CS 61C Spring 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 Pipelining the CPU datapath results in instructions being executed with higher latency and higher throughput.

True. Recall that latency is the time for one instruction to finish, while throughput is the number of instructions processed per unit time. Pipelining results in a higher throughput because more instructions are run at once. At the same time, latency is also higher as each individual instruction may take longer from start to finish because each cycle must last as long as the longest cycle. Additionally, hazards may be introduced.

1.2 Without forwarding, data hazards will usually result in 3 stalls.

True. The next instruction must wait for the previous instruction to finish EX, MEM, and WB, before it can begin its EX.

1.3 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 the MEM stage, making a stall necessary (normally forwarding sends from the output of EX stage).

1.4 Stalling is the only way to resolve control hazards.

False. While one way to resolve control hazards is to stall until the result of the branch instruction is determined, there are other more advanced techniques such as branch prediction, which predicts which path the branch will take and flushes the pipeline if the prediction is wrong.

2 Pipelining Registers

In order to pipeline, we add registers between the five datapath stages. Label each of the five stages (IF, ID, EX, MEM, and WB) on the diagram attached at the end of the worksheet.

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 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.

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**	150 ps
Mux	25 ps	**Memory read**	250 ps	**RegFile setup**	20 ps

3.1 With the delays provided above for each of the datapath components, what would be the fastest possible clock time for a single cycle datapath?

$$\begin{aligned} t_{\text{clk}} &\geq t_{\text{PC clk-to-q}} + t_{\text{IMEM read}} + t_{\text{RF read}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{DMEM read}} + t_{\text{mux}} + t_{\text{RF setup}} \\ &\geq 30 + 250 + 150 + 25 + 200 + 250 + 25 + 20 \\ &\geq 950 \text{ ps} \end{aligned}$$

$$\frac{1}{950 \text{ ps}} = 1.05 \text{ GHz}$$

Note that the delay of branch comparator is omitted because branch comparison is done in parallel with RegFile/ALU, which takes much longer time.

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

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

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

NOTE: For the **EX** stage, the branch comparator time is overshadowed by the ALU computation (The same would be true in the ID stage as well, but since there is no mentioned time for Immediate Generator, we assumed here it is trivial):

$$\begin{aligned}
\text{Branch comparator}: &\; t_{\text{PC clk-to-q}} + t_{\text{Branch comp.}} = 30 + 75 = 105 \text{ ps} \\
\text{ALU computation}: &\; t_{\text{Reg clk-to-q}} + t_{\text{mux}} + t_{\text{ALU}} + t_{\text{Reg setup}} = 25 + 200 = 275 \text{ ps}
\end{aligned}$$

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

$\frac{950 \text{ ps}}{300 \text{ ps}}$, or a 3.2 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: because of hazards, which require additional logic to resolve, the actual speedup would likely be even less than 3.2 times.

4 Hazards

One of the costs of pipelining is that it introduces three types of pipeline hazards: structural hazards, data hazards, and control hazards.

Structural Hazards

Structural hazards occur when more than one instruction needs to use the same datapath resource at the same time. There are two main causes of structural hazards:

Register File The register file is accessed both during ID, when it is read, and during WB, when it is written to. We can solve this by having separate read and write ports. 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 also known as double pumping.

Memory Memory is accessed for both instructions and data. 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 will always assume that instructions are always going 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.

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 computation completed in the EX stage. We could stall the pipeline for control hazards, but this decreases performance.

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 EX stage in C3 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 mux 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 inserting 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 (think, 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, this could look something like the following:

```
if (rs1(n + 1) == rd(n) || rs2(n + 1) == rd(n) && RegWen(n) == 1) {
    forward ALU output of instruction 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 computation completed in 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

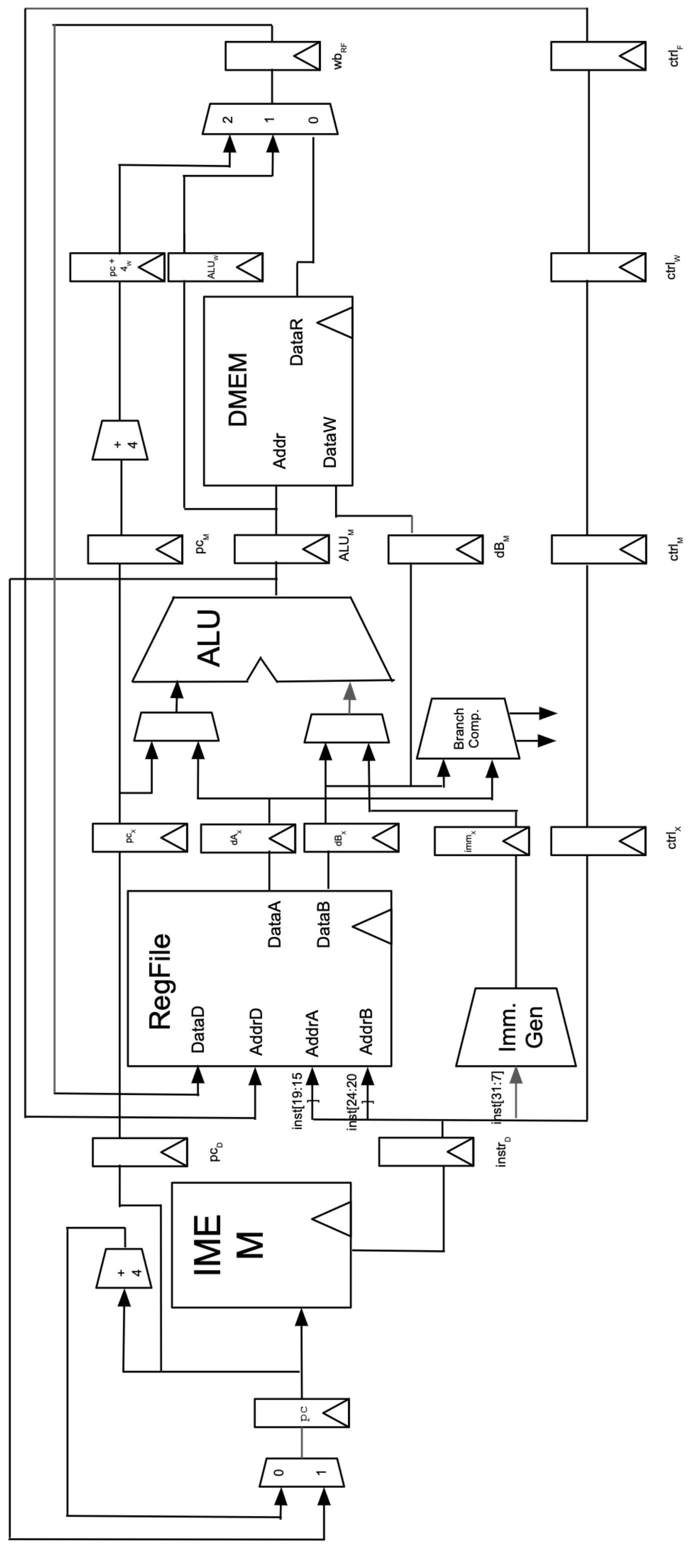
4.6 Given the RISC-V code above and a pipelined CPU with no forwarding, how many hazards would there be? What types are each hazard? Consider all possible hazards from all pairs of instructions, and feel free to use any techniques in class (i.e. branch prediction) to limit the number of stalls.

How many stalls would there need to be in order to fix the data hazard(s)? What about the control hazard(s)?

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, and two stalls are needed between instructions 2 and 3. No stalls are needed for the control hazard, because it can be handled with branch prediction/flushing the pipeline.


pc
0
1
+
4
IMEM
pc_D
instr_D
inst[19:15]
inst[24:20]
inst[31:7]
RegFile
DataD
AddrD
AddrA
AddrB
DataA
DataB
Imm. Gen
pc_X
dA_X
dB_X
imm_X
ctrl_X
ALU
Branch Comp.
pc_M
ALU_M
dB_M
ctrl_M
+
4
DMEM
Addr
DataW
DataR
pc + 4_W
ALU_W
ctrl_W
2
1
0
wb_RF
ctrl_F