컴퓨터 구조 Homework #3

201911050 도담록, 2023-05-13

1. 코드 설명

i. Binary.h/Binary.cpp, Memory.h/Memory.cpp

Homework #2의 구조가 그대로 사용되었다.\

ii. State.h/State.cpp

State.h와 State.cpp는 State Register과, 그들을 연결하는 Stage Class들을 가지고 있다.

Stage Class는 IF, ID, EX, MEM, WB 5개, Stage Register Class는 IF_ID, ID_EX, EX_MEM, MEM_WB가 있다. Stage Class들은 Stage Register Class들을 연결해 주는 역할을 하며, input을 받아 계산하여 output을 다음 Stage Register에 전달해 준다. 예외로 IF stage의 경우에는 input을 Memory에서 가져오고, WB stage의 경우에는 output Stage Register 없이 바로 Write Back만 진행한다.

여기서는 Hazard나 PC값 계산을 생각하지 않고, Signal 등 상태 정보와 Stage에서 계산된 값들을 앞으로 전달하기만 한다.

iii. Main 함수의 구조

Main(hw3.cpp)의 구조 입력 인자를 받아 파일을 읽어 메모리에 저장하고, Stage와 Stage Register들을 만들어 실행한다.

끝을 확인하기 위해, PC가 영역 밖으로 벗어났는지를 PC가 0x00400000+textsize와 0x00400000 사이에 있는지 확인한다. 다만, PC가 영역 밖으로 벗어나더라도 flush될 수 있으므로 바로 종료 신호를 보내지는 않는다.

입력 인자에 따라 출력과 -n에 따른 멈춤을 구현했다.

Flush는 Stage Register class의 모든 Signal들을 없애고, instruction에 0을 넣어 Noop로 만들도록 구현하였다.

Stall은 bool array stall[5]를 이용하여, 이전 Stage들을 실행하지 않도록 하였고, cycle마다 stall을 모두 false로 만들어 주었다.

구현은 WB stage부터 뒤에서부터 진행되기 때문에, 자동으로 Read after Write 문제는 해결된다 (Write Back이 먼저 실행되므로).

Jump의 경우에는 한 사이클 stall을 진행하고, PC를 update 해준다.

PC는 자동으로 +4로 update되지만(IFstage에서 자동으로 넘겨 준다), 가장 먼저 update되기 때문에 뒤에 있는 Jump 등에 의해 덮어씌워질 수 있다. (mux 역할)

Forwarding의 경우는 MEM_WB to EX, EX_MEM to EX, MEM_WB to MEM을 순서대로 구현하였다. EX_MEM to EX가 MEM_WB to EX 뒤에 있기 때문에, 두 hazard가 동시에 일어나면 MEM_WB의 forwarding이 EX_MEM에 의해 덮어씌워지기 때문에 문제가 해결된다.

Branch의 경우는 Always taken인 경우에는 taken이기 때문에 Flush를 한 번 발생시키고, not taken의 경우는 그대로 진행한다.

Memstage에서 prediction이 틀린 것을 확인한 경우, not taken의 경우 Branch addr로 이동하고 3개의 flush를 발생시킨다. Taken의 경우는 저장해 놓은 PC 값으로 이동하고, 3개의 flush를 발생시킨다.

2. 과제의 컴파일 방법 및 컴파일 환경

Ubuntu 20.04 환경에서 g++ 9.4.0 버전을 이용하여 컴파일 하였다. Binary.h, binary.cpp, memory.h, memory.cpp, State.h, State.cpp, hw3.cpp, makefile을 같은 폴더에 넣고 커맨드에 make를 입력하면 output file인 runfile 이 나온다. (실행 가능 파일)

```
CC = g++
OBJS = binary.o memory.o state.o hw3.o
TARGET = runfile

runfile: $(OBJS)
	$(CC) $(OBJS) -o runfile

state.o : state.h state.cpp memory.o binary.o
	$(CC) memory.h -c state.cpp

memory.o : memory.h memory.cpp
	$(CC) -c memory.cpp
binary.o : binary.h binary.cpp
	$(CC) -c binary.cpp
hw3.o: hw3.cpp binary.h
	$(CC) -c hw3.cpp

.PHONY: clean
clean:
	rm -f $(OBJS)
```

Makefile

3. 과제의 실행 방법 및 실행 환경

컴파일해서 나온 파일인 runfile을 실행한다. 실행 또한 Ubuntu 20.04 환경에서 진행하였다.

과제 메뉴얼에 적힌 대로,

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

-p를 입력하면 현재 실행 중인 instruction들의 PC를 표시한다.

-atp를 입력하면 always taken, -antp를 입력하면 always not taken으로 실행된다. 기본은 antp이다.

-m addr1:addr2를 입력하면 instruction들이 끝난 후 그 메모리를 콘솔에 출력한다.

-d를 입력하면 매 instruction마다 register 값을 형식에 맞춰 출력한다. 이때, -m이 있다면 메모리도 출력한다. -d가 없다면 모든 instruction이 끝나고 마지막 상태만 출력한다.

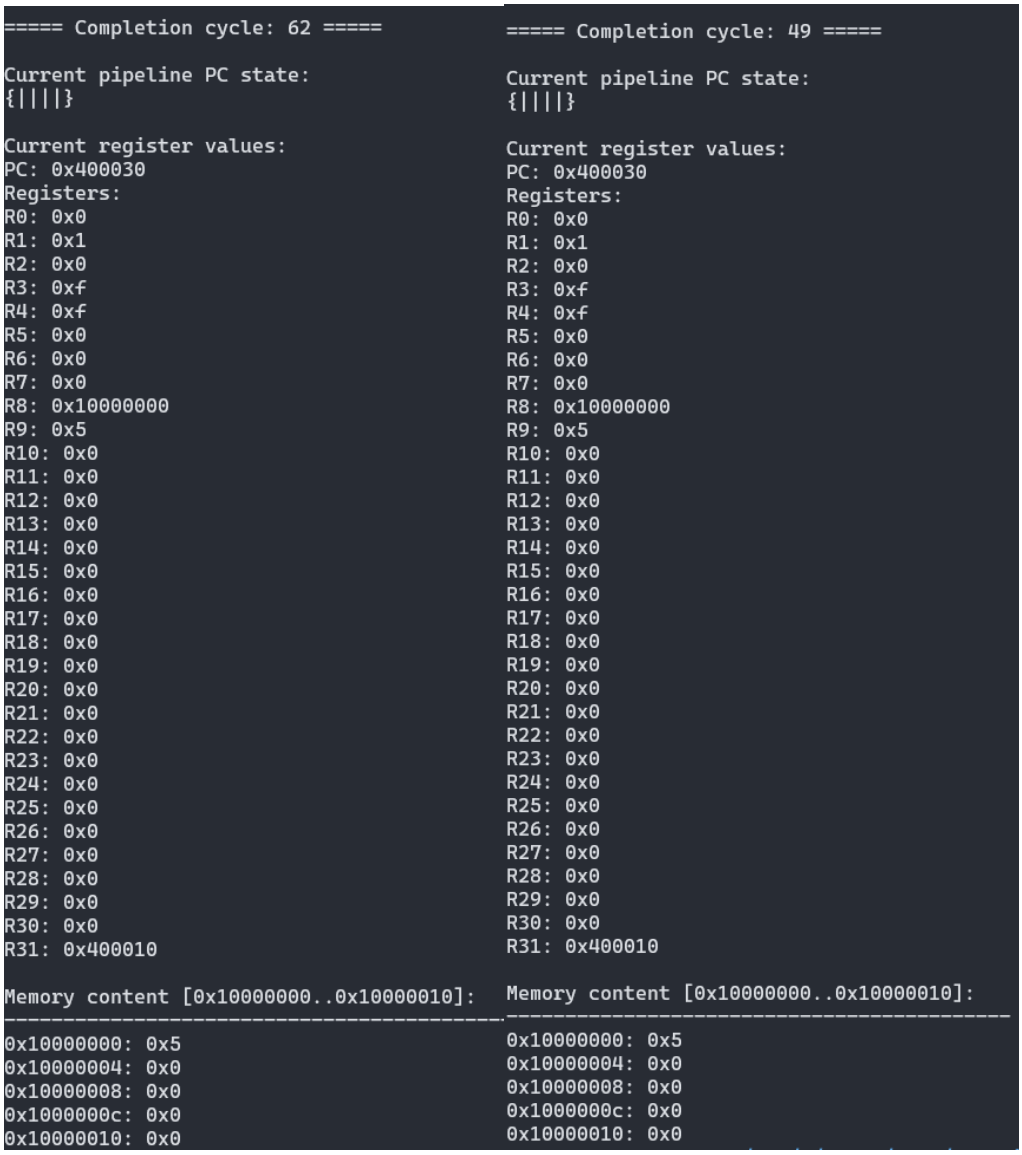
-n num_instruction이 있으면 **num_instruction**만큼만 실행한다.

Input file의 이름을 받아 그 파일을 실행한다.

Sample.o의 경우에는 atp와 antp 관계없이 24 cycle이 진행되었다. Branch prediction을 할 BEQ, BNE가 없기 때문이다.

Sample2.o의 경우에는 atp가 antp보다 13 cycle 더 진행되었다. 왜냐하면, sample2의 경우에는 branch가 top of the loop에 있었기 때문에, 일반적으로 branch가 발생하지 않아 antp에서는 prediction이 계속 맞았지만, atp에서는 계속 틀려 3 cycle flush가 계속 발생하였기 때문이다.

총 5번의 branch에서 15 cycle 차이가 났고, 마지막 cycle에서는 branch가 성공하기 때문에 atp가 antp보다 2 cycle 빨라(3-1, atp는 반드시 1 cycle flush가 있으므로) 15-2 = 13 cycle 차이가 난 것으로 보인다. 다만, Cycle에만 변동이 있고 register값이나 마지막 PC값, memory는 atp와 antp가 똑 같은 결과가 나왔다.



Always Taken in sample2.o /Always not Taken in sample2.o