Project 3: Pipelined MIPS emulator Report

201811118 이 구

1. Code Flow

1.1 State Register Info

Pipeline state register는 struct 형태로 구현하여 사용하였다. Project 3 안내 pdf에 예시로 제공 정보들은 그 용도가 함께 설명되어 있으므로 이에 대한 설명은 생략한다. 각 stage 별로 추가적으로 사용한 정보는 아래와 같다.

1) IF/ID stage register
   current_pc : 출력해야 할 정보 중 하나인 current pipelined PC state를 저장

2) ID/EX stage register
   current_pc: 출력해야 할 정보 중 하나인 current pipelined PC state를 저장
   opcode: 실제로는 ALU control bit를 사용하지만, 구현상의 편의를 위해 opcode를 그대로 다음 EX stage로 전달
   is_jump: 현재 명령어가 j, jal, jr인 경우, 한 cycle의 stall이 필요하고, 이를 나타내기 위해 현재 명령어가 j, jal, jr인 경우 1로 설정
   jump_target: 현재 명령어가 j, jal, jr인 경우, jump 할 memory address를 저장
   is_load_use: load_use hazard를 detection 한 경우 1로 설정

3) EM/MEM stage register
   current_pc: 출력해야 할 정보 중 하나인 current pipelined PC state를 저장
   write_to_mem: MEM stage에서 memory에 write를 해야 할 경우(sw, sb) 1로 설정
   write_to_mem_val: MEM stage에서 memory에 write 할 값을 저장
   read_from_mem: MEM stage에서 memory에서 read 해야 할 경우(lw, lb) 1로 설정
   write_to_reg: WB stage에서 register에 write를 해야 할 경우 1로 설정
   reg_num: WB stage에서 write 할 register의 번호를 저장
   is_branch: branch 명령어(beq, bne)인 경우, 1로 설정
   predict_correct: 사용한 static branch prediction의 종류(always taken or always not taken)에 따라 예측이 맞았다면 1로 설정

4) MEM/WB stage register
   current_pc: 출력해야 할 정보 중 하나인 current pipelined PC state를 저장
   write_to_reg: WB stage에서 register에 값을 write하는 경우 1로 설정
   reg_num: WB stage에서 write 할 register의 번호를 저장
   select_alu: WB stage에서 값을 register에 write 할 때, alu_out의 값을 이용할지, mem_out 값을 이용할 지를 결정하기 위한 정보. alu_out을 이용하는 경우 1로 설정
   is_branch: branch 명령어(beq, bne)인 경우, 1로 설정
   br_target: predict가 틀린 경우, pc값을 수정할 address 저장
   read_from_mem: MEM stage에서 memory를 읽은 경우 1로 설정 (lw, lb)
   predict_correct: 사용한 static branch prediction의 종류(always taken or always not taken)에 따라 예측이 맞았다면 1로 설정

위의 4개 state register를 모두 저장하는 구조체인 State Register를 구현하여 사용하였고, update 전의 정보와 update 후의 정보를 저장하는 prev_state, current_state 총 두 개의 state register를 이용하여 전체적인 연산 과정을 구현하였다.

1.2 Implementation Details

1.2.1 IF Stage

IF Stage에서는 현재 pc에 해당하는 instruction을 fetch해 온다. 구현상의 편의를 위해, pc 값의 update는 1.2.6에서 수행하였다.

1.2.2. ID Stage

ID Stage에서는 instruction의 decoding, load-use hazard detection, jump(j, jr, jal) instruction detection을 수행하며, -atp flag를 입력받은 경우 branch(beq, bne) instruction에 대해서도 jump instruction과 동일한 연산을 수행한다. 이때, 실제 stall은 1.2.6에서 수행한다. load-use hazard의 경우, load 다음의 instruction이 동일한 register를 사용하는 경우 stall을 추가한다. 이때, MEM stage에서 MEM/WB to MEM forwarding을 구현하였으므로, (lw, lb) 후 (sw, sb) instruction이 같은 register를 사용하는 경우 따로 stall을 추가하지 않는다.

1.2.3. EX Stage

EX Stage에서는 Forwarding('EX/MEM to EX', 'MEM/WB to EX') 및 ALU에서 수행되는 연산들을 진행한다. 이때, ALU의 결과를 EX/MEM state register의 alu_out에 저장한다. branch instruction의 경우, 예측이 틀렸을 경우에만 br_target을 계산하도록 하였고, -atp를 입력받은 경우 br_target으로 next_pc를, -antp를 입력받은 경우 br_target으로 next_pc + 4 * imm 를 사용하였다.

1.2.4. MEM Stage

MEM Stage에서는 forwarding('MEM/WB to MEM') 및 memory를 사용하는 instruction(sw, sb, lw, lb)들에 대한 연산을 수행한다. 직접적으로 메모리에 값을 쓰고 읽어오는 과정을 진행하고, 값을 읽어오는 경우 그 값을 MEM/WB state register의 mem_out에 저장한다.

1.2.5. WB Stage

WB Stage에서는 register에 값을 쓰는 과정을 수행한다. 이때, MEM/WB stage에 저장되어 있는 정보들을 이용하여 몇 번 register에 write할 지를 결정하고, select_alu 값을 통해 alu_out과 mem_out 중 어떤 값을 write 할 지를 결정한다.

1.2.6. etc

위의 모든 과정을 마친 후, 그 결과에 따라 pc 값을 update하고, noop instruction을 두 instruction 사이에 추가하거나, flush 한다.

1) noop instruction을 추가하는 경우

이는 IF stage에 jump 또는 -atp 를 입력받았을 때 branch instruction (j, jr, jal, Always taken predictor를 사용하는 경우 beq, bne)이 존재하는 경우이거나, load-use hazard를 발견한 경우이다. 첫 번째 경우, pc값을 jump target으로 update한 후, IF/ID state register를 0으로 초기화 한다. 두 번째 경우, pc 값을 update 하지 않고, ID/EX state register를 0으로 초기화한다.

2) flush 하는 경우

이는 branch prediction이 틀렸을 때 발생한다. branch prediction이 잘못되었다는 것은 EX stage가 수행된 후 알 수 있으므로, branch instruction이 MEM Stage에 있을 때 그 앞 세 개의 Stage 정보를 날려야 한다. 이때, EX stage에서 계산한 br_target 값을 알고있으므로, pc를 br_target으로 업데이트하고, 그 후 IF/ID, ID/EX, EX/MEM state register의 값을 모두 0으로 채워주면 된다.

2. Compile & Execution

제출한 압축 파일은 global.h, MIPS_pipelined_emulator.cpp, Makefile, Report로 구성되어있다. 압축을 해제한 폴더에서 아래의 명령어를 통해 컴파일 및 실행할 수 있다.

```
$ (sudo) make
$ (sudo) ./runfile <-atp / -antp> [-m addr1:addr2] [-d] [-n num_instruction] <object file>
```

3. Environment

코드는 Ubuntu 22.04.3 LTS 환경에서 c++로 작성되었으며, g++ 9.5.0을 사용하여 컴파일하였다. 테스트는 Windows Subsystem for Linux (Ubuntu 20.04) 환경에서 g++ 9.4.0을 사용하여 진행하였다.