- **자신이 작성한 과제에 대한 간략한 설명**

이번 과제에서 동시에 수행되는 stage들을 수행하기 위해, data forwarding을 고려하여 역순으로 각 stage를 실행시켰다. 즉, WB→MEM→EX→ID→IF 순으로 실행시켜 pipelined execution을 구현하였다. Hazard를 해결하기 위한 forwarding이나 stall, flush 등의 알고리즘은 ppt에 있는 알고리즘을 사용하여 구현하였다. 과제 pdf대로 출력결과를 출력시켰다.

(1) 기본 작동 과정 :

1. main에서 옵션에 대한 정보를 처리 → <>옵션이 들어오지 않을 경우 에러처리
2. main에서 read_file 함수 호출
3. main에서 pipeline_exe 함수 호출

(2) pipeline_exe함수 동작 설명:

- n 옵션의 case를 나누어 각 stage별 실행을 호출하며, output()함수를 이용해 옵션에 맞게 출력한다. 각 stage는 IF_stage(), ID_stage() 등과 같이 각각 함수로 구현되어서 실행된다. pipeline_exe 함수는 이들 stage 함수를 역순으로 호출한다.

1) 각 stage 별 구현 및 pipeline state register 구현:

1. IF stage:

   IF stage를 관리하기 위해 객체 IF를 만들고(**pipeline state register는 아님**) flush, lu_hazard 멤버변수를 만들었다.

   IF stage는 PC가 pipeline state register 역할을 하기에 따로 pipeline state register를 구현하지 않았다.

```
class IF{ //IF
public:
    int flush=0;//초기값 flush아님 //flush일어나면 1로
    int lu_hazard=0;
};
```

   - IF.flush는 control hazard(J, JAL, JR, BEQ, BNE에서 발생)를 해결하기 위한 signal의 flag로 사용된다. 0일 때는 control hazard가 발생하지 않아서 IF stage가 flush되지 않는다는 의미로, PC값을 이용해 instrunction을 fetch해오고 PC값을 PC+4로 업데이트 시킨 후 IF_ID pipeline state register에 NPC로 보낸다. IF.flush가 1일 경우, IF stage를 flush시킨다.

   - IF.lu_hazard는 load-use data hazard를 해결하기 위해 사용된다. IF.lu_hazard가 1이 되면 IF stage를 stall 시킨다. (0인 경우는 load-use data hazard가 발생하지 않은 경우를 의미)

   - IF stage 작동 방식: PC값에 해당하는 주소에 Instruction이 존재할 때만 IF stage가 수행된다. IF_ID pipeline state register에 필요한 값들을 저장하여 보내준다. (저장하는 값들은 IF_ID pipeline state register의 멤버변수 값들이다.) Flush의 경우 아예 해당 stage에서 수행 중인 instruction을 날려버리는 것이므로 -p옵션출력 시 IF stage에서는 아무것도 출력되지 않도록 만들었다. stall의 경우는 해당 instruction이 머물러있는 것이므로 -p옵션출력시에 해당 instruction PC값을 출력할 수 있도록 만들었다.

2. ID stage:

   IF_ID pipeline state register를 관리하기 위해 class ID를 만들고 객체로 IF_ID를 만들었다.

   즉 사용할때는 IF_ID.Instr, IF_ID.PC 등으로 클래스 안의 멤버변수를 사용하여 IF_ID pipeline state register를 관리할 수 있다.

```
class ID{ //IF_ID를 읽어와
public:
    string Instr;
    long long PC;
    long long NPC=0;
    int flush;
};
```

- IF_ID.Instr: IF stage에서 fetch된 instruction이 32 bit binary conde로 담겨있다.
- IF_ID.PC: ID stage에서 수행할 instruction의 PC값을 가리킨다. (p옵션을 위해 사용된다.)
- IF_ID.NPC: ID stage에서 수행할 instruction의 PC+4 값을 가리킨다. 즉, 다음 instruction의 PC값을 나타내는데 이는 계속 전달되어 다른 stage에서 branch target address를 계산하기 위해 쓰인다.
- IF_ID.flush는 1일 경우에만 ID stage를 flush 시킨다.

- ID stage에서 hazard해결:

먼저 ID stage에서 **load-use data hazard**가 발생했는지 확인한 후, load-use data hazard가 일어나면 EX stage에 noop을 넣고(ID_EX.flush=1로 설정(편의상 멤버 함수명을 flush로 만듦)) 현재 IF와 ID stage의 값이 앞 stage로 이동하지 못하게 stall시킨다. 다음 사이클이 되면 EX는 noop을 수행하며 IF와 ID는 이전 사이클에서 수행했던 instruction을 똑같이 수행한다. load-use data hazard를 detect하는 방법은 ppt에 나와있는 알고리즘을 따랐는데, 이를 위해 lw, lb의 경우 ID stage에서 decode하여 ID_EX pipeline state register에 보낼 때, 미리 ID_EX.MemRead=1을 함께 저장하게 만들었다. (ppt의 ID Hazard detection Unit 알고리즘을 사용)

점프 인스트럭션의 control hazard를 막기 위해 ID stage에서 점프 인스트럭션(J, JAL, JR)을 마주치면 PC 값을 분기할 주소로 설정하고 IF.flush=1로 세팅하여 IF stage를 flush 시켰다.

조건 분기 인스트럭션(BEQ, BNE)의 control hazard를 막기 위해 ID stage에서 옵션 atp/antp정보를 확인한 후, atp일 경우에 미리 분기주소로 PC값을 설정하고 IF.flush=1로 세팅하여 IF stage를 flush 시켰다. (이후에는 Mem stage에서 EX_MEM.Zero_signal를 통해 해당 정적 예측의 성공, 실패에 따른 지연여부를 결정한다.)

- ID stage 작동 방식: Load-use data hazard가 발생하여 stall이 생기고 ID stage가 flush되는 경우를 제외하고는 ID_EX pipeline state register에 다음 사이클에 EX stage에서 수행시킬 값들($rs, $rt, IMM, offset 등)을 decode하여 전달하였다. (저장 값들은 ID_EX pipeline state register를 구성하는 멤버변수들의 값이다.).

3.EX stage:

ID_EX pipeline state register를 관리하기 위해 class EX를 만들고 객체로 ID_EX를 만들었다.

즉 사용할때는 ID_EX.Instr_type, ID_EX.PC 등으로 클래스 안의 멤버변수를 사용하여 ID_EX pipeline state register를 관리할 수 있다.

```
class EX{ //ID_EX
public:
    long long PC;
    int MemRead=0;
    long long NPC=0;
    string Instr_type;
    int Instr_funct;
    int Instr_opcode;
    int rs; //레지스터 번
    int rt;
    int rd;
    int shamt;
    long long rs_regi;
    long long rt_regi;
    long long rd_regi;
    string IMM;
    int flush;
};
```

- ID_EX.PC: EX stage의 PC값을 가리킨다. (p옵션을 위해 이용된다)

- ID_EX.MemRead: 앞서 설명한 **load-use data hazard를 해결하기** 위해 사용된다. 말그대로 M stage에서 memory를 읽어야하는 lw, lb instruction의 경우 미리 ID stage에서 ID_EX.MemRead가 1로 flag되어 저장된다. ID_EX.MemRead의 신호(0 또는 1)와 관련 레지스터 번호의 동일성(값의 의존성) 여부를 확인한 후, load-use data hazard에 해당하면 한 사이클 지연이 생길 수 있게 만들었다.

- ID_EX.NPC: 마찬가지로 EX stage에서 수행되는 instruction의 PC+4값을 가리킨다. branch target address를 구할 때 이용된다.

- ID_EX.Instr_type과 ID_EX.Instr_opcode, ID_EX.Instr_funct는 ALU()함수에서 instruction별 수행하는 ALU연산을 지정하기 위해 사용된다.

- ID_EX.rs, rt, rd는 각각 $rs, $rt, $rd 레지스터번호를 갖고있다.

- ID_EX.shamt는 모든 instruction 수행에 사용되는 것은 아니지만 어떤 instruction이 들어올지 모르므로 모든 instruction의 index 21~26에 해당하는 binary를 decimal로 바꾸어 저장되어 있다.

- ID_EX.rs_regi, rt_regi, rd_regi는 register_file에서 해당 레지스터번호에 있는 값을 저장한다.

-ID_EX.IMM의 경우도 shamt와 마찬가지로 모든 instruction 수행에 필요한 것은 아니지만 ID stage에서는 어떤 instruction이 들어올지 모르기 때문에 항상 모든 instruction에 대해 index 16번부터 0번까지의 binary를 ID_EX.IMM에 저장한다.

-ID_EX.flush의 경우 앞서 설명한 load-use data hazard를 해결하기 위해서 Ex stage에 noop을 수행하거나, 또는 Mem stage에서 -antp의 분기예측이 실패했을 때 IF,ID,EX에 잘못 fetch된 instruction을 flush시키기 위해 사용된다. (물론, ID_EX.flush는 EX stage를 flush시키며, IF와 ID는 각각 IF.flush와 IF_ID.flush를 이용해 해당 stage를 flush 시킬 수 있다.)

-EX stage 작동방식: ALU()함수를 만들어서 ALU함수안에서 ID_EX pipeline state register값을 참고하여 ALU 연산을 instruction에 맞게 수행할 수 있도록 만들었다. 또한 ALU 연산결과를 포함하여 MEM stage를 수행하기 위해 필요한 값들을 EX_MEM pipeline state register에 저장하게 만들었다. (저장하는 값들은 아래 EX_MEM pipeline state register의 멤버변수들의 값이다.)

4. MEM stage:

마찬가지로, Mem stage에서 EX_MEM pipeline state register를 사용하기 위한 class이다. 객체로 EX_MEM을 만들어서 EX_MEM.PC, EX_MEM.RegWrite 등의 표현으로 사용한다.

```
class MEM{
public:
    long long PC;
    int RegWrite=0;
    int MemWrite=0;
    int MemRead=0;
    int Zero_signal=0; //bne, beq signal
    int RegisterRd;
    unsigned int RegisterRd_regi;
    int store_source;
    int get_word=0;
    unsigned int ALU_OUT=0;
    long long BR_TARGET=0;
};
```

- EX_MEM.PC: EX stage에 있는 instruction의 PC를 가리킨다. (p 옵션에서 사용)

- EX_MEM.RegWrite는 RegWrite가 1 혹은 -1일 때 현재 Mem stage에 있는 instruction이 register_file에 write를 하는 instruction임을 의미한다. 1은 R type instruction과 lw, lb, jal 또는 그 밖에 Register에 값을 새롭게 저장해야하는 instruction일 때 -1은 lui일 때이다. instruction이 register_file을 write할지말지는 EX stage의 ALU함수를 통해 setting되고 저장되는 signal이다. 1 혹은 -1일 경우에만 후에 register_file에 write하게 된다. EX/MEM to EX data forwarding을 지원하기 위해, EX_MEM.RegWrite가 1또는 -1이면 EX/MEM to EX data hazard가 있는지 검사하고, 있다면 data forwarding을 수행한다. (ppt의 EX forward Unit알고리즘을 사용)

- EX_Mem.MemWrite는 해당 Mem stage에서 memory를 쓰는 행위가 수행되는지를 나타내는 signal이다. 1일경우, sw 또는 sb instruction을 의미하며 Memory를 write한다.

- EX_MEM.MemRead는 해당 Mem stage에서 memory를 읽는 행위가 수행되는지를 나타내는 signal이다. 1일경우, lw 또는 lb instruction을 의미하며 Memory를 Read해온다.

- EX_MEM.Zero_signal은 branch instruction이 EX stage에서 ID_EX.rs_regi와 ID_EX.rt_regi값을 비교하여 branch의 수행여부를 결정할 때, 만약 branch를 수행한다면 Zero_signal을 1로 세팅하게 된다. (0일 경우 branch를 수행하지 않는다.)

만약 옵션이 antp인데 mem stage에서 Zero_signal이 1면 EX, ID, IF stage를 flush 시킨다.

만약 옵션이 atp인데 mem stage에서 Zero_signal이 0이면 EX,ID,IF stage를 flush 시킨다.

- EX_MEM.RegisterRd_regi는 EX stage의 ALU()함수의 연산결과를 저장하고 있다. 이름은 RegisterRd_regi이지만, Rd번호에 해당하는 값만 들어있지는 않다. R type의 경우, 각 연산을 수행한 후, $rd에 넣을 값을 의미하며 I type의 경우, 각 해당 연산을 수행한 후 $rt에 넣을 값을 의미한다. J type의 Jal의 경우 $ra(31번)에 저장될 다음 실행할 명령어의 주소를 의미한다.

- EX_MEM.RegisterRd는 위의 EX_MEM.RegisterRd_regi값을 저장할 레지스터 번호를 저장하고 있다.

- EX_MEM.get_word는 lw, lb를 구분하는 signal로 memory에서 read해올 때 word단위로읽어올지말지를 결정한다. lw의 경우 1로 memory에서 word(4Byte만큼)로 읽어오며 0의 경우(lb) word로 아닌 1byte를 읽어온다.

- EX_MEM.ALU_OUT은 이전 사이클에서 EX stage에서 ALU(여기서는 ALU함수로 구현됨)를 이용하여 ALU 연산 결과로 배출된 값을 저장하고 있다.

- EX_MEM.BR_TARGET은 이전 사이클에서 EX stage에서 계산된 branch instruction의 target 주소를 가지고 있다.

- EX_MEM.store_source는 WB stage에서 설명하겠지만, **MEM/WB to MEM data hazard를 방지하기 위한 forwarding**에 사용된다. sw,sb instruction에서 $rt 레지스터 번호 값을 저장하고 있다. 이후 MEM/WB to

MEM data hazard에서 레지스터 번호 값을 비교해야하기 때문에 따로 저장하였다.

MEM stage에서 EX forward unit(EX/MEM to EX data hazard)을 처리하는데, 수업 ppt의 EX forward unit의 알고리즘을 그대로 따랐다. EX_MEM.RegWrite가 1 또는 -1일 때 EX_MEM.RegisterRd와 ID_EX.rs 혹은 ID_EX.rt 값이 같으면, ppt의 알고리즘대로 forwarding여부를 결정하고 수행하였다.

- MEM stage 작동 방식: EX/MEM to EX data hazard를 해결하기 위한 EX forwarding을 수행하고, EX_MEM.MemRead나 EX_MEM.MemWrite를 참고하여 instruction에 맞게 memory를 읽거나 쓰기를 수행한다. 또 EX_MEM.Zero_signal을 참고하여 분기 예측 실패에 따른 지연을 수행한다. 또한 다음 사이클에서 WB stage의 수행에 필요한 값들을 MEM_WB pipeline state register 멤버변수들에 저장한다.

5.WB stage:

마찬가지로, WB stage에서 MEM_WB pipeline state register를 사용하기 위한 class이다. 객체로 MEM_WB을 만들어서 MEM_WB.PC, MEM_WB.RegWrite 등의 표현으로 사용한다.

```
class WB{
public:
    long long PC;
    int RegisterRd;
    int MemRead=0;
    unsigned int RegisterRd_regi;
    int RegWrite=0;
    unsigned int ALU_OUT=0;
    string MEM_OUT;
};
```

- MEM_WB.PC: 현재 WB stage에서 수행되고 있는 instruction의 PC값
- MEM_WB.RegisterRd: 레지스터에 값을 쓰는 행위를 할 때 해당 레지스터 번호를 나타낸다.
- MEM_WB.MemRead: **MEM/WB to MEM data hazard** 해결을 위한 forwarding 여부를 확인하고 해결하기 위해 사용된다.

이때 MEM_WB.RegWrite가 1이고 EX_MEM.MemWrite값이 1일 때 EX_MEM.store_source와 MEM_WB.RegisterRd값을 같이 확인하여 data hazard가 발생하는지 확인한다.

- MEM_WB.RegisterRd_regi는 MEM_WB.RegWirte=1인 상황에서 Register에 쓰는 것들을 저장하고있다. MEM_WB.ALU_OUT 혹은 MEM_WB.MEM_OUT 둘 중 하나를 레지스터에 쓰는데, 코드의 편의를 위해 미리 MEM_WB.RegisterRd_regi에 둘 중 하나의 값을 정리하여 저장하기 때문에 존재한다. 따라서 실제 WB stage에서 Register에 쓰는 값은 MEM_WB.RegisterRd_regi에 존재한다.
- MEM_WB.RegWrite는 WB stage에서 register에 wirte하는 일을 수행해야하는지의 여부를 나타낸다.
- MEM_WB.ALU_OUT은 EX stage에서 수행된 ALU의 output을 가지고 있다.
-MEM_WB.MEM_OUT은 register에 write할 때 Memory에서 읽어온 데이터를 쓸 경우 사용하며, Memory에서 읽어온 데이터를 의미한다.

WB stage에서 **MEM/WB to EX data hazard를 처리**하는데, 수업 ppt의 MEM forward unit의 알고리즘을 따랐다. MEM_WB.RegWrite가 1 또는 -1일 때 추가적으로 레지스터 번호 비교(MEM_WB.RegisterRd와 ID_EX.rs 혹은 ID_EX.rt가 같은지 비교. 이때 EX_MEM.RegisterRd가 ID_EX.rs나 ID_EX.rt와 같지 않을 때만 data forwarding을 수행한다)를 통해 hazard가 발생하는지 확인한 후, ppt의 알고리즘대로 forwarding 여부를 결정하고 수행하였다. 또, **MEM/WB to MEM data hazard**를 해결하기 위해 MEM_WB.MemRead와 MEM_WB.RegWrite를 확인(현재 WB stage에 있는 instruction이 lw, lb인지 확인)하고 EX_MEM.MemWrite와 EX_MEM.store_sorce를 추가로 확인(현재 Mem stage에 있는 instruction이 sw,sb인지 확인, 관련 레지

스터 번호가 같은지도 확인)하여 MEM_WB.RegisterRd와 EX_MEM.store_source값이 같으면 해당 data forwarding을 수행한다.

-WB stage 작동 방식: MEM/WB to MEM data hazard, MEM/WB to Ex data hazard를 해결하기 위한 data forwarding을 지원하며, MEM_WB.RegWrite가 1 또는 -1인 경우 레지스터에 값을 쓸 수 있게 만들었다.

- **(C, C++)과제의 컴파일 방법 및 컴파일 환경(사용한 OS, 컴파일러 버전)**

C++ 로 코드를 작성하였으며 VMware workstation 환경에서 g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0 버전을 이용하여 다음 명령어로 컴파일 하였다.

g++ -o main main.cpp

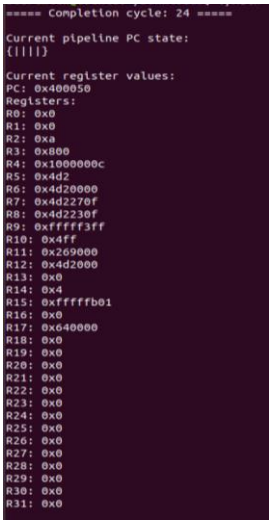
- **과제의 실행 방법 및 실행 환경**

과제 안내 pdf예제 명령어들과 유사하게 다음과 같은 명령어들로 실행하였으며, .o파일이 들어있는 파일경로에서 아래의 명령어를 이용하여 실행하였다.

예시 출력옵션과 출력결과는 다음과 같다.

Ex1) ./main -antp sample.o

atp/antp 옵션 외에 다른 옵션이 존재하지 않을 경우에는 아래와 같이 Completion cycle 형식을 출력한다.



Ex2) ./main -atp -d -p sample.o

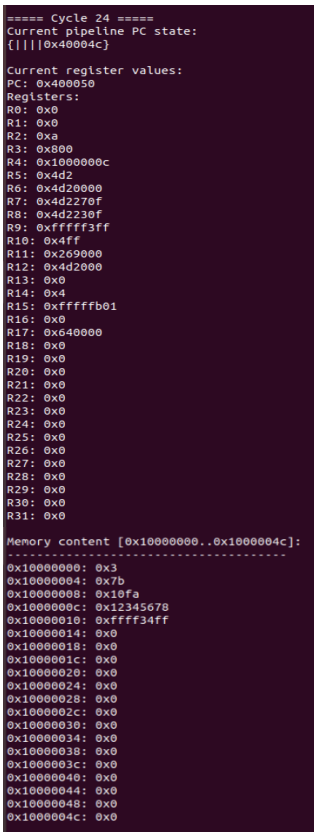
출력의 한 부분은(마지막은 Completion cycle출력) 다음과 같다.

```
===== Cycle 24 =====
Current pipeline PC state:
{||||0x40004c}

Current register values:
PC: 0x400050
Registers:
R0: 0x0
R1: 0x0
R2: 0xa
R3: 0x800
R4: 0x1000000c
R5: 0x4d2
R6: 0x4d20000
R7: 0x4d2270f
R8: 0x4d2230f
R9: 0xfffff3ff
R10: 0x4ff
R11: 0x269000
R12: 0x4d2000
R13: 0x0
R14: 0x4
R15: 0xfffffb01
R16: 0x0
R17: 0x640000
R18: 0x0
R19: 0x0
R20: 0x0
R21: 0x0
R22: 0x0
R23: 0x0
R24: 0x0
R25: 0x0
R26: 0x0
R27: 0x0
R28: 0x0
R29: 0x0
R30: 0x0
R31: 0x0
```

EX3) ./main -atp -m 0x10000000:0x1000004c -d -p sample.o

출력의 한 부분은(마지막은 Completion cycle출력) 다음과 같다.



EX4) ./main -atp -m 0x10000000:0x1000004c -d -p -n 10 sample.o

최종출력부분은 다음과 같다. (completion cycle출력은 옵션이 있을 때는 선택사항이기에 n의 개수가 정해져 있을 때는 completion cycle을 출력하지 않음. 단, -n 0 일 때 혹은 총 instruction 수를 n이 가리킬 때는 completion cycle을 출력함)