Computer Architecture and Mobile Process

파이프라인

Project3

2022년 5월 21일

Left day 5

모바일시스템공학과

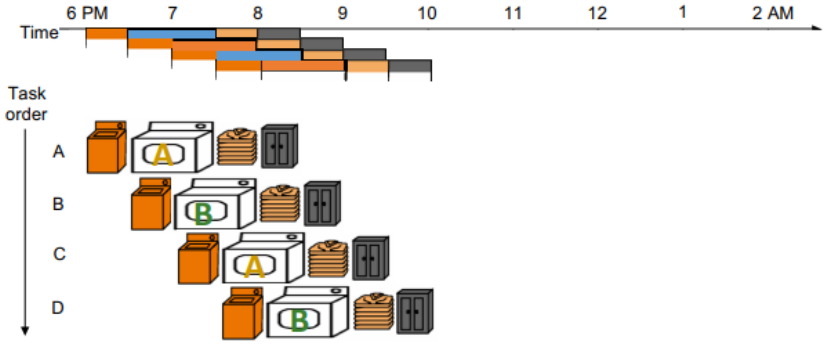
32184939 허찬용

■파이프 라인이 나오게 된 계기.

파이프 라인이 나오기 전에는 싱글 사이클이 있었다. 싱글 사이클 프로세서는 1 클럭에 걸리는 시간동안 한가지의 명령어를 처리하는 프로세서를 의미한다. 싱글 사이클은 Fetch, Decode, Execute, Memory Access, Write back 총 5단계가 있다. 싱글 사이클 같은 경우 Fetch가 이루어지면 Decode, Execute, Memory Access, Write back 단계를 수행하는데 여기서 문제가 생긴다. 싱글 사이클의 경우 Fetch가 진행될 때 Decode, Execute, Memory Access, Write Back에 필요한 하드웨어들이 쉬기 때문이다. 이처럼 각 과정에 필요한 하드웨어 하나가 작동하면 나머지 과정에 사용되는 하드웨어들은 작동을 하지 않는다는 문제가 있다. 이러한 문제를 해결하기 위해 파이프라인 프로세서가 등장하게 되었다.

■파이프 라인이란 무엇일까?

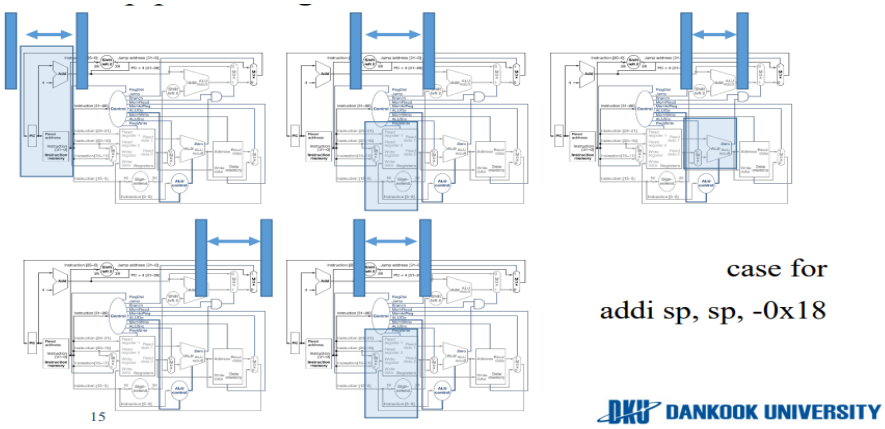
파이프 라인은 싱글 사이클 프로세서의 문제를 해결하기 위해 등장한 하나의 프로세서이다. 파이프라인 프로세서는 매 클럭마다 다수의 명령어를 중복 단계 없이 실행한다. 파이프 라인 프로세서는 각 단계별 처리 시간이 어느 정도 일정해야 한다. 그 이유는 파이프 라인에는 레치라는 하드웨어가 있기 때문이다. 단계별로 걸리는 시간이 다르다고 가정을 했을 때의 상황은 다음과 같다. 우선, 프로세서에서 가장 빨리 끝나는 단계를 Decode라고 가정한다. Decode 과정이 끝나고 Decode 단계를 통해 얻은 결과값들을 latch에 저장하는데 아직 Execute 단계는 실행 중에 있다. Decode가 다 끝났다고 계속해서 결과값들을 latch에 저장을 하게 되면 데이터가 덮어 씌워지는 경우도 발생하게 된다. 그래서 최대한 동일하게 단계별로 시간을 할당하고 약간의 딜레이 시간을 넣어준다. 이와 같이 파이프 라인을 구성하면 아래 그림처럼 명령어를 수행할 수 있다.



싱글 사이클 같은 경우는 A 단계가 끝나고 B 단계를 수행하겠지만 파이프 라인은 쉬는 하드웨어 없이 항상 돌아가게 될 것이다. 싱글 사이클의 각 단계에서 걸리는 시간과 파이프 라인의 각 단계에서 걸리는 시간이 모두 같다고 가정하면 파이프라인 프로세서는 싱글 사이클 프로세서 보다 약 5배는 빠른 성능을 보여줄 것이다.

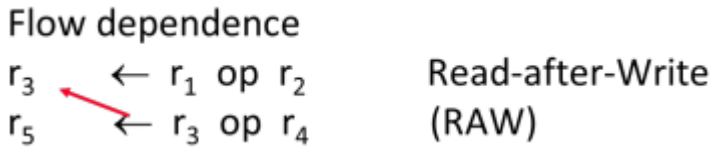
■ Latch

래치가 파이프라인의 꽃이라고 생각한다. 래치가 없었다면 파이프라인 구현이 힘들었을 것이기 때문이다. 래치는 각 단계별로 수행하여 나온 값들을 저장하는 중간 장치이다. 앞 단계에서 수행하여 얻은 값들을 저장하는 이유는 각 단계를 쉬게 두지 않고 전부 작동할 수 있게 만들기 위해서이다. 싱글사이클 같은 경우 Fetch를 수행할 때 나머지 4단계는 전부 쉬고 있다. 이 경우 일의 효율성이 떨어진다. 그래서 모든 단계를 다 수행하기 위해서 래치가 필요한 것이다. 뿐만 아니라 각 단계별로 끝나는 시간이 다르기 때문에 무작정 데이터를 보낼 수도 없는 노릇이다. 그러므로 약간의 딜레이를 주어 각 단계가 끝나는 시간을 비슷하게 맞추어 준다. 그 후, 앞선 단계에서 얻은 값들로 다시 다음 단계를 실행시키면 되기 때문에 모든 단계가 쉬지 않고 작동할 수 있는 것이다.

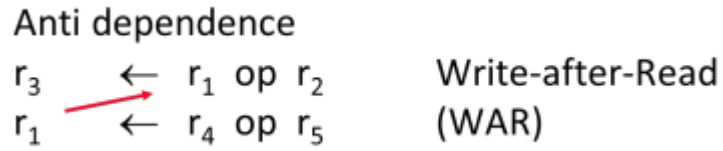


■ Data Dependency

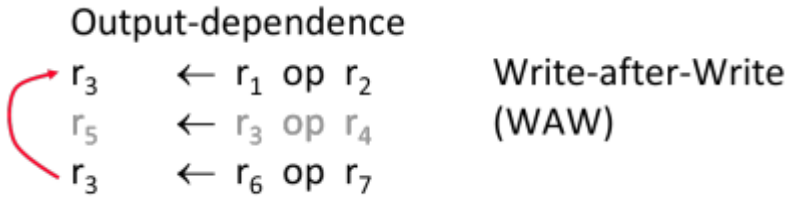
데이터 디펜던시란 파이프라인 프로세서를 사용하고자 할 때 다른 단계에서 같은 레지스터를 사용하게 될 때 생기는 문제점을 이야기하는 용어이다. 데이터 디펜던시에는 총 세가지가 있다. 첫 번째는 Flow Dependence이다. Flow Dependence는 쓰기 후 읽기라고 불린다. 그 이유는 1번 명령어를 add r3 r1 r2 2번 명령어를 add r5 r4 r3 라고 가정할 경우, 2번 명령어의 r3는 1번 명령어의 Write Back 단계에서 저장된 r3값을 사용해야 한다. 하지만 2번 명령어가 decode 단계에서 r3에 저장된 값을 가져오기 때문에 업데이트가 되지 않은 레지스터 값을 가져오게 된다. 이 경우 올바른 연산을 하지 못하게 되는 것이다.



두 번째는 Anti-Dependence이다. Anti-Dependence는 읽은 후 쓰기라고 불린다. 예를 들어, 1번 명령어가 add r3 r1 r2 2번 명령어가 add r1 r4 r5 일 때 다수의 스레드가 명령어를 수행한다면 1번 명령어의 r1을 읽고나서 2번 명령어의 r1을 업데이트 시켜야 한다는 것이다. 우리가 1번 명령어에서 사용할 값은 r1에 현재 있는 값이지 2번 명령어를 수행하고 나서 업데이트 된 r1 값이 아니기 때문이다.



마지막 세 번째는 Output-Dependence이다. Output-Dependence는 적은 후 적기라고 불린다. 간략한 설명을 하면 1번 명령어를 add r3 r1 r2 2번 명령어 add r5 r3 r4 3번 명령어 add r3 r6 r7이라고 가정했을 때, 다중 스레드를 사용하는 경우 r3값은 1번 명령어가 완료되고 저장이 된 뒤 2번 명령어에서 사용이 되고 3번 명령어에서 r3가 저장되어야 한다는 것이다. 만약 그렇지 않으면 3번째 명령어에서 업데이트 된 r3값이 2번 명령어에서 사용될 수 있기 때문이다.



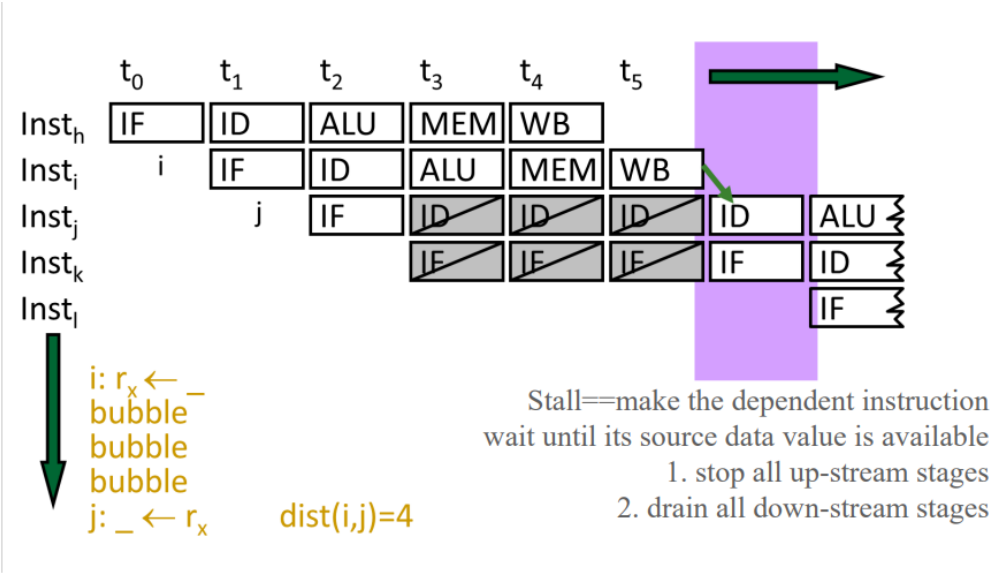
본 레포트에서는 한가지 스레드만 사용하는 경우에 대하여 실험을 할 것이기 때문에 Data flow Dependency만 집중하여 보면 된다.

■Data Dependency 해결방법

Data Dependency를 해결하기 위해서는 다양한 방법들이 있다. 본 수업에서 배운 방법은 크게 Stalling, Forwarding, Score Boarding이 있다. 본 레포트에서는 Fetch, Decode, Execute, Memory Access, Write back 5단계로 이루어져 있다고 가정한다.

■ Stalling

Stalling은 영어 의미 그대로 '지연시키다'라는 의미이다. Stalling에 대한 예시를 한가지 들겠다. 1번 명령어가 add r3 r1 r2 2번 명령어가 add r1 r3 r2 라고 가정하자. 2번 명령어에서 사용될 r3는 1번 명령어가 write back 단계를 거치고 업데이트 된 r3값을 사용해야 하지만 2번 명령어는 decode 단계에서 r1 값을 가지고 오기 때문에 업데이트가 되지 않은 값을 2번 명령어에서 사용하게 된다. 이런 문제가 생기는 것을 방지하기 위해서 decode 단계에서 정보를 적을 레지스터 번호와 사용할 레지스터의 번호가 같다면 앞선 명령어가 Write back 할 때까지 현재 명령어의 Decode 단계 수행을 지연시켜 사용할 레지스터의 업데이트를 기다린다.



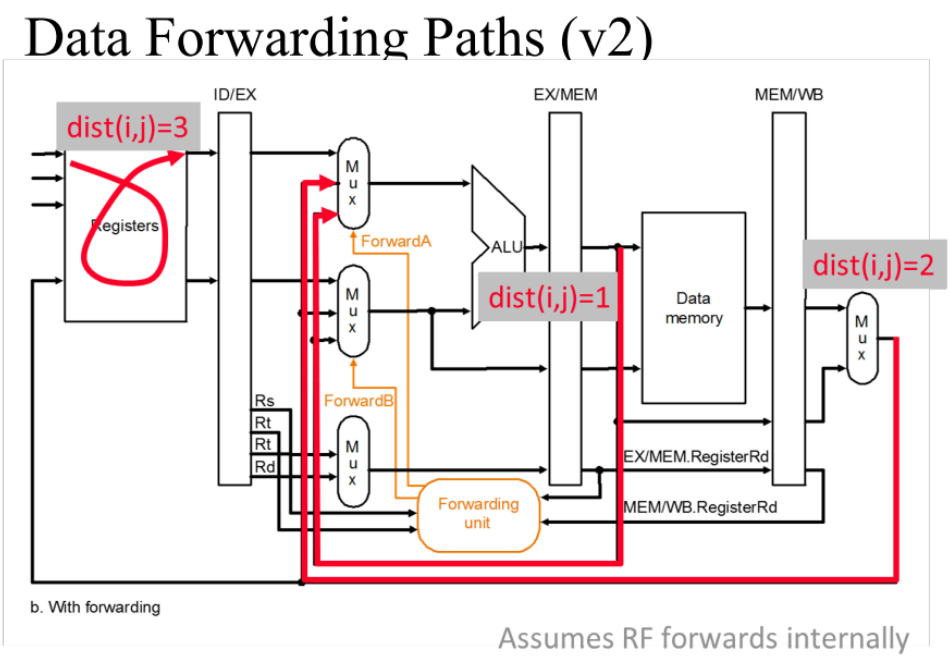
■ Forwarding

Forwarding은 영어 의미 그대로 '앞으로 보내다'라는 의미를 가지고 있다. Forwarding을 하면 Data flow 문제를 해결할 수 있다. Forwarding 방식은 Stalling 방식처럼 레지스터가 업데이트 될 때까지 앞선 명령어들을 지연시키는 것이 아니라 레지스터에 저장되어질 값들을 앞 단계로 보내는 것이다. 아래 그림을 보면 dist가 1에서 3까지로 세 가지 경우로 나뉘어진다. Dist가 1인 경우는 Memory Access 단계에서 Alu_Result 값을 Execute 단계로 보내어 업데이트 될 값을 미리 보내어 Data Dependency를 해결한다.

Dist 2인 경우는 Write_back 단계에서 저장할 레지스터의 번호와 Execute에서 사용할 레지스터의 번호가 같은 경우이다. 레지스터 번호가 같은 경우에는 저장할 데이터 값을 execute 단계로 보내어 업데이트 될 데이터 값을 연산에 사용하여 Data Dependency를 해결한다.

Dist 3인 경우는 Write_back 단계에서 저장할 레지스터의 번호와 Decode 단계에서 사용하는 레지스터의 번호가 같은 경우 Write_back에서 저장할 데이터 값으로 레지스터에 읽을 값을 업데이트 해준다. Dist 3인 경우도 생각해야하는 이유는 Write_back이 끝나고 Execute에서 업데이트 시키면 되겠지 하면 MEM/WB 레치는 업데이트 되어서 우리가 업데이트 시키고자 하는 데이터가 없기 때문이다.

위의 세 가지를 고려한 것이 바로 Data Forwarding이다.



■ Score Boarding

Score board는 Decode 단계에서 업데이트가 된다. 아래 그림과 같이 r2 레지스터가 적혀져야 하면 2번 레지스터의 valid가 0이 되고 write back 단계가 끝날 때까지 사용할 수 없게 된다. Write back 단계가 끝나면 다시 1이 되고 사용 가능한 상태가 된다. 만약 Decode 단계에서 r2를 읽어야 한다면 stalling을 통해 r2가 Write back 될 때까지 기다려야 한다. 여기서 눈 여겨봐야 하는 것은 tag파트이다. 예를 들어, tag가 없다고 생각해보자. 1번 명령어가 add r2, r3, r1 2번 명령어가 add r2 r3 r1 5번 명령어가 add r2 r3 r2라고 가정하면 5번 명령어는 1번 명령어의 r2 값이 아니라 2번 명령어의 r2 레지스터 값을 가지고 와야 한다. 이와 같은 이유로 tag를 적어 몇 번째 명령어의 레지스터 값을 가져와서 사용할 것인지 확인할 수 있도록 해야 한다.

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

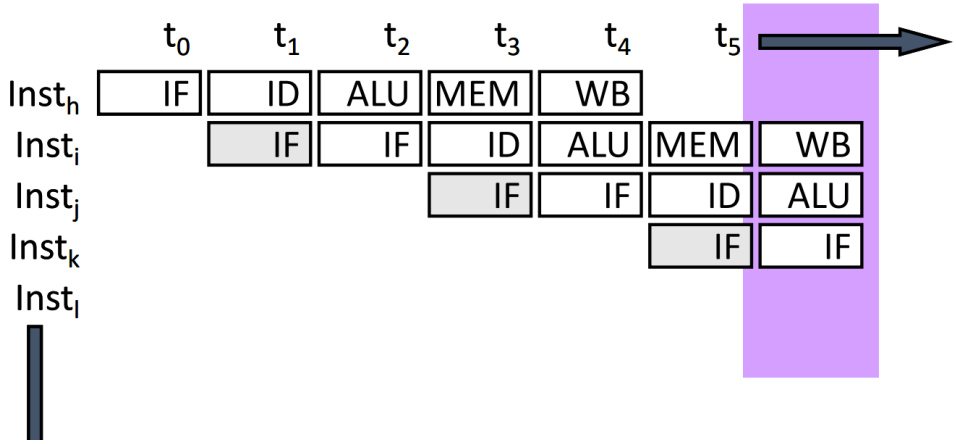
meaning: if you want to read r2, wait until inst1 writes back

■ Control Dependency

Control dependency란 수행해야 하는 다음 명령어가 있는 주소를 업데이트 시켜주는 것이다. 그래서 fetch 단계에서 pc를 업데이트 해야 한다. 하지만 명령어 중에서 jump나 jump and link, jump register, branch not equal, branch equal 과 같이 pc값이 pc+4가 아닌 다른 pc값으로 뛰어넘는 경우가 있다. 이와 같은 경우에는 fetch에서 업데이트하기 어렵다. fetch에서는 단지 pc를 가지고 메모리에 접근하여 명령어를 가져오고 pc를 업데이트 하는 역할을 하기 때문이다. Jump, Jal, Jr 같은 경우는 Decode에서 어디로 갈지 정해진다. 물론 ppt에서는 register dependent 때문에 Execute 단계에서 주소가 해결된다고 하지만 앞서 배운 Data Dependency의 dist가 3인 경우를 생각한다면 충분히 Decode 단계에서도 register dependent를 해결하고 점프할 수 있다. Beq나 Bne는 execution에서 branch address가 정해지기 때문에 Execution 단계에서 주소가 해결이 된다. 하지만 pc는 계속해서 pc+4로 Fetch 단계에서 업데이트 되기 때문에 Execute 단계나 Decode 단계에서 pc가 업데이트 되면 문제가 발생하게 된다. 해당 문제를 해결하기 위해서 크게 Stalling, branch prediction, branch delay slot을 사용한다.

■ Control Dependency Stall

Control Dependency Stall은 다음 pc가 사용가능 할 때까지 지연시키는 것이다. 그렇게 되면 Decode 단계에서 어떤 명령어인지 확인을 하고 pc를 업데이트 시켜야 하므로 Fetch는 계속 한번 쉬어야 한다. 만약 branch가 나오게 된다면 Execute 단계까지 쉬어야 하기 때문에 효율성이 많이 떨어진다.



■ Control Dependency delay slot

Delay slot은 branch 명령어 다음을 nop 명령어로 채워 Execute 단계에서 Branch taken을 할 것인지 안 할 것인지를 정하여 pc를 업데이트 시켜준다. 그러면 IF_ID의 래치는 flush가 되고 알맞은 pc에 맞게 branch가 실행된다.

■ Control Dependency branch prediction

Branch prediction은 fetch 단계에서 어디로 갈 것인지 예측하는 것이다. Branch prediction에도 여러 가지가 있다. BTB(Branch Target Buffer)를 만들어 해당 pc가 branch 인지 확인을 하고 branch인 것을 확인한 뒤 Direction predictor를 사용하여 branch taken을 할 것인지 안 할 것인지를 정하고 pc를 업데이트 시켜준다. 만약 예측이 틀리게 된다면 Fetch 과정과 Decode 과정에서 얻은 결과들은 전부 flush 해주어야 한다.

◆ Always taken

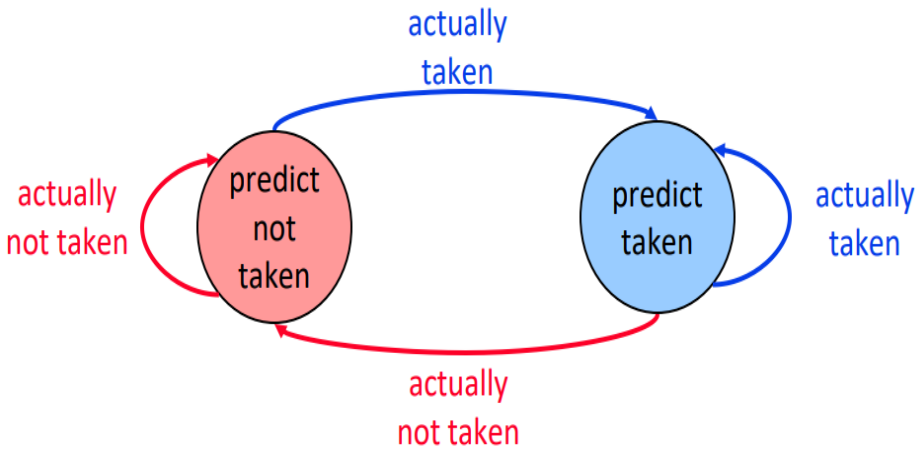
Fetch 단계에서 현재 pc가 branch 명령어인 것을 확인하면 항상 Branch taken 주소로 간다. Execute 단계에서 branch not taken인 것을 확인하면 앞선 두 래치는 flush가 된다. 그리고 pc를 현재 branch 명령어의 주소 +4를 해준다.

◆ Always not taken

Fetch 단계에서 현재 pc가 branch 명령어인 것을 확인하면 항상 Branch not taken 주소로 간다. Execute 단계에서 branch taken인 것을 확인하면 앞선 두 래치는 flush가 된다. 그리고 pc를 현재 branch target 주소로 업데이트 해준다.

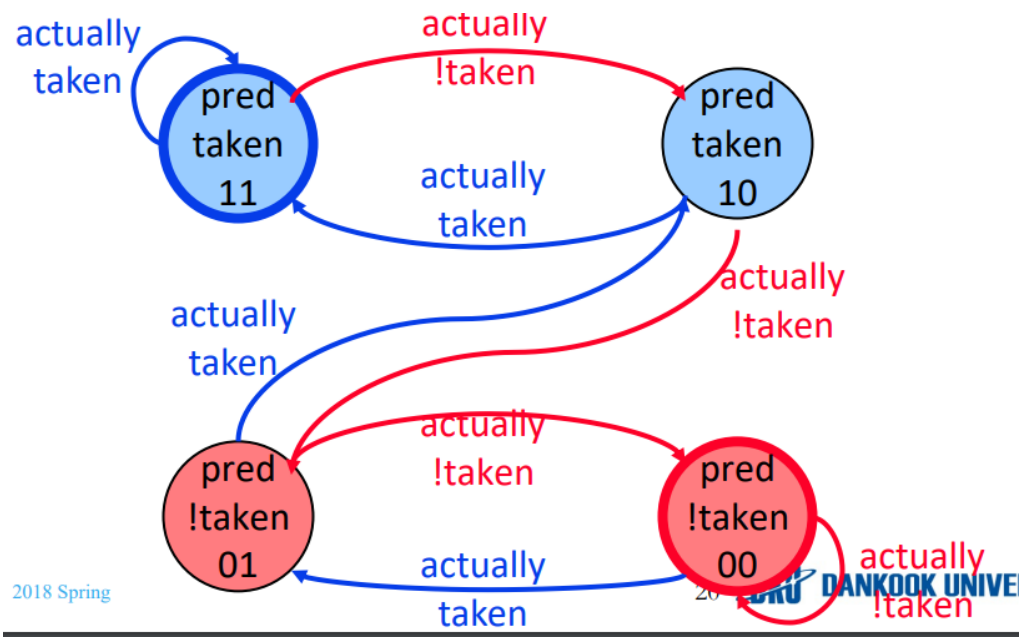
◆ Last-Time Prediction

Fetch 단계에서 현재 pc가 branch 명령어인 것을 확인하면 Direction predictor를 확인하여 1인 경우 branch taken으로 pc를 업데이트하고 Direction predictor가 0인 경우 branch not taken 주소로 pc를 업데이트한다. Execute 단계에서 예측을 실패한 것을 확인하면 앞선 두 래치는 flush가 되고 branch 결과에 맞게 pc를 업데이트 시켜준다. branch not taken인 경우 Direction predictor는 0 branch taken인 경우는 1로 업데이트 시킨다.



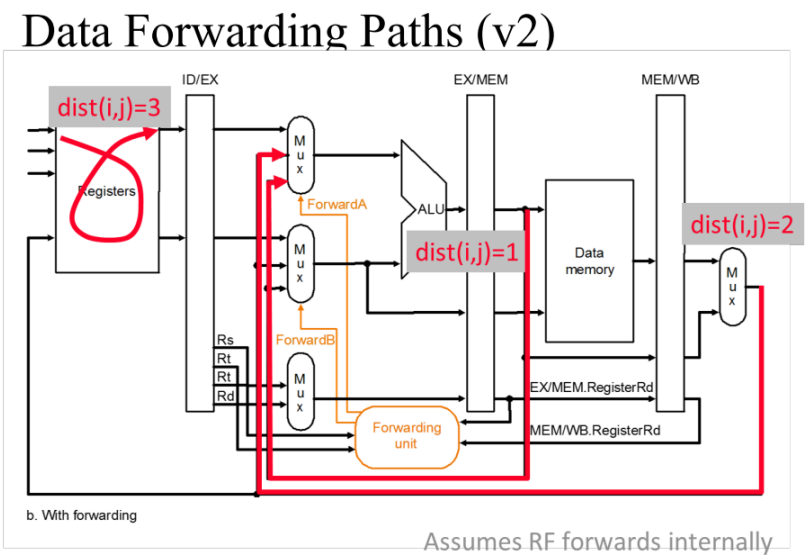
◆ Two-bit Counter Based Prediction

Fetch 단계에서 현재 pc가 branch 명령어인 것을 확인하면 Direction predictor를 확인하여 0b11 또는 0b10인 경우 branch taken으로 pc를 업데이트하고, Direction predictor가 0b01 또는 0b00인 경우 branch not taken 주소로 pc를 업데이트한다. Execute 단계에서 예측을 실패한 것을 확인하면 앞선 두 래치는 flush가 되고 branch 결과에 맞게 pc를 업데이트 시켜준다. Execute 단계에서 branch taken이라고 결정이 나면 Direction predictor에 1을 더해주고 branch not taken이라고 결정이 나면 Direction predictor에 1을 빼준다. 여기서 0b11일 때 branch taken이면 그대로 0b11이고 0b00일 때 branch not taken이면 0b00이다.



■ Data Forwarding 코드

본 레포트에 사용한 forwarding의 초석은 아래 Data Forwarding Path(v2)이다. 아래 그림을 토대로 생각해야하는 부분은 Forwarding unit이다.



Forwarding Unit은 만드는 방법은 아래 코드와 같다.

Forward A 설명

- Forward A가 3인 경우
  - Rs_id != 0
  - Rs_id==dest_wb
  - RegWrite_wb

```
if(id_ex->rs!=0 && id_ex->rs == mem_wb->write_reg&&mem_wb->RegWrite && mem_wb->MemtoReg){
    id_ex->read_data1=mem_wb->mem_read_data;
}else if(id_ex->rs == mem_wb->write_reg&&mem_wb->RegWrite){
    id_ex->read_data1=mem_wb->alu_result;
```

- Forward A가 2인 경우
  - Rs_ex != 0
  - Rs_ex==dest_wb
  - RegWrite_wb
  - Dest_mem != Dest_wb

```
//forward a 에 대한 것
if((id_ex->rs!=0)&&(mem_wb->write_reg !=ex_mem2->write_reg)&&mem_wb->write_reg ==id_ex->rs&& mem_wb->RegWrite){
    // 거리가 2일 때 rs와 rd가 같은 경우
    forward_A=2;
```

- Forward A가 1인 경우
  - Rs_ex != 0
  - Rs_ex==dest_mem
  - RegWrite_mem==1

```
}else if((id_ex->rs!=0)&&ex_mem2->write_reg==id_ex->rs  && ex_mem2->RegWrite) {
    // 거리가 1일 때 rs와 rd가 같은 경우
    forward_A=1;
}
```

- Forward_A가 0인 경우
  - 나머지 경우(Data Dependency가 없음)

Forward_A가 정해지면 forwarding을 하여 Read Data1을 업데이트 시켜준다. 물론 하드웨어 상에서는 Read Data 1을 업데이트 하여 보내는 것이 아니지만 코드상 번수가 많아져 Read Data 1을 업데이트하는 방법을 사용하였다.

Forward_A가 3인 경우 Write back 단계의 메모리에서 읽은 값을 reg[rs]에 넣을지 아니면 Write back 단계의 Alu Result 값을 reg[rs]에 넣을지 정해줘야 한다. 해당 구분은 Mem_Wb 래치에 있는 MemtoReg 와 RegWrite로 할 것이다.

```
if(id_ex->rs!=0 && id_ex->rs == mem_wb->write_reg&&mem_wb->RegWrite && mem_wb->MemtoReg){
    id_ex->read_data1=mem_wb->mem_read_data;
    reg[id_ex->rs]=mem_wb->mem_read_data;
}else if(id_ex->rs == mem_wb->write_reg&&mem_wb->RegWrite){
    id_ex->read_data1=mem_wb->alu_result;
    reg[id_ex->rs]=mem_wb->alu_result;
}else{
    id_ex->read_data1=reg[id_ex->rs];
}
```

Forward_A가 2인 경우는 2가지를 생각해야한다. 메모리를 읽어서 얻은 값을 사용할지 아니면 Alu Result 값을 사용할지 정해야 한다. 해당 부분은 mem_wb->MemtoReg이면 메모리에서 읽은 값으로 Read Data1을 업데이트하고 그게 아닌 경우는 mem_wb에 있는 Alu Result값으로 Read Data 1을 업데이트 해준다.

Forward_A가 1인 경우는 Read Data 1을 ex_mem의 Alu_Result로 업데이트 시켜준다. 코드에서 ex_mem2로 한 이유는 EX_MEM 래치를 만들 때 EX_MEM[1] 값이 이전 명령어의 Alu Result 값을 가지고 있기 때문이다.

forward_A가 0인 경우는 Decode에서 얻은 Read Data1의 결과를 그대로 가지게 된다.

```
// rs값 구해주기
if(forward_A==2){
    // 거리 2 rs와 rd가 같은 경우
    if(mem_wb->MemtoReg){
        id_ex->read_data1=mem_wb->mem_read_data;
    }else{
        id_ex->read_data1=mem_wb->alu_result;
    }
}else if(forward_A==1){
    // 거리 1 rs와 rd가 같은 경우
    id_ex->read_data1=ex_mem2->alu_result;
}else if(forward_A==0){
    id_ex->read_data1=reg[id_ex->rs];
}
```

Forward_B 설명

- Forward_B 가 3인 경우
  - Rt_id != 0
  - Rt_id==dest_wb

- RegWrite_wb

```
if(id_ex->rt == mem_wb->write_reg&&mem_wb->RegWrite && mem_wb->MemtoReg){
    id_ex->read_data2=mem_wb->mem_read_data;
}else if(id_ex->rt ==mem_wb->write_reg&& mem_wb->RegWrite){
    id_ex->read_data2=mem_wb->alu_result;
}else{
    id_ex->read_data2=reg[id_ex->rt];
}
```

- Forward_B 가 2인 경우
  - Rt_ex != 0
  - Rt_ex==dest_wb
  - RegWrite_wb
  - Dest_mem != Dest_wb

```
//forward b 에 대한 것
if ((id_ex->rt!=0)&&(mem_wb->write_reg !=ex_mem2->write_reg)&&mem_wb->write_reg ==id_ex->rt&& mem_wb->RegWrite){
    // 거리가 2일 때 rt와 rd가 같은 경우
    forward_B=2;
```

- Forward_B 가 1인 경우
  - Rt_ex != 0
  - Rt_ex==dest_mem
  - RegWrite_mem==1

```
}else if((id_ex->rt!=0)&&ex_mem2->write_reg==id_ex->rt&&ex_mem2->RegWrite){
        // 거리가 1일 때 rt와 rd가 같은 경우
    forward_B=1;
}
```

- Forward_B가 0인 경우
  - 나머지 경우

Forward_B가 정해지면 forwarding을 하여 Read Data2을 업데이트 시켜준다. 물론 하드웨어 상에서는 Read Data 2을 업데이트 하여 보내는 것이 아니지만 코드상 번수가 많아져 Read Data 2을 업데이트하는 방법을 사용하였다.

Forward_B가 3인 경우 Write back 단계의 메모리에서 읽은 값을 reg[rt]에 넣을지 아니면 Write back 단계의 Alu Result 값을 reg[rt]에 넣을지 정해줘야 한다. 해당 구분은 Mem_Wb 래치에 있는 MemtoReg 와 RegWrite로 할 것이다.

```
if(id_ex->rt == mem_wb->write_reg&&mem_wb->RegWrite && mem_wb->MemtoReg){
    id_ex->read_data2=mem_wb->mem_read_data;
    reg[id_ex->rt]=mem_wb->mem_read_data;
}else if(id_ex->rt ==mem_wb->write_reg&& mem_wb->RegWrite){
    id_ex->read_data2=mem_wb->alu_result;
    reg[id_ex->rt]=mem_wb->alu_result;
}else{
    id_ex->read_data2=reg[id_ex->rt];
}
```

Forward_B가 2인 경우는 두 가지를 생각해야한다. 메모리를 읽어서 얻은 값을 사용할지 혹은 Alu Result 값을 사용할지 정해야 한다. 해당 부분은 mem_wb->MemtoReg이면 메모리에서 읽은 값으로 Read Data 2을 업데이트하고 그게 아닌 경우는 mem_wb에 있는 Alu Result값으로 Read Data 2을 업데이트 해준다. Forward_B가 1인 경우는 Read Data 2을 ex_mem의 Alu_Result로 업데이트 시켜준다. 코드에서 ex_mem2로 한 이유는 EX_MEM 래치를 만들 때 EX_MEM[1] 값이 이전 명령어의 Alu Result 값을 가지고 있기 때문이다. 그리고 forward_B가 0인 경우는 Decode에서 얻은 Read Data2의 결과를 그대로 가지게 된다.

```
//rt 값 구하기
if(forward_B==2){
    // 거리 2 rt와 rd가 같은 경우
    if(mem_wb->MemtoReg){
        id_ex->read_data2=mem_wb->mem_read_data;
    }else{
        id_ex->read_data2=mem_wb->alu_result;
    }
}else if(forward_B==1){
    // 거리 1 rt와 rd가 같은 경우
    id_ex->read_data2=ex_mem2->alu_result;
}else if(forward_B==0){
    id_ex->read_data2=reg[id_ex->rt];
}
```

■ Control Dependency 코드

■ BTB 코드

Branch Target Buffer를 아래와 같이 구조체로 만든다. Branch pc는 해당 pc가 branch 인지 아닌지를 확인할 때 사용되고 Target은 Branch taken이 되었을 때 주소를 prediction_bit는 last time predict와 two bit predict를 사용할 때 필요한 예측 비트이다.

```
typedef struct BTB{
    int branch_pc;
    int Target;
    int prediciton_bit;
}BTB;
```

■ BTB에 PC 존재 유무 확인 코드

BTB 구조체 배열에 Branch 명령어가 있는지 확인을 하고 존재하면 index에 pc가 있는 곳을 업데이트 시켜준다. 그리고 chk를 1로 바꾸어 준다. Chk는 num_br는 다른 pc를 가지고 있는 branch 명령어의 개수를 의미한다. Chk가 0인 경우는 어떤 prediction을 쓰는지에 따라 다르긴 하지만 prediction의 종류에 맞게 BTB 구조체 배열에 차곡차곡 업데이트 된다.

```
if(id_ex->opcode==bne||id_ex->opcode==beq){
    for(int k=0; k<=num_br;k++){
        // 현재 pc가 branch_pc에 있는가 확인해야함.
        if(btb_[k].branch_pc==ex_mem->pc_4-4){
            chk=1;
            index=k;
            break;
        }
    }
```

■ Always Taken

Always Taken prediction의 경우에는 항상 branch Target 값으로 Fetch 단계에서 pc가 branch target으로 업데이트 시켜준다. 만약에 해당 branch가 처음 나오는 pc인 경우 BTB에 없기 때문에 Execute 단계에서 BTB를 업데이트하게 된다.

```
for(int i=0;i<=num_br;i++){
    if(pc==btb_[i].branch_pc && pc!=0){
        pc=btb_[i].Target;
        valid=0;
        break;
    }
}
```

Fetch.c

Execute 단계에서 branch 명령어이나 BTB에 저장되어 있지 않은 것을 확인하면 BTB에 삽입을 해준다. Fetch 단계에서 BTB에 pc가 저장되어 있지 않은 경우에는 항상 pc+4로 pc를 업데이트 해주기 때문에 Execute 단계에서 branch taken이 되면 Execute 단계 앞의 래치들은 flush가 되어져야 하고 Branch Target 값으로 업데이트 되어야한다. ex_mem->branch_addr은 branch Target 값이다.

```
if(chk==0){
    btb_[num_br].branch_pc=ex_mem->pc_4-4;
    btb_[num_br].Target=ex_mem->branch_addr;
    if(ex_mem->zero_){
        //branch taken이다.
        pc=ex_mem->branch_addr;
        memset(if_id,false,sizeof(IF_ID));
        memset(id_ex1,false,sizeof(ID_EX));
    }
    num_br+=1;
}
```

Branch_pc.c

Execute 단계에서 BTB에 해당 branch 명령어의 pc가 저장되어 있는 경우 한가지 경우만 체크하면 된다. fetch 단계에서 항상 pc를 branch taken값으로 업데이트하기 때문에 ex_mem->zero_(zero_는 branch의 결과이다. 1인 경우 Branch taken 0인 경우 branch not taken)가 아닌 경우 pc를

btb_[index].branch_pc + 4 값으로 다시 업데이트 하고 앞선 두 래치를 flush 해줘야한다.

```
if(chk!=0 && !ex_mem->zero_&& id_ex->Br_taken){
        pc=btb_[index].branch_pc+4;
        memset(if_id,false,sizeof(IF_ID));
        memset(id_ex1,false,sizeof(ID_EX));
        num_fail+=1;
}else if(chk!=0&&ex_mem->zero_&& id_ex->Br_taken ){
    num_success+=1;
```

Branch_pc.c

■ Always not Taken

현재 pc가 branch 명령어인지 아닌지 BTB를 통해 확인을 한다. 만약 현재 pc가 branch 명령어이면 항상 btb_[i].branch_pc+4로 pc를 업데이트 시켜준다.

```
for(int i=0;i<=num_br;i++){
    if(pc==btb_[i].branch_pc && pc!=0){
        pc=btb_[i].branch_pc+4;
        valid=0;
        break;
    }
}
```

fetch.c

현재 명령어가 branch 명령어이나 BTB에 없다면 BTB를 업데이트 해준다. 그리고 branch taken이면 pc값을 Branch Address로 업데이트를 해주고 not taken이면 pc를 업데이트 하지 않는다.

```
    if(chk==0){
        btb_[num_br].branch_pc=ex_mem->pc_4-4;
        btb_[num_br].Target=ex_mem->branch_addr;
        if(ex_mem->zero_){
            //branch taken이다.
            pc=ex_mem->branch_addr;
            memset(if_id,false,sizeof(IF_ID));
            memset(id_ex1,false,sizeof(ID_EX));
        }
        num_br+=1;
    }
}
```

branch_pc.c

만약 BTB에 있다면 Branch Taken 인지 아닌지 확인해줘야 한다. Fetch 단계에서 항상 not taken이라고 예상하여 pc+4로 pc를 업데이트 하였기 때문이다. 따라서 ex_mem->zero_의 값이 1인 경우 Execute 단계 앞의 래치들은 flush해주고 pc를 btb_[index].Target 값으로 업데이트 해주어야 한다.

```
if(chk!=0 && ex_mem->zero_&& id_ex->Br_taken){
        pc=btb_[index].Target;
        memset(if_id,false,sizeof(IF_ID));
        memset(id_ex1,false,sizeof(ID_EX));
        num_fail+=1;
}else if(chk!=0&&ex_mem->zero_&& id_ex->Br_taken ){
    num_success+=1;
}
```

branch_pc.c

■ Last-Time Prediction

Last-Time Prediction은 Prediction bit를 이전 결과가 어땠는지 보고 예측을 하는 것이다. 만약 prediction bit가 1이면 Target address로 pc를 업데이트하고 prediction bit가 0이면 pc를 pc+4로 업데이트한다.

```
if(pc==btb_[i].branch_pc){
    if(btb_[i].prediciton_bit==1){
        pc=btb_[i].Target;
        valid=0;
    }else{
        pc=pc+4;
        valid=0;
    }
    break;
}
```

fetch.c

만약 BTB에 branch 명령어의 pc가 없다면 BTB에 pc값을 넣어준다. branch taken인 경우에는 prediction bit를 1로 초기화해주고 pc를 branch address로 업데이트 해주고 Execute 단계 앞의 래치는 전부 flush 해준다.

```
if(chk==0){
    btb_[num_br].branch_pc=ex_mem->pc_4-4;
    btb_[num_br].Target=ex_mem->branch_addr;
    if(ex_mem->zero_){
        btb_[num_br].prediciton_bit=1;
        pc=ex_mem->branch_addr;
        memset(if_id,false,sizeof(IF_ID));
        memset(id_ex1,false,sizeof(ID_EX));
    }else{
        btb_[num_br].prediciton_bit=0;
        pc=id_ex->pc_4;
    }

    num_br+=1;
}
```

branch_pc.c

Branch taken인 경우 앞선 예측이 맞는지 확인을 해야 한다. 앞선 예측이 무엇인지 확인하기 위해서는 prediction bit를 확인하면 된다. Prediction bit가 1이면 앞서 예측은 branch taken이라는 것이고 0이면 branch not taken이라는 것이다. 만약 branch taken이라고 예측을 했는데 틀린 경우 prediction bit를 0으로 업데이트 하고 pc를 branch 명령어의 pc에 +4를 해준 값으로 pc를 업데이트 시키고 앞선 래치들은 전부 flush 해준다.

```
if(chk!=0&&ex_mem->zero_&& ex_mem->Br_taken && btb_[index].prediciton_bit==1){
        //branch 성공한 경우 그냥 놔두면 됨 어차피 예측 성공이니까.
        num_success+=1;
}else if(chk!=0&&!ex_mem->zero_&& ex_mem->Br_taken && btb_[index].prediciton_bit==1){
    //predict 실패
    btb_[index].prediciton_bit=0;
    pc=btb_[index].branch_pc+4;
    num_not_branch+=1;
    memset(if_id,false,sizeof(IF_ID));
    memset(id_ex1,false,sizeof(ID_EX));
    num_fail+=1;
}
```

branch_pc.c

Branch not Taken인 경우도 앞선 예측이 맞는지 확인을 해야 한다. 앞선 예측이 무엇인지 확인하기 위해 prediction bit를 확인한다. 아래 코드와 같이 branch not taken이 정답이고 앞선 예측도 branch not taken이면 모든 단계들이 각자의 역할을 수행하고 branch not taken이 정답인데 예측을 branch taken으로 했으면 prediction bit를 1로 바꿔주고 Execute 앞의 래치들을 전부 flush 해준다. 그리고 pc는 btb_[index]. Target 값으로 업데이트 해준다.

```
else if(chk!=0&&!ex_mem->zero_&& ex_mem->Br_taken && btb_[index].prediciton_bit==0){
    // br not taken 정답
    num_not_branch+=1;
    num_success+=1;
}
else if(chk!=0&&ex_mem->zero_&& ex_mem->Br_taken && btb_[index].prediciton_bit==0){
    //branch 가야하고 예측 실패
    btb_[index].prediciton_bit=1;

    num_fail+=1;
    pc=btb_[index].Target;
    memset(if_id,false,sizeof(IF_ID));
    memset(id_ex1,false,sizeof(ID_EX));
}
```

■ Two-bit Counter Based Prediction

Two-bit Counter Based Prediction은 2비트를 사용하여 branch taken인지 아닌지를 판단한다. Prediction이 0b11 또는 0b10 인 경우는 branch taken으로 예측을 하고 0b01 또는 0b00 인 경우는 branch not taken으로 예측한다.

```
if(pc==btb_[i].branch_pc){
    if(btb_[i].prediciton_bit==3 || btb_[i].prediciton_bit==2){
        pc=btb_[i].Target;
        valid=0;
    }else{
        pc=pc+4;
        valid=0;
    }
    break;
}
```

fetch.c

BTB에 branch 명령어의 pc가 들어있지 않은 경우는 BTB를 업데이트 해준다. 만약 Execute 단계에서 얻은 branch의 결과가 branch taken이면 pc를 branch target 값으로 업데이트 시켜주고 prediction bit는 0b10을 할당한다. 그리고 Execute 단계 앞의 래치들은 전부 flush 처리해준다. 만약 branch not taken인 경우는 fetch 단계에서 pc+4를 하였으므로 prediction bit에 0b01을 할당한다.

```
if(chk==0){
    btb_[num_br].branch_pc=ex_mem->pc_4-4;
    btb_[num_br].Target=ex_mem->branch_addr;
    if(ex_mem->zero_){
        //branch taken이다.
        btb_[num_br].prediciton_bit=2;
        pc=ex_mem->branch_addr;
        memset(if_id,false,sizeof(IF_ID));
        memset(id_ex1,false,sizeof(ID_EX));
    }else{
        //branch not taken이다 .
        btb_[num_br].prediciton_bit=1;
    }

    num_br+=1;
}
```

branch_pc.c

BTB에 branch 명령어의 pc가 있는 경우 fetch에서 예측을 제대로 했는지 확인을 해야 한다. Branch의 결과가 Branch Taken인 경우 예측 결과가 맞다면 prediction bit만 1을 더해준다. 단, prediction bit는 3을 넘어서는 안 된다.

```
if(chk!=0&&ex_mem->zero_&& ex_mem->Br_taken && btb_[index].prediciton_bit>=2){
        //branch 성공한 경우 그냥 놔두면 됨 어차피 예측 성공이니까.
        //pc=ex_mem->branch_addr;
        if(btb_[index].prediciton_bit<3){
            btb_[index].prediciton_bit+=1;
        }
        num_success+=1;
}
```

branch_pc.c

예측 결과가 틀렸다면 Execute 단계 앞의 레치들을 flush 해주고 pc를 btb[index].Target 으로 업데이트 시켜준다. 그리고 실제 정답은 branch taken 이므로 prediction bit는 1 증가시켜 준다. 단, prediction bit는 3을 넘어서는 안된다.

```
else if(chk!=0&&ex_mem->zero_&& ex_mem->Br_taken && btb_[index].prediciton_bit<2){
    // branch prediction 실패
    // 실제로는 branch 하지만 branch 되지 않은 경우
    // prediction 실패하면 fetch decode 부분 실행하면 안된다. flush 해줘야한다.
    // prediction_bit 한개 줄여줘야 한다.
    if(btb_[index].prediciton_bit<3){
            btb_[index].prediciton_bit+=1;
        }
    num_fail+=1;
    pc=btb_[index].Target;
    memset(if_id,false,sizeof(IF_ID));
    memset(id_ex1,false,sizeof(ID_EX));
}
```

branch_pc.c

Branch의 결과가 Branch not taken인 경우를 생각해보자. Fetch 단계에서의 예측이 맞았다면 아래와 같이 prediction bit를 1만큼 감소시킨다. Prediction bit는 0보다 작아지면 안 된다.

```
else if(chk!=0&&!ex_mem->zero_&& ex_mem->Br_taken && btb_[index].prediciton_bit<2){
    // br not taken 정답
    if(btb_[index].prediciton_bit>0){
            btb_[index].prediciton_bit-=1;
        }
    num_not_branch+=1;
    num_success+=1;
}
```

branch_pc.c

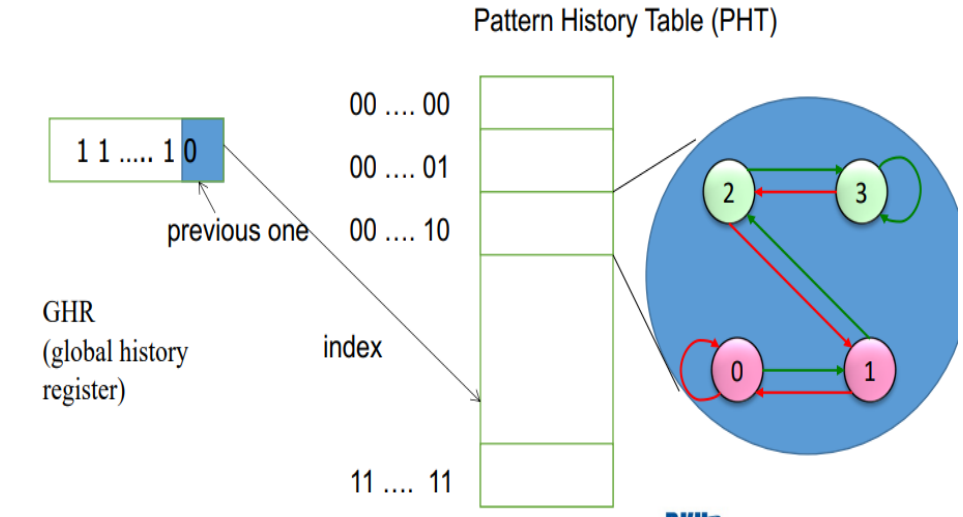
Branch의 결과가 Branch not taken이고 Fetch 단계에서 예측이 틀렸다면 아래와 같이 prediction bit를 1만큼 감소시킨다. 단, prediction bit는 0보다 작을 수 없다. 또, pc를 branch 명령어의 pc에 4를 더한 값으로 변경해주고 Execute 앞의 래치들을 flush 해준다.

```
else if(chk!=0&&!ex_mem->zero_&& ex_mem->Br_taken && btb_[index].prediciton_bit>=2){
    if(btb_[index].prediciton_bit>0){
            btb_[index].prediciton_bit-=1;
        }
    pc=btb_[index].branch_pc+4;
    num_not_branch+=1;
    memset(if_id,false,sizeof(IF_ID));
    memset(id_ex1,false,sizeof(ID_EX));
    num_fail+=1;
}
```

branch_pc.c

■ Two-Level Global Prediction

Two-Level Global prediction은 Two Bit prediction과 거의 흡사하다. 다른 점은 GHR이라는 global history register를 사용한다는 것이다. 해당 GHR은 코드 상에 다양한 branch 명령어가 있다고 가정하면 GHR을 index로 하여 pc를 prediction하는 것이다. 즉 GHR을 index로 하여 PHT(Pattern History Table)에 접근하여 2bit값을 가지고 pc를 prediction한다. GHR은 왼쪽으로 1만큼 shift하고 Branch taken인지 확인하고 branch taken이면 1을 더해주고 branch not taken이면 그대로 둔다.



eh and Patt, "Two-Level Adaptive Training Branch Prediction," MICRO 1991.30 DANKOOK UNIVERSITY

BTB에 branch 명령어가 들어가 있다면 아래와 같은 코드가 수행되고 GHR을 통해 PHT에 접근한다. PHT값이 3이거나 2이면 branch Target 값으로 pc가 업데이트 되고 PHT값이 1 또는 0이면 pc+4값으로 pc가 업데이트 된다.

```
for(int i=0;i<=num_br;i++){
    if(pc==btb_[i].branch_pc&& pc!=0){
        if(PHT[GHR]==3 || PHT[GHR]==2){
            pc=btb_[i].Target;
            valid=0;
        }else{
            pc=pc+4;
            valid=0;
        }
        break;
    }
}
```

fetch.c

만약 branch 명령어의 pc가 BTB에 존재하지 않는다면 BTB에 branch 명령어의 pc와 target address를 업데이트 해준다. 만약 branch taken이면 PHT[GHT]를 2로 업데이트 하고, 그게 아니라면 PHT[GHR]=1로 업데이트한다. 실제로는 모든 PHT값은 1로 초기화 되어있기 때문에 굳이 PHT[GHR]=1 을 할 필요는 없지만 헷갈리지 않기 위해서 적어주었다.

```
if(id_ex->opcode==bne||id_ex->opcode==beq){
    for(int k=0; k<=num_br;k++){
        // 현재 pc가 branch_pc에 있는가 확인해야함.
        if(btb_[k].branch_pc==ex_mem->pc_4-4){
            chk=1;
            index=k;
            break;
        }
    }
    if(chk==0){
        btb_[num_br].branch_pc=ex_mem->pc_4-4;
        btb_[num_br].Target=ex_mem->branch_addr;
        if(ex_mem->zero_){
            //branch taken이다.
            PHT[GHR]=2;
            pc=ex_mem->branch_addr;
            memset(if_id,false,sizeof(IF_ID));
            memset(id_ex1,false,sizeof(ID_EX));
        }else{
            //branch not taken이다 .
            PHT[GHR]=1;
        }

        num_br+=1;
    }
```

branch_pc.c

아래 코드는 2 bit prediction을 설명할 때 이야기했던 내용과 동일하다. 단 2 bit prediction 같은 경우 btb_[index].prediction을 업데이트 했지만 해당 코드에서는 PHT[GHR]로 바뀌었다는 것이다.

```
if(chk!=0&&ex_mem->zero_&& ex_mem->Br_taken && PHT[GHR]>=2){
        //branch 성공한 경우 그냥 놔두면 됨 어차피 예측 성공이니까.
        //pc=ex_mem->branch_addr;
        if(PHT[GHR]<3){
            PHT[GHR]+=1;
        }
        num_success+=1;
}
else if(chk!=0&&ex_mem->zero_&& ex_mem->Br_taken && PHT[GHR]<2){
    // branch prediction 실패
    // 실제로는 branch 하지만 branch 되지 않은 경우
    // prediction 실패하면 fetch decode 부분 실행하면 안된다. flush 해줘야한다.
    // prediction_bit 한개 줄여줘야 한다.
    if(PHT[GHR]<3){
            PHT[GHR]+=1;
        }
    num_fail+=1;
    pc=btb_[index].Target;
    memset(if_id,false,sizeof(IF_ID));
    memset(id_ex1,false,sizeof(ID_EX));
}
```

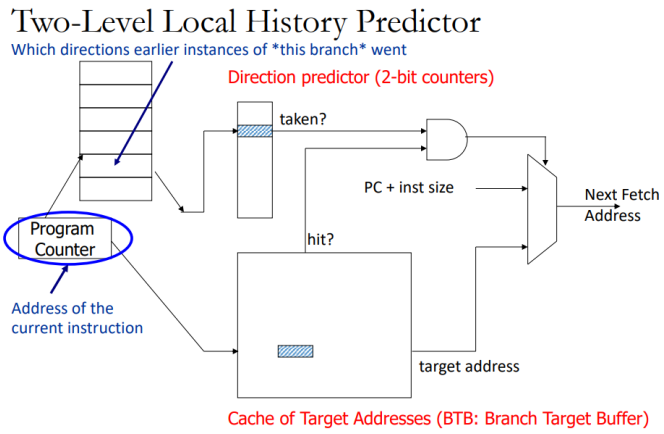
branch_pc.c

또 한 가지 다른 점은 GHR을 업데이트 시켜줘야 한다는 점이다. GHR은 한 번 예측이 끝나면 Left shift를 한번 해주고 Branch Taken이면 1 Branch Not taken이면 0을 더해준다. 그래서 branch_pc 함수 마지막에는 아래 코드가 들어가야 한다.

```
if(ex_mem->Br_taken){
    GHR<<=1;
    GHR|=ex_mem->zero_;
}
```

■ Two-Level Local Prediction

Local prediction은 Global prediction에서 한 가지가 다르다. Global prediction같은 경우에는 동일한 GHR과 동일한 PHT를 사용하였다. 하지만 Local prediction에서는 각 branch 명령어에 따라 각기 다른 GHR과 PHT를 갖는다.



그래서 BTB 라는 구조체를 아래 코드와 같이 구현했다. 물론 BTB, GHR, PHT 는 따로 있어야 하지만 코드 구현상 아래와 같이 설정하는 것이 편리하다는 생각이 들어 아래와 같이 구조체를 만들었다. 그 이유는 branch 명령어 마다 GHR 과 PHT 가 있어야 하기 때문이다.

```
typedef struct BTB{
    int branch_pc;
    int Target;
    unsigned char GHR;
    int PHT[256];
}BTB;
```

먼저 Fetch 단계에서 BTB 에 branch 명령어의 pc 가 있는지 확인을 하고 btb 에 branch 명령어의 pc 가 있다면 btb_[i].PHT[btb_[i].GHR]에 접근하여 다음 pc 값을 예측한다. 존재하지 않는다면 pc+4 로 pc 를 업데이트한다.

```
for(int i=0;i<=num_br;i++){
    if(pc==btb_[i].branch_pc&&pc!=0){
        if(btb_[i].PHT[btb_[i].GHR]==3 || btb_[i].PHT[btb_[i].GHR]==2){
            pc=btb_[i].Target;
            valid=0;
        }else{
            pc=pc+4;
            valid=0;
        }
        break;
    }
}
```

fetch.c

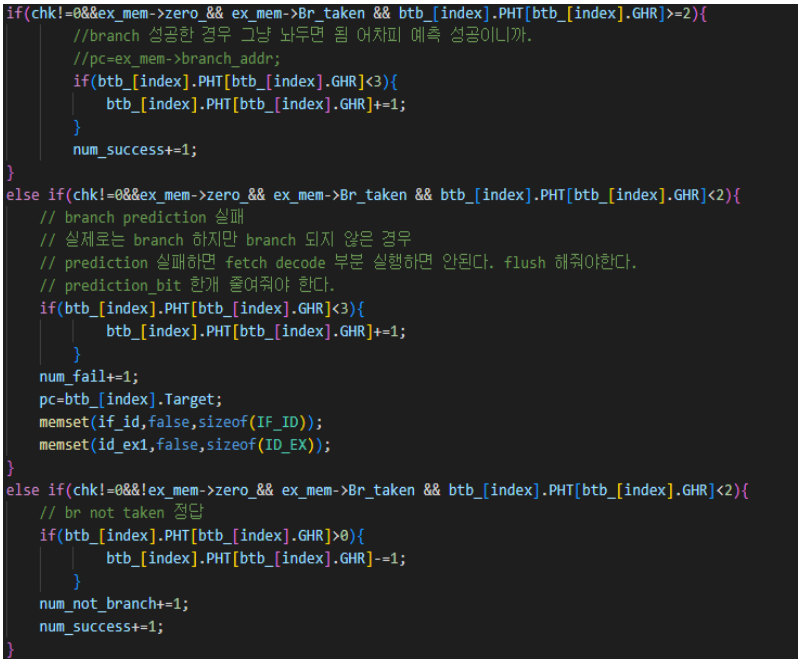
Execution 단계에서 branch 명령어가 btb 에 없다면 아래 코드와 같이 업데이트 해주어야 한다. Global prediction 과 다른 점은 btb_[num_br].PHT[btb_ [num_br].GHR]로 2bit 값을 업데이트 시켜야 한다는 점이다. 그리고 만약 앞선 과정에서 pc+4 로 pc 를 업데이트 하였으면 앞선 두 레치를 전부 flush 과정을 해줘야 한다.

```
if(chk==0){
    btb_[num_br].branch_pc=ex_mem->pc_4-4;
    btb_[num_br].Target=ex_mem->branch_addr;
    if(ex_mem->zero_){
        //branch taken이다.
        btb_[num_br].PHT[btb_[num_br].GHR]=2;
        pc=ex_mem->branch_addr;
        memset(if_id,false,sizeof(IF_ID));
        memset(id_ex1,false,sizeof(ID_EX));
    }else{
        //branch not taken이다 .
        btb_[num_br].PHT[btb_[num_br].GHR]=1;
    }

    num_br+=1;
}
```

branch_pc.c

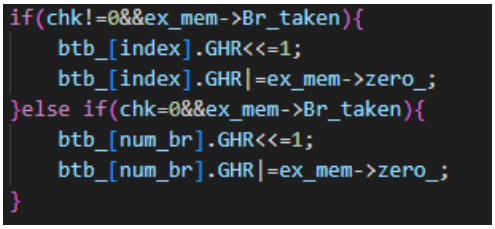
그리고 prediction 이 성공했는지 여부를 확인해야 한다. 아래 과정은 global 이나 two bit prediction 방법과 유사하다. 다른 점 하나는 btb_[index].PHT[btb_ [index].GHR]로 2bit 값을 업데이트 시킨다는 것이다. 이 과정을 통해 예측 성공을 확인하고 앞선 두 래치를 flush 를 할지 안 할지 결정을 한다.



```
if(chk!=0&&ex_mem->zero_&& ex_mem->Br_taken && btb_[index].PHT[btb_[index].GHR]>=2){
        //branch 성공한 경우 그냥 놔두면 됨 어차피 예측 성공이니까.
        //pc=ex_mem->branch_addr;
        if(btb_[index].PHT[btb_[index].GHR]<3){
            btb_[index].PHT[btb_[index].GHR]+=1;
        }
        num_success+=1;
}
else if(chk!=0&&ex_mem->zero_&& ex_mem->Br_taken && btb_[index].PHT[btb_[index].GHR]<2){
    // branch prediction 실패
    // 실제로는 branch 하지만 branch 되지 않은 경우
    // prediction 실패하면 fetch decode 부분 실행하면 안된다. flush 해줘야한다.
    // prediction_bit 한개 줄여줘야 한다.
    if(btb_[index].PHT[btb_[index].GHR]<3){
            btb_[index].PHT[btb_[index].GHR]+=1;
        }
    num_fail+=1;
    pc=btb_[index].Target;
    memset(if_id,false,sizeof(IF_ID));
    memset(id_ex1,false,sizeof(ID_EX));
}
else if(chk!=0&&!ex_mem->zero_&& ex_mem->Br_taken && btb_[index].PHT[btb_[index].GHR]<2){
    // br not taken 정답
    if(btb_[index].PHT[btb_[index].GHR]>0){
            btb_[index].PHT[btb_[index].GHR]-=1;
        }
    num_not_branch+=1;
    num_success+=1;
}
```

branch_pc.c

위 과정을 다 마치고 나서 GHR 을 업데이트 해줘야 한다. 아래 코드는 Global 과 조금 다르다. Global prediction 같은 경우에는 하나의 GHR 이 있지만 Local prediction 에서는 각 Branch 명령어마다 GHR 이 있기 때문에 아래와 같이 업데이트를 시켜 준다. 만약 btb 에 존재하는 branch 명령어라면 index 를 통해 GHR 을 업데이트 시키고, 새로 업데이트 된 branch 명령어라면 num_br 을 통해 GHR 을 업데이트 시킨다는 점이 다르다. GHR 을 업데이트 시키는 방법은 Global prediction 과 동일하다.



```
if(chk!=0&&ex_mem->Br_taken){
    btb_[index].GHR<<=1;
    btb_[index].GHR|=ex_mem->zero_;
}else if(chk=0&&ex_mem->Br_taken){
    btb_[num_br].GHR<<=1;
    btb_[num_br].GHR|=ex_mem->zero_;
}
```

branch_pc.c

■ Delay Slot

Delay Slot은 Mips compiler가 branch 명령어 뒤에 nop 명령어를 넣어주어 forwarding만 해주면 해결된다. Fetch 단계에서 항상 pc+4 값으로 pc를 업데이트하기 때문에 branch taken인 경우에 IF_ID 래치를 아래와 같이 업데이트 하면 된다. 물론 flush를 하는 방법도 있지만 pc를 업데이트하고 Decode 단계에 forwarding을 하면 코드상으로 더 구현이 편하기 때문에 아래와 같이 코드 구현을 하였다.

```
void branch_pc(IF_ID *if_id,EX_MEM * ex_mem){
    if(ex_mem->zero_ && ex_mem->Br_taken){
            pc=ex_mem->branch_addr;
            if_id->inst=Memory[pc/4];
            if_id->pc_4=pc+4;
            pc=pc+4;
        }
}
```

branch_pc.c

■ 실험 결과

Branch prediction success 와 Branch prediction fail의 개수의 합은 branch 명령어의 개수는 같아야 하지만 본인이 설계한 코드는 그 값이 다르다. 왜냐하면 BTB에 branch 명령어의 pc가 저장되어 있지 않기 때문에 실제로 예측을 성공했다고 보기 힘들기 때문이다. 그 부분을 유의하고 읽어야한다.

1. Always Taken

Simple3.bin 의 결과

```
Final result: 5050
Total number of excution cycles: 1340
Total number of instructions: 1023
Total number of memory ops 613
Total number of branches ops 102
Total number of not-taken branches ops 1
Total number of jump 2
Total number of predict success 100
Total number of predict fail 1
```

Simple4.bin 의 결과

```
Final result: 55
Total number of excution cycles: 273
Total number of instructions: 202
Total number of memory ops 100
Total number of branches ops 10
Total number of not-taken branches ops 1
Total number of jump 22
Total number of predict success 8
Total number of predict fail 1
```

Fib.bin 의 결과

```
Final result: 55
Total number of excution cycles: 3069
Total number of instructions: 2133
Total number of memory ops 1095
Total number of branches ops 109
Total number of not-taken branches ops 55
Total number of jump 274
Total number of predict success 53
Total number of predict fail 55
```

Gcd.bin 의 결과

```
Final result: 1
Total number of excution cycles: 1228
Total number of instructions: 821
Total number of memory ops 486
Total number of branches ops 73
Total number of not-taken branches ops 28
Total number of jump 103
Total number of predict success 43
Total number of predict fail 28
```

Input4.bin 의 결과

```
Final result: 85
Total number of excution cycles: 23374554
Total number of instructions: 16266206
Total number of memory ops 7116606
Total number of branches ops 2029699
Total number of not-taken branches ops 868
Total number of jump 104
Total number of predict success 2028828
Total number of predict fail 868
```

2. Always not Taken

Simple3.bin 의 결과

```
Final result: 5050
Total number of excution cycles: 1538
Total number of instructions: 1023
Total number of memory ops 613
Total number of branches ops 102
Total number of not-taken branches ops 1
Total number of jump 2
Total number of predict success 1
Total number of predict fail 100
```

Simple4.bin 의 결과

```
Final result: 55
Total number of excution cycles: 287
Total number of instructions: 202
Total number of memory ops 100
Total number of branches ops 10
Total number of not-taken branches ops 1
Total number of jump 22
Total number of predict success 1
Total number of predict fail 8
```

Fib.bin 의 결과

```
Final result: 55
Total number of excution cycles: 3065
Total number of instructions: 2133
Total number of memory ops 1095
Total number of branches ops 109
Total number of not-taken branches ops 55
Total number of jump 274
Total number of predict success 55
Total number of predict fail 53
```

Gcd.bin 의 결과

```
Final result: 1
Total number of excution cycles: 1258
Total number of instructions: 821
Total number of memory ops 486
Total number of branches ops 73
Total number of not-taken branches ops 28
Total number of jump 103
Total number of predict success 28
Total number of predict fail 43
```

Input4.bin 의 결과

```
Final result: 85
Total number of excution cycles: 27430474
Total number of instructions: 16266206
Total number of memory ops 7116606
Total number of branches ops 2029699
Total number of not-taken branches ops 869
Total number of jump 104
Total number of predict success 868
Total number of predict fail 2028828
```

3. Last-Time Prediction

Simple3.bin 의 결과

```
Final result: 5050
Total number of excution cycles: 1340
Total number of instructions: 1023
Total number of memory ops 613
Total number of branches ops 102
Total number of not-taken branches ops 1
Total number of jump 2
Total number of predict success 100
Total number of predict fail 1
```

Simple4.bin 의 결과

```
Final result: 55
Total number of excution cycles: 273
Total number of instructions: 202
Total number of memory ops 100
Total number of branches ops 10
Total number of not-taken branches ops 1
Total number of jump 22
Total number of predict success 8
Total number of predict fail 1
```

Fib.bin 의 결과

```
Final result: 55
Total number of excution cycles: 3041
Total number of instructions: 2133
Total number of memory ops 1095
Total number of branches ops 109
Total number of not-taken branches ops 55
Total number of jump 274
Total number of predict success 67
Total number of predict fail 41
```

Gcd.bin 의 결과

```
Final result: 1
Total number of excution cycles: 1198
Total number of instructions: 821
Total number of memory ops 486
Total number of branches ops 73
Total number of not-taken branches ops 28
Total number of jump 103
Total number of predict success 58
Total number of predict fail 13
```

input4.bin 의 결과

```
Final result: 85
Total number of excution cycles: 23376114
Total number of instructions: 16266207
Total number of memory ops 7116607
Total number of branches ops 2029699
Total number of not-taken branches ops 868
Total number of jump 104
Total number of predict success 2028049
Total number of predict fail 1647
```

4. Two-Bit Counter

Simple3.bin 의 결과

```
Final result: 5050
Total number of excution cycles: 1340
Total number of instructions: 1023
Total number of memory ops 613
Total number of branches ops 102
Total number of not-taken branches ops 1
Total number of jump 2
Total number of predict success 100
Total number of predict fail 1
```

Simple4.bin 의 결과

```
Final result: 55
Total number of excution cycles: 273
Total number of instructions: 202
Total number of memory ops 100
Total number of branches ops 10
Total number of not-taken branches ops 1
Total number of jump 22
Total number of predict success 8
Total number of predict fail 1
```

Fib.bin 의 결과

```
Final result: 55
Total number of excution cycles: 3065
Total number of instructions: 2133
Total number of memory ops 1095
Total number of branches ops 109
Total number of not-taken branches ops 55
Total number of jump 274
Total number of predict success 55
Total number of predict fail 53
```

Gcd.bin 의 결과

```
Final result: 1
Total number of excution cycles: 1196
Total number of instructions: 821
Total number of memory ops 486
Total number of branches ops 73
Total number of not-taken branches ops 28
Total number of jump 103
Total number of predict success 59
Total number of predict fail 12
```

input4.bin 의 결과

```
Final result: 85
Total number of excution cycles: 23374688
Total number of instructions: 16266206
Total number of memory ops 7116606
Total number of branches ops 2029699
Total number of not-taken branches ops 868
Total number of jump 104
Total number of predict success 2028761
Total number of predict fail 935
```

5. Two Level Global Prediction

Simple3.bin 의 결과

```
Final result: 5050
Total number of excution cycles: 1356
Total number of instructions: 1023
Total number of memory ops 613
Total number of branches ops 102
Total number of not-taken branches ops 1
Total number of jump 2
Total number of predict success 92
Total number of predict fail 9
```

simple4.bin 의 결과

```
Final result: 55
Total number of excution cycles: 289
Total number of instructions: 202
Total number of memory ops 100
Total number of branches ops 10
Total number of not-taken branches ops 1
Total number of jump 22
Total number of predict success 0
Total number of predict fail 9
```

Fib.bin 의 결과

```
Final result: 55
Total number of excution cycles: 3009
Total number of instructions: 2133
Total number of memory ops 1095
Total number of branches ops 109
Total number of not-taken branches ops 55
Total number of jump 274
Total number of predict success 83
Total number of predict fail 25
```

Gcd.bin 의 결과

```
Final result: 1
Total number of excution cycles: 1220
Total number of instructions: 821
Total number of memory ops 486
Total number of branches ops 73
Total number of not-taken branches ops 28
Total number of jump 103
Total number of predict success 47
Total number of predict fail 24
```

Input4.bin 의 결과

```
Final result: 85
Total number of excution cycles: 23374642
Total number of instructions: 16266206
Total number of memory ops 7116606
Total number of branches ops 2029699
Total number of not-taken branches ops 868
Total number of jump 104
Total number of predict success 2028784
Total number of predict fail 912
```

6. Two Level Local Prediction

Simple3.bin 의 결과

```
Final result: 5050
Total number of excution cycles: 1354
Total number of instructions: 1023
Total number of memory ops 613
Total number of branches ops 102
Total number of not-taken branches ops 1
Total number of jump 2
Total number of predict success 93
Total number of predict fail 8
```

Simple4.bin 의 결과

```
Final result: 55
Total number of excution cycles: 287
Total number of instructions: 202
Total number of memory ops 100
Total number of branches ops 10
Total number of not-taken branches ops 1
Total number of jump 22
Total number of predict success 1
Total number of predict fail 8
```

Fib.bin 의 결과

```
Final result: 55
Total number of excution cycles: 3007
Total number of instructions: 2133
Total number of memory ops 1095
Total number of branches ops 109
Total number of not-taken branches ops 55
Total number of jump 274
Total number of predict success 84
Total number of predict fail 24
```

Gcd.bin 의 결과

```
Final result: 1
Total number of excution cycles: 1206
Total number of instructions: 821
Total number of memory ops 486
Total number of branches ops 73
Total number of not-taken branches ops 28
Total number of jump 103
Total number of predict success 54
Total number of predict fail 17
```

Input4.bin 의 결과

```
Final result: 85
Total number of excution cycles: 23374688
Total number of instructions: 16266206
Total number of memory ops 7116606
Total number of branches ops 2029699
Total number of not-taken branches ops 868
Total number of jump 104
Total number of predict success 2028761
Total number of predict fail 935
```

7. Delay slot

Simple3.bin 의 결과

```
Final result: 5050
Total number of excution cycles: 1437
Total number of instructions: 1023
Total number of memory ops 613
Total number of branches ops 102
Total number of jump 2
```

Simple4.bin 의 결과

```
Final result: 55
Total number of excution cycles: 278
Total number of instructions: 202
Total number of memory ops 100
Total number of branches ops 10
Total number of jump 22
```

Fib.bin 의 결과

```
Final result: 55
Total number of excution cycles: 3011
Total number of instructions: 2133
Total number of memory ops 1095
Total number of branches ops 109
Total number of jump 274
```

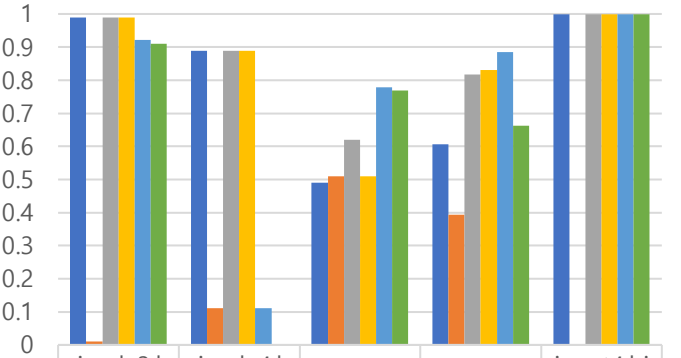
Gcd.bin 의 결과

```
Final result: 1
Total number of excution cycles: 1213
Total number of instructions: 821
Total number of memory ops 486
Total number of branches ops 73
Total number of jump 103
```

input4.bin 의 결과

```
Final result: 85
Total number of excution cycles: 25401644
Total number of instructions: 18296108
Total number of memory ops 7116606
Total number of branches ops 2029699
Total number of jump 104
```

■ 결과 분석



	simple3.bin	simple4.bin	fib.bin	gcd.bin	input4.bin
Always Taken	0.9901	0.8889	0.4907	0.6056	0.9996
Always Not Taken	0.0099	0.1111	0.5093	0.3944	0.0004
Last-Time Prediction	0.9901	0.8889	0.6204	0.8169	0.9992
Two Bit Counter	0.9901	0.8889	0.5093	0.831	0.9995
Global prediction	0.9208	0.1111	0.7778	0.8852	0.9995
Local prediction	0.9109	0	0.7685	0.662	0.9996

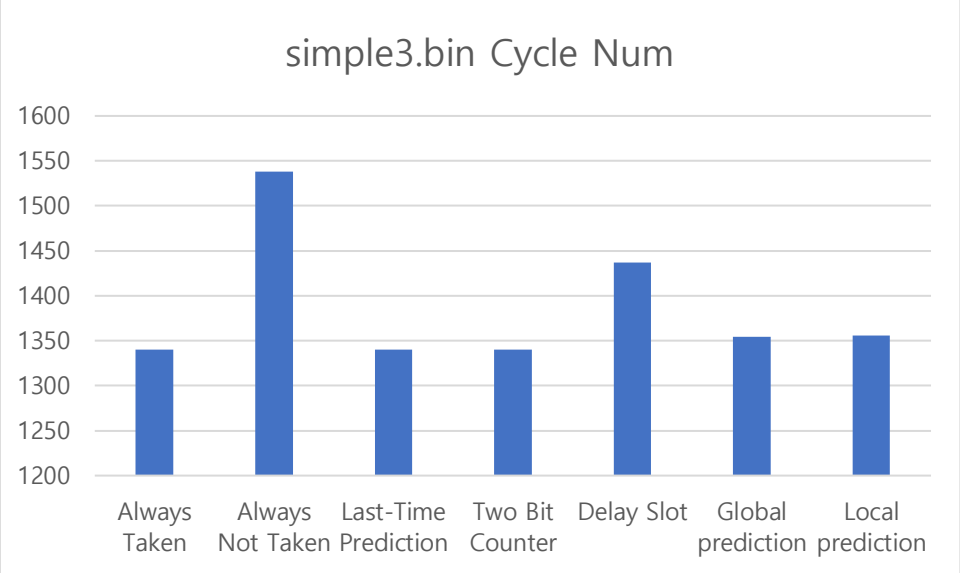
예측 성공률을 Always Taken 부터 차례대로 분석해보겠다. Always Taken 인 경우 fib.bin 파일과 gcd.bin 파일을 제외하고는 높은 정확도를 보여주고 있다. fib.bin 파일과 gcd.bin 파일은 재귀함수에 조건문이 들어가 있기 때문에 그 때 그 때 상황이 달라져서 약 50%의 성공률을 보여준다.

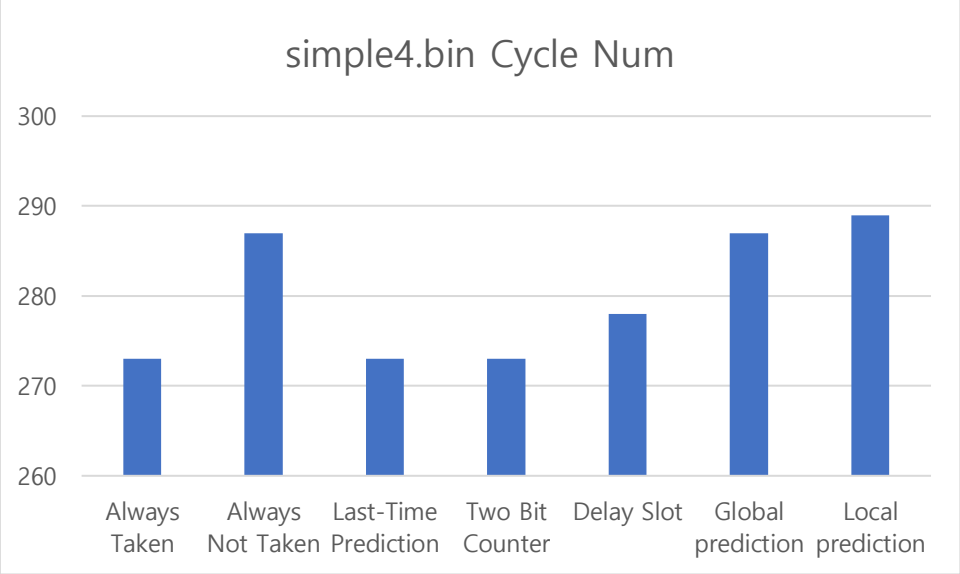
Always Not Taken 인 경우 대부분의 파일에서 예측을 잘하지 못한다. 그 이유는 대부분의 경우가 branch taken 이기 때문이다. 하지만 gcd 나 fib 의 경우는 branch taken 과 branch not taken 이 골고루 있어 약 40% 확률로 예측이 성공하는 것을 확인할 수 있다.

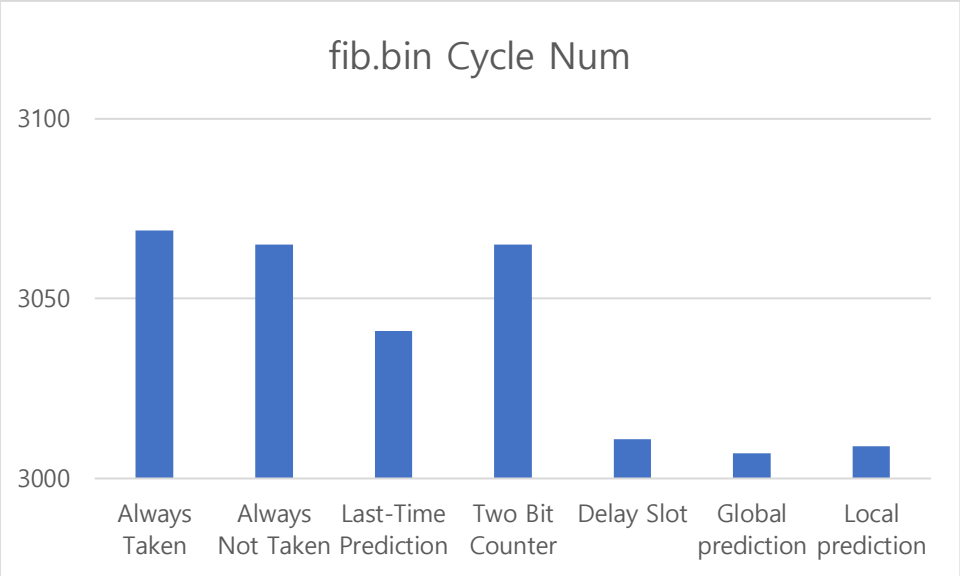
Last-Time prediction 인 경우 Fib 파일을 실행할 때 다른 prediction 에 비해 높고 나머지 파일들도 약 90% 이상으로 높은 것을 확인할 수 있다.

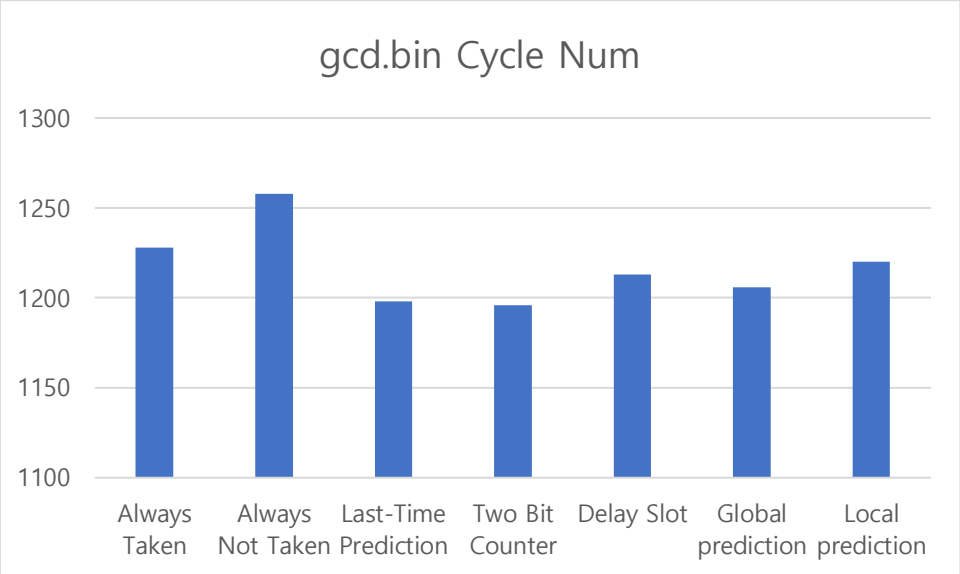
Two-Bit Counter 인 경우 fib.bin 파일을 실행할 때 약 50%로 낮은 정확도를 보여주고 나머지 파일을 실행을 시키면 다른 prediction 에 비해 높은 것을 확인할 수 있다.

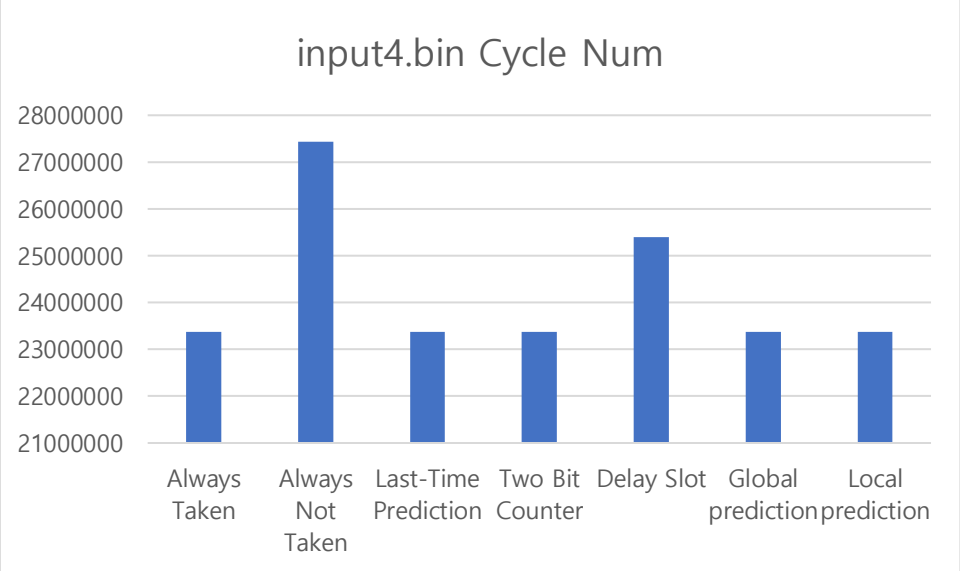
파일을 실행할 때 돌아가는 사이클 수






fib.bin Cycle Num
3100
3050
3000
Always Taken
Always Not Taken
Last-Time Prediction
Two Bit Counter
Delay Slot
Global prediction
Local prediction


gcd.bin Cycle Num
1300
1250
1200
1150
1100
Always Taken
Always Not Taken
Last-Time Prediction
Two Bit Counter
Delay Slot
Global prediction
Local prediction


input4.bin Cycle Num
28000000
27000000
26000000
25000000
24000000
23000000
22000000
21000000
Always Taken
Always Not Taken
Last-Time Prediction
Two Bit Counter
Delay Slot
Global prediction
Local prediction

위 막대 그래프를 보았을 때 prediction이 잘 된 경우는 Cycle이 나머지에 비해 낮은 것을 확인할 수 있다. 본 레포트의 파이프라인은 5단계로 나뉘어져 있다. 그래서 branch prediction이 실패할 경우 2개의 사이클을 버리게 되는 것이다. 만약 단계를 더 세세하게 나눈 파이프 라인은 어떨까? 더 많은 사이클을 낭비하게 된다. 위와 같은 이유로 아직까지 branch prediction의 정확도를 높이는 방법들이 연구되고 있다.

■프로젝트 과정의 고뇌와 느낀 점

본 프로젝트를 통해 아직도 많이 부족하다는 것을 느꼈다. 코드를 작성하기 전에 이번에는 정말 많은 생각을 하고 나름대로 잘 구상했다고 생각했다. 하지만 생각과는 다른 점이 몇 가지 있었다. 그중, 특히 Data Forwarding에서 발목을 잡혔다. 제일 먼저 Store word 명령어가 수행될 때 저장될 데이터 값은 Forwarding을 하고 한 뒤에 업데이트 했어야 했는데 앞에 있어 틀렸었다. 그렇게 simple3.bin, simple4.bin, gcd.bin, fib.bin가 돌아가니 너무 행복했다. 하지만 input4.bin에서 정말 근본도 없는 84가 나온 것이었다. 정말 황당했고 말이 나오지 않았다. 분명 앞선 파일들은 다 돌아가는데 왜 input4만 돌아가지 않는가? 라는 좌절에 빠졌었다. Single Cycle을 할 때는 795가 나온 친구들이 있었는데 그것은 lui 명령어를 구현을 하지 않아서 발생한 문제였다. 하지만. 84는 정말 생각지 못한 숫자였다. 다시 Data Forwarding의 의사 코드를 보고 교수님의 자료를 보면서 곰곰이 생각을 했다. 1시간 산책을 하면서 뭐가 문제인지 또다시 생각해보다 갑자기 떠올랐던 부분이 있었다. 바로 Memory access 단계와 Write back 단계에서 똑같은 dest를 사용한다면 Memory access 단계의 값을 forwarding을 해야 하는데 그러지 않았다는 것이다. 이후 그 부분을 수정하여 input4 파일을 돌리니 85가 나왔다. 너무 행복했다. 역시 컴퓨터는 거짓말을 하지 않는다는 것을 느꼈고 몸이 뜨거워지는 것을 느꼈다. 컴퓨터를 너무 봐서 열이 올라온 것인지 혹은 끝났다는 기쁨에 그런 것인지 그날 정말 편안한 밤을 보냈다.

다음날 branch prediction을 구현을 하기 위해 다시 컴퓨터 앞에 앉았다. 2 bit counter와 last prediction 모델을 구현하고 always taken과 always not taken을 구현해보았다. 그런데 문제가 발생했다. Always taken을 구현하였을 때 무한루프가 도는 것이었다. Branch prediction할 때 틀렸나 보다 하고 Branch prediction에 집

중하였다. 하지만 알고 보니 Branch prediction이 문제가 아니었다. Fetch에서 문제가 생긴 것이었다. pc가 업데이트가 되지 않은 것이다. 가장 큰 실수는 Branch 명령어의 pc가 BTB에 있는지 확인할 때 시작 pc가 0이고, btb를 초기화 할 때 모든 변수를 0으로 하여 pc가 업데이트가 되지 않아 무한 루프를 돌았던 것이다. 역시 디코드를 하면서 하나씩 확인하면서 하는 것이 중요하다는 것을 다시 한번 깨달았다.

컴퓨터 구조 수업을 들으면서 지금까지 3개의 프로젝트를 진행하였다. 이 과정을 통해 나날이 실력이 늘어간다고 생각을 하지만 아직은 부족한 실력인 것을 매번 깨닫는다. 앞으로도 지금의 실력에 자만하거나 만족하지 않고 더욱 더 시간을 들여 열심히 배우고 복습해야겠다고 다짐했다.