CS 61C Fall 2022 — RISC-V Pipelining and Hazards

Discussion 8

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

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

False. Because we implement registers between each stage of the datapath, the time it takes for an instruction to finish execution through the 5 stages will be longer than the single-cycle datapath we were first introduced with. A single instruction will take multiple clock cycles to get through all the stages, with the clock cycle based on the stage with the longest timing.

1.2 A pipelined CPU datapath results in instructions being executed with higher throughput (than the single-cycle CPU).

True. Recall that throughput is the number of instructions processed per unit time. Pipelining results in a higher throughput because more instructions are run at once, which utilizes more parts of the datapath simultaneously.

1.3 Through adding additional hardware, we can implement two 'read' ports as well as a 'write' port to the RegFile (where registers can be accessed). This solves the hazard of two instructions reading and writing to the same register simultaneously.

False. The addition of independent ports to the RegFile allows for multiple instructions to access the RegFile at the same time (such as one instruction reading values of two operands, while another instruction is writing to a return register). However, this does not work if both instructions are reading and writing to the same register. Some solutions to this data hazard could be to stall the latter instruction by 1 cycle or to forward the read value from a previous instruction, bypassing the RegFile completely.

1.4 All data hazards can be resolved with forwarding.

False. Hazards following `lw` cannot be fully resolved with forwarding because the output is not known until after the MEM stage, making a stall necessary.

1.5 As stalling reduces performance significantly, we generally prefer other solutions to fixing pipeline hazards, even at the cost of complexity or hardware. In a modern-day pipelined CPU, are there still use-cases for stalling to resolve potential hazards? If so, describe a program that would.

True. Say we have the RISC-V program where a0 is a pointer to an array of integers, and we want to load a1 with the first element * 2:

```
lw t1 0(a0)
add t2 t1 t1
mv a1 t2
```

In this program, there are no other instructions to move into the load delay slot, so we are forced to `nop` the next instruction and repeat it afterwards, essentially stalling for one cycle. While we do have many tools and alternative solutions to lessen possible performance loss, in some cases it is unavoidable.

2 Pipelining Registers

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.

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 as shown in the figure, allow the datapath to provide a pipeline for multiple instructions to operate at the same time, each accessing different resources. A pipelined datapath is provided for you on the last page. Use it to answer the following questions.

2.1 What is the purpose of the new registers?

When we pipeline the datapath, the values from each stage need to be passed on at each clock cycle. Each stage in the pipeline only operates on a small set of values, but those values need to be correct with respect to the instruction that is currently being processed. Say we use load word (lw) as an example: if it is in the EX stage, then the EX stage should look like a snapshot of the single-cycle datapath. The values on the rs1, rs2, immediate, and PC values should be as if lw was the only instruction in the entire path. This also includes the control logic: the instruction is passed in at each stage, the appropriate control signals are generated for the stage of interest, and that stage can execute properly.

2.2 Looking at the way PC is passed through the datapath, there are two places where +4 is added to the PC, once in the **IF** and **MEM** stage. Why do we add +4 to the PC again in the memory stage?

We add +4 to the PC again in the memory stage so we don't need to pass both PC and PC+4 along the whole pipeline. This would use more registers, adding unnecessary hardware. We also can't just pass only PC+4 through the pipeline, as we need the original PC value in operands like `auipc`.

2.3 Why do we need to save the instruction in a register multiple times?

We need to save the instruction in a register multiple times because each pipeline stage needs to receive the right control signals for the instruction currently in that stage.

3 Performance Analysis

Register clk-to-q 30 ps	**Branch comp.** 75 ps	**Memory write** 200 ps
Register setup 20 ps	**ALU** 200 ps	**RegFile read** 100 ps
Register hold 10 ps	**Imm. Gen.** 15 ps	
Mux 25 ps	**Memory read** 250 ps	**RegFile setup** 20 ps

Given above are sample delays for each of the datapath components and register timings. In the questions below, use these in conjunction with the pipelined datapath on the last page to answer them.

3.1 What would be the fastest possible clock time for a single cycle datapath? Recall from last week's discussion that one instruction which exercises the critical path is `lw`.
(HINT: $t_{\text{clk-cycle}} \geq t_{\text{clk-to-q}} + t_{\text{longest-combinational-path}} + t_{\text{setup}}$)

$$\begin{aligned} t_{\text{clk}} &\geq t_{\text{PC clk-to-q}} + t_{\text{IMEM read}} + t_{\text{RF read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{DMEM read}} + t_{\text{mux}} + t_{\text{RF setup}} \\ &\geq 30 + 250 + 100 + 25 + 200 + 250 + 25 + 20 \\ &\geq 900 \text{ ps} \end{aligned}$$

Note that the delay in the immediate generator as well as the branch comparator are omitted because the immediate generator and branch comparison is done in parallel with the RegFile read and ALU computation respectively, the latter two taking much longer time.

3.2 What is the fastest possible clock time for a pipelined datapath?

$$\begin{aligned} \textbf{IF}: &\ t_{\text{PC clk-to-q}} + t_{\text{IMEM read}} + t_{\text{Reg setup}} = 30 + 250 + 20 = 300 \text{ ps} \\ \textbf{ID}: &\ t_{\text{Reg clk-to-q}} + t_{\text{RF read}} + t_{\text{Reg setup}} = 30 + 100 + 20 = 150 \text{ ps} \\ \textbf{EX}: &\ t_{\text{Reg clk-to-q}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{Reg setup}} = 30 + 25 + 200 + 20 = 275 \text{ ps} \\ \textbf{MEM}: &\ t_{\text{Reg clk-to-q}} + t_{\text{DMEM read}} + t_{\text{Reg setup}} = 30 + 250 + 20 = 300 \text{ ps} \\ \textbf{WB}: &\ t_{\text{Reg clk-to-q}} + t_{\text{mux}} + t_{\text{RF setup}} = 30 + 25 + 20 = 75 \text{ ps} \end{aligned}$$

$$t_{\text{clk}} \geq \max(\textbf{IF}, \textbf{ID}, \textbf{EX}, \textbf{MEM}, \textbf{WB}) = 300 \text{ ps}$$

Again, the immediate generator and branch comparator delays are overshadowed by the longer delays of RegFile read and ALU.

3.3 What is the speedup from the single cycle datapath to the pipelined datapath? Why is the speedup less than 5×?

$\frac{900 \text{ ps}}{300 \text{ ps}}$, or a 3 times speedup. The speedup is less than 5 because of (1) the necessity of adding pipeline registers, which have clk-to-q and setup times, and (2) the need to set the clock to the maximum of the five stages, which take different amounts of time.

Note: Due to hazards, which require additional logic to resolve, the actual speedup would likely be even less than 3 times.

4 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 (hardware not sufficient), data hazards (using wrong values in computation), and control hazards (executing the wrong instruction).

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 pipeline, **there aren't structural hazards**, unless there are active changes 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 the corresponding register values; and during WB, when it is written to in the `rd` register. If the RegFile only had one port, then it wouldn't work since we have one instruction being decoded and another writing back.
  - We resolve this by having separate read and write ports. However, this only works if the read/written registers are different.
  - 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 in the second half. This is an implementation of the idea of **double pumping**, which is when data is transferred along data buses at double the rate, by utilising both the rising and falling clock edges in a clock cycle.
- **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 going through IF and attempting to fetch an instruction from memory cannot happen at the same time as instruction B attempting to read (or write) to data portions of memory.

- Having a separate instruction memory (abbreviated IMEM) and data memory (abbreviated 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, we see data hazards when an instruction **reads** a register before a previous instruction has finished **writing** to that register.

There are two types of data hazards:

- **EX-ID:** this hazard exists because the output from the execute stage is not written back to the RegFile until the writeback stage, yet can be requested by the subsequent instruction in 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 can be requested from the decode stage, just as in EX-ID.

Solving Data Hazards

For all questions, assume **no branch prediction or double-pumping**.

Forwarding

Most data hazards can be resolved by forwarding, which is when the result of the EX or MEM stage is sent to the EX stage for a following instruction to use.

4.1 Look for data hazards in the code below, and figure out how forwarding could be used to solve them.

Instruction	C1	C2	C3	C4	C5	C6	C7
1. `addi t0, a0, -1`	IF	ID	EX	MEM	WB		
2. `and s2, t0, a0`		IF	ID	EX	MEM	WB	
3. `sltiu a0, t0, 5`			IF	ID	EX	MEM	WB

There are two data hazards, between instructions 1 and 2, and between instructions 1 and 3. The first could be resolved by forwarding the result of the EX stage in C3 to the beginning of the EX stage in C4, and the second could be resolved by forwarding the result of the MEM stage in C4 to the beginning of the EX stage in C5.

4.2 Imagine you are a hardware designer working on a CPU's forwarding control logic. How many instructions after the addi instruction could be affected by data hazards created by this addi instruction?

Three instructions. For example, with the addi instruction, any instruction that uses t0 that has its ID stage in C3, C4, or C5 will not have the result of addi's writeback in C5. If, however, we are allowed to assume double-pumping (write-then-read to registers), then it would only affect two instructions since the ID stage of instruction

4 would be allowed to line up with the WB stage of intruction 1. (Side note: how is this implemented in hardware? We add 2 wires: one from the beginning of the MEM stage for the output of the ALU and one from the beginning of the WB stage. Both of these wires will connect to the A/B muxes in the EX stage.)

Stalls

4.3 Look for data hazards in the code below. One of them cannot be solved with forwarding—why? What can we do to solve this hazard?

Instruction	C1	C2	C3	C4	C5	C6	C7	C8
1. `addi s0, s0, 1`	IF	ID	EX	MEM	WB			
2. `addi t0, t0, 4`		IF	ID	EX	MEM	WB		
3. `lw t1, 0(t0)`			IF	ID	EX	MEM	WB	
4. `add t2, t1, x0`				IF	ID	EX	MEM	WB

There are two data hazards in the code. The first hazard is between instructions 2 and 3, from t0, and the second is between instructions 3 and 4, from t1. The hazard between instructions 2 and 3 can be resolved with forwarding, but the hazard between instructions 3 and 4 cannot be resolved with forwarding. This is because even with forwarding, instruction 4 needs the result of instruction 3 at the beginning of C6, and it won't be ready until the end of C6.

We can fix this by stalling: insert a nop (no-operation) between instructions 3 and 4.

4.4 Say you are the compiler and can re-order instructions to minimize data hazards while guaranteeing the same output. How can you fix the code above?

Reorder the instructions 2-3-1-4, because instruction 1 has no dependencies.

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 (why does this make sense?).

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 the result of the ALU available after the EX stage. We could stall the pipeline for control hazards, but this decreases performance.

4.5 Besides stalling, what can we do to resolve control hazards?

We can predict which way branches will go, and when this prediction is incorrect, flush the pipeline and continue with the correct instruction. (The most naive prediction method is to simply predict that branches are always not taken).

Extra for Experience

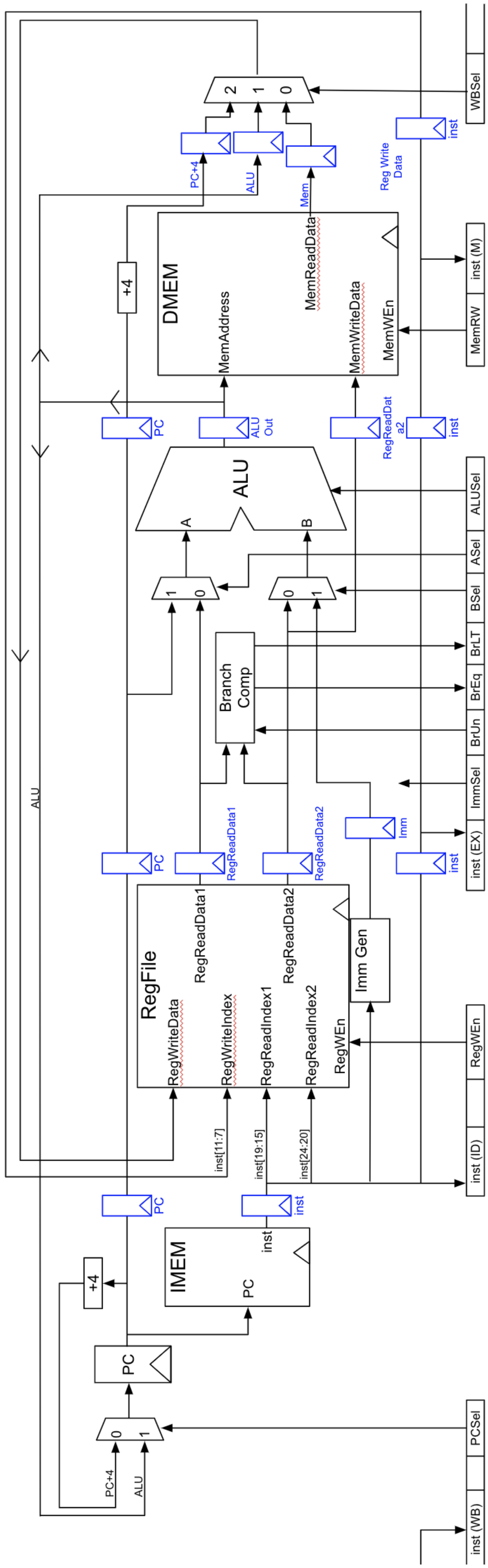
4.6 Given the RISC-V code above and a pipelined CPU with no forwarding, how many hazards would there be? What types are each hazard? Consider all possible hazards between all instructions.

How many stalls would there need to be in order to fix the data hazard(s), if we use double-pumping? What about the control hazard(s), if we use branch prediction?

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9
1. `sub t1, s0, s1`	IF	ID	EX	MEM	WB				
2. `or s0, t0, t1`		IF	ID	EX	MEM	WB			
3. `sw s1, 100(s0)`			IF	ID	EX	MEM	WB		
4. `bgeu s0, s2, loop`				IF	ID	EX	MEM	WB	
5. `add t2, x0, x0`					IF	ID	EX	MEM	WB

There are four hazards: between instructions 1 and 2 (data hazard from t1), instructions 2 and 3 (data hazard from s0), instructions 2 and 4 (from s0), and instructions 4 and 5 (a control hazard).

Assuming that we can read and write to the RegFile on the same cycle, two stalls are needed between instructions 1 and 2 (WB→ID), and two stalls are needed between instructions 2 and 3 (WB→ID). For the control hazard, if we predicted correctly, then no stalls are needed, but if we predicted incorrectly, then we need 3 stalls while flushing the pipeline (MEM→1 cycle before IF). We don't need to stall for the hazard between 2 and 4 because stalling for instruction 3 already handles that.



PC
+4
PC+4
ALU
0
1
PCSel
inst (WB)
IMEM
PC
inst
inst
RegFile
RegWriteData
RegWriteIndex
RegReadIndex1
RegReadIndex2
RegWEn
RegReadData1
RegReadData2
inst[11:7]
inst[19:15]
inst[24:20]
Imm Gen
inst (ID)
RegWEn
RegReadData1
RegReadData2
Imm
inst
inst (EX)
ImmSel
BrUn
BrEq
BrLT
BSel
ASel
ALUSel
Branch Comp
A
B
ALU
ALU Out
RegReadData2
DMEM
MemAddress
MemReadData
MemWriteData
MemWEn
MemRW
inst (M)
Mem
PC+4
Reg Write Data
2
WBSel