October 3rd Lecture Summary

2023-12675 박지호

In this lecture, we further discussed the pipelining hazards mentioned in the previous lecture, and how to counteract that.

But before that, we explored how pipelining can be implemented in the CPU circuit. In order to realize pipelining, pipeline registers must be placed between each steps. These store result produced by the previous step, so it can be used to further evaluate the instruction in the next step. For example, some information, like the load operation's write register number, needs to be passed from the later steps, so couple additional data paths needs to be added. Likewise, the control signal, generated once during instruction decode step, needs to be stored then propagated to later steps. This wasn't necessary in a non-pipelining CPU circuit since at any given moment the entire circuit would be evaluating the same instruction and thus able to acquire the information from instruction code directly. However, since this isn't the case in a pipelining CPU circuit, the information needs to be stored somewhere and be passed around. This last part was never explicitly stated in the lecture, but how I interpreted it.

We then looked at how load and store instruction would be evaluated with this pipelining CPU circuit. We were also shown a diagram to help understand what really happens in the CPU circuit when the pipelining happens, as well as an actual circuit diagram with additional data paths.

Then we moved on to pipelining hazards. It was stated in the previous lecture that data hazard refers to a dependency that a later instruction may have to a data created by a prior instruction, and that forwarding, sending the necessary information directly to the step which requires it, can help resolve this problem.

However, the problem still remains of detecting when a forwarding is required. Forwarding is required is when the EX step or MEM step writes to the register that ID step reads from. For this to happen, the destination register number of writing step and the source register number of ID step should match, the writing step should actually write to the register, and the register in question shouldn't be `x0`. Furthermore, in case a double data hazard happens, which is when forwarding happens from both EX and MEM step, the value from EX should be prioritized. The forwarding unit checks for these conditions and sends a signal to a multiplexer in the EX step to make it output an appropriate value.

However, as mentioned in the last lecture, forwarding cannot solve every data hazard scenario. For instance, when the result of load instruction is required for computation in the next step, bubble is unavoidable. In these scenarios, the question becomes how we detect such situation and insert bubble to the pipeline. To achieve the latter, We force control values in ID/EX register to 0, making EX, MEM, and WB step to not perform any further instructions. Then, we can prevent update of PC and IF/ID register. Obviously, this is detrimental to performance, and compilation optimization must be performed to prevent such situation from ocurring in the first place.

The other form of hazard we discussed was control hazard. When performing branch instruction, the CPU cannot determine which instruction to performed next until after the MEM step. It was mentioned that branch prediction is performed to counteract this. One way to do this is to simply expect the branch to not happen and continue evaluating the instruction in order. If the branch does indeed happen, we may flush the pre-evaluated instructions and evaluate the branched instruction instead. Additionaly, we would need to undo the memory or register update if there were any, however, it was said that this would be covered later in the semester. Also, we may move the target address adder and register comparator to the ID step to reduce the out step. When doing this, we must account for few scenarios. First, if the coparison register is a destination registre of preceding 2nd or 3rd ALU instruction, we need apply forwarding. And if the comparison register is the destination register of preceding ALU instruction or preceding load instruction, we need to apply bubbling.