 바꾼다.

2. One level branch prediction

```
<Check BTB>
if (PC is equal to BTB's saved PC)
   hit becomes 1, load Branch Target
else
   hit becomes 0

<Check PHT>
load pattern from PHT(PC)
```

```
<Branch Prediction>
if (hit is 1 and pattern is 2 or 3)
   PC is predicted as Branch Target
else
   PC is predicted as PC + 4
```

다) 동적 분기 예측: Two level global branch prediction

세 번째, Two level global branch prediction 역시 1단계(Instruction Fetch)에서 분기를 예측하고 3단계(Execute)에서 분기 예측을 확인한다. 먼저 1단계에서는 현재 명령어의 PC가 BTB에 저장되어 있는지 확인한다. PC가 BTB에 저장되어 있으면 PC와 짝지어진 분기 주소(Branch Target)를 불러온다. 다음으로 PHT에서 PC | GHR를 인덱스로 하는 패턴을 불러온다. PC가 BTB에 저장되어 있고 패턴이 2 또는 3이면 분기를 한다고 예측하여, 다음 명령어의 PC에 불러온 분기 주소(Branch Target)를 저장한다. 3단계 과정은 One level branch prediction과 동일하다.

3. Two level global branch prediction

```
<Check BTB>
if (PC is equal to BTB's saved PC)
   hit becomes 1, load Branch Target
else
   hit becomes 0

<Check PHT>
load pattern from PHT (((PC & 0x3) << 4)
| (GHR & 0xf))
```

```
<Branch Prediction>
if (hit is 1 and pattern is 2 or 3)
   PC is predicted as Branch Target
else
   PC is predicted as PC + 4
```

라) 동적 분기 예측: Two level gshare branch prediction

네 번째, Two level global branch prediction 역시 1단계(Instruction Fetch)에서 분기를 예측하고 3단계(Execute)에서 분기 예측을 확인한다. 먼저 1단계에서는 현재 명령어의 PC가 BTB에 저장되어 있는지 확인한다. PC가 BTB에 저장되어 있으면 PC와 짝지어진 분기 주소(Branch Target)를 불러온다. 다음으로 PHT에서 PC ^ GHR를 인덱스로 하는 패턴을 불러온다. PC가 BTB에 저장되어 있고 패턴이 2 또는 3이면 분기를 한다고 예측하여, 다음 명령어의 PC에 불러온 분기 주소(Branch Target)를 저장한다. 3단계 과정은 One level branch prediction과 동일하다.

```
4. Two level gshare branch prediction
<Check BTB>
if (PC is equal to BTB's saved PC)
   hit becomes 1, load Branch Target
else
   hit becomes 0

<Check PHT>
load pattern from PHT ((PC & 0x3f) ^
GHR)

<Branch Prediction>
if (hit is 1 and pattern is 2 or 3)
   PC is predicted as Branch Target
else
   PC is predicted as PC + 4
```

3. Pipelined MIPS Micro Architecture

제 3장, Pipelined MIPS Micro Architecture에서는 제 2장에서 학습한 내용을 바탕으로 구현한 Pipelined MIPS 프로그램을 분석한다.

가. 개발 환경

프로젝트3 Pipelined MIPS Micro Architecture 프로그램은 Windows 10 운영 체제에서 Visual Studio 2019를 통해 C언어로 작성되었다. simple.bin, simple2.bin, simple3.bin, simple4.bin, fib.bin, gcd.bin, input4.bin 파일은 PipelinedMain.c와 같은 폴더 안에 있어야 한다.

나. 프로그램 분석

이 프로그램은 사용자가 프로그램을 더 쉽고 직관적으로 이해할 수 있도록 PipelinedHeader.h, PipelinedMain.c, 그리고 PipelinedFunction.c 세 부분으로 나누어 작성되었다. 먼저, PipelinedHeader.h는 프로그램에서 사용한 헤더 파일, 구조체, 함수, 그리고 전역 변수가 선언되어 있으므로 이것을 살펴보며 프로그램의 간략한 틀을 파악할 수 있다. 다음으로 PipelinedFuction.c는 파이프라인 데이터패스를 바탕으로 하는 파이프라인 단계와 전방전달, 그리고 분기 예측을 위한 함수가 작성되어 있으므로, 명령어가 파이프라인에서 처리되는 순서대로 천천히 따라가며 전방전달과 분기 예측이 어떻게 실현되는지 살펴보자. 마지막으로, PipelinedMain.c는 우리의 시야를 넓혀 메모리에 파일이 로드되는 처음과 명령어가 병렬로 처리되어 결과가 도출되는 끝을 보며 파이프라인이 어떻게 완성되는지 알아보자. 프로그램 분석은 파이프라인 구현과 종속성 해결을 중심으로 서술하려고 한다. 따라서 싱글 사이클에서 이미 다루었거나, 이해하기 어렵지 않은 일부 코드는 분석에서 제외하였다.

1) PipelinedHeader.h

가) 전역 변수(Global Variables)

```
// Global Variables
extern int INSTMEMORY[0x00400000];
extern int DATAMEMORY[0x00400000]; ……①
extern int Register[32]; ……②

unsigned int PC;
unsigned int printOption;
unsigned int predictOption;
unsigned int countInst;…… ③
unsigned int countNop;
unsigned int countCycle;
unsigned int countLW, countSW;
unsigned int countPrediction;
unsigned int countJump;
float countCorrectPrediction;
float countWrongPrediction;
unsigned int countNotTakenBranch;
unsigned int countTakenBranch;
float BranchAccuracy;

int eof; // End of file …… ④

// Global Branch History Register…… ⑤
int GHR;
// Branch Target Buffer, 256KB
extern int BTB[0x8000][2];
// Pattern History Table, 128KB
extern int PHT[0x8000];
int hit, taken;
int BTBindex;
int BranchTarget;
```

① 메모리를 명령어 메모리 배열과 데이터 메모리 배열로 각각 구분하여 INSTMEMORY[0x00400000], DATAMEMORY[0x00400000]를 선언한다. 메모리의 용량은 각각 16MB에 해당한다.

② MIPS ISA의 32가지 범용 레지스터를 위한 배열 Register[32]를 선언한다.

③ 프로젝트3 추가구현 목록에 해당하는 명령어 수, nop 명령어 수, 클럭 사이클 수, 데이터 메모리 접근 횟수, 분기 예측이 맞은 횟수, 분기 예측이 틀린 횟수, 점프한 횟수, 분기한 횟수, 분기하지 않은 횟수, 분기 정확도를 변수로 선언한다.

④ 바이너리 파일의 마지막을 감지할 eof 변수를 선언한다.

⑤ Global History Register에 해당하는 GHR, 256KB Branch Target Buffer에 해당하는 BTB, 128KB Pattern History Table에 해당하는 PHT 등 동적 분기 예측에 사용할 변수와 배열을 선언한다.

나) 구조체; 레치와 전방전달 유닛(Structures for Latch, Forwarding Unit)

```
(헤더 파일 선언 생략)
(일부 구조체 선언 생략)

typedef struct {
  // others
  int PC, PC4;
  int inst;
  int valid;

  // branch prediction
  int PHTindex;
  int BranchPrediction;
} IFID_LATCH; ……①

typedef struct {
  // control signals
  int RegDst;
  int ALUOp;
  int ALUSrcA, ALUSrcB;
  int Jump;
  int BranchEqual, BranchNotEqual;
  int MemRead, MemWrite;
  int RegWrite;
  int MemtoReg;
  int writeReg;

  // others
  int PC, PC4, PC_Jump;
  int readData1, readData2;
  int extimm;
  int shamt;
```

```
  int valid;

  // forwarding unit
  int rs_Forward, rt_Forward;

  // branch prediction
  int PHTindex;
  int BranchPrediction;
} IDEX_LATCH; ····· ②

typedef struct {
  // control signals
  int Jump, Branch;
  int MemRead, MemWrite;
  int RegWrite;
  int MemtoReg;

  // others
  int PC, PC4, PC_Jump, PC_Branch;
  int ALUresult;
  int readData2;
  int writeReg;
  int valid;
} EXMEM_LATCH; ····· ③
```

```
typedef struct {
  // control signals
  int RegWrite;

  // others
  int PC_Jump;
  int writeData;
  int writeReg;
  int valid;
} MEMWB_LATCH; ····· ④

typedef struct {
  // forwarding unit signals
  unsigned int SelectA, SelectB;
  unsigned int SelectC, SelectD;
} Forwarding; ····· ⑤

(함수 선언 생략)
```

① 구조체 IFID LATCH는 1단계를 마친 명령어의 데이터를 저장하는 IFID 레치로, 다음 명령어의 주소를 저장할 PC, 명령어 워드를 저장할 inst, Pattern History Table의 인덱스로 사용할 PHTindex, 그리고 분기 예측 데이터를 저장할 BranchPrediction 변수가 선언되어 있다.

② 구조체 IDEX LATCH는 2단계를 마친 명령어의 데이터를 저장하는 IDEX 레치로, 명령어를 해독하여 얻은 제어 신호, 다음 명령어의 주소를 저장할 PC들, ALU의 피연산자로 사용할 readData1, readDate2, 부호 확장된 immediate, shamt, 전방전달 레지스터와 번호를 비교할 rs_Forward, rt_Forward, 그리고 전달받은 PHTindex와 BranchPrediction을 저장할 변수가 선언되어있다.

③ 구조체 EXMEM LATCH는 3단계를 마친 명령어의 데이터를 저장하는 EXMEM 레치로, Jump와 Branch를 포함하는 제어 신호, 점프할 주소와 분기할 주소를 저장할 PC_Jump와 PC_Branch, ALU 연산 결과를 저장할 ALUresult, 그리고 명령어 메모리에 저장할 readData2 변수가 선언되어 있다.

④ 구조체 MEMWB LATCH는 4단계를 마친 명령어의 데이터를 저장하는 MEMWB 레치로, 점프할 주소를 저장하는 PC_Jump, 레지스터 파일 쓰기를 통제하는 제어 신호 regWrite, 쓰기 레지스터 번호 writeReg, 그리고 레지스터에 쓸 결과 writeData가 선언되어 있다.

⑤ 구조체 Forwarding은 전방전달 제어 신호를 저장하는 구조체로 EX 종속성과 MEM 종속성을 통제하는 제어 신호 ForwardA, ForwardB와 WB 종속성을 통제하는 제어 신호 ForwardC, ForwardD가 선언되어 있다.

2) PipelinedFunction.c

가) Fetch 함수

```
void Fetch(void) {
  if (printOption)
    printf("\n\t CYCLE No.%d\n", countCycle);

  // count the number of clock cycle
  countCycle++;
  countInst++;

  switch (predictOption) {····· ①
  case 1:
    MUX_Branch();
    MUX_Jump();
    break;

  case 2:
  case 3:
  case 4:
    MUX_Jump();
    break;

  default:
    break;
  }

  // fetch instruction from instruction memory
  InstMem.inst = INSTMEMORY[PC / 4];····· ②

  // set valid
  if (PC == 0xffffffff) {·····③
    // for the last instruction
    IFID_IN.valid = 0;
    if (printOption)
      printf("\n\t [I F]\tvalid\t: 0");
    return;
  }
  else if (InstMem.inst == 0x00000000) {
  // for nop instruction····· ④
    IFID_IN.valid = 0;
    countNop++;
    countInst--;
    PC = PC + 4;
    if (printOption)
      printf("\n\t [I F]\tnop\t: 0x%08x",
      InstMem.inst);
    return;
  }
  else
    IFID_IN.valid = 1;

  // store to IFID_IN LATCH····· ⑤
  IFID_IN.PC = PC;
  IFID_IN.PC4 = PC + 4;
  IFID_IN.inst = InstMem.inst;

  if (printOption) {
    printf("\n\t [I F]\tpc\t: 0x%08x", PC);
    printf("\n\t\tinst\t: 0x%08x",
    InstMem.inst);
  }

  switch (predictOption) {····· ⑥
  case 1:
    PC = PC + 4;
    countPrediction++;
    break;

  case 2:
  case 3:
  case 4:
    DynamicBranchPrediction();
    break;
  }
  return;
}
```

① 사용자가 분기 예측을 1번 Always not taken으로 할 경우, MUX_Branch 함수와 MUX_Jump 함수를 호출하여 현재 명령어의 주소를 결정한다. 그 외 2번 One level branch prediction, 3번 Two level global branch prediction, 4번 two level gshare preidiction으로 할 경우, MUX_Jump 함수를 호출하여 현재 명령어의 주소를 결정한다.

② 명령어 메모리로부터 명령어를 인출한다.

③ 프로그램 카운터가 –1을 가리킬 경우, 모든 명령어를 실행했다는 의미이므로 레치의 valid를 0으로 설정하고 함수에서 빠져나온다.

④ 프로그램 카운터가 0을 가리킬 경우, nop 명령어에 해당하므로 레치의 valid를 0으로 설정하고 PC를 4만큼 증가시킨 다음, 함수에서 빠져나온다.

⑤ 파이프라인 2단계에서 사용할 변수들을 IFID IN 레치에 저장한다.

⑥ 사용자의 분기 예측 옵션에 따라서 다음 명령어의 주소를 결정한다. 1번의 경우, 다음 명령어의 주소로 PC+4, 그 외의 경우 DynamicBranchPrediction 함수에서 다음 명령어의 주소를 결정한다.

(1) DynamicBranchPrediction 함수

```
void DynamicBranchPrediction(void) {
  hit = checkBTB(PC);·····①

  switch (predictOption) {
  // one Lv. branch prediction
  case 2:·····②
    taken = checkPHT(PC);
    IFID_IN.PHTindex = PC;
    break;

  // two Lv. global branch prediction
  case 3:·····③
    taken = checkPHT(((PC & 0x3) << 4) |
    (GHR & 0xf));
    IFID_IN.PHTindex = ((PC & 0x3) << 4) |
    (GHR & 0xf);
    break;

  // two Lv. gshare branch prediction
  case 4:·····④
    taken = checkPHT((PC & 0x3f) ^ GHR);
    IFID_IN.PHTindex = (PC & 0x3f) ^ GHR;
    break;

  default:
    break;
  }
  switch (hit) {
  case 0:·····⑤
    IFID_IN.BranchPrediction = 0;
    PC = PC + 4;
    countPrediction++;
    break;

  case 1:·····⑥
    if (taken == 0b00 || taken == 0b01) {
      IFID_IN.BranchPrediction = 0;
      PC = PC + 4;
      countPrediction++;
    }
    else if(taken == 0b10 || taken == 0b11) {
      IFID_IN.BranchPrediction = 1;
      PC = BranchTarget;
      countPrediction++;
    }
    else;
    break;

  default:
    break;
  }
  return;
}
```

① checkBTB 함수의 인자로 PC를 전달하고 반환된 0 또는 1을 hit에 저장한다.

② One level branch prediction은 checkPHT 함수의 인자로 PC를 전달하고, 반환된 0부터 3 사이의 패턴을 taken에 저장한다.

③ Two level branch prediction은 checkPHT 함수의 인자로 PC의 하위 2비트와 4비트 GHR를 OR 연산한 6비트를 전달하고, 0부터 3 사이의 반환된 패턴을 taken에 저장한다.

④ Two level gshare prediction은 checkPHT 함수의 인자로 PC의 하위 6비트와 6비트 GHR를 XOR 연산한 6비트를 전달하고, 0부터 3 사이의 반환된 패턴을 taken에 저장한다.

⑤ BTB 배열에 해당 PC의 주소가 저장되어 있지 않아 분기를 하지 않는다고 예측한다. 다음 명령어의 주소로 PC+4를 저장한다.

⑥ BTB 배열에 해당 PC의 주소가 저장되어있어 PHT 배열로부터 반환받은 taken에 따라서 분기를 결정한다. taken이 0(strongly not taken) 또는 1(weakly not taken)인 경우, 분기를 하지 않는다고 예측하고 다음 명령어의 주소로 PC+4를 저장한다.

⑦ taken이 2(weakly taken) 또는 3(strongly taken)인 경우, 분기를 한다고 예측하고 다음 명령어의 주소로 BranchTarget을 저장한다.

(2) checkBTB, checkPHT 함수

```
int checkBTB(int PC) {····· ①
  for (int i = 0; i <= BTBindex; i++) {
    if (BTB[i][0] == PC && PC != 0) {
    // 1st instruction's pc is 0x0
      BranchTarget = BTB[i][1];
      return 1;
    }
    else;
  }
  return 0;
}
```

```
int checkPHT(int index) {····· ②
  return PHT[index];
}
```

① 명령어의 주소를 인자로 전달받은 checkBTB 함수는 BTB 배열에서 해당 명령어의 주소와 같은 주소가 배열에 저장되어 있는지 for 반복문으로 확인한다. 단, 첫 번째 명령어는 pc가 0이므로 제외한다. 같은 주소가 있으면 분기할 주소를 BranchTarget에 저장하고 1을 반환한다. 같은 주소가 없으면 0을 반환한다.

② index를 인자로 전달받은 checkPHT 함수는 PHT 배열로부터 해당 index의 2비트 패턴을 반환한다.

나) Decode 함수

```
void Decode(void) {
  // set valid
  IDEX_IN.valid = IFID_OUT.valid;····· ①
  if (!IFID_OUT.valid) {
    if (printOption)
      printf("\n\t [I D]\tvalid\t: 0");
    return;
  }

  // decode instruction····· ②
  (명령어 해독 중략)

  // set control signals
  and extend immediate
  Control();····· ③
  SignExtend();····· ④

  // register file····· ⑤
  REG.readReg1 = INST.rs;
  REG.readReg2 = INST.rt;
  REG.readData1 = Register[REG.readReg1];
  REG.readData2 = Register[REG.readReg2];

  // store to IDEX_IN Latch
  IDEX_IN.RegDst = CTR.RegDst;
  (IDEX 레치 저장 중략)

  // Data Dependency No.3
  // forwarding from write back stage
  ForwardingCDUnit();····· ⑥
  MUX_ForwardC();
  MUX_ForwardD();

  // MUX_RegDst, JumpAddress;
  MUX_RegDst();····· ⑦
  JumpAddress();····· ⑧

  (프린트 옵션 생략)
  return;
}
```

① 2단계(Instruction Decode)의 valid를 확인한다. 이전 단계의 valid가 0이면 Decode 함수에서 빠져나온다.

② 명령어를 해독하여 opcode, rs, rt, rd, shamt, funct, imm, addr를 얻는다.

③ Control 함수는 opcode와 fucnt를 분석하여 명령어의 제어 신호를 결정한다.

④ SignExtend 함수는 immediate를 16비트에서 32비트로 확장한다.

⑤ 명령어에서 읽은 레지스터 번호를 레지스터 파일로 전달해 레지스터 파일에 저장된 데이터를 불러온다.

⑥ ForwardingCDUnit 함수는 WB 종속성 조건을 확인하고 전방전달 제어 신호를 결정한다. 전방전달 제어 신호에 따라서 ForwardC 멀티플렉서 함수와 ForwardD 멀티플렉서 함수가 어떤 데이터를 반환할지 결정한다.

⑦ RegDst 멀티플렉서 함수는 RegDst 제어 신호에 따라서 목적지 레지스터로 rt, rd, 또는 31을 내보낸다.

(1) ForwardingCDUnit 함수

ForwardingCDUnit 함수는 2단계(Instruction Decode)에서 WB 종속성을 해결하기 위해 정의된 함수로 종속성 조건에 따라서 제어 신호 SelectC와 SelectD를 결정한다.

```
void ForwardingCDUnit(void) {
  // default SelectC, SelectD
  FORWARD.SelectC = 0b00;
  FORWARD.SelectD = 0b00;

  // check write back forwarding for SelectC
  if (MEMWB_OUT.RegWrite &&····· ①
    (MEMWB_OUT.writeReg != 0) &&
    (REG.readReg1 == MEMWB_OUT.writeReg))
      FORWARD.SelectC = 0b01;

  // check write back forwarding for SelectD
  if ((MEMWB_OUT.RegWrite &&····· ②
    (MEMWB_OUT.writeReg != 0) &&
    REG.readReg2 == MEMWB_OUT.writeReg))
      FORWARD.SelectD = 0b01;

  return;
}
```

① 세 단계 앞선 명령어의 제어 신호, 쓰기 레지스터와 현재 명령어의 첫 번째 읽기 레지스터를 비교한다. 세 가지 조건 '레지스터 쓰기를 한다.', '레지스터 번호가 0이 아니다.', '레지스터 번호가 첫 번째 읽기 레지스터와 같다.'를 모두 만족하면 SelectC에 0b01을 저장한다.

② 세 단계 앞선 명령어의 제어 신호, 쓰기 레지스터와 현재 명령어의 두 번째 읽기 레지스터를 비교한다. 세 가지 조건 '레지스터 쓰기를 한다.', '레지스터 번호가 0이 아니다.', '레지스터 번호가 두 번째 읽기 레지스터와 같다.'를 모두 만족하면 SelectD에 0b01을 저장한다.

(2) MUX_ForwardC, MUX_ForwardD 함수

```
void MUX_ForwardC(void) {
  switch (FORWARD.SelectC) {····· ①
  case 0:
    IDEX_IN.readData1 = REG.readData1;
    break;

  case 1:// forwarding from write back stage
    IDEX_IN.readData1 =
    MEMWB_OUT.writeData;
    break;

  default:
    break;
  }
  return;
}
```

```
void MUX_ForwardD(void) {
  switch (FORWARD.SelectD) {····· ②
  case 0:
    IDEX_IN.readData2 = REG.readData2;
    break;

  case 1:// forwarding from write back stage
    IDEX_IN.readData2 =
    MEMWB_OUT.writeData;
    break;

  default:
    break;
  }
  return;
}
```

① MUX_ForwardC 함수는 SelectC 제어 신호가 0인 경우, 전방전달을 하지 않고 레지스터에서 읽은 데이터 readData1을 IDEX IN 레치에 저장한다. 제어 신호가 1인 경우, MEMWB OUT 레치의 쓰기 데이터를 전달받아 IDEX IN 레치에 저장한다.

② MUX_ForwardD 함수는 SelectD 제어 신호가 0인 경우, 전방전달을 하지 않고 레지스터에서 읽은

데이터 readData2을 IDEX IN 레치에 저장한다. 제어 신호가 1인 경우, MEMWB OUT 레치의 쓰기 데이터를 전달받아 IDEX IN 레치에 저장한다.

(3) JumpAddress 함수

```
void JumpAddress(void) {// consider nop instruction after j, jal, jr, jalr
  EXT.jumpAddress = (IFID_OUT.PC4 & 0xf0000000) | (INST.addr << 2);····· ①

  if ((INST.opcode == 0x0 && INST.funct == 0x8) ||····· ②
    (INST.opcode == 0x0 && INST.funct == 0x9)) {// jr, jalr

    if (INST.rs && (INST.rs == EXMEM_OUT.writeReg))// check 1st forwarding
      IDEX_IN.PC_Jump = EXMEM_OUT.ALUresult;

    else if (INST.rs && (INST.rs == MEMWB_OUT.writeReg))// check 2nd forwarding
      IDEX_IN.PC_Jump = MEMWB_OUT.writeData;

    else
      IDEX_IN.PC_Jump = Register[REG.readReg1];
  }
  else
    IDEX_IN.PC_Jump = EXT.jumpAddress;

  return;
}
```

① 명령어의 address 필드와 IFID OUT 레치의 PC를 이용하여 점프할 주소를 계산한다.

② jump register와 jump and link register는 점프할 주소로 레지스터 값을 저장하기 때문에 데이터 종속성이 발생할 수 있다. 레지스터 번호가 0이 아니고, EXMEM OUT 레치의 쓰기 레지스터와 같으면 EX 종속성에 해당하므로 점프할 주소로 ALUresult를 저장한다. 레지스터 번호가 0이 아니고, MEMWB OUT 레치의 쓰기 레지스터와 같으면 MEM 종속성에 해당하므로 점프할 주소로 writeData를 저장한다.

다) Execute 함수

```
void Execute(void) {
  // set valid
  EXMEM_IN.valid = IDEX_OUT.valid;·····①
  if (!IDEX_OUT.valid) {
    if (printOption)
      printf("\n\t [E X]\tvalid\t: 0");
    return;
  }

  // Data Dependency No.1, No.2; Forwarding
  ForwardingABUnit();····· ②
  MUX_ForwardA();
  MUX_ForwardB();

  // store readData2 for store word
  EXMEM_IN.readData2 = ALU.operand2;

  // read IDEX_OUT
  MUX_ALUSrcA();····· ③
  MUX_ALUSrcB();
  ALUControl();····· ④
  switch (CTR.ALUCtr) {·····⑤
  (ALU 연산 생략)
  }

  // set branch address
  BranchAddress();·····⑥

  // store to EXMEM_IN LATCH
  (EXEME_IN 레치 저장 생략)
  (프린트 옵션 생략)
  return;
}
```

① 3단계(Execution)의 valid를 확인한다. 이전 단계의 valid가 0이면 Execute 함수에서 빠져나온다.
② ForwardingABUnit 함수는 EX, MEM 종속성 조건을 확인하고 전방전달 제어 신호를 결정한다. 전방전달 제어 신호에 따라서 ForwardA 멀티플렉서 함수와 ForwardB 멀티플렉서 함수가 어떤 데이터를 반환할지 결정한다.
③ ALUSrcA 멀티플렉서 함수와 ALUSrcB 멀티플렉서 함수는 명령어의 R, I형식에 따라서 ALU의 피연산자로 사용할 데이터를 결정한다.
④ ALUControl 함수는 ALUop과 funct를 분석하여 제어 신호 ALUCtr를 결정한다.
⑤ 제어 신호 ALUCtr에 따라서 ALU 연산을 수행한다.
⑥ BranchAddress 함수는 분기 예측의 성공 여부에 따라 다음 명령어의 주소를 바꾸거나, 레치를 플러시한다.

(1) ForwardingABUnit 함수

```
void ForwardingABUnit(void) {
  // default SelectA, SelectB
  FORWARD.SelectA = 0b00;
  FORWARD.SelectB = 0b00;

  // check execution forwarding for SelectA ($0 제외)·····①
  if (EXMEM_OUT.RegWrite && (EXMEM_OUT.writeReg != 0) &&
    (EXMEM_OUT.writeReg == IDEX_OUT.rs_Forward))
      FORWARD.SelectA = 0b01;

  // check memory access forwarding for SelectA ($0 제외, EX 전방전달 제외)·····②
  else if(MEMWB_OUT.RegWrite && (MEMWB_OUT.writeReg != 0) &&
        (MEMWB_OUT.writeReg == IDEX_OUT.rs_Forward) &&
        (EXMEM_OUT.writeReg != IDEX_OUT.rs_Forward))
          FORWARD.SelectA = 0b10;
  else;

  // check execution forwarding for SelectB ($0 제외)·····③
  if (EXMEM_OUT.RegWrite && (EXMEM_OUT.writeReg != 0) &&
    (EXMEM_OUT.writeReg == IDEX_OUT.rt_Forward))
      FORWARD.SelectB = 0b01;

  // check memory access forwarding for SelectB ($0 제외, EX 전방전달 제외)·····④
  else if(MEMWB_OUT.RegWrite && (MEMWB_OUT.writeReg != 0) &&
        (MEMWB_OUT.writeReg == IDEX_OUT.rt_Forward) &&
        (EXMEM_OUT.writeReg != IDEX_OUT.rt_Forward))
          FORWARD.SelectB = 0b10;
  else;
  return;
}
```

① 한 단계 앞선 명령어의 제어 신호, 쓰기 레지스터와 현재 명령어의 첫 번째 피연산자 레지스터(rs)를 비교한다. 세 가지 조건 '레지스터 쓰기를 한다.', '레지스터 번호가 0이 아니다.', '레지스터 번호가 첫 번째 피연산자 레지스터와 같다.'를 모두 만족하면 SelectA에 0b01을 저장한다.
② 두 단계 앞선 명령어의 제어 신호, 쓰기 레지스터와 현재 명령어의 두 번째 피연산자 레지스터(rt)를 비교한다. 네 가지 조건 '레지스터 쓰기를 한다.', '레지스터 번호가 0이 아니다.', '레지스터 번호가 두 번째 피연산자 레지스터와 같다.', 그리고 'EX 종속성이 발생하지 않았다.'를 모두 만족하면 SelectA에 0b10을 저장한다.
③ , ④ 마찬가지로 두 번째 피연산자 레지스터(rt)와 비교하여 SelectB에 0b01, 0b10을 저장한다.

(2) MUX_ForwardA, MUX_ForwardB 함수

```
void MUX_ForwardA(void) {
  switch (FORWARD.SelectA) {····· ①
  case 0:
    ALU.operand1 = IDEX_OUT.readData1;
    break;

  case 1:// forwarding from 3rd stage
    ALU.operand1 = EXMEM_OUT.ALUresult;
    break;

  case 2:// forwarding from 4th stage
    ALU.operand1 = MEMWB_OUT.writeData;
    break;

  default:
    break;
  }
  return;
}
```

```
void MUX_ForwardB(void) {
  switch (FORWARD.SelectB) {····· ②
  case 0:
    ALU.operand2 = IDEX_OUT.readData2;
    break;

  case 1:// forwarding from 3rd stage
    ALU.operand2 = EXMEM_OUT.ALUresult;
    break;

  case 2:// forwarding from 4th stage
    ALU.operand2 = MEMWB_OUT.writeData;
    break;

  default:
    break;
  }
  return;
}
```

① ALU의 첫 번째 피연산자로 사용할 전방전달 데이터를 결정한다. SelectA가 0인 경우, 레지스터에서 읽은 readData1를 사용하고 SelectA가 1인 경우, EXMEM OUT 레치로부터 ALUresult를 전방전달 받는다. SelectA가 2인 경우, MEMWB OUT 레치로부터 writeData를 전방전달 받는다.

② ALU의 두 번째 피연산자로 사용할 전방전달 데이터를 결정한다. SelectB가 0인 경우, 레시스터에서 읽은 readData2를 사용하고 SelectB가 1인 경우, EXMEM OUT 레치로부터 ALUresult를 전방전달 받는다. SelectB가 2인 경우, MEMWB OUT 레치로부터 writeData를 전방전달 받는다.

(3) BranchAddress 함수

```
void BranchAddress(void) {
  EXT.branchAddress = ····· ①
    IDEX_OUT.PC4 + (IDEX_OUT.extimm << 2);

  EXMEM_IN.Branch =
    ((IDEX_OUT.BranchEqual == 1) &&
    (ALU.zero == true)) ||
    ((IDEX_OUT.BranchNotEqual == 1) &&
    (!ALU.zero == true));

  switch (predictOption) {
  case 1: // 1. Always not taken, or flush
    if (EXMEM_IN.Branch) {····· ②
      IFID_IN.valid = 0;// IF stage flush
      EXMEM_IN.PC_Branch =
        EXT.branchAddress;
      countInst--;
      countWrongPrediction++;
      countTakenBranch++;
    }
    else {····· ③
      if (IDEX_OUT.BranchEqual == 0 &&
          IDEX_OUT.BranchNotEqual == 0)
          countCorrectPrediction += 0;
      else {····· ④
          countCorrectPrediction++;
          countNotTakenBranch++;
      }
    }
    break;

  case 2:// One level
  case 3:// Two level global
  case 4:// Two level gshare

/** 경우 1: 분기 한다고 예측, 분기를 안 한 경우 **/
    if (IDEX_OUT.BranchPrediction == 1 &&
EXMEM_IN.Branch == 0) {····· ⑤
    // wrong prediction; flush IF, ID stage
      IFID_IN.valid = 0;
      IDEX_IN.valid = 0;
      PC = IDEX_OUT.PC + 4;
      countWrongPrediction++;
      countInst = countInst - 2;
```

```
        // 원인 1: 분기 명령어가 아니어서
        if (IDEX_OUT.BranchEqual == 0 &&
          IDEX_OUT.BranchNotEqual == 0)
          PC = PC;
        // 원인 2: 분기를 안 하는 분기 명령어여서
        else
          countNotTakenBranch++;
      }

/** 경우 2: 분기 안 한다고 예측, 분기를 한 경우 **/
      else if (IDEX_OUT.BranchPrediction == 0
            && EXMEM_IN.Branch == 1) {…… ⑥
      // wrong prediction; flush IF stage(nop)
        IFID_IN.valid = 0;
        PC = EXT.branchAddress;
        countWrongPrediction++;
        countTakenBranch++;
        countInst--;
      }
/** 경우 3: 분기 한다고 예측, 분기 한 경우 **/
      else if (IDEX_OUT.BranchPrediction == 1
            && EXMEM_IN.Branch == 1) {…… ⑦
        countCorrectPrediction++;
        countTakenBranch++;
      }

/** 경우 4: 분기 안 한다고 예측, 분기 안 한 경우 **/
      else if (IDEX_OUT.BranchPrediction == 0
            && EXMEM_IN.Branch == 0) {…… ⑧

        // 원인 1: 분기 명령어가 아니어서
        if (IDEX_OUT.BranchEqual == 0
          && IDEX_OUT.BranchNotEqual == 0)
          PC = PC;
        // 원인 2: 분기를 안 하는 분기 명령어여서
        else {
          countCorrectPrediction++;
          countNotTakenBranch++;
        }
      }
      else;

      updateBTB();
      updatePHT();
      break;

    default:
      break;
    }
    return;
}
```

① 명령어의 PC와 필드와 IFID OUT 레치의 부호 확장된 immediate를 이용하여 Branch Address를 계산한다.

② 분기를 하지 않는다고 예측했는데 분기를 한다면 예측에 실패한 것이므로 다음 명령어의 주소로 Branch Address를 저장한다. IFID IN 레치를 플러시한다. IDEX IN 레치는 nop 명령어이므로 플러시하지 않는다.

③ 분기를 하지 않는다고 예측했는데 실제로 분기를 하지 않았다면, 분기 명령어가 아니었거나 분기 명령어였지만 분기를 하지 않았다는 것을 의미한다.

④ 분기 명령어였지만 분기를 하지 않은 경우로, Always not taken이라는 예측이 맞았기 때문에 countCorrectPrediction 변수와 countNotTakenBranch 변수에 1을 더한다.

⑤ 동적 분기 예측에서 분기를 한다고 예측했지만 분기를 하지 않은 경우이다. 예측이 틀렸기 때문에 countWrongPrediction 변수에 1을 더한다. 또한 다음 명령어의 주소로 PC+4를 저장하고, IFID 레치와 IDEX 레치를 플러시한다. 그 원인이 분기를 하지 않는 분기 명령어라면 countNotTakenBranch 변수에 1을 더한다.

⑥ 동적 분기 예측에서 분기를 안 한다고 예측했지만 분기를 한 경우이다. 예측이 틀렸기 때문에 countWrongPrediction 변수와 countTakenBranch 변수에 1을 더한다. 또한 다음 명령어의 주소로 BranchAddress를 저장하고, IFID 레치를 플러시한다.

⑦ 동적 분기 예측에서 분기를 한다고 예측하고 분기를 한 경우이다. 예측이 맞았기 때문에 countCorrectPrediction 변수와 countTakenBranch 변수에 1을 더한다.

⑧ 동적 분기 예측에서 분기를 하지 않는다고 예측하고 분기를 하지 않은 경우이다. 예측이 맞았기 때문에 countCorrectPrediction 변수에 1을 더한다. 그 원인이 분기를 하지 않는 분기 명령어라면 countNotTakenBranch 변수에 1을 더한다.

(4) updateBTB, updatePHT 함수

```
void updateBTB(void) {
  if ((IDEX_OUT.BranchEqual == 1) ||
    (IDEX_OUT.BranchNotEqual == 1)) {
    for (int i = 0; i <= BTBindex; i++) {…… ①
      if (BTB[i][0] == IDEX_OUT.PC) return;
    }
    BTB[BTBindex][0] = IDEX_OUT.PC;…… ②
    BTB[BTBindex][1] = EXT.branchAddress;
    BTBindex++;
  }
  return;
}
```

```
void updatePHT(void) {
  if ((IDEX_OUT.BranchEqual == 1) ||
    (IDEX_OUT.BranchNotEqual == 1)) {

    if (EXMEM_IN.Branch == 1) {
      PHT[IDEX_OUT.PHTindex]++;…… ③

      if (PHT[IDEX_OUT.PHTindex] > 0b11)
        PHT[IDEX_OUT.PHTindex] = 0b11;

      GHR = ((GHR << 1) + 1) &…… ⑤
      0x0000003f;// update 6bit GHR
    }

    else if (EXMEM_IN.Branch == 0) {
      PHT[IDEX_OUT.PHTindex]--;…… ④

      if (PHT[IDEX_OUT.PHTindex] < 0b00)
        PHT[IDEX_OUT.PHTindex] = 0b00;

      GHR = ((GHR << 1) + 0) &…… ⑥
      0x0000003f;// update 6bit GHR
    }
    else;
  }
  else;
  return;
}
```

① updateBTB 함수는 BTB 배열에 PC와 Branch Target을 저장하는 함수로, 먼저 PC가 BTB 배열에 있는지 for 반복문으로 확인한다.

② PC가 BTB 배열에 없으면 BTBindex번째 배열의 첫 번째 열에 PC를 저장하고, 두 번째 열에 Branch Target을 저장한다.

③ updatePHT 함수는 분기 여부에 따라 2비트 패턴을 바꾸는 함수로, 분기 명령어가 분기를 했다면 PHT 배열의 PHTindex번째 값을 1만큼 올린다.

④ 분기 명령어가 분기를 하지 않았다면 PHT 배열의 PHTindex번째 값을 1만큼 내린다.

⑤ 분기 명령어가 분기를 했다면 GHR를 오른쪽으로 1만큼 비트 연산하고 1을 더한다. 6비트 GHR로 만들기 위해 0x3f와 &연산한다.

⑥ 분기 명령어인데 분기를 하지 않았다면 GHR를 오른쪽으로 1만큼 비트 연산하고 0을 더한다. 6비트 GHR로 만들기 위해 0x3f와 &연산한다.

3) PipelinedMain.c

가) PipelinedMain 함수

```
#include "PipelinedHeader.h"

INSTMEMORY[0x00400000] = { 0, };
DATAMEMORY[0x00400000] = { 0, };
Register[32] = { 0, };
BTB[0x8000][2] = { 0, };
PHT[0x8000] = { 0, };

int main(void) {
  Initialize();·····①
  ReadBinaryFiles();·····②
  while (true) {·····③
    Fetch();
    Decode();
    Execute();
    MemoryAccess();
    eof = WriteBack();

    if(eof == -1) break;
    LatchUpdate();·····④
  }
  BasicImplementation();·····⑤
  AdditionalImplementation();
  return 0;
}
```

① Initialize 함수로 파이프라인 구조체와 전역 변수를 0으로 초기화한다.
② ReadBinaryFile 함수로 바이너리 파일을 메모리 배열에 저장한다.
③ 파이프라인 단계 순서대로 Fetch, Decode, Execute, MemoryAccess, WriteBack 함수를 호출한다. WriteBack 함수가 1을 반환하면 while 반복문에서 빠져나온다.
④ LatchUpdate 함수는 한 클럭 사이클마다 레치를 업데이트하고, 구조체를 초기화한다.
⑤ Basic, AdditionalImplementation 함수로 기본 및 추가 구현 목록을 출력한다.

나) LatchUpdate 함수

```
void LatchUpdate(void) {
  // LATCH_OUT are updated from LATCH_IN
  IFID_OUT = IFID_IN;·····①
  IDEX_OUT = IDEX_IN;
  EXMEM_OUT = EXMEM_IN;
  MEMWB_OUT = MEMWB_IN;

  // initialize LATCH_IN to 0
  memset(&IFID_IN, 0, sizeof(IFID_IN));·····②
  memset(&IDEX_IN, 0, sizeof(IDEX_IN));
  memset(&EXMEM_IN, 0, sizeof(EXMEM_IN));
  memset(&MEMWB_IN, 0, sizeof(MEMWB_IN));

  (프린트 옵션 생략)
  return;
}
```

① LATCH IN 구조체에 저장된 데이터를 LATCH OUT 구조체로 복사한다.
② 새로운 데이터를 받아들이기 전, LATCH IN 구조체를 0으로 초기화한다.

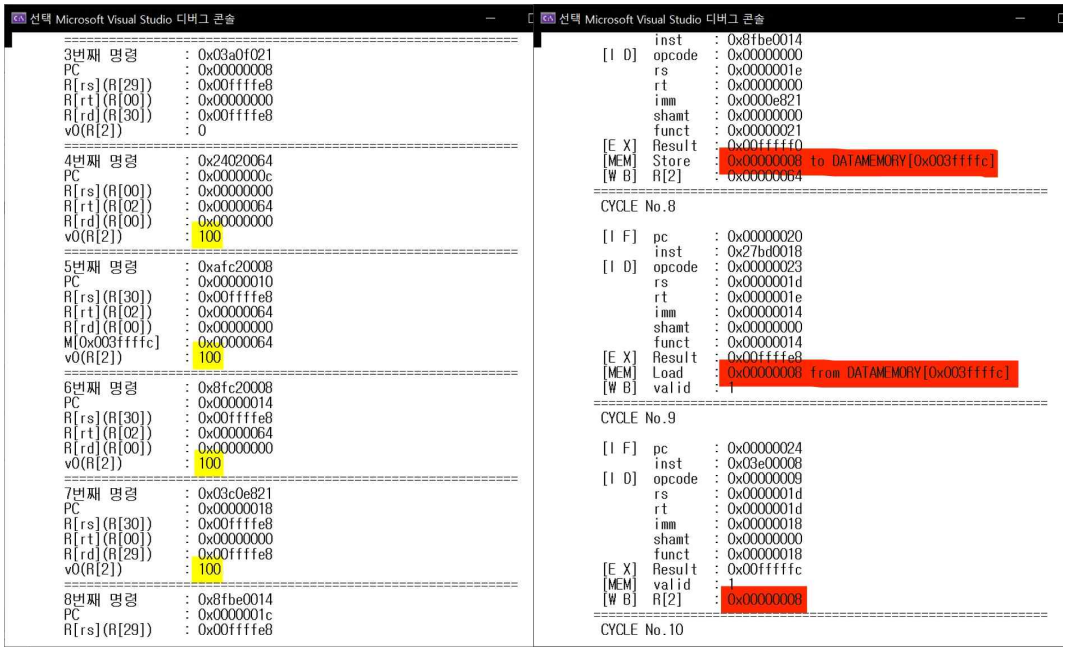
4. 문제 및 해결

제 4장, 문제 및 해결에서는 프로그램을 작성하면서 발생했던 몇 가지 문제와 그 해결 과정을 담았다. 디버깅 과정에서 문제가 됐던 부분을 하나하나 분석하고 문제를 해결하는 과정을 살펴보도록 하자.

가. 디버깅

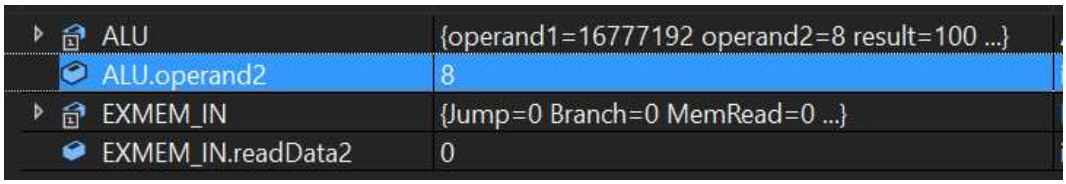
1) Execute 함수

simple2.bin의 결과가 8로 잘못 출력되는 오류가 발생했다. 파이프라인의 결과는 싱글 사이클의 결과와 같아야하기 때문에 결과는 100으로 출력되어야 했다. 다행히 simple2.bin은 클럭 사이클 수가 많지 않은 프로그램이었기 때문에 싱글 사이클의 중간 실행 결과 비교해보았다. <그림 4.1>와 같이 7번째 클럭 사이클에서 파이프라인이 8을 데이터 메모리에 저장하는 것을 확인할 수 있었다.



<그림 4.1> simple2.bin 싱글 사이클 중간 실행 결과(왼쪽)와 파이프라인 중간 실행 결과(오른쪽). 파이프라인의 7번째 클럭 사이클부터 8이 저장되고 로드되어 결과로 잘못 전달되었다.

한 단계 코드 실행으로 데이터 메모리에 writeData를 전달하는 EXMEM_OUT.readData2가 8이었고 이것은 ALU.operand2로부터 생성된 값이었다. 즉, ALUSrcB 멀티플렉서를 거치기 전의 ALU.operand2를 EXMEM_IN.readData2에 저장했어야 하는데 멀티플렉서를 지난 ALU.operand2를 저장했던 것이다. 이 오류는 EXMEM_IN.readData2 = ALU.operand2;를 ALUSrc 멀티플렉서 전으로 옮겨 해결할 수 있었다.



<그림 4.2> MUX_ALUSrcB 함수를 거친 ALU.operand2. 100이 멀티플렉서로 잘못된 순서로 인하여 8로 바뀌었다.

2) switch break 누락

if-else로 구현된 Control 함수를 switch-break로 프로그램을 최적화하는 과정에서 R-형식 명령어의 ALUSrcA, ALUSrcB 제어 신호와 R-형식 명령어의 CTR.RegWrite 제어 신호를 누락했었다. 이 누락은 gcd.bin을 실행하였을 때 무한루프가 발생하게 하는 사고를 불러일으켰고 결국 싱글 사이클 MIPS 프로그램과 다시 비교하여 오류를 해결할 수 있었다.

```
/***** ALUSrcA, B Control *****/
  switch (INST.opcode) {
  case 0x4:// beq
  case 0x5:// bne
    CTR.ALUSrcA = 0;
    CTR.ALUSrcB = 0;
    break;

  case 0xf:// lui
    CTR.ALUSrcA = 2;
    CTR.ALUSrcB = 1;
    break;

  case 0x0:// R type
    if (INST.funct == 0x0 ||
      INST.funct == 0x2) {// sll, srl
      CTR.ALUSrcA = 1;
      CTR.ALUSrcB = 2;
    }
    else {
      CTR.ALUSrcA = 0; /* DEBUG1 */
      CTR.ALUSrcB = 0;
    }
    break;

  default:
    CTR.ALUSrcA = 0;
    CTR.ALUSrcB = 1;
    break;
  }
```

```
/***** RegWrite Control *****/
  switch (INST.opcode) {
  case 0x4:
  case 0x5:
  case 0x2b:
  case 0x2:
    CTR.RegWrite = 0;
    break;
  case 0x0:
    if (INST.funct == 0x8)
      CTR.RegWrite = 0;
    else
      CTR.RegWrite = 1; /* DEBUG2 */
    break;
  default:
    CTR.RegWrite = 1;
    break;
  }
  return;
```

3) jr, jalr 종속성

simple4.bin을 실행하였는데 결과가 나오지 않고 무한루프를 도는 현상이 발생했다. simple4.bin을 살펴보며 무한루프를 돌게 하는 원인으로 몇 가지 가능성 있는 가설을 세웠다.

1. jal에 잘못된 점프 주소가 전달되었다.
2. jr에 잘못된 레지스터가 전달되었다.
3. j에 잘못된 점프 주소가 전달되었다.
4. bne에 잘못된 분기 주소가 전달되었다.

가능성 있는 원인을 살펴보던 중 jump register 명령어가 점프할 주소로 레지스터의 데이터를 사용한다는 사실한다는 사실을 간과하고 있었다는 걸 알게 되었다. 즉, 2단계(Instruction Decode)에서 점프할 주소를 계산하는 jumpAddress 함수가 jr과 jalr 명령어를 위해 레지스터의 전방전달을 고려해야 했었는데 이것을 구현하지 않아 종속성이 있는 경우를 처리하지 못한 것이었다. 따라서 jr과 jalr 명령어에서 종속성이 발생할 수 있는 종속성을 나누었고 EXMEM OUT 레치와 MEMWB OUT 레치로부터 ALUreslt, writeReg를 전방 전달하는 하드웨어를 구현했다. jr 명령어의 전방 전달을 통해 simple4.bin의 무한루프 오류를 해결할 수 있었다.

4) lui 명령어

제어 종속성과 데이터 종속성을 어느 정도 구현하고 바이너리 파일을 테스트하는 과정에서 input4.bin의 결과가 제대로 나오지 않았다. Assam 서버에 접속하여 lui 명령어의 input4.o 파일을 확인했다. input4.o 파일에서 파일 중후반부에 lui 명령어가 들어가 있다는 것을 알게 되었고, 이것은 내가 데이터 패스에 연결하지 못하고 잠시 미루어두었던 lui가 드디어 발목을 잡은 사실을 알게 해주었다. lui 명령어의 결과를 MemtoReg 멀티플렉서와 직접 연결했던 싱글 사이클과 다르게 새로운 하드웨어를 추가하는 것이 다른 파이프라인 하드웨어에 영향을 미칠 가능성이 컸기 때문에, lui 명령어를 ALU에서 { 0 + imm << 16 } 연산으로 처리하도록 했다. 따라서 ALUSrcA 멀티플렉서에 0을 인가하는 제어 신호 2번을 추가했고, Sign extend 함수에 lui 명령어에 해당하는 { imm << 16 }를 추가했다. 결과적으로 lui 명령어를 ALU 연산에 통합하는 방향이 옳았고, input4.bin으로부터 올바른 결과를 얻을 수 있었다.

나. 프로그램 구조화 및 최적화

1) nop 명령어

제어 종속성을 설계할 때, nop 명령어를 적극적으로 활용하여 프로그램 코드를 작성하고자 하였다. 주어진 bin 파일들을 살펴보았을 때 nop 명령어가 lw, beq, bne, jal, jr 등의 명령어에 삽입되는 규칙을 발견했다. nop 명령어를 이용하여 lw 명령어에서 데이터 종속성이 발생하여 한 클럭 사이클을 지연해야하는 코드를 줄일 수 있었고, beq, bne 명령어의 분기 예측이 틀렸을 때 IDEX 레치를 플러시하는 코드를 줄여주었다. jal, jr, j 명령어 또한 IFID 레치를 플러시해야 하는 코드를 줄일 수 있었다.

2) Control 함수

저번 싱글 사이클 설계 때부터 한눈에 들어오지 않았던 Control 함수를 switch로 대거 변경했다. 디폴트 제어 신호를 설정하고 if로 제어 신호를 통제하던 코드를 모두 없애고 switch의 default로 교체하였다. if-else로 인해 좌우로 지나치게 길던 코드가 위아래로 정렬되어 코드를 읽기가 훨씬 편해졌다. 그러나 switch-break로 옮기는 과정에서 일부 제어 신호를 누락하여 디버깅을 하는 불상사를 경험했다.

3) 제어 종속성 통합

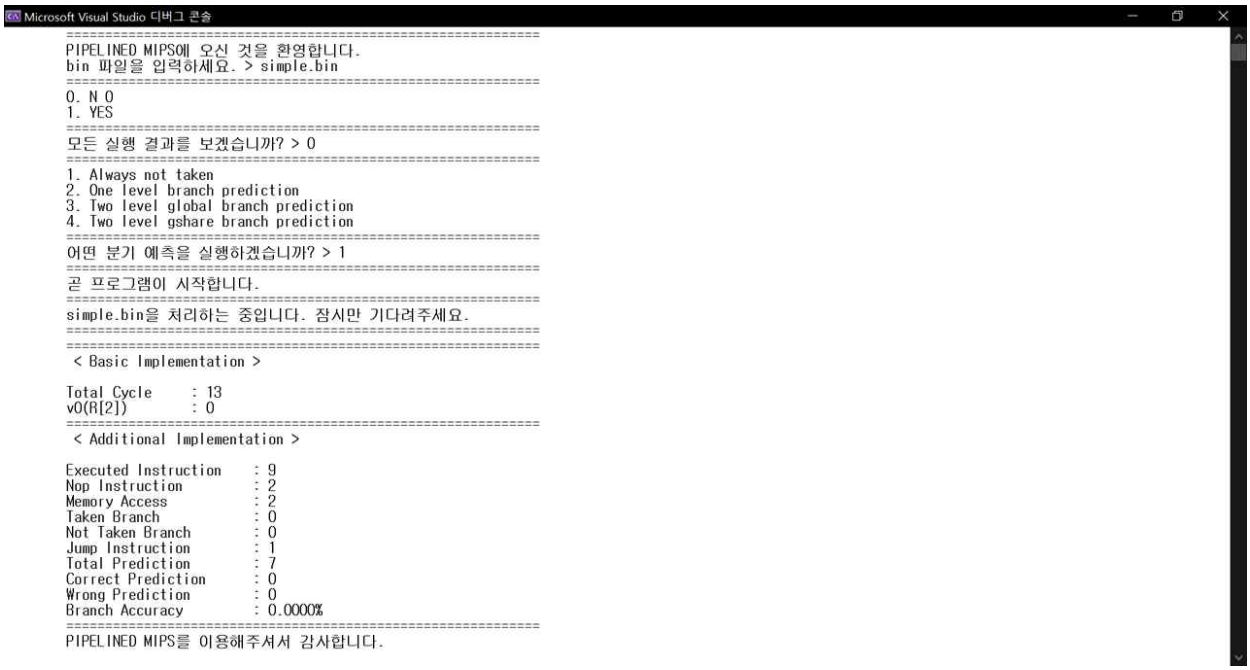
이번 프로젝트3에서 구현한 네 가지 분기 예측을 Fetch 함수와 Execution 함수 안의 branchAddress 함수로 통합하였다. branchAddress 함수를 크게 정적 분기 예측 부분과 동적 분기 예측 부분으로 나누고, 3단계(Execution)에서 분기 예측이 올바른지 확인하는 매커니즘이 거의 비슷한 One level branch prediction, Two level global branch prediction, 그리고 Two level gshare prediction을 하나로 묶었다.

5. 결과

가. 바이너리 파일 실행

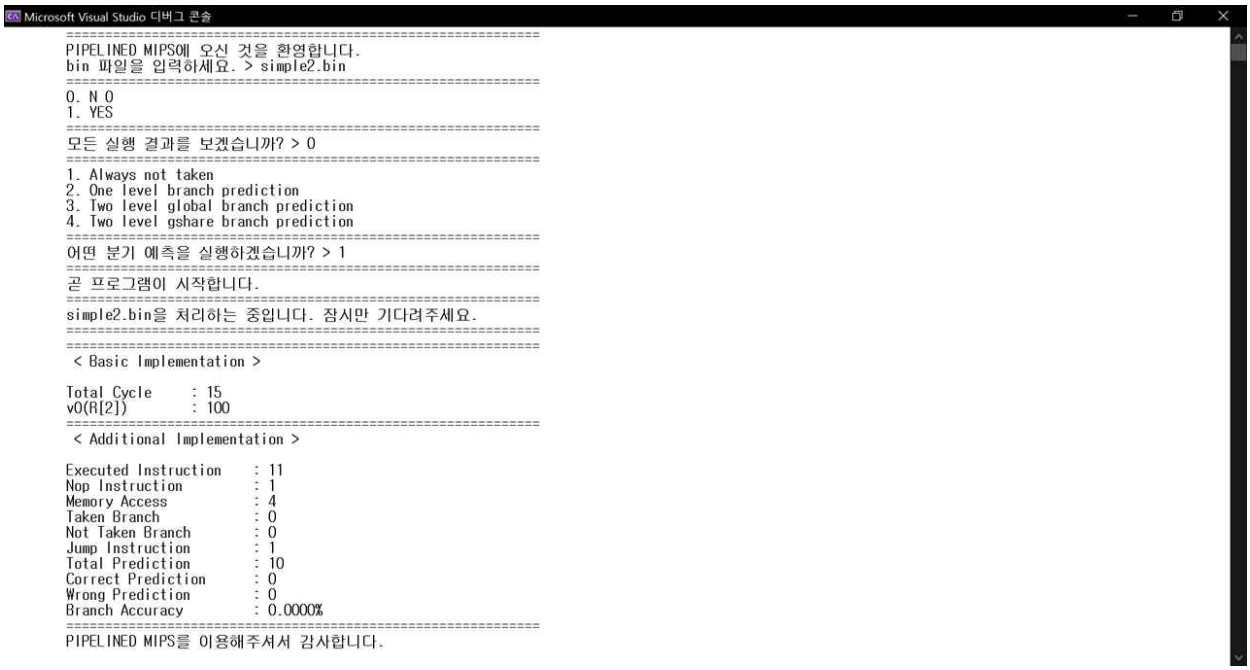
1) simple.bin

- **Always not taken to Two level Gshare Prediction**



2) simple2.bin

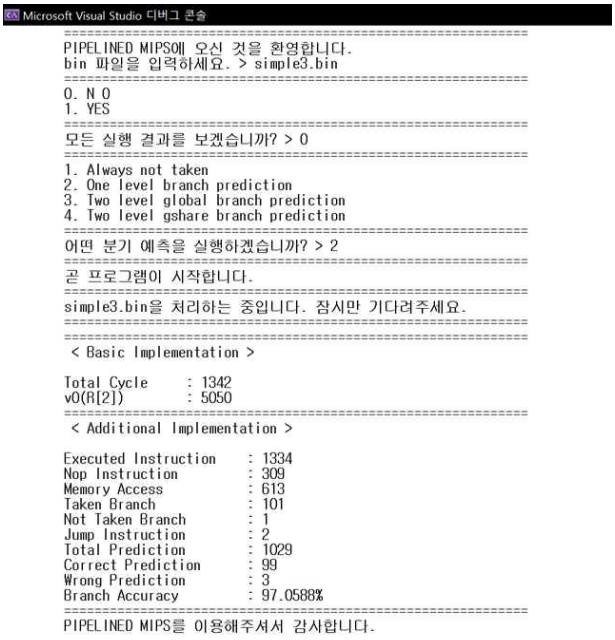
- **Always not taken to Two level Gshare Prediction**



3) simple3.bin

- **Always not taken**

- **One level branch prediction**

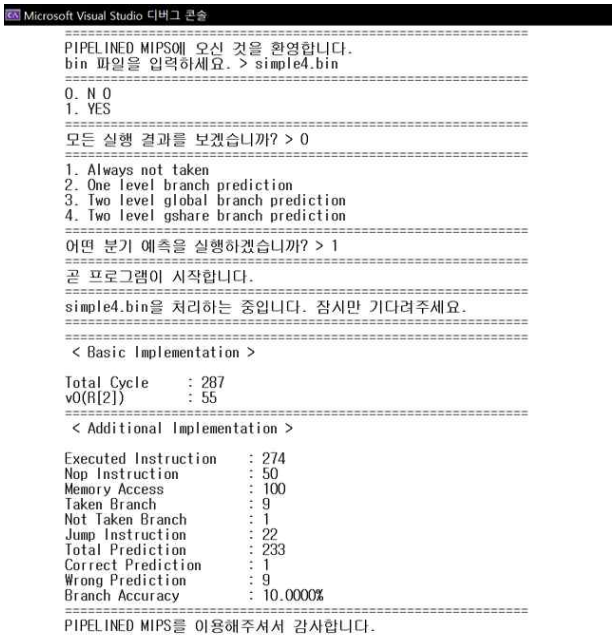


- **Two level Global branch prediction**

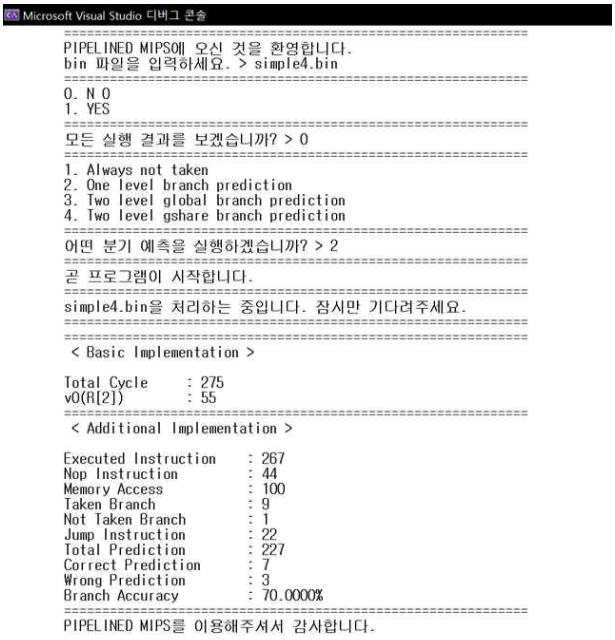
- **Two level Gshare branch prediction**

4) simple4.bin

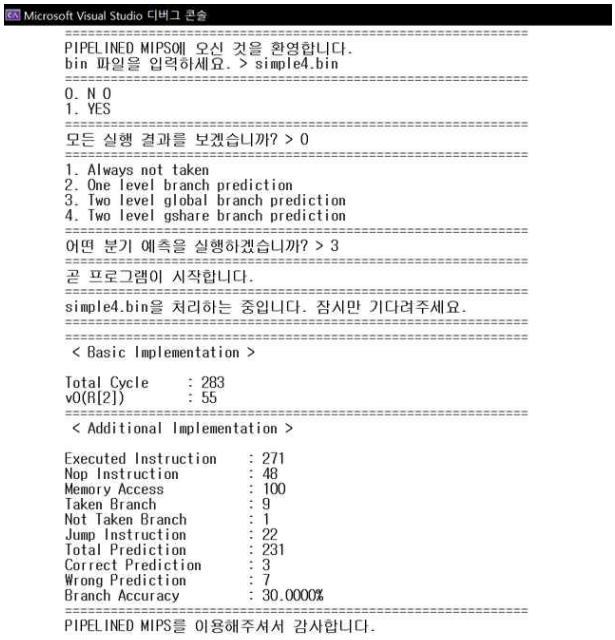
- **Always not taken**



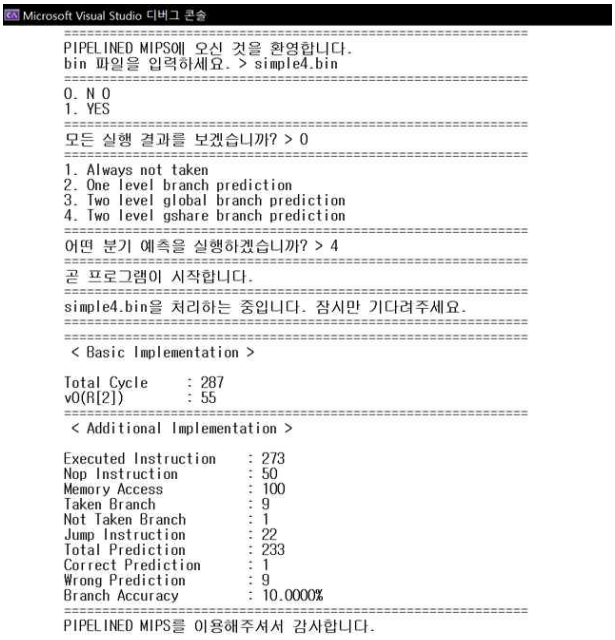
- **One level branch prediction**



- **Two level Global branch prediction**

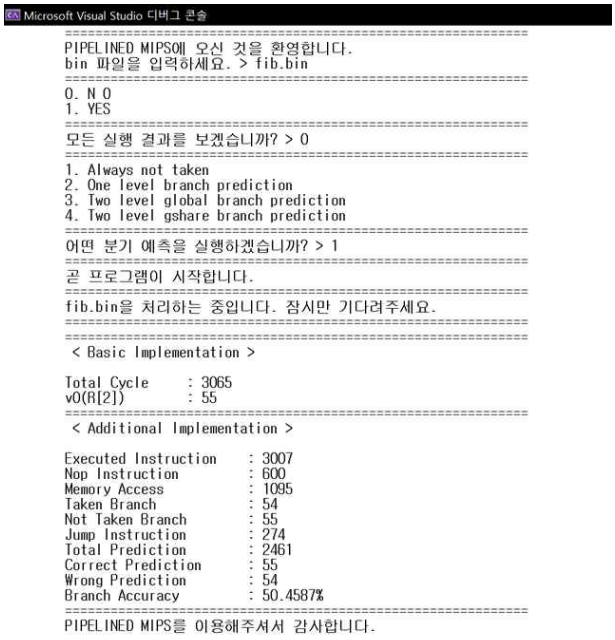


- **Two level Gshare branch prediction**



5) fib.bin

- **Always not taken**

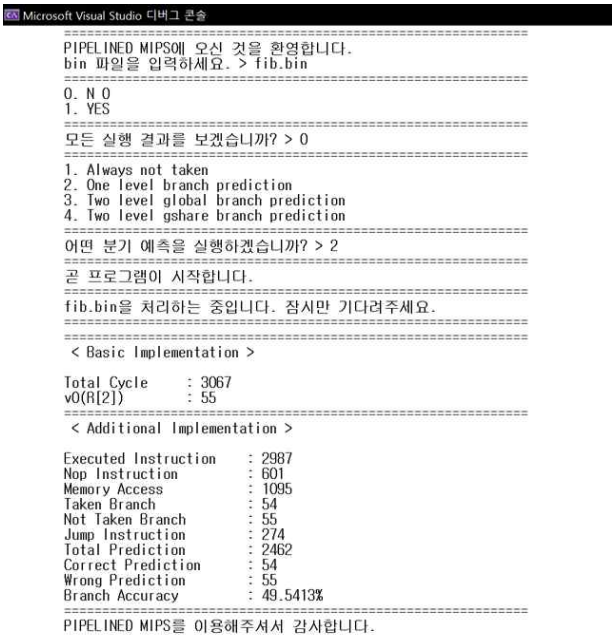


```
Microsoft Visual Studio 디버그 콘솔
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > fib.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 1
============================================================
곧 프로그램이 시작합니다.
============================================================
fib.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 3065
v0(R[2])        : 55
============================================================
 < Additional Implementation >

Executed Instruction    : 3007
Nop Instruction         : 600
Memory Access           : 1095
Taken Branch            : 54
Not Taken Branch        : 55
Jump Instruction        : 274
Total Prediction        : 2461
Correct Prediction      : 55
Wrong Prediction        : 54
Branch Accuracy         : 50.4587%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

- **One level branch prediction**

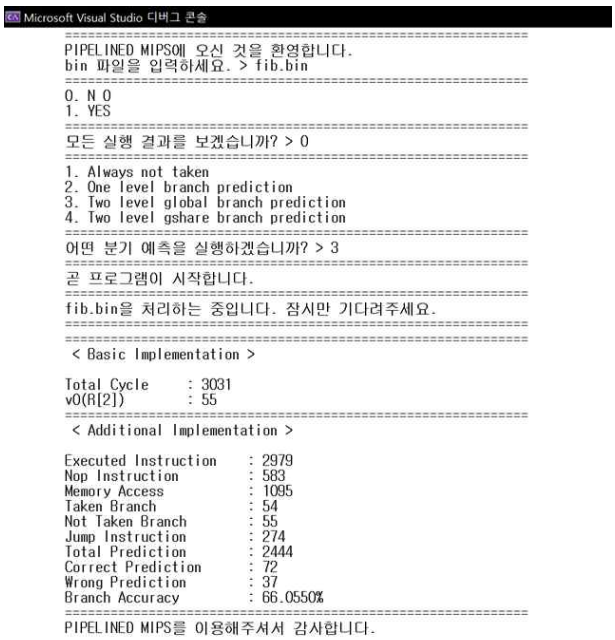


```
Microsoft Visual Studio 디버그 콘솔
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > fib.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 2
============================================================
곧 프로그램이 시작합니다.
============================================================
fib.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 3067
v0(R[2])        : 55
============================================================
 < Additional Implementation >

Executed Instruction    : 2987
Nop Instruction         : 601
Memory Access           : 1095
Taken Branch            : 54
Not Taken Branch        : 55
Jump Instruction        : 274
Total Prediction        : 2462
Correct Prediction      : 54
Wrong Prediction        : 55
Branch Accuracy         : 49.5413%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

- **Two level Global branch prediction**

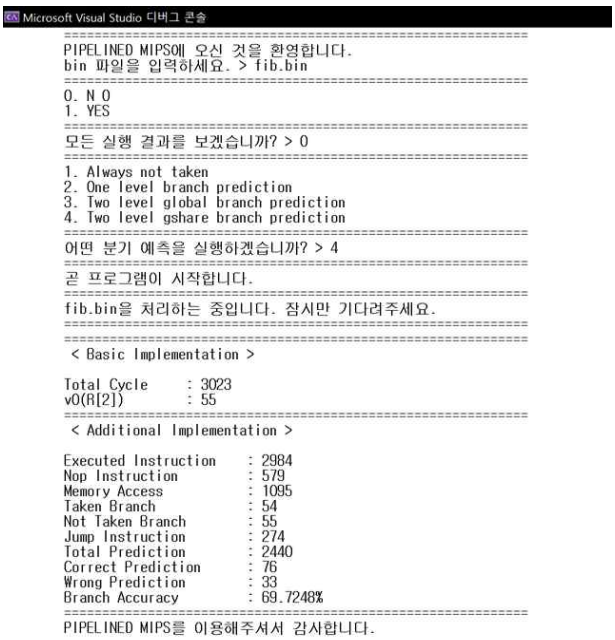


```
Microsoft Visual Studio 디버그 콘솔
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > fib.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 3
============================================================
곧 프로그램이 시작합니다.
============================================================
fib.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 3031
v0(R[2])        : 55
============================================================
 < Additional Implementation >

Executed Instruction    : 2979
Nop Instruction         : 583
Memory Access           : 1095
Taken Branch            : 54
Not Taken Branch        : 55
Jump Instruction        : 274
Total Prediction        : 2444
Correct Prediction      : 72
Wrong Prediction        : 37
Branch Accuracy         : 66.0550%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

- **Two level Gshare branch prediction**



```
Microsoft Visual Studio 디버그 콘솔
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > fib.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 4
============================================================
곧 프로그램이 시작합니다.
============================================================
fib.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 3023
v0(R[2])        : 55
============================================================
 < Additional Implementation >

Executed Instruction    : 2984
Nop Instruction         : 579
Memory Access           : 1095
Taken Branch            : 54
Not Taken Branch        : 55
Jump Instruction        : 274
Total Prediction        : 2440
Correct Prediction      : 76
Wrong Prediction        : 33
Branch Accuracy         : 69.7248%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

6) gcd.bin

- **Always not taken**

Microsoft Visual Studio 디버그 콘솔

```
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > gcd.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 1
============================================================
곧 프로그램이 시작합니다.
============================================================
gcd.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 1258
v0(R[2])        : 1
============================================================
 < Additional Implementation >

Executed Instruction    : 1209
Nop Instruction         : 285
Memory Access           : 486
Taken Branch            : 45
Not Taken Branch        : 28
Jump Instruction        : 103
Total Prediction        : 969
Correct Prediction      : 28
Wrong Prediction        : 45
Branch Accuracy         : 38.3562%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

- **One level branch prediction**

Microsoft Visual Studio 디버그 콘솔

```
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > gcd.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 2
============================================================
곧 프로그램이 시작합니다.
============================================================
gcd.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 1196
v0(R[2])        : 1
============================================================
 < Additional Implementation >

Executed Instruction    : 1175
Nop Instruction         : 251
Memory Access           : 486
Taken Branch            : 45
Not Taken Branch        : 28
Jump Instruction        : 103
Total Prediction        : 941
Correct Prediction      : 59
Wrong Prediction        : 14
Branch Accuracy         : 80.8219%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

- **Two level Global branch prediction**

Microsoft Visual Studio 디버그 콘솔

```
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > gcd.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 3
============================================================
곧 프로그램이 시작합니다.
============================================================
gcd.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 1208
v0(R[2])        : 1
============================================================
 < Additional Implementation >

Executed Instruction    : 1183
Nop Instruction         : 259
Memory Access           : 486
Taken Branch            : 45
Not Taken Branch        : 28
Jump Instruction        : 103
Total Prediction        : 945
Correct Prediction      : 53
Wrong Prediction        : 20
Branch Accuracy         : 72.6027%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

- **Two level Gshare branch prediction**

Microsoft Visual Studio 디버그 콘솔

```
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > gcd.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 4
============================================================
곧 프로그램이 시작합니다.
============================================================
gcd.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 1216
v0(R[2])        : 1
============================================================
 < Additional Implementation >

Executed Instruction    : 1188
Nop Instruction         : 264
Memory Access           : 486
Taken Branch            : 45
Not Taken Branch        : 28
Jump Instruction        : 103
Total Prediction        : 948
Correct Prediction      : 49
Wrong Prediction        : 24
Branch Accuracy         : 67.1233%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

7) input4.bin

- **Always not taken**

```
Microsoft Visual Studio 디버그 콘솔
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > input4.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 1
============================================================
곧 프로그램이 시작합니다.
============================================================
input4.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 27430474
v0(R[2])        : 85
============================================================
 < Additional Implementation >

Executed Instruction    : 25401640
Nop Instruction         : 7105428
Memory Access           : 7116606
Taken Branch            : 2028830
Not Taken Branch        : 869
Jump Instruction        : 104
Total Prediction        : 20325042
Correct Prediction      : 869
Wrong Prediction        : 2028830
Branch Accuracy         : 0.0428%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

- **One level branch prediction**

```
Microsoft Visual Studio 디버그 콘솔
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > input4.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 2
============================================================
곧 프로그램이 시작합니다.
============================================================
input4.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 23374694
v0(R[2])        : 85
============================================================
 < Additional Implementation >

Executed Instruction    : 23372890
Nop Instruction         : 5077538
Memory Access           : 7116606
Taken Branch            : 2028830
Not Taken Branch        : 869
Jump Instruction        : 104
Total Prediction        : 18297152
Correct Prediction      : 2028759
Wrong Prediction        : 940
Branch Accuracy         : 99.9537%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

- **Two level Global branch prediction**

```
Microsoft Visual Studio 디버그 콘솔
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > input4.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 3
============================================================
곧 프로그램이 시작합니다.
============================================================
input4.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 23374648
v0(R[2])        : 85
============================================================
 < Additional Implementation >

Executed Instruction    : 23372859
Nop Instruction         : 5077515
Memory Access           : 7116606
Taken Branch            : 2028830
Not Taken Branch        : 869
Jump Instruction        : 104
Total Prediction        : 18297129
Correct Prediction      : 2028782
Wrong Prediction        : 917
Branch Accuracy         : 99.9548%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

- **Two level Gshare branch prediction**

```
Microsoft Visual Studio 디버그 콘솔
============================================================
PIPELINED MIPS에 오신 것을 환영합니다.
bin 파일을 입력하세요. > input4.bin
============================================================
0. N O
1. YES
============================================================
모든 실행 결과를 보겠습니까? > 0
============================================================
1. Always not taken
2. One level branch prediction
3. Two level global branch prediction
4. Two level gshare branch prediction
============================================================
어떤 분기 예측을 실행하겠습니까? > 4
============================================================
곧 프로그램이 시작합니다.
============================================================
input4.bin을 처리하는 중입니다. 잠시만 기다려주세요.
============================================================
============================================================
 < Basic Implementation >

Total Cycle     : 23374698
v0(R[2])        : 85
============================================================
 < Additional Implementation >

Executed Instruction    : 23372935
Nop Instruction         : 5077540
Memory Access           : 7116606
Taken Branch            : 2028830
Not Taken Branch        : 869
Jump Instruction        : 104
Total Prediction        : 18297154
Correct Prediction      : 2028757
Wrong Prediction        : 942
Branch Accuracy         : 99.9536%
============================================================
PIPELINED MIPS를 이용해주셔서 감사합니다.
```

나. 결과 분석

Data Dependency / Control Dependency		Single Cycle MA	Pipelined MA Forwarding/Bypassing				Result
		. / .	Always not taken	One level	Two level Global	Two level Gshare	
simple.bin	CPU Clock Cycle	8	13	13	13	13	0
	Branch Accuracy	.	0	0	0	0	
simple2.bin	CPU Clock Cycle	10	15	15	15	15	100
	Branch Accuracy	.	0	0	0	0	
simple3.bin	CPU Clock Cycle	1330	1538	1342	1350	1354	5050
	Branch Accuracy	.	0.98%	97.06%	93.14%	91.18%	
simple4.bin	CPU Clock Cycle	243	287	275	283	287	55
	Branch Accuracy	.	10%	70%	30%	10%	
fib.bin	CPU Clock Cycle	2679	3065	3067	3031	3023	55
	Branch Accuracy	.	50.46%	49.54%	66.06%	69.72%	
gcd.bin	CPU Clock Cycle	1061	1258	1196	1208	1216	1
	Branch Accuracy	.	38.36%	80.82%	72.60%	67.12%	
input4.bin	CPU Clock Cycle	23,372,706	27,430,474	233,746,94	23,374,648	23,372,935	85
	Branch Accuracy	.	0.04%	99.95%	99.95%	99.95%	

- 파이프라인 마이크로프로세서와 싱글 사이클 마이크로프로세서의 바이너리 파일 실행 결과는 같다.
- 싱글 사이클보다 파이프라인에서 클럭 사이클 수가 전체적으로 증가한다. 그러나 클럭 사이클 시간은 파이프라인이 싱글 사이클보다 약 5분의 1로 줄어들 것이다.
- Always not taken 분기 예측은 제공된 거의 모든 파일에서 가장 효율이 나쁘다.
- 파일이 짧고 규칙적인 분기가 많을수록 One level 예측이 효율이 좋고, 파일이 길고 불규칙적인 분기가 많을수록 Two level 예측의 효율이 좋다.

6. 고찰 및 느낀 점

싱글 사이클을 설계한 경험을 바탕으로 이번 프로젝트3를 가볍게 시작했다. 먼저, 과제 설명을 읽어보았는데 데이터 종속성과 제어 종속성을 한꺼번에 구현하는 것은 불가능하다고 생각했다. 나는 우선순위에 차등을 두어 이루고자 하는 세 가지 소목표를 정했다. 첫 번째 목표는 "데이터 종속성을 해결하여 분기 명령어가 없는 simple.bin과 simple2.bin의 올바른 결과를 얻어 보자.", 두 번째 목표는 "제어 종속성을 해결하여 simple3.bin과 simple4.bin의 올바른 결과를 얻어 보자.", 그리고 마지막 목표는 "1차 디버깅을 끝내고 input4의 올바른 결과를 얻어 보자."였다.

첫 번째 소목표를 달성하기 위해서 컴퓨터로 그렸던 싱글 사이클 데이터 패스를 프린터로 뽑아 파이프라인 설계를 시작했다. 교수님께서 알려주신 내용과 전공 서적을 참고하여 연필로 파이프라인 단계를 구분하는 레치를 그리고 데이터와 제어 신호의 흐름을 레치에 연결했다. "이미 설계한 멀티플렉서는 어느 단계에 두어야 하지?", "제어 신호는 어느 단계의 레치까지 전달되어야 할까?" 등 스스로에게 던진 질문을 잊어버리지 않게 메모하여 인터넷 검색과 자료 수집을 통해 답을 얻었다. 파이프라인의 다른 단계에 영향을 미치지 않도록 섬세하게 설계하는 일은 참 어려웠다. 쉽게 해결되지 않고 끊임없이 나를 물고 늘어지는 질문 몇 가지가 많은 생각을 요구했는데 그런 질문에 대한 답은 대부분 이미 설계한 싱글 사이클에 커다란 살을 붙일 뿐만 아니라, 일부 하드웨어의 위치를 바꾸어야 했기 때문이었다. 가령, WB 종속성을 해결하는 전방전달 유닛이나, 데이터 종속성의 순서를 설계하는 부분이 그러했다. jr, jalr의 종속성을 미처 생각하지 못하고 고생한 게 개인적으로 너무 아쉽지만, 그럼에도 불구하고 디버깅을 거쳐 전방전달의 역할을 충실히 해내는 코드를 작성할 수 있었다.

두 번째 소목표를 달성하기 위해서 처음에 정적 분기 예측인 Always not taken과 동적 분기 예측인

One level branch prediction을 선택했었다. Always not taken은 구현하는데 크게 오래 걸리지 않았지만, 동적 분기 예측에 대한 학습을 충분히 하지 않아 원 One level을 이해하려고 노력했다. 추가 학습을 통해 동적 분기 예측에 PHT와 BTB를 올바르게 구현하는 게 출발점이라는 것을 알게 되었고, PHT와 BTB를 각각 1, 2차원 배열로 선언하여 하드웨어를 새로 만들었다. 한번 하드웨어를 설계하니 의외로 One level이 어렵지 않게 느껴졌고, 덕분에 Two level 분기 예측을 추가로 구현할 수 있었다.

항상 컴퓨터구조와모바일프로세서 프로젝트를 진행할 때마다 무지(無知)가 너무 고통스럽다. 알지 못하는 영역을 학습하는 일은 쉽지 않고, 그 영역을 깊게 파고들수록 튀어나오는 또 다른 영역들을 스스로 찾아 깨우쳐나가는 일은 더 쉽지 않다. 여전히 복합적인 프로그램을 차분하게 다루는 역량이 많이 부족하다는 생각도 든다. 하지만 알지 못하는 영역을 아는 영역으로 바꾸고 나면 이상하게도 더 자신감이 생긴다. 그리고 열정을 담아서 이것도 해보고, 저것도 해보고 싶은 욕심이 생긴다.

학기가 슬슬 마무리되고 있다. 간단한 계산기부터 싱글 사이클, 그리고 파이프라인까지 달려왔고 이제 캐시만을 남겨두고 있다. 남은 프로젝트4를 잘 마무리하여 컴퓨터구조와모바일프로세서 수업이 나에게 좋은 경험과 자산이 될 수 있도록 노력해보려고 한다.

7. 참고 문헌

David A. Patterson, John L. Hennessy.(1994) 컴퓨터구조 및 설계 하드웨어/소프트웨어 인터페이스(고려대학교 박명순, 하순회, 장훈 옮김). 한티 미디어