

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 1 + 0.2 or 1.2 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\ &= \frac{4.4 \text{ ns}}{1.2 \text{ ns}} = 3.7 \text{ times}\end{aligned}$$

The 0.2 ns overhead essentially establishes a limit on the effectiveness of pipelining. If the overhead is not affected by changes in the clock cycle, Amdahl's law tells us that the overhead limits the speedup.

This simple RISC pipeline would function just fine for integer instructions if every instruction were independent of every other instruction in the pipeline. In reality, instructions in the pipeline can depend on one another; this is the topic of the next section.

C.2

The Major Hurdle of Pipelining—Pipeline Hazards

There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to *stall* the pipeline. Avoiding a hazard often requires that some instructions in the pipeline be allowed to proceed while others are delayed. For the pipelines we discuss in this appendix, when an instruction is stalled, all instructions issued *later* than the stalled instruction—and hence not as far along in the pipeline—are also stalled. Instructions issued *earlier* than the stalled instruction—and hence farther along in the pipeline—must continue, since otherwise the hazard will never clear. As a result, no new instructions are fetched during the stall. We will see several examples of how pipeline stalls operate in this section—don't worry, they aren't as complex as they might sound!

Performance of Pipelines with Stalls

A stall causes the pipeline performance to degrade from the ideal performance. Let's look at a simple equation for finding the actual speedup from pipelining, starting with the formula from the previous section:

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\ &= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}} \\ &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

Pipelining can be thought of as decreasing the CPI or the clock cycle time. Since it is traditional to use the CPI to compare pipelines, let's start with that assumption. The ideal CPI on a pipelined processor is almost always 1. Hence, we can compute the pipelined CPI:

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

If we ignore the cycle time overhead of pipelining and assume that the stages are perfectly balanced, then the cycle time of the two processors can be equal, leading to

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

One important simple case is where all instructions take the same number of cycles, which must also equal the number of pipeline stages (also called the *depth of the pipeline*). In this case, the unpipelined CPI is equal to the depth of the pipeline, leading to

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of the pipeline.

Alternatively, if we think of pipelining as improving the clock cycle time, then we can assume that the CPI of the unpipelined processor, as well as that of the pipelined processor, is 1. This leads to

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} \\ &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

In cases where the pipe stages are perfectly balanced and there is no overhead, the clock cycle on the pipelined processor is smaller than the clock cycle of the unpipelined processor by a factor equal to the pipelined depth:

$$\begin{aligned}\text{Clock cycle pipelined} &= \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}} \\ \text{Pipeline depth} &= \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

This leads to the following:

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} \\ &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}\end{aligned}$$

Thus, if there are no stalls, the speedup is equal to the number of pipeline stages, matching our intuition for the ideal case.

Structural Hazards

When a processor is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a *structural hazard*.

The most common instances of structural hazards arise when some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle. Another common way that structural hazards appear is when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute. For example, a processor may have only one register-file write port, but under certain circumstances, the pipeline might want to perform two writes in a clock cycle. This will generate a structural hazard.

When a sequence of instructions encounters this hazard, the pipeline will stall one of the instructions until the required unit is available. Such stalls will increase the CPI from its usual ideal value of 1.

Some pipelined processors have shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data memory reference, it will conflict with the instruction reference for a later instruction, as shown in [Figure C.4](#). To resolve this hazard, we stall the pipeline for 1 clock cycle when the data memory access occurs. A stall is commonly called a *pipeline bubble* or just *bubble*, since it floats through the pipeline taking space but carrying no useful work. We will see another type of stall when we talk about data hazards.

Designers often indicate stall behavior using a simple diagram with only the pipe stage names, as in [Figure C.5](#). The form of [Figure C.5](#) shows the stall by indicating the cycle when no action occurs and simply shifting instruction 3 to

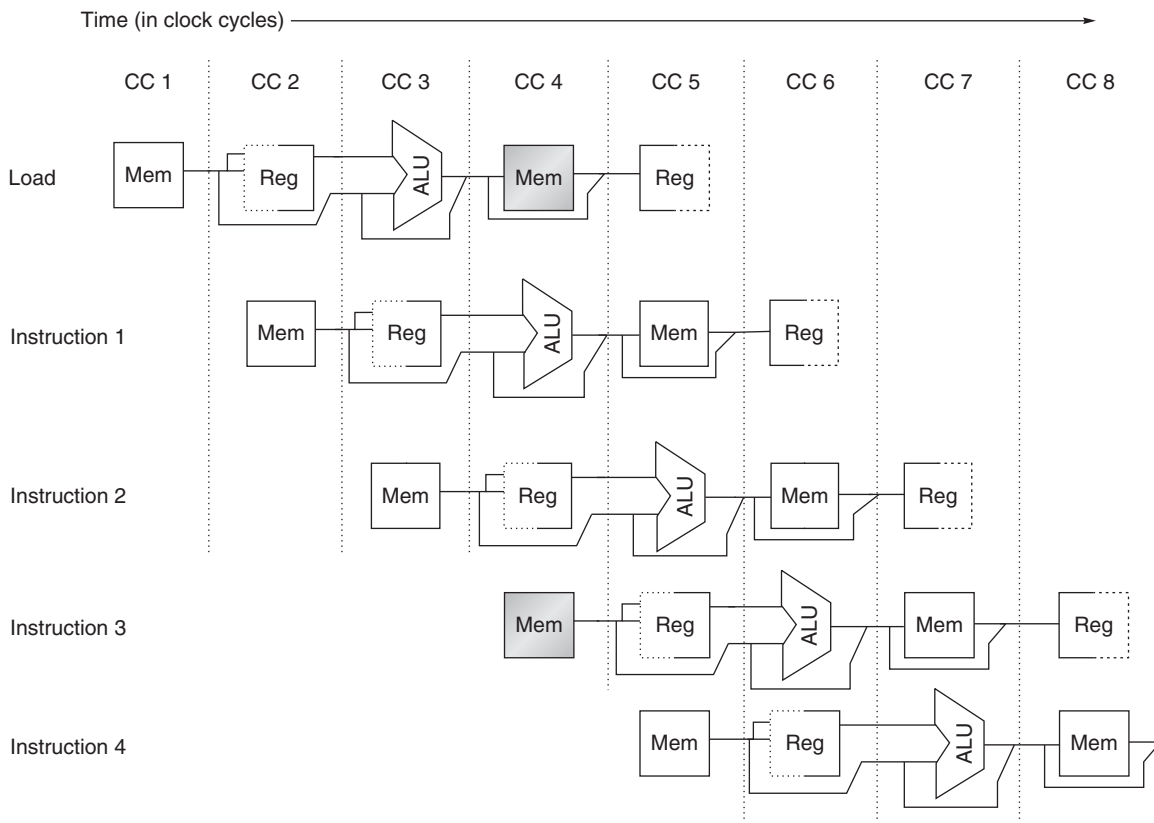


Figure C.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

the right (which delays its execution start and finish by 1 cycle). The effect of the pipeline bubble is actually to occupy the resources for that instruction slot as it travels through the pipeline.

Example Let's see how much the load structural hazard might cost. Suppose that data references constitute 40% of the mix, and that the ideal CPI of the pipelined processor, ignoring the structural hazard, is 1. Assume that the processor with the structural hazard has a clock rate that is 1.05 times higher than the clock rate of the processor without the hazard. Disregarding any other performance losses, is the pipeline with or without the structural hazard faster, and by how much?

Answer There are several ways we could solve this problem. Perhaps the simplest is to compute the average instruction time on the two processors:

$$\text{Average instruction time} = \text{CPI} \times \text{Clock cycle time}$$

Instruction	Clock cycle number									
	1	2	3	4	5	6	7	8	9	10
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				Stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

Figure C.5 A pipeline stalled for a structural hazard—a load with one memory port. As shown here, the load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall—no instruction is initiated on clock cycle 4 (which normally would initiate instruction $i + 3$). Because the instruction being fetched is stalled, all other instructions in the pipeline before the stalled instruction can proceed normally. The stall cycle will continue to pass through the pipeline, so that no instruction completes on clock cycle 8. Sometimes these pipeline diagrams are drawn with the stall occupying an entire horizontal row and instruction 3 being moved to the next row; in either case, the effect is the same, since instruction $i + 3$ does not begin execution until cycle 5. We use the form above, since it takes less space in the figure. Note that this figure assumes that instructions $i + 1$ and $i + 2$ are not memory references.

Since it has no stalls, the average instruction time for the ideal processor is simply the Clock cycle time_{ideal}. The average instruction time for the processor with the structural hazard is

$$\begin{aligned}
 \text{Average instruction time} &= \text{CPI} \times \text{Clock cycle time} \\
 &= (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05} \\
 &= 1.3 \times \text{Clock cycle time}_{\text{ideal}}
 \end{aligned}$$

Clearly, the processor without the structural hazard is faster; we can use the ratio of the average instruction times to conclude that the processor without the hazard is 1.3 times faster.

As an alternative to this structural hazard, the designer could provide a separate memory access for instructions, either by splitting the cache into separate instruction and data caches or by using a set of buffers, usually called *instruction buffers*, to hold instructions. Chapter 5 discusses both the split cache and instruction buffer ideas.

If all other factors are equal, a processor without structural hazards will always have a lower CPI. Why, then, would a designer allow structural hazards? The primary reason is to reduce cost of the unit, since pipelining all the functional units, or duplicating them, may be too costly. For example, processors that support both an instruction and a data cache access every cycle (to prevent the structural hazard of the above example) require twice as much total memory

bandwidth and often have higher bandwidth at the pins. Likewise, fully pipelining a floating-point (FP) multiplier consumes lots of gates. If the structural hazard is rare, it may not be worth the cost to avoid it.

Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This overlap introduces data and control hazards. Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor. Consider the pipelined execution of these instructions:

DADD	R1, R2, R3
DSUB	R4, R1, R5
AND	R6, R1, R7
OR	R8, R1, R9
XOR	R10, R1, R11

All the instructions after the DADD use the result of the DADD instruction. As shown in [Figure C.6](#), the DADD instruction writes the value of R1 in the WB pipe stage, but the DSUB instruction reads the value during its ID stage. This problem is called a *data hazard*. Unless precautions are taken to prevent it, the DSUB instruction will read the wrong value and try to use it. In fact, the value used by the DSUB instruction is not even deterministic: Though we might think it logical to assume that DSUB would always use the value of R1 that was assigned by an instruction prior to DADD, this is not always the case. If an interrupt should occur between the DADD and DSUB instructions, the WB stage of the DADD will complete, and the value of R1 at that point will be the result of the DADD. This unpredictable behavior is obviously unacceptable.

The AND instruction is also affected by this hazard. As we can see from [Figure C.6](#), the write of R1 does not complete until the end of clock cycle 5. Thus, the AND instruction that reads the registers during clock cycle 4 will receive the wrong results.

The XOR instruction operates properly because its register read occurs in clock cycle 6, after the register write. The OR instruction also operates without incurring a hazard because we perform the register file reads in the second half of the cycle and the writes in the first half.

The next subsection discusses a technique to eliminate the stalls for the hazard involving the DSUB and AND instructions.

Minimizing Data Hazard Stalls by Forwarding

The problem posed in [Figure C.6](#) can be solved with a simple hardware technique called *forwarding* (also called *bypassing* and sometimes *short-circuiting*). The key insight in forwarding is that the result is not really needed by the DSUB until