Project 3

-Pipelined MIPS-

모바일시스템공학과

32212061

변윤성

Left days 0

1. Intro

2. Concepts

2-1 Single-Cycle Microarchitecture

2-2 Multi-Cycle Microarchitecture

2-3 Pipelined Microarchitecture

2-3-1 Datapath

2-3-2 Latch

2-3-3 Control signals

2-4 Data Dependency

2-4-1 Stalling

2-4-2 Scoreboarding

2-4-3 Forwarding/Bypassing

2-5 Control Dependency

2-6 Branch Prediction

2-7 Static Prediction

2-7-1 Always Not Taken

2-7-2 Always Taken

2-7-3 BTFN (Backward Taken Forward Not Taken)

2-8 Dynamic Prediction

2-8-1 Last-Time Prediction

2-8-2 Two-Bit Counter Based Prediction

2-8-3 Two-Level Prediction

2-8-4 Global Two-Level Prediction

3. Implementation

3-1 set.h

3-2 globals.c

3-3 main.c

3-4 functions.c

3-4-1 control unit

3-4-2 ALU

3-4-3 IF 단계

3-4-3 ID 단계

3-4-4 EX 단계

3-4-5 mem 보조

3-4-6 MEM 단계

3-4-7 WB 단계

3-4-8 latch update

3-5 branch\_control.c

3-5-1 Always Not Taken

3-5-2 Always Taken

3-5-3 BTFN

3-5-4 One-Bit Branch

3-5-5 Two-Bit Branch

3-5-6 Gshare

3-5-7 LocalTwoLevel

3-6 hazard\_handling.c

4. Result

5. Analysis

6. Feelings

1. Intro

컴퓨터가 프로그램을 실행하는 것은 명령어들을 처리하는 과정이다. 이러한 과정중에 가장 간단한 방법은 Single-Cycle 방식이다. 모든 명령어를 종류에 상관없이 하나의 사이클안에 완료한다. 설계는 단순하지만 느린 명령어가 있다면 빨리 끝나는 작업이라도 그 하나의 명령어 때문에 느려지니 효율이 떨어진다. 이런 비효율을 개선하기 위해 하는 것이 Multi-Cycle 방식이다. 명령어 처리를 여러 단계로 나누고 각 단계를 사이클로 처리한다. 각 단계마다 실행시간도 달라지니 간단한 연산은 적은 사이클로 빨리 끝내고 긴 연산은 여러 개의 사이클을 사용할 수 있다. 여기서 더 성능을 극대화하기 위해 Pipelining이 나왔다. Multi-Cycle처럼 명령어 처리 과정을 여러 단계로 나누는 것은 동일하지만 이 단계들을 서로 다른 명령어들이 동시에 사용하게 된다. 이런 걸 직접 알아보기 위해 Pipelined MIPS simulator를 구현해보는 게 이번 과제이다. 이번 과제를 통해서 Single-Cycle과는 어떤 부분이 다르고 Pipeline은 얼마나 더 빠른지 알 수 있을거 같다.

2. Concepts

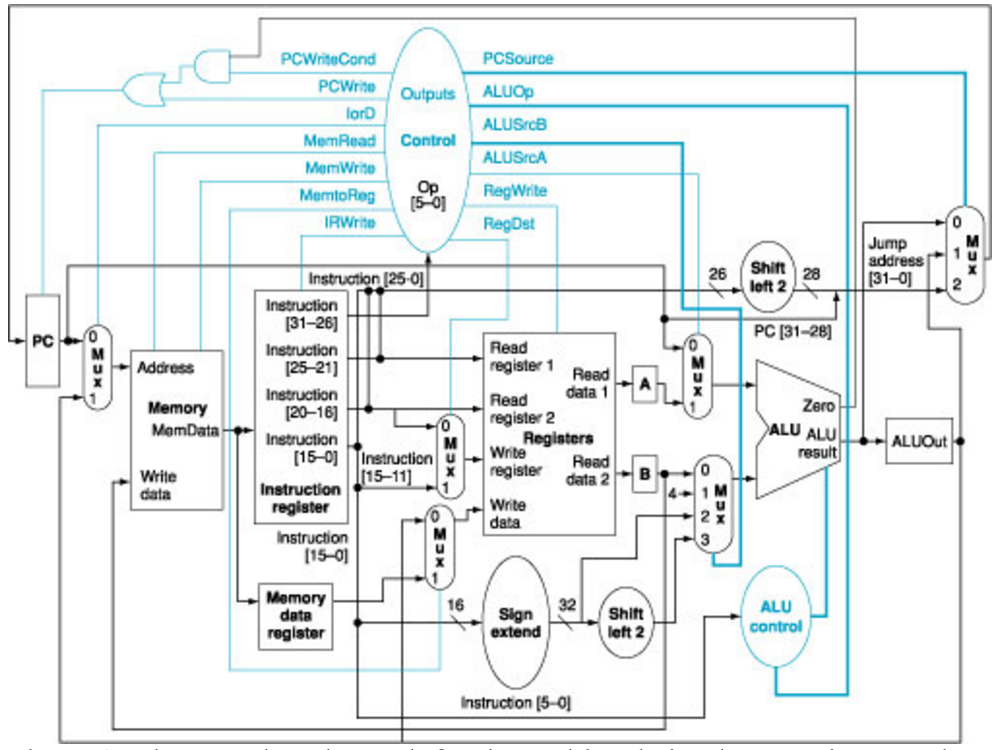
**2-1 Single-Cycle Microarchitecture**

도표, 평면도, 기술 도면, 개략도이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

가장 기본적인 프로세스 설계 방식으로 이름에 맞게 모든 종류의 명령어를 단 하나의 긴 사이클동안 완료되도록 설계된다. 구조가 단순해서 이해하기 쉽다는 장점이 있지만 간단한 연산도 불필요하게 긴 시간을 소모하게 될 수도 있어서 성능이 좋지는 않다.

**2-2 Multi-Cycle Microarchitecture**



단일 사이클 방식의 성능을 개선하고자 멀티 사이클 구조가 등장한다. 명령어 처리 과정을 여러 단계로 나누고 각 단계를 더 짧아진 사이클동안 처리한다. 명령어마다 필요한 단계 수가 다르므로 전체 실행 시간도 명령어별로 달라진다. 속도도 높일 수도 있고 ALU 같이 하드웨어도 여러 사이클에서 재사용이 가능하기 때문에 성능적으로 향상된다.

**2-3 Pipelined Microarchitecture**

처리 속도를 더 높이기 위해 파이프 라인 구조가 나왔다. 멀티 사이클처럼 명령어 처리 과정을 여러 단계 (MIPS에서는 IF, ID, EX, MEM, WB)로 나누지만 차이점이라 하면 이런 단계가 동시에 사용된다. 즉 한 명령어가 다음 단계로 넘어가면 비어있는 이전 단계를 바로 다음 명령어가 채우면서 겹쳐서 실행된다.

각 명령어가 시작부터 끝까지 완료되는 데 걸리는 시간은 여전치 여러 사이클이 필요하지만 매 사이클마다 하나의 명령어가 결과물을 내놓을 수 있다. 이런 식으로 진행되면 프로그램의 성능은 엄청 향상된다. 이러한 구조를 원활하게 동작 시키려면 각 단계 사이에서 명령어 정보와 중간 결과들을 보관하는 Latch라는 하드웨어가 필수적이고 각 단계를 제어하는 Control Signal도 단계에 맞혀서 함께 전달되야 된다.

**2-3-1 Datapath**

**텍스트, 도표, 평면도, 개략도이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

그림 하단의 문구 "WriteReg must arrive at same time as Result"는 파이프라인의 중요한 동기화 원리를 나타낸다. 결과값과 그 값을 쓸 레지스터 주소가 Writeback 단계에 동시에 도착해야 정확한 작업이 이루어진다는 뜻이다. 이는 목적지 레지스터 주소 정보도 데이터처럼 파이프라인을 따라 함께 전달되어야 함을 의미하고 그림의 빨간색 선이 이 경로를 보여준다. 이 데이터패스는 여러 명령어를 병렬로 처리하여 성능을 높이는 파이프라인의 핵심 구조를 나타낸다.

**2-3-2 Latch**

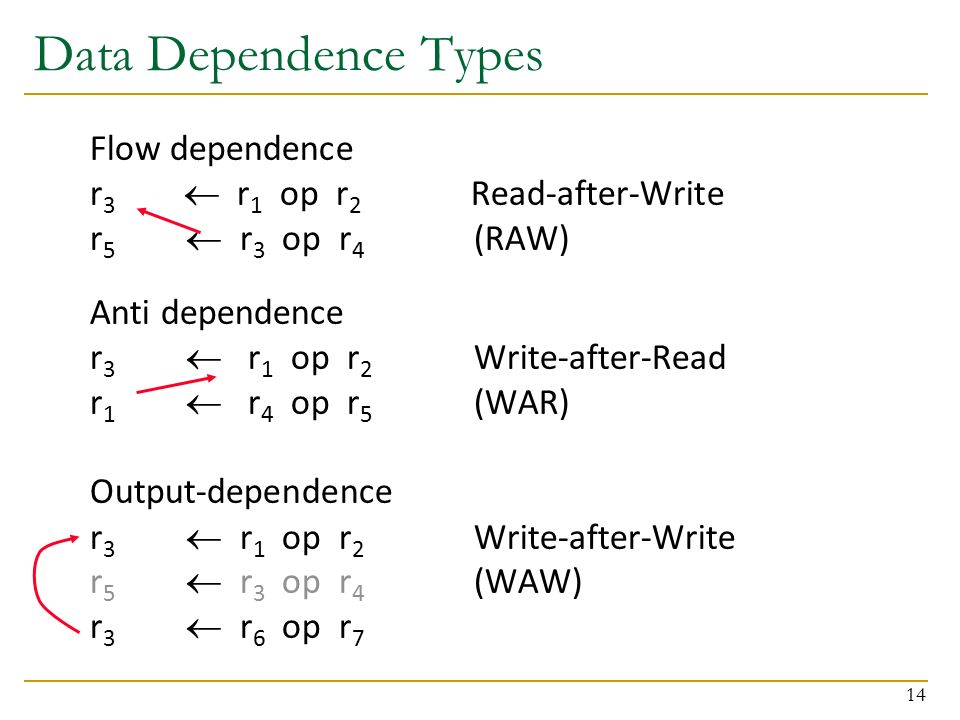
Latch는 기본적으로 신호에 동기화되어 작동하는 레지스터로 한 단계의 처리결과와 다음 단계를 위한 제어 신호 등을 저장했다가 다음 사이클에 다음 단계로 전달하는 역할을 한다. 이 프로젝트의 시뮬레이터에서는 C언어의 구조체 (struct 등)을 사용해서 각 파이프라인 단계 사이(IF/ID, ID/EX, EX/MEM, MEM/WB)의 래치를 구현해서 각 래치에는 해당 단계에서 필요한 데이터와 제어신호들이 멤버 변수로 포함된다.

**2-3-3 Control signals**

파이프라인 구조에서는 여러 명령어가 동시에 서로 다른 단계를 거치게 된다. 이때 각 명령어마다 실행해야 할 동작이 다르기 때문에 각 파이프라인 단계에서 필요한 제어 신호들을 정확히 전달하고 적용하는 것이 중요하다. 예를 들어 어떤 명령어는 레지스터에 값을 써야 하고 어떤 명령어는 메모리에서 값을 읽어야 한다. 이런 동작들을 구분하고 제어하기 위해 제어 신호들이 사용된다. 이 제어 신호들은 명령어의 해독(Decode) 단계를 거치면서 생성되고 파이프라인 래치를 통해 다음 단계로 계속 전달되어 각 단계가 올바른 동작을 수행하도록 돕는다. 결국 제어 신호는 파이프라인이 마치 오케스트라처럼 각자의 역할을 정확히 수행하게 만드는 지휘자 같은 역할을 한다고 볼 수 있다.

**2-4 Data Dependency**

파이프라인에서 명령어를 중첩해서 처리하다 보면 이전 명령어가 아직 결과를 내놓지 않았는데 다음 명령어가 그 결과를 사용해야 하는 경우가 생긴다. 이런 상황을 데이터 종속성이라고 부른다. 데이터 종속성은 크게 세 가지 종류로 나눌 수 있다.



그림과 같이 각각의 레지스터가 같은 레지스터를 사용하려고 할 때 문제가 일어난다. 먼저 이전 명령어가 특정 레지스터에 값을 쓴 후에 다음 명령어가 그 레지스터 값을 읽으려고 할 때 발생하는 RAW (Read-after-Write) 종속성이 있다. 그리고 이전 명령어가 레지스터 값을 읽기도 전에 다음 명령어가 같은 레지스터에 값을 덮어쓰려고 할 때 생기는 WAR (Write-after-Read) 종속성도 있다. 마지막으로 이전 명령어가 레지스터에 값을 쓰기도 전에 다음 명령어가 같은 레지스터에 또 다른 값을 쓰려고 하는 WAW (Write-after-Write) 종속성이 있다. 이런 데이터 종속성들은 파이프라인의 흐름을 방해하고 잘못된 결과를 만들 수 있기 때문에 반드시 해결해야 하는 중요한 문제다.

**2-4-1 Stalling**

**텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

데이터 종속성 문제를 해결하는 가장 간단한 방법이 바로 stalling이다. 스톨링은 말 그대로 파이프라인을 잠시 멈추는 것이다. 만약 특정 명령어가 이전 명령어의 결과를 기다려야 한다면 필요한 데이터가 준비될 때까지 해당 명령어와 그 뒤에 오는 명령어들의 진행을 멈춘다. 필요한 데이터가 준비되면 멈췄던 파이프라인 단계를 다시 진행시킨다. 스톨링은 구현하기는 비교적 쉽지만 파이프라인이 멈추는 시간만큼 성능 손실이 발생한다는 단점이 있다.

**2-4-2 Scoreboarding**

Scoreboarding은 데이터 종속성을 관리하고 스톨을 발생시킬지 여부를 결정하는 중앙 집중적인 제어 방식이다. 각 레지스터마다 현재 사용 중인지 아닌지를 나타내는 플래그(valid bit) 같은 것을 둔다. 명령어가 특정 레지스터를 사용하려고 할 때 스코어보드는 해당 레지스터의 상태를 확인한다. 만약 레지스터가 아직 이전 명령어에 의해 쓰이는 중이어서 유효하지 않은 상태라면 해당 명령어가 레지스터를 읽거나 쓰지 못하도록 파이프라인을 멈춘다. 이후 레지스터가 사용 가능해지면 멈췄던 명령어를 다시 진행시킨다. 스코어보딩은 하드웨어 자원을 효율적으로 관리하고 데이터 종속성을 체계적으로 처리할 수 있지만 제어 로직이 복잡해질 수 있다.

**2-4-3 Forwarding/Bypassing**

**텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

Forwarding/Bypassing은 데이터 종속성으로 인한 스톨을 줄이는 매우 효과적인 기법이다. 이전 명령어가 계산을 마친 결과값이 아직 레지스터 파일에 최종적으로 저장되지 않았더라도 그 결과값이 필요한 다음 명령어가 있다면 중간 결과값을 파이프라인의 뒤에서 앞으로 직접 전달해주는 방식이다. 예를 들어 ALU 연산 결과가 EX 단계를 마치자마자 바로 다음 명령어의 EX 단계 입력으로 사용될 수 있도록 전달해준다. 이렇게 하면 굳이 레지스터 파일에 값이 쓰일 때까지 기다릴 필요가 없어져서 불필요한 스톨을 상당 부분 제거할 수 있다. 포워딩은 파이프라인의 효율을 크게 높여주는 핵심 기술 중 하나다. 그렇기에 이번 과제에서 포워딩으로 구현을 했다.

**2-5 Control Dependency**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

제어 종속성은 주로 분기(branch) 명령어와 같은 제어 흐름 변경 명령어 때문에 발생한다. 파이프라인은 기본적으로 다음 명령어가 현재 명령어의 바로 다음 주소에 있다고 가정하고 미리 명령어들을 가져와서 처리한다. 하지만 분기 명령어의 경우 조건에 따라 프로그램 카운터(PC) 값이 예상치 못한 주소로 점프할 수 있다. 이렇게 되면 파이프라인이 미리 가져와서 처리하던 명령어들은 쓸모없게 되고 파이프라인을 비우고(flush) 올바른 위치의 명령어를 다시 가져와야 한다. 이런 상황을 제어 종속성 또는 제어 해저드라고 부른다. 제어 종속성은 파이프라인의 성능을 저하시키는 주요 원인 중 하나다.

**2-6 Branch Prediction**

**텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

**텍스트, 스크린샷, 도표, 다채로움이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.**

첫 번째 그림을 보면 분기 예측이 없을 때는 분기 명령어 다음에 엉뚱한 명령어들을 파이프라인에 미리 채워 넣었다가 분기가 확정되면 이 명령어들을 다 버리고 새로 시작해야 해서 시간이 오래 걸린다. 하지만 분기 예측을 해서 만약 예측이 딱 맞아떨어지면 파이프라인을 멈추거나 버리는 일 없이 쭉쭉 실행해서 시간을 훨씬 아낄 수 있다. 즉 분기 예측은 파이프라인이 헛도는 걸 막아줘서 프로그램 실행 속도를 높여주는 중요한 기술이다.

두 번째 그림은 만약 분기 예측이 틀렸을 때 어떤 상황이 발생하는지 보여준다. CPU가 분기가 안 일어날 거라고 잘못 예측하고 다음 명령어들을 파이프라인에 잔뜩 밀어 넣었는데 나중에 "분기해야 된다" 라고 깨달으면 이미 파이프라인에 들어와 있는 명령어들을 싹 다 비워내야 한다. 이걸 파이프라인 플러시라고 하는데 이렇게 되면 시간 손해가 발생한다.

**2-7 Static Prediction**

컴파일 시점 분기 예측은 프로그램이 실행되기 전 컴파일러가 코드를 기계어로 번역하는 단계에서 분기 명령어의 결과를 미리 결정하여 파이프라인의 흐름을 제어하는 방식이다. 이는 프로그램 실행 중 예측에 필요한 복잡한 하드웨어나 지연 없이 분기 방향을 정하려는 시도이다.

**2-7-1 Always Not Taken**

Always Not-Taken 방식은 분기 명령어를 만나면 해당 분기의 조건이 항상 성립하지 않는다고 가정하는 가장 단순한 정적 예측 방법이다. 따라서 파이프라인은 다음 실행될 명령어를 현재 PC 값에 4를 더한 PC+4 주소에서 가져오도록 처리된다. 이러한 결정은 컴파일 단계에서 이루어진다.

이 방식의 장점은 구현이 매우 쉽고, 분기 예측을 위한 별도의 하드웨어 추가가 필요 없다는 점이다. 그러나 만약 EX (실행) 단계에서 실제 분기 조건이 만족하는 것으로 판명되면 이미 파이프라인으로 잘못 가져온 명령어들을 초기화하는 작업이 수반된다. 이 때문에 분기가 자주 발생하거나 예측이 많이 빗나가는 프로그램에서는 정확도나 전체적인 처리 성능 향상을 크게 기대하기는 어렵다.

**2-7-2 Always Taken**

Always Taken 방식은 Always Not-Taken 방식과 정반대로, 분기 명령어가 나타나면 항상 해당 분기의 조건이 성립한다고 가정한다. 따라서 다음 명령어는 분기 명령어에 명시된 목표 주소(Branch Target Address)에서 가져와 실행하게 된다. 이 방식은 특정 유형의 코드, 예를 들어 루프의 끝에서 반복적으로 분기하는 경우 Always Not-Taken 방식보다 더 높은 정확도를 보일 수 있다.

하지만 이 방식을 효율적으로 구현하기 위해서는 분기 목표 주소를 신속하게 파악할 수 있는 BTB (Branch Target Buffer)와 같은 추가적인 하드웨어가 필요할 수 있다. Always Not-Taken 방식과 마찬가지로 컴파일러가 컴파일 시점에 해당 분기가 항상 발생할 것이라는 힌트를 코드에 포함시키는 방식으로 동작하며 만약 실제 분기 조건이 예측과 다를 경우에는 파이프라인 플러시 작업이 필요하다.

**2-7-3 BTFN (Backward Taken, Forward Not Taken)**

BTFN (Backward Taken, Forward Not Taken) 방식은 앞서 설명한 두 가지 정적 예측 방식보다 조금 더 정교한 접근법을 사용한다. 이 방식의 핵심 아이디어는 분기 명령어의 목표 주소 방향성에 기반하여 예측하는 것이다.

만약 분기 명령어의 목표 주소가 현재 명령어의 주소보다 이전이라면, 이는 주로 루프의 끝에서 루프의 시작 부분으로 돌아가는 경우에 해당하므로 이 분기는 조건이 성립할 것으로 예측한다. 반대로 목표 주소가 현재 명령어의 주소보다 이후라면 이는 주로 조건문의 특정 블록을 건너뛰는 경우에 해당하므로 이 분기는 조건이 성립하지 않을 것으로 예측하고 다음 명령어는 PC+4에서 가져온다. 이 방식 역시 분기 목표 주소를 효율적으로 관리하기 위해 BTB와 같은 하드웨어가 필요할 수 있다.

**2-8 Dynamic Prediction**

동적 분기 예측은 프로그램이 실제로 실행되는 과정에서 분기 명령어의 이력을 바탕으로 다음 분기 결과를 예측하는 기법이다. 정적 예측과 달리 프로그램의 동적인 행위를 반영하여 예측 정확도를 높이려는 시도이다. 이 방식을 사용하기 위해서는 명령어 인출 단계에서 몇 가지 정보를 확인하고 이를 저장할 추가적인 하드웨어가 필요하다. 대표적으로 분기 목표 주소를 저장하는 BTB(Branch Target Buffer)와 이전 분기의 실행 결과를 기록하는 BHT(Branch History Table)가 사용된다.

**2-8-1 Last-Time Prediction**

1-bit 예측 방식은 BHT에 각 분기 명령어의 가장 최근 실행 결과만을 단일 비트로 저장하여 예측에 활용한다. 예를 들어 이전에 분기가 발생했다면 1로, 발생하지 않았다면 0으로 기록하는 식이다. 예측은 이 저장된 비트 값을 따른다.

이 방식은 구현이 비교적 간단하며, 동일한 분기 결과가 반복되는 프로그램에서는 정적 예측보다 높은 정확도를 보일 수 있다. 그러나 분기 결과가 자주 바뀌는 패턴, 예를 들어 Taken과 Not-Taken이 번갈아 나타나는 경우에는 매번 예측이 빗나가 정확도가 매우 낮아지는 단점이 있다. 상태 변경이 너무 빠르게 일어나기 때문이다.

**2-8-2 Two-Bit Counter Based Prediction**

2-bit 카운터 기반 예측 방식은 1-bit 예측의 단점을 보완하기 위해 각 분기 명령어의 이력을 2개의 비트로 표현한다. 일반적으로 2비트는 네 가지 상태(Strongly Taken, Weakly Taken, Weakly Not-Taken, Strongly Not-Taken)를 나타낸다. 분기가 실제로 발생하면 카운터 값을 증가시키고, 발생하지 않으면 감소시키는 방식으로 상태를 갱신한다. 예측은 주로 카운터의 최상위 비트를 따른다. 예를 들어 두 비트가 '11' 또는 '10'이면 Taken으로 예측하고 '01' 또는 '00'이면 Not-Taken으로 예측한다.

이 방식은 분기 결과가 한두 번 바뀌더라도 예측이 바로 반전되지 않기 때문에 1-bit 예측보다 안정적인 성능을 제공한다. 그 결과 예측 정확도는 일반적으로 85~90%까지 기대할 수 있다. 하지만 1-bit 예측보다 더 많은 하드웨어 자원(BHT의 각 엔트리당 2비트)을 필요로 한다. 그럼에도 불구하고 명령어의 수가 매우 많아지면 여전히 예측 실패로 인한 성능 저하가 발생할 수 있다. 예를 들어 99%의 정확도라 할지라도 1000개의 명령어 중 10개는 예측에 실패하며 이는 전체 실행 시간에 영향을 미친다.

**2-8-3 Two-Level Prediction**

2단계 예측은 단일 단계의 이력만으로는 잡아내기 어려운 복잡한 분기 패턴을 학습하여 예측 정확도를 더욱 높이기 위해 개발된 기법이다. 이 방식은 크게 두 가지 주요 하드웨어 구성요소를 사용한다. 첫번째는 분기 이력 레지스터(BHR 또는 GHR)이고, 두번째는 패턴 이력 테이블(PHT)이다. BHR은 최근 실행된 분기들의 결과를 기록하고, PHT는 이 BHR의 값을 인덱스로 사용하여 특정 패턴에 대한 예측 카운터(주로 2-bit 카운터)를 저장한다.

**2-8-4 Global Two-Level Prediction**

전역 이력 기반 2단계 예측에서는 모든 분기 명령어의 최근 실행 결과가 하나의 GHR에 기록된다. 예를 들어, GHR이 m개의 비트로 구성되어 있다면, 가장 최근 m번의 분기 결과(Taken/Not-Taken)가 이 레지스터에 저장된다. 이 GHR의 값을 PHT의 인덱스로 사용하여 해당 패턴에 대한 2-bit 예측 카운터를 찾고, 이를 통해 다음 분기를 예측한다. 분기가 실행된 후에는 실제 결과에 따라 GHR이 갱신되고, PHT의 해당 엔트리도 업데이트된다.

이 방식은 서로 다른 분기 명령어들 사이의 연관성까지 학습할 수 있다는 장점이 있다. 하지만 서로 다른 위치의 분기 명령어들이 우연히 동일한 GHR 패턴을 생성하여 PHT의 같은 엔트리를 참조하게 되면 서로의 예측에 간섭을 일으켜 정확도를 떨어뜨릴 수 있다.

3. Implementation

구현에 있어서 set.h, globals.c, main.c, functions.c, hazard\_handling.c, branch\_control.c, prediction.c로 총 7개의 파일들로 나눴다. 일단 이렇게 나눈 이유는 단순하게 가독성이 쉽고 작업이 편하다. 자세히 설명해보면 분기 예측과 같은 부분은 prediction.c 와 branch\_control.c 두 파일만 관리하고 prediction.c는 다양한 분기 예측 알고리즘을 관리하고 branch\_control.c는 이러한 예측의 정확성을 검증하며 예측 실패 시 필요한 제어 로직을 처리한다. 이 2개의 파일은 분기 예측 알고리즘을 관리하려고 분할했다. 파이프라인 수행 중 발생할 수 있는 데이터 해저드 및 제어 해저드의 처리는 hazard\_handling.c에서 처리하려고 나눴다. set.h 파일은 모든 모듈이 공유하는 전역 변수 선언, 구조체 정의, 함수 프로토타입 등을 포함하여 시스템 전체의 인터페이스 및 공통 데이터 구조를 정의하기 위해 만들었다. globals.c는 set.h에서 extern로 선언된 전역 변수들의 실제 정의 및 전역 상태 정보의 저장소 역할을 한다. main.c는 프로그램의 전체적인 흐름을 나타내고 function.s는 이번 과제 중에 제일 많이 수정하고 제일 힘들게 만든 파일이 아닐까 싶은데 CPU의 파이프라인 단계 (IF, ID, EX, MEM, WB)의 로직과 ALU, 제어 유닛과 같은 주요 연산들이 담겨 있다.

**3-1 set.h**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 폰트, 스크린샷, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

여기에서는 파이프라인의 주요 3단계(EX, M, WB)에서 사용될 제어 신호들을 그룹화하기 위한 구조체를 각각 정의한다.

struct EX는 Execute 단계에서 필요한 제어 신호들을 포함한다. RegDst는 목적지 레지스터 필드 선택, ALUSrc는 ALU의 두 번째 입력 선택, ALUOp는 ALU 연산 종류 지정, jr, jal, branch는 각각 점프 레지스터, 점프 및 링크, 분기 명령어 여부, zero는 제로 확장 여부를 나타낸다.

struct M은 Memory Access 단계의 제어 신호로, memoryWrite는 메모리 쓰기 활성화, memoryRead는 메모리 읽기 활성화, PCSrc1은 J 타입 명령어의 PC 변경 여부를 나타낸다.

struct WB는 Write Back 단계의 제어 신호로, memtoReg는 메모리에서 읽은 값을 레지스터에 쓸지 ALU 결과를 쓸지 결정, PCSrc2, RegWrite는 레지스터 쓰기 활성화, j\_and\_l\_toReg는 JAL 명령어 수행 시 PC+4 값을 레지스터에 저장할지 여부를 나타낸다.

텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 디스플레이이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

struct IFID\_는 IF 단계에서 ID 단계로 전달될 정보를 담는다. struct IDEX\_는 ID 단계에서 EX 단계로 전달될 정보를 담는다. 여기에는 앞서 정의된 제어 신호 구조체들, 래치 유효성 비트 등이 담겨 있다. 다른 구조체도 비슷한 구조로 담겨 있다.

텍스트, 스크린샷, 폰트, 메뉴이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

시뮬레이터 전체에서 사용될 전역 변수들을 extern 키워드를 사용하여 선언한다. 이 변수들의 메모리 할당은 globals.c에 존재한다. 앞서 정의된 래치 구조체 타입으로 각 파이프라인 래치를 크기 2의 배열로 선언한다. 일반적으로 현재 사이클과 다음 사이클의 값을 구분하여 파이프라인 동작을 시뮬레이션하기 위함이다.

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

프로젝트 내의 다른 C 소스 파일에 정의된 함수들의 프로토타입을 선언한다. 각 함수가 다른 파일에서 호출될 수 있도록 컴파일러에게 함수를 알려준다.

**3-2 globals.c**

텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

MIPS 프로세서의 핵심적인 상태를 저장하기 위한 전역 변수들과 분기 예측 메커니즘의 동작에 필요한 다양한 테이블과 상태 변수들을 정의한다.

텍스트, 폰트, 스크린샷, 번호이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

시뮬레이션의 진행 상태와 프로세서의 특정 레지스터 값을 저장하는 변수들이랑 분기 예측의 성능을 평가하기 위한 통계치를 수집하는 변수들이다.

텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

시뮬레이션 동안 실행된 명령어의 종류별 횟수 및 총 명령어 수를 기록하는 변수들이랑 파이프라인의 각 단계 사이에 데이터를 전달하고 상태를 유지하기 위한 latch 변수들을 정의한다. set.h에 정의된 각 래치 구조체(IF\_ID, ID\_EX, EX\_MEM, MEM\_WB) 타입으로 크기가 2인 배열로 선언된다. 일반적으로 한쪽은 현재 사이클에서 다음 단계로 전달될 데이터를 쓰고 다른 쪽은 이전 사이클에서 전달받은 데이터를 현재 단계에서 읽는 이중 버퍼링 또는 현재/다음 상태를 구현하기 위해 사용됐다.

**3-3 main.c**

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

MIPS 실행 파일을 읽어 시뮬레이터의 내부 명령어 메모리에 적재하는 역할을 한다. 파일이 정상적으로 열리면 while 루프를 통해 파일에서 한 번에 4바이트씩 읽어온다. MIPS 명령어는 32비트이고 Big Endian 방식을 따르므로 읽어온 4개의 바이트(in[0]~in[3])를 올바른 순서로 조합하여 하나의 unsigned int 형의 instruction\_word를 생성한다.

**3-4 functions.c**

함수들을 지금과 같은 방식으로 만든 주된 이유는 MIPS 파이프라인의 각 단계별 동작과 CPU의 주요 기능을 분리하고 파이프라인 래치를 통해 다음 단계와 정보를 주고받음으로써 전체 파이프라인 시스템이 완활하게 움직이도록 설계했다.

**3-4-1 control unit**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

control 함수는 명령어 종류에 따라 제어 신호를 만드는 곳이다. 함수 시작 시 memset으로 모든 제어 신호를 0으로 초기화하는 건 중요하다. 이렇게 하면 이전에 설정된 값이 남아있는 걸 방지하고 현재 명령어에 필요한 신호만 확실하게 켤 수 있다. opcode에 따라 R-타입과 I/J-타입을 크게 나누고 그 안에서 다시 세부 명령어별로 필요한 제어 비트들을 켜는 방식은 MIPS 제어 유닛 설계를 그대로 따른 것이다.

**3-4-2 ALU**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

ALU 함수는 ALUcontrol에서 받은 연산 코드에 따라 실제 산술/논리 연산을 수행한다. R-타입과 I-타입을 나눠서 switch 문을 두 번 쓰는 건 명령어 종류에 따라 연산 결정 방식이 다르기 때문이다. 각 case는 MIPS의 다양한 연산을 구현하고 Beq, bne 명령어의 경우 비교 결과를 becond 전역 변수에 저장하고 branchinsts를 증가시키는 건 EX 단계에서 분기 조건 판단과 통계 수집을 함께 처리하려는 것이다. 연산 결과는 mem\_latch[0].aluResult에 저장해서 다음 MEM 단계로 넘긴다.

**3-4-3 IF 단계**

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

먼저 fetch 함수는 명령어 인출 단계인데 여기서 가장 먼저 pc가 -1인지 확인한다. 이건 프로그램이 이미 종료 상태면 더 이상 아무 작업도 하지 않도록 하는 깔끔한 처리 방식이다. 그리고 메모리 주소 범위를 벗어났는지도 꼼꼼히 따진다. 잘못된 주소에서 명령어를 읽으려 하면 바로 pc를 -1로 만들고 현재 래치를 무효화해서 오류를 전파시키지 않으려는 의도이다. 다양한 분기 예측기를 #if defined로 나눠둔 건 테스트와 수정을 쉽게 하려고 만들었다. 마지막으로 예측된 next\_pc로 pc를 바로 업데이트하고 현재 IF/ID 래치를 유효하다고 표시한다. 이렇게 해야 다음 ID 단계로 끊김 없이 정보를 넘길 수 있다.

**3-4-3 ID 단계**

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

decode 함수는 명령어를 각 필드로 나누는 건 MIPS 명령어 형식을 그대로 따른 것이다. j나 jal 명령어의 경우 절대 주소를 여기서 미리 계산해두면 EX 단계에서 바로 쓸 수 있어서 효율적이다. control 함수와 reg 함수를 호출해서 필요한 제어 신호와 레지스터 값을 준비하고 imm은 id\_ex\_latch[0].ex.zero 제어 신호에 따라 부호 확장 또는 제로 확장을 선택적으로 수행한다.

**3-4-4 EX 단계**

execute 함수는 가장 복잡한 부분이다. 데이터 포워딩 로직이 여기에 집중되어 있다

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

rs\_ex와 rt\_ex가 필요로 하는 값을 EX/MEM 래치나 MEM/WB 래치에서 가져오도록 조건문을 구성했다. EX/MEM 래치를 MEM/WB 래치보다 우선적으로 확인하는 건 가장 최신 값을 사용하려는 파이프라인 포워딩의 일반적인 원칙을 따른 것이다. rt\_needs\_forwarding 조건으로 rt가 실제로 사용되는 경우에만 포워딩을 시도하는 것도 효율적이라고 생각한다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

점프 계열 명령어는 ALU 연산 전에 먼저 처리해서 불필요한 계산을 피하고 즉시 PC를 변경하고 파이프라인을 flush한다. 이는 제어 해저드를 최대한 빨리 해결하려는 설계이다. j 명령어의 경우 다음 래치를 무효화해서 아무것도 쓰지 않도록 하고 jal은 $ra 저장을 위해 래치를 유효하게 유지하는 차이도 정확히 구현했다. 일반 ALU 연산 시에는 ALUSrc와 zero 제어 신호에 따라 두 번째 피연산자를 결정하고 ALU 함수를 호출한다. 분기 명령어는 ALU 연산 후에 타겟 주소를 계산하고 분기 예측 검사 함수를 호출한다. 여기서 분기 예측이 틀렸다면 PC 수정과 파이프라인 플러시가 일어날 것이다. 이 모든 정보를 EX/MEM 래치에 담는 것이 이 단계의 역할이다.

**3-4-5 mem 보조**

텍스트, 스크린샷, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

mem 함수는 memaccess 함수의 실제 메모리 작업을 도와주는 역할이다. EX/MEM 래치의 유효성을 확인하고 memoryWrite나 memoryRead 신호에 따라 메모리 쓰기 또는 읽기를 수행한다. 메모리 연산 횟수도 여기서 센다.

**3-4-6 MEM 단계**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

mem 함수를 호출해서 메모리 작업을 시키고 EX/MEM 래치의 다른 값들을 MEM/WB 래치로 그대로 넘겨준다. 역할 분담을 위해 이렇게 나눴다.

**3-4-7 WB 단계**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

wb 함수는 파이프라인의 마지막 단계로 레지스터에 최종 결과를 쓴다. MEM/WB 래치가 유효하고 RegWrite 신호가 켜져 있을 때만 그리고 목적지가 $zero가 아닐 때만 실제 쓰기를 수행한다. j\_and\_l\_toReg (jal의 경우) memtoReg (lw의 경우) 신호에 따라 $ra에 PC+4를 쓰거나 메모리에서 읽은 값을 쓰거나 ALU 결과를 쓰는 등 MIPS 명령어의 쓰기 동작을 정확히 구분해서 처리한다.

**3-4-8 latch update**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

마지막으로 Latch\_Update함수는 각 클록 사이클의 끝에서 모든 래치의 내용을 다음 단계로 넘기는 역할을 한다. latch[0](현재 단계의 출력)을 latch[1]으로 복사하고 latch[0]의 valid 비트를 0으로 만들어 다음 사이클에 새로운 값이 쓰일 수 있도록 준비한다. 이는 파이프라인이 한 사이클씩 정확하게 진행되도록 하는 동기화 로직이다.

**3-5 branch\_control.c**

다양한 분기 예측기들이 실제로 예측한 결과와 실제 분기 실행 결과를 비교하고 그에 따라 후속 조치를 어떻게 할지 결정하려고 이렇게 나눴다.

**3-5-1 AlwaysNotTaken**

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

AlwaysNotTaken 함수는 가장 단순하다. 이 예측기는 항상 분기가 일어나지 않는다고 가정한다. 그래서 실제 분기가 일어나지 않으면 예측 성공이다. 반대로 실제 분기가 일어나면 예측 실패이므로 파이프라인 앞부분을 Flush\_IFID() Flush\_IDEX()로 비우고 pc를 실제 분기 주소 b\_address로 바꿔준다. 그리고 mis\_predict를 증가시킨다. 이처럼 단순한 예측기는 로직도 간단 명료하다.

**3-5-2 AlwaysTaken**

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

AlwaysTaken함수는 조금 더 복잡하다. BTB(Branch Target Buffer)를 확인하는 부분이 추가된다. 만약 BTB에 해당 분기 명령어의 정보가 있다면 이전에 이 분기가 Taken이었을 가능성이 높다고 보는 것이다. 그래서 BTB 히트 시 실제 분기가 일어나면(b\_taken) 예측 성공이다. 틀리면 파이프라인을 비우고 pc는 bpc + 4(Not Taken)로 수정한다. BTB에 정보가 없다면 이 분기는 처음 보거나 이전에 Not Taken이었을 수 있다. 그래서 이때 실제 분기가 일어나지 않으면 예측 성공이고 실제 분기가 일어나면 예측 실패로 처리하면서 BTB에 b\_address를 새로 기록하고 pc도 b\_address로 바꿔준다. BTB를 활용해서 더 효율적으로 해보려고 했다.

**3-5-3 BTFN**

텍스트, 스크린샷, 디스플레이, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

BTFN함수는 BTB에 있는 타겟 주소와 현재 bpc를 비교한다. 타겟 주소가 bpc보다 작으면 뒤쪽으로 점프하는 Backward 분기이므로 Taken으로 예측(predicted\_taken = 1)한다. 그 외에는 Forward 분기이거나 BTB 미스이므로 Not Taken으로 예측한다. 이렇게 예측한 predicted\_taken과 실제 b\_taken을 비교해서 성공 실패를 처리한다. 실패 시 PC 수정은 AlwaysTaken과 유사하지만 BTB 업데이트는 실제 뒤로 가는 분기가 일어났을 때만 해주는 점이 다르다.

**3-5-4 OneBitBranch**

텍스트, 스크린샷, 디스플레이이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

OneBitBranch함수는 1비트 분기 예측기의 상태를 업데이트한다. B\_Taken[bpc] 배열이 각 분기 명령어 주소의 1비트 상태(0이면 Not Taken 예측 1이면 Taken 예측)를 저장한다. BTB 히트 여부와 현재 1비트 상태에 기반한 예측과 실제 b\_taken을 비교한다. 예측이 틀리면 파이프라인 플러시 PC 수정과 함께 B\_Taken[bpc] 상태를 반대로 바꿔준다 (0->1 또는 1->0). BTB 미스 시에는 실제 분기 결과에 따라 B\_Taken[bpc]와 BTB를 초기화하거나 업데이트한다. 과거의 한 번의 결과를 기억해서 다음 예측에 반영하는 것이다.

**3-5-5 TwoBitBranch**

텍스트, 스크린샷, 폰트, 디스플레이이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트, 디자인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

TwoBitBranch함수는 2비트 포화 카운터를 사용한다. B\_Taken[bpc]가 0 (Strongly Not Taken), 1 (Weakly Not Taken), 2 (Weakly Taken), 3 (Strongly Taken) 중 하나의 상태를 가진다. 상태가 2 이상이면 Taken으로 예측한다. 예측이 실제 결과와 같으면 상태를 더 강한 쪽으로 한 단계 옮기고 (예: Weakly Taken -> Strongly Taken) 틀리면 약한 쪽으로 한 단계 옮긴다. 상태가 0 미만이나 3 초과로 가지 않도록 경계 처리도 해준다. 예측 실패 시에는 물론 플러시와 PC 수정도 한다. BTB 업데이트는 1비트 예측기와 비슷하다. 2비트 카운터는 일시적인 반대 경향에 덜 민감해서 1비트보다 일반적으로 성능이 좋은 거 같다. 그래서 이렇게 상태 천이 로직을 구현한 것이다.

**3-5-6 Gshare**

텍스트, 스크린샷, 폰트, 디스플레이이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

Gshare 방식 함수는 전역 히스토리 정보와 현재 분기 명령어 주소를 XOR한 값을 PHT(Pattern History Table 여기서는 B\_Taken 배열) 인덱스로 사용한다. 이렇게 하면 서로 다른 분기 명령어라도 과거 실행 패턴이 비슷하면 같은 PHT 엔트리를 공유하게 된다. PHT 엔트리가 2비트 카운터로 동작하는 것은 TwoBitBranch예측과 동일하다. 예측이 틀리면 PHT 상태를 업데이트하고 GHR도 현재 분기 결과를 반영해서 왼쪽 시프트 후 새 결과를 맨 오른쪽에 넣어 업데이트한다.

**3-5-7 LocalTwoLevel**

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

LocalTwoLevel함수는 지역 히스토리 기반 예측이다. BPHT[bpc]에 각 분기 명령어별 과거 실행 패턴을 저장하고 이 패턴을 PHT(B\_Taken 배열) 인덱스로 사용한다. 즉 각 분기 명령어마다 자신만의 PHT를 가지는 효과를 내려는 것이다. PHT의 2비트 카운터 동작 방식이나 예측 실패 시 처리 예측 성공 시 상태 업데이트는 Gshare과 유사하다. 다만 GHR 대신 BPHT를 업데이트하는 점이 다르다. 각 분기의 고유한 패턴을 학습하는 데 중점을 뒀다.

모든 함수에서 예측 실패 시 Flush\_IFID()와 Flush\_IDEX()를 호출하고 pc를 수정하며 mis\_predict를 증가시킨다. 예측 성공 시에는 predict\_correction을 증가시킨다. 이는 파이프라인 제어와 성능 측정을 위한 공통적인 로직이다. 이처럼 각 분기 예측기의 특성에 맞춰 상태 업데이트와 BTB 관리 로직을 다르게 구현하면서도 공통적인 후속 처리는 일관되게 유지하는 방식으로 코드를 작성했다.

**3-6 hazard\_handling.c**

텍스트, 스크린샷, 소프트웨어, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

파이프라인에서 앞선 명령어가 아직 계산 중인 값을 바로 다음 명령어가 사용하려고 하면 데이터 해저드가 발생한다. 이 때문에 파이프라인을 멈춰야 할 수도 있는데 이는 성능 저하로 이어진다. 그래서 포워딩 방식을 채택했다. 아직 레지스터 파일에 쓰이지 않은 결과값이라도 ALU나 메모리 접근 단계에서 바로 가져와 다음 명령어의 실행 단계에서 사용할 수 있도록 경로를 만들어주는 것이다. 이렇게 두 함수로 나누어 각 소스 레지스터(rs, rt)에 대한 포워딩 가능성을 독립적으로 검사하도록 구현했다.

텍스트, 스크린샷, 디스플레이, 폰트이(가) 표시된 사진

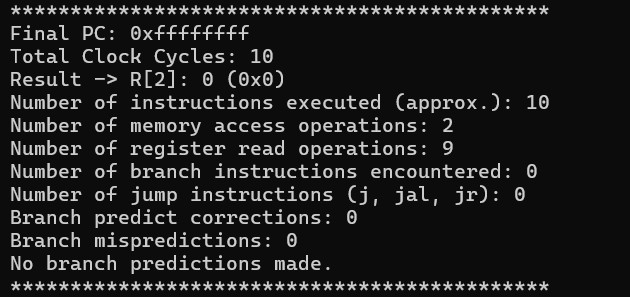
AI가 생성한 콘텐츠는 부정확할 수 있습니다.

제어 해저드는 분기나 점프 명령어 때문에 발생한다. 예측이 틀리거나 무조건 점프를 해야 할 때 이미 파이프라인에 들어와 있는 잘못된 명령어들을 제거해야 한다. 이 과정을 flush 라고 하며 이 함수들이 그 역할을 한다. 3개의 플러시 함수를 구현한 이유는 제어 해저드의 종류에 따라 파이프라인을 최소한으로 비우면서도 정확한 실행 흐름을 유지하기 위함이다.

4. Result

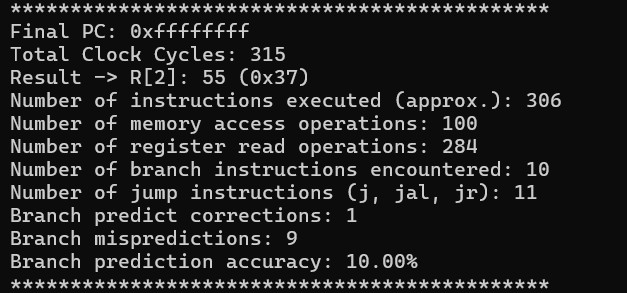
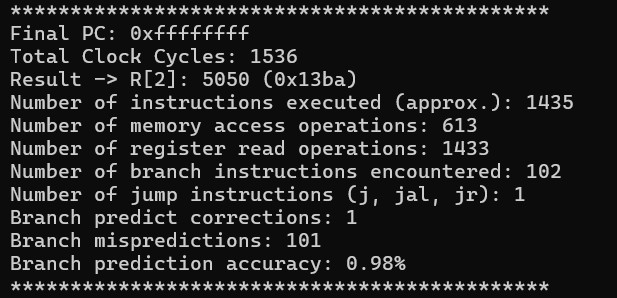
gcc main.c globals.c functions.c hazard\_handling.c branch\_control.c prediction.c -DPREDICTOR\_ALWAYS\_NOT\_TAKEN -o mips\_sim\_ant 등등 전처리기 지시문을 사용하여 조건부 컴파일을 해서 각 예측기별로 출력을 했다.

- **AlwaysNotTaken**

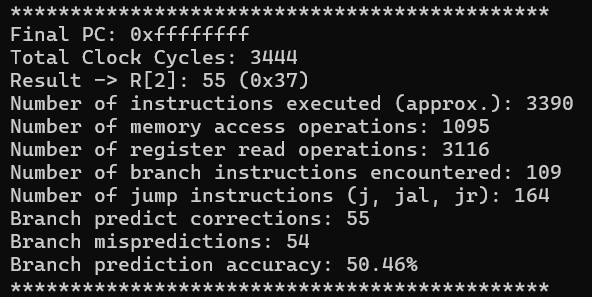
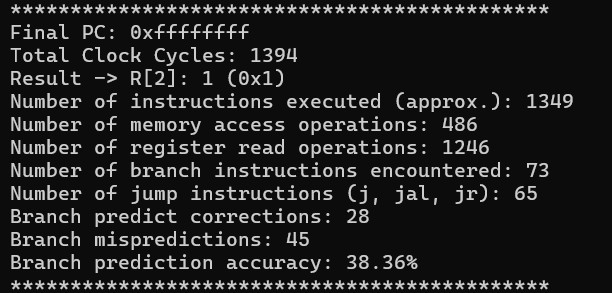
텍스트, 스크린샷, 폰트, 흑백이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

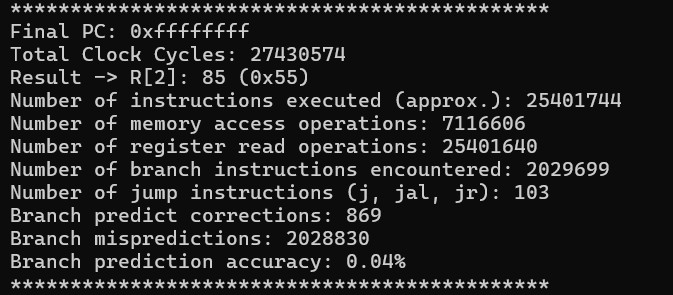
simple.bin simple2.bin



simple3.bin simple4.bin

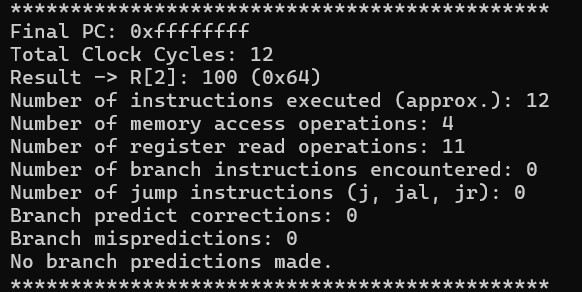
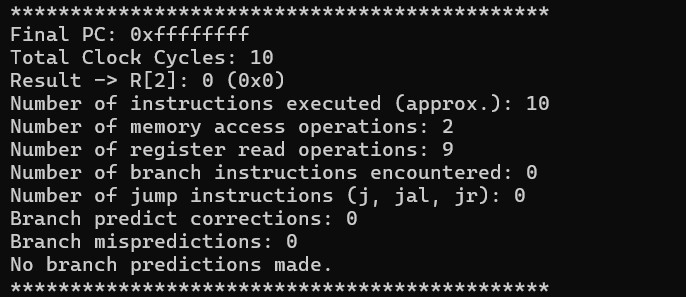


gcd.bin fib.bin

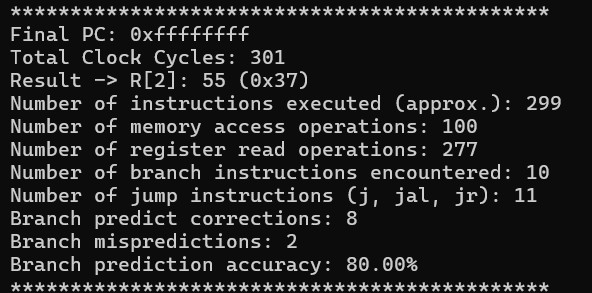
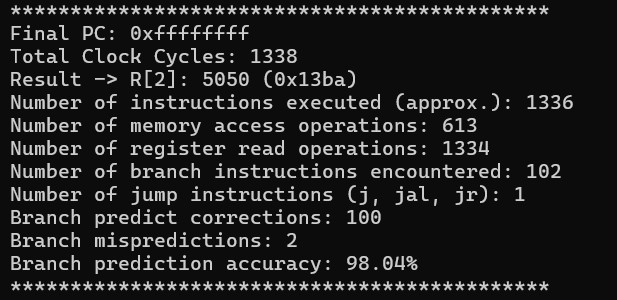


input4.bin

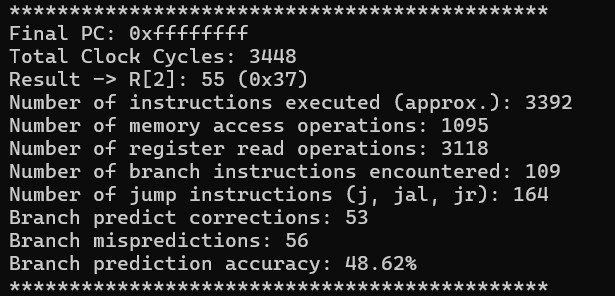
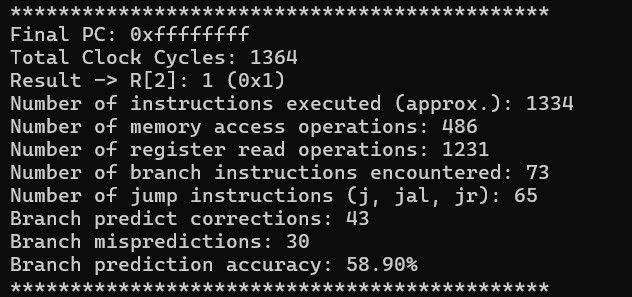
- **Always Taken**



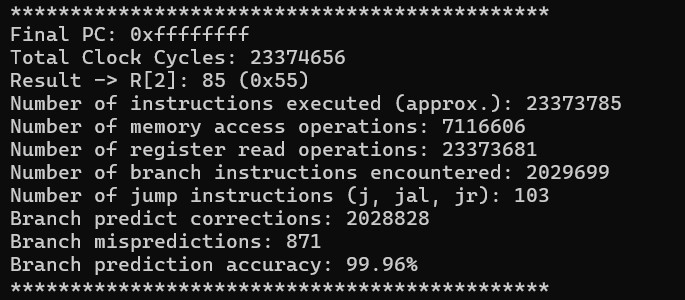
simple.bin simple2.bin



simple3.bin simple4.bin

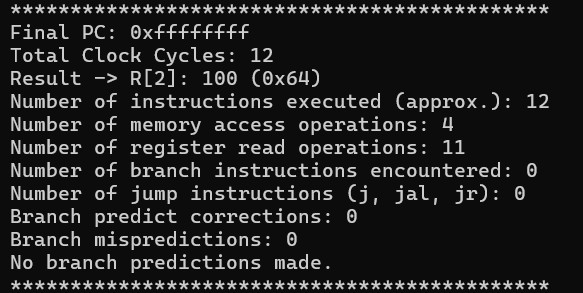
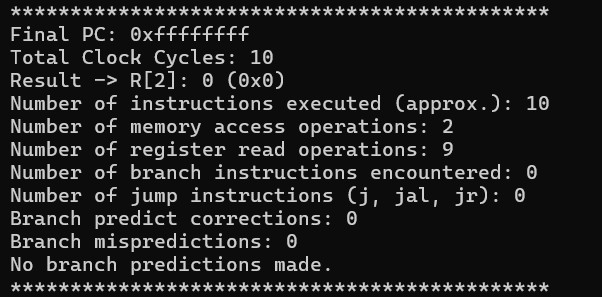


gcd.bin fib.bin

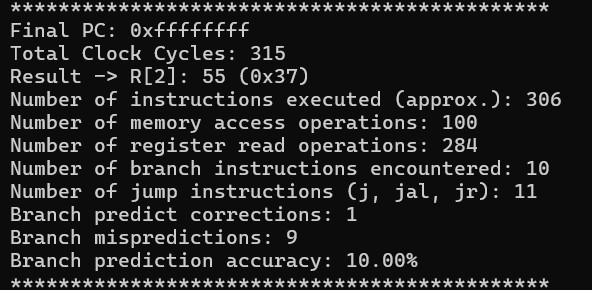
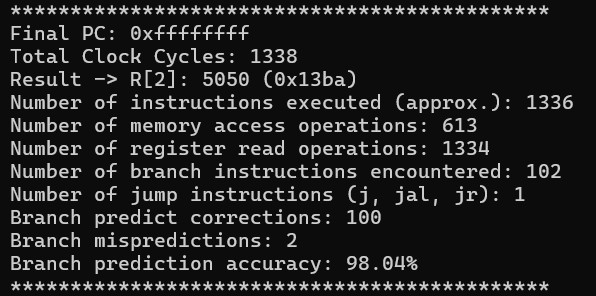


input4.bin

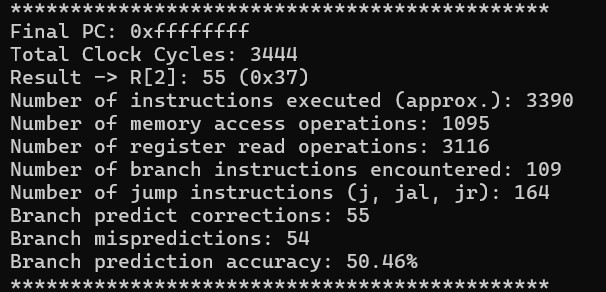
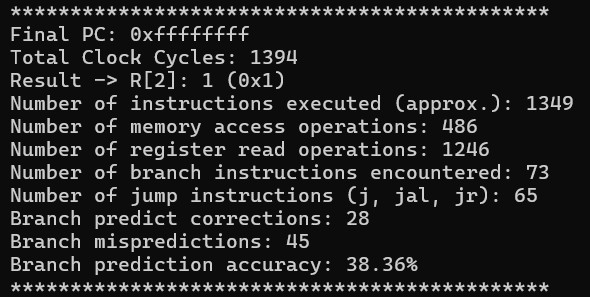
- **BTFN**



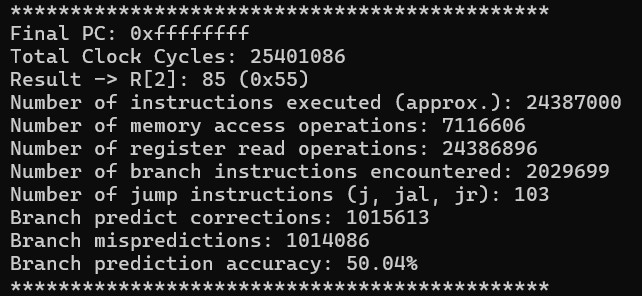
simple.bin simple2.bin



simple3.bin simple4.bin

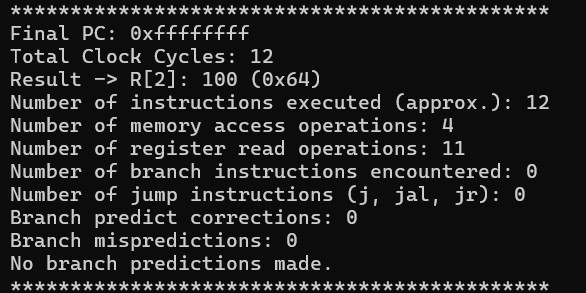
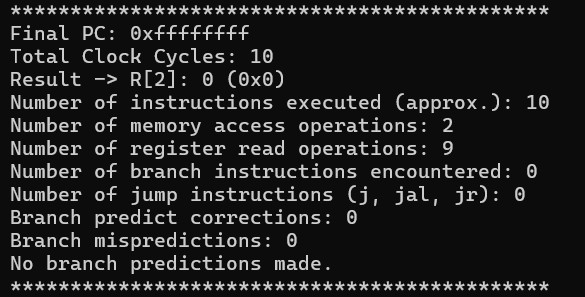


gcd.bin fib.bin

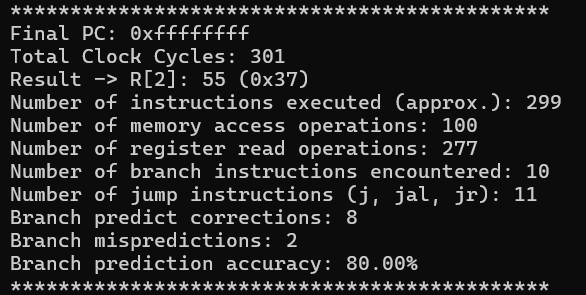
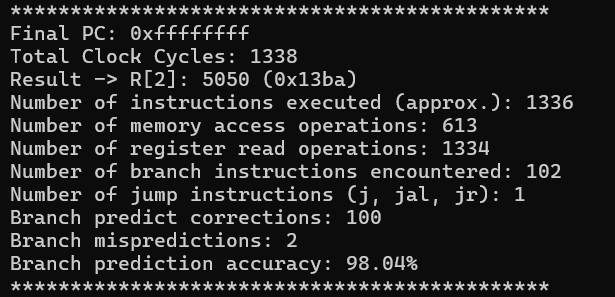


input4.bin

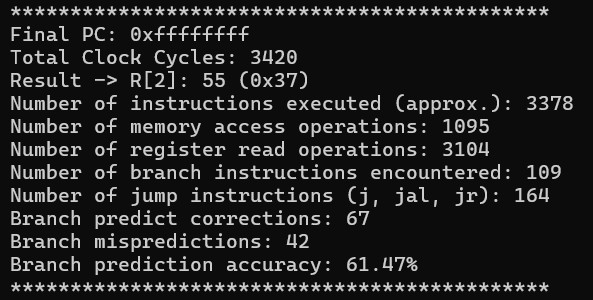
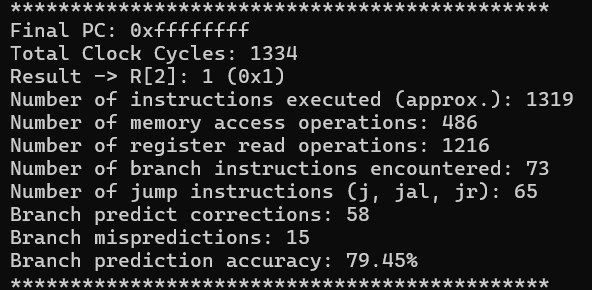
- **One bit**



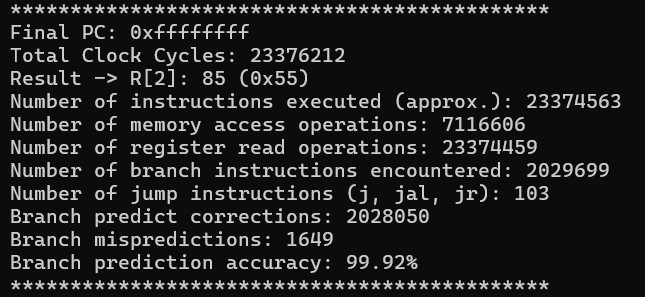
simple.bin simple2.bin



simple3.bin simple4.bin

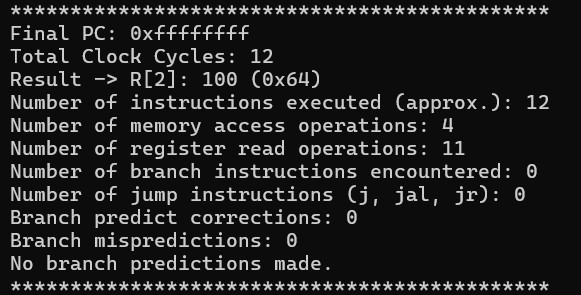
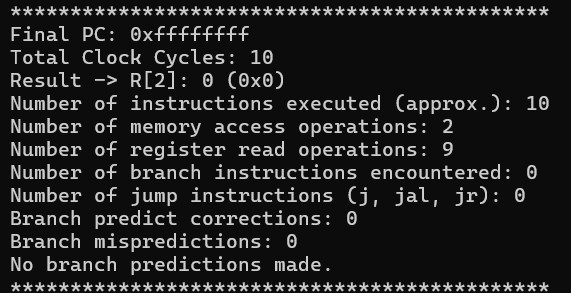


gcd.bin fib.bin

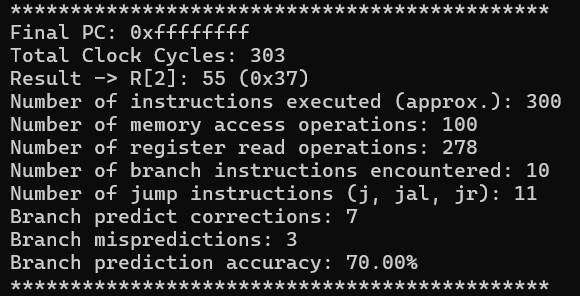
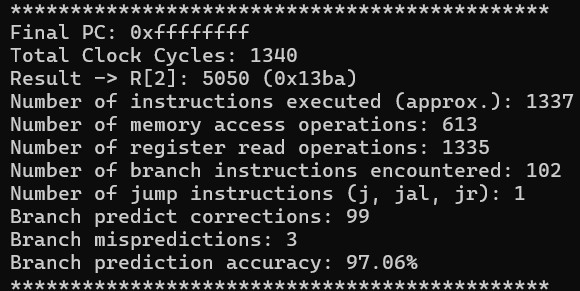


input4.bin

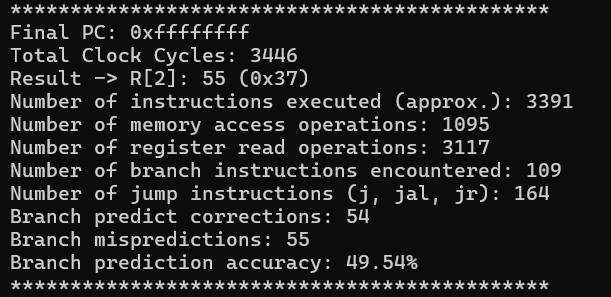
- **Two bit**



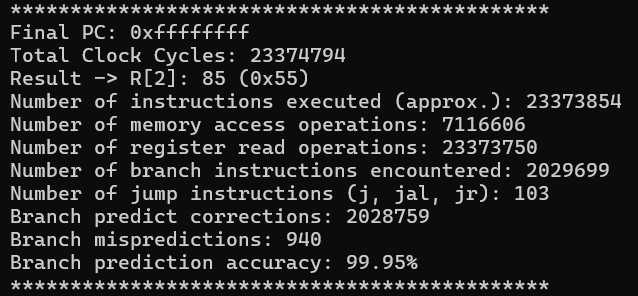
simple.bin simple2.bin



simple3.bin simple4.bin

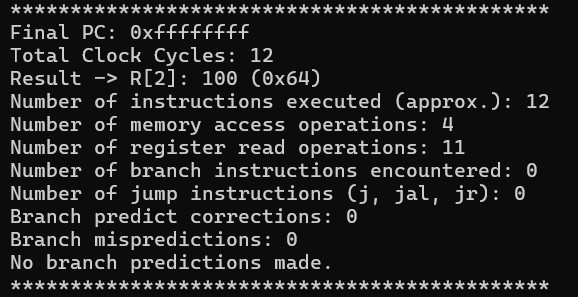
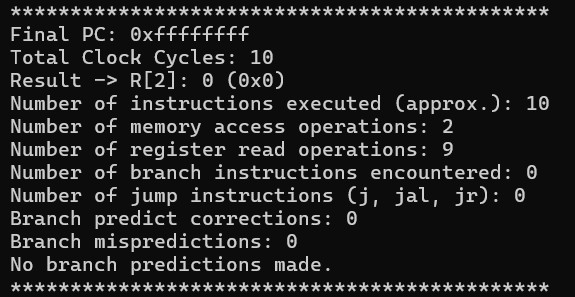


gcd.bin fib.bin

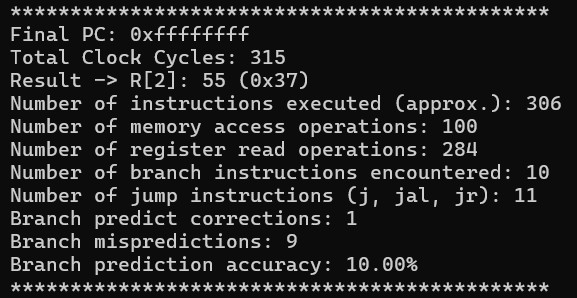
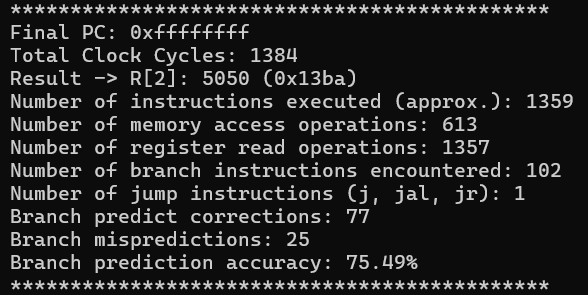


input4.bin

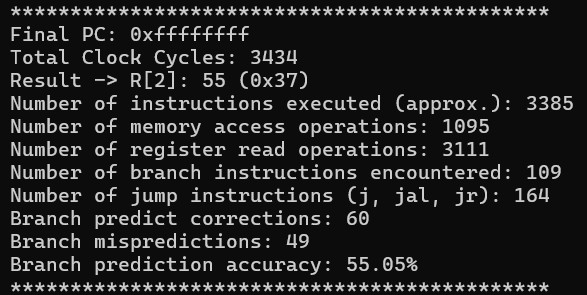
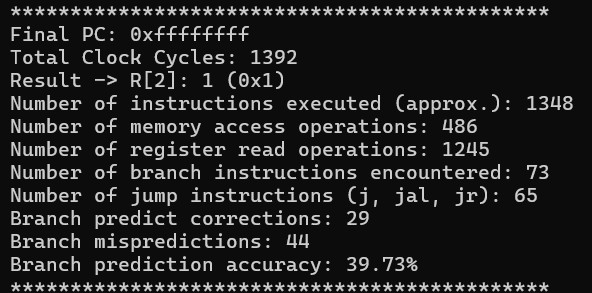
- **Gshare**



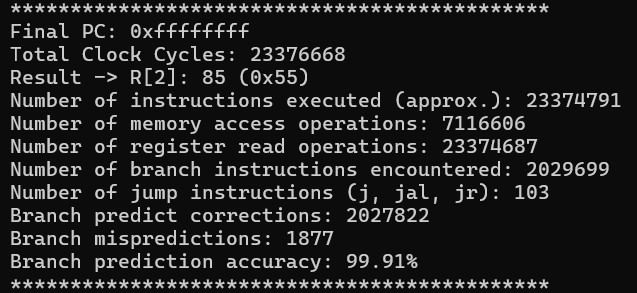
simple.bin simple2.bin



simple3.bin simple4.bin



gcd.bin fib.bin



input4.bin

- **Local Two Level**

텍스트, 스크린샷, 폰트, 흑백이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 폰트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

simple.bin simple2.bin

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

simple3.bin simple4.bin

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

gcd.bin fib.bin

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

input4.bin

5. Analysis

- simple3.bin

- simple4.bin

- gcd.bin

- fib.bin

- input4.bin

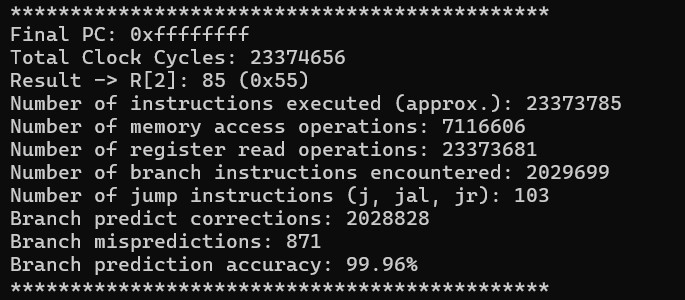
simple, simple2 없는 이유는 프로그램을 실행 중 조건 분기 명령어가 한 번도 실행되지 않기 때문이다. 위의 그래프들을 분석해보자면 Always Not Taken에서는 항상 Not Taken으로 예측하기 때문에 프로그램에서의 대부분의 조건 분기가 Not Taken이면 정확도가 높다. Always Taken은 위와 반대로 항상 Taken으로 예측해서 대부부분 Taken이면 정확도가 높다. BTFN은 뒤로 가는 주소로의 분기(주로 루프의 끝에서 시작으로 돌아가는 경우)는 Taken, 앞으로 가는 주소로의 분기(주로 if 조건문)는 Not Taken으로 예측해서 이 구조랑 비슷한 프로그램이라면 정확도가 높다. 여기까지는 정적 예측으로 일반적인 구조나 가정에 의해서 정해지므로 프로그램이랑 부합하는 정도에 따라서 성능이 바뀐다.

동적 예측기인 One-bit는 각 분기마다 최근 한 번의 실행 결과(Taken/Not Taken)를 기억하여 다음을 예측해서 틀리면 예측 방향을 바꾸기 때문에 예측이 계속해서 틀리다면 fib.bin 처럼 재귀패턴에 민감해서 정확도가 높지 않을 수 있다. Two-bit는 각 분기마다 2비트 포화 카운터 4가지 상태를 사용한다. 한두 번의 다른 결과에도 예측을 쉽게 바꾸지 않기 때문에 One-bit 보다는 안정적일수 있지만 그래프에서는 조금 더 낫거나 낮은 모습을 보여준다. Gshare는 전역 분기 이력과 현재 분기 명령어의 주소를 XOR하여 패턴 히스토리 테이블(PHT)의 인덱스를 만들어서 학습한다. input4, fib 같은 프로그램에서 상대적으로 강점을 보인다. Local Two-Level은 각 분기 명령어 주소마다 별도의 지역 분기 이력을 BHT에 저장하고 이 지역 이력을 패턴 히스토리 테이블(PHT, 2비트 카운터 배열)의 인덱스로 사용해서 학습한다. 이러한 지역적 패턴을 잘 포착하여 그래프를 보면 Gshare와 비슷하거나 높은 정확도를 가진다. 이와 같은 동적 예측기는 실행 중 학습을 통해 적응하므로 특히 일관되거나 반복적인 패턴을 가진 분기에 대해 높은 정확도를 보이는 경향이 있다.

- single & pipeline comparison

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.



지난번 과제였던 싱글 사이클의 input4하고 input4에서 정확도가 제일 높았던 always taken 파이프라인을 input4.bin을 실행한 결과를 비교해보면 사이클 수는 23372706 사이클과 파이프라인 23374656 사이클로 순수 사이클로만 보면 싱글 사이클이 미세하게 앞선다. 왜냐하면 파이프라인이 초기화하고 flush 하는 오버헤드가 있기 때문에 좀더 많다. 하지만 실제 실행시간에서 보면 싱글 사이클은 모든 연산이 한 사이클에 처리되기 때문에 파이프라인과 비교했을 때 시간 차이가 많이 난다. 예를 들어 위의 동일한 2300만개의 명령어를 처리할 때 싱글은 긴 클록 주기로 대략 10ns 파이프라인은 짧은 클록이기 때문에 2ns 된다고 치면 싱글은 23300000 x 10ns = 233.0ms이고 파이프라인은 23300000 x 2ns = 46.8ms로 파이프라인이 싱글 사이클 대비 대략 5배 정도 파이프라인이 빠른 처리속도를 가진다.

6. Feelings

데이터 의존성과 제어 의존성이 파이프라인의 흐름을 어떻게 방해하고 이를 해결하기 위한 forwarding, flush와 같은 메커니즘이 왜 필요한지 그리고 이를 코드 상에서 어떻게 정확히 구현해야 할지 개념적으로 생각하는 데 시간이 소요되었다. 예를 들면 포워딩 유닛이 EX/MEM latch와 MEM/WB latch 중 어떤 값을 우선적으로 전달해야 하는지 또는 Load-Use 해저드 발생 시 파이프라인 버블을 어떻게 삽입하고 제어 신호를 어떻게 변경해야 하는지 등을 명확히 이해하고 코드로 옮기는 과정이 어려웠다. 그리고 디버깅을 하는데 단순히 printf만으로는 파악하기가 어려워서 GDB를 통해 이전 명령어의 원인인지 아니면 해저드 탐지 유닛의 문제인지 등 문제 파악을 했지만 쉬운 일은 아니었다. 이런 디버깅할 때 gcd.bin이 계속 이상한 숫자가 나와서 gcd (24, 32)로 비교적 작은 숫자 만들어서 계속 해본 다음 해결했다.

이번 과제를 하면서 지난번 과제에 비해 구현 난이도가 급격하게 상승해서 너무 어려워서 포기할 뻔했지만 여기에 쏟은 시간이 아깝고 오기가 생겨서 끝까지 계속 했던거 같다. 이번 과제에 시간도 많이 들인 만큼 직접 구현하고 구현하기 전에 생각하는 습관도 들어서 많이 배운 거 같다.