Project 3 : Simulating Pipelined Execution Report

추성재 (201911185)

1. 개발 환경

   이번 프로젝트에서 사용한 언어는 C++이다. Ubuntu 20.04에서 g++ 9.4.0을 이용해 컴파일 했고, make를 이용해 빌드했다.

2. 컴파일 방법

   압축파일 속 "realProjects" 폴더 속에 Makefile이 있다. 다음과 같은 명령어를 통해 컴파일하면 된다.

   ```
   $make clean; make
   ```

   컴파일이 완료되면, "runfile"이라는 이름의 실행파일이 나온다.

3. 실행 방법

   실행파일 뒤에 과제 안내에 명시된 인자를 다음과 같이 입력 후 실행하면 된다. 인자들의 순서나 목적 파일을 제외한 인자들의 유무는 상관없다.

   ```
   ./runfile <-atp 또는 -antp> [-m addr1:addr2] [-d] [-p] [-n num_instruction] <sample.o>
   ```

4. 코드의 flow

   이번 프로젝트의 코드는 다음과 같이 네 개의 소스코드로 이루어져 있다.

   A. 00main.cpp

      메모리와 레지스터의 선언 및 초기화, 목적 파일을 불러온 후 메모리에 로드, Emulator 옵션을 받아서 실행 방식을 결정하는 등 Pipeline의 흐름 관리를 제외한 부분은 Project 2와 동일하다. 이번 프로젝트에서는 00main.cpp에서 Pipeline 구조를 반영해서 instruction들을 수행한다. 여기에는 Pipeline의 다섯 stage를 수행하고, flush나 stall을 반영해서 Pipeline Stage를 조정하는지 등이 반영되어 있다. 보다 자세한 설명은 "E. Pipeline의 구현"에 있다.

      이번 프로젝트에서는 debug mode를 추가했다. 00main.cpp에 DEBUG, DEBUG2, DEBUG_CYCLE 세 개의 bool 변수들이 있다. 이 변수들을 true로 바꾼 후 compile하면 debug mode가 활성화된다. debug mode에서는 각 pipeline stage가 끝난 후 StateReg 속 member variable들의 변화를 볼 수 있다. 또한 실행할 cycle의 개수를 직접 지정할 수도 있다.

   B. 01containers.cpp

프로젝트에서 사용되는 저장공간인 Memory(메모리), Register(레지스터), StateReg(상태 레지스터)를 class로 만들고, 이 저장공간들이 가져야 하는 member variable들과 member function들을 정의해 두었다. 또한 Pipeline, Register, Memory의 상태를 프로그램의 종료 또는 옵션에 따라 매 cycle마다 출력하는 statusCout 또한 여기에 정의되어 있다.

- Memory

  메모리를 표현하는 class이다.

Map<unsigned int, string> mem	메모리의 주소를 unsigned int 형태의 key로, 값을 string 형태의 value로 저장하는 map이다. 값들은 byte 단위, big-endian으로 저장된다.
Memory : constructor	Memory가 생성될 때 시작 주소를 받아서 Memory를 초기화한다.
MemRead	주소를 받아서 그 주소에 해당하는 값을 int 형태로 반환한다. Word 단위로 반환할 수도 있고 byte 단위로 반환할 수도 있다.
MemWrite	주소와 값을 받아서 그 주소에 해당하는 메모리 위치에 값을 저장한다. Word 단위로 저장할 수도 있고 byte 단위로도 반환할 수 있다.

- Register

  상태 레지스터를 제외한 레지스터를 나타내는 class이다.

Map<unsigned int, int> Reg	PC를 제외한 나머지 레지스터를 레지스터의 번호를 key로, key에 해당하는 레지스터의 값을 value로 하는 map으로 표현했다.
PC	PC를 표현하는 unsigned int 형태의 변수다.
Register : constructor	생성되면서 PC값을 0x400000으로 초기화하고, Reg 속 값을 0으로 초기화 한다.
RegWrite	레지스터의 번호와 값을 받으면 해당 레지스터에 값을 저장하는 member function이다.
RegRead	레지스터의 번호를 받아서 해당 레지스터의 값을 반환한다.

- StateReg

  상태 레지스터를 표현하는 class이다. StateReg는 각 상태 레지스터가 가져야 하는 값들(ALU 연산의 결과, control signal 등)을 모두 가지고 있다. StateReg가 가지고 있는 member variable들은 다음과 같이 나눌 수 있다.

i. StateReg의 정보를 나타내는 member variable

Tag	StateReg가 어떤 상태 레지스터인지를 나타낸다. Noop : 0, PC : 1, IFID : 2, IDEX : 3, EXMEM : 4, MEMWB : 5
stall	Pipeline 속에서 이 StateReg 뒤의 StateReg들을 stall 해야 하는 경우를 나타낸다. 0이면 stall이 없고, 1이면 1 cycle을 stall한다.
flush	Pipeline 속에서 이 StateReg 뒤의 StateReg들을 flush 해야 하는지 여부를 나타낸다. 0이면 flush 하지 않고, 1이면 flush한다.

ii. StateReg가 각 Pipeline의 State를 지나면서 받아오는 정보를 담는 member variable

PCaddr	Instruction이 fetch 될 때 그 instruction의 주소를 저장한다.
instruction	IF stage에서 PCaddr 주소에서 가져오는 instruction을 담고 있다.
op, rs, rt, rdt	IF stage에서 PCaddr 주소에서 가져온 instruction의 op, rs, rt, rd 값을 담고 있다.
imm	IF stage에서 PCaddr 주소에서 가져온 instruction의 imm 값을 담고 있다. imm에는 R형식의 shamt, J형식의 target 또한 포함되어 있다.
funct	IF stage에서 PCaddr 주소에서 가져온 instruction이 R형식일 경우 여기에 funct 값을 담는다.
regData1	ID stage에서 rs 레지스터의 값을 담는 곳이다.
regData2	ID stage에서 rt 레지스터의 값을 담는 곳이다.
IDjumpAddr	jal이나 always taken prediction에서 beq, bne의 jump 주소를 담는다.
aluResult	EX stage에서 ALU 연산의 결과를 담는다.
memData	MEM stage에서 메모리에 접근해 값을 읽어온 후 이곳에 그 값을 저장한다

iii. ID stage에서 CONTROL, DataForwardingUnit이 생성하는 control signal들을 담는 member variable

MemWrite	StateReg 속 instruction이 MEM stage에서 메모리에 값을 저장한다면 1, 아니라면 0으로 설정된다.
MemRead	StateReg 속 instruction이 MEM stage에서 메모리에 값을 읽어온다면 1, 아니라면 0으로 설정된다.
RegWrite	StateReg 속 instruction이 WB stage에서 레지스터에 값을 저장한다면 1, 아니라면 0으로 설정된다.
ALUop	StateReg 속 instruction이 EX stage에서 연산을 수행할 때 수행해야 할 연산의 종류를 설정한다. 연산의 종류는 add, sub, shiftleft, shiftright, solt(set on less than), and, or, nor이다.

MemByte	StateReg 속 instruction이 MEM stage에서 메모리에 값을 읽거나 쓸 때 byte 단위(1)로 값을 다룰지 word 단위(0)로 다룰지를 설정한다.
ALUsrc	StateReg 속 instruction이 EX stage에서 ALU에 들어가는 data2를 regData2에서 들고 올 지(0), imm에서 들고오는지(1)를 결정한다.
ALUsrc2`	StateReg 속 instruction이 EX stage에서 ALU에 들어가는 data1를 regData1에서 들고 올 지(0), regData2에서 들고오는지(1)를 결정한다.
RegWriteSource	StateReg 속 instruction이 WB stage에서 레지스터에 값을 저장할 때, aluResult을 저장할 지(0), memData(1)을 저장할지를 설정한다.
rsForwarding	StateReg가 현재 stage에서 사용하는 값 중 rs에 해당하는 값에 forwarding이 필요한지를 설정한다. 1일 때 forwarding이 필요하고 0이면 필요없다.
rtForwarding	StateReg가 현재 stage에서 사용하는 값 중 rt에 해당하는 값에 forwarding이 필요한지를 설정한다. 1일 때 forwarding이 필요하고 0이면 필요없다.
isSignExtend	imm의 값에 sign-extend를 적용할 지(1), zero-extend(0)를 적용할 지를 결정한다.
Branch	StateReg 속 instruction이 Branch 혹은 Jump를 하는 instruction인지를 나타낸다. 1이면 Jump, 2면 Branch, 0이면 둘 중 어느 것도 아니다.

iv. 생정자

StateReg()	noop을 생성하기 위한 생성자. 모든 StateReg의 값이 0이다.
StateReg(unsigned int pcaddr)	noop을 제외한 StateReg를 생성하기 위한 생성자. 생성할 때 PCaddr에 생성시 받아온 주소를 저장하고 Tag를 1로 설정한다.

C. 02functions.cpp

Pipeline의 각 Stage별로 사용되는 Unit들이 함수로 정의되어 있다. 이 Unit들은 각 Pipeline Stage에서 상태 레지스터의 값들을 바꾸어서 Pipeline이 정상적으로 동작하도록 만든다.

MUX	Signal과 data를 최대 4개까지 받아서 signal에 따른 data를 반환한다.
CONTROL	ID stage에서 StateReg 속 instruction의 op와 funct를 본 후 control

	signal들을 생성한다.
EX_ALU	EX stage에서 연산을 수행하는 Unit이다. data1, data2, ALUop를 인자로 받아 ALUop에 해당하는 연산을 수행 후 값을 반환한다.
DataForwardUnit	StateReg 속 instruction이 각 stage에서 forwarding이 필요한 지를 판단 후 적절한 control signal을 생성한다.
HazardUnit	Pipeline stage 속 StateReg들이 가지고 있는 hazard를 판단한 후 Pipeline의 stall 여부를 결정한다.
BranchHandler	ID stage에서 jump instruction, always taken prediction 시 branch instruction의 jump 주소를 계산하고, MEM stage에서는 ID stage에의 jump 주소 예측의 성공 여부를 판단하고 예측에 실패했을 때는 flush를 설정한 후 원래 실행했어야 할 instruction의 주소를 반환한다.

D. 03datamanage.cpp

00main.cpp에서 Project 2의 코드를 사용한 부분이 있는데, 이 부분을 지원하기 위해서 Project 2에서 가져온 코드이다.

함수명	기능
DecimalToHex	unsigned int를 16진수 형태의 문자열로 바꾸는 함수다. Project 1과는 달리, isSigned 인자가 추가되어 기본값인 false일때는 unsigned int로 처리하고, true일때는 int형으로 명시적 casting 후 처리한다.
SplitLine	Project 1에서와 동일하게 string str가 들어오면 char seperator를 기준으로 분리한 후 이를 담은 vector를 반환한다.
StrHaveChar	string str에 char c가 있는지 여부를 확인하고 있으면 true, 없으면 false를 반환한다.
NumToBit	String 형식의 수를 받으면 이 중 least significant bit를 size의 크기에 맞춰서 string 형식으로 반환한다.
StrIsNum	string str이 숫자인지 판별한다. isHex의 기본값 false일때는 숫자를 10진수를 기준으로 판별하고, isHex가 true일때는 16진수를 기준으로 판별한다.
BitExtended	string 형식의 bit를 받으면 이를 isSignEx가 true일때는 sign extend, false일때는 zero extend한 후 그 결과를 string 형식의 bit로 반환한다.

E. Pipeline의 구현

코드에서 pipeline의 흐름을 다음과 같이 구현했다. 이 흐름은 00main.cpp의 instruction을 수행하는 while문에 구현되어 있다.

i. Pipeline과 상태 레지스터, 각 stage의 구현

우선 모든 instruction들은 fetch가 될 때 저마다 고유한 StateReg를 동적할당으로 부여받는다. 이 StateReg는 해당 instruction의 상태 레지스터가 된다. 각 instruction이 현재 어떤 stage에 있는지는 StateReg 속 Tag가 나타낸다. 또한 instruction의 pipeline 속 위치를 관리하는 vector PipelineStage를 정의한다. PipelineStage.at(0) 속 StateReg는 IF stage, PipelineStage.at(1) 속 StateReg는 ID stage, PipelineStage.at(2) 속 StateReg는 EX stage, PipelineStage.at(3) 속 StateReg는 MEM stage, PipelineStage.at(4) 속 StateReg는 WB stage이다.

ii. 한 cycle이 끝난 후 pipeline의 동작

하나의 cycle이 끝나면, WB stage를 완료한 instruction의 StateReg는 동적할당 해제된다. 그 후 MEM stage에 있는 StateReg부터 stall이나 fluch 여부를 판단하면서 PipelineStage에서 위치를 다음 stage로 옮긴다. 이때 stall이면 StateReg를 다음 위치로 옮기는 것을 중단하고, flush이면 해당하는 instrcution들의 StateReg를 모두 동적할당 해제한다. 그 후 Reg.PC에 해당하는 값을 이용해 새로운 StateReg를 동적할당한 후 PipelineStage의 맨 앞에 저장한다.

아래의 figure들은 위 과정들을 sample2.o를 input file로 받아 실행했을 때 구현된 모습이다.

<table>
<tr><td>

```
===== Cycle 1 =====
Current pipeline PC state:
{0x400000||||}

===== Cycle 2 =====
Current pipeline PC state:
{0x400004|0x400000|||}
```

</td><td>

```
===== Cycle 4 =====
Current pipeline PC state:
{0x40000c|0x400008|0x400004|0x400000|}

===== Cycle 5 =====
Current pipeline PC state:
{0x40000c|0x400008||0x400004|0x400000}
```

</td></tr>
<tr><td>fig 1 : pipeline에서 일반적으로 instruction의 StateReg들이 한 cycle이 끝난 이후 앞으로 이동하는 모습</td><td>fig 2 : 0x400004 instruction 이후 stall이 발생하면 이후 StateReg들은 PipelineStage의 앞으로 이동할 수 없다.</td></tr>
<tr><td>

```
===== Cycle 5 =====
Current pipeline PC state:
{0x40000c|0x400008||0x400004|0x400000}

===== Cycle 6 =====
Current pipeline PC state:
{|0x40000c|0x400008||0x400004}
```

</td><td>

```
===== Cycle 11 =====
Current pipeline PC state:
{0x40002c|0x400028||0x400018|0x400014}

===== Cycle 12 =====
Current pipeline PC state:
{0x40001c||||0x400018}
```

</td></tr>
<tr><td>fig 3 : 0x400000 instruction이 WB stage 이후 동적할당 해제되어 PipelineStage에서 사라지는 모습</td><td>fig 4 : 0x400018에서 flush가 발생해 이후 instruction들이 모두 동적할당 해제되어 사라지고 새로이 instruction을 fetch해 PipelineStage에 추가했다.</td></tr>
</table>

iii. 각 stage와 Unit들의 순서

이번 프로젝트에서는 Simultaneous Register Access를 구현하기 위해 각 stage들의 실행 순서를 IF -> WB -> ID -> EX -> MEM으로 설정했다. 이러면 WB에서 레지스터에 값을 저장한 이후 ID stage에서 레지스터에 접근하게 된다.

또한 ID stage와 EX stage 사이에서 HazardUnit이 작동해 stall과 flush 여부를 판단한다.

MEM 부분까지 실행한 이후에는 앞선 stage에서 실행 결과들을 바탕으로 Reg.PC를 update한다. 이 부분까지 실행하면 한 cycle이 끝났으므로 이를 PipelineStage에 ii. 에서처럼 반영한다. 실행 옵션으로 -p, -d 등이 들어온다면 이에 걸맞는 내용들 stateCout을 불러와 출력한다.

모든 cycle이 끝나거나, -n 옵션에서 지정한 instruction들을 모두 수행하면 stateCout을 불러와 최종 결과를 출력한다.

iv. IF stage

IF stage에서는 IF stage에 있는 StateReg의 PCaddr을 본 후 TextMem에서 op, rs, rt, rd, imm, funct 값을 읽어서 StateReg에 저장한다.

v. ID stage

ID stage에서는 ID stage에 있는 StateReg의 control signal들을 만들어내고, 레지스터에서 값들을 읽어서 regData1과 regData2에 값을 저장한다. 또한 imm의 값에 zero-extend 또는 sign-extend가 필요하다면 이를 수행한 후 다시 imm에 저장한다. 그 후 StateReg 속 instruction이 jump 또는 branch instruction일 경우에는 이를 BranchHandler에 넘겨 처리한 후 Jump 할 주소를 받아온다.

vi. EX stage

DataForwardUnit을 불러와 EX stage에 있는 StateReg의 data forwarding 여부를 결정한다. 그 후 StateReg의 control signal에 맞춰서 ALUdata1과 ALUdata2를 설정한 후 EX_ALU를 불러와 연산을 수행한 후 결과 값을 StateReg의 aluResult에 저장한다.

vii. MEM stage

DataForwardUnit을 불러와 MEM stage에 있는 StateReg의 data forwarding 여부를 결정한다. 그 후 StateReg의 control signal에 맞춰서 DataMem에 값을 저장하거나 DataMem에서 값을 읽어온다.

MEM stage에 있는 StateReg가 branch instruction이라면 BranchHandler에 넘겨서 taken 여부를 판단한다.

viii. WB stage

WB stage에 있는 StateReg가 레지스터에 값을 쓰는 instruction이라면 StateReg의 control signal에 맞춰서 레지스터에 값을 저장한다.