DANKOOK UNIVERSITY

DANKOOK UNIVERSITY

REPORT

Project #3
Pipeline

과 목 : 컴퓨터 구조 및 모바일 프로세서

학 과 : 모바일 시스템공학과

학 번 : 32184801

이 름 : 하승원

목 차

1. Concept

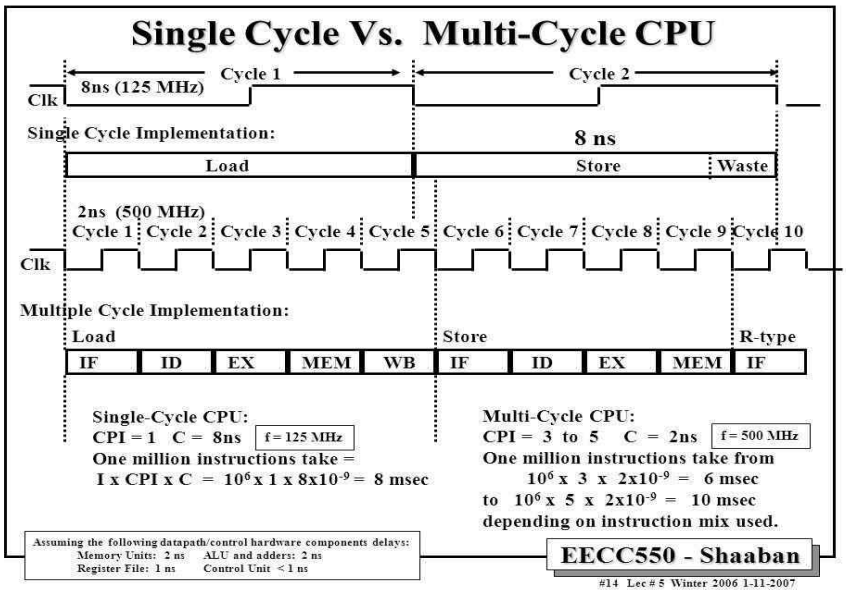
1.1 Signal Cycle

steps	IF	ID	EX	MEM	WB	Delay
resources	mem	RF	ALU	mem	RF	
R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

<그림1> Signal Cycle 처리 시간

싱글 사이클의 경우 하나의 명령어가 실행되기 위해서는 한 번의 사이클을 모두 완료해야 다음 사이클을 실행할 수 있다. 즉 1번의 사이클 당 1개의 명령어만 실행할 수 있다. 여기서 문제가 발생한다 <그림1> 과같이 LW 명령어를 기준으로 사이클 시간을 설정한다면 나머지 명령어를 실행할 때 처리 시간의 낭비가 발생한다. 또한, 싱글 사이클을 실행하면서 하드웨어의 활용도도 낮다. 이런 문제를 보완하기 위해서 멀티 사이클이 등장하게 되었다.

1.2 Multi-Cycle



<그림2> Signal Cycle, Multi Cycle 비교

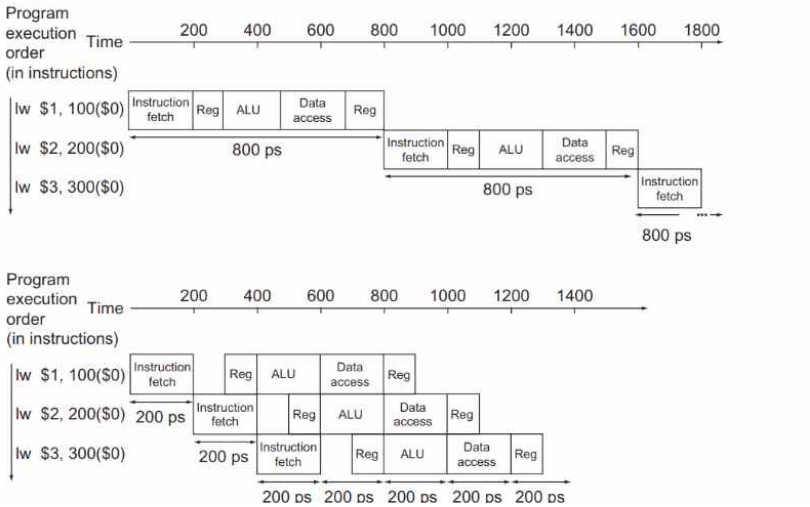
멀티 사이클이랑 싱글 사이클의 한계를 보완하기 위하여 등장하였다. 먼저 멀티 사이클의 이론은 명령어 1개가 충분히 실행할 수 있도록 1개의 사이클을 지정하는 싱글 사이클 방식과는 다르게 사이클 시간(Clock Cycle Time)을 각 명령어의 처리 5단계의 단계마다 나눠서 사이클을 나눠준다. 이때 한 단계당 Clock Cycle Time은 모든 명령어가 각 단계에서 처리가 가능한 시간이어야 함으로 Critical Path(단계 중 가장 시간이 많이 필요함)를 기준으로 그 기준을 정한다. <그림2> 에서 보는 것과 같이 싱글 사이클에서는 100만 개의 명령어를 실행한다면 사이클 당 8ns라고 생각했을 경우 8msec의 처리 속도가 나온다.

반면 멀티 사이클에서는 한 사이클 당 2ns로 기준을 정했고 명령어 1개당 3번의 사이클이 필요하지만, 명령어의 조합에 따라 6msec~10msec로 싱글 사이클보다 효율적이거나 비효율적일 수 있다. 즉 명령어 처리 속도가 긴 명령어가 많은 프로그램에서는 싱글 사이클보다 더 비효율적일 수 있다. 이런 문제를 보완하고자 파이프라인이라는 개념이 나오게 되었다.

1.3 What Is Pipeline

파이프라인은 단일 명령어 처리인 싱글 사이클과 멀티 사이클의 한계를 보완하기 위하여 나온 개념으로 명령어를

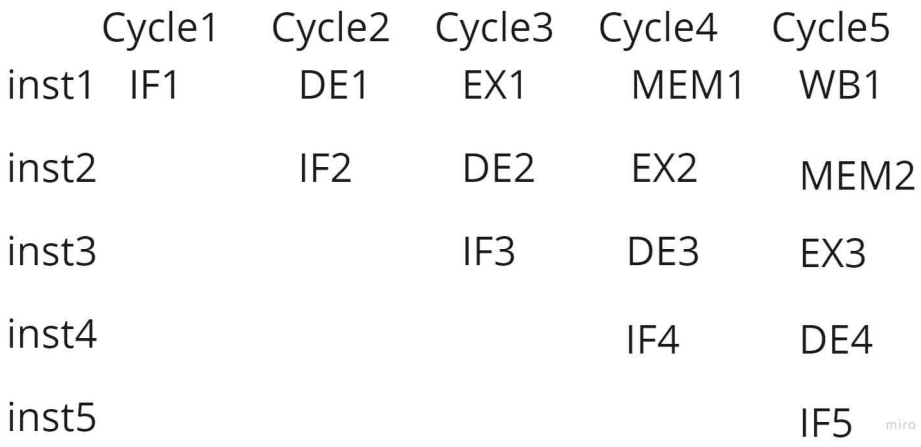
병렬로 처리하여 명령어를 처리해서 단일 명령어 처리 방식보다 처리 속력을 높이는 방식이다.



<그림3> Signal Cycle, Pipeline 명령어 처리 시각화

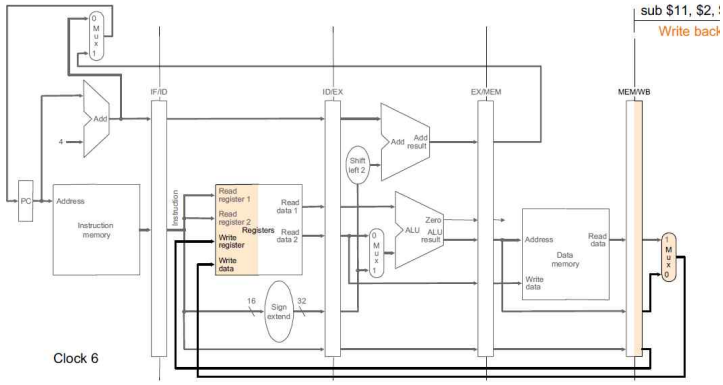
<그림3>에서와 같이 단일 명령어 처리는 보통 1개의 명령어가 처리되는 것을 기준으로 처리 속도를 측정한다. 하지만 <그림3>처럼 파이프 라이닝 방식에서는 성능이 더욱 향상되며 명령어가 충분히 많아진다면 200ps마다 한 개의 명령어가 처리된다고 볼 수 있으므로 단일 명령어 처리보다 4배의 성능이 향상되었다고 이야기를 할 수 있다. 즉 한 개의 명령어를 처리하는 총시간을 줄어드는 것이 아니지만 병렬로 명령어를 처리하는 ILP(Instruction-Level Parallelism) 방식을 명령어들을 중첩하여 실행하므로 전체 프로그램 처리 성능을 높인다.

1.3.1 Latch



<그림4> Pipeline 명령어 처리 순서 시각화

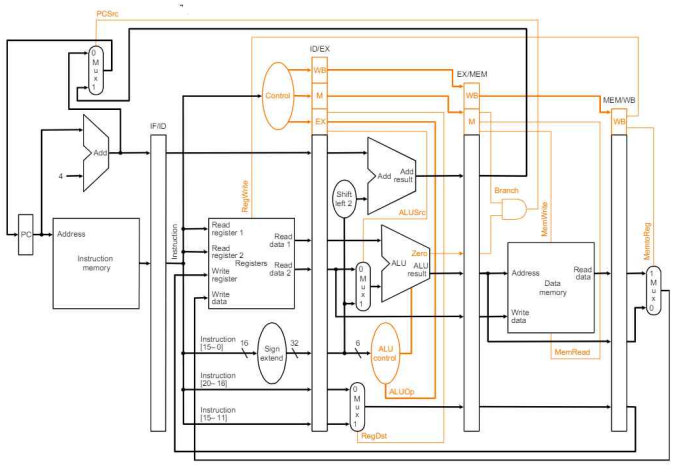
파이프라인을 구현하면서 이제 단계별로 다른 명령어를 처리해주어야 한다. 즉 <그림4> 처럼 첫 명령어가 IF 단계를 거치면 한 번의 사이클이 종료된다. 이후 두 번째 사이클에서는 다음 명령어를 IF를 해주고 이전 명령어는 DE 단계를 처리해준다. 이후 세 번째 사이클에서도 그다음 명령어를 Fetch 해주어야한다. 이때 어떤 명령어의 값을 처리하고 또 어떤 명령어의 값을 저장해야 하는지 명령어마다 구분해주지 않는다면 파이프라인 구조를 실행할 수 없다. 그러니 명령어마다 정보를 저장해주는 하드웨어 “Latch”가 필요하다.



<그림5> Pipeline 구조 DataPath

<그림5> 와 같이 Latch는 단계별로 4개의 래치를 두어서 단계별로 실행되는 명령어들의 정보들을 저장하는 하드웨어이다. Latch는 프로그램이 실행되는 첫 단계에서는 모두 값이 없이 비어 있는 상태로 시작한다. 그리고 1개의 Latch는 input, output 2개로 또 나누어지며 각 단계가 실행되면 명령어의 결괏값을 Latch의 input에 저장하고 이후 다음 단계에서는 Latch의 output 값을 통해서 명령어를 처리한다.

1.3.2 ControlSignals



<그림6> Pipeline 구조 DataPath

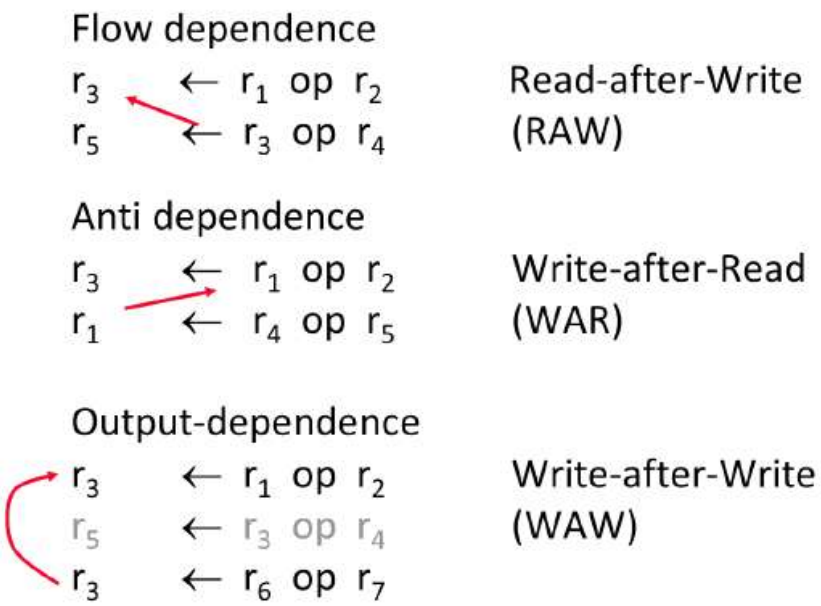
파이프라인 구조에서는 각각 명령어가 병렬처리로 프로그램을 처리한다. 이때 각각 명령어 처리 단계에서 필요한 MUX를 조절해주는 ControlSignal을 다르게 처리해주어야 한다. 파이프라인 구조에서는 이제 ControlSignal들을 저장하는 래치와 같은 레지스터를 래치와 같이 단계별로 사용할 수 있도록 해주었다.

1.4 Data Dependency

파이프라인 이론에서는 독립적인 명령어를 병렬로 처리하며 프로그램의 처리 성능을 높여주는 이론이었다. 하지만 실제 프로그램을 실행하면 여러 가지 문제가 발생한다. 먼저 항상 같은 명령어가 오는 것이 아니며 명령어마다 Dependency가 있다.

-What is Data Dependency

위의 설명에서 Dependency가 명령어마다 발생할 수 있다고 이야기를 했는데 파이프라인 구조에서는 크게 Data Dependency와 ControlDependency 크게 두 가지로 나누어서 설명할 수 있다. 먼저 Data Dependency를 먼저 설명하자면 Dependency는 일정 거리 이내의 명령어가 같은 레지스터의 값을 사용하면서 발생한다.



<그림7> Data Dependency가 발생하는 경우

<그림7>에서 처럼 가까운 거리의 명령어가 각각 같은 레지스터를 사용할 때 문제가 발생한다. 여러 개의 명령어가 한 사이클 안에서 작동하는 파이프 라이닝에서는 WB 단계 이후 레지스터에 값이 저장되기 이전에 그 레지스터의 값을 사용하면 예상 결과에 따라 처리가 되지 않을 수 있다. <그림7>에서 처럼 3가지 정도의 경우 Data Dependency가 발생하는데 Read-After-Write(RAW)의 경우에는 레지스터에 값을 저장하고 그 값을 다음 명령어에서 사용할 때 발생한다. 이 경우에는 레지스터에 값을 저장하는 WB 단계와 레지스터를 읽는 DE 단계에서는 Dependency가 발생하는데 값을 먼저 저장하고 그 값을 사용하도록 처리를 해주어야 한다. Write-After-Read(WAR), Write-After-Write(WAW)는 파이프 라이닝 구조에서 병렬로 명령어를 처리하면서 발생할 수 있는데 이후에 오는 명령어가 먼저 처리될 때 발생할 수 있다. 그래서 3가지 종류의 Data Dependency를 처리해주어야 한다.

1.4.1 Handle The DataDependencies

-Hardware Interlocking

1. Detect and Wait: Data Dependency가 발생하면 Dependency가 발생하는 이전 명령어가 모든 처리가 끝나도록 기다렸다가 다음 명령어를 처리한다.
2. Detect and Forward/Bypass: Data Dependency가 발생하면 모든 명령어 처리 5단계를 거치기 전에 각 Dependency가 발생하는 이전 명령어의 처리된 값을 이후 명령어 처리 단계로 값을 넘겨서 모든 과정이 끝나지 않고도 그 값을 사용하여 명령어를 처리한다.
3. Predict Verify: Dependency가 발생하는 명령어에서 결괏값을 0, 1 같은 간단한 결괏값으로 예측해서 값을 처리해준다.
4. Do Something Else(Fine-Grained Multithreading): 값을 처리하는 레지스터 스레드를 5개 지정해준다. 그래서 PC값을 5개로 나누어 처리해줌으로써 명령어 처리 과정에서 Dependency가 발생하지 않고 각각 명령어가 처리되는 값은 각기 다른 스레드에 저장된다. 5개의 이상의 명령어가 처리되어야 하며 또 추가적인 하드웨어가 더 필요하다.

-Software Based Interlocking

1. Putting nop in Software: 컴파일하는 단계에서 Dependency가 발생하면 Dependency가 발생하는 명령어들 사이에 nop 명령어를 추가해서 처리해준다.
2. Detect and Eliminate: 컴파일 단계에서 Dependency가 발생하는 명령어를 값을 처리하는 다른 연산 처리로 바꿔서 명령어를 처리해준다.

1.4.2 Score boarding

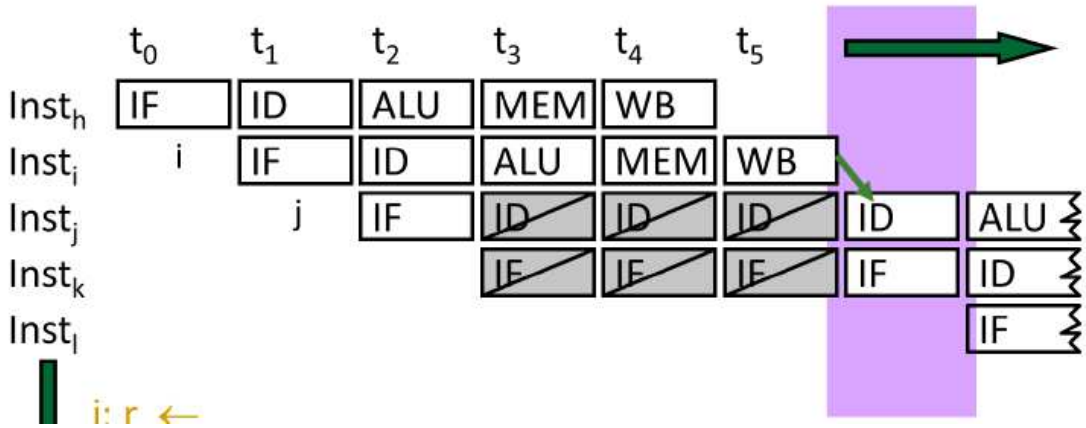
Reg #	data	valid	tag
0	0	1	
1	aaa	1	
2	bbb	1→0	1
…	…	1	
31	XXX	1	

<그림8> Score boarding이 작동하는 방법 시각화

Score boarding은 모든 레지스터에 2가지 value 값을 지정해주고 마치 점수판처럼 그 레지스터의 상태를 나타낸다. 먼저 valid는 현재 레지스터가 파이프라인에 사용되고 있는지 상태를 나타내 주며 tag는 몇 번째 명령어에서 마지막으로 사용되는지를 표시한다. 즉 2번 레지스터를 1번 명령어와 2번 명령어에서 사용한다면 1번 명령어를 Decode 하면서 valid 값을 0으로 만들어 주고 tag를 1로 설정한다. 그리고 2번 명령어를 Decode 하면서 valid 값을 그대로 두고 tag를 2도 초기화 해준다.

이런 식으로 모든 명령어가 처리되고 2번에 값이 저장되면 valid 값을 1로 만들어 주고 tag는 마지막으로 사용되었던 명령어의 숫자로 초기화해주며 Dependency가 valid 값으로 탐지할 수 있다. 이 방식을 통해서 WAR, WAW Dependency를 처리할 수 있다.

1.4.3 Stalling



<그림9> Stalling 방식으로 명령어를 처리했을 경우를 시각화

- Stall when
  - $(rs(IR_{ID})==dest_{EX})$ && $use\_rs(IR_{ID})$ && $RegWrite_{EX}$ or
  - $(rs(IR_{ID})==dest_{MEM})$ && $use\_rs(IR_{ID})$ && $RegWrite_{MEM}$ or
  - $(rs(IR_{ID})==dest_{WB})$ && $use\_rs(IR_{ID})$ && $RegWrite_{WB}$ or
  - $(rt(IR_{ID})==dest_{EX})$ && $use\_rt(IR_{ID})$ && $RegWrite_{EX}$ or
  - $(rt(IR_{ID})==dest_{MEM})$ && $use\_rt(IR_{ID})$ && $RegWrite_{MEM}$ or
  - $(rt(IR_{ID})==dest_{WB})$ && $use\_rt(IR_{ID})$ && $RegWrite_{WB}$

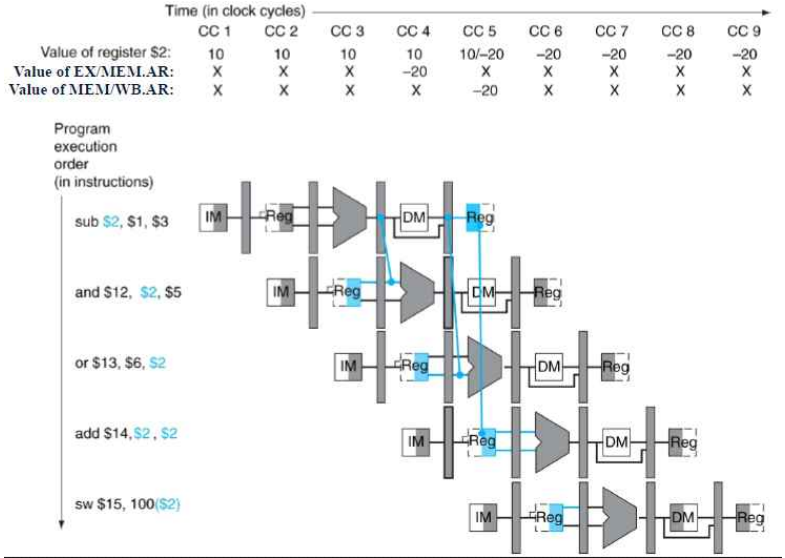
<그림10> Stalling이 발생하는 조건

<그림9>에서 본다면 먼저 Dependency가 발생하면 이전 명령어 처리가 완료될 때까지 기다려주는 작업을 해주어야 하는데 레지스터를 읽는 rs, rt 레지스터의 DE 단계에서 <그림10> 조건에 따라 Stalling 작업을 해줘야 한다.

하지만 Stalling 방식은 구현 방식에서 간단함이 있지만, Dependency가 많이 발생하면 Stalling 타임(Stalling 3 cycle)이 많아지면서 전체적인 처리 속도를 지연시키는 문제가 발생한다. 그래서 이번 프로젝트에는 Forwarding 방식을 채택하였다.

1.4.4 Forwarding/Bypass

Forwarding을 사용하는 이유는 Stalling과는 다르게, 이전 명령어가 처리되는 값을 WB 단계를 거치기 이전에 값을 가지고 와서 Data Dependency를 처리하는 방식이다. 이런 방식을 Stalling에서 불필요한 Cycle Time을 줄일 수 있으므로 Stalling 처리보다 효율적이라고 말할 수 있다.



<그림11> Forwarding Bypass 방식 시각화

```
if (rsEX!=0) && (rsEX==destMEM) && RegWriteMEM  then
   forward operand from MEM stage          // dist=1
else if (rsEX!=0) && (rsEX==destWB) && RegWriteWB  then
   forward operand from WB stage  // dist=2
else
   use AEX (operand from register file)          // dist >= 3
```

<그림12> Forwarding Bypass 일어나는 조건

<그림11> 번처럼 Data Dependency가 발생하는 경우 Dependency가 발생하는 거리를 Detection하고 그 거리만큼의 처리 결괏값들을 상황에 맞춰 EX 단계에서 ALU에 들어가는 레지스터의 값을 바꿔서 연산 처리를 해준다. 그림12)을 살펴보면 거리가 1개 차이인 명령어에서는 MEM 단계에서의 연산 처리 이후에 값을 Forwarding 해서 이후 명령어에 연산 처리 과정에서 사용한다. 거리가 2인 명령어 Dependency는 WB 해주는 값을 Forwarding 해서 이후 명령어에 연산 처리하는 값으로 사용한다. 이렇게 1, 2거리 차이의 값들은 이전 단계의 결괏값으로 연산 처리를 해주는데 거리가 3인 명령어는 Forwarding 방식을 사용하지 않아도 된다. 그 이유는 3거리인 Dependency는 LW일 경우에 발생하는데 이 경우 gcc에서 Dependency가 발생하는 경우 nop 명령어를 추가해줌으로써 이번 MIPS 코드 구현에서는 거리가 3인 Dependency를 구현하지 않고도 코드 구현이 가능했다.

1.5 Control Dependency

WB 단계 이후의 Control Signal로 PC 주솟값을 지정해주어야 하는데 이때도 Data Dependency와 같이 Control 신호에 따라 다음 명령어를 결정하는 PC값도 Dependency가 발생하게 된다.

Jump Types

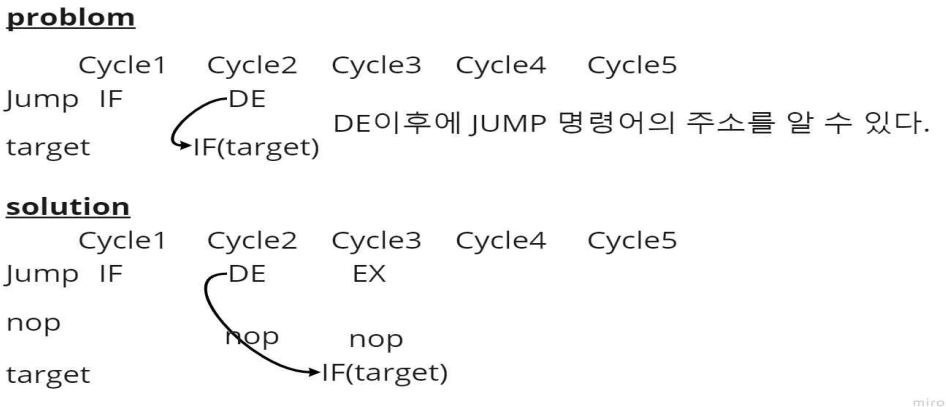
Type	Direction at fetch time	Number of possible next fetch addresses?	When is next fetch address resolved?
Conditional (BEQ, BNE)	Unknown	2	Execution (register dependent)
Unconditional (J)	Always taken	1	Decode (PC + offset)
Call (JAL)	Always taken	1	Decode (PC + offset)
Return (JR)	Always taken	Many	Execution (register dependent)
Indirect (JR)	Always taken	Many	Execution (register dependent)

Different branch types can be handled differently

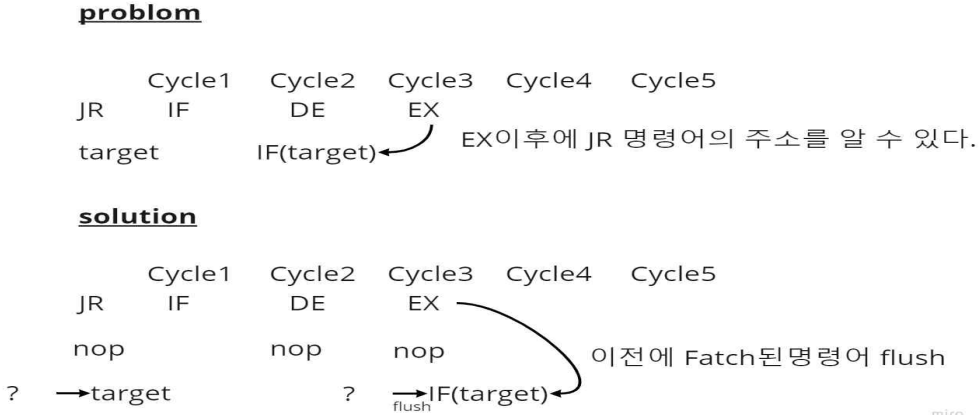
<그림13> PC값을 Jump 하는 명령어들

<그림13>을 보면 언제 Dependency가 발생하는지 알 수 있다. 명령어 처리는 파이프라인에서 병렬로 처리가 되는데 항상 다음 주소에 있는 명령어를 불러온다. 하지만 <그림13> 와 같은 특정 명령어는 이전 명령어의 다음 주소가 아닌 특정 주소로 명령어를 읽어와야 하므로 이때 파이프라인 구조에서 Dependency가 발생한다. 그래서 Data Dependency와 비슷하게 처리해주고 명령어 처리를 해줘야 한다.

그렇다면 Dependency가 발생하는 명령어들을 살펴보면 <그림13>과 같이 여러 가지 명령어가 Dependency가 발생할 수 있다. 여기서 Jump 타입의 명령어들은 각각 다르게 해결이 된다.



<그림14> 점프하는 모습(J, JAL)



<그림15> JR 모습

<그림14>는 Jump, JAL 명령어를 처리하는 방식으로 DE 단계에서 명령어의 목표 주솟값을 알 수 있다. 하지만 gcc 컴파일러에서 이런 문제를 해결하기 위해서 nop 명령어 즉 아무것도 아닌 명령어를 사이에 추가해줌으로써 이를 컴파일 단계에서 해결할 수 있도록 해주었다. 그래서 추가적인 처리 없이도 Jump, JAL 명령어는 처리할 수 있다. 하지만 <그림15>처럼 JR 명령어는 조금 다르다. JR 명령어는 EX 단계에서 R[rs]의 값을 이용하여 PC값을 조정하는 명령어이다.

그래서 이는 nop 명령어가 1개 있더라도 그냥 처리할 수 없다. 그래서 이번 프로젝트에서는 나는 목표 주소를 얻는 EX 단계에서 nop명령어 이후에 나오는 명령어를 flush 시켜줌으로써 제거해주고 목표 주소의 값을 Fetch 시켜서 구현을 처리했다.

하지만 Branch 명령어는 다른 명령어와 다르게 조건에 따라 주솟값을 다르게 설정하게 된다. 그래서 Branch 명령어는 Jump 명령어와 다르게 처리해야 한다.

1.5.1 How to Handle Control Dependencies

Dependency를 처리하는 방식에는 총 6가지 방식이 존재한다.

1. Stalling: Data Dependency와 비슷하게 Dependency가 발생하는 명령어 사이에 멈추는 Stall 작업을 해줘서 이전 명령어가 다 실행이 완료되고 다음 명령어를 실행하게 처리하는 방식이다.
2. Branch Prediction: Dependency가 발생하면 이후에 올 명령어의 주소를 예측하여 다음 명령어를 Fetch 한다.
3. Branch Delay Slot: Data Dependency와 비슷하게 gcc 컴파일러가 dependency가 발생하면 명령어 이후에 nop 명령어를 추가하여 이를 해결한다.
4. Fine-Grained Multithreading: 이것도 Data Dependency와 같이 5개의 쓰레드를 두고 각각 다른 PC값을 이용하여 명령어를 처리하므로 Dependency를 해결한다.
5. Predicated Execution: branch 명령어를 제거하고 Predicated bit의 조건에 따라 명령어를 처리한다.
6. Multipath Execution: 가능한 두 가지의 주소의 명령어를 Fetch한다.

1.5.2. Stalling

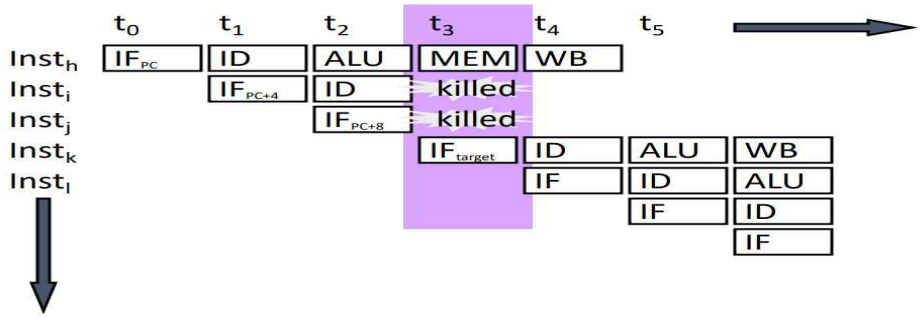
위의 <그림9> 과와 같이 Dependency가 발생하면 이전 명령어가 처리가 완료될 때까지 멈추는 작업을 해주고 그 이후에 PC값을 이용하여 다음 명령어를 실행하기 시작한다. 하지만 여기서 문제는 Data Dependency와 비슷하게 구현하거나 작동 방식이 굉장히 단순하지만, stall 해주는 추가적인 하드웨어가 필요하고 또 stall이라면서 실행 과정 중 지연되는 시간이 발생하면서 전체적인 처리 속도를 저하할 수 있다. 그리고 이후 추가로 파이프라인의 길이가 길어지는 컴퓨터 구조에서는 더욱더 이 방식을 이용한다면 한계점이 명확하게 성능으로 드러나게 될 것이다.

1.5.3 Branch Prediction

이번 방식은 Stall 작업을 해주는 과정에서 발생하는 단점을 보완하고자 나온 이론으로 Dependency가 발생하면 이후 실행될 명령어의 주소를 예측해서 실행하는 방식이다. 다시 말해 IF 단계에서 주소를 예측하여 명령어를 메모리에서 불러온다.

1.5.3.1 Different Type of Branch Prediction

Branch Prediction을 제일 간단한 방식인 항상 PC값에 4를 더하는 방식으로 실행 방식을 설명하자면



<그림16> Prediction 예측 실패 시 처리 과정 시각화

<그림16> 번은 프로그램을 실행하다가 Dependency가 발생하는 Branch 명령어가 실행되고 조건이 성립하여 목표 주소로 명령어를 실행해야 하는 상황이다. 먼저 Branch 명령어는 EX 단계에서 값을 비교하여 그 값을 통해서 목표하는 주소로 PC값을 바꾸어주는 명령어이다. 파이프라인 구조에서는 병렬로 명령어를 처리하려면 Branch 명령어가 Fetch 되고 이후 단계로 진행될 때 이후 명령어도 계속 Fetch 된다. 여기서는 Branch 다음의 명령어를 PC+4값으로 예상하고 계속 프로그램을 실행했지만, 프로그램이 구현이 잘 되려면 Branch 다음 명령어는 Branch 조건에 맞는 명령어가 나와야 한다.

하지만 EX 단계에서 그것을 확인할 수 있으므로 예측이 틀렸을 때는 이전 단계에 실행되고 있는 명령어들의 값을 초기화해주고 목표 주소에 있는 명령어를 불러온다. 이렇게 Branch Perdition의 기본적인 작동 방식이었고 예측해서 PC값을 지정 해주는 방식에 따라서 여러 가지 Branch Prediction의 종류가 나누어진다.
Perdition은 컴파일 당시에 예측할 수 있도록 처리하는 Static 방식과 실행하는 과정에서 예측하는 Dynamic 방식으로 나뉜다. Static 방식에서는 컴파일하면서 Branch 명령어에 힌트가 될 수 있는 비트를 설정하여서 종류에 따라서 다음 값을 예측하도록 하는 방식이다. Dynamic 방식은 프로그램이 실행되고 있는 과정에서 이전 Branch의 결과를 통해서 이후에 나오는 Branch에 조건을 예측하여 다음 명령어를 실행한다.

1.5.3.2 Compile Time Prediction

-Always not-Taken

이 방식은 Dependency가 발생하지 않는 파이프라인 구조와 같이 다음 PC값을 Branch가 조건이 성립하지 않는 PC+4갑으로 실행되도록 컴파일 단계에서 처리해준다. 이 방식은 EX 단계에서 만약 Branch 명령어가 조건이 만족한다면 이전 값을 초기화해주는 flush 작업이 필요하므로 정확도와 처리 성능의 향상을 높이긴 어렵다. 하지만 이런 방식의 장점은 다른 처리 방식보다 구현이 쉽고 또 추가적인 하드웨어가 필요하지 않다.

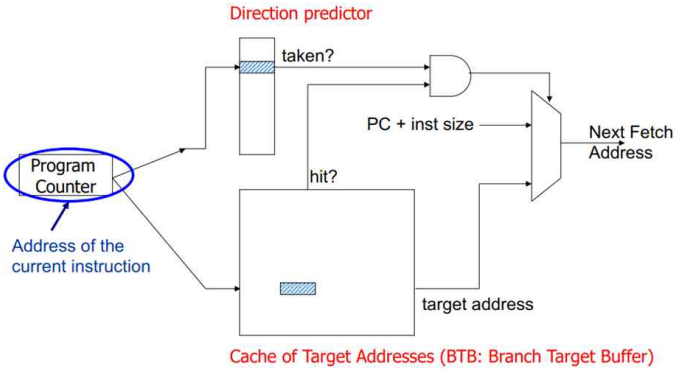
-Always Taken

Always Taken 방식은 Not Taken 방식과 반대로 항상 Branch 명령어가 나오면 항상 Branch의 명령어로 다음 명령어를 실행하는 방식을 말한다. 이 방식은 이전 방식보다는 조금 더 정확도를 높일 수 있다. 하지만 추가적인 BTB(Branch Target Buffer: Branch의 목표 주소가 저장된 하드웨어)가 필요하다. 이 방식도 컴파일 단계에서 지정이 되며 Always not Taken 방식과 비슷하게 컴파일 단계에서 이것을 결정하도록 힌트를 주도록 컴파일된다.

-BTFN (Backward Taken, Forward not Taken)

BTFN 방식은 위의 두 방식보다 조금 복잡한 방식이다. Branch 명령어가 만약 Branch 명령어보다 이전 명령어를 목표 주소로 잡는다면 그 Branch는 조건이 성립되도록 다음 명령어를 예측한다. 만약 목표 주소가 Branch 명령어보다 이후에 있는 명령어이면 이 Branch 이후에 명령어는 조건이 성립하지 않는다고 예측한다. 즉 이후 명령어인 PC+4갑으로 다음 명령어를 예측해서 프로그램을 실행한다. 이 방식에도 Always Taken 방식처럼 BTB라는 추가적인 하드웨어가 있어야 한다.

1.5.3.3. Run Time Prediction(Dynamic)



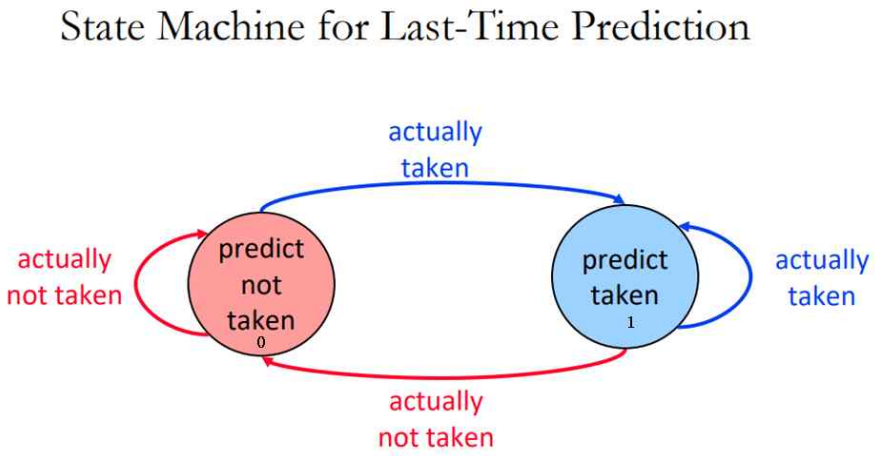
<그림17> Dynamic Prediction Data Path

Dynamic Prediction 방식은 프로그램이 실행되는 과정에서 이번 Branch 명령어와 조건을 이용하는 방식이다. 하지만 이 방식을 사용하려면 Fetch 단계에서 3가지 조건을 생각해주어야 한다. 첫째, 명령어가 예측이 필요한 Branch 명령어인지 확인한다. 둘째, 이전 Branch 명령어가 연산 조건이 성립했는지 안 했는지 확인한다. 셋째, 이번 Branch 명령어의 목표 주소가 어디인지 확인한다. 이렇게 추가적인 요소를 Fetch 단계에서 확인해야 하므로 각각

필요한 요소들이 저장된 추가적 하드웨어를 요구한다. 대표적으로 Branch 명령어의 목표 주소가 저장된 BTB(Branch Target Buffer)와 이전 Branch의 연산 결과가 있는 BHT(Branch History Table)가 필요하다.

-Last time Prediction(Signal-Bit)

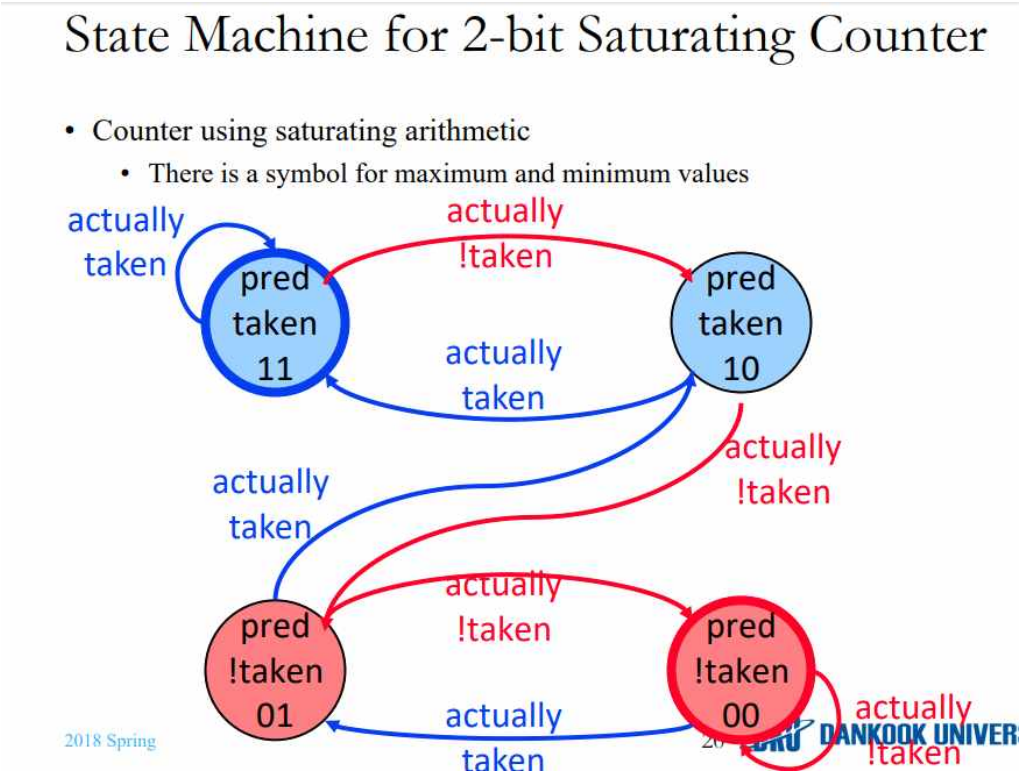
이번 방식은 <그림17>에서 BHT(Branch History Table) 하드웨어에서 이전 Branch 명령어의 연상 조건을 1개의 비트로만 나타내는 방식이다. (BTB에도 비트가 저장되어 처리할 수 있지만, 이번 프로젝트에서 분리되게 구현하였다.) 이 방식의 장점은 프로그램의 같은 결과가 반복되는 Branch 수가 늘어난다면 Static에서의 기본적인 처리 방식들보다 정확도가 올라간다. 하지만 항상 연산 처리의 결과가 성립하거나 성립하지 않는 Branch 명령어가 계속하여 나타난다면 정확도가 0에 가깝게 나타나는 현상이 나타난다.



<그림18> One Bit Predictor State Machine

이 방식이 작동되는 방식을 간단하게 설명하자면 <그림18> 번과 같이 첫 시작에는 0으로 시작해서 프로그램을 실행하다가 이후 첫 Branch 명령어를 EX 단계에서 연산 처리를 해주고 그 조건이 만족한다면 상태를 1로 바꿔준다. 이후 첫 Branch 명령어의 목표 주솟값을 BTB에 저장해준다. 하지만 여기서 항상 발생하는 지연은 이전 Branch 명령어와 이후에 나오는 Branch 명령어가 연산 결과가 다르다면 예측이 실패한다. 그래서 비트가 1개인 One Bit Prediction은 너무 빠르게 조건이 바뀐다. 그래서 이후 나오는 방식으로 이런 문제들을 보완해준다.

-Two-Bit Counter Based Prediction



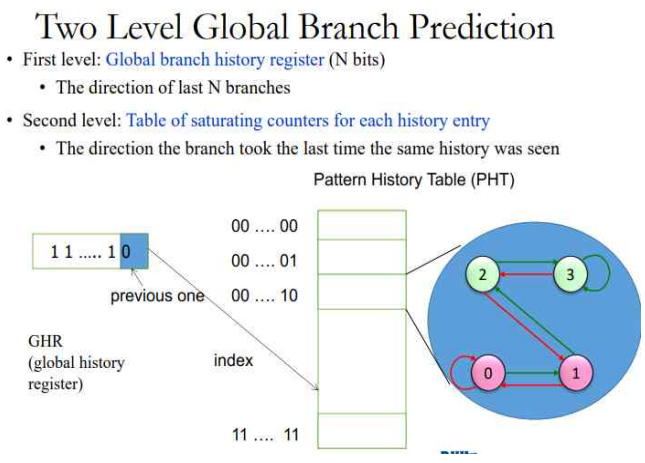
<그림19> Two Bit Predictor State Machine

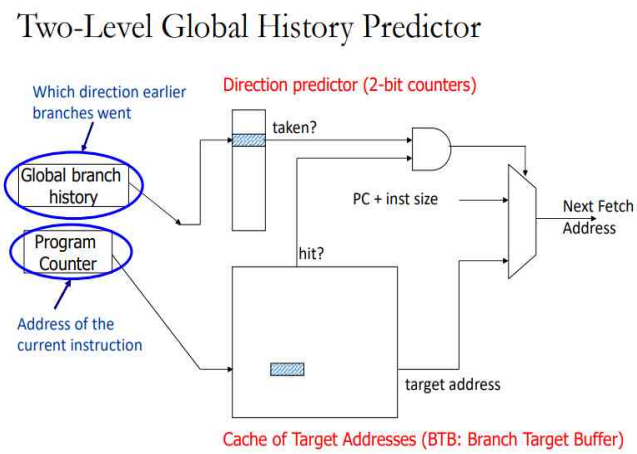
이 방식은 1 Bit와 구성은 같지만, Table에 2 Bit로 이전 명령어의 상태를 나타낸다.
이 방식의 장점은 1 Bit의 빠른 조건 변화의 단점 보안이 가능하다. 이번 명령어와 비교하여 2번 이상의 변화가 있어야 상태가 변경되기 때문에 1 Bit보다 성능이 조금 더 향상될 수 있다. 위의 <그림19>의 방식처럼 11, 과 10을 Branch 명령어가 성립되는 조건으로 다음 명령어를 예측하고 01, 00은 Branch 명령어가 연산 조건이 성립하지 않는 조건으로 다음 명령어를 예측한다. 이후 EX 단계에서 예측한 명령어와 Branch 명령어가 조건에 따라 목표로 하는 주소와 같은지 비교를 하고 맞으면 1을 더하여 상태를 변화시키고 연산 조건이 만족하지 않는다면 1을 빼주는 형식으로 상태를 바꿔준다. 이런 방식을 이용한다면 정확도가 조금을 더 향상되지만, 추가적인 하드웨어가 필요하다. 1 Bit에 비해서 1 Bit 더 추가된 하드웨어적 비용이 들어간다.

하지만 이 방법 또한 충분히 프로그램을 실행하기에는 충분한 방식은 아니다. 2 Bit Prediction의 정확도는 85~90% 정도까지 기대가 되지만 이는 사이클의 수가 많아지면 많아질수록 불필요한 사이클 수가 증가하게 된다. 가령 1000개의 명령어가 99%의 정확도로 처리된다고 가정해보자. 모두 예상이 맞게 Fetch 한 경우 1000 사이클이 필요하다. 하지만 1%의 비 정확성 때문에 추가로 10개의 사이클이 발생한다. 이는 전체적으로 봤을 경우 명령어가 많아지면 많아질수록 1%의 비 정확성도 프로그램 성능에 치명적일 수 있다. 그래서 이번 방식보다 더욱 정확성이 높은 방식이 필요했고 이는 이후 설명하겠다.

-Two Level Prediction

-Global





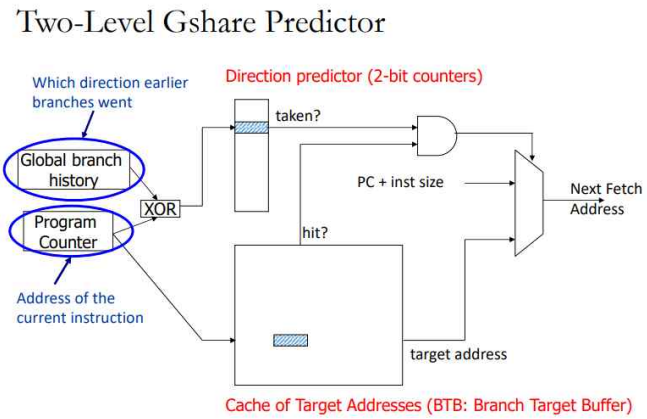
<그림20> Two Level Global History Predictor <그림21> Two Level Global History Predictor Data Path

2 Bit Prediction 방식에서의 한계 즉 처리하는 명령어의 수가 늘어났을 경우 필연적인 예측의 실패들로 정확성의 성능을 저하하는 문제가 발생했다. 이런 한계를 보완하기 위하여 Two Level Prediction 기술이 개발되었다. 먼저 이 기술을 구현하기 위해서는 2가지의 하드웨어가 필요하다. GHR(Global History Register) 이것은 이전 Branch 명령어들의 연산 결과를 가지고 결괏값을 이용하여 패턴을 만들고 그것을 저장하는 하드웨어이다. 첫 시작 상태에서는 0값으로 시작하지만 이후 프로그램을 실행하면서 EX 단계에서 Branch 명령어가 연산 처리가 되고 그 결괏값을 이전 GHR 값을 Left Shift를 해준 다음 결괏값을 마지막에 추가해준다. 이렇게 일정한 결괏값으로의 패턴을 만들어 내고 이후에는 PHT(Pattern History Table) 에 그 패턴을 인덱스값으로 사용하여 이전 값들의 History를 확인한다.

여기서 PHT 의 History는 이전 2 Bit table과 같이 2 Bit로 이루어져 있으며 2 Bit 와 똑같이 Branch 연산이 성립하면 1을 더해주고 성립하지 않으면 1을 빼주는 처리를 해준다. 하지만 이런 방식에도 문제가 발생하는데 만약 여러 다른 종류의 Branch 명령어가 프로그램에 존재하면 문제가 발생한다. Branch 명령어를 연산 처리하면 GHR에 패턴을 만들어서 값을 저장하는데 이때 다른 Branch 명령어여도 패턴이 만들어지고 그 값을 인덱스값으로 사용하여 테이블에 접근하므로 다른 Branch 명령어여도 같은 인덱스 값으로 History를 참조할 수 있다.

예를 들어 <그림20>과 같이 1번 Branch 명령어의 패턴이 1111이고 2번째 Branch 명령어의 패턴도 1111인 경우에 값은 인덱스값을 사용하면서 원하는 실행이 이루어지지 않을 수 있다. 그래서 이런 문제를 해결하고 정확도를 높여주기 위해 Gshare 방식이 개발되었다.

-Gshare

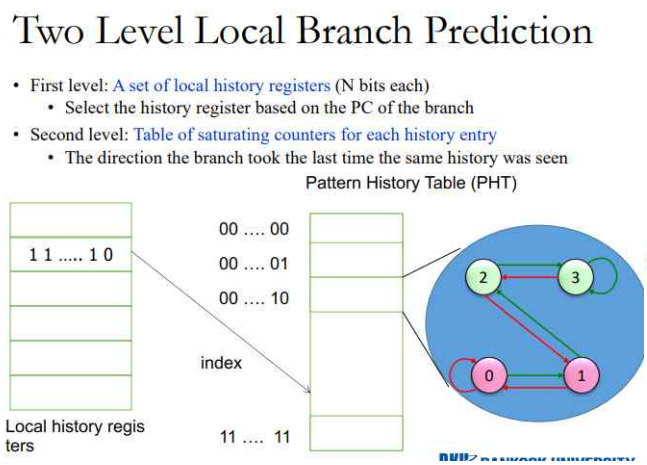


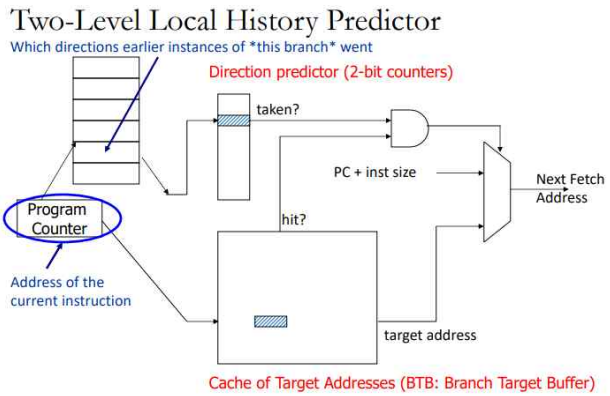
<그림22> Gshare Two Level Global History Predictor

이전 방식은 다른 Branch 명령어가 같은 테이블을 사용할 수 있다는 문제가 발생했는데 이런 문제를 해결하기 위해 PC값과 GHR을 XOR 연산을 해주는 방식인 Gshare방식이 개발되었다. 이 방식은 PC의 값과 그리고 GHR을 XOR 연산 처리를 해줌으로써 PHT 테이블의 인덱스 값을 지정하는 값이 PC와 GHR 2가지가 된다. 그렇게 되면서 다른 PC 주소에 존재하고 있는 Branch 명령어는 이 방식을 통해서 다른 인덱스값을 가지게 되고 테이블에 각각 다르게 접근하여 History를 업데이트 해주어야 한다. 그래서 이전 방식보다 조금 더 정확도가 올라가게 되고 효율적이라고 말할 수 있다. 하지만 이전 방식보다 처리하는 단계가 한 개 더 많아지면서 지연 시간이 증가한다는 단점도 존재한다.

Global Two Level Prediction 방식은 만약 비트 수를 늘리는 방식을 선택한다면 테이블의 크기가 증가하면서 정확도를 올리는 것도 가능하지만 단점도 발생한다.

-Local





<그림23> Two Level Local Branch Prediction

<그림24> Two Level Local Branch Prediction Data Path

Local Branch Prediction 방식은 이전 방식과는 다르게 PC값에 따라 다르게 패턴을 만들어 주는 방식이다. 이전 방식의 문제는 GHR이 다른 종류의 Branch 명령어도 같이 패턴을 만들어서 문제가 발생했는데 이 방식은 PC값 즉 다른 Branch 명령어에 따라 다르게 패턴을 지정해준다. 이를 통해서 이 값을 인덱스값으로 테이블에 접근한다면 각기 다른 인덱스 값으로 처리할 수 있으며 충분한 학습이 된다면 이전 방식보다 더 겹치지 않게 해줌으로써 성능이 향상된다.

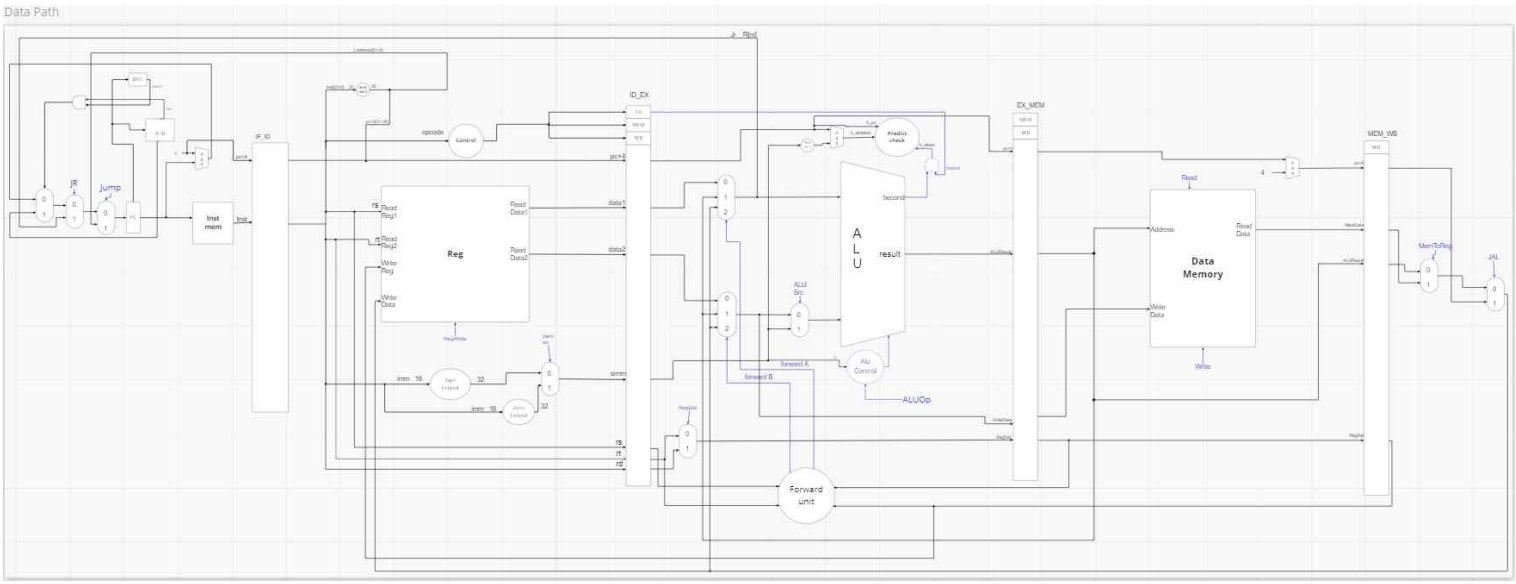
-Two Level Prediction의 특징

2 Level Prediction의 특징은 충분히 많은 명령어가 있는 프로그램을 처리할 때 성능향상을 기대할 수 있다는 것이다. 그 이유는 먼저 패턴을 만들어 주고 그 값을 통해서 또 테이블에 접근해서 History를 파악을 해야 하는데 이는 충분히 명령어의 수가 많지 않아서 테이블에 충분한 학습이 되지 않는다면 예측이 항상 어긋날 수 있다. 그래서 이 방식을 사용하려면 충분한 학습이 가능한 많은 명령어의 수가 필요하다.

2. Implementation

2.1 Requirement

Num	Requirement
1	**The Objective: compare performance of pipelined vs. Signal-cycle u-processor.** A) Your program should produce correct output. (and execution should be the same with codesemantics) B) Compare the number of CPU clock cycles for Signal-cycle execution and pipelined execution. Intheory, we can reduce CPU clock cycle time by 1/5 in 5-stage pipelined execution.
2	**Machine Initialization** A)Before the execution, the binary file is loaded into the MEMory. Note that MEMory can be a datastructure defined with large array. Read all the file content into your MEMory (Datastructure).Assume that all register values are all zero, except for RA, of which value is 0xFFFF:FFFF. Thus,when your PC becomes 0xFFFF:FFFF, your machine completes execution, and halts. Yourapplication is loaded to 0x0, and stack pointer is 0x1000000.
3	**Implementation requirements** A)You should implement five-stage pipelined MIPS processor emulator. The emulator shouldimplement IF, ID, EX, MEM, WB stages. B)For each stage, instruction execution has to be Latched. That is, you have to store the executionstate for each stages. Latches are the temporal storage that reMEMber the execution states for eachstages execution. C)You can have separated instruction MEMory and DataMEMory. D)The emulated processor runs with the emulated clock cycle. You need to generate clock, which isa variable. The clock cycle increases only after all the work dOne in the all the stages. Note thatmax. 5 different instructions can run at the same cpu clock cycle. E)You need to resolve Data Dependency among instructions. That is, you have to implement eitherstall, forwarding or register renaming. Forwarding is recommended as a basic implementation, butyou can optionally choose to implement stall. You should make sure that it works, as in the programorder (semantics). F)You need to resolve ControlDependency among instructions. According to the MIPS ISA, Onedelayed Branch slot is defined. Invalidation is basic implementation, but you can optionallyimplement Branch Prediction mechanisms.
4	**Program completion/terminal condition** A) At the end of the execution, you need to print out the calculated result value. We know the end ofexecution by moving PC to 0xFFFF:FFFF. Along with the calculated result, you should print outthe statistics of the execution. B) The final return value is stored in v0 (or r2) register. C) The print out statistics may include: the total # of instructions, # of MEMory operation instructions,# of register operation instructions, # of Branch instructions, # of not-Taken Branches, and # of jumpinstructions.
5	**Evaluation** A)Different credits are given for implementations, and demos. B)More credits are allotted for different (hardware-based) hazard resolving techniques such as BranchPrediction, forwarding, stalling, etc



<그림25> My Code Data Path

https://miro.com/app/board/uXjVOyDTOgk=/?share_link_id=624304882312

(그림이 흐려서 제가 만든 링크 첨부했습니다.)

2.2 Latch

```
struct IF_ID {
    int valid;
    int inst;
    int pc_4;
};

struct ID_EX {
    struct EX ex;
    struct M m;
    struct WB wb;

    int valid;
    int pc_4;
    int Rdata1;
    int Rdata2;
    int simm;
    int rs;
    int rd;
    int rt;
    int shamt;
    int imm;
};
```

<그림26> IF_ID, ID_EX Latch

```
struct EX_MEM {
    struct M m;
    struct WB wb;

    int valid;
    int pc_4;
    int becond;
    int aluResult;
    int writeData;
    int regDst;
    int addresultpc;
    int jrs;
    int simm;

};


struct MEM_WB {
    struct WB wb;

    int valid;
    int mReadData;
    int aluResult;
    int regDst;
    int jrs;
    int pc_4;

};
```

<그림27> EX_MEM, MEM_WB Latch

```
struct IF_ID latch1[2];
struct ID_EX latch2[2];
struct EX_MEM latch3[2];
struct MEM_WB latch4[2];
```

<그림28> Separate Input Output Latch

위에서 설명한 것과 같이 파이프라인 구조에서 명령어를 병렬로 처리를 해주려면 처리 단계를 나눠주고 그 단계에서 명령어를 처리 후 결괏값들을 저장해주는 Latch가 필요하다 이번 프로젝트에서는 나는 C 언어의 구조체를 이용하여 래치를 구분 지어 주었으며 그 안에 변수들을 통해서 각 래치에 어떤 값들이 저장되어야 하는 지를 직관적으로 코드를 구현하였다. 그리고 <그림28>과 같이 배열을 통해 input, output 래치를 구분 지어 주면서 구현상 값들을 이용하기 편하게 래치를 나누어 주었다. 이후 래치에 값이 있는지 없는지를 판단해주는 valid 비트를 지정해주어서 래치의 상태를 나타내 주는 값으로 이용하였다.

2.3 Control Signals

```
struct EX{
    //ex
    int RegDst;
    int ALUSrc;
    int ALUOp;
    int jr;
    int jal;
    int branch;
    int zero;
};

struct M {
    //m

    int memoryWrite;
    int memoryRead;
    int PCSrc1;
};

struct WB{
    //wb
    int memtoReg;
    int PCSrc2;
    int RegWrite;
    int j_and_I_toReg;

};
```

<그림29> Control Signal Structure

이 부분에서는 래치와 같이 각 단계에서 필요한 MUX 들을 처리하기 위해서 구현한 방식으로 래치와 같이 구조체를 이용하여 각 래치에서 <그림29>와 같이 선언을 해줌으로써 각 래치와 명령어 처리 단계에서 필요한 Control Signal 들을 사용할 수 있도록 해주었다.

```
int control(int opcode, int funct) {//function to initialize the control signal
    latch2[0].ex.RegDst = (opcode == 0);//r type instrution
    if (opcode == 0x2b)latch2[0].ex.RegDst = 1;
    latch2[0].ex.ALUSrc = (opcode != 0) && (opcode != 0x4) && (opcode != 0x5);
    latch2[0].ex.jr = (opcode == 0x0) && (funct == 0x08);//jr mux signal.
    latch2[0].ex.branch = (opcode == 0x4) || (opcode == 0x5);//branch.
    latch2[0].ex.ALUOp = opcode;
    latch2[0].ex.jal = (opcode == 0x3);

    latch2[0].m.memoryRead = opcode == 0x23;//load word
    latch2[0].m.memoryWrite = opcode == 0x2b;//store word
    latch2[0].m.PCSrc1 = (opcode == 0x2) || (opcode == 0x3); //jump or not



    latch2[0].wb.PCSrc2 = (opcode == 0x4) && (opcode == 0x5);//bcond statisfied
    latch2[0].wb.j_and_I_toReg = (opcode == 0x3);
    latch2[0].wb.memtoReg = opcode == 0x23;
    latch2[0].wb.RegWrite = (opcode != 0x4) && (opcode != 0x5) && (opcode != 0x2) && (opcode != 0x2b);



    if (opcode == 0x3) {// jump and link
        latch2[0].ex.RegDst = 2;
        latch2[0].wb.RegWrite = 1;
    }
    if (opcode == 0xd) {//or immediate
        latch2[0].ex.zero = 1;
    }
}
```

```
latch3[0].m.memoryRead = latch2[1].m.memoryRead;
latch3[0].m.memoryWrite = latch2[1].m.memoryWrite;
latch3[0].m.PCSrc1 = latch2[1].m.PCSrc1;


latch3[0].wb.memtoReg = latch2[1].wb.memtoReg;
latch3[0].wb.PCSrc2 = latch2[1].wb.PCSrc2;
latch3[0].wb.RegWrite = latch2[1].wb.RegWrite;
latch3[0].wb.j_and_I_toReg = latch2[1].wb.j_and_I_toReg;
```

<그림30> Handle the Control Signals

이후 이 값들은 명령어 DE 단계에서 Control Unit에서 처리해주어야 하니 각 부분에 필요한 값들을 opcode와 funct 으로 판단하고 초기화해주었다. 그 이후에 각각 이후 단계에 필요한 값들은 이전 단계가 끝날 때 값을 그대로 옮겨 주었다.

2.4 IF/DE/EX/MEM/WB

-IF

```
void fetch() {
    inst = memory[pc / 4];
    latch1[0].inst = inst;
    numOfInsts++;
    latch1[0].pc_4 = pc + 4;

    //AlwaysNotTaken
    //pc = pc + 4;
    /*
    //Alwaystaken

    if (AlwaysTakenPredict(pc)) {
        pc = BTB[pc];
    }
    else {
        pc = pc + 4;
    }
    */
    /*
    //BTFN
    if (BTFN(pc)) {
        pc = BTB[pc];
    }
    else {
        pc = pc + 4;
    }
    */

    /*
    //OneBitPredict
    if (OneBitBranchPredict(pc)) {
        pc = BTB[pc];
    }
    else {
        pc = pc + 4;
    }

    */
    /*
    //TwoBitPredict

    if(TwoBitBranchPredict(pc)){
        pc = BTB[pc];
    }
```

<그림31> Fetch 함수

```
    if (IR) {
        pc = JRaddress;
        IR = 0;
    }
    else if (Jump) {
        pc = Jumpaddress;
        Jump = 0;
    }

    printf(_Format: "pc updata : 0x%x\n", pc);
    //printf("inst: 0x%x\n", latch1[0].inst);
    latch1[0].valid = 1;
}
```

<그림32> PC 업데이트

IF/DE/EX/MEM/WB 5단계 중 IF 단계를 먼저 설명하겠다. Data Path 구조 <그림25> 번을 참고하면 먼저 Fetch 이전 단계에서 값을 예측하는 Predictor 들의 조건들과 그리고 PC값을 지정해주는 조건들을 전역변수로 처리하여 상황에 따라 PC값을 업데이트해주었다. 그 후 값을 래치에 저장하고 래치에 값이 있다고 판단을 해줄 수 있도록 valid 비트를 1로 업데이트해주면서 첫 번째 래치의 input부분(Latch1[0])에 값이 있다고 상태를 나타내 주었다.

-ID

```
void decode() {
    if (latch1[1].valid == 0)return;
    inst = latch1[1].inst;
    int rs = 0;
    int rd = 0;
    int rt = 0;
    unsigned int imm = 0;
    int jtarget = 0;
    unsigned int funct = 0;
    int shamt = 0;
    unsigned int temp_in = inst;
    int opcode = (inst >> 26) & 0x3f;  // get some opcode bit from the instruction

    rs = (inst >> 21) & 0x1f;
    rt = (inst >> 16) & 0x1f;
    rd = (inst >> 11) & 0x1f;
    if (opcode != 0)rd = -1;
    shamt = (inst >> 6) & 0x1f;
    funct = inst & 0x3f;

    imm = temp_in & 0xffff;
    if (opcode == 0x2 || opcode == 0x3) {//jump and jump and link( j type instruction)
        jtarget = temp_in & 0x3fffffff;
        jumpinsts++;
    }

    control(opcode, funct);//initialize the control signals by opcode.->explain

    //reg
    reg(reg1:rs, reg2:rt);

    if (opcode == 0x2 || opcode == 0x3) {// shift 2bits and add the current upper pc 4bit to use jump address
        int tempj = jtarget << 2;
        Jumpaddress = (tempj + (latch1[1].pc_4 & (0xf << 28)));
        Jump = 1;
    }
    int temp = imm;

    //sign extend
    if (temp >> 15 == 0) {//imm sign extend
        latch2[0].simm = 0xffffffff & imm;
```

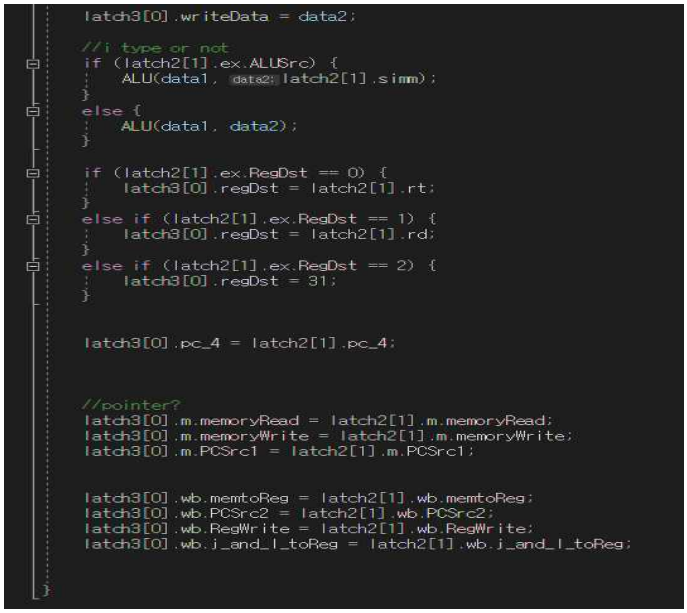
<그림33> DE 함수

```
    //sign extend
    if (temp >> 15 == 0) {//imm sign extend
        latch2[0].simm = 0xffffffff & imm;
    }
    else {
        latch2[0].simm = (0xffff << 16) | imm;
    }

    if (latch2[0].ex.zero) {//zeroextend of ori
        int zeroextend = imm;
        latch2[0].simm = 0xffffffff & imm;


    }

    latch2[0].rs = rs;
    latch2[0].rd = rd;
    latch2[0].rt = rt;
    latch2[0].shamt = shamt
    latch2[0].imm = imm;
    latch2[0].valid = 1;

    latch2[0].pc_4 = latch1[1].pc_4;
    return;
}
```

<그림34> DE 함수에서 다음 래치로 값을 저장하는 방법

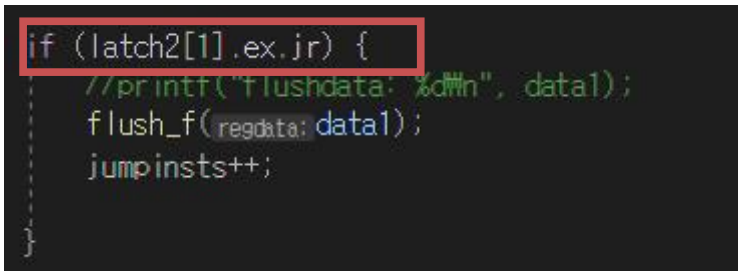
Decode 단계에서는 먼저 첫 번째 래치에 상태를 보고 처리를 해주어야 한다. 즉 첫 번째 output 래치에 값이 있는지 없는지를 판단 후에 함수를 실행시켜야 한다. 그래서 이전에 설명했던 valid를 통해서 값이 있는지 없는지 판단할 수 있으니 조건이 성립하면 DE 단계가 실행되도록 해주었다. 그 이후에 그다음 래치인 두 번째 래치에 필요한 요소들의 값들을 저장해주고 다시 valid 비트를 1로 만들어 주면서 값이 저장되었다는 상태를 나타내 주었다.

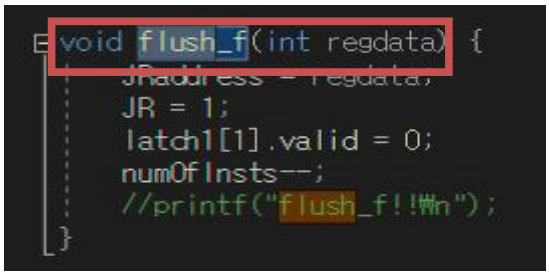
-EX



<그림35> EX 함수 <그림36> EX 함수에서 다음 래치로 값을 저장하는 방법

이전 단계와 마찬가지로 valid 비트가 1인 상태 즉 래치에 값이 있으면 함수를 실행하도록 해주었고 여기서는 EX Control Signal 레지스터를 사용하여 조건문에 이용하였다. 이는 Data Path 상에서 MUX 들을 조정하는 Signal을 이렇게 구현하였다.

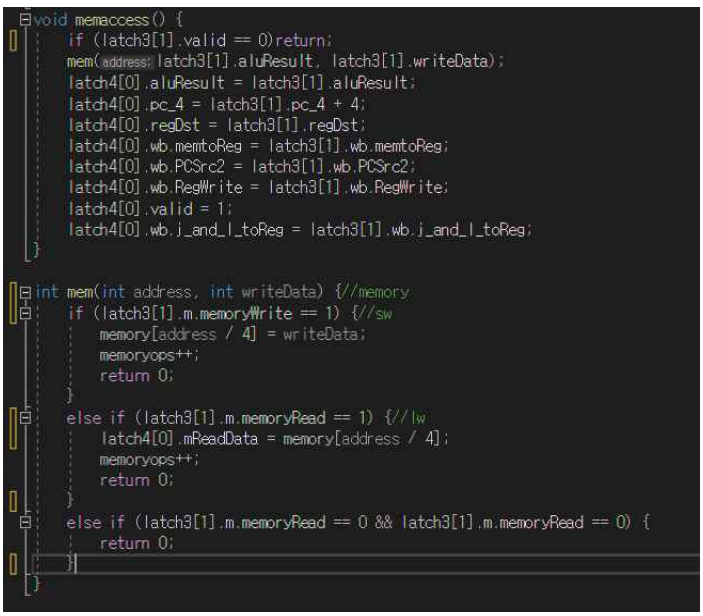


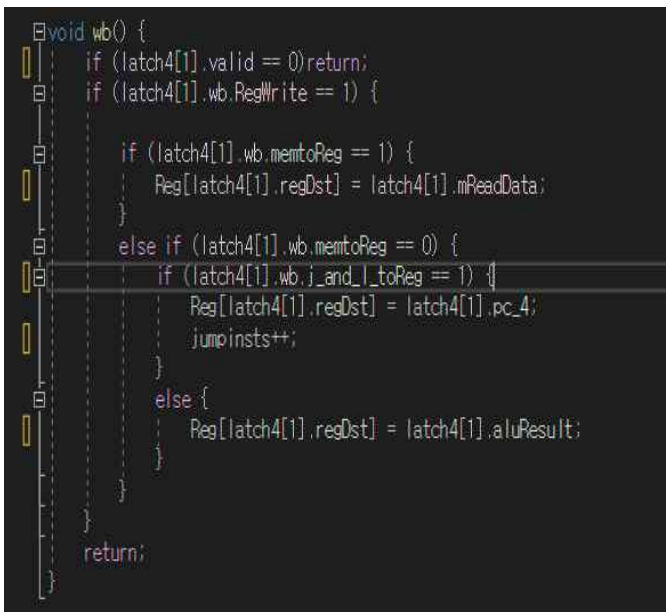


<그림37, 38> JR 명령어면 이전 명령어를 제거하는 조건과 함수

여기서는 JR을 EX 단계에서 처리하는 방법을 나타내는 코드이다. 이전 설명 <그림15>을 보면 JR을 해주려면 이전에 처리되는 명령어를 제거하는 flush 작업이 필요한데 그 과정을 처리하기 위한 코드이다. Control Signal : JR 조건에 따라서 만약 JR 명령어라면 flush_f 함수를 통해서 전역변수를 1로 업데이트 해주고 R[rs](data1)값을 JR이 목표로 하는 주소를 JR address 전역변수에 저장해주었다. 이후 래치에 저장된 요소들을 각각 제거해주는게 아니라 valid 비트를 이용해서 래치에 값이 저장돼있지 않다고 표현을 해주었다.

-MEM&WB





<그림39> MEM 함수 <그림40> WB 함수

이 단계는 이전 싱글 사이클 방식과 비슷하게 각각 Control Signal 값들을 통해서 각각 다르게 명령어를 처리해주었다.

2.5 Data Forwarding

```
int ForwardA(int rs, int  regDst1, int regDst2) {
    if (rs == regDst1)return 1;
    else if (rs == regDst2)return 2;
    else return 0;
}
int ForwardB(int rt, int  regDst1, int regDst2) {
    if (rt == regDst1)return 1;
    else if (rt == regDst2)return 2;
    else return 0;
}
```

<그림41> Forward MUX

1.4.5 Forwarding/Bypass의 <그림11> 번에서와 같은 조건에 따라 Forwarding을 해주었다. <그림41>처럼 먼저 EX 단계에서 ALU로 넘어가는 값들을 지정해주는 조건의 함수를 만들어 주었다.

```
switch ((int)ForwardA(latch2[1].rs, latch3[1].regDst, latch4[1].regDst)) {
case 0:

    break;

case 1:
    data1 = latch3[1].aluResult;

    break;
case 2:

    if (latch4[1].wb.RegWrite == 1) {
        if (latch4[1].wb.memtoReg == 1) {
            data1 = latch4[1].mReadData;

        }
        else if (latch4[1].wb.memtoReg == 0) {
            data1 = latch4[1].aluResult;

        }
        else if (latch4[1].wb.j_and_l_toReg == 1) {
            data1 = latch4[1].pc_4;


        }
    }

    break;
}
```

```
switch ((int)ForwardB(latch2[1].rt, latch3[1].regDst, latch4[1].regDst)) {
case 0:

    break;

case 1:

    data2 = latch3[1].aluResult;

    break;
case 2:
    if (latch4[1].wb.RegWrite == 1) {
        if (latch4[1].wb.memtoReg == 1) {
            data2 = latch4[1].mReadData;

        }
        else if (latch4[1].wb.memtoReg == 0) {
            data2 = latch4[1].aluResult;

        }
        else if (latch4[1].wb.j_and_l_toReg == 1) {
            data2 = latch4[1].pc_4;


        }
    }
    break;
}
```

<그림42> Chose the Operands of ALU

이전 함수를 이용하여 ALU에 들어가는 피연산자 값들을 지정해준다. 만약 Data Dependency가 발생하지 않는다면 case: 0 번으로 빠지면서 일반적인 명령어 처리를 해주지만 만약 거리가 1, 2인 Dependency가 발생한다면 데이터를 각각 단계의 처리가 끝난 값들을 가지고 와서 명령어를 연산 처리해준다. 거리가 3인 Dependency는 파이프라인 구조에서 LW에서 발생하지만, 이는 MIPS compiler gcc에서 처리해주어서 Dependency가 발생하는 LW 사이에 nop 명령어를 추가해줌으로써 따로 구현하지 않아도 처리할 수 있었다.

Fowarding을 구현하면서 코드상 발생하는 문제는 Signal cycle 구조로 한다면 ex 이후에 MEM,WB 단계가 실행되는데 그렇다면 MEM, WB 처리가 끝난 값을 이용하는 forwarding 방식에 어려움이 있다고 판단하였다. 그래서 이를 보완하기 위해서 새로운 방식을 구현하였다.

```
//begin execution loop
while (1) {
    //in the loop..
    memaccess();
    wb();

    if (pc == -1) {
        break;
    }
    fetch();
    decode();
    execute();

    Switch();
```

<그림43> Handle the Problem

Data Forwarding 방식은 MEM,WB 단계 이후에 처리된 값들을 사용해야 하는 문제가 있어서 같은 사이클 안에서 처리되는 단계의 순서를 변경해 주었다. MEM, WB 단계를 먼저 실행해 주고 Fetch, Decode, Execute 단계를 실행해 주고 그 이후에 사이클 수를 증가시켜 주면서 같은 사이클 안에서 구현이 되도록 하면서 이는 Data Forwarding 방식을 조금 더 쉽게 처리할 수 있도록 해주었다.

2.6 Branch Prediction

```
void Flush1() {
    latch1[0].valid = 0;
    latch1[1].valid = 0;
}
void Flush2() {
    latch2[0].valid = 0;
    latch2[1].valid = 0;
}
```

```
//brach con
int b_taken = latch2[1].ex.branch & becond;
int b_address = (latch2[1].pc_4) + (latch2[1].simm << 2);

if (latch2[1].ex.ALUOp == 0x4 || latch2[1].ex.ALUOp == 0x5) {

    //AlwaysNotTakenPredictCheck(b_address, b_taken, latch2[1].pc_4 - 4);
    //AlwaysTakenPredictCheck(b_address, b_taken, latch2[1].pc_4-4);
    //BTFNPredictCheck(b_address, b_taken, latch2[1].pc_4 - 4);
    //OneBitBranchPredictCheck(b_address, b_taken, latch2[1].pc_4-4);
    //TwoBitBranchPredictCheck(b_address, b_taken, latch2[1].pc_4-4);
    //TwoLevelPredictCheck(b_address, b_taken, latch2[1].pc_4 - 4);
    LocalTwoLevelPredictCheck(b_address, b_taken, latch2[1].pc_4 - 4);
}
```

<그림44> Mis Predict 상황에서 이전 명령어를 제거하는 함수
<그림45> Branch 명령어이면 조건에 따라 예측을 확인하는 함수

Prediction 종류마다의 구조에 들어가기 이전에 공통으로 사용되는 부분을 먼저 설명하겠다. Prediction을 구현하면서 먼저 예측이 실패하는 경우 이전에 처리되고 있는 명령어들을 제거해주는 flush 작업이 필요하다. 이는 예측이 실패하는 경우 꼭 필요한 작업이고 공통으로 사용되어서 따로 함수를 구현해서 처리해주었다. 그리고 이는 valid 비트를 이용하여서 값이 없다고 처리를 해주었다.

그리고 EX 단계에서 예측한 Prediction의 주소가 Branch 명령어가 EX 단계에서 연산 처리가 되었을 때 예측이 맞았는지 틀렸는지를 판단해주는 함수들을 Branch 명령어들이 처리될 때마다 실행되도록 해주었다.

2.6.1 Always not Taken

```
void AlwaysNotTakenPredictCheck(int b_address, int b_taken, int bpc) {
    if (!b_taken) {//not taken
        predict_correction += 1;
    }
    else {// taken
        Flush1();
        Flush2();
        pc = b_address;
        mis_predict += 1;
    }
}
```

<그림46> Always not Taken Prediction Check

```
pc = pc + 4;
```

Always not Taken 방식은 항상 Branch 다음 명령어를 PC+4값으로 예측하는 방식이다. 그래서 Branch 이후 명령어를 지정해주는 함수를 따로 구현하지 않고 PC+4값을 하는 일반적인 처리 방식으로 다음 주소를 업데이트해주었다. 이후 EX 단계에서 그 주소가 맞는지 확인해주는 함수를 위의 그림과 같이 구현해주었으며 예측이 실패했을 때는 이전 명령어를 제거해주고 PC값을 목표 주소로 지정해주었다.

2.6.2 Always Taken

```
int AlwaysTakenPredict(int pc) {
    if (BTB[pc]!=0) {
        return 1;
    }
    else {
        return 0;
    }
}
```

<그림47> Always Taken Predicton

```
//Alwaystaken
if (AlwaysTakenPredict(pc)) {
    pc = BTB[pc];
}
else {
    pc = pc + 4;
}
```

<그림48> Prediction 결과에 따라 주소 예측

이번 방식은 Always not Taken 방식과는 다르게 항상 Branch 명령어가 성립하는 것을 가정하고 예상하여 명령어를 처리하는 방식이다. 이론상으로는 컴파일하면서 Branch 명령어에 힌트를 줄 수 있는 비트를 설정하고 이후 프로그램을 실행하면서 주솟값을 업데이트해 준다. 하지만 코드를 구현하면서 컴파일 타임에서 힌트를 주는 비트를 지정해주지 못함으로 이번 프로젝트에서는 첫 Branch 명령어만 예측이 실패하도록 지정해주고 이후 Branch에서는 항상 그 목표 주소로 PC 주솟값을 예측하도록 해주었다.

이 방식을 사용하기 위해서 주솟값을 저장해주는 BTB를 사용해 주었으면 인덱스값을 PC값을 통해서 값을 불러왔다.

```
void AlwaysTakenPredictCheck(int b_address, int b_taken, int bpc) {
    if (BTB[bpc] != 0) {//Predict to taken
        if (b_taken) {//taken
            predict_correction += 1;
        }
        else {//not taken
            Flush1();
            Flush2();
            pc = bpc + 4;
            mis_predict += 1;
        }
    }
    else {
        if (!b_taken) {//not taken
            predict_correction += 1;
        }
        else {// taken
            Flush1();
            Flush2();
            BTB[bpc] = b_address;
            pc = b_address;
            mis_predict += 1;
        }
    }
}
```

<그림49> Always Taken Predicton Check

다음 명령어를 예측하고 그 이후 EX 단계에서 명령어의 조건이 부합하는지 확인을 해주는 함수를 지정해주었다. 먼저 BTB에 값이 있는지 없는지로 첫 조건을 구분 지어서 이 Branch 명령어가 처음 실행이 되는지 판단을 해주었다. 만약 명령어가 처음 실행하는 것이고 조건이 성립한 면 BTB에 그 목표 주솟값을 저장해주고 PC값을 Branch 명령어의 목표 주소로 업데이트시켜준다. 그리고 이전에 실행시키고 있던 명령어들을 제거하는 flush 작업을 거친다. 그렇게 프로그램을 실행하다가 이후에 Branch 명령어의 조건이 성립하지 않는다면 Branch 명령어의 다음 값으로 주소를 업데이트시켜준다.

2.6.3 BTFN

```
int BTFN(int pc) {
    if (BTB[pc] != 0) {
        if (BTB[pc] < pc)return 1;
        return 0;
    }
    else {
        return 0;
    }
}
```

<그림50> BTFN(Backward Taken Forward Not Taken)

```
//BTFN
if (BTFN(pc)) {
    pc = BTB[pc];
}
else {
    pc = pc + 4;
}
```

<그림51> Prediction 결과에 따라 주소 예측

BTFN 기술은 Backward Taken Forward Not Taken으로 만약 Branch 명령어가 목표로 하는 주솟값이 현재 Branch 명령어보다 이전의 주소를 목표로 한다면 Branch 명령어가 연산 처리하면 조건이 성립한다고 예측하여 다음 주소값을 Branch의 목표 주소로 설정한다. 하지만 그 반대일 경우에서 Not Taken 해주면서 그다음 명령어를 실행하도록 PC값을 업데이트시켜준다. 이것도 이전 방식처럼 컴파일 단계에서 명령어에 힌트를 주지만 이번 구현에서는 실행 통해서 조건이 성립했을 때만 값을 저장해주었기 때문에 조건이 성립하는 첫 번째 Branch 명령어는 예측이 실패하였다.

```
void BTFNPredictCheck(int b_address, int b_taken, int bpc) {
    if (BTB[bpc] != 0) {
        if (b_taken) {
            predict_correction += 1;
        }
        else {
            Flush1();
            Flush2();
            pc = bpc + 4;
            mis_predict += 1;
        }
    }
    else {
        if (!b_taken) {//not taken
            predict_correction += 1;
        }
        else {// taken
            Flush1();
            Flush2();
            if(b_address<bpc)BTB[bpc] = b_address;
            pc = b_address;
            mis_predict += 1;
        }
    }
}
```

<그림52> BTFN Prediction Check

이제 예측 부분을 지나서 그 예측된 주소와 이후 Branch 명령어가 조건에 맞는 주소를 비교해주는 BTFN Prediction Check 함수를 만들어 주었다. 이 방식에서도 주솟값을 저장해주는 BTB를 사용하였으며 조건이 성립하고 또 그 값이 뒤에 있는 명령어를 목표로 한다면 그 값을 BTB에 저장하도록 설정해 주었다.

2.6.4 One Bit

```
int OneBitBranchPredict(int pc) {
    int hit = 0;
    int taken = 0;
    if (BTB[pc] != 0) {
        hit = 1;
        switch ((int)B_Taken[pc])
        {
        case 0x0:
            taken = 0;
            break;
        case 0x1:
            taken = 1;
            break;
        }
    }
    return hit && taken;
}
```

<그림53> One Bit Prediction

```
//OneBitPredict
if (OneBitBranchPredict(pc)) {
    pc = BTB[pc];
}
else {
pc = pc + 4;
```

<그림54> Prediction 결과에 따라 주소 예측

One Bit 방식 이후부터는 static 방식과는 다르게 Run Time Predictor이다. 이 말인즉슨 프로그램을 실행하면서 이전 Branch 명령의 결과에 따라 예측을 하는 방식이다. 그래서 One Bit 방식부터는 추가적인 하드웨어 BTB, BHT들이 필요하다. One Bit 방식에는 추가적인 하드웨어를 PC값을 인덱스로 활용하여 배열로 처리를 하였다.
<그림18>의 구조를 따랐으며 One Bit 방식은 먼저 PC값을 통해서 BTB에 주솟값이 있는지를 확인하고 만약 값이 있다면 hit = 1로 반환한다. 그리고 BHT에서 이전 Branch 명령어의 History를 1개의 비트로 두고 만약 1이면 Taken=1로 반환하여 이후 PC값을 지정해주는 MUX 의 Control Signal로 사용하여 값을 지정해주었다.

```
void OneBitBranchPredictCheck(int b_address, int b_taken, int bpc) {
    if (BTB[bpc] != 0) {
        if (B_Taken[bpc] == 0x1) {//predict to taken
            if (b_taken) {//taken
                B_Taken[bpc] += 1;
                if (B_Taken[bpc] > 0x1) {
                    B_Taken[bpc] = 0x1;
                }
                predict_correction += 1;
            }
            else {//not taken
                Flush1();
                Flush2();
                B_Taken[bpc] -= 1;
                if (B_Taken[bpc] < 0x0) {
                    B_Taken[bpc] = 0x0;
                }
                pc = bpc+4;
                mis_predict+=1;
            }
        }
        else {//predict to not taken
            if (!b_taken) {//not taken
                B_Taken[bpc] -= 1;
                if (B_Taken[bpc] < 0x0) {
                    B_Taken[bpc] = 0x0;
                }
                predict_correction += 1;
            }
            else {// taken
                Flush1();
                Flush2();
                B_Taken[bpc] += 1;
                if (B_Taken[bpc] > 0x1) {
                    B_Taken[bpc] = 0x1;
                }
                pc = b_address;
                mis_predict+=1;
            }
        }
```

```
    else {
        if (!b_taken) {//not taken
            //printf("predict₩n");
            B_Taken[bpc] -= 1;
            if (B_Taken[bpc] < 0x0) {
                B_Taken[bpc] = 0x0;
            }
            predict_correction += 1;
        }
        else {// taken
            Flush1();
            Flush2();
            //printf("mis predict₩n");
            B_Taken[bpc] += 1;
            if (B_Taken[bpc] > 0x1) {
                B_Taken[bpc] = 0x1;
            }
            BTB[bpc] = b_address;
            pc = b_address;
            mis_predict+=1;

        }
    }
}
```

<그림55, 56> One Bit Prediction Check

이후 예측한 주소와 Branch 명령어의 목표 주소가 같은지 판단하는 함수를 구현했으며 이전 방식과는 다르게 만약 명령어의 연산 조건이 성립하면 BHT에 있는 값에 1을 더했다. 그리고 1 Bit로 값을 나타내야 하므로 값이 1이 넘어가면 다시 1값을 저장하도록 지정해주었다. 반대로 연산 조건이 성립하지 않으면 이전 값에서 -1을 빼주었다. 이것 또한 값이 1bit 표현 방식임으로 0보다 작아질 때는 다시 0을 저장하도록 해주었다.

2.6.5 Two Bit

```
int TwoBitBranchPredict(int pc) {
    int hit = 0;
    int taken = 0;
    if (BTB[pc] != 0) {
        hit = 1;
        switch ((int)B_Taken[pc])
        {
        case 0:
        case 1:
            taken = 0;
            break;
        case 2:
        case 3:
            taken = 1;
            break;
        }
    }
    return hit && taken;
}
```

<그림57> One Bit Prediction

```
///TwoBitPredict

if(TwoBitBranchPredict){
    pc = BTB[pc];
}
else {
    pc = pc + 4;
}
```

<그림58> Prediction 결과에 따라 주소 예측

Two Bit 방식은 구현 방식은 One Bit과 비슷하지만, 이론상으로 1개의 비트로만 history로 나타내면 발생하는 단점을 보완하려고 2개의 비트로 Branch의 History를 나타내 주었다. 그래서 이 방식에서는 BHT에서 값을 확인할 때 2진수라고 생각을 했을 경우 00 값과 01 값은 조건이 성립하지 않는다고 예측하고 10, 11 값은 조건이 성립한다고 예측을 하였다. 그래서 10진수 값으로 0, 1 값은 성립하지 않게 예측하고 2, 3은 예측하도록 구현을 해주었다.

```
void TwoBitBranchPredictCheck(int b_address, int b_taken, int bpc) {
    if (BTB[bpc] != 0) {
        if (B_Taken[bpc] >= 2) {//predict to taken
            if (b_taken) {//taken
                B_Taken[bpc] += 1;
                if (B_Taken[bpc] > 3) {
                    B_Taken[bpc] = 3;
                }
                predict_correction += 1;
            }
            else {//not taken
                Flush1();
                Flush2();
                B_Taken[bpc] -= 1;
                if (B_Taken[bpc] < 0x00) {
                    B_Taken[bpc] = 0x00;
                }
                pc = bpc + 4;
                mis_predict += 1;
            }
        }
        else {//predict to not taken
            if (!b_taken) {//not taken
                B_Taken[bpc] -= 1;
                if (B_Taken[bpc] < 0x00) {
                    B_Taken[bpc] = 0x00;
                }
                predict_correction += 1;
            }
            else {// taken
                Flush1();
                Flush2();
                B_Taken[bpc] += 1;
                if (B_Taken[bpc] > 3) {
                    B_Taken[bpc] = 3;
                }
                pc = b_address;
                mis_predict += 1;
            }
        }
    }
    else {
        if (b_taken) {
            Flush1();
            Flush2();
            B_Taken[bpc] += 1;
            if (B_Taken[bpc] > 3) {
                B_Taken[bpc] = 3;
            }
            BTB[bpc] = b_address;
            pc = b_address;
            mis_predict += 1;
        }
    }
}
```

<그림59, 60> Two Bit Prediction Check

Branch 명령어의 연산 처리에 따라 예측을 하였는지 확인해주는 코드를 작성했다. 1bit와 같이 첫 번째 Branch 명령어 인지 아닌지를 BTB에 값이 있는지 없는지로 판단을 했으면 이후 값이 있었을 때는 BHT(B_Taken[]) 의 2bit 상태에 따라서 예측해주었다. 이후 Branch 명령어의 연산 처리 결과(b_taken)에 따라 예측이 실패했을 경우와 성공했을 경우들의 조건을 나눠주어서 각각 상황에 따라 다른 처리를 해주었다.

2.6.6 Global Two Level

```
int TwoLevelPredict(int GHR, int pc) {
    int hit = 0;
    int taken = 0;
    if (BTB[pc] != 0) {
        hit = 1;
        switch (B_Taken[pc ^ GHR])
        {
        case 0:
        case 1:
            taken = 0;
            break;
        case 2:
        case 3:
            taken = 1;
            break;
        }
    }
    return hit && taken;
}
```

<그림61> Two Level Prediction

Two Level Prediction의 2가지 종류 중에서 Gshare 방식을 선택했으며 BHT의 인덱스 값을 결정할 때 GHR의 값과 PC값을 XOR 연산 시켜주어 그 값을 사용하였다. 2 Bit Predictor와 마찬가지로 BHT에서 이전 Branch 명령어들의 상태는 2 Bit로 나타내 주었으며 2진수로 00, 01 값이면 예측은 하지 않게 10, 11 값이면 예측을 하도록 값을 반환해 주었다.

```
void TwoLevelPredictCheck(int b_address, int b_taken, int bpc) {

    if (BTB[bpc] != 0) {
        if (B_Taken[bpc^GHR] >= 2) {//predict to taken
            if (b_taken) {//taken

                B_Taken[bpc ^ GHR] += 1;
                if (B_Taken[bpc ^ GHR] > 3) {
                    B_Taken[bpc ^ GHR] = 3;
                }
                predict_correction += 1;
            }
            else {//not taken
                Flush1();
                Flush2();
                B_Taken[bpc ^ GHR] -= 1;
                if (B_Taken[bpc ^ GHR] < 0x00) {
                    B_Taken[bpc ^ GHR] = 0x00;
                }
                pc = bpc + 4;
                mis_predict += 1;
            }
        }
```

```
        else {//predict to not taken
            if (!b_taken) {//not taken
                B_Taken[bpc ^ GHR] -= 1;
                if (B_Taken[bpc ^ GHR] < 0x00) {
                    B_Taken[bpc ^ GHR] = 0x00;
                }
                predict_correction += 1;
            }
            else {// taken
                Flush1();
                Flush2();
                B_Taken[bpc ^ GHR] += 1;
                if (B_Taken[bpc ^ GHR] > 3) {
                    B_Taken[bpc ^ GHR] = 3;
                }
                pc = b_address;
                mis_predict += 1;
            }
        }
    }
    else {
        if (b_taken) {
            Flush1();
            Flush2();
            B_Taken[bpc ^ GHR] += 1;
            if (B_Taken[bpc ^ GHR] > 3) {
                B_Taken[bpc ^ GHR] = 3;
            }
            BTB[bpc] = b_address;
            pc = b_address;
            mis_predict += 1;
        }
    }
    GHR << 1;
    GHR = GHR | b_taken;
}
```

<그림62, 63> <그림61> Two Level Prediction Check

이후 예측된 값이 맞았는지 판단하는 함수에서는 two Bit에서 확인해주었던 방식을 비슷하게 이용했으며 PHT(B_Taken)의 인덱스값을 GHR과 PC값을 XOR 연산해 준 값을 사용하였다. 그 후 GHR의 마지막 비트를 새로운 Branch 명령어의 연산 결과 비트로 추가하여 새로운 패턴을 만들어 다시 GHR에 저장해주었다.

2.6.7 local two level

```
int LocalTwoLevelPredict(int pc) {
    int hit = 0;
    int taken = 0;
    if (BTB[pc] != 0) {
        hit = 1;
        switch (B_Taken[BPHT[pc]])
        {
        case 0:
        case 1:
            taken = 0;
            break;
        case 2:
        case 3:
            taken = 1;
            break;
        }
    }
    return hit && taken;
}
```

<그림64> Local Two Level Prediction

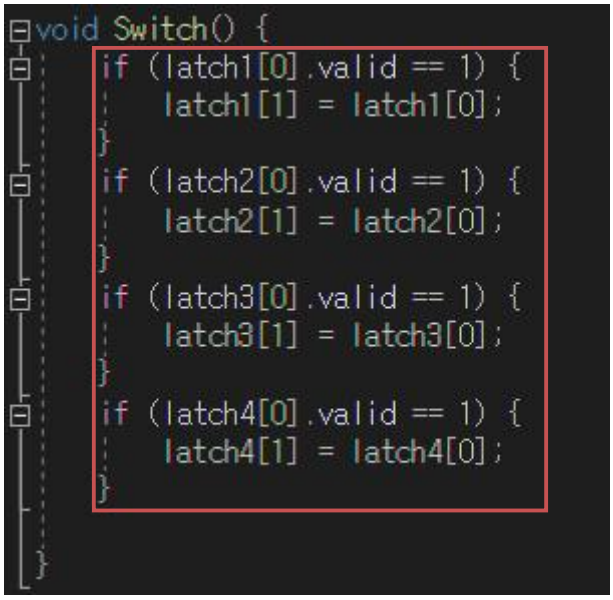
```
void LocalTwoLevelPredictCheck(int b_address, int b_taken, int bpc) {

    if (BTB[bpc] != 0) {
        if (B_Taken[BPHT[bpc]] >= 2) {//predict to taken
            if (b_taken) {//taken

                B_Taken[BPHT[bpc]] += 1;
                if (B_Taken[BPHT[bpc]] > 3) {
                    B_Taken[BPHT[bpc]] = 3;
                }
                predict_correction += 1;
            }
            else {//not taken
                Flush1();
                Flush2();
                B_Taken[BPHT[bpc]] -= 1;
                if (B_Taken[BPHT[bpc]] < 0x00) {
                    B_Taken[BPHT[bpc]] = 0x00;
                }
                pc = bpc + 4;
                mis_predict += 1;
            }
        }
        else {//predict to not taken
            if (!b_taken) {//not taken
                B_Taken[BPHT[bpc]] -= 1;
                if (B_Taken[BPHT[bpc]] < 0x00) {
                    B_Taken[BPHT[bpc]] = 0x00;
                }
                predict_correction += 1;
            }
            else {// taken
                Flush1();
                Flush2();
                B_Taken[BPHT[bpc]] += 1;
                if (B_Taken[BPHT[bpc]] > 3) {
                    B_Taken[BPHT[bpc]] = 3;
                }
                pc = b_address;
                mis_predict += 1;
            }
        }
    }
```

```
    else {
        if (b_taken) {
            Flush1();
            Flush2();
            B_Taken[BPHT[bpc]] += 1;
            if (B_Taken[BPHT[bpc]] > 3) {
                B_Taken[BPHT[bpc]] = 3;
            }
            BTB[bpc] = b_address;
            pc = b_address;
            mis_predict += 1;
        }
    }
    BPHT[bpc] << 1;
    BPHT[bpc] |= b_taken;
}
```

<그림65, 66> Local Two Level Prediction Check

이번 방식에서는 추가적인 하드웨어 BPHT(Branch pattern History table)를 추가로 구현을 해주었다. 이는 Branch 명령어마다 다른 패턴을 저장할 수 있도록 해주기 위한 하드웨어로 각각의 Branch 명령어마다 패턴을 저장하여 그 값을 인덱스값으로 활용하여 통해서 PHT(B_Taken)에 접근한다.

2.7 Switch



<그림67> Switch the Data output Latch from input Latch

이제 한 사이클이 종료되고 래치의 input 값에 저장되어 있던 값들을 output 래치로 넘겨야 한다. 이는 래치에서 값을 이용하고 저장하는 방식을 편하게 하기 위함이고 또한 이후의 사이클에서 그 값을 이용해야 하므로 이렇게 값들을 이동시켜주는 Switch 함수로 구현하였다.

3. Result

-Always not Taken

```
*****************************************
Cycle: 10
Result -> R[2]: 0
Number of instructions: 9
Number of memory access instructions: 2
Number of Register ops: 9
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
*****************************************
```

simple.bin

```
*****************************************
Cycle: 12
Result -> R[2]: 100
Number of instructions: 11
Number of memory access instructions: 4
Number of Register ops: 11
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
*****************************************
```

simple2.bin

```
*****************************************
Cycle: 1535
Result -> R[2]: 5050
Number of instructions: 1332
Number of memory access instructions: 613
Number of Register ops: 1332
Number of branch instruction: 102
Number of jump instruction: 108
Predict correct : 1  , mis predict : 101 ,total predict: 102
Accurate : 0
*****************************************
```

simple3.bin

```
*****************************************
Cycle: 294
Result -> R[2]: 55
Number of instructions: 265
Number of memory access instructions: 100
Number of Register ops: 275
Number of branch instruction: 10
Number of jump instruction: 41
Predict correct : 1  , mis predict : 9 ,total predict: 10
Accurate : 10
*****************************************
```

simple4.bin

```
*****************************************
Cycle: 3171
Result -> R[2]: 55
Number of instructions: 2953
Number of memory access instructions: 1095
Number of Register ops: 3062
Number of branch instruction: 109
Number of jump instruction: 437
Predict correct : 55  , mis predict : 54 ,total predict: 109
Accurate : 50
*****************************************
```

fib.bin

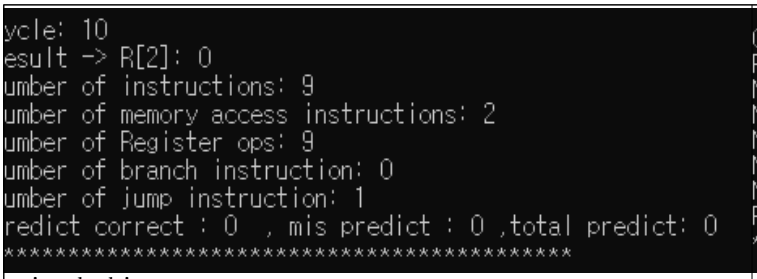
```
*****************************************
Cycle: 1292
Result -> R[2]: 1
Number of instructions: 1164
Number of memory access instructions: 486
Number of Register ops: 1201
Number of branch instruction: 73
Number of jump instruction: 185
Predict correct : 28  , mis predict : 45 ,total predict: 73
Accurate : 38
*****************************************
```

gcd.bin

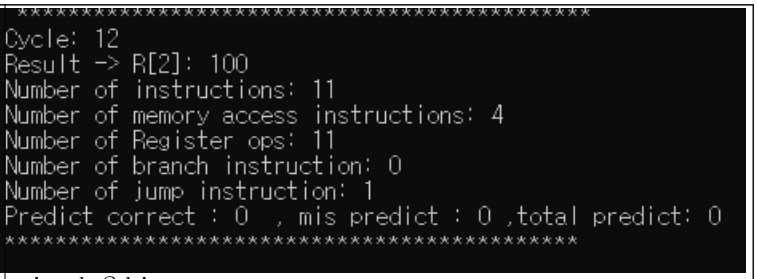
```
*****************************************
Cycle: 27430471
Result -> R[2]: 85
Number of instructions: 23372810
Number of memory access instructions: 7116606
Number of Register ops: 23372810
Number of branch instruction: 2029699
Number of jump instruction: 2028934
Predict correct : 869  , mis predict : 2028830 ,total predict: 2029699
Accurate : 0
*****************************************
```

input4.bin

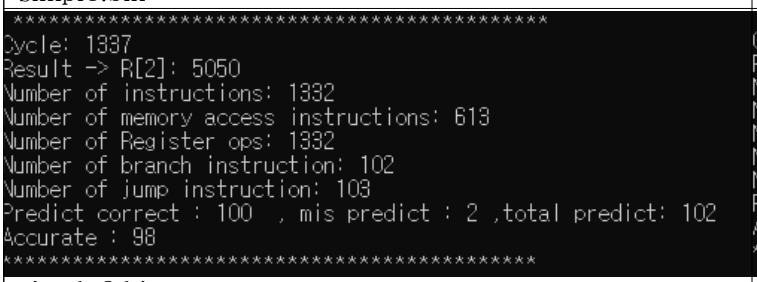
Always Taken



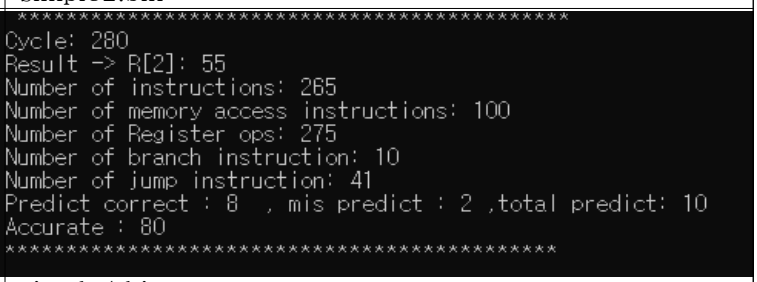
simple.bin



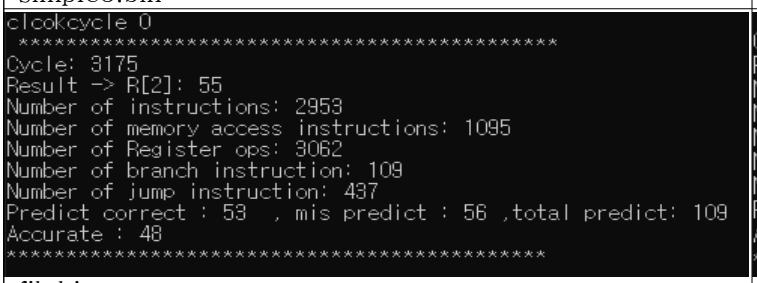
simple2.bin



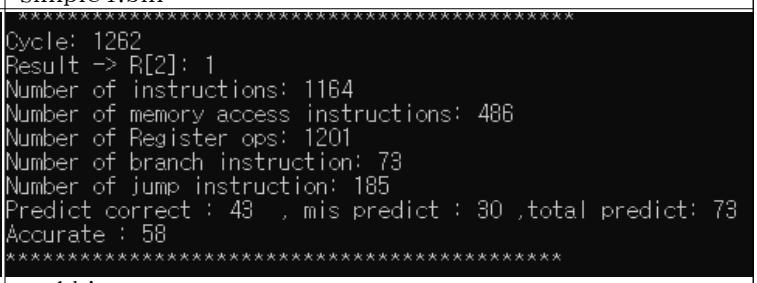
simple3.bin



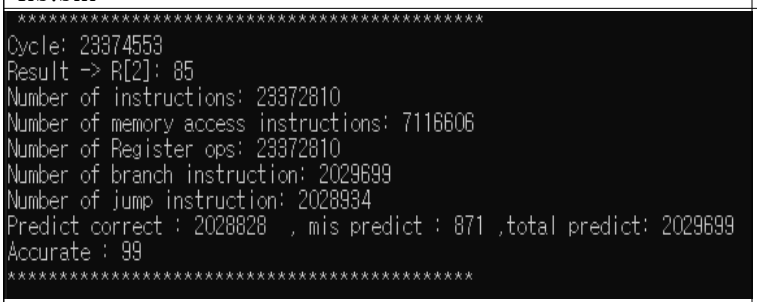
simple4.bin



fib.bin



gcd.bin



input4.bin

BTFN

```
**************************************************
Cycle: 10
Result -> R[2]: 0
Number of instructions: 9
Number of memory access instructions: 2
Number of Register ops: 9
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
**************************************************
```

simple.bin

```
**************************************************
Cycle: 12
Result -> R[2]: 100
Number of instructions: 11
Number of memory access instructions: 4
Number of Register ops: 11
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
**************************************************
```

simple2.bin

```
**********************************************
Cycle: 1337
Result -> R[2]: 5050
Number of instructions: 1332
Number of memory access instructions: 613
Number of Register ops: 1332
Number of branch instruction: 102
Number of jump instruction: 103
Predict correct : 100  , mis predict : 2 ,total predict: 102
Accurate : 98
**********************************************
```

simple3.bin

```
**********************************************
Cycle: 294
Result -> R[2]: 55
Number of instructions: 265
Number of memory access instructions: 100
Number of Register ops: 275
Number of branch instruction: 10
Number of jump instruction: 41
Predict correct : 1  , mis predict : 9 ,total predict: 10
Accurate : 10
**********************************************
```

simple4.bin

```
**********************************************
Cycle: 3171
Result -> R[2]: 55
Number of instructions: 2953
Number of memory access instructions: 1095
Number of Register ops: 3062
Number of branch instruction: 109
Number of jump instruction: 437
Predict correct : 55  , mis predict : 54 ,total predict: 109
Accurate : 50
**********************************************
```

fib.bin

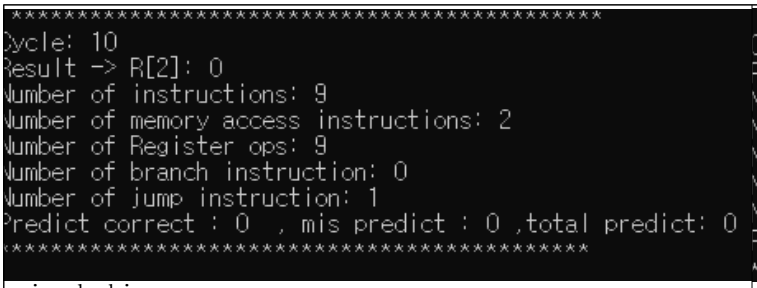
```
**********************************************
Cycle: 1292
Result -> R[2]: 1
Number of instructions: 1164
Number of memory access instructions: 486
Number of Register ops: 1201
Number of branch instruction: 73
Number of jump instruction: 185
Predict correct : 28  , mis predict : 45 ,total predict: 73
Accurate : 38
**********************************************
```

gcd.bin

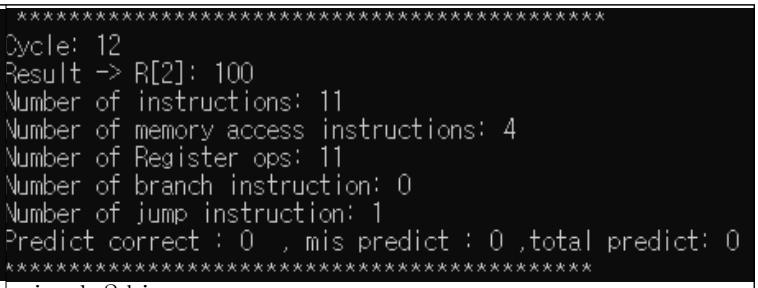
```
**********************************************
Cycle: 25400983
Result -> R[2]: 85
Number of instructions: 23372810
Number of memory access instructions: 7116606
Number of Register ops: 23372810
Number of branch instruction: 2029699
Number of jump instruction: 2028934
Predict correct : 1015613  , mis predict : 1014086 ,total predict: 2029699
Accurate : 50
**********************************************
```

input4.bin

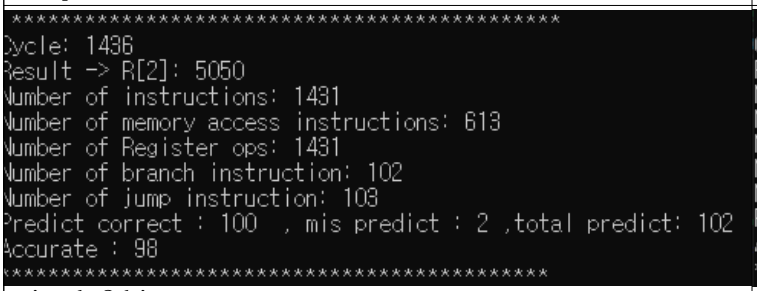
One Bit



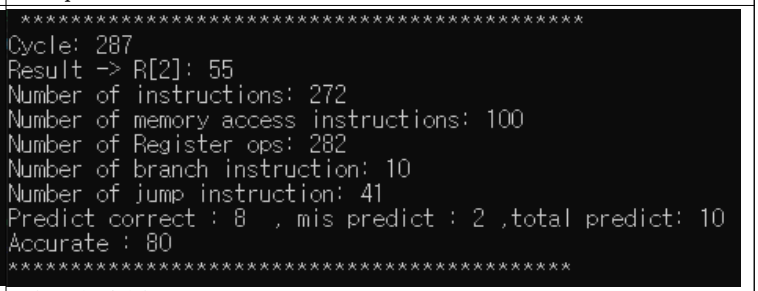
simple.bin



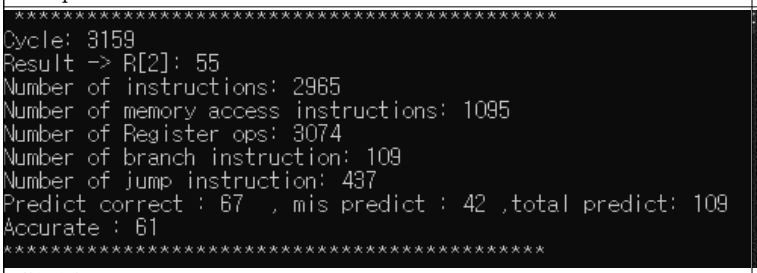
simple2.bin



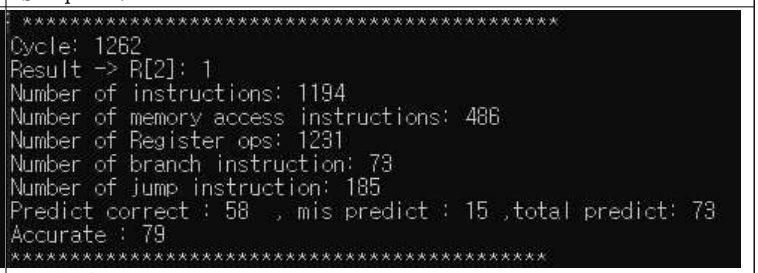
simple3.bin



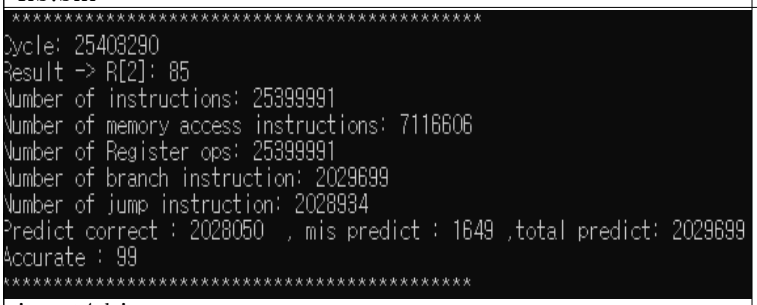
simple4.bin



fib.bin



gcd.bin



input4.bin

Two Bit

```
*********************************************
Cycle: 10
Result -> R[2]: 0
Number of instructions: 9
Number of memory access instructions: 2
Number of Register ops: 9
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
*********************************************
```

simple.bin

```
*********************************************
Cycle: 12
Result -> R[2]: 100
Number of instructions: 11
Number of memory access instructions: 4
Number of Register ops: 11
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
*********************************************
```

simple2.bin

```
*********************************************
Cycle: 1339
Result -> R[2]: 5050
Number of instructions: 1332
Number of memory access instructions: 613
Number of Register ops: 1332
Number of branch instruction: 102
Number of jump instruction: 103
Predict correct : 99  , mis predict : 3 ,total predict: 102
Accurate : 97
*********************************************
```

simple3.bin

```
*********************************************
Cycle: 282
Result -> R[2]: 55
Number of instructions: 265
Number of memory access instructions: 100
Number of Register ops: 275
Number of branch instruction: 10
Number of jump instruction: 41
Predict correct : 7  , mis predict : 3 ,total predict: 10
Accurate : 70
*********************************************
```

simple4.bin

```
*********************************************
Cycle: 3173
Result -> R[2]: 55
Number of instructions: 2953
Number of memory access instructions: 1095
Number of Register ops: 3062
Number of branch instruction: 109
Number of jump instruction: 437
Predict correct : 54  , mis predict : 55 ,total predict: 109
Accurate : 49
*********************************************
```

fib.bin

```
*********************************************
Cycle: 1230
Result -> R[2]: 1
Number of instructions: 1164
Number of memory access instructions: 486
Number of Register ops: 1201
Number of branch instruction: 73
Number of jump instruction: 185
Predict correct : 59  , mis predict : 14 ,total predict: 73
Accurate : 80
*********************************************
```

gcd.bin

```
*********************************************
Cycle: 23374691
Result -> R[2]: 85
Number of instructions: 23372810
Number of memory access instructions: 7116606
Number of Register ops: 23372810
Number of branch instruction: 2029699
Number of jump instruction: 2028934
Predict correct : 2028758  , mis predict : 940 ,total predict: 2029698
Accurate : 99
*********************************************
```

input4.bin

Two Level

```
*********************************************
Cycle: 10
Result -> R[2]: 0
Number of instructions: 9
Number of memory access instructions: 2
Number of Register ops: 9
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
*********************************************
```

simple.bin

```
*********************************************
Cycle: 12
Result -> R[2]: 100
Number of instructions: 11
Number of memory access instructions: 4
Number of Register ops: 11
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
*********************************************
```

simple2.bin

```
*********************************************
Cycle: 1341
Result -> R[2]: 5050
Number of instructions: 1332
Number of memory access instructions: 613
Number of Register ops: 1332
Number of branch instruction: 102
Number of jump instruction: 103
Predict correct : 98  , mis predict : 4 ,total predict: 102
Accurate : 96
*********************************************
```

simple3.bin

```
*********************************************
Cycle: 284
Result -> R[2]: 55
Number of instructions: 265
Number of memory access instructions: 100
Number of Register ops: 275
Number of branch instruction: 10
Number of jump instruction: 41
Predict correct : 6  , mis predict : 4 ,total predict: 10
Accurate : 60
*********************************************
```

simple4.bin

```
*********************************************
Cycle: 3175
Result -> R[2]: 55
Number of instructions: 2953
Number of memory access instructions: 1095
Number of Register ops: 3062
Number of branch instruction: 109
Number of jump instruction: 437
Predict correct : 53  , mis predict : 56 ,total predict: 109
Accurate : 48
*********************************************
```

fib.bin

```
*********************************************
Cycle: 1232
Result -> R[2]: 1
Number of instructions: 1164
Number of memory access instructions: 486
Number of Register ops: 1201
Number of branch instruction: 73
Number of jump instruction: 185
Predict correct : 58  , mis predict : 15 ,total predict: 73
Accurate : 79
*********************************************
```

gcd.bin

```
*********************************************
Cycle: 23374693
Result -> R[2]: 85
Number of instructions: 23372810
Number of memory access instructions: 7116606
Number of Register ops: 23372810
Number of branch instruction: 2029699
Number of jump instruction: 2028934
Predict correct : 2028757  , mis predict : 941 ,total predict: 2029698
Accurate : 99
*********************************************
```

input4.bin

Local Two Level

```
*********************************************
Cycle: 10
Result -> R[2]: 0
Number of instructions: 9
Number of memory access instructions: 2
Number of Register ops: 9
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
*********************************************
```

simple.bin

```
*********************************************
Cycle: 12
Result -> R[2]: 100
Number of instructions: 11
Number of memory access instructions: 4
Number of Register ops: 11
Number of branch instruction: 0
Number of jump instruction: 1
Predict correct : 0  , mis predict : 0 ,total predict: 0
*********************************************
```

simple2.bin

```
*********************************************
Cycle: 1341
Result -> R[2]: 5050
Number of instructions: 1332
Number of memory access instructions: 613
Number of Register ops: 1332
Number of branch instruction: 102
Number of jump instruction: 108
Predict correct : 98  , mis predict : 4 ,total predict: 102
Accurate : 96
*********************************************
```

simple3.bin

```
*********************************************
Cycle: 284
Result -> R[2]: 55
Number of instructions: 265
Number of memory access instructions: 100
Number of Register ops: 275
Number of branch instruction: 10
Number of jump instruction: 41
Predict correct : 6  , mis predict : 4 ,total predict: 10
Accurate : 60
*********************************************
```

simple4.bin

```
*********************************************
Cycle: 3175
Result -> R[2]: 55
Number of instructions: 2953
Number of memory access instructions: 1095
Number of Register ops: 3062
Number of branch instruction: 109
Number of jump instruction: 437
Predict correct : 53  , mis predict : 56 ,total predict: 109
Accurate : 48
*********************************************
```

fib.bin

```
*********************************************
Cycle: 1266
Result -> R[2]: 1
Number of instructions: 1164
Number of memory access instructions: 486
Number of Register ops: 1201
Number of branch instruction: 73
Number of jump instruction: 185
Predict correct : 41  , mis predict : 32 ,total predict: 73
Accurate : 56
*********************************************
```

gcd.bin

```
*********************************************
Cycle: 23374557
Result -> R[2]: 85
Number of instructions: 23372810
Number of memory access instructions: 7116606
Number of Register ops: 23372810
Number of branch instruction: 2029699
Number of jump instruction: 2028934
Predict correct : 2028825  , mis predict : 873 ,total predict: 2029698
Accurate : 99
*********************************************
```

input4.bin

-Conclusion

1. Signal Cycle & Pipeline

```
clcokcycle: 23372691
clcokcycle: 23372692
clcokcycle: 23372693
clcokcycle: 23372694
clcokcycle: 23372695
clcokcycle: 23372696
clcokcycle: 23372697
clcokcycle: 23372698
clcokcycle: 23372699
clcokcycle: 23372700
clcokcycle: 23372701
clcokcycle: 23372702
clcokcycle: 23372703
clcokcycle: 23372704
clcokcycle: 23372705
clcokcycle: 23372706
**********result**********
R[2]:85
I-type: 103 R-type: 10152862 I-type: 13219741
MemooryAccess: 7116606
BranchCount: 2029699
*********************************
```

```
*********************************************
Cycle: 23374557
Result -> R[2]: 85
Number of instructions: 23372810
Number of memory access instructions: 7116606
Number of Register ops: 23372810
Number of branch instruction: 2029699
Number of jump instruction: 2028934
Predict correct : 2028825  , mis predict : 873 ,total predict: 2029698
Accurate : 99
*********************************************
```

<그림56> Signal Cycle input4.bin result <그림57> Pipeline input4.bin result

이전 설명 1.1과 같이 싱글 사이클 방식은 한 번의 명령어가 처리되려면 충분히 명령어 1개 가 처리될 수 있는 시간을 기준으로 1번의 사이클당 1개의 명령어를 처리한다. <그림1> 번처럼 그 기준은 LW 기준으로 600ps라고 가정해보자.

input4.bin 파일을 기준으로 싱글 사이클 구조는 총 23372706번의 사이클을 거치고 프로그램이 종료된다. 그리고 파이프라인 구조에서 Local Two Level Branch Predictor는 23374557번의 사이클을 수행해야 프로그램이 종료된다. 결괏값으로 본다면 사이클의 수는 증가한다. 그 이유를 정리하자면 파이프라인 구조에서 명령어를 병렬처리 하면서 Latch를 통해 각 단계를 쪼개서 명령어를 처리하는데 이는 싱글 사이클과는 다르게 사이클 타임의 크기가 줄어들게 된다. 그래서 만약 파이프라인에서 처리 속도가 가능 느린 메모리 접속하는 명령어의 처리 속도를 200ps라고 가정하고 이 값을 각 사이클의 처리 속도라고 생각한다면 싱글사이클은 14ms가 걸리고 파이프라인은 4ms가 걸린다. 이는 파이프라인 구조를 활용한다면 싱글 사이클 구조보다 3~4배 정도의 속도 향상의 효과를 얻을 수 있다.

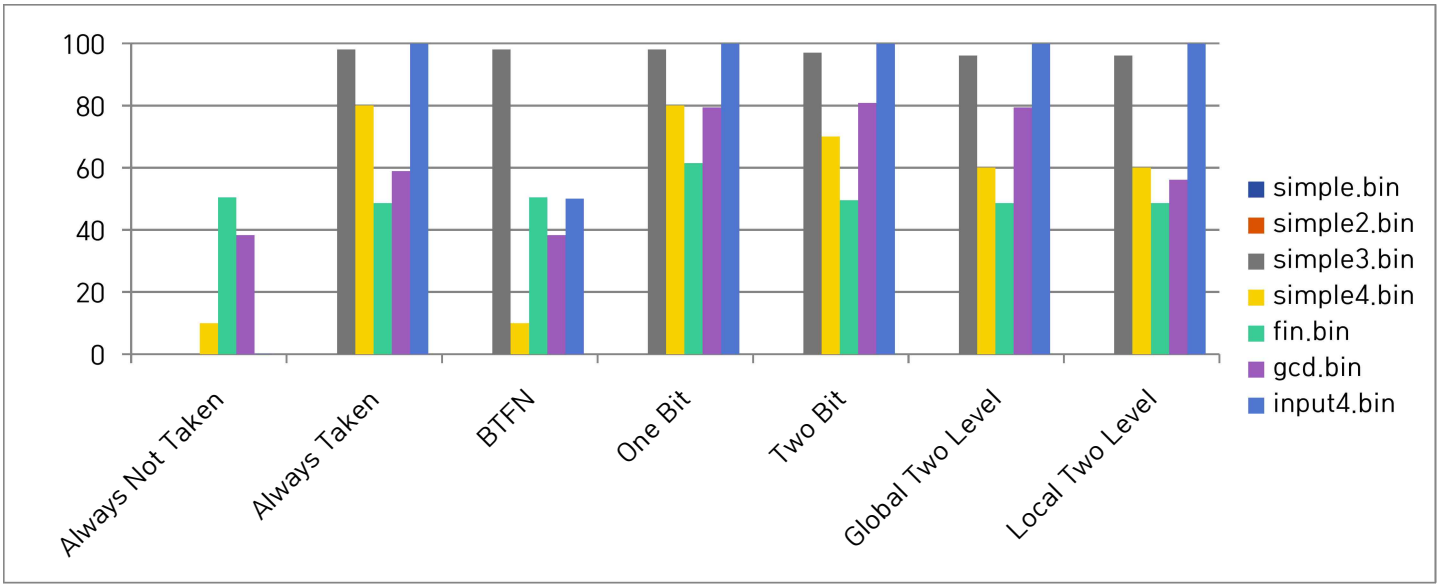
2. Branch Prediction Accuracy

Predictor	simple.bin	simple2.bin	simple3.bin	simple4.bin	fib.bin	gcd.bin	input4.bin
Always Not Taken	0	0	0	10	50.45	38.35	0.04
Always Taken	0	0	98.03	80.00	48.62	58.90	99.955
BTFN	0	0	98.03	10.00	50.45	38.35	50.037
One Bit	0	0	98.03	80.00	61.46	79.45	99.918
Two Bit	0	0	97.05	70.00	49.54	80.82	99.953
Global Two Level	0	0	96.07	60.00	48.62	79.45	99.953
Local Two Level	0	0	96.07	60.00	48.62	56.16	99.956

<표1> Accuracy of the Input Files

이번 프로젝트의 결괏값들에서 Branch Predictor마다 정확도를 <표1> 에 나타냈다. 이번 프로젝트에서는 simple3.bin부터 Branch 명령어가 사용되며 이를 처리하는 방법을 다르게 하여 처리를 해주었다. 먼저 3값을 확인해보자면 적당한 크기의 프로그램에서는 항상 Branch 명령어의 조건이 성립하지 않는다고 가정하는 방식인 Always not Taken 방식 이외에는 높은 정확도를 보였다. 하지만 이제 simple4.bin 파일에서는 총 Branch 명령어가 10번 정도 사용되는데 이는 정확도를 측정하기에는 충분하지 않았다. 즉 명령어의 양이 많지 않아서 이전 명령어들로 상태를 이용하여 충분한 학습이 필요한 Two Bit과 일정한 패턴을 만들어야 하는 Two Level Predictor의 경우에 이전 명령어들보다 낮은 정확도를 보였다. 이후 fib 와 gcd도 마찬가지로 예측을 모두 학습하기 이전에 프로그램이 종료되므로 정확도가 조금 떨어지는 결괏값이 나왔다. 그리고 이 결괏값에서 알 수 있듯이 프로그램 그리고 Predictor의 종류에 따라 조금씩 다른 정확도를 보이는 특징도 알 수 있었다.

마지막은 input4.bin의 결과이다. 이 프로그램은 다른 프로그램보다 더 많은 명령어가 존재했으며 이는 정확도를 판단하기에는 가장 적합하다고 볼 수 있다. 이제 결괏값들을 비교해 보자면 static Predictor 들은 결괏값이 좋은 경우(Always Taken)도 있지만, 항상 예측하는 방식이 같아서 나쁜 경우도 발생한다(Always Not Takne,BTFN). 하지만 Dynamic 방식에서는 One Bit에서 Local Two Level까지의 결괏값들이 조금씩 향상된 정확도를 보이는 것을 알 수 있다. 이는 충분히 학습이 가능한 정도의 프로그램이라면 이는 Local Two Level 방식이 가장 높은 정확도를 보인다고 이야기를 할 수 있다.



4. Personal Problems & Solutions

Q1. 왜 4를 빼고 Branch check 했는가.

```
pc: 0x20
pc updata : 0x24
bpc:0x54, b_address:0x20
pc: 0x58
pc updata : 0x5c
pc: 0x5c
pc updata : 0x60
pc: 0x60
pc updata : 0x64
pc: 0x64
pc updata : 0x68
pc: 0x68
pc updata : 0x6c
pc: 0x6c
pc updata : 0x70
pc: 0x70
pc updata : 0x74
*****************************************
Cycle: 1438
Result -> R[2]: 5050
Number of instructions: 1429
Number of memory access instructions: 613
Number of Register ops: 1429
Number of branch instruction: 102
Number of jump instruction: 103
Predict correct : 98  , mis predict : 4 ,total predict: 102
Accurate : 96
*****************************************
```

<그림68> Branch Check not -4

```
pc updata : 0x28
bpc:0x50, b_address:0x20
pc: 0x54
pc updata : 0x58
pc: 0x58
pc updata : 0x5c
pc: 0x5c
pc updata : 0x60
pc: 0x60
pc updata : 0x64
pc: 0x64
pc updata : 0x68
pc: 0x68
pc updata : 0x6c
pc: 0x6c
pc updata : 0x70
pc: 0x70
pc updata : 0x74
*****************************************
Cycle: 1341
Result -> R[2]: 5050
Number of instructions: 1332
Number of memory access instructions: 613
```

<그림69> Branch Check -4

Branch 명령어를 처리할 때 현재 주솟값 즉 이후 BTB에 사용될 인덱스값을 <그림68>은 그림은 Latch에서 넘겨받은 값이고 <그림69>는 -4를 해주고 처리를 해주었다. 이는 구조상으로 Branch 명령어를 Fetch 하면 예측을 해야 하는 조건에 맞지 않으므로 이는 Branch 명령어의 주소를 사용하기 위해서 -4를 처리해주었다. 그래서 BTB에 명령어의 본 주솟값을 인덱스로 값을 저장했다.

Q2. 디버깅 어떻게 했는가?

이번 프로젝트를 진행하면서 가장 시간이 오래 걸리고 또 하기 어려웠던 작업이 디버깅 작업이었다. 이전 싱글 사이클에서는 단순한 구조로 명령어들의 처리 과정을 print문으로 구분할 수 있었지만, 이번 프로젝트부터는 병렬 처리된 값들을 모두 표시하여 디버깅하기란 굉장히 어려운 작업이었다. 그래서 이번 프로젝트를 진행하면서 기본적인 구현하는 코드들을 먼저 체크를 했고 그 이후 디버깅을 했을 때 에러가 난다면 visual studio 안에 있는 기본 틀을 사용하여 변숫값의 변화와 그리고 함수가 잘 작동하는지를 확인하면서 손으로 MIPS 코드를 작성해보면서 디버깅을 하였다.

Q3. PC up DataMUXs

```
//LocalTwoLevelPredict
if (LocalTwoLevelPredict(pc)) {
    pc = BTB[pc];
}
else {
    pc = pc + 4;
}

if (JR) {
    pc = JRaddress;
    JR = 0;
}
else if (Jump) {
    pc = Jumpaddress;
    Jump = 0;
}
```

<그림70> 조건에 따라 PC 업데이트

코드를 구현하다가 Data path를 그리면서 먼저 전역변수 PC값에 상황마다 업데이트해주었다 하지만 Data path를 그리고 나서 PC값을 업데이트해줘야 하는 부분의 위치를 정할 수 있었고 이후 코드를 수정하여 PC값을 구조에 따라 업데이트해주도록 구현했다.

4.2 Conclusion and Feeling

내가 생각하는 이번 과제의 결론은 먼저 이전 초기 단계의 구조인 싱글 사이클 구조와 비교해 구현이 더욱 까다롭다. 또 추가로 필요한 CPU 하드웨어가 많아지면서 이전 모델보다 복잡하고 비싼 구조의 CPU를 디자인해야 하는 단점이 있지만, 요즘같이 프로그램의 크기와 그리고 처리해야 하는 데이터의 양이 많아지는 시대에는 복잡한 구조를 가지더라도 조금 더 처리 속도가 빠르고 또 정확도 있게 처리가 가능한 구조가 더욱 많이 사용되고 있다고 생각한다.

이번 파이프라인 구현 이전 과제인 싱글 사이클 과제에서보다 이번 과제가 더욱 시간이 매우 필요했다. 싱글 사이클 구조는 초기 컴퓨터 CPU 명령어 처리 방식이니만큼 그만큼 단순하고 구현 또한 단순했었다. 하지만 이번 과제부터 비교적 최근에 사용되었던 기술들과 그리고 구조를 구현하는 과제여서 더욱 복잡하고 또 생각해야 할 부분이 많았다고 생각한다. 이번 과제를 하면서 가장 힘들었던 점은 파이프라인 구조를 구현하면서 병렬 처리되는 명령어들을 확인해보지 하나하나 확인을 하기 힘들다는 문제로 디버깅하는 게 가장 힘들었다. visual studio에 있는 기본적인 디버깅 방식을 사용한다고 하더라도 이전처럼 print문으로 확인하는 데에는 한계가 있었다. 그래서 이전 과제보다 더욱 파이프라인 구조에 맞는 코드를 구현했는가 확인하는 작업에 시간을 많이 투자했었다. 그리고 디버깅을 하면서는 오류 나는 부분을 찾기 위해 이전 과목인 '시스템 프로그래밍'에서 했던 손으로 어셈블리 MIPS 코드를 구현하는 과정을 통해서 MIPS 코드가 어떻게 구현이 되는지를 먼저 알아보고 PC 주솟값이 잘 업데이트되는지를 보면서 디버깅을 하였다.
그리고 Branch Predictor를 구현하면서 복잡하고 새로운 방식의 방법들이 이전 방식과 정확도 면에서 비슷한 모습들을 보인다는 것에 놀랐고 Predictor의 종류에 따라 프로그램을 처리하는 정확성이 각각 다른 눈에 띄게 차이가 난다는 것 또한 놀랐다.
이번 과제는 시간이 오래 걸리는 과제이니만큼 학업 진도와 같이 복습하면서 내용을 정리하였지만, 과제 기간에 진도를 나가면서 생각보다 정리 작업의 기간이 오래 걸렸다. 그리고 코드 작업은 그날그날 진도 맞춰서 진행하였다. 하지만 이번 과제의 코드 작업이 Data Dependency 구현부터 이후의 작업은 모두 연관되어 코드가 돌아가서 이번 과제를 진행하면서 이론 공부의 복습이 완벽하게 이뤄지지 않았다면 생각보다 코드를 짜는 시간이 더 오래 걸렸던 것 같다.

이렇게 3번째 과제를 마치며 중간고사 이후부터 시작해서 한 달 동안 공부를 하면서 시간을 많이 들였는데 그만큼 이번 과목에 대하여 많이 배울 기회가 된 것 같고 또 구현을 직접 해보면서 코딩 실력이나 구현하는 실력 또한 늘었다고 생각한다. 시간을 많이 투자한 만큼 많은 것을 얻어가는 기회가 된 것 같아서 좋았다. 이제 마지막 남은 과제 또한 열심히 해서 잘 마무리를 하고 싶다.