Lecture 3 notes

- Review of the classic 5-stage pipeline of MIPS. We outlined two pipeline representations (single instance, multiple instances) and the five stages (IF, ID, EX, MEM, WB).
- Concept of hazards: Conflicts between instructions that may cause a pipeline to stall for one or more cycles
    - Structural hazards: Hazards introduced due to concurrent accesses from multiple instructions to the same hardware resource (e.g. single-ported memory)
    - Data hazards: Hazards introduced due to data dependences between instructions, more specifically, when an instruction I produces a result in a register, which is consumed by a following instruction J.
        - We described three types of data hazards, RAW, WAW, and WAR. RAW is a true data hazard in the sense that it occurs due to true communication of values between instructions. We discussed forwarding to avoid RAW hazards, using feedback from the execution stage, the memory stage and the writeback stage to the execution stage. We also discussed that while WAW and WAR hazards do not happen in the simple MIPS pipeline, they are possible if the processor allows some form of out-of-order execution. The presence of hazards prevents certain instruction reorderings, which could otherwise be used to improve performance.
        - Among the hazards we discussed, hazards between integer arithmetic/logic RR or RI instructions issued in 2, 3, or 4 consecutive cycles can be resolved with forwarding, or by writing and reading the register file in the same clock cycle, taking advantage of edge-triggered logic. Load-store hazards can also be resolved with forwarding from the memory read/write stage back to the same stage. Load-use hazards though can not be resolved with forwarding.
        - We gave an example of how software can be used to resolve load-use hazards. If we can insert at least one instruction between the load and the use, we can fill the otherwise idle slot with useful computation. This optimization is feasible provided we can find enough non-dependent instructions to move around in the execution sequence. Never forget that moving instructions should be done so that the execution does not violate the original program order, which was also the intention of the programmer!
    - Control hazards occur because of conditional branches. The processor can not know whether the branch is taken or not taken until the 3rd stage of execution of the branch. Notice that the

conditional branch requires two arithmetic operations, one to compute the target address, and one to perform the comparison and decide whether the branch is taken or not taken. We can do the target address calculation as soon as the ID stage, regardless of whether the current instruction in the ID stage is a branch. Still, if we have to wait to do the comparison in the EX stage, then each branch will necessarily require 3 stall cycles: First stall cycle, branch is in the ID stage, second stall cycle, branch is in the EX stage, third stall cycle we know the outcome of the comparison (taken/not-taken) from the branch's EX stage, but we also need to update the program counter with the new target address. We will discuss static solutions to this problem in the next lecture.

- Exceptions:
    - We discussed exceptions and interrupts, as seen from the perspective of the processor's pipeline. The key idea we elaborated on is to prevent instructions that trigger exceptions from updating the state of the processor (i.e. avoid commits of results to the register file or memory). We discussed how exceptions can be triggered in the ID stage (e.g. illegal instruction), the EX stage (e.g. overflow), and the MEM stage (e.g. bad memory address). We discussed how the hardware can "kill" instructions to prevent them from writing back any results (by zeroing out control bits and results in the latches). Exceptions can happen at almost any stage in the pipeline. The term "precise exceptions" implies the ability of the processor to recover from exceptions by restarting the execution at a specific interrupted instruction X, and the processor state is such that the program continues as if all instructions before X have completed and no instruction after X has issued.