
REPORT



Project #03 : Pipelined Simulator

- <기존에 남아있던 Freedays : 1일, 이번 과제에 사용할 Freeday : 1일>

과목명	컴퓨터구조및 모바일프로세서	담당교수	유시환 교수님
학 번	32204292	전 공	모바일시스템공학과
이 름	조민혁	제 출 일	2024/05/31

목 차

1. Introduction	1
2. Requirements.....	1
3. Concepts	2
3-1. Single Cycle uarch.....	3
3-2. Multi Cycle uarch.....	4
3-3. Pipelined uarch	5
3-4. Data Dependency	7
3-5. IDEA to Solve Data Dependency	8
3-5-1. Stalling.....	8
3-5-2. Score Boarding.....	9
3-5-3. Data Forwarding	10
3-6. Control Dependency.....	11
3-7. IDEA to Solve Control Dependency.....	11
3-7-1. Static Branch Prediction.....	12
3-7-2. Dynamic Branch Prediction	12
4. Implements	16
4-1. Build Enviroment	16
4-2. Overview of Pipelined Simpulator Datapath.....	17
4-3. Control Unit.....	17
4-4. Latch.....	19
4-5. Data Forwarding.....	20
4-6. Branch Predictor.....	22
4-7. Verify Branch Predict.....	26
4-8. Stage(IF, ID, EXE, MEM, WB)	27
5. Results.....	33
5-1. simple.bin	
5-2. simple2.bin	
5-3. simple3.bin	
5-4. simple4.bin	
5-5. fib.bin	
5-6. gcd.bin.....	

6. Conclusion & Feelings	36
--------------------------------	----

1. Introduction

이번 프로젝트에서 구현해볼 것은 'Pipelined Simulator MIPS'이다. 기존에 구현하였던 Single Cycle 보다 훨씬 복잡하며 Cycle 수도 압도적으로 증가하는 형태이다. 그럼에도 불구하고 현대 컴퓨터에서는 거의 대부분 Pipelined Microarchitecture 를 사용하고 있다. 왜냐하면 파이프라인 구조를 사용함으로써 특정 예외 상황을 제외하고는 컴퓨터 하드웨어의 구성 요소들을 각기 다른 명령어들을 필요한만큼의 Cycle 만큼 처리하게 해줄 수 있으며, 특정 하드웨어 부분이 지속적으로 동작할 수 있게 할 수 있기 때문에 파이프라인을 통해 효율적으로 하드웨어를 처리할 수 있기 때문이다. 이에 따라, 컴퓨터의 처리 능력 또한 향상되며, 기존 Single Cycle Architecture 와 Multi Cycle Architecture 가 가지고 있던 단점을 극복시켜준다. 이번 프로젝트에서는 MIPS 기반의 명령어를 바탕으로 Pipelined Simulator 를 구현을 해볼 것이다.

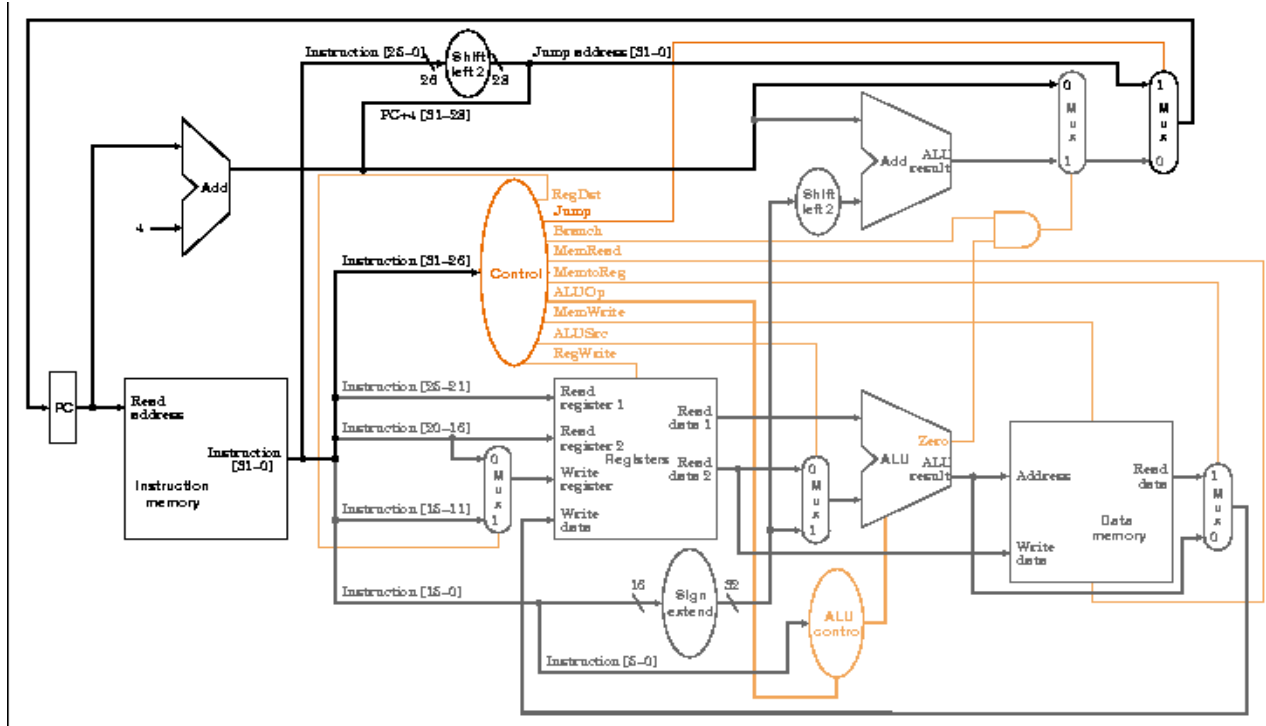
2. Requirements

Index	Requirements
1	<p>Single Cycle uarch 와 Pipelined uarch 의 성능을 비교할 것</p> <p>A. Program 은 반드시 실행되어 Correct Output 을 출력해야함</p> <p>B. CPU Clock Cycles 의 수를 비교할 것 (Cycle 수는 증가할 것임)</p> <p>C. CPU Clock Times 를 비교할 것 (Times 는 감소할 것임)</p>
2	<p>Machine Initialization</p> <p>A. Program 실행 전, Binary File 은 Memory 에 load 되어야함</p> <p>B. File Content 를 Memory 에 모두 Read 할 것</p> <p>C. RA Register (R[31]) -> 0xFFFF:FFFF 로 가정 -> PC 가 0xFFFF:FFFF 되면 Machine 은 실행을 끝내고, Program 은 Halt 됨</p> <p>D. Stack Pointer Register (R[29]) -> 0x1000000 으로 가정</p> <p>E. 그 외 Register 값들은 모두 0 으로 초기화할 것</p>
3	<p>Implementation Requirements :</p> <p>A. Simulator 에는 IF, ID, EX, MEM, WB Stage 가 구현되어야함 -> 각 Stage 의 결과는 Latch 에 저장되어야함</p> <p>B. Inst Memory 와 Data Memory 는 따로 구분할 것</p> <p>C. Clock Cycle 을 생성시키고, 모든 Stage 가 종료시 Clock Cycle 을 증가시킬 것</p> <p>D. Data Dependency 를 해결할 것 -> Data Forwarding 구현</p> <p>E. Control Dependency 를 해결할 것 -> Branch Prediction 구현</p>
4	<p>Prints Out</p> <p>A. 각 Cycle 이 종료 시, Simulator 는 이전 Stage 와의 변화를 출력</p> <p>B. r2 에 Final Result 를 저장하고 출력</p> <p>C. Total number Execution Cycles 출력</p> <p>D. Instruction Statistics 출력 (R-type, I-type, J-type)</p> <p>E. Total # inst, # of Memory ops, # of reg ops, # of branches, # of not-taken branches, # of jumps 출력</p>

3. Concepts

Pipelined Simulator 를 구현하기 전에 먼저 이전에 사용된 Cycle uarch 를 짧게나마 복습하고, Pipelined uarch 에 사용되는 구성 요소들을 확인하고, 사용함으로써 발생하는 문제점, 그에 대한 해결 방법을 알아보는 시간을 가지겠다.

3-1. Single Cycle uarch



Single uarch 를 살펴보면 먼저, PC 가 해당 주소의 Instruction Memory 에서 명령어에 대한 기계어를 가져온다. 이를 Instruction Fetch (IF) Stage 라 하며, 그 다음 가져온 명령어를 해석하는 Instruction Decode (ID) Stage 를 진행한다. 명령어를 해석함으로써 필요에 따라 opcode, rs, rt, rd, shamt, funct 를 가진다. 이후, Execution Stage (EXE) 단계를 진행하는데 opcode 또는 funct 에 따른 필요한 계산을 수행하게 된다. 그 다음, Data Memory 에 접근하여 값을 Data Memory 에 저장하거나, 가져오는 작업을 거친 후 Write Back Stage (WB) 단계를 마지막으로 진행하고, 이 단계에서는 수행된 명령의 값을 Update 하는 작업을 한다.

위에서 언급한 단계만 보면 간단하게 보이고, 순차적인 느낌이 강하게 든다. 하지만, Single Cycle 구조에는 치명적인 단점이 있다. 각 명령어는 자신에게 필요한 Stage 수가 정해져있는데, Single Cycle 은 가장 많은 Stage 를 필요로 하는 명령어를 기준으로 모든 명령어의 Stage 수를 동일하게 가져간다는 점이다. 'J' Inst 의 경우에는 ID 단계까지만 필요하다 하지만 5 Stage 수를 기준으로 Single Cycle 이 실행되면 J inst 는 WB 단계까지 기다리는, 이에 따라 3 단계나 시간이 낭비되는

치명적인 단점을 가지고 있다. 이걸 극복하고자 나온 개념이 바로 아래에 나오는 Multi Cycle uarch 개념이다.

3-2. Multi Cycle uarch

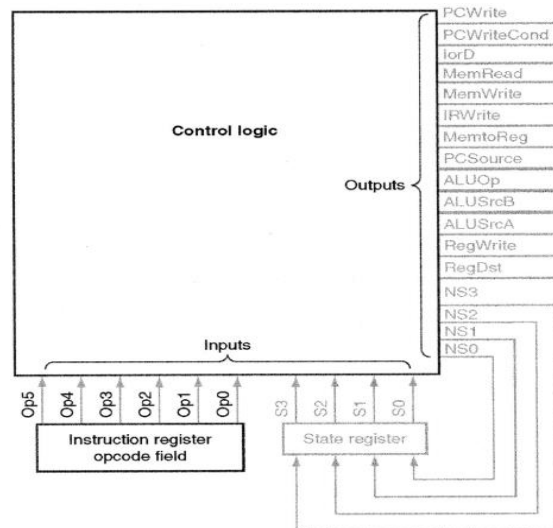


FIGURE D.3.2 The control unit for MIPS will consist of some control logic and a register to hold the state. The state register is written at the active clock edge and is stable during the clock cycle.

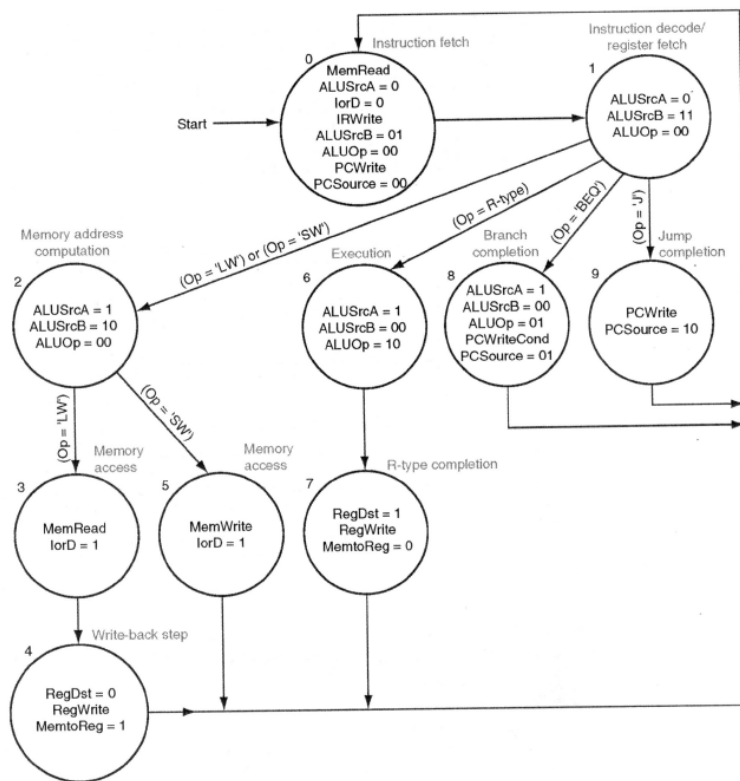
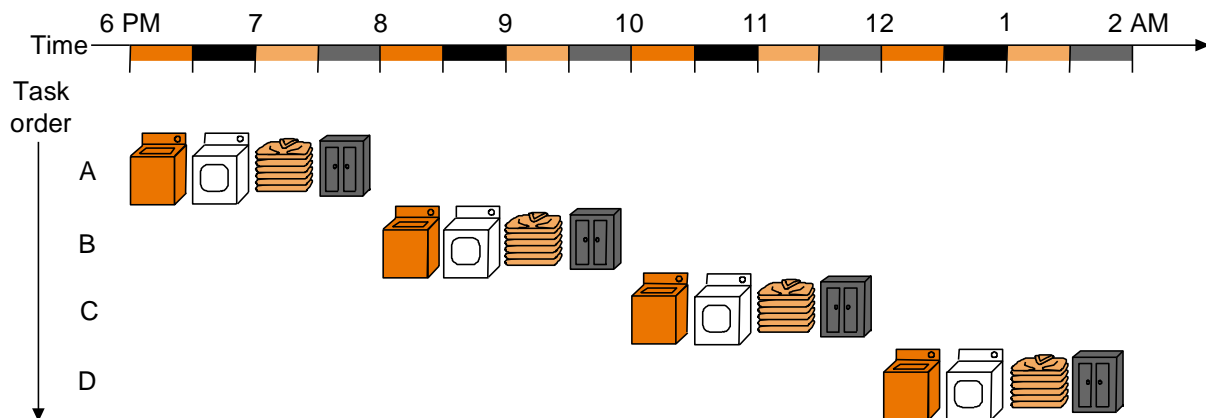


FIGURE D.3.1 The finite-state diagram for multicycle control.

Multi Cycle 은 Single Cycle 의 모든 명령어가 같은 Stage 를 가지고, 이에 따라 낭비되는 시간을 줄이고자 나온 개념이다. Multi Cycle 의 목표는 명령어 마다 각기 다른 Stage 를 가지게 해주는 것을 목표로 하고 있으며, 이에 따라 Control Signal 의 역할이 중요해진다. Control Signal 을 On/Off 해줌으로써 사전에 정의된 Data Path Table 을 통해 명령어의 행동이 정해진다.

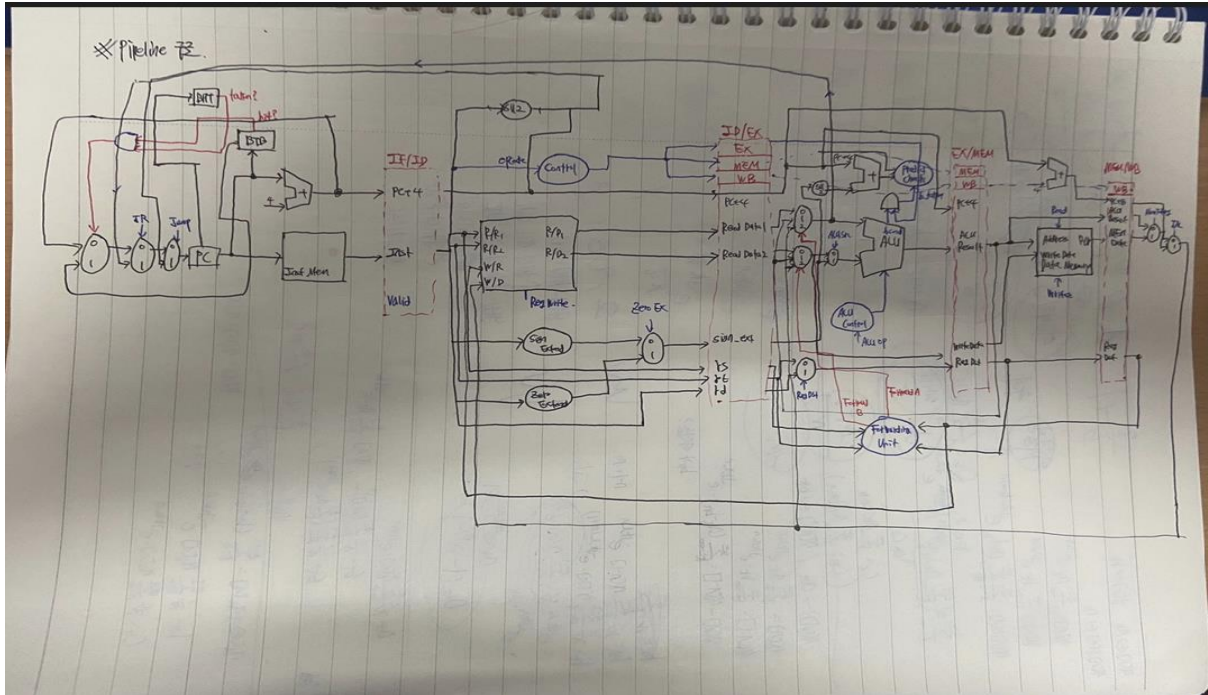
이에 따라 각 명령어마다 다른 Stage 수를 가지게 되므로 하나의 Clock 을 가지면서 명령어를 처리하는 Single Cycle 과는 다르게 각 Stage 마다 하나의 Clock 을 가지게 함으로써 여러 개의 Cycle 을 가지게 되는 것이다.

Multi Cycle 의 목표를 통해 Program 의 Total Clock Time 이 줄어들었다. 또한 효율적으로 하드웨어를 사용할 수 있게 되었다. 하지만 Multi Cycle 에게도 단점은 존재한다. 그 단점은 바로 명령어가 실행될 때 쉬고 있는 하드웨어가 존재한다는 것이다. 아래 그림을 보자

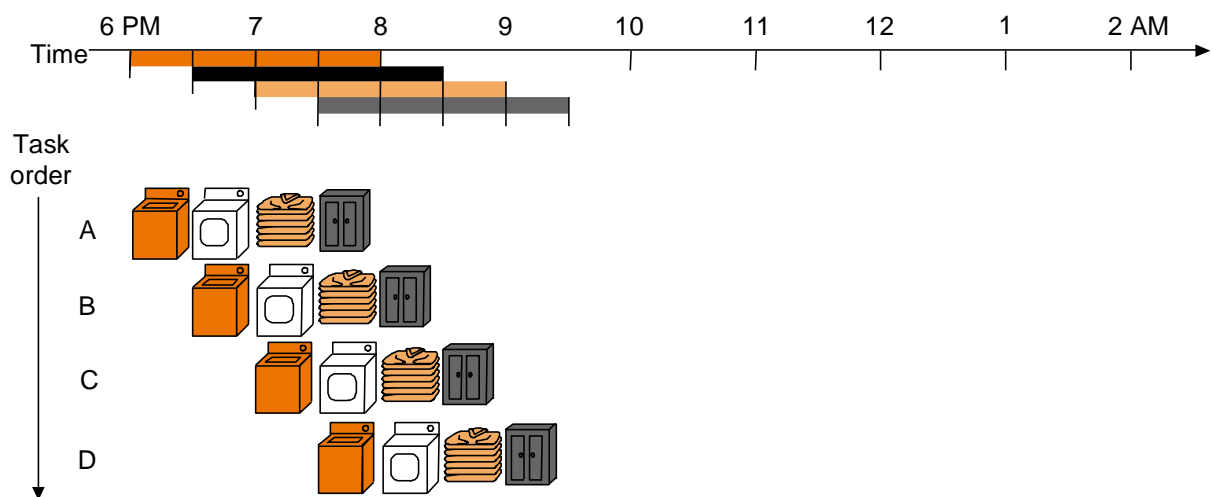


Multi Cycle 을 살펴보면 위와같이 하나의 명령어의 작업이 끝날 동안 하나의 명령어가 하드웨어를 독점하고 있는 구조이다. 이에 따라 동작하지 않는 하드웨어가 생기고, 이는 비효율적인 구조라고 생각할 수 있다. 이에 따라 우리의 목표는 작동하지 않는 하드웨어 없이 지속적으로 프로그램이 실행되는 동안 특정 상황을 제외하고는 모든 하드웨어를 지속적으로 동작시키는 것이다. 이런 생각을 기반으로 나온 개념이 Pipeline uarch 개념이다.

3-3. Pipelined uarch



위의 그림은 내가 직접 그린 Pipeline uarch 이다. 다음 그림을 살펴보면 Single Cycle, Multi Cycle 에는 존재하지 않던 하드웨어가 존재한다. 바로 Latch 인데, Latch 는 각 Stage 의 결과를 저장해주고 다음 Stage 에게 값을 전달해주는 역할을 한다. 이러한 Latch 가 값을 저장해주는 역할 덕분에 프로그램이 실행 시 모든 하드웨어를 사용할 수 있는 것이다. 아래 그림을 보자



하드웨어를 지속적으로 사용한다는 말은 위의 그림과 같이 어느 특정 하드웨어가 사용된 후 모든 작업을 기다리는 것이 아닌, 또 다른 작업을 그 하드웨어에서 진행시키는 것이다. 이렇게 함으로써 기존의 Single Cycle, Multi Cycle 와 비교하였을 때 엄청난 성능 향상이 된다. Multi Cycle 에서의 세탁 작업은 모든 사람이 세탁을 하는데에 총 8 시간이 걸렸다. 하지만 Pipeline uarch 에서는 약 3 시간 30 분정도 밖에 걸리지 않는다. 무려 4 시간 30 분이 감소한 시간이다.

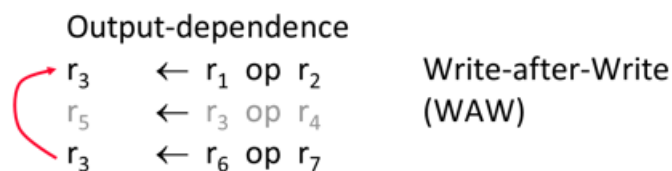
프로그램이 커지고, 명령어 수가 많아지면서 Pipeline 의 실행 성능은 Single Cycle, Multi Cycle 에 비해 점점 증가할 것이다. 이론적으로는 Pipeline uarch 만큼 효율적인 architecture 는 아직 없다. 하지만 이러한 Pipeline uarch 에도 단점이 존재한다.

3-4. Data Dependency

Pipeline uarch 에서 발생하는 첫 번째 문제는 Data Dependency 이다. Pipeline uarch 에서는 위에서 언급한 것과 마찬가지로 어느 특정 명령어의 작업이 진행되는 동안 다른 명령어도 작업을 진행시키는 특징이 있다. 작업이 진행되면서 Register 간에 연산이 이루어지고 WB 단계 후에야 연산된 값이 Update 된다.

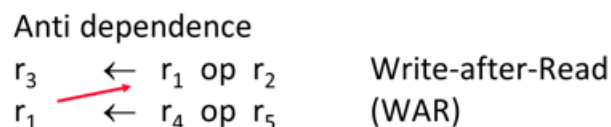
만약 어떤 명령어의 작업이 진행되는 동안 다른 명령어의 작업에서 아직 Update 되지 않은 Register 값을 사용하려고 하면은 치명적인 문제가 발생할 것이고, 올바른 프로그램의 결과가 나오지 않을 것이다. 이를 Data Dependency 라 하며 Data Dependency 는 3 가지의 Data Dependency 가 존재한다.

첫 번째로는 Write And Write (WAW, Output Dependence)이다. 아래 그림을 보자



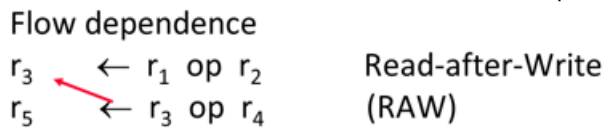
위의 그림을 보면 $r_1 \text{ op } r_2$ 의 값이 r_3 에 저장되고 있다. 이후 $r_3 \leftarrow r_6 \text{ op } r_7$ 에서 r_3 에 값을 쓰려고 한다. 여기서 기억해야할 것은 r_3 는 아직 Update 되지 않았다는 것이다. 따라서 Data Dependency 가 발생하게 된다.

두 번째로는 Write And Read (WAR, Anti Dependence)이다. 아래 그림을 보자



위의 그림을 보면 $r_3 \leftarrow r_1 \text{ op } r_2$ 후 바로 아래에서 r_1 의 값을 Write 하려한다. 여기서 발생하는 문제점 역시 r_1 역시 아직 Update 되지 않았다는 점이다. 이에 따라 Data Dependency 가 발생하게된다.

마지막으로는 Read And Write (RAW, Flow Dependence)이다. 아래 그림을 보자



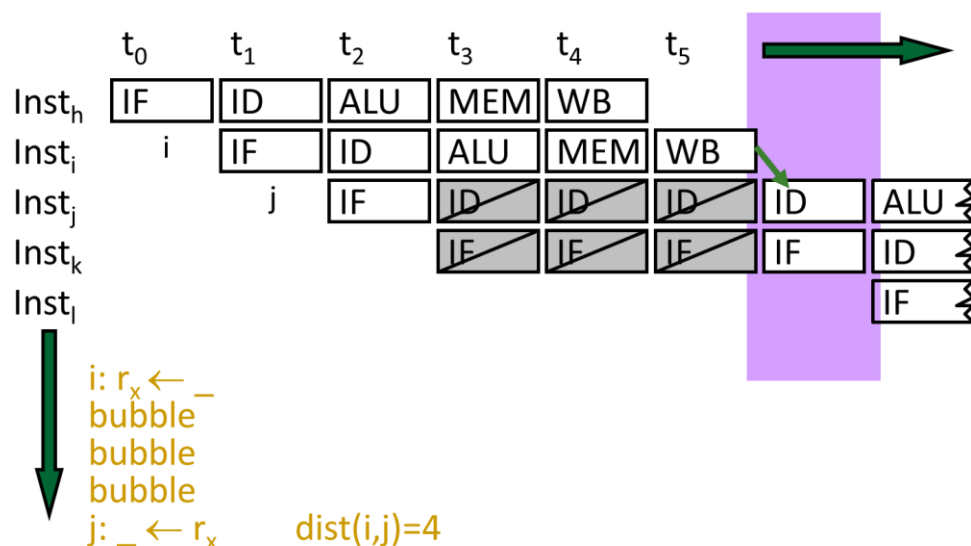
위의 그림을 보면 r3 에 값이 저장되고 아래 명령어에서 r3 를 사용하려고 있다. 하지만 이 상황 역시 아직 r3 는 update 되지 않은 상태이다. 따라서 update 되지 않은 값을 가져와 연산을 수행하게 되고 잘못된 결과를 불러올 것이다. 이에 따라 Data Dependence 가 발생하게 된다.

이러한 3 가지의 Data Dependence 를 해결해야 우리는 Pipeline uarch 를 사용할 수 있을 것이다. WAW 와 WAR 의 경우에는 프로그램 명령어 순서를 조절하면 해결할 수 있는 문제일 것이다. 하지만 RAW 의 경우에는 직접적으로 레지스터간의 연관성이 강하기 때문에 순서를 바꾸는 것으로만으로는 Data Dependence 를 해결할 수는 없다. 이에 따라 해결방법이 따로 필요하며 이를 아래에서 살펴보겠다.

3-5. IDEA to Solve Data Dependency

3-5-1. Stalling

Data Dependence 를 해결하는 첫 번째 방법으로는 Stalling 이다. Stalling 은 만약 Data Dependence 가 발생했을 경우 해당 명령어의 Register Value 가 Update 될 때까지 기다리는 것 즉, Stalling 하는 것이다. WB Stage 를 거친 후에는 Data Dependence 가 해결될 것이고, 이후 해당 Register 를 사용할 수 있을 것이다. 어떻게 이루어지는 지 아래 그림을 보자



위의 그림을 보면 Inst_i 와 Inst_j 간에 Data Dependence 가 발생한 것을 알 수 있다. 이에 따라 Inst_j 는 Inst_i 가 WB Stage 를 지날 때까지 기다리게 된다. 위의 그림에서는 3 번의 Time 동안 기다린 후에 사용이 가능해진다. 따라서 우리는 Data Dependence 의 조건을 알 수 있다. 보통 ID 단계에서 사용될 레지스터의 Index 를 알 수 있으므로 ID <-> EX, MEM, WB 의 관계에서 Data Dependency 가 발생한다. 따라서 이를 Pseudo Code 로 살펴보면 다음과 같다.

- $(rs(IR_{ID}) == dest_{EX}) \ \&\& \ use_rs(IR_{ID}) \ \&\& \ RegWrite_{EX}$ 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
- $(rt(IR_{ID}) == dest_{WB}) \ \&\& \ use_rt(IR_{ID}) \ \&\& \ RegWrite_{WB}$

이러한 조건에서 Data Dependency 가 발생하고 이에 따른 Stalling 이 진행되게 된다. 하지만 Stalling 을 살펴보면 WB Stage 가 끝나기를 기다려야한다는 특징이 있다. 이에 따라 Pipeline uarch 의 목표와는 다르게 쉬게 되는 하드웨어가 발생하게 된다. 따라서 Stalling 은 해결방법은 되지만 명쾌한 해결 방법이 될 수는 없다. 따라서 다음 해결방법을 살펴보자

3-5-2. Score Boarding

Score Boarding 은 사전에 Register 들에 대한 표를 만들어 Data Dependency 를 해결하는 것이다. 아래 그림을 보자

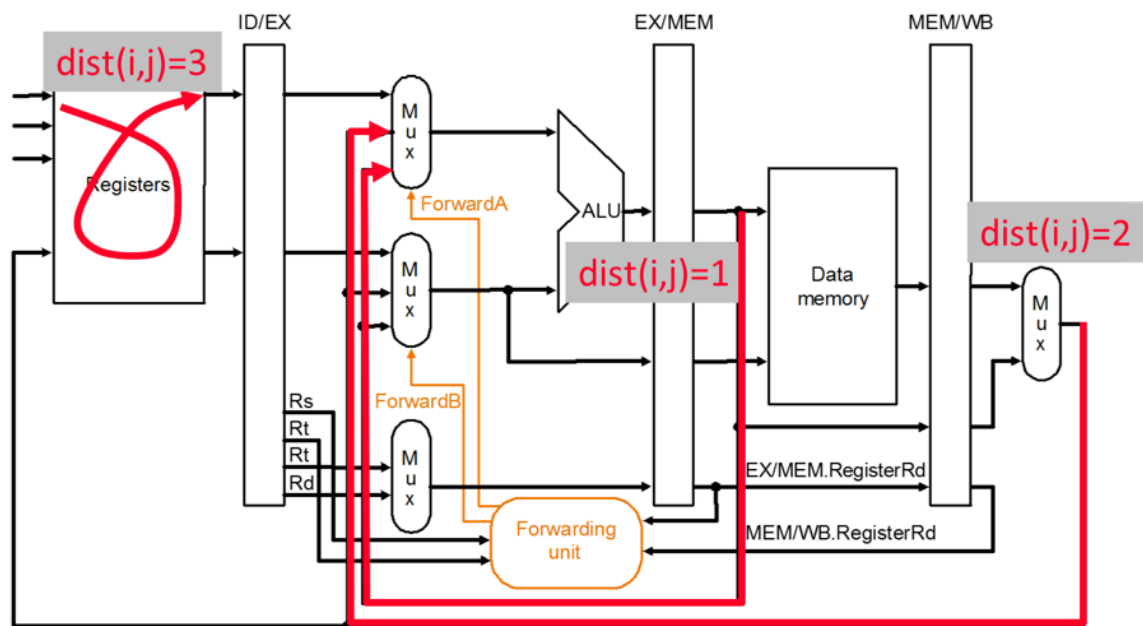
1) Initial State			2) when decoding, disable writeback register			3) enable back after write back		
Reg #	data	valid	Reg #	data	valid	Reg #	data	valid
0	0	1	0	0	1	0	0	1
1	aaa	1	1	aaa	1	1	aaa	1
2	bbb	1	2	bbb	1 → 0	2	bbb	1 → 0 → 1
...	...	1	1	1
31	XXX	1	31	XXX	1	31	XXX	1

위의 그림을 살펴보면 Reg Index 와 Reg Data, Valid 값이 존재한다. 만약 해당 명령어가 특정 번호의 레지스터를 사용한다면 valid 가 0 이 될 것이다. 그리고, WB Stage 가 끝난 후에야 valid 가 다시 1 로 되면서 해당 레지스터를 사용할 것이다. 이것 역시 간단하지만, 표를 따로 제작해야한다는 점, 또한 Stalling 을 피할 수 없다는 점이 있다. 마지막 해결방법은 Data Forwarding 인데 아래 에서 설명하도록 하겠다.

3-5-3. Data Forwarding

Data Forwarding 은 어쩌면 Data Dependency 를 해결할 가장 효율적인 방법이다. 본 레포트에서 Data Forwarding 을 채택하여 Data Dependency 를 해결하였다. Data Forwarding 은 Data Dependency 가 발생했을 경우, 위의 명령어가 EXE 단계가 끝났다면 이미 Register 값은 Update 만 되지 않고 연산의 결과는 Register 에 존재하기 때문에 연산의 결과를 WB Stage 가 끝날 때까지 기다리지 않고, 해당 값을 가져오는 것이다. 아래 그림을 보자

Data Forwarding Paths (v2)



b. With forwarding

위의 그림을 보면 만약 Data Dependency 가 발생했을 경우 해당 명령어가 WB Stage 가 끝날 때까지 기다리지 않고, Latch 의 output 에서 가져와 값을 사용한다는 특징이 있다. 이를 통해 하드웨어의 쉬는 시간 없이 효율적으로 지속적으로 사용할 수 있다.

```

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

```

위의 그림은 Data Dependency 의 발생 조건이다 해당 레지스터는 Data Dependency 가 EXE Stage 이후에 발생하는 지 알 수 있으므로, EXE Stage <-> MEM Stage, EXE Stage <-> WB Stage 에서 Data Dependency 가 발생할 수 있고, 이에 따라 Latch 를 통해 값을 가져오는 과정을 거치면 된다. 이를 통해 Stalling 을 하지 않고, 좀 더 간단한 조건 속에서 Data Dependency 를 해결할 수 있다.

3-6. Control Dependency

Control Dependency 는 Data 즉, Value 에 따른 문제가 아닌 Flow 즉, Control 에 대한 의존성이다. J-Type Inst, Branch Instruction 과 같이 Control Flow 를 제어하는 명령어에 대한 문제이다.

만약 Decode 한 명령어가 Branch 하는 명령어일 경우 이미 또 다른 Fetch 된 명령어가 존재할 것이다. 만약 Execution 단계에서 해당 명령어가 Branch 하는 명령어라는 것을 알게되면 이미 Fetch, Decode Stage 에 존재하는 명령어는 문제가 발생할 것이다. 이러한 문제점을 Control Dependency 라고 한다. 해당 명령어가 Branch 인지 아닌지를 모르는 상태이기 때문에 해당 명령어가 어떤 명령어인지 알기위해 Decode, Execution Stage 까지 Stalling 을 해야할까? 만약 그렇다면 Pipeline 은 너무 비효율적인 uarch 일 것이다.

따라서 이를 해결하기 위한 방법이 필요한데, 해당 명령어를 예측하고 예측한대로 프로그램을 진행한 후 예측이 맞았는지 틀렸는지 확인하는 곳에서 예측이 맞았을 경우 그대로 진행하고, 틀렸을 경우에는 알맞은 주소값으로 지정해주고, 이에 따라 필요한 만큼의 Bubble 을 발생시켜주는 과정으로 해결해준다. 이에 대한 여러 방법들이 존재하는데 이는 아래에서 살펴보겠다.

3-7. IDEA to Solve Control Dependency

3-7-1. Static Branch Prediction

Static Branch Prediction 은 말 그대로 정적 분기 예측이다. 이게 무엇이나 하면은 사전에 명령어에 대한 예측을 정해놓고 프로그램을 실행하는 것이다. 이에 대한 방법은 두 가지가 존재한다.

첫 번째로는 Always Not Taken 방식이다. Always Not Taken 방식은 말 그대로 Branch 여부를 항상 분기하지 않는다고 가정하는 것이다.

두 번째로는 Always Taken 방식이다. 이는 Branch 여부를 항상 Branch 한다고 가정하고 프로그램을 실행한다.

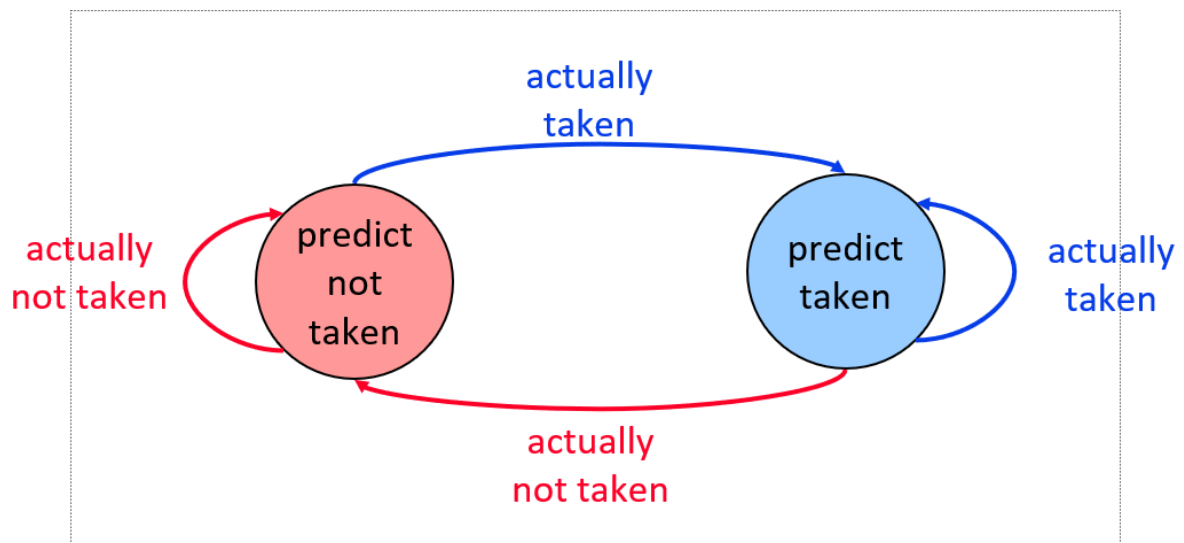
세 번째로는 Back Taken, Forward Not Taken(BTFN) 방식인데, Branch 할 주소가 현재 PC 값보다 뒤에 있으면 분기하고, 앞에 있으면 분기하지 않는 방식이다.

위 세가지 방법들 모두 프로그램 실행 전 정해놓은 방식인데, Always Not Taken 의 경우 30~40%의 낮은 정확도를 보이고, Always Taken 의 경우 60~70%의 정확도이고, BTFN 은 위 2 가지 방식들보다 통계적으로 봤을 때 정확한 특징이 있다.

3-7-2. Dynamic Branch Prediction

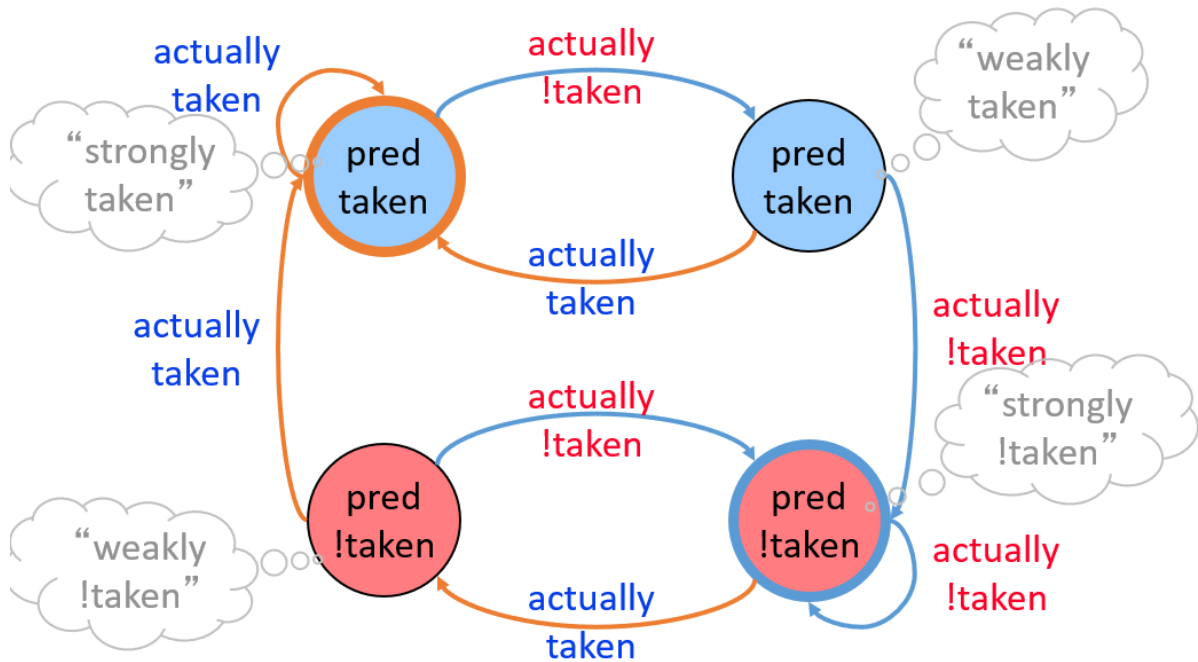
Dynamic Branch Prediction 은 동적 분기 예측이다. Static Branch Prediction 과는 다르게 사전에 분기 예측을 고정하지 않고, 프로그램의 실행 과정에서 과거 분기 여부에 따라 분기를 예측하는 것이다. 따라서 Dynamic Branch Prediction 의 경우에는 Branch Address 를 저장할 Branch Target address Buffer (BTB), Branch History Table (BHT), Global Branch History (GBH) 등 어떻게 Dynamic Branch Prediction 을 할 거냐에 따라 구성 요소도 달라지게 된다. 이러한 Dynamic Branch Prediction 의 경우에는 여러가지 종류가 존재한다.

첫 번째로는 One Bit Last Time Prediction (OBLTP)이다. OBLTP 의 경우에는 1bit 만 가지고 분기 예측을 진행한다.



위의 그림과 같이 Table 에 존재하는 Branch 여부를 1bit 로 저장하는데 해당 Table 에서 참고한 bit 가 1 이면 분기하고 0 이면 분기하지 않는다. 만약 분기를 하는 경우 BTB 에서 분기할 Address 를 가져와서 Branch 를 한다. OBLTP 의 경우에는 평균적으로 90%의 높은 예측율을 보이지만 만약 Table 에 존재하는 값이 10101010 과 같이 1 과 0 이 반복되는 형태로 Table 에 존재한다면 정확도는 0 이 된다는 단점이 있다.

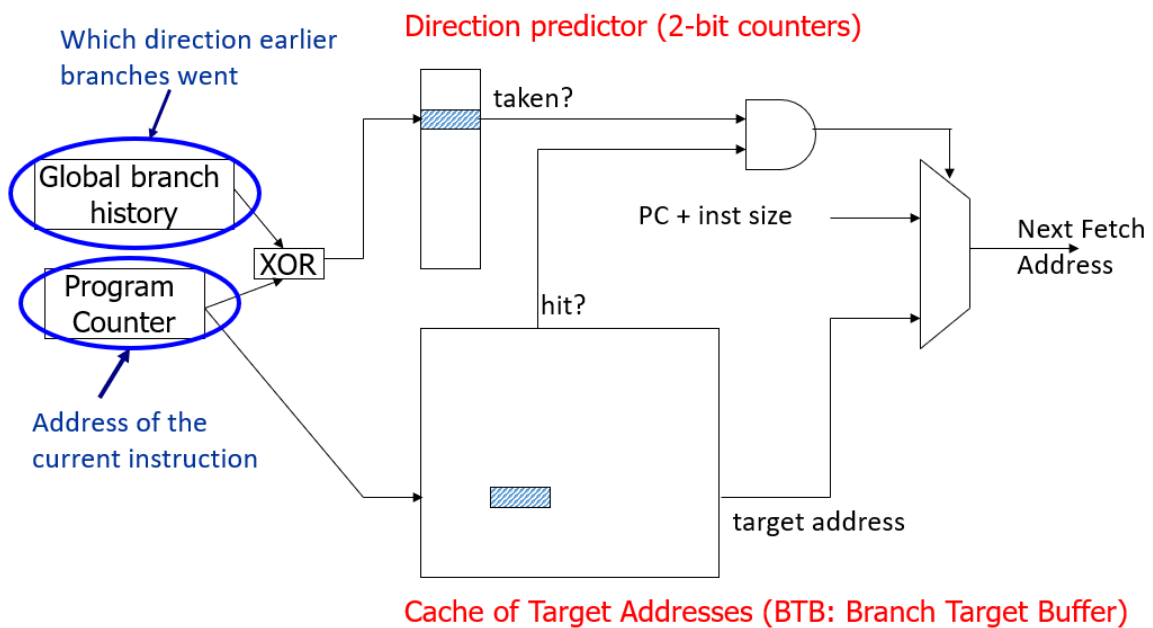
두 번째로는 Two Bit Last Time Predictor (TBLTP)이다. TBLTP 의 경우에는 OBLTP 를 강화한 버전으로, 2 bit 를 가지고 분기 예측을 진행한다.



위의 그림과 같이 00, 01 인 경우에는 분기를 하지 않고, 10, 11 인 경우에는 분기를 하도록한다. 만약 00 에서 분기가 틀렸을 경우에는 01 로 바뀌고 01 에서 한번 더 틀리면 10 으로 바뀐다. 이에 따라 Branch 여부가 TFTFTFTF 처럼 반복되는 경우에 정확도가 0 까지 떨어지는 걸 방지할 수 있다. 또한 평균적으로 90%의 정확도를 보이고 있어 OBLTP 보다 뛰어난 성능을 가지는 것을 알 수 있다. 하지만 OBLTP 나 TBLTP 의 경우에는 정확도가 90%까지는 기대할 수 있지만 그 이상은 힘들다는 특징이 있다. 만약 100 개의 명령어 중 1 개가 틀린다면 추가적인 사이클 수가 있을 것이고, 이는 프로그램의 크기가 커질수록 불필요한 Cycle 의 수는 증가할 것이다. 따라서 이를 극복하기 위해 Two Level Predictor 가 등장하였다.

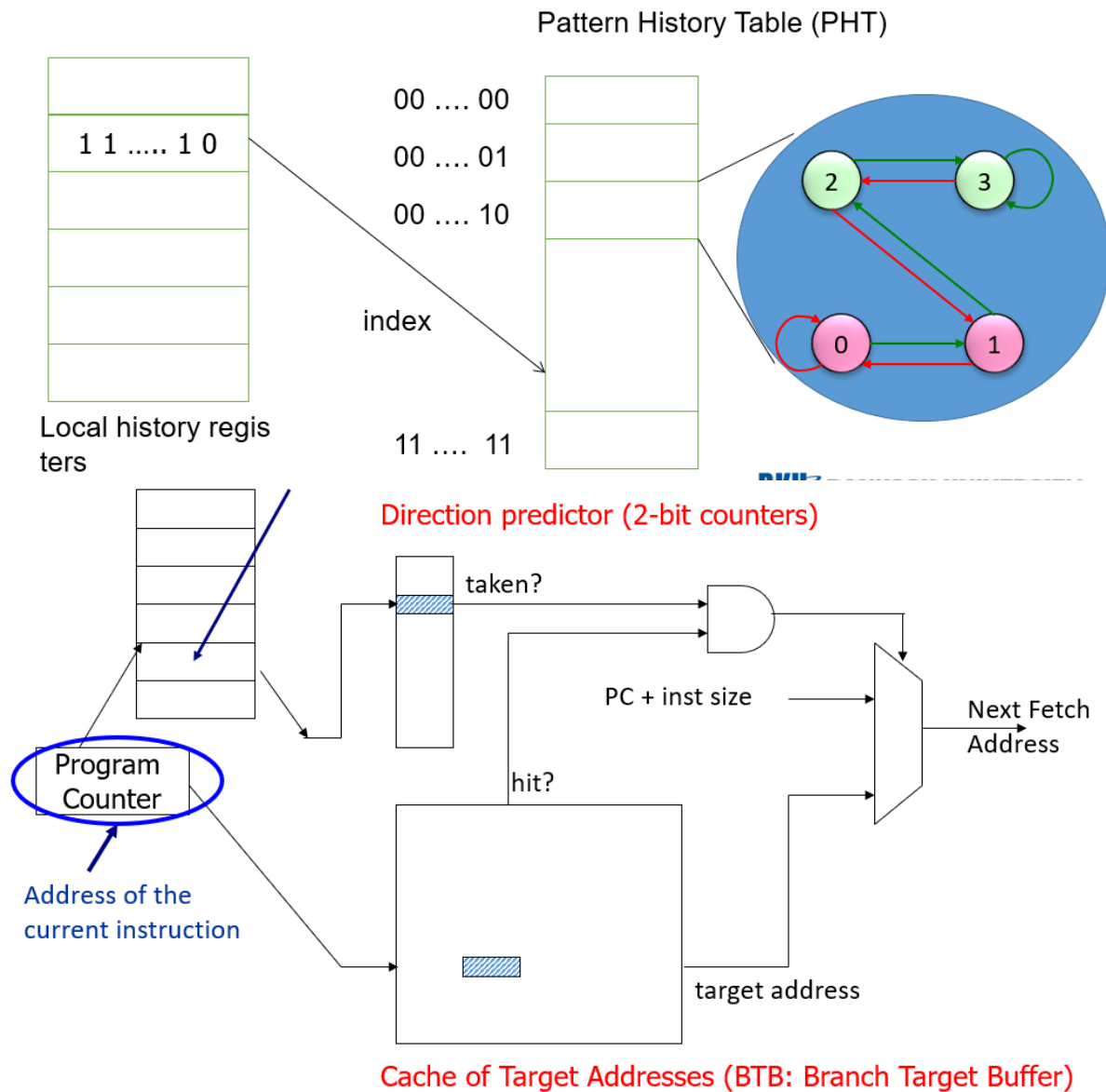
세 번째로는 위에서 언급한 One Level Predictor 와 다르게 Two Level Predictor 에 대한 설명이다. Two Level Predictor 는 One Level Predictor 과 다르게 추가적인 하드웨어 구성요소가 필요하다. 먼저 과거 Branch 여부를 저장하는 Global History Table (GHT)이고, 두 번째 하드웨어는 Pattern History Table (PHT)는 GHT 의 결과를 Index 로 가지고 그 안에 00, 01, 10, 11 과 같은 Value 를 가지는 형식이다. One Level Predictor 가 pc 값을 바탕으로 BTB 에 접근하고, 2-bit counters(PHT)에 접근했다면 Two Level Predictor 는 pc 값은 BTB 에 접근하도록하고, GHT 의 value 가 2-bit

counters 에 접근하도록한다. 이러한 방식을 Two Level Predictor 중 Two Level Global History Predictor 라고한다. 2-bit counters 에 접근하는 또 다른 방식이 있다.



위의 그림과 같이 2-bit counters 에 GHT 의 value 와 PC value 를 XOR 한 값을 바탕으로 PHT 에 접근하는 것이다. 이러한 방식은 Two Level Gshare Predictor 라고 한다. XOR 을 하기에 PC 의 값과 조금 더 연관성을 높일 수 있어 정확도가 증가할 수 있다.

네 번째로는 Two Level Local Branch Predictor(TLLBP)이다. TLLBP 의 경우에는 아래 사진과 같이 진행된다.



위의 사진과 같이 Local History Register(LHR) 구성요소가 필요하며, PC 가 가르키는 Branch 명령어의 서로 다른 주소값을 LHR 에 저장하고 PC 가 해당 Branch Address 을 가진다면 LHR 을 참고하여 이 값을 PHT 의 Index 로 가져와 분기 여부를 결정할 것이다. 이렇게 한다면 기존과는 다르게 분기 명령어의 주소값을 LHR 에 저장하기에 각 Branch 명령어를 구분하여 조금 더 정확성을 높일 수 있다.

4. Implements

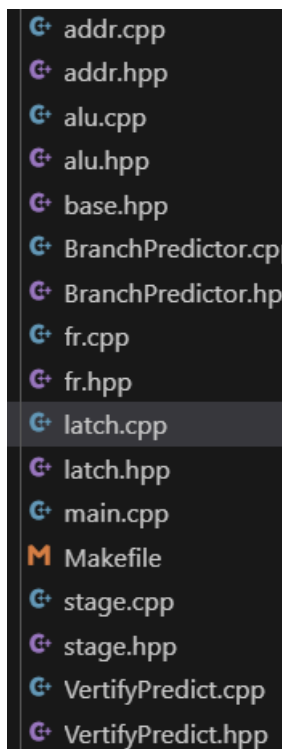
4-1. Build Environments

먼저 구현한 코드 설명에 앞서 해당 프로그램을 개발한 IDE, 프로그램을 실행하기 위한 Build 방식을 설명하겠다.


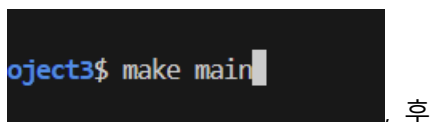
✓ 개발 환경 : Ubuntu 22.04, VSCode

✓ 프로그래밍 언어 : C++

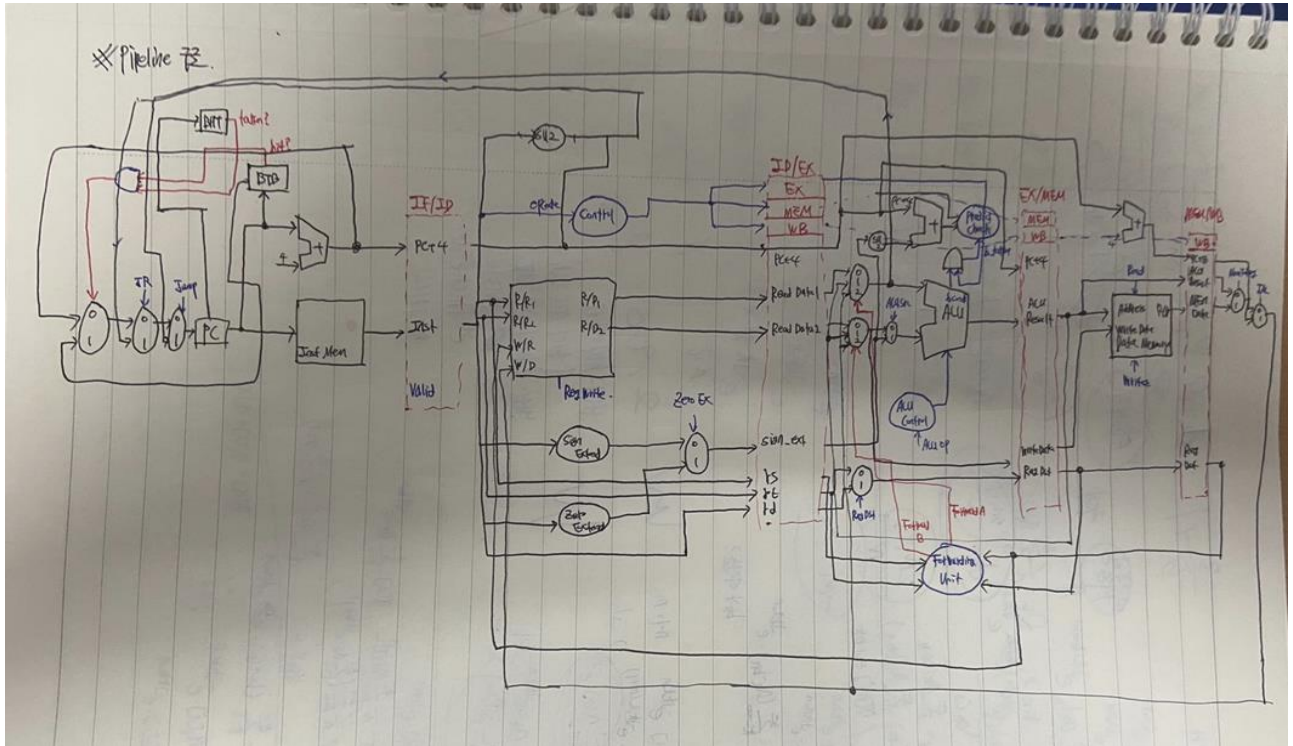
✓ 프로그램 구성 :



✓ 프로그램 실행 방식 : Makefile 이용 (아래 사진 참고)



4-2. Overview of Pipelined Simulator Data Path



아래에서 설명할 구현한 코드는 다음과 같은 Data Path 를 바탕으로 구현하였다.

4-3. Control Unit

```
/* Control Unit Class */
class EX_Signal{
private:
    int RegDst = 0;
    bool ALU_Src = false;
    int ALU_Op = -1;
    bool j = false;
    bool jr = false, jal = false;
    bool branch = false;
    bool sign = false;
public:
    EX_Signal();
    EX_Signal(int RegDst_, bool ALU_Src_, int ALU_Op_, bool j_,
              bool jr_, bool jal_, bool branch_, bool sign_);
    int get_RegDst() const;
    bool get_ALU_Src() const;
    bool get_j() const;
```

```

    bool get_jr() const;
    bool get_jal() const;
    bool get_sign() const;
    int get_ALU_Op() const;
    bool get_branch() const;
    void set_EX(int opcode_, int funct_);
};

```

```

class MEM_Signal{
private:
    bool MemWrite = false;
    bool MemRead = false;
    bool PcSrc1 = false;
public:
    MEM_Signal();
    MEM_Signal(bool MemWrite_, bool MemRead_, bool PcSrc1_);
    bool get_MemWrite() const;
    bool get_MemRead() const;
    bool get_PcSrc1() const;
    void set_MEM(int opcode_, int funct_);
};

```

```

class WB_Signal{
private:
    bool MemToReg = false;
    bool RegWrite = false;
    bool JalToReg = false;
    bool PcSrc2 = false;
public:
    WB_Signal();
    WB_Signal(bool MemToReg_, bool RegWrite_, bool JalToReg_, bool
PcSrc2_);
    bool get_MemToReg() const;
    bool get_RegWrite() const;
    bool get_JalToReg() const;
    bool get_PcSrc2() const;
    void set_WB(int opcode_, int funct_);
};

```

Control Unit 은 다음과 같이 EX_Sinal, Mem_Signal, WB_Sinal Class 로 총 3 개의 Class 로 구현하였다. 3 개의 Control Unit Class 들은 opcode, funct, 연산 결과에 따라 값이 저장되며 특정 Stage 에서 사용되고, Latch 의 Input 과 Output 으로 저장되어서 각, Stage 가 Control Signal 에 따라 특정 행위를 진행하도록 구현하였다.

4-4. Latch

```
class IF_ID{  
    public:  
        int valid;  
        int next_pc;  
        int instruction;
```

<IF/ID Latch>

```
class ID_EX{  
    public:  
        EX_Signal EX;  
        MEM_Signal MEM;  
        WB_Signal WB;  
        int valid;  
        int next_pc;  
        int ReadData1, ReadData2;  
        int rs, rt, rd;  
        uint32_t imm;  
        int shamt, funct;  
        int opcode;
```

<ID/EX Latch>

```
class EX_MEM{  
    public:  
        MEM_Signal MEM;  
        WB_Signal WB;  
        int valid;  
        int next_pc;  
        int ALU_Result;  
        int bcond;  
        int write_data;  
        int reg_dst;  
        uint32_t addr;
```

<EX/MEM Latch>

```
class MEM_WB{  
    public:  
        WB_Signal WB;  
        int valid;  
        int next_pc;  
        int reg_dst;  
        int ALU_Result;  
        int mem_data;
```

<MEM/WB Latch>

Latch 의 경우에는 총 IF/ID, ID/EX, EX/MEM, MEM/WB Latch 로 4 가지의 Latch 를 class 로 구현하였다. 각 Latch Class 안에는 Control Unit Class 를 가지게 된다. 이에 따라 Latch 를 통해 각 Stage 를 Input 으로 저장할 수 있고, Output 으로 다음 Stage 에 전달할 수 있게 된다. 또한 valid 를 변수로 가짐으로써 각 Latch 의 ON/OFF 를 결정하여 필요에 따라 ON/OFF 를 통해 Stage 를 멈추거나 Bubble 을 실행할 수 있게 된다.

```
IF_ID IFID_Latch[2];
ID_EX IDEX_Latch[2];
EX_MEM EXMEM_Latch[2];
MEM_WB MEMWB_Latch[2];
```

<Latch Declaration>

그 후 Stage 단계에서 Latch 를 선언해준다. Latch[2]의 이유는 각각 input 과 output 에 대한 Latch 부분이 필요하기 때문에 이와 같이 선언하였다.

4-5. Data Forwarding

```
/* Forwarding Unit */

int Forward_Rs(uint32_t rs_, uint32_t regDst1_, uint32_t regDst2_){
    if(rs_ == regDst1_){return 1;} // dist = 1
    else if(rs_ == regDst2_){return 2;} // dist = 2
    else{return 0;}
}

int Forward_Rt(uint32_t rt_, uint32_t regDst1_, uint32_t regDst2_){
    if(rt_ == regDst1_){return 1;} // dist = 1
    else if(rt_ == regDst2_){return 2;} // dist = 2
    else{return 0;}
}
```

<Data Forwarding>

Data Forwarding 의 경우에는 Rs 와 Rt 를 대상으로 발생하는 것이기 때문에 Rs 에 대한 Forward Unit 과 Rt 에 대한 Forward Unit 으로 구현하였다. 만약 rs_ 또는 rt_와 regDst1_ (Ex Stage 결과에 관한 것), regDst2_ (MEM Stage 결과에 관한 것)과의 Dependency 를 파악하여 각 stage 와의 distance 를 반환하도록 하여 구현하였다. 반환된 값에 따라 처리된다.

```

case 1: // dist 1
    if(IDEX_Latch[1].get_opcode() == 0x23){ // lw
        data1 = R[IDEX_Latch[1].get_rs()];
        break;
    }
    else if(IDEX_Latch[1].get_opcode() == 0x02 || IDEX_Latch[1].get_opcode() == 0x03){ // j inst
        break;
    }
    else{
        data1 = EXMEM_Latch[1].get_ALU_Result();
        break;
    }
}

```

Rt 에 대한 Data Forwarding 도 Rs 와 같이 처리되므로 Rs 를 대표적으로 가져와 설명하겠다. 먼저 Distance 가 1 인 경우 3 가지의 CASE 가 발생한다. 먼저, lw 명령어의 경우에는 계산된 결과가 아닌 이전 명령어의 인덱스를 가져와 해당 인덱스의 값을 가져와 저장하도록 한다. J-Type 명령어의 경우에는 Execution Stage 가 불필요하므로 그냥 break 하도록 한다. 그 외 명령어에는 ALU Result 를 가지도록 한다.

```

case 2: // dist 2
    if(MEMWB_Latch[1].WB.get_RegWrite() == 1 && MEMWB_Latch[1].WB.get_MemToReg() == 1 && MEMWB_Latch[1].WB.get_JalToReg() == 0){ // lw
        data1 = MEMWB_Latch[1].get_mem_data();
        break;
    }
    else if(MEMWB_Latch[1].WB.get_RegWrite() == 1 && MEMWB_Latch[1].WB.get_MemToReg() == 0 && MEMWB_Latch[1].WB.get_JalToReg() == 0){ // ALU Result
        data1 = MEMWB_Latch[1].get_ALU_Result();
        break;
    }
    else if(MEMWB_Latch[1].WB.get_RegWrite() == 1 && MEMWB_Latch[1].WB.get_JalToReg() == 1){ // JAL TO Reg
        data1 = MEMWB_Latch[1].get_next_pc() + 4;
        break;
    }
    else if(IDEX_Latch[1].get_opcode() == 0x02 || IDEX_Latch[1].get_opcode() == 0x03){
        cout<<"J inst doesn't need Data Dependency"<<endl;
        break;
    }
}

```

Rs 와의 Distance 가 2 인 경우에는 이전 명령어가 MEM Stage 이후이기 때문에 처리 방식이 다르다.

만약 MEM Stage 까지 처리한 명령어의 Control Signal 이 lw 를 위한 것, ALU Result 를 위한 것, JAL Value($R[31] = pc + 8$)를 위한 것일 수 있기 때문에, 각 CASE 를 분류하여 처리하였다.

만약 LW 에 대한 Data Dependency 인 경우에는 이미 MEM Stage 를 처리했기에 Memory Data 를 가져오도록 하고, MEM Stage 에서 처리는 하였지만 Data Memory 에 접근을 하지 않는 경우에는 단지 ALU Result 만 가져와 저장하도록 한다. 또한 $R[31]$ 의 경우에는 Next PC 값을 가져와 4 만큼 더해준 값을 저장하여 $R[31] = pc + 8$ 을 저장하는 JAL 명령어가 올바르게 실행되도록 한다.

Rt 에 대한 Data Dependency 도 위와 같이 진행하도록 한다.

4-6. Branch Predictor

```
extern uint32_t AlwaysNotTaken(uint32_t pc_);
extern uint32_t AlwaysTaken(uint32_t pc_);
extern int BTFN_Predictor(uint32_t pc_);
extern int OneBitPredictor(uint32_t pc_);
extern int TwoBitPredictor(uint32_t pc_);
extern int GlobalHistoryPredictor(uint32_t pc_, int GHR);
extern int GsharePredictor(uint32_t pc_, int GHR);
extern int LocalBranchPredictor(uint32_t pc_);
```

Branch Predictor 의 경우에는 다음과 같이 Static Branch Predictor 인 AlwaysNotTaken, AlwaysTaken, BTFN 을 구현하였고, Dynamic Branch Predictor 의 경우에는 1 Level Predictor 인 One Bit Last Time Predictor, Two Bit Last Time Predictor 를 구현하였고, 2 Level Predictor 는 Global History Predictor, Gshare Predictor, Local Branch Predictor 를 구현하였다. 다음과 같은 Branch Predictor 들은 모두 Instruction Fetch 단계에서 이루어지도록 하여 예측하여 PC 값을 조정하도록 한다.

```
/* Static Branch Predictor*/
uint32_t AlwaysNotTaken(uint32_t pc_){ // No Need BTB
    pc_ = pc_ + 4;
    return pc_; // Not Taken
}

uint32_t AlwaysTaken(uint32_t pc_){
    if(BTB[pc_] != 0){
        pc_ = BTB[pc_];
    }
    else{
        pc_ = pc_ + 4;
    }
    return pc_; // Taken
}
```

<ANT & AT>

```
int BTFN_Predictor(uint32_t pc_){
    predict_pc = pc_;
    if(BTB[pc_] == 0){
        return 0;
    }
    else{
        if(BTB[pc_] < pc_){ // Backward Taken
            return 1;
        }
        else{ // Forward Not Taken
            return 0;
        }
    }
}
```

<BTFN>

위의 코드는 Static Predictor 의 구현이다 먼저 AlwaysNotTaken 은 항상 분기하지 않는 것으로 예측하기에 return pc value 를 pc+4 로 해주어서 다음 명령어를 순차적으로 실행하도록 한다.

Always Taken 는 항상 분기하는 것으로 생각하기에 Branch Target address Buffer 에 저장되어 있는 주소를 가져와 pc 값에 저장하여 분기하도록 한다. 하지만 처음에는 어떠한 주소도 BTB 에 저장되어 있지 않기 때문에 pc+4 를 저장하여 return 하도록 한다.

BTFN의 경우에는 현재 주소보다 Branch Target Address가 뒤에 있을 경우 분기하기에 1을 반환하고, 앞에 있을 경우 분기를 안하기에 0을 반환해준다.

```

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

```

<BTFN 처리 과정>

이후 받은 값에 따라 Fetch Stage에서 Target Address를 값으로 줄지 안줄지를 결정한다.

```

/* Dynamic Branch Predictor */

/* Last Time Predictor (One level) */
int OneBitPredictor(uint32_t pc_){
    int hit = 0, taken = 0;
    if(BTB[pc_] != 0){ // Target Address exist
        hit = 1;
        switch(TakenHistory[pc_]){
            case 0x0:
                taken = 0;
                break;
            case 0x1:
                taken = 1;
                break;
        }
    }
    return (hit & taken);
}

```

<One Bit Last Time Predictor>

```

int TwoBitPredictor(uint32_t pc_){
    int hit = 0, taken = 0;
    if(BTB[pc_] != 0){
        hit = 1;
        switch(TakenHistory[pc_]){
            case 0x00:
                taken = 0;
                break;
            case 0x10:
                taken = 1;
                break;
        }
    }
    return (hit && taken);
}

```

<Two Bit Last Time Predictor>

다음은 One Level Branch Predictor인 One Bit Last Time Predictor와 Two Bit Last Time Predictor를 살펴보겠다. One Bit Last Time Predictor(OBLTP)의 경우에는 먼저 BTB에 값이 존재하는지를 살펴보고, 값이 존재하는 경우 BTB에 접속하기에 hit = 1이 설정이 된다. 그리고 pc값을 통해 과거 Branch 여부를 기록하고 있는 TakenHistory를 통해 값을 가져온다. 이후 0일 경우 분기하지 않기에 taken여부가 0으로 기록되고, 1일 경우 분기하기에 taken여부가 1로 기록된다.

Two Bit Last Time Predictor(TBLTP)의 경우에도 1bit에서 2bit로만 변하였지 같은 메커니즘으로 Branch Prediction을 진행한다.

```

/* Two-level Predictor*/
int GlobalHistoryPredictor(uint32_t pc_, int GHR){
    int hit = 0, taken = 0;
    if(BTB[pc_] != 0){
        hit = 1;
        switch(TakenHistory[GHR]){
            case 0x00:
            case 0x01:
                taken = 0;
                break;
            case 0x10:
            case 0x11:
                taken = 1;
                break;
        }
    }
    return (hit & taken); // pc+4 need after return
}

```

<Global History Predictor>

다음은 Two Level Branch Predictor 인 Global History Predictor 와 GShare Predictor 이다. One Level Predictor 와 크게 달라진 것 없이 단지 TakenHistory 에 대한 접근 방식만 달라졌다. 기존에는 PC 값으로 접근하였지만 PC 는 BTB 에만 접근하고 Global History Register 인 GHR 을 통해 Taken History 에 접근한다. 이후 메커니즘은 Two Bit Last Time Predictor 와 동일하게 진행한다.

GSharePredictor 의 차이점 역시 TakenHistory 에 대한 접근 방식인데 Global History Predictor 가 GHR 값을 통해 접근하였다면 GSharePredictor 는 PC 와 GHR 의 XOR 연산을 통해 TakenHistory 에 접근하도록 한다. 이후 메커니즘 역시 Two Bit Last Time Predictor 와 동일하게 진행한다.

이러한 방식으로 예측을 진행하도록 하고 이후 Execution 단계에서는 해당 명령어의 Branch 여부를 확실하게 알 수 있기에 분기 예측을 EXE Stage 에서 진행하도록 한다.

```

int GsharePredictor(uint32_t pc_, int GHR){
    int hit = 0, taken = 0;
    if(BTB[pc_] != 0){
        hit = 1;
        switch(TakenHistory[pc_ ^ GHR]){
            case 0x00:
            case 0x01:
                taken = 0;
                break;
            case 0x10:
            case 0x11:
                taken = 1;
                break;
        }
    }
    return (hit & taken);
}

```

<GShare Predictor>

```

int LocalBranchPredictor(uint32_t pc_){
    int hit = 0, taken = 0;
    if(BTB[pc_] != 0){
        hit = 1;
        switch(TakenHistory[PHT[pc_]]){
            case 0x00:
            case 0x01:
                taken = 0;
                break;
            case 0x10:
            case 0x11:
                taken = 1;
                break;
        }
    }
    return (hit & taken);
}

```

<Local Branch Predictor>

마지막으로 살펴볼 Branch Predictor 는 Local Branch Predictor 이다. Local Branch Predictor 의 경우에도 이전 Predictor 와 큰 차이는 없다. 하지만 Pattern History Table (PHT)를 이용하여 TakenHistory 에 접근한다. pc 값을 통해 PHT 에 접근하여 값을 가져오고 이를 다시 TakenHistory 에 Index 로 활용하여 값을 가져와 Branch Predict 을 진행하도록 한다.

4-7. Verify Branch Predict

```
void VerifyAlwaysNotTaken(uint32_t addr_, int taken_){
    if(taken_ != 1){ // Predict
        predict_count++;
    }
    else{ // Mispredict
        flush_IF_ID();
        flush_ID_EX();
        pc = addr_;
        mispredict_count++;
    }
}

void VerifyAlwaysTaken(uint32_t addr_, int taken_){
    if(taken_ == 1){ // Predict
        predict_count++;
    }
    else{
        flush_IF_ID();
        flush_ID_EX();
        pc = pc + 4;
        mispredict_count++;
    }
}
```

```
void VerifyBTFN(uint32_t addr_, uint32_t pc_, int taken_){
    if(BTB[pc_] < pc_){
        if(taken_){
            predict_count++;
        }
        else{
            flush_IF_ID();
            flush_ID_EX();
            pc = pc + 4;
            mispredict_count++;
        }
    }
    else{ // Forward
        if(!taken_){
            predict_count++;
        }
        else{
            flush_IF_ID();
            flush_ID_EX();
            pc = addr_;
            if(addr_ < pc_){
                BTB[pc_] = addr_; // BTB Update
            }
            mispredict_count++;
        }
    }
}
```

다음은 IF Stage 에서 예측한 Branch 여부를 EXE Stage 에서 검증하는 부분이다. 먼저 Static Branch Predictor 인 AlwaysNotTaken, AlwaysTaken, BTFN 을 살펴보겠다. AlwaysNotTaken, AlwaysTaken 의 경우에는 실제로 예측이 일어났는지에 대한 taken_ 변수를 통해 예측을 검증하게 된다. 예측이 맞을 경우 예측 성공 카운트를 증가시키고 끝내지만, 예측이 틀릴 경우에는 기존에 가져왔던 2 개의 명령어를 실행하면 안되는데, 이러한 두개의 명령어는 IF Stage, ID Stage 에 각각 있기에 이를 실행하지 않도록 flush_IF_ID(), flush_ID_EX() 함수를 통해 각 Stage 가 실행되지 않도록 한다. 이 후 pc 에 올바른 주소값을 할당함으로써 다음 Fetch 주소가 올바른 주소가 오도록한다.

BTFN 을 검증하는 부분은 나머지는 다 유사하게 진행하지만 Target Address 가 PC 값보다 앞에 존재하는 경우 분기하지 않는다고 예측하였지만 분기한다면 이 경우 BTB 를 Update 를 하는 과정을 거쳐준다.

```
        pc = addr_;
        BTB[pc_] = addr_;
        TakenHistory[pc_] = 1;
        mispredict_count++;
    }
    else{
        TakenHistory[pc_] = 0;
        predict_count++;
    }
```

다음은 Dynamic Branch Prediction 에 대해 예측 검증 부분이다. 위의 코드는 예측 검증 후 핵심이 되는 부분이다. 예측에 실패했다면 BTB 를 Update 해주는 것은 유사하지만 TakenHistory 에 대해 분기 여부를 지속적으로 기록하는 과정을 거쳐준다.

```
        BTB[pc_] = addr_;
        TakenHistory[pc_]++;
        if(TakenHistory[pc_] > 3){
            TakenHistory[pc_] = 3;
        }
        mispredict_count++;
    }
    else{
        TakenHistory[pc_]--;
        if(TakenHistory[pc_] < 0){
            TakenHistory[pc_] = 0;
        }
        predict_count++;
    }
}
```

Two Bit Last Time Predictor 에 대한 예측 검증은 0x00, 0x01, 0x02, 0x03 에 대한 값을 기준으로 분기 여부를 따지기에 분기 하지 않는다고 예측 하였는데 분기를 한다면 TakenHistory 에 대해 값을 증가시키고, 분기한다고 예측하였는데 분기 한다면 TakenHistory 를 역시나 증가시켜준다.

하지만 3 을 넘어가면 안되기에 혹시라도 값이 3 이 넘어간다면 3 으로 값을 조정해주고, 반대의 경우에도 마찬가지로 진행해주고, 0 아래로 값이 떨어지면 0 으로 값을 조정해준다.

2 Level Branch Predictor 의 경우에는 Two Bit Last Time Predictor 와 같이 예측을 검증하고 값을 Update 해주지만 위에서 설명한 것처럼 Index 에 대한 접근 방식만 다를 뿐이기에 따로 설명을 하지는 않겠다.

4-8. Stage (IF, ID, EXE, MEM, WB)

구현의 마지막으로는 Stage 부분이다 Stage 는 IF, ID, EXE, WB 로 총 5 단계의 Stage 로 이루어져 있다. 먼저 IF Stage 부터 살펴보겠다.

```

/* Fetch Stage */

void Fetch(){
    cout<<"====Fetch Stage===="<<endl;
    instruction = memory[pc / 4];
    IFID_Latch[0].instruction = instruction;
    IFID_Latch[0].next_pc = pc + 4;
    IFID_Latch[0].valid = 1;
    total_inst_count++;

    /* Branch Prediction */
    pc = AlwaysNotTaken(pc);
    //pc = AlwaysTaken(pc);

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

```

Fetch Stage에서는 메모리에서 명령어를 가져오고 이를 Latch에 저장한다. 이외에도 Output을 위한 next pc 설정과 Latch가 활성화 되었다는 valid 값의 설정을 하게 된다. 그리고 Fetch 단계에서 Branch Prediction을 진행하도록 한다.

```

/* Decode Stage */
void Decode(){
    if(IFID_Latch[1].valid == 0){
        return;
    }

    cout<<"====Decode Stage===="<<endl;
    cout<<"ID PC: "<<IFID_Latch[1].get_next_pc()-4<<endl;
    instruction = IFID_Latch[1].instruction;
    opcode = (instruction >> 26) & 0x3f; // Extract 6bits
    if(opcode == 0){ // R-type
        cout<<"R-Type"<<endl;
        R_inst++;
        rs = (instruction >> 21) & 0x1f;
        rt = (instruction >> 16) & 0x1f;
        rd = (instruction >> 11) & 0x1f;
        shamt = (instruction >> 6) & 0x1f;
        funct = instruction & 0x3f;

        /* Control Signal Generate */
        IDEX_Latch[0].EX.set_EX(opcode, funct);
        IDEX_Latch[0].MEM.set_MEM(opcode, funct);
        IDEX_Latch[0].WB.set_WB(opcode, funct);

        /* Read Register File */
        IDEX_Latch[0].rs = rs;
        IDEX_Latch[0].rt = rt;
        IDEX_Latch[0].rd = rd;
        IDEX_Latch[0].imm = 0;
        IDEX_Latch[0].shamt = shamt;
        IDEX_Latch[0].funct = funct;
        IDEX_Latch[0].next_pc = IFID_Latch[1].next_pc;
        IDEX_Latch[0].valid = 1;
        IDEX_Latch[0].opcode = opcode;
    }
}

```

Decode 단계에서는 가져온 명령어에서 먼저 opcode 를 추출하여 opcode 에 따른 MIPS 명령어 형식에 따라 나머지 부분에 대한 Decode 를 진행한다. 또한 Decode 된 변수들에 따라 Control Signal 을 활성화 시켜주고 Decode 된 변수들을 저장해주는 작업을 거쳐준다.

```

    }
    else if(opcode == 0x02 || opcode == 0x03){ // J-type
        cout<<"J-Type"<<endl;
        J_inst++;
        imm = instruction & 0x3fffffff;
        jump_count++;
        pc = JumpAddr(pc, imm);
    }

```

J-Type 의 경우는 조금 특별한데 J-Type 은 Decode Stage 까지만 명령어가 수행되기에 J-Type 에서 추출된 주소값이 바로 PC 에 저장되어 다음 Fetch Stage 에서 해당 명령어가 Fetch 되도록한다.

```

if((IDEX_Latch[1].EX.get_ALU_Src() == 0)){ // R-Type
    ALU_Result = ALU(data1, data2, IDEX_Latch[1].get_funct(), IDEX_Latch[1].get_shamt(), IDEX_Latch[1].get_opcode());
}
else if(IDEX_Latch[1].EX.get_ALU_Src() != 0 && IDEX_Latch[1].get_opcode() != 0x02 && IDEX_Latch[1].get_opcode() != 0x03){ // I-Type
    if(IDEX_Latch[1].get_opcode() == 0x2b || IDEX_Latch[1].get_opcode() == 0x23){
        ALU_Result = ALU(data1, IDEX_Latch[1].get_ext_imm(), 0, 0, IDEX_Latch[1].get_opcode());
        mem_addr = MemoryAddr(data1, IDEX_Latch[1].get_ext_imm());
    }
    else{
        ALU_Result = ALU(data1, IDEX_Latch[1].get_ext_imm(), 0, 0, IDEX_Latch[1].get_opcode());
    }
}
}

```

이후 EXE Stage 를 진행하는데 Data Dependency 를 Check 를 Execution 에서 진행하지만 위에서 설명하였으므로 여기서는 넘어가도록 하겠다. 위의 사진은 ALU 연산을 진행하는 부분으로 I-Type 중에서 SW, LW 의 경우에는 계산된 명령어 주소를 따로 저장해주는 작업을 거치도록한다. 이외에는 ALU 연산을 하여 ALU_Result 변수에 저장되도록한다.

```

if(IDEX_Latch[1].EX.get_RegDst() == 0){
    cout<<"RT HIHI : "<<IDEX_Latch[1].get_rt()<<endl;
    EXMEM_Latch[0].reg_dst = IDEX_Latch[1].get_rt();
}
else if(IDEX_Latch[1].EX.get_RegDst() == 1){
    cout<<"RD HIHI: "<<IDEX_Latch[1].get_rd()<<endl;
    EXMEM_Latch[0].reg_dst = IDEX_Latch[1].get_rd();
}
else if(IDEX_Latch[1].EX.get_RegDst() == 2){
    cout<<"JAL HIHI"<<endl;
    EXMEM_Latch[0].reg_dst = 31;
}
}

```

이후에는 계산된 Write Back 단계에서 Write 될 Register 의 Index 를 판단하기 위해 RegDst 의 Control Signal 에 따라 알맞은 값이 저장되도록 한다.


```

if(IDEX_Latch[1].EX.get_jr()){ // inst is jr, R-Type
    pc = data1;
    IFID_Latch[1].valid = 0;
    jump_count++;
}

```

Control Signal 에서 JR 이 활성화될 경우 JR 명령어가 수행된 것이기에 해당 주소로 Return 해주어야 하기 때문에 해당 주소값을 pc 에 저장해두고 IF Stage 를 비활성화 시킴으로써 실행되서는 안될 명령어가 실행되지 않도록 적절하게 Return 되도록한다.

```

/* Verify Branch Prediction */
if(IDEX_Latch[1].EX.get_ALU_Op() == 0x04 || IDEX_Latch[1].EX.get_ALU_Op() == 0x05){
    VerifyAlwaysNotTaken(branch_addr, branch_taken);
}
//    VerifyAlwaysNotTaken(branch_addr, branch_taken);
//    VerifyAlwaysTaken(branch_addr, branch_taken);
//    VerifyBTFN(branch_addr, IDEX_Latch[1].get_next_pc(), branch_taken);
//    VerifyOneBitPredictor(branch_addr, IDEX_Latch[1].get_next_pc(), branch_taken);
//    VerifyTwoBitPredictor(branch_addr, IDEX_Latch[1].get_next_pc(), branch_taken);
//    VerifyGHPredictor(branch_addr, IDEX_Latch[1].get_next_pc(), branch_taken);
//    VerifyGSharePredictor(branch_addr, IDEX_Latch[1].get_next_pc(), branch_taken);
//    VerifyLBPredictor(branch_addr, IDEX_Latch[1].get_next_pc(), branch_taken);
// }

```

이후 다음과 같이 Branch Prediction 에 대한 검증을 진행하고, 검증 과정은 위에서 설명하였으므로 넘어가도록한다.

```

/* Memory Access Stage */
void MEM(){
    if(EXMEM_Latch[1].valid == 0){
        return;
    }
    cout<<"=====MEM Stage===== "<<endl;
    if(EXMEM_Latch[1].MEM.get_MemRead()){ // lw
        MEMWB_Latch[0].mem_data = memory[EXMEM_Latch[1].get_addr() / 4];
        memory_access_count++;
    }
    else if(EXMEM_Latch[1].MEM.get_MemWrite()){ // sw
        memory[EXMEM_Latch[1].get_addr() / 4] = EXMEM_Latch[1].get_write_data();
        memory_access_count++;
    }
}

```

이후 MEM Stage 에서는 계산된 Memory Address 와 Write Data 를 바탕으로 Data Memory 에 접근하여 적절한 작업이 수행되도록한다.

```

/* Write Back Stage */
void WB(){
    if(MEMWB_Latch[1].valid == 0){
        return;
    }
    cout<<"=====WB Stage===== "<<endl;
    if(MEMWB_Latch[1].WB.get_RegWrite()){
        if(MEMWB_Latch[1].WB.get_MemToReg() == 0x01){
            // Mem Data to Reg
            R[MEMWB_Latch[1].get_reg_dst()] = MEMWB_Latch[1].get_mem_data();
        }
        else if(MEMWB_Latch[1].WB.get_MemToReg() == 0x00){
            if(MEMWB_Latch[1].WB.get_JalToReg() == 1){
                // JAL TO REG
                R[MEMWB_Latch[1].get_reg_dst()] = MEMWB_Latch[1].get_next_pc()+4;
            }
            // ALU Result to Reg
            else{
                R[MEMWB_Latch[1].get_reg_dst()] = MEMWB_Latch[1].get_ALU_Result();
            }
        }
    }
}
return;
}

```

마지막으로 WB Stage에서는 Register Value 를 Update 해주어야하기 때문에 활성화된 Control Signal 에 따라 Register 에 값을 Update 해주는 과정을 거친다.

```

/* Copy Latch (input -> output)*/
void copy_latch(){
    if(IFID_Latch[0].get_valid() == 1){
        IFID_Latch[1] = IFID_Latch[0];
    }
    if>IDEX_Latch[0].get_valid() == 1){
       >IDEX_Latch[1] =>IDEX_Latch[0];
    }
    if(EXMEM_Latch[0].get_valid() == 1){
       >EXMEM_Latch[1] =>EXMEM_Latch[0];
    }
    if(MEMWB_Latch[0].get_valid() == 1){
       >MEMWB_Latch[1] =>MEMWB_Latch[0];
    }
}

```

이후 모든 Stage 가 종료되면 각 Input Latch 를 Output Latch 로 옮겨주는 작업을 진행한다.

5. Results

5-1. Simple.bin

```
=====PROGRAM RESULT=====
Total Cycle Count of Program : 10
The Result is R[2] : 0
Total Instruction Count : 9
R-Type Instruction Count : 4
I-Type Instruction Count : 4
J-Type Instruction Count : 0
Jump Count : 1
Branch Count : 0
Memory Access Count : 2
Prediction Count : 0
Predict Count : 0
Mispredict Count : 0
=====
```

5-2. Simple2.bin

```
=====PROGRAM RESULT=====
Total Cycle Count of Program : 12
The Result is R[2] : 100
Total Instruction Count : 11
R-Type Instruction Count : 4
I-Type Instruction Count : 7
J-Type Instruction Count : 0
Jump Count : 1
Branch Count : 0
Memory Access Count : 4
Prediction Count : 0
Predict Count : 0
Mispredict Count : 0
=====
```

5-3. Simple3.bin

```
=====PROGRAM RESULT=====
Total Cycle Count of Program : 1535
The Result is R[2] : 5050
Total Instruction Count : 1534
R-Type Instruction Count : 512
I-Type Instruction Count : 920
J-Type Instruction Count : 1
Jump Count : 2
Branch Count : 101
Memory Access Count : 613
Prediction Count : 102
Predict Count : 1
Mispredict Count : 101
=====
```

5-4. Simple4.bin

```
=====PROGRAM RESULT=====
Total Cycle Count of Program : 294
The Result is R[2] : 55
Total Instruction Count : 293
R-Type Instruction Count : 120
I-Type Instruction Count : 153
J-Type Instruction Count : 11
Jump Count : 22
Branch Count : 9
Memory Access Count : 100
Prediction Count : 10
Predict Count : 1
Mispredict Count : 9
=====
```

5-5. fib.bin

```
=====PROGRAM RESULT=====
Total Cycle Count of Program : 3171
The Result is R[2] : 55
Total Instruction Count : 3170
R-Type Instruction Count : 1255
I-Type Instruction Count : 1697
J-Type Instruction Count : 164
Jump Count : 274
Branch Count : 54
Memory Access Count : 1095
Prediction Count : 109
Predict Count : 55
Mispredict Count : 54
=====
```

5-6. gcd.bin

```
=====PROGRAM RESULT=====
Total Cycle Count of Program : 1843
The Result is R[2] : 1
Total Instruction Count : 1842
R-Type Instruction Count : 755
I-Type Instruction Count : 926
J-Type Instruction Count : 55
Jump Count : 110
Branch Count : 106
Memory Access Count : 707
Prediction Count : 107
Predict Count : 1
Mispredict Count : 106
=====
```

6. Results

이번 Pipelined Simulator Project 는 지난번 Single Cycle Project 와는 다르게 비교적 빠르게 시작했다. 하지만 Pipelined Simulator 의 난이도는 이론 개념부터 만만치 않았다. 따라서 개념을 이해하기 위해 많은 정보를 찾아보고 이해를 하였다. 하지만 Code 로 구현하는 건 또 다른 문제였다. 이론을 이해했어도 코드로 바로 옮기는 것은 쉬운 일이 아니었다. 또한 Single Cycle Project 에 대한 test_prog 도 Simple3.bin 까지만 성공 시켰기에 지난 번 과제를 참고해도 큰 도움이 되지 못하였다.

이론에 대한 난이도도 높은데 코딩까지 하려니 정말 막막하였다. 그래도 노트에 적어가며 그림을 그려가며 하나씩 어느정도 느낌이라는게 왔고, 이를 구현하는 작업을 해보았다. 놀랍게도 100 프로는 아니지만 어느정도 구현이 된 것이다. 하나의 파일에 담기에는 코드의 양이 너무 많아 분할하여 코드를 구현하였는데 분할된 코드들이 정확하게 동작하는 것을 보고 놀라웠다. 하지만 실행은 되어도 원하는 결과가 나오는 것은 아니었다.

분명 잘못된 곳이 있는데 어디가 잘못됐는지는 모르겠고, 답답하였다. 그래서 의심이 되는 모든 부분에 디버깅을 위한 출력문을 생성해서 하나하나 주소값을 따라가며 잘못된 부분을 찾았다. 조금씩 잘못된 곳을 찾게 되었고, 정말 많은 시간을 디버깅하는데 할애했다. 약 3 주가량 Pipeline Simulator 구현에 투자하다 보니 어느덧 프로젝트 마감일이 다가왔다. 이론 공부와 코드 작성은 1 주일 걸렸지만, 디버깅은 2 주가량이 걸렸다. Input4.bin 에 대한 디버깅을 하고 원하는 결과값을 얻고 싶었지만, 어느덧 과제 제출일이 다가왔고, 아쉽게도 input4.bin 에 대한 결과는 얻지 못하였다. 이 부분이 너무 아쉽지만 개인적으로 많은 공부를 하였고, 공부한 내용을 이해한다라고 하더라도 코드 구현은 또 다른 문제라는 것을 알았다. 앞으로 코딩할 일이 많은 만큼 컴퓨터 과학에서의 이론을 코딩으로 구현하는 관점에서 바라보면서 앞으로의 학업을 해보고자 한다. 또한 마지막 과제에서는 지금보다 좀 더 발전된 완성형 프로그램을 만들고 싶다. 정말 의미있는 3 주의 시간이었다.