컴퓨터 구조 Project 3, Simulating Pipelined Execution

201911039 김태연

1. 수행 과제에 대한 간략한 설명

object file을 받아 instruction을 실행하는 파이프라인을 구현하는 과제이다. memory와 register는 프로그램 실행 처음에 새로 유일하게 생성되며, 1 파이프라인 cycle마다 update된다. 각 pipeline state register는 구조체 타입으로 생성하였으며, 멤버 변수로 int형 변수들을 갖는다. 5개 stage는 하나의 while loop 안에서 실행되어 파이프라인을 모방하였다. 각 stage에서의 정확한 작업 수행을 위해 state register가 보관하고 있는 값들이 이론적으로 배운 것과 완전히 동일하지는 않지만, 단계별 동작의 분리는 동일하게 구현하였다.

2-1. Pipeline state Register

1. IF_ID

- INSTR: fetch해 온 PC 주소에서 읽어 온 instruction.
- NPC: fetch해온 PC. 즉, IF stage에 와 있는 PC.

2. ID_EX

- NPC: ID stage에서 실행된 instruction의 PC.
- Rs: Instruction의 25-21번째 bit 값. instruction에 따라 사용되지 않을 수 있다.
- Rt: Instruction의 20-16번째 bit 값. instruction에 따라 사용되지 않을 수 있다.
- Rd: Instruction의 15-11번째 bit 값. instruction에 따라 사용되지 않을 수 있다.
- Immd: Instruction의 하위 16bit. instruction에 따라 사용되지 않을 수 있다.
- Shamt: Instruction의 10-6번째 bit 값. instruction에 따라 사용되지 않을 수 있다.
- JMP_TARGET: J, JAL, JR인 경우, ID stage에서 계산된 Jump target을 저장한다. Jump instruction이 아니면 이 값은 0이다.
- WB_DEST: WB이 일어나는 Instruction의 경우, Rt와 Rd 중 Instruction의 종류에 따라 알맞은 wirte register로 저장된다. 이렇게 따로 저장하는 이유는, 이후 WB stage에서 레지스터 쓰기를 진행하기 위한 signal을 단순히 하기 위해서이다.
- ReadData1: Rs에서 읽어온 값을 저장한다.
- ReadData2: Rt에서 읽어온 값을 저장한다.
- ALU_SIG: OP코드를 보고 지정된 ALU연산에 대핸 시그널을 저장한다. Instruction의

종류에 따라 서로 다른 ALU signal을 가진다. 편의상 op code, R-type의 경우 funct를 signal로 설정하였다. 같은 연산 과정을 가지는 instruction은 같은 ALU signal을 갖는다.

- WB_SIG: Write Back이 일어나는지를 나타내는 시그널. 일어나면 1, 일어나지 않으면 0으로 설정된다.
- MEM_SIG: Memory에 접근하는 작업이 일어나는지를 나타내는 시그널. 0이면 일어나지 않고, 1이면 4byte word단위의 접근이 일어나며, 2면 1byte단위의 접근이 일어난다. 이 신호(2)는 lb와 sb를 위해 사용된다.
- JMP_SIG: Jump가 일어나는지를 나타내는 시그널. 0이면 jump, 1이면 jump하지 않는다.
- BR_TARGET: Branch instruction의 경우 분기 주소가 계산되어 저장된다.

3. EX_MEM

- NPC: EX stage에서 실행된 instruction의 PC.
- Rt: ID_EX 레지스터에서 넘겨받은 Rt값. store operation을 위해 사용한다.
- WB_DEST: ID_EX 레지스터에서 넘겨받은 Write Regsister 번호.
- ALU_OUT: EX stage ALU의 연산 결과. ALU연산이 일어나지 않는 경우에는 0이다.
- JMP_TARGET: ID_EX 레지스터에서 넘겨받은 jump 목적지. MEM_WB 레지스터에 전달하기 위한 값이다. JUMP signal 대신으로 사용되어, WB stage에서 조건검사에 사용된다.
- MEM_SIG: ID_EX 레지스터에서 넘겨받은 메모리 컨트롤 시그널.
- WB_SIG: ID_EX 레지스터에서 넘겨받은 Write Back 시그널.
- BR_TARGET: ID_EX 레지스터에서 넘겨받은 분기 주소.

4. MEM_WB

- NPC: MEM stage에서 실행된 instruction의 PC를 저장한다.
- WB_DEST: EX_MEM 레지스터에서 넘겨받은 Write Regsister 번호.
- ALU_OUT: EX_MEM 레지스터에서 넘겨받은 ALU 연산 결과
- JMP_TARGET: EX_MEM 레지스터에서 넘겨받은 jump 목적지. JUMP signal 대신으로 사용되어, WB stage에서 조건검사에 사용된다.
- MEM_OUT: MEM stage에서 메모리에서 읽어온 값(읽어올 메모리 주소는 ALU_OUT에 저장되어 있었음.)

- MEM_SIG: EX_MEM 레지스터에서 넘겨받은 메모리 컨트롤 시그널. 이 시그널은 WB stage에서 ALU_OUT을 적을지, MEM_OUT을 적을지 결정하는 데 사용된다. MEM_SIG가 0이 아니면 MEM_OUT을 가져다 적는다.

- WB_SIG: EX_MEM 레지스터에서 넘겨받은 Write Back 시그널.

2-2. 전방전달

전방전달은 main.cpp에서 구현하였다. 각 forwarding 상황 당 전방전달되는 변수는 다음과 같다.

EX/MEM to EX forwarding: EX_MEM.ALU_OUT

MEM/WB to EX forwarding: MEM_WB.ALU_OUT 또는 MEM_WB.MEM_OUT

MEM/WB to MEM forwarding: MEM_WB.MEM_OUT

전방 전달을 위한 조건 검사는 아래와 같이 이루어졌다.

```
// Forwarding
// EX/MEM to EX Forwarding(arith)
if( EX_MEM.MEM_SIG == 0 &&
    EX_MEM.WB_SIG != 0 && EX_MEM.WB_DEST != 0){
        if(EX_MEM.WB_DEST == ID_EX.Rs)
                ID_EX.ReadData1 = EX_MEM.ALU_OUT;   //ForwardA
        if(EX_MEM.WB_DEST == ID_EX.Rt)
                ID_EX.ReadData2 = EX_MEM.ALU_OUT;   //ForwardB
}
// MEM/WB to EX Forwarding(arith, load)
if(MEM_WB.WB_SIG != 0 && MEM_WB.WB_DEST != 0
                && MEM_WB.WB_DEST != EX_MEM.WB_DEST){
        int forward;
        if(MEM_WB.MEM_OUT == 0)
                forward = MEM_WB.ALU_OUT;   //arith
        else
                forward = MEM_WB.MEM_OUT;   // load
        if(MEM_WB.WB_DEST == ID_EX.Rs)
               ID_EX.ReadData1 = forward;
        if(MEM_WB.WB_DEST == ID_EX.Rt)
               ID_EX.ReadData2 = forward;
}
// MEM/WB to MEM Forwarding(load)
int Mf_val = 0;
if(MEM_WB.MEM_SIG != 0 && MEM_WB.WB_SIG != 0 &&
            EX_MEM.MEM_SIG != 0 && EX_MEM.WB_SIG == 0 &&
            MEM_WB.WB_DEST == EX_MEM.Rt){ // if store follows and use same regs
        EX_MEM.Rt = -1;
        Mf_val = MEM_WB.MEM_OUT;
}
```

3. 컴파일 및 실행 방법과 환경

Ubuntu 20.04 환경에서 g++ 버전 9.3.0을 이용하여 다음 명령어로 컴파일 하였다.

```
g++ -o runfile main.cpp pipeFunc.cpp memory.cpp
```

위의 컴파일 명령으로 실행 파일인 runfile을 생성하여 다음과 같은 명령어로 실행하였다.

```
./runfile -atp sample.o
```

위의 예시에서는 필수 옵션만 포함했으나, 과제에서 제시된 바(아래)와 같이 여러 옵션을 주어 테스트했다.

./runfile <-atp 또는 -antp> [-m addr1:addr2] [-d] [-p] [-n num_instruction] <input file>

4. 실행 예시

실행 명령에 따른 최종 출력 예시는 다음과 같다.

./runfile -atp –m 0x10000000:0x10000014 -d –p sample.o

```
===== Completion cycle: 25 =====

Current pipeline PC state:
{|||}

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

Memory content [0x10000000..0x10000014]:
0x10000000: 0x3
0x10000004: 0x7b
0x10000008: 0x10fa
0x1000000c: 0x12345678
0x10000010: 0xffff34ff
0x10000014: 0x0
```

./runfile -atp –m 0x10000000:0x10000004 -d –p sample2.o

```
===== Completion cycle: 63 =====

Current pipeline PC state:
{|||}

Current register values:
PC: 0x400030
Registers:
R0: 0x0
R1: 0x1
R2: 0x0
R3: 0xf
R4: 0xf
R5: 0x0
R6: 0x0
R7: 0x0
R8: 0x10000000
R9: 0x5
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
R16: 0x0
R17: 0x0
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: 0x400010

Memory content [0x10000000..0x10000004]:
0x10000000: 0x5
0x10000004: 0x0
```

./runfile -antp –m 0x10000000:0x10000014 -d –p sample.o

```
===== Completion cycle: 25 =====

Current pipeline PC state:
{|||}

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

Memory content [0x10000000..0x10000014]:
0x10000000: 0x3
0x10000004: 0x7b
0x10000008: 0x10fa
0x1000000c: 0x12345678
0x10000010: 0xffff34ff
0x10000014: 0x0
```

./runfile -antp –m 0x10000000:0x10000004 -d –p sample2.o

```
===== Completion cycle: 50 =====

Current pipeline PC state:
{|||}

Current register values:
PC: 0x400030
Registers:
R0: 0x0
R1: 0x1
R2: 0x0
R3: 0xf
R4: 0xf
R5: 0x0
R6: 0x0
R7: 0x0
R8: 0x10000000
R9: 0x5
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
R16: 0x0
R17: 0x0
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: 0x400010

Memory content [0x10000000..0x10000004]:
0x10000000: 0x5
0x10000004: 0x0
```