MIPS Pipeline Microprocessor Implementation Report

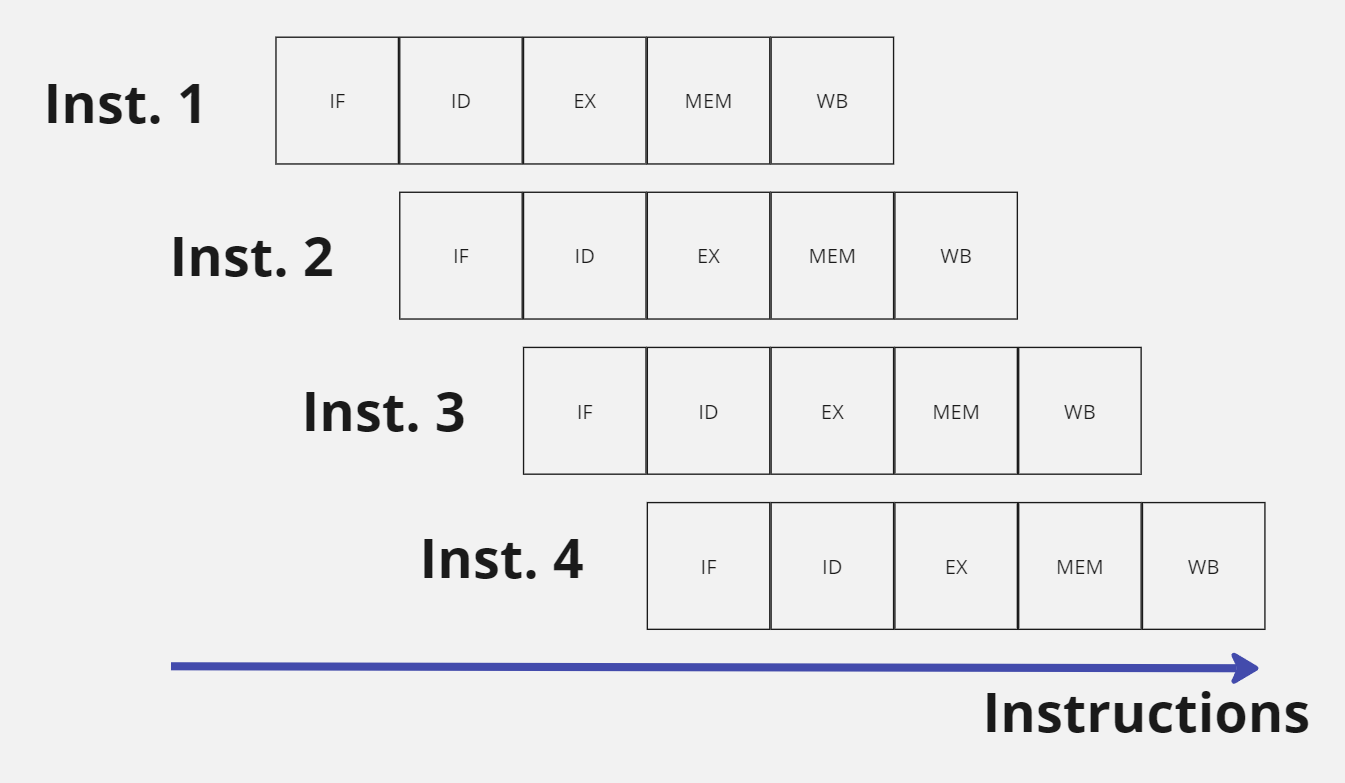
차 호 현(#32224560), [outcider112@dankook.ac.kr](mailto:outcider112@dankook.ac.kr)[[1]](#footnote-1)  
Undergraduate Student in Mobile System Engineering, Dankook University

Code Repository:

**1. Introduction**

기존 Single Cycle 구조는 하나의 HW Clock에 맞추어 동작한다는 점에서 매우 간단한 마이크로 아키텍처 구조를 형성한다. 하지만, Single Cycle 구조는 하나의 클럭에 모든 컴포넌트가 타이밍을 맞추어 동작해야 한다는 한계점 때문에, CPU의 성능을 높이려 클럭의 주파수를 높일 수 없다. 이러한 관점에서 등장한 것이 Multi-Cycle Architecture이며, Single Cycle의 각 단계를 분할하여, 분할된 모든 단계가 동시에 처리될 수 있도록 한 구조가 Pipeline 구조이다, (이하 파이프라인).

파이프라인은 각각 명령어 인출(IF), 명령어 해독(ID), 명령어 실행(EX), 메모리 접근(MEM), 결과값 저장(WB) 단계로 구분한다. 각각의 단계는 하나의 명령어가 실행될 수 있다. 파이프라인을 통해, CPU의 클럭 주파수를 높일 수 있으며, 이를 통해서 CPU의 Throughput을 향상시킬 수 있다. 하지만, Single Cycle과 다르게, 한 마이크로프로세서 내의 여러 명령어의 상태가 존재하기 때문에, 의도하지 않은 데이터의 출력 혹은, 의도하지 않은 명령어의 처리가 발생할 수 있다. 이를 의존성 혹은 해저드라고 표현하며, 파이프라인 구조는 이러한 의존성 혹은 해저드를 해결하기 위해 다양한 처리 기법이 존재한다.



이 보고서에서는 MIPS Single Cycle Architecture에서, Piplelined Architecture로 발전시키기 위한 기본적인 구조를 설명하며, 파이프라인이 정상적으로 구현할 수 있도록 의존성과 해저드를 해결하는 기법을 포함하여 설명 및 구현한다.

**2. Concept of the Pipeline and Ideal Performance Improvement**

**2-1. Pipeline**

파이프라인은 Single Cycle Architecture와 Multi Cycle Achitecture와 다르게, 명령어의 병렬처리를 통해서 성능을 높힌다. 텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

Single Cycle은 모든 조합 회로에 동일한 클럭을 사용하게 되는데, 각 부품 별로 보장되어야 하는 시간으로 인하여 클럭 주파수를 높이는데 어려움이 있다. 이러한 문제점은 Single Cycle Architecture에서 클럭을 최소로 낮출 수 있는 한도가, 가장 처리에 오래 걸리는 Data Path에 따라 결정되며, 이는 성능의 악영향을 미치게 된다. 이를 개선하기 위해서 등장한 것이 Mutli-Cycle Architecture이며, Single Cycle에서 각 단계 별로 클럭에 맞춰 동작하도록 한다.

Pipeline Architectue(이하 파이프라인)은 Mutli-cycle에서 더 나아가, 각 단계 별로 명령어를 처리하도록 하여, 동일한 시간 동안 파이프라인이 다른 Architecture보다 더욱 많은 처리량이 발생한다.

**2-2. Dependency (Hazard)**

파이프라인은 Single Cycle Architecture와 다르게, 마이크로프로세서 내에 여러 명령어가 존재한다. 그에 따라서 명령어 간의 State가 상충될 수 있으며, 혹은 의도하지 않은 State에서 명령어 처리가 발생할 수 있다. 이러한 State 간의 충돌을 Dependency라고 하며, Dependency는 파이프라인 성능의 악영향 혹은 의도하지 않은 결과를 초래한다.

\*이 보고서 내에서는 hazard라고 표현하지 않고, dependency라고 표현한다.

**2-2-1 Data Dependency**

Data dependency는 데이터에 접근하고자 하는 대상이 SW 흐름 상 의도하지 않은 상태에서 처리가 되는 것을 의미한다. 이러한 dependency가 발생하는 이유는 파이프라인의 특징인 동시에 여러 명령어 처리하는 것이다.

다음 유형은 파이프라인에서 발생할 수 있는 dependency 유형이다.

**Read-after-Write(RAW)**



데이터를 작성하고 나서, 데이터를 읽는 작업을 수행하는 경우 발생하는 dependency이다. 파이프라인 구조에서는, ID 단계에서 레지스터 파일로부터 데이터를 읽어 들여야 하지만, 앞선 명령어의 결과가 작성되지 않음에 따라서 위 r3 레지스터가 첫번째 명령어가 실행되기 이전의 값으로 두번째 명령어에서 참조하게 된다.

**Write-after-Write(WAW)**

****

r3의 값을 두 번 덮어 쓰는 경우이다.

**Write-After-Read(WAR)**



r1을 읽는 명령어와 r1을 쓰는 명령어가 뒤 따라 나오는 경우이다.

War과 WAW의 경우, 프로그램의 순차적인 처리하고 한 단계에서 데이터를 저장하게끔 함에 따라서 해결할 수 있다. 즉, WB 단계에서 모든 데이터 저장이 발생하기 때문에 해결될 수 있다.

하지만, RAW의 경우, 의도했던 데이터가 앞선 명령어에 존재하기 때문에 다른 해법을 찾아야 한다. 이에, Forwarding과 Scoreboarding, Stalling 등과 같은 기법이 존재한다.

Stalling은 해당 레지스터가 사용할 수 있을 때까지 데이터를 기다리는 것을 의미한다. 이 때, 데이터를 기다리기 위해서는 EX 단계에서 데이터 처리가 발생하지 않고, EX 이후 단계가 처리되어 데이터가 입력되도록 파이프라인 내에 Bubble을 주입한다.

도표, 평면도, 스케치, 기술 도면이(가) 표시된 사진

자동 생성된 설명

이를 통해서 레지스터가 사용가능할 때까지, 파이프라인을 대기하여, RAW 의존성을 해결할 수 있다. 이때, 파이프라인은 조합회로를 통해서 ID 단계 이후에 의존성을 가지고 있는지 확인한다.

텍스트, 폰트, 타이포그래피이(가) 표시된 사진

자동 생성된 설명

**2-2-2 Score Boarding**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

Score Boarding은 레지스터에 추가적인 비트 데이터를 저장한다. 해당 비트는 레지스터가 다른 명령어에 의해서 접근되었는지 의미하며, valid 비트에 따라서 파이프라인에 Stall을 발생하도록 한다. 앞서 Stalling과 다르게 Scoreboarding의 경우, RAW dependency와 다르게 WAW와 WAR dependency에도 Stall이 발생한다.

**2-1-3 forwarding**

Forwarding은 EX보다 앞선 단계에서 계산된 값을 직접 조합회로를 통해 가져오는 것을 의미한다. 이는 조합회로를 통해서 즉시 데이터를 가지고 오기 때문에 파이프라인 Stalling 없이 처리할 수 있다는 장점이 존재한다.

텍스트, 도표, 라인, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Data dependency를 가지고 있는 명령어 사이의 거리는 최대 2라는 것에 착안하여, EX 단계에서 앞선 MEM단계와 WB 단계의 결과 값을 가지고 온다.

Data forwarding을 수행하는 경우, R/I 타입 명령어와 LW 명령어에 의해서 앞선 단계에서 데이터 의존성이 발생한다. 이중 LW 명령어의 경우, 데이터를 메모리로부터 로드 한 후에 데이터를 읽어올 필요가 있다. 따라서 LW 명령어 뒤에는 인위적으로 1 클럭을 Stall 혹은 nop 명령어를 삽입한다. 이 경우, mips 컴파일러가 LW 명령어 이후에 data dependency가 발생할 경우, nop 명령어를 삽입한다.

**2-3 Control Dependency**

파이프라인 내에 분기 명령어는 EX 단계에서 처리된다. 특히, Branch 명령어의 경우 EX 단계에서 점프할 주소가 정해져야 한다. 이 경우, EX 단계까지 IF와 ID 단계는 Branch 여부에 따라서 의도하지 않은 명령어 처리가 발생하게 될 수 있다. 이렇게 PC가 프로그램이 의도한 대로 명령어를 인출하지 못하여 올바른 프로그램을 처리하지 못하게 되며, 이러한 문제는 control dependendcy라고 한다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

MIPS Architecture에서 분기 흐름을 제어하는 명령어는 위 5개가 존재한다. 이 중, Conditional(BXX)와 Unconditional(j) Call(jal)의 경우, 흐름을 예측할 수 있다는 특징이 있다. 이러한 control dependency으로 인하여 의도하지 않은 architectural state의 변화가 발생하지 않도록 latch의 invalidation을 수행한다. 이 경우 파이프라인의 2 클럭이 낭비됨에 따라 성능이 저하된다. .

**2-3-1 Stalling**

텍스트, 도표, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Stalling은 data dependency와 마찬가지로 다음 PC 값을 알 수 있을 때까지 파이프라인의 동작을 정지한다. 이는 파이프라인 성능의 악영향이 동일하게 발생할 수 있다.

**2-3-2 branch prediction**

파이프라인에서, IF 단계 명령어를 인출할 때, 다음 명령어를 예측하여 PC 값을 예측한 값으로 변경한다. 분기 예측을 통해서 파이프라인에서 분기 시 발생하는 Invalidation으로 인한 2 클럭 낭비를 줄일 수 있다.

도표, 라인, 텍스트, 평면도이(가) 표시된 사진

자동 생성된 설명

분기 예측을 수행하는 경우, Static 하게 분기 예측을 수행하는 경우와 Dynamic하게 분기 예측하는 경우가 존재한다.

Static Branch Prediction의 경우, 항상 분기를 하지 않는 경우, BTFN 규칙에 따르는 경우, 등이 존재한다. 항상 분기하지 않는 경우로 예측하면, 정확성이 ~30-40%의 낮은 확률을 가지고 있으며, 특정한 방향으로 분기하는지 확인하지 않는다. BTFN의 경우, Branch 방향에 따라 분기 주소를 예측한다. Loop와 같이 특정 횟수 혹은 조건을 만족할 때까지 반복하는 경우 항상 분기를, 이외의 경우에는 분기하지 않는 것으로 예측한다. 이 프로젝트 내에서는 항상 분기하지 않는 경우에 대해 구현하였다.

Dynamic Branch Prediction의 경우, 과거 분기 history에 기반하여 분기를 예측한다. 이 경우, history를 저장하기 위해서 BTB라는 장치가 포함된다. IF 단계에서 BTB에 현재 인출할 PC 주소의 history가 존재하는지 확인하고 분기 여부를 저장하는 비트에 따라서 다음 PC 값을 동적으로 업데이트한다. 이 프로젝트 내에서는 Last time 1-bit prediction과 2-bit prediction을 다룬다.

**2-3-4 predicated execution**



Predictation execution은 명령어 자체에 조건문을 결합하여 사용하는 형태로, 특정 레지스터의 값이 존재할 때 명령어를 실행할 수 있도록 한다. 전체 프로그램에서 branch가 적게 발생, 즉 PC 값의 변화가 자주 발생하지 않는다는 점에서, 예측이 쉽고 stall을 적게 발생할 수 있다. 하지만, 이는 MIPS 내에 구현되지 않고, ISA 자체에서 지원해야 하며, 첫번째 조건이 거짓이더라도 다른 명령어 모두 조건을 평가해야 하는 문제가 존재한다.

**3. Design Structure of Pipeline Architecture**

파이프라인을 구현하기 위해, 우선, 컴포넌트가 각 단계 별로 구분되어야 할 필요가 있다. 파이프라인은 IF/ID/EX/MEM/WB 단계가 존재하며, 각 단계는 하나의 명령어의 처리하기 위한 조합/순차회로가 동작하고 있다. 싱글 사이클 아키텍처에서는 한 클럭에 모든 부품이 하나의 명령어를 처리할 수 있도록 동작하기 때문에, 여러 명령어를 여러 단계에 걸쳐서 한 번에 실행하기 위해서는, 각 단계가 구분 및 격리되어야 할 필요가 있다. 이를 위해서 각 단계 별로 어떠한 동작이 발생할 것인지-해당 동작에 따른 어떠한 조합회로 및 부품(이하 컴포넌트)가 필요한지 확인할 필요가 있다.

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

우선 각 단계에 필요한 동작을 구분하면 다음 표와 같다.

|  |  |  |
| --- | --- | --- |
| 단계 | 요구되는 동작 | 동작에 필요한 조합/순차 회로 |
| IF (명령어 인출) | 메모리로부터 명령어 인출 | PC, adder, MEM |
| ID (명령어 해독) | 명령어 해독 후 제어 신호 및 레지스터로부터 데이터 인출 | Control Unit, Register, Sign\_extender., MUX, Etc., |
| EX (명령어 실행) | 준비된 데이터를 통해 ALU로 연산 수행, PC 값 업데이트 | MUX, ALU, adder, etc., … |
| MEM (메모리 접근) | 메모리에 접근하여 I/O 수행 | MEM |
| WB (결과값 저장) | 레지스터에 데이터 저장 | Register, MUX, etc., … |

여기서 Single Cycle 구조에서 주의 깊게 관찰할 부분은, WB 단계에서 Memory로부터 instruction을 해독한다면, Writeback을 수행하기 전에 데이터를 저장할 Index 정보가 이후에 인출된 명령어에 의해서 전달하고자 한 데이터를 올바른 주소에 저장할 수 없다. 또한 한 명령어에는 Unique한 Control Signal이 생성되므로, 이 Signal이 보존되어야 의도한 마이크로프로세서의 동작을 수행할 수 있다.

따라서, 이러한 신호를 독립적으로 구분하기 위해서, Latch라는 데이터 저장 장치를 도입한다. Latch는 마이크로아키텍처의 Data path 중간에 포함되어, micro-architectural state를 유지할 수 있도록 한다. 따라서 latch는 다음 Figure와 같이 각 단계 별로 필요한 조합회로를 포함하며, 각 단계에서 발생한 Micro-architectural state를 저장한다.

텍스트, 도표, 평면도, 평행이(가) 표시된 사진

자동 생성된 설명

각 Latch를 통해서 각 단계 별로 분리가 되었다면, 매 클럭마다 처리가 발생하게 되고나서 다음 단계로 넘어가서 실행된다.

각 Latch는 두 개의 slot으로 구성되어 있으며 In-Latch slot과 Out-Latch slot로 구분되어 구현된다. 각 파이프라인 단계는 In-Latch slot으로부터 이전 단계의 결과를 입력받으며, Out-Latch slot에 데이터를 저장한다. 한 클럭이 끝나고 나면 Latch는 Out-Latch slot에 존재하는 데이터를 삭제하고, 해당 슬롯에 In-Latch slot의 값으로 저장된다. 이후, Out-Latch slot은 데이터를 비우며 다음 클럭에서 데이터를 받아들일 준비한다.

텍스트, 도표, 라인, 그래프이(가) 표시된 사진

자동 생성된 설명

따라서, 파이프라인의 전반적인 디자인은 다음과 같이 구현된다.[[2]](#footnote-2)

텍스트, 도표, 평면도, 평행이(가) 표시된 사진

자동 생성된 설명

**Forwarding Unit(Implementation에 통합)**

도표, 평면도, 텍스트, 개략도이(가) 표시된 사진

자동 생성된 설명

EX 단계에서 data dependency를 해결하기 위해, Forwarding unit을 사용한다. Forwarding Unit은 앞선 단계에서 데이터를 저장하려는 Register의 Index와 현재 사용하려는 Register의 Index와 상충되는지 확인하여 control signal을 생성한다.

MIPS micro-architecture에서 rs와 rt 필드에 있는 레지스터를 고정적으로 가지고 오기 때문에, 데이터를 가지고 오도록 한다. 이 때, 데이터를 overwrite를 할 때, latch에서 Regsiter의 rs의 값과 Register의 rt 값에 오버라이딩하여, SW에 의도한 연산이 발생하도록 한다.

**Branch Prediction**

도표, 기술 도면, 평면도, 라인이(가) 표시된 사진

자동 생성된 설명 텍스트, 도표, 평면도, 지도이(가) 표시된 사진

자동 생성된 설명

Branch prediction는 두 가지 동작, ‘History 참조-예측’과 ‘History 업데이트’로 구성되며 이는 즉 두 가지 조합회로로 구성될 수 있다. IF 단계에서 BTB에 접근하여 Branch에 관련된 History를 탐색한다. 만약 BTB 내에 History가 없을 경우 혹은 History bit에 따라 not branch로 판단되는 경우, PC+4를 다음 PC로 지정한다. 이외에는 History를 참고하여, BTB에 작성되어 있는 history의 target address를 다음 address로 지정한다.

EX 단계에서는 Branch prediction이 성공적이었는지 확인하며, 만약 branch가 아닌 경우 IF 단계와 ID 단계를 Invalid 처리하고, “EX 단계 PC 주소 + 4”로 PC 주소를 업데이트 한다. 또한 History가 존재하지 않 경우 History를 추가한다.

**3. Implementation**

**Latch**

파이프라인 구조 도입에 앞서, 각 파이프라인을 분리하는 Latch를 구현한다.



각 Latch는 위와 같은 파이프라인의 micro-architectural state를 저장하는 slot을 배열 형식으로 구성한다.



각 Latch는 데이터를 관리하기 위한 함수로 Latch의 동작을 구현하였다. 함수는 OutLatch에 존재하는 slot을 Out-Latch로 내보내는 flush 함수, 그리고 slot의 invalid 조건에 따라 zero를 반환하는 함수로 구성되어 있다. 후자의 경우, Invalid 신호에 따라서, 파이프라인 내에 존재하는 Micro-architectural state를 0으로 일괄 초기화하여 이후 SW state를 변경하지 못하게 한다.



**Pipeline stage**

각 파이프라인 단계 별로, 함수를 구성하였으며, 각 함수는 Latch의 slot을 함수의 인자로 받아들인다. 각 단계는 slot을 통해서 의도된 동작을 수행한다. 파이프라인 단계의 구현은 코드가 긴 관계로, 핵심 동작 위주로 작성한다.

IF Stage : Pipeline Termination, Branch Prediction, instruction fetch

ID Stage : set micor architectural state, get control signal from CU, get register value with rs, rt

EX Stage : forwarding logic, set ALU operand, ALU operation, Evaluate Branch prediction

MEM Stage : Access Memory

WB Stage : Write back data to register with write\_index value

**Main Function**

MIPS 파이프라인이 정상적으로 동작하기 위해, 초기 Instruction Load를 수행하며, Latch, RA, SP, PC 값을 초기화 한다. 이후, Termination이 활성화되어 있는 경우, 남아있는 명령어가 정상적으로 수행될 수 있도록 IF 단계에서 지속적으로 NOP를 전달하여 종료 이후에는 Invalid로 파이프라인을 채운다. 이후 모든 파이프라인에 Invalid가 활성화된 경우, 파이프라인을 종료한다.

파이프라인 논리적 동작 순서는 WB->IF->ID->EX->MEE 순서이다. 기존 파이프라인 순서와 상이하지만, 실제로는 한 클럭에 동시에 동작하므로 IF->ID 순서로 시작하지 않아도 문제 없다. WB Stage를 우선 실행하는 이유는 forwarding unit으로 해결하지 못하는 case가 존재하기 때문이다.

해당 case는 파이프라인의 논리적인 동작 순서의 문제이다. 가령, 명령어가 다음과 같이 구성되어 있다고 하자.



파이프라이닝이 수행됨에 따라서, 4번째 명령어가 ID 단계에 도달하면, R3과 R2의 값을 가지고 온다. 하지만, 파이프라인의 논리적 처리 순서가 IF->ID->EX->MEM->WB이므로, WB에서 R3을 저장하지 않은 상태에서 R3에 접근하게 되므로, 데이터 의존 문제가 해결되지 않는 것이다. 이에 논리적 순서를 WB가 먼저 오도록 구성하면, ID는 WB이 처리된 후에 레지스터에 접근하게 되므로 data dependency 문제를 해결할 수 있다.

**Forwarding Unit**

****

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

Forwarding Unit에서는 rs와 rt의 의존성을 검사하고, 해당 결과를 통해서 WB단계와 MEM 단계의 결과 값으로 ID\_EX\_SLOT의 레지스터 반환 값을 덮어쓴다.

도표, 평면도, 텍스트, 개략도이(가) 표시된 사진

자동 생성된 설명

**Last Branch Prediction (1-BIT)**

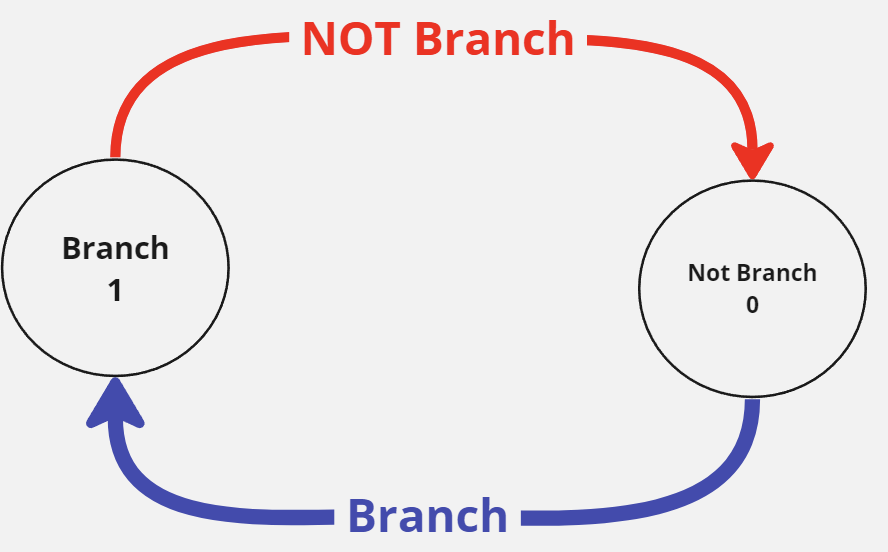
****

BTB는 Latch와 마찬가지로 구조체 배열을 통해 만들었다. 해당 구조체 배열을 관리하는 함수는 링 버퍼와 같이 동작하며, 링 버퍼가 가득찰 경우, 오래된 Branch 정보를 새로운 Branch 정보로 덮어 쓰여진다.

텍스트, 스크린샷, 폰트, 소프트웨어이(가) 표시된 사진

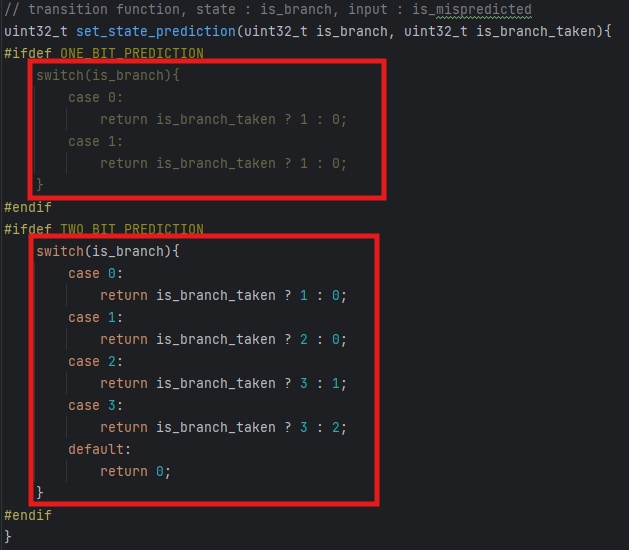
자동 생성된 설명

위 빨간 상자의 구문이 btb에 접근하여 해당 branch 정보가 branch 여부를 결정한다. 이 때, get\_state\_prediction 함수는 1-Bit, 2-Bit에 대해서 반환해야 할 prediction 정보를 제공한다.

 원, 클립아트, 디자인이(가) 표시된 사진

자동 생성된 설명

왼쪽은 1-Bit Predictor의 State diagram, 2-Bit Predictor의 State diagram을 나타낸다. 1-bit에서 2-bit으로 확장됨에 따라서, 1-bit에서 탐지하기 어려운 Branch와 Not Branch 모두 빈번하게 반복되어 나타나는 패턴을 효과적으로 예측할 수 있다.

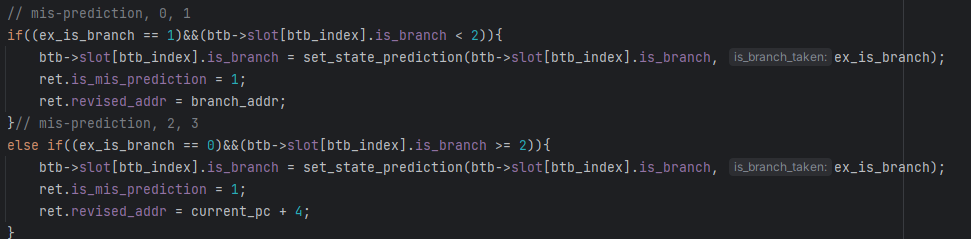


이후 Ex 단계에서는 Prediction에 대한 평가가 수행되며, 만약 Misprediction이 관찰되면 IF, ID 단계를 Invalid로 설정하고 다음 Fetch할 주소를 올바르게 수정한다.

Misprediction이 되는 경우는, 다음 두 경우에 해당된다.

1. Branch를 해야 하지만, Branch를 하지 않은 경우
2. Branch를 하면 안되지만, Branch를 한 경우

이를 바로 잡기 위해서, 각 케이스 별로 다음 인출할 주소가 달라지며, 1번의 경우 EX 단계에서 결정된 Branch 주소로, 2번의 경우는 “현재 Ex 단계에서 처리 중인 명령어의 주소 + 4”로 설정된다.



**구현된 명령어**

구현된 명령어는 이전 Single Cycle과 동일하다.

|  |  |  |  |
| --- | --- | --- | --- |
| 명령어 | 명령어 구조/타입 | 동작 | ALU 동작 |
| add | R | R[rd] = R[rs] + R[rt] | ADD |
| addiu | I | R[rt] = R[rs] + s\_imm | ADD |
| addu | R | R[rd] = R[rs] + R[rt] | ADD |
| addi | I | R[rt] = R[rs] + s\_imm | ADD |
| sub | R | R[rd] = R[rs] - R[rt] | SUB |
| subu | R | R[rd] = R[rs] – R[rt] | SUB |
| and | R | R[rd] = R[rs] & R[rt] | AND |
| andi | I | R[rt] = R[rs] & ZeroExtendImm | AND |
| or | R | R[rd] = R[rs] | R[rt] | OR |
| ori | I | R[rt] = R[rs] | ZeroExtendImm | OR |
| nor | R | R[rd] = ~(R[rs] | R[rt]) | NOR |
| lui | I | R[rt] = {Imm, 16’b0} (concatenated) | LUI |
| lw | I | R[rt] = M[R[rs] + s\_imm] | ADD |
| sw | I | M[R[rs] + s\_imm] = R[rt] | ADD |
| slt | R | R[rd] = (R[rs] < R[rt]) ? 1 : 0 | Set Value |
| slti | I | R[rt] = (R[rs] < s\_imm) ? 1 : 0 | Set Value |
| sltiu | I | R[rt] = (R[rs] < s\_imm) ? 1 : 0 | Set Value |
| sll | R | R[rd] = R[rt] << shamt | SLL |
| srl | R | R[rd] = R[rt] >> shamt | SRL |
| j | J | PC = {(PC+4)[31-28],(imm << 2)} | nop |
| jal | J | PC = {(PC+4)[31-28],(imm << 2)}  R[31] = PC + 8 | nop |
| jr | R | PC = R[rs] | nop |
| jalr | R | PC = R[rs], R[31] = PC + 8 | nop |
| beq | I | if(R[rs] == R[rt]) PC = PC + 4 + BranchAddr | SUB |
| bne | I | if(R[rs] != R[rt]) PC = PC + 4 + BranchAddr | SUB |

**5. Build Environment and Screenshot**

프로그램의 테스트 환경은 Windows 11, CLion 2023.2.1, Cmake 3.2x를 사용하였다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명 Input4.bin의 실행 결과

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명 파이프라인 실행 로그

관련 매크로

- ONLY\_RESULT : 최종 결과만 출력, 결과는 output.txt에 저장

- REGISTER\_LOG : GP Register 전부 출력

**5. Evaluation**

테스트 프로그램은 E-Learning으로 배부된 camp\_test\_prog.tar을 사용하였다. 테스트 case는 2-Bit Prediction, 1-Bit Prediction, Always Not Branch로 구성하였다.

Test Program

|  |  |  |  |
| --- | --- | --- | --- |
| 이름 | 간단한 동작 설명 | 명령어 수 | 사용하는 명령어(위에서부터 누적) |
| simple | 메모리 I/O 및 덧셈 연산 | 9 | addiu, sw, move, lw, sw, jr, sll |
| simple2 | 메모리 I/O 및 결과 값 반환 | 12 | li |
| simple3 | 1 ~ 100까지 덧셈 연산 | 28 | addu, slti, bne(bnez) |
| simple4 | 재귀 함수 처리 | 43 | jal |
| gcd | 최대 공약수 연산 | 63 | subu |
| fib | 피보나치 10항까지 연산 | 52 | beq(beqz) |
| fib\_edited | 피보나치 10항까지 연산 | 55 | jalr |
| input4 | 배열에서 101번째로 작은 값 | 25533 | lui, sll (not nop) |

Single Cycle Result

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | v[0] | # of executed Inst | # of  R type | # of  I type | # of  J type | # of  mem access | # of  branch |
| simple | 0 | 8 | 4 | 4 | 0 | 2 | 0 |
| simple2 | 100 | 10 | 3 | 7 | 0 | 4 | 0 |
| simple3 | 5050 | 1330 | 409 | 920 | 1 | 613 | 102 |
| simple4 | 55 | 243 | 79 | 153 | 11 | 100 | 10 |
| fib | 55 | 2679 | 818 | 1697 | 164 | 1095 | 109 |
| fib\_edit | 55 | 3005 | 1200 | 1750 | 55 | 1095 | 109 |
| gcd | 1 | 1061 | 359 | 637 | 65 | 486 | 73 |
| input4 | 85 | 23372706 | 101527862 | 13219741 | 103 | 7116606 | 2029699 |

Pipeline Result (for 2-Bit Branch Prediction)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | v[0] | # of  cycles | # of executed Inst | # of  R type | # of  I type | # of  J type | # of  mem access | # of  branch |
| simple | 0 | 12 | 10 | 6 | 4 | 0 | 2 | 0 |
| simple2 | 100 | 14 | 12 | 5 | 7 | 0 | 4 | 0 |
| simple3 | 5050 | 1340 | 1335 | 413 | 921 | 1 | 614 | 102 |
| simple4 | 55 | 293 | 268 | 103 | 154 | 11 | 101 | 10 |
| fib | 55 | 3313 | 2996 | 1114 | 1718 | 164 | 1116 | 109 |
| fib\_edit | 55 | 3530 | 3213 | 1222 | 1936 | 55 | 1116 | 109 |
| gcd | 1 | 1299 | 1180 | 471 | 644 | 65 | 493 | 73 |
| input4 | 85 | 23376218 | 23374462 | 10153793 | 13220566 | 103 | 7117431 | 2029699 |

Pipeline Result (for 1-Bit Branch Prediction)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | v[0] | # of  cycles | # of executed Inst | # of  R type | # of  I type | # of  J type | # of  mem access | # of  branch |
| simple | 0 | 12 | 10 | 6 | 4 | 0 | 2 | 0 |
| simple2 | 100 | 14 | 12 | 5 | 7 | 0 | 4 | 0 |
| simple3 | 5050 | 1340 | 1335 | 413 | 921 | 1 | 614 | 102 |
| simple4 | 55 | 293 | 268 | 103 | 154 | 11 | 101 | 10 |
| fib | 55 | 3313 | 2996 | 1114 | 1718 | 164 | 1116 | 109 |
| fib\_edit | 55 | 3530 | 3213 | 1222 | 1936 | 55 | 1116 | 109 |
| gcd | 1 | 1299 | 1180 | 471 | 644 | 65 | 493 | 73 |
| input4 | 85 | 23376218 | 23374462 | 10153793 | 13220566 | 103 | 7117431 | 2029699 |

Pipeline Result (Always Not Branch)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | v[0] | # of  cycles | # of executed Inst | # of  R type | # of  I type | # of  J type | # of  mem access | # of  branch |
| simple | 0 | 12 | 10 | 6 | 4 | 0 | 2 | 0 |
| simple2 | 100 | 14 | 12 | 5 | 7 | 0 | 4 | 0 |
| simple3 | 5050 | 1540 | 1435 | 514 | 920 | 1 | 613 | 102 |
| simple4 | 55 | 309 | 276 | 112 | 153 | 11 | 100 | 10 |
| fib | 55 | 3447 | 3063 | 1202 | 1697 | 164 | 1095 | 109 |
| fib\_edit | 55 | 3664 | 3280 | 1310 | 1915 | 55 | 1095 | 109 |
| gcd | 1 | 1415 | 1238 | 536 | 637 | 65 | 486 | 73 |
| input4 | 85 | 27432314 | 25402510 | 12182666 | 13219741 | 103 | 7116606 | 2029699 |

텍스트, 스크린샷, 도표, 그래프이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Simple | Simple2 | Simple3 | Simple4 | Fib | Fib\_edit | Gcd | Input4 |
| Improvement | 0% | 0% | 12.9% | 4% | 3.9% | 3.6% | 8.19% | 14.7% |
| Hit rate | NULL | NULL | 99% | 90% | 62% | 62% | 82% |  |

각 테스트 케이스 별로 측정한 결과, branch가 존재하지 않는 simple과 simple2를 제외한 모든 경우에서 성능 향상이 존재하였으나, 재귀 함수와 같이 Branch Case가 일정하지 않은 경우 낮은 성능 향상을 보여주었다. 이러한 경향이 보이는 이유 중 하나로, 재귀 함수의 전개에 따라 branch가 달라지기 때문이다. Fib.c를 보면 관찰할 수 있는데, fib.c는 fib 재귀 함수를 트리처럼 전개하는 것이 특징이다. 재귀 호출 형식은 preorder로 탐색한다. 이 경우, 재귀 CallStack은 311212 형식으로 전개되는데, 함수 Call이 1일 때 다른 함수 call stack 일때와 다르게 Branch를 하지 않고 즉시 함수 J를 수행하게 된다. 이러한 함수 Callstack이 불규칙적으로 발생하기 때문에, 이러한 last time branch을 사용하는 현재 구조에서 예측의 성능이 매우 떨어지게 된다.

5. Conclusion

텍스트, 스크린샷, 도표, 그래프이(가) 표시된 사진

자동 생성된 설명

Single Cycle Architecture에서 더욱 높은 성능을 낼 수 있도록, Multi Cycle Architecture를 거쳐 파이프라이닝을 통해 병렬 처리를 구현하여 CPU의 성능을 높일 수 있도록 하였다. 실제 파이프라인 논리 설계를 바탕으로 파이프라인에서 어떻게 micro-architectural state를 보존하여, 의도한 SW의 동작을 나타낼 수 있도록 구현하기 위해 data dependency를 해결하는 forwarding unit과 control dependency를 해결하는 branchy prediction을 통해 파이프라인을 구현하였다. 또한 기존 static하게 branch prediction을 수행하였던 것을 Last time prediction을 구현하여 더욱 효율적인 파이프라인을 구현할 수 있었다. 하지만, Last time predition의 한계점으로 빈번하게 변화하는 branch history를 가진 경우에는 효율적으로 예측이 어려우며, 이는 파이프라인 성능의 저하로 이어졌다.

Appendix. Personal Feeling and Matrix Multiplication

시간이 부족하여, 파일 내 프로젝트에 첨부하였으나, 실행과 검증을 하지 못하였습니다. 파이프라인을 구현하기 위해서, 많은 시간을 투자하지 못하였고 5월 한 달간 다사다난하게 지난 탓에 single cycle과 다르게 여유롭게 생각을 깊게하고 구현할 수 있는 상황이 아니어서 아쉬울 따름입니다.

Matrix Multiplication을 구현하였지만, 마감 시간을 훌쩍 지난 시간 급하게 각각의 테스트 케이스에 대해서 확실하게 검증하려고 하였으나, 개인 컴퓨터가 Input4를 실행하던 도중에 Segmentation Fault를 발생하였습니다. 물론, 코드에 어떠한 변화를 주지도 않았고 이전까지 잘 실행되었으나 Segmentation Fault가 발생하여 심각한 시간 낭비가 소요되었습니다.

Matrix Multiplication을 검증해보기 전에 이미 너무 많은 시간을 소비함에 따라서, 부득이하게 검증에 실패하고 제출합니다.

1. Free day : 0 day left, 1 exceeded day [↑](#footnote-ref-1)
2. Computer-Architecture/Pipeline – Project3/docs/Overall Architecture.pdf 참조 [↑](#footnote-ref-2)