Mobile Processor

Project #3

-pipeline MIPS emulator

모바일시스템공학과

32182490

안소민

Dankook University

2020 6 20

Freedays left: 2

목차

1. Project Introduction

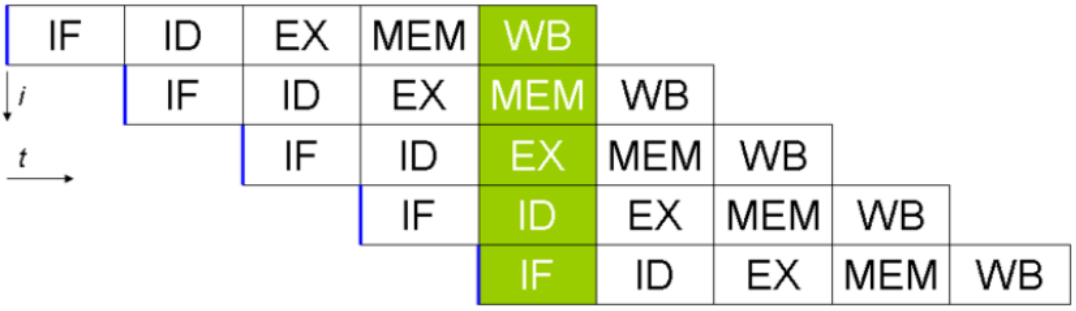
Project3 컴퓨터의 한 데이터 처리방식인 Pipeline 구조를 구현하여 Pipe MIPS emulator를 만든다. Pipeline 구성 단계인 IF, ID, Execution, Memory Access, Write Back로 나누어 구현했다. 또한 Pipeline에서 고려해야하는 Control dependency, data dependency 문제를 Data forwarding과 Branch prediction방법으로 문제를 해결하였다.

2. Project Goal

- 파이프라인 구조를 5단계를 이해하고 구현한다.
- 파이프라인에서 발생하는 dependency를 무엇이고 그 해결법을 이해하여 구현할 수 있다.
- Project2에서 single cycle 방법과 cycle 수를 비교할 수 있다.

3. Concepts used in Pipeline MIPS emulator

■ Pipeline의 구조

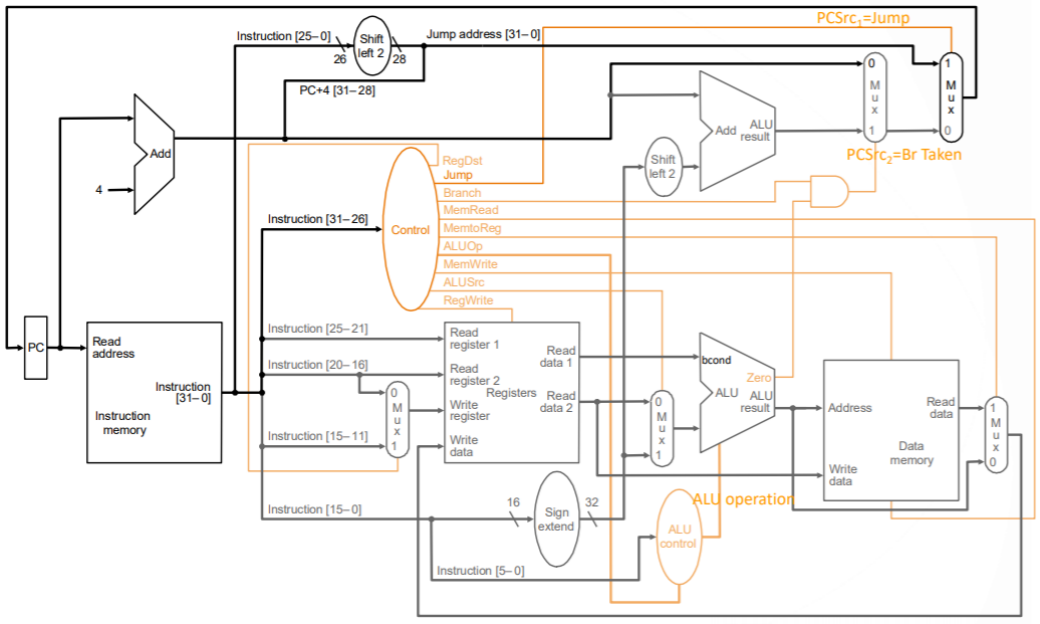


파이프라인은 이전 프로젝트에서 수행했던 single cycle processor과 달리 한 사이클에 한 명령어를 처리하는 것이 아니라 한 사이클에 여러 명령어들이 5단계로 동시에 처리되는 것으로 위와 같은 형태로 데이터가 처리되는 방식이다. 따라서 5단계에서는 모두 다른 명령어를 처리하고 있는 것이다.

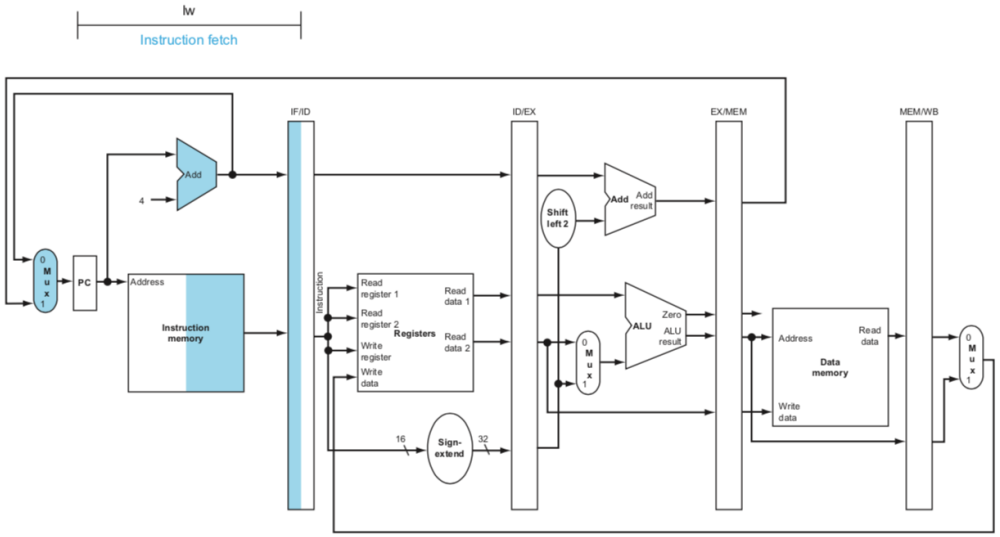
동시에 여러 데이터를 처리하기 위해서는 전 명령어 단계에서 처리했던 정보를 저장할 무언가 필요한데, latch를 이용하여 각 단계에서 처리결과 output를 저장해준다.

■ Control signal

각 단계에서 다른 명령어가 동시에 처리되는데 이를 제어하는 총 9개의 control signal이 존재하며 항상 각 latch에 저장해야 한다.



1. Instruction Fetch(IF)



일반적인 명령어의 흐름으로는 현재 명령어가 실행되는 pc값에서 +4를 하여 다음 명령어를 메모리로부터 가져온다.

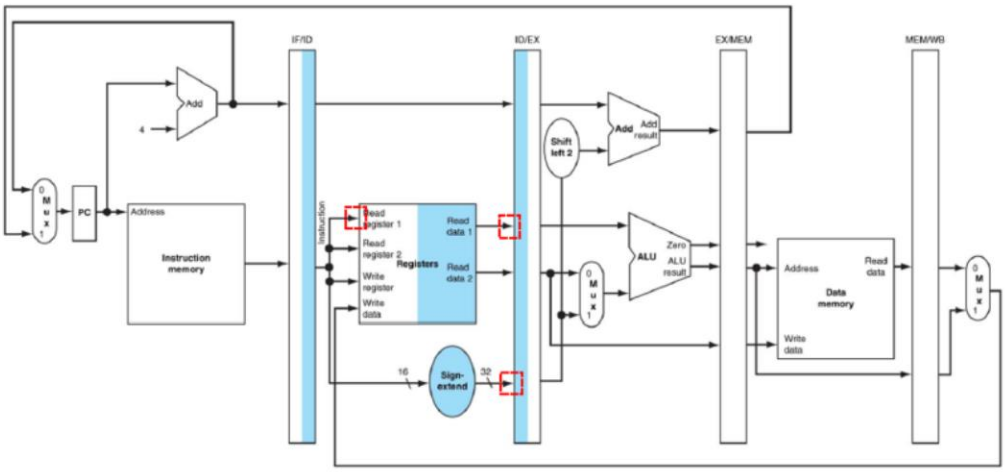
뒤에 execute, decode stage에서 분기 여부를 알 수 있는 Branch나 jump

instruction을 만났을 때 target address로 다음 pc값을 결정한다. IF단계에 나오는 output은

- ✓ pc + 4
- ✓ Instruction from Instruction from memory

이를 IF/ID Latch에 저장하여 ID단계에서 decode를 할 수 있도록 한다.

2. Instruction Decode(ID)

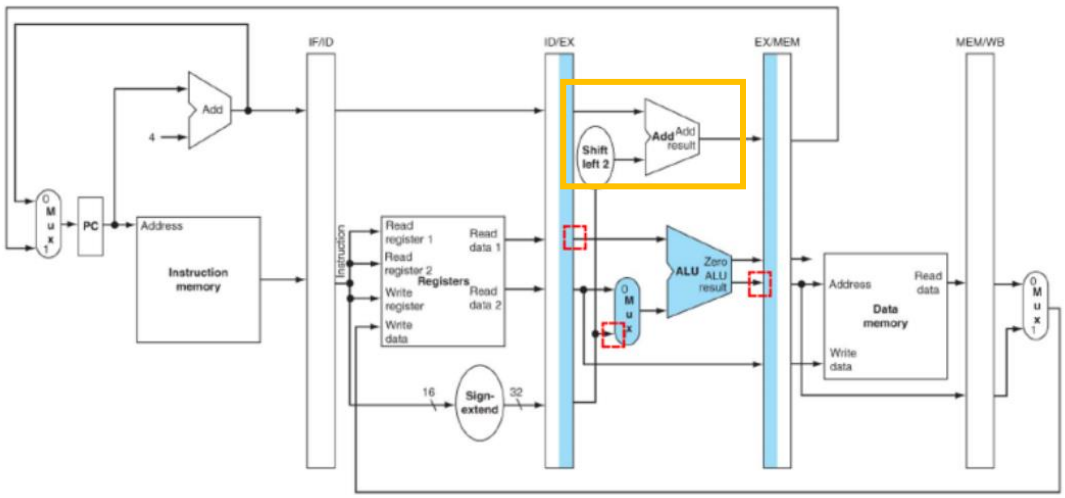


IF/ID latch로부터 명령어를 decode하는 과정으로, 32bit 명령어를 쪼개어 opcode, RD register index, rs register index, rt register index, function code, shamnt, sign/zero immediate 값을 나눈다. 그리고 해당 index register의 value를 가져온다. Opcode를 알 수 있기 때문에 control signal도 정해진다. 따라서 ID에서 나오는 output은(ID/EX latch input)

- ✓ pc + 4
- ✓ Decode한 결과 값(control signal, rd, rs, rt Reg, opcode 등)
- ✓ Write Reg index -그림에는 없지만 Write Back stage에서 계산 결과 값을 저장할 register를 정해야 하기 때문에 latch에 해당 register index를 전달한다.

Decode 단계에서 분기할 수 있는 jump instruction(jump and link, jump)은 다음 pc값이 해당 target address로 이동할 수 있도록 했다.

3. Execution



ID/EX latch에서 받은 Reg[rs] 값과 Reg[rt] 값, sign/zero immediate 값을 연산을 하는 단계이다. 해당 MUX에서는 Reg[rt], immediate value 중 어떤 값을 쓸지 ALUsrc control signal를 통해 결정한다. Ex) I type inst이라면 Mux에 1이 들어가 immediate value와 Reg[rs] 이 계산된다. 어떤 연산을 할지는 ALUOP(opcode) control signal을 통해서 계산을 할 수 있다.

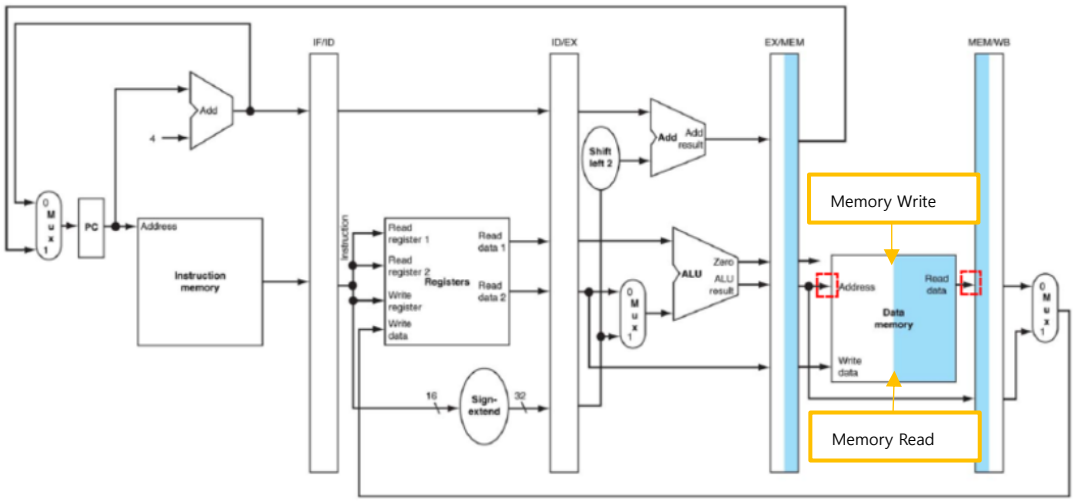
이 단계에서는 branch instruction이 들어왔을 때 두 값이 같거나(beq) 다른 것(bne)을 결과를 신호로 이용해서 target address을 계산하여 다음 pc값에 할당할 수 있다.

Execute단계에서 나오는 output은

- ✓ Pc값(branch/general pc)
- ✓ ALU통한 연산 결과 값
- ✓ Control signal
- ✓ Instruction 요소들(registers, opcode 등)
- ✓ Reg[rt](memory access 단계에서 write data의 경우 Reg[rt]를 쓴다.)

이 값들이 EX/MEM latch의 input 값이 된다.

4. Memory access



EX/MEM latch에서 들고 온 ALU 연산 결과 값은 Memory access가 필요한 store word, load word instruction 경우에 Memory의 주소 값이 된다. 아닐 경우에는 연산 결과 값이 WB stage에서 쓰이기 때문에 그림에서 화살표가 두 개로 나뉜다.

Control signal 중 Memory Write가 1일 때에는 store word 연산으로, 해당 ALU 값이 memory data의 address가 되어 그 데이터에 Reg[rt] 값을 저장한다. (WB 단계 없음)

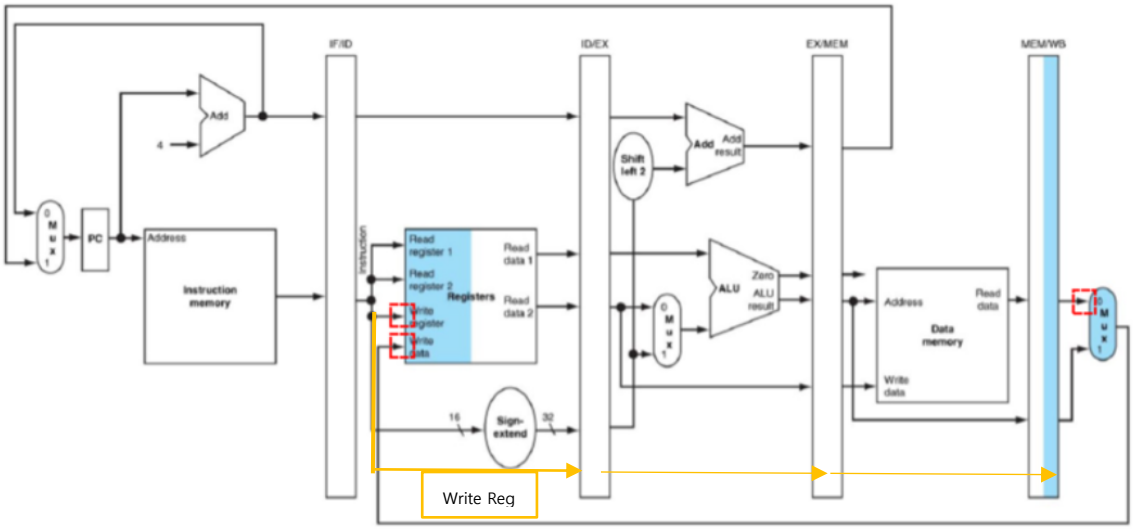
Memory Read가 1일 때에는 load word연산으로, 해당 address를 가진 데이터를 읽어오는 단계이다.

Data memory에서 해당 주소를 가진 데이터를 읽어와서 다음 단계로 전달한다.

Memory access output & MEM/WB latch input

- ✓ Data from memory
- ✓ ALU result
- ✓ Control signal

5. Write Back



MEM/WB latch에서 전달받은 ALU 결과 값이나 Memory data 값을 Register에 write 하는 단계이다. 그림에 있는 MUX는 본인은 memory access 단계에서 처리하는 것으로 구현했다. 이 MUX가 1 값을 가질 때에는 control signal MemtoReg이 1이어야 한다. memory값이 Reg에 저장되는 경우이다. ex)load word연산

그림의 노란색 화살표는 Write Reg로, write가 될 Reg index를 앞의 decode에서 계속 latch에 전달해왔다. 이 WB stage에서 해당 index Reg에 MEM/WB latch로부터 받은 결과 값을 쓴다.

■ Data dependency

Pipeline은 여러 명령어를 동시에 여러 단계에서 실행한다. 여기서 발생하는 문제점 중 하나는 Data dependency로, 만약 전 단계의 명령어가 아직 execute 단계인데, 바로 다음 명령어 decode가 되는 상태라면 이미 과거의 값을 읽어오는 것이기 때문에 그대로 두면 잘못된 연산을 하게 된다.

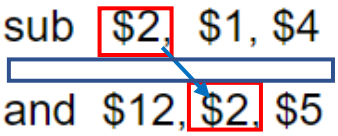
sub $2, $1, $4

and $12, $2, $5

이를 예로 들자면 and inst 가 decode단계에 있고 sub inst가 execute 단계에 있다. Reg[2]에 대하여 sub는 결과 값을 써야 하고 and inst는 그 결과 값을 기반으로 연산을 해야 하는 상황이다. 하지만 이미 decode를 해버 리기 때문에 원하는 결과 값이 나오지 않게 된다. 이를 해결할 수 있는 방법으로는 stalling과 forwarding이 대표적인 방법으로 본인은 forwarding 방법을 구현했다.

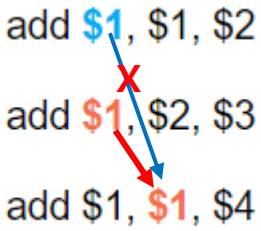
■ Data Forwarding

위에서 언급한 Data dependency의 해결 방법으로 decode 단계에서 execute 단계로 넘어오기 전에 미리 결과 값을 다음 명령어가 연산할 수 있도록 ID/EX latch에 넘겨주는 방법이다.



Data dependency는 decode 단계와 Memory access 단계사이에서도 발생한다. 위의 그림처럼 첫 번째 Inst 와 and Inst와의 dependency가 없는 Inst이 존재할 때이다. 이런 경우도 생각해서 Memory access 단계에서도 forwarding이 필요하다.

하지만 여기서 만약 두 Inst사이에 dependency가 있는 Instruction이 존재한다면 다른 문제가 추가된다.



이 때는 최근에 들어온 Inst의 Reg[1] 값이 다음 명령어의 Reg에 전달되어야 한다. 따라서 그 사이의 Inst가 RegWrite control이 1인지 또는 같은 값을 가지는 Reg에 대해서 write를 하는 지 등 잘못된 값이 전달되지 않도록 해결방법을 고려해야한다. Detect를 하는 방법으로 scoreboarding과 Combinational check logic이 있으며 본인은 Combinational check logic을 구현했다.

■ Control Dependency

일반적인 명령어가 처리가 될 때에는 기본적인 연산만 수행하는 등 pc값이 항상 +4가 되어 그 다음 명령어가 계속 처리가 된다. 하지만 Jump나 Branch는 분기가 될 수 있는 명령어를 만났을 때 그 흐름이 깨지게 된다.

Jump Instruction

Jump Inst이 Fetch 단계에 들어왔을 때에는 decode를 하지 않았기 때문에 Jump Inst인지 아닌지 모른다. Decode를 해야만 알 수 있다. 이미 decode를 했을 때에는 그 다음 Inst이 Fetch 단계에 들어와 버리기 때문에 문제가 생긴다. 이 때 앞의 명령어들을 없애는 flush하는 방법으로 target address로 pc값이 다음 사이클에 할당될 수 있는 방법으로 구현했다. 하나씩 flush되므로 cycle이 지연되는 점을 감안해야 한다.

Branch Instruction

Jump instruction과 달리 conditional type이라 조건에 맞아야만 분기를 할 수 있다. Beq는 rs rt register 값이 같아야 할 때, bne는 두 register 값이 달라야만 분기를 할 수 있다. 따라서 Execute 단계에서 분기여부를 알 수 있다. 해결 방법으로는 stall, branch prediction, branch delay slot 등 이 있다. Stall처럼 execute 단계까지 마냥 기다리는 것은 비효율적이라 그리 좋은 방법은 아니다. 따라서 제안할 수 있는 다른 방법으로 branch prediction인데 Branch instruction이 분기를 할지 말지 예측하는 방법이다.

■ Branch prediction

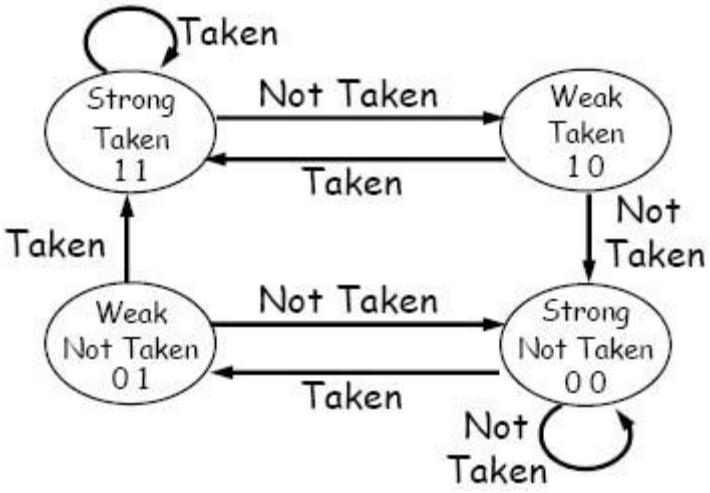
대표적으로 static branch prediction과 Dynamic branch prediction이 있다.

Static 은 predict-taken, predict-not-taken으로 branch instruction이 한 프로그램에서 15-20% 정도를 차지하는데, 단지 미리 branch inst가 나타났을 때 미리 분기하지 않을 것이다 또는 분기할 것이다 만을 예측하여 mispredict하면 flush하고 retry를 하는 방법이다. 하지만 이 방법은 나중에 예측이 틀린 것을 알게 되면 flush를 많이 하게 되는 낭비를 가져온다.

Dynamic branch prediction은 이 단점을 좀 더 보완한 방법으로, static branch prediction 방식을 수행하되 mispredict가 일어나면 유동적으로 control 흐름을 바꿀 수 있다.

■ Two-bit prediction scheme

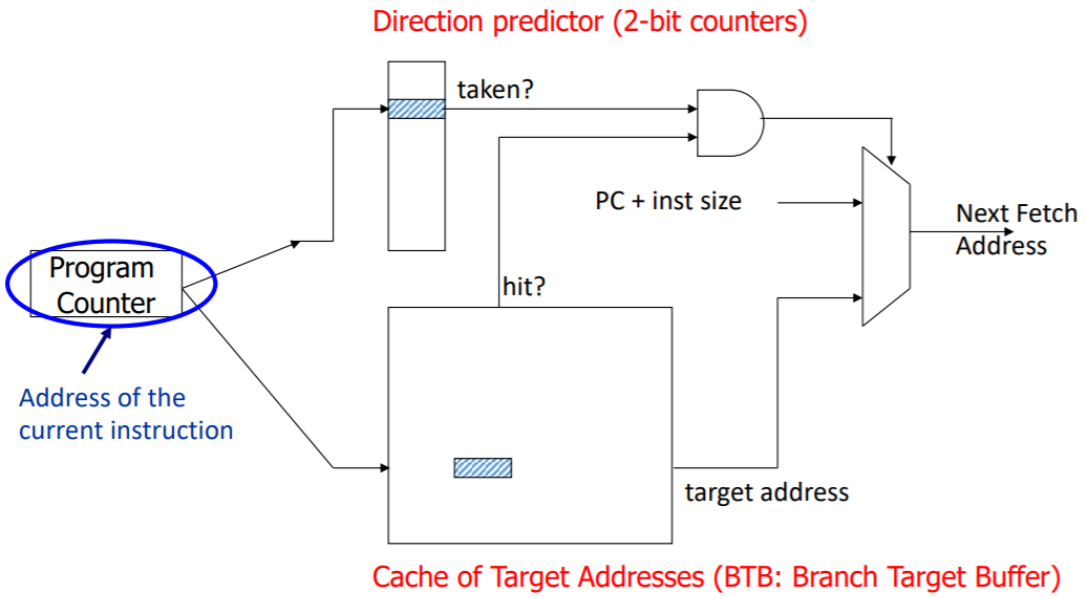
위의 Dynamic branch prediction을 수행하기 위해서 분기예측을 하는 방법으로, miss predict을 두 번 연속으로 틀릴 경우에만 state가 변경되는 되기 때문에 one bit prediction scheme처럼 틀릴 때마다 state가 변경되지 않는다. 반복문을 돌 때 거의 처음과 마지막 빼고는 계속 taken이 된다. 만약 루프문이 또 있거나 할 때 첫 번째 반복문의 마지막 루프 때 miss predict이 되더라도 taken이 유지된다. 이 scheme은 분기 예측 방법의 pattern history table로 이용된다.



Dynamic branch prediction을 수행하기 위해서는 one/two bit prediction scheme과 함께 쓰일 다른 장치가 필요한데, branch inst이 분기를 했는지와 해당 명령어를 기억하는 Branch Target Buffer가 필요하다.

이를 이용한 Dynamic Branch prediction의 방법으로는 여러가지 방법이 있는데, 그 중에서 One level Branch Predictor를 구현하였다.

■ One level Branch Predictor



Direction predictor(Pattern History Table)은 위의 Two bit counter scheme 방식을 나타낸 것으로, 분기 예측을 할지 말지 예측을 하는데 도움을 준다. BTB는 Fetch 단계에서 해당 명령어가 branch인지 기억을 하기 위해서 저장해 두는 공간이다. 이 때 명령어, target address를 저장해 둔다. 위의 그림처럼 pc를 index로 이용하여 BTB에 현재 명령어가 들어가 있고(hit) PHT에서 0b10/0b11 상태일 때, 즉 분기가 예측이 되면 pc가 target address을 가져와서 분기를 할 수 있게 된다.

4. Program Structure

```
ifid_latch[1] = ifid_latch[0];
idex_latch[1] = idex_latch[0];
exmem_latch[1] = exmem_latch[0];
memwb_latch[1] = memwb_latch[0];
```

- >updatelatch 함수

■ Main 함수 – main.c

```
int main() {

    initializeMemory();
    setting();
    while (1) {

        if(pc == 0xFFFFFFFF){
            //when -1 come in at fetch stage, remain cycles are mem write update write.
            printf("■ ■ ■ ■ ■ ■ ■ ■ ■ cycle%d ■ ■ ■ ■ ■ ■ ■ ■ ■ \n",cycle);
            writeback(memwb_latch[1]);
            printf("pc: 0x%x\n",pc);
            printf("********************terminate*****************\n");
            printf("%d\n",Reg[2]);
            break;
        }
        writeback(memwb_latch[1]);
        fetch();
        decode(ifid_latch[1]);
        execute(idex_latch[1]);
        memoryaccess(exmem_latch[1]);
        updatelatch();
    }
}
```

Pipeline 구조로 5단계로 Write Back, Fetch, Decode, Execute, Memory Access로 함수를 나누었으며 data dependency 가 일어나는 상황을 조금이라도 막기 위해서 WB를 먼저 시작하는 것으로 구조를 만들었다. 동시에 여러 명령어가 구현돼야 하기 때문에 구조체를 배열을 이용해서 각 단계의 latch output을 input으로 넣어주는 update latch()구현했다.

프로그램 종료 시, execute 단계에서 target address로 점프하는 것으로 구현했기 때문에 남은 명령어 하나가 수행해야하는 WB단계를 마지막으로 호출하여 함수를 종료한다.

■ Data Forwarding – function.c

위에서 이론을 설명한 바와 같이 Execute 단계에서 dependency와 memory access에서 data dependency가 발생할 수 있기 때문에 두 단계 함수에서 forwarding을 이용하여 해결했다. – ExHazard(), MemHazard()

execute함수

```
ForwardA = 0b00;
ForwardB = 0b00;
ExHazard(idex_input.Inst);

if(ForwardA == 0b10)//rs register..
    idex_latch[0].v1 = exmem_latch[0].v3;
else if(ForwardB == 0b10)
    idex_latch[0].v2 = exmem_latch[0].v3;
```

여기서 ForwardA,B라는 전역변수를 사용하여 A는 rs register에 대한 b는 rt register에 대한 data dependency를 detect해주는 역할을 한다. 전역변수이기 때문에 항상 다음 사이클에서는 초기화 될 수 있도록 00으로 초기화한다. Forward 전역변수의 값이 10 일 때, ALU 연산 결과 값이 전달된다. 01 값이 전달이 될 때에는 memory access 단계에서 나온 결과 값(ALU 결과 값 또는 data from Memory)를 forwarding 한다.

ExHazard함수

```
void ExHazard(int inst){
    //when Rtype instruction have data dependency
    if(exmem_latch[0].opcode == 0x00){
        if(exmem_latch[0].control.RegWrite && (idex_latch[0].rs == exmem_latch[0].rd)){
            ForwardA = 0b10;//when forwarding ALU result, forwarding signal
        }
        else if(exmem_latch[0].control.RegWrite && (idex_latch[0].rt == exmem_latch[0].rd))
            ForwardB = 0b10;//when forwarding ALU result, forwarding signal
    }
    //when I type instruction have data dependency
    if(exmem_latch[0].opcode != 0x00){
        if(exmem_latch[0].control.RegWrite && (idex_latch[0].rs == exmem_latch[0].rt)){
            if(exmem_latch[0].opcode == 0x23)LW_Hazard(inst);
            else ForwardA = 0b10;//when forwarding ALU result, forwarding signal
        }
        else if(exmem_latch[0].control.RegWrite && (idex_latch[0].rt == exmem_latch[0].rt))
        {   if(exmem_latch[0].opcode == 0x23)LW_Hazard(inst);
            else ForwardB = 0b10;//when forwarding ALU result, forwarding signal
```

값을 전달하는 Inst가 I/R type inst 인지 경우를 나누고 RegWrite == 1이고 writeReg index가 rs 또는 rt와 같은 지 if else 문을 이용하여 구현하였다.

여기서 I type에서는 Lw instruction을 만났을 때 dependency가 생기면 아직 memory access 단계에 가지 못했기 때문에 forwarding이 어려워서 LW_Hard() 호출하여 중간에 nop inst을 넣어주어서 해결했다.

MemHazard함수 일부

```
if(memwb_latch[0].control.RegWrite && (idex_latch[0].rs == memwb_latch[0].rd))
    ForwardA = 0b01;// forwarding result in memory stage

else if(memwb_latch[0].control.RegWrite && (idex_latch[0].rt == memwb_latch[0].rd))
    ForwardB = 0b01;
stopMemHazard(1);
```

Exhazard함수와 같은 방식으로 구현했다. 하지만 memhazrd 경우에는 중간에 다른 명령어가 dependency를 가지는 경우가 있기 때문에 그럴 때는 hazard를 멈추어야 하므로, stophazard()를 구현했다.

StopHazard함수

```
void stopMemHazard(int Case){
            if(exmem_latch[0].opcode == 0x00 &&(exmem_latch[0].rd == idex_latch[0].rt || exmem_latch[0].rd == idex_latch[0].rs))
                if(((memwb_latch[0].opcode != 0) &&(memwb_latch[0].rt == exmem_latch[0].rd))|| ((memwb_latch[0].opcode == 0) &&(memwb_latch[0].rd == exmem
_latch[0].rd)))
            {   ForwardA = 0;
                ForwardB = 0;
            }


else if(exmem_latch[0].opcode != 0x00 &&(exmem_latch[0].rt == idex_latch[0].rt || exmem_latch[0].rt == idex_latch[0].rs))
    if(((memwb_latch[0].opcode != 0) && (memwb_latch[0].rt == exmem_latch[0].rt))|| ((memwb_latch[0].opcode == 0) &&(memwb_latch[0].rd == exmem_latch[0].r
t)))
        if(exmem_latch[0].control.RegWrite == 1)
        {   ForwardA = 0;
            ForwardB = 0;
```

Combinational check logic 방법을 이용하다 보니 그 조건에 맞지 않으면 맘대로 프로그램이 forwarding을 실행해서 프로그램은 종료되어도 결과 값이 틀릴 때가 있었다. 이 점이 많이 고민이 되어 생각나는 경우를 모두 생각해서 통합하여 if 문의 조건으로 넣게 되었다.

처음에는 중간에 있는 명령어가 !(RegWrite Signal == 1)&&.. 로 시작해서 조건문을 만들었지만 1이여도 해당 register에 대해 data read를 하지 않거나 read가 아닌 write을 하게 되면 memhazard를 할 수 있게 된다.

경우를 다시 생각해서 Hazard를 할 수 없는 조건을 만들어서 구현했다.

이 또한 I type, R type을 나누었는데 보다 정확한 조건문을 만들기 위해서였다.

R type일 때는 rd register에 대해서 index가 같은 지 확인을 하면 되는데 만약 중간의 명령어가 I type 일 수도, R type일 수도 있기 때문에 경우를 또 나누었다.

I type 일 때는 rt 값에 대해서 write하고 R 일 때는 rd 값에 대해서 write를 하기 때문에 이와 같이 구현하게 되었다.

Else if 문에서 memory access의 명령어가 I type 일 때는 사이에 존재할 수 있는 명령어가 SW도 있기 때문에 추가 적으로 RegWrite ==1 라는 조건을 만들었다. 구현을 하면서 다른 결과 값이 나와서 애먹었는데, sw는 write를 하지 않기 때문에 hazard를 항상 할 수 있게 된다.

■ Branch prediction- one level branch predictor – branch.c

fillBTB()

```
fill_BTB(int index ,int simm){//set pc to index
BTB[index/4].pc = index;//store branch inst in BTB
BTB[index/4].target_address = index + 4 + simm*4;
```

Branch inst인 것은 decode stage에서 알 수 있기 때문에 decode 함수에서 이 함수를 호출한다. Pc 값을 index으로 하여 해당 pc 값과 target address를 저장한다.

PHT update()- two bit counter

```
location = exmem_latch[0].pc4-4;
if(branch==1){// when branch!
    if(PHT[location/4]==0b11)PHT[location/4]=0b11;//when state is 0b11(highest state)keep it.
    else
        PHT[location/4] = PHT[location/4] + 1;// else - > +1
}
else if(branch==0)//when not branch!i
{
    if(PHT[location/4]==0b00) PHT[location/4]=0b00;//when state is ob00(lowest state)keep it.
    else
        PHT[location/4] = PHT[location/4] - 1;
```

만약 분기 성공시 +1을 해서 state를 증가시키고 아닐시, -1을 해서 감소시킨다.

0b11 넘게 또는 0b00 보다 작게 stat가 변하면 안되므로 조건문을 이용했다.

One_LV_branch_predic()

```
int hit;
hit = BTB_search(pc);

if(hit && (PHT[pc/4] > 1))// 11 10 01 00
    return 1;
else return 0;
```

해당 pc값이 BTB에 존재한다면 state도 10/11일 때 분기한다.

Beq_con()

```
if(exmem_latch[0].v3 == 0)
{
    PHT_update(1);//1 means do the branch inst
    if(Presig == 0){// when miss prediction- case: did not predict branch inst, but actually branch.
        flush1(simm);
        miss++;
    }
    else
        if(already_branch !=1)branch1(simm);
    already_branch = 0;
}
else{
    PHT_update(0);//0 means do not the branch inst
```

branch inst이 beq일 때 이 함수를 수행하는 것으로, ALU 결과 값이 0 이 되었을 때

PHT_update를 수행하고, 만약 Presig(0: 분기 예측을 하지 못했을 시) 이미 그 다음 명령어들이 들어왔으므로 flush1함수(예측 못했을 시 flush하는 함수)를 수행하여 그 전 명령어들을 없앤다.

분기가 중복이 될 때가 있다. 나중에 분기를 그만할 때 조건을 보기 위해 branch Inst을 execute stage까지 남겨두었더니 반복문을 수행할 때 2번씩 분기를 하게 되어서 already branch라는 변수를 따로 두게 되었다. Else 경우, **bne_con** 함수 모두 같은 형태로 함수를 구현하였다.

5. Problems and solutions

1) Data forwarding

포워딩 개념은 잘 이해했다고 생각하고 simple1,2 파일 까지는 한 번에 잘 구현이 되었다. 하지만 simple3 파일부터 처음에 구현했던 Hazard 조건이 맞지 않았는지 계속 다른 값이 나왔다. 처음에는 코드가 많이 길진 않다 보니 single_cycle 프로젝트 했을 때 결과랑 비교하면서 틀린 경우를 찾아서 코드를 수정해 나갔다. 그렇게 디버깅을 하니까 너무 비효율적이고 시간 낭비였다. 처음부터 combinational check logic을 이용해서 detect를 하다보니 제대로 모든 경우를 담아내지 못하면 바로 값이 잘못나와 버렸다.

Input4 파일을 실행할 때 제일 힘들었다. 모니터링을 하다가 너무나도 큰 실행 파일이기 때문에 Memhazard함수에서 문제가 발생할거라고 예상하고 MemHazard함수 부분을 좀 더 자세하게 조건문을 만들어서 Hazard가 잘못되지 않게 stopHazard라는 함수를 하나 더 구현하여 문제를 해결할 수 있었다.

2) Branch prediction

분기를 하는 함수를 구현하다가 한 번은 함수가 무한으로 실행이 되었다. 그래서 출력을 해보니 분기가 중복이 되어 종료가 되지 않았었다. 나중에 분기가 종료되는 시점을 확인하기 위해서는 flush를 하더라도 branch inst는 execute stage까지는 남아 있어야한다. 그러다 보니 반복문에서는 2번 분기가 되었음을 알았고 전역변수 already_branch를 만들어서 그 문제를 해결했다.

6. Build Environment

Compilation: Linux Assam, with GCC

To compile, please type:

gcc -o pipeline branch.c function.c main.c pipeline.h

To run, please type:

./pipeline

Input4파일에서는 실행파일을 만들어서 실행했습니다.

➢ 실행파일 실행 환경

gcc branch.c function.c main.c pipeline.h -o pipeline.o

./pipeline.o> pipeline_input4

vi pip_input4

7. Screen capture

각 사이클마다 과정 출력

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle6 ■ ■ ■ ■ ■ ■ ■ ■ ■
pc: 0x18
[Fetch] 0x27bd0008
-----------------Decode-----------------------
pc: 0x14
opcode: 0x23 rs: 29 rt: 30 rd: 0 func: 0x4
Reg[29]: 0xffff8,Reg[30]: 0xffff8,Reg[0]: 0x0
simm: 4
-----------------Execute----------------------
pc: 0x10
v1: 0xffff8, v2: 0x0, simm: 0xffffe821
v4: 0x0
v3: 0xffff8
----------------memoryaccess------------------
pc: 0xc
opcode: 0x0
memory address or ALU result: 0x0
write data in memory :rt -> 0x0
value to write in Reg memory(ALU result): 0x0
-----------------writeback--------------------
pc: 0xc
no write back
```

=> v1: rs value, v2: rt value, v4: mux result(rt or simm), v3: ALU result(v1+v4)

1. Simple.bin-결과 출력

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle10 ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback--------------------
pc: 0x18
Reg[29]: 0xffff8 ----> 0x100000
pc: 0xffffffff
********************terminate*****************
******************result**********************
reg[2] (final return)value: 0
 # of instructions: 12
 # of memory operation instructions: 2
 # of register operation instructions: 8
 # of branch instructions: 0
 # of not-taken branches: 0
 # of jump instructions: 0
********************************************
```

2. Simple2.bin 결과 출력

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle12 ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback--------------------
pc: 0x20
Reg[29]: 0xfffe8 ----> 0x100000
pc: 0xffffffff
********************terminate*****************
******************result**********************
reg[2] (final return)value: 100
 # of instructions: 12
 # of memory operation instructions: 4
 # of register operation instructions: 9
 # of branch instructions: 0
 # of not-taken branches: 0
 # of jump instructions: 0
********************************************
```

3. Simple3.bin 출력

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle1339 ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback--------------------
pc: 0x64
Reg[29]: 0xfffe8 ----> 0x100000
pc: 0xffffffff
********************terminate*****************
******************result**********************
reg[2] (final return)value: 5050
 # of instructions: 28
 # of memory operation instructions: 614
 # of register operation instructions: 1029
 # of branch instructions: 102
 # of not-taken branches: 1
 # of jump instructions: 1
********************************************
```

4. Simple4.bin 출력

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle292 ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback--------------------
pc: 0x28
Reg[29]: 0xfffe0 ----> 0x100000
pc: 0xffffffff
********************terminate*****************
******************result**********************
reg[2] (final return)value: 55
 # of instructions: 44
 # of memory operation instructions: 101
 # of register operation instructions: 219
 # of branch instructions: 10
 # of not-taken branches: 1
 # of jump instructions: 11
****************************************
```

5. Fib.bin 출력

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle3282 ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback--------------------
pc: 0x34
Reg[29]: 0xfffd8 ----> 0x100000
pc: 0xffffffff
********************terminate*****************
******************result**********************
reg[2] (final return)value: 55
 # of instructions: 56
 # of memory operation instructions: 1116
 # of register operation instructions: 2405
 # of branch instructions: 109
 # of not-taken branches: 55
 # of jump instructions: 164
****************************************
```

6. Gcd.bin 출력

```
■ ■ ■ ■ ■ ■ ■ ■ ■ cycle1267 ■ ■ ■ ■ ■ ■ ■ ■ ■
-----------------writeback--------------------
pc: 0x40
Reg[29]: 0xfffd0 ----> 0x100000
pc: 0xffffffff
********************terminate*****************
******************result**********************
reg[2] (final return)value: 1
 # of instructions: 64
 # of memory operation instructions: 489
 # of register operation instructions: 938
 # of branch instructions: 73
 # of not-taken branches: 28
 # of jump instructions: 65
****************************************
```

7- input4.bin 출력

```
514268843 ■ ■ ■ ■ ■ ■ ■ ■ ■ cycle23374691 ■ ■ ■ ■ ■ ■ ■ ■ ■
514268844 ------------------writeback--------------------
514268845 pc: 0x18eec
514268846 Reg[29]: 0xf8010 ----> 0x100000
514268847 pc: 0xffffffff
514268848 ********************terminate*****************
514268849 ******************result********************
514268850 reg[2] (final return)value: 85
514268851  # of instructions: 25536
514268852  # of memory operation instructions: 7117466
514268853  # of register operation instructions: 20318758
514268854  # of branch instructions: 2029699
514268855  # of not-taken branches: 869
514268856  # of jump instructions: 103
514268857 ****************************************
```

8. Personal feeling

이번 프로젝트는 single cycle 때와 달리 너무 어렵게 느껴져서 시작을 할 수 없었다. 겁을 먹어버려서 이론 공부만 하고 정말 늦게 시작했다. 그 시기에 다른 과제가 겹쳐서 더 늦게 시작을 했더니 너무 힘들었던 것 같다. Pipeline을 구현하기 위한 개념들이 어려웠다. 여러 번 강의를 보거나 자료를 찾아봤었다. 개념이 어려우니 구현을 시작하기 까지도 조금 시간이 걸렸다. 이해를 다 하고 나서 구현을 조금씩 시작 해보니 생각보다 많이 어렵진 않았다. 이번이 시험기간이 아니었다면 더 여러가지를 구현해볼 수 있었을 것 같다. Branch prediction을 one level밖에 못 해봐서 아쉬웠다. 시간이 더 남았다면 하나 이상은 구현해 볼 수 있었을 것이다. 이번에 좋았던 것은 segmentation fault 오류는 나온 적이 없었다. Data forwarding 하는 데에만 문제가 있었다. 조건문으로 하다 보니 놓치는 케이스가 있어서 결과 값이 다르게 나온 것 같다. 이미 combinational check logic을 써서 시작해버렸기 때문에 다시 다른 방법으로 구현하는 건 포기하고 좀 더 생각해보고 구현을 했다. Input4는 항상 0이 결과 값으로 나와서 수정해도 결과 값이 안 나왔다. 시험기간이기도 하고 그냥 포기할까 생각도 했었다. 마음이 급하니까 더 어렵게 느껴졌던 것 같다. 코드 수정하는 시간은 얼마 걸리지 않았는데 바로 결과 값을 출력할 수 있었다. 개인적으로 재미있고 프로젝트였던 것 같다.