CS61C Spring 2026

Precheck: RISC-V Pipelining, Hazards

Discussion 9

1 Pre-Check: T/F?

1.1 By pipelining the CPU datapath, each single instruction will execute faster because pipelining reduces the latency per instruction (resulting in a speed-up in performance).

1.2 A pipelined CPU datapath results in instructions being executed with higher throughput compared to the single-cycle CPU.

1.3 Having two 'read' ports and a 'write' port to the Register File solves the hazard of two instructions that read and write to the same register simultaneously.

1.4 Without forwarding or write-then-read, data hazards will usually result in 3 stalls.

1.5 All data hazards can be resolved with forwarding.

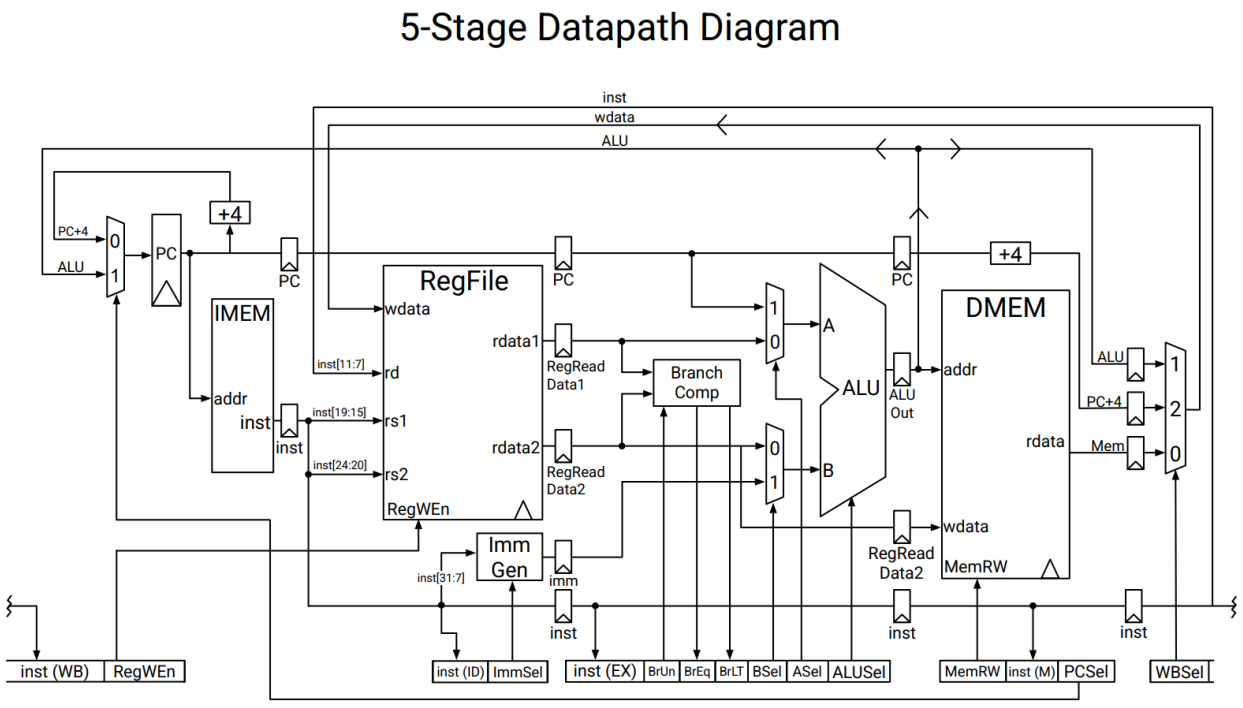
1.6 Stalling is the only way to resolve control hazards.

2 Pipelining

In order to pipeline, we separate the datapath into 5 discrete stages, each completing a different function and accessing different resources on the way to executing an entire instruction.

Recall the five stages: In the **IF** stage, we use the Program Counter to access our instruction as it is stored in IMEM. Then, we separate the distinct parts we need from the instruction bits in the **ID** stage and generate our immediate, the register values from the RegFile, and other control signals. Afterwards, using these values and signals, we complete the necessary ALU operations in the **EX** stage. Next, anything we do in regards with DMEM (not to be confused with RegFile or IMEM) is done in the **MEM** stage, before we hit the **WB** stage, where we write the computed value that we want back into the return register in the RegFile.

These 5 stages, divided by registers, allow operating different stages of the datapath in the same clock period. Different instructions can use different stages at the same time. At each clock cycle, the necessary inputs into a particular stage are sampled at the rising clock edge (and available after the clk-to-q delay). After the stage operates on the inputs, the corresponding outputs are fed into pipeline registers for the next stage. Note, pipeline registers may also be required to pass information that may not be necessary for the next immediate stage, but some future stage instead.



3 Hazards

One of the costs of pipelining is that it introduces pipeline hazards. Hazards, generally, are issues with something in the CPU's instruction pipeline that could cause the next instruction to execute incorrectly.

The 5-stage pipelined CPU introduces three types: structural hazards (insufficient hardware), data hazards (using outdated values in computation), and control hazards (executing the wrong instructions).

Structural Hazards

Structural hazards occur when more than one instruction needs to use the same datapath resource at the same time. In the standard 5-stage RISC-V pipeline, **there aren't structural hazards**, unless changes are made to the pipeline. The structural hazards that could exist are prevented by RV32I's hardware requirements.

There are two main causes of structural hazards:

- **Register File:** The register file is accessed both during ID, when it is read to decode the instruction and fetch the corresponding register values; and during WB, when it is written to in the `rd` register.
  - We resolve this by having separate read and write ports. However, this only works if the read/written registers are different.
- **Main Memory:** Main memory is accessed for both instructions and data. If memory could only support one read/write at a time, then instruction A (in the IF stage) attempting to fetch from memory would conflict with instruction B attempting to read or write to the data portion of memory.
  - Having separate instruction memory (IMEM) and data memory (DMEM) solves this hazard.

Something to remember about structural hazards is that they can always be resolved by adding more hardware.

Data Hazards

Data hazards are caused by data dependencies between instructions. In CS 61C, where we always assume that instructions go through the processor in order, data hazards occur when an instruction **reads** a register before a previous instruction has finished **writing** to the same register.

Data hazards occur between different stages. Some examples are:

- **EX-ID:** This hazard exists because the output from the execute stage is not written back to the RegFile until the writeback stage, yet it can be requested by the subsequent instruction during the decode stage.
- **MEM-ID:** This hazard exists because the output from the memory access stage is not written back to the RegFile until the writeback stage, but it can still be requested from the decode stage —just like in EX-ID.

  To account for reads and writes to the same register, processors usually write to the register during the first half of the clock cycle and read from it during the second half. This is an implementation of the idea of **write-then-read**, where data is transferred along buses at double

the rate by using both the rising and falling clock edges within a single clock cycle. With write-then-read, we can reduce the number of stalls needed for data hazards by one.

Detecting Data Hazards

Say we have the `rs1`, `rs2`, `RegWEn`, and `rd` signals for two instructions (instruction `n` and instruction `n + 1`) and we wish to determine if a data hazard exists across the instructions. We can simply check to see if the `rd` for instruction `n` matches either `rs1` or `rs2` of instruction `n + 1`, indicating that such a hazard exists.

We could then use our hazard detection to determine which forwarding paths/number of stalls (if any) are necessary to take to ensure proper instruction execution. In pseudo-code, part of this could look something like the following:

```
if (rs1(n + 1) == rd(n) && RegWen(n) == 1) {
      set ASel for (n + 1) to forward ALU output from n
  }
if (rs2(n + 1) == rd(n) && RegWen(n) == 1) {
    set BSel for (n + 1) to forward ALU output from n
}
```

Control Hazards

Control hazards are caused by **jump and branch instructions**, because for all jumps and some branches, the next PC is not `PC + 4`, but rather the result of the ALU, which is only available after the EX stage.

One way to handle control hazards is to stall the pipeline until the correct PC is known—but this reduces performance. Another option is to implement a branch predictor to try to "guess" whether a branch should be taken or not and, instead of stalling, execute the next instruction based on our guess. We discover the "correctness" of the guess when the branch instruction reaches the EX stage: an incorrect guess means we have to change the incorrect instructions in the pipeline to a NOP. On a correct guess, we save some cycles!