Computer Architecture Report

CSE30101

20151509 Hyunjoon Jeong

0. Introduction

In this project, our goal was to devise and implement the MIPS simulator, which can support two types of mode. In mode 0, the simulator only provides register file forwarding (internal forwarding). On the other hand, in mode 1, it also provides data forwarding with hazard detection.

1. Functional Verification

To verify the functionality of the simulator, I used seven provided examples, and each of results are in output directory. To compare this simulator outputs with actual results, I referred to following web page. (https://cpulator.01xz.net/?sys=mipsr5b) In mode 0, each of examples does not work properly, because of data hazard. On the other hand, in mode 1, the simulator returns proper outputs.

First of all, the simulator initializes some data – mode, limitation of cycle, register initialization file, and code file. Then, the simulation executed by ordering WB, MEM, EX, ID, and IF. Actually, it would seem like reversed order with general case: IF, ID, EX, MEM, and WB, but it has advantage of internal forwarding. (see section 2)

In case of memory operation, the memory is implemented by using key-value hash map with little endian. So, the memory address is used as key of hash map and its data is used as value. So, one key-value pair act as one byte, two key-value pair act as one half-word, and four pairs are equal to one word.

2. Implementation of register file forwarding

In my simulator code, after the WB stage appear, then ID stage executed. That is, the writing to register runs faster in WB stage than reading from register in ID stage of same cycle. So, it can avoid structural hazard between WB stage and ID stage. As a result, the part of structural hazard can be solved.

3. Implementation of data hazard detection and data forwarding

Before this implementation, I need to explain the data structure of each register and algorithm. In “register.cpp”, there are 4 lists data structure and they refer IF/ID, ID/EX, EX/MEM and

MEM/WB. At the end of each stage, each of executed results are pushed back to this list structure with information, which is stored in the "Reg" class.

According to this process, the front of each list has information that should be executed in the next cycle. So, if the simulator checks the register, which is located at the front of each list, we can easily detect data hazard detection. Then, the "hazard" boolean variable will turned into true value in EX/MEM or MEM/WB list. (This is because these stages are executed faster than ID stages in this simulator.)

To handle detected data hazard, the simulator introduced to direct register line from ALU process of EX stage. (in this code, "internal_reg array") When the mode 1 was turned on, each calculated result from EX stage and MEM stage is transferred to internal_reg array. Then, the ID stage, which is detected data hazard, uses internal_reg array instead of cpu_reg array, which is originally used as registers cluster. When the instruction solved data hazard, the "hazard" variable turned into false value.

4. About mode 0 with example

From execution of example code file in mode 0, the simulator found only structural hazard and it solved by internal forwarding. To verify behavior of mode 1, I used DH1.s example. First of all, I used this web page (http://shell-storm.org/online/Online-Assembler-and-Disassembler/) to get binary code. Then, I had to set initialized register value for this example. The initialized registers are following this. $12 = A, $13 = 3, $15 = 2, others are equal to zero. Then, the expected result is, $14 = B, $16 = D and others are equal to initialized value. However, in mode 0, $14 has FFFFFFFE value. On the other hand, mode 1 shows expected value. The reason is that the mode 0 does not support data forwarding. As a result, $14 register did not wait WB stage of ADD instruction and execute SUB. In mode 1 case, the EX stage of ADD transfer the result to ID stage of SUB by using data forwarding. So, SUB operation can get correct value from $15 register.

*Convert binary issue: Originally, DH1.s file uses $t0, $t1 and $t3. However, the web page returns other register locations ($12, $13). (Assemble/Disassemble web page에서 DH1.s를 변환하는 경우, 레지스터가 DH1.s에 나온 것과 다른 레지스터를 binary로 변환해주는 것 같습니다. 따라서 .s 파일에 나온 레지스터와 다른 레지스터를 레지스터 초기화로 사용했습니다.)

5. Key features

In this simulator, there are some personal features. In case of stalling (or bubble), the instruction shown in the console is the cause of stalling. For example, when the BEQ instruction decodes in ID stage, it requests stall instruction. (In the simulator code, the bubble integer count was increased one.) Then, in next cycle, there will be no operation code will be fetched in IF stage, but console would show BEQ instruction, which causes stalling. Of course, the NOP instruction, which is 000000, can be handled by same method with stalling, but in this case, the console would show NOP hexadecimal instruction. This stalling occurs from branch, jump and load memory operations.

6. Conclusion

From this MIPS simulator implementation, it is important to prevent three types of hazard: structural, control and data hazard. In case of structural hazard, it is caused by the concurrency of reading and writing memory space or register in same cycle. Then, it can be solved by internal forwarding (register file forwarding), which do writing first and reading next. In case of control hazard, it is caused by branch operation. There is branch prediction technique, but this simulator did not use in both modes. Instead of prediction, the simulator does NOP operation after fetching the branch instruction. Finally, in case of data hazard, it is solved by data forwarding. In this simulator, by using temporarily storage after ALU, ID stage can access correct register value directly.