



Pipelining: Basic and Intermediate Concepts

It is quite a three-pipe problem.

Sir Arthur Conan Doyle
The Adventures of Sherlock Holmes

C.1

Introduction

Many readers of this text will have covered the basics of pipelining in another text (such as our more basic text *Computer Organization and Design*) or in another course. Because [Chapter 3](#) builds heavily on this material, readers should ensure that they are familiar with the concepts discussed in this appendix before proceeding. As you read [Chapter 2](#), you may find it helpful to turn to this material for a quick review.

We begin the appendix with the basics of pipelining, including discussing the data path implications, introducing hazards, and examining the performance of pipelines. This section describes the basic five-stage RISC pipeline that is the basis for the rest of the appendix. [Section C.2](#) describes the issue of hazards, why they cause performance problems, and how they can be dealt with. [Section C.3](#) discusses how the simple five-stage pipeline is actually implemented, focusing on control and how hazards are dealt with.

[Section C.4](#) discusses the interaction between pipelining and various aspects of instruction set design, including discussing the important topic of exceptions and their interaction with pipelining. Readers unfamiliar with the concepts of precise and imprecise interrupts and resumption after exceptions will find this material useful, since they are key to understanding the more advanced approaches in [Chapter 3](#).

[Section C.5](#) discusses how the five-stage pipeline can be extended to handle longer-running floating-point instructions. [Section C.6](#) puts these concepts together in a case study of a deeply pipelined processor, the MIPS R4000/4400, including both the eight-stage integer pipeline and the floating-point pipeline.

[Section C.7](#) introduces the concept of dynamic scheduling and the use of scoreboards to implement dynamic scheduling. It is introduced as a crosscutting issue, since it can be used to serve as an introduction to the core concepts in [Chapter 3](#), which focused on dynamically scheduled approaches. [Section C.7](#) is also a gentle introduction to the more complex Tomasulo's algorithm covered in [Chapter 3](#). Although Tomasulo's algorithm can be covered and understood without introducing scoreboarding, the scoreboarding approach is simpler and easier to comprehend.

What Is Pipelining?

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. Today, pipelining is the key implementation technique used to make fast CPUs.

A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car. Each step operates in parallel with the other steps, although on a different car. In a computer pipeline, each step in the pipeline completes a part of an instruction. Like the

assembly line, different steps are completing different parts of different instructions in parallel. Each of these steps is called a *pipe stage* or a *pipe segment*. The stages are connected one to the next to form a pipe—instructions enter at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

In an automobile assembly line, *throughput* is defined as the number of cars per hour and is determined by how often a completed car exits the assembly line. Likewise, the throughput of an instruction pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time, just as we would require in an assembly line. The time required between moving an instruction one step down the pipeline is a *processor cycle*. Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage, just as in an auto assembly line the longest step would determine the time between advancing the line. In a computer, this processor cycle is usually 1 clock cycle (sometimes it is 2, rarely more).

The pipeline designer's goal is to balance the length of each pipeline stage, just as the designer of the assembly line tries to balance the time for each step in the process. If the stages are perfectly balanced, then the time per instruction on the pipelined processor—assuming ideal conditions—is equal to

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

Under these conditions, the speedup from pipelining equals the number of pipe stages, just as an assembly line with n stages can ideally produce cars n times as fast. Usually, however, the stages will not be perfectly balanced; furthermore, pipelining does involve some overhead. Thus, the time per instruction on the pipelined processor will not have its minimum possible value, yet it can be close.

Pipelining yields a reduction in the average execution time per instruction. Depending on what you consider as the baseline, the reduction can be viewed as decreasing the number of clock cycles per instruction (CPI), as decreasing the clock cycle time, or as a combination. If the starting point is a processor that takes multiple clock cycles per instruction, then pipelining is usually viewed as reducing the CPI. This is the primary view we will take. If the starting point is a processor that takes 1 (long) clock cycle per instruction, then pipelining decreases the clock cycle time.

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike some speedup techniques (see [Chapter 4](#)), it is not visible to the programmer. In this appendix we will first cover the concept of pipelining using a classic five-stage pipeline; other chapters investigate the more sophisticated pipelining techniques in use in modern processors. Before we say more about pipelining and its use in a processor, we need a simple instruction set, which we introduce next.

The Basics of a RISC Instruction Set

Throughout this book we use a RISC (reduced instruction set computer) architecture or load-store architecture to illustrate the basic concepts, although nearly all the ideas we introduce in this book are applicable to other processors. In this section we introduce the core of a typical RISC architecture. In this appendix, and throughout the book, our default RISC architecture is MIPS. In many places, the concepts are significantly similar that they will apply to any RISC. RISC architectures are characterized by a few key properties, which dramatically simplify their implementation:

- All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register).
- The only operations that affect memory are load and store operations that move data from memory to a register or to memory from a register, respectively. Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.
- The instruction formats are few in number, with all instructions typically being one size.

These simple properties lead to dramatic simplifications in the implementation of pipelining, which is why these instruction sets were designed this way.

For consistency with the rest of the text, we use MIPS64, the 64-bit version of the MIPS instruction set. The extended 64-bit instructions are generally designated by having a D on the start or end of the mnemonic. For example DADD is the 64-bit version of an add instruction, while LD is the 64-bit version of a load instruction.

Like other RISC architectures, the MIPS instruction set provides 32 registers, although register 0 always has the value 0. Most RISC architectures, like MIPS, have three classes of instructions (see Appendix A for more detail):

1. *ALU instructions*—These instructions take either two registers or a register and a sign-extended immediate (called *ALU immediate instructions*, they have a 16-bit offset in MIPS), operate on them, and store the result into a third register. Typical operations include add (DADD), subtract (DSUB), and logical operations (such as AND or OR), which do not differentiate between 32-bit and 64-bit versions. Immediate versions of these instructions use the same mnemonics with a suffix of I. In MIPS, there are both signed and unsigned forms of the arithmetic instructions; the unsigned forms, which do not generate overflow exceptions—and thus are the same in 32-bit and 64-bit mode—have a U at the end (e.g., DADDU, DSUBU, DADDIU).
2. *Load and store instructions*—These instructions take a register source, called the *base register*, and an immediate field (16-bit in MIPS), called the *offset*, as operands. The sum—called the *effective address*—of the contents of the base register and the sign-extended offset is used as a memory address. In the case of a load instruction, a second register operand acts as the destination for the

data loaded from memory. In the case of a store, the second register operand is the source of the data that is stored into memory. The instructions load word (LD) and store word (SD) load or store the entire 64-bit register contents.

3. *Branches and jumps*—Branches are conditional transfers of control. There are usually two ways of specifying the branch condition in RISC architectures: with a set of condition bits (sometimes called a *condition code*) or by a limited set of comparisons between a pair of registers or between a register and zero. MIPS uses the latter. For this appendix, we consider only comparisons for equality between two registers. In all RISC architectures, the branch destination is obtained by adding a sign-extended offset (16 bits in MIPS) to the current PC. Unconditional jumps are provided in many RISC architectures, but we will not cover jumps in this appendix.

A Simple Implementation of a RISC Instruction Set

To understand how a RISC instruction set can be implemented in a pipelined fashion, we need to understand how it is implemented *without* pipelining. This section shows a simple implementation where every instruction takes at most 5 clock cycles. We will extend this basic implementation to a pipelined version, resulting in a much lower CPI. Our unpipelined implementation is not the most economical or the highest-performance implementation without pipelining. Instead, it is designed to lead naturally to a pipelined implementation. Implementing the instruction set requires the introduction of several temporary registers that are not part of the architecture; these are introduced in this section to simplify pipelining. Our implementation will focus only on a pipeline for an integer subset of a RISC architecture that consists of load-store word, branch, and integer ALU operations.

Every instruction in this RISC subset can be implemented in at most 5 clock cycles. The 5 clock cycles are as follows.

1. *Instruction fetch cycle (IF):*

Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 (since each instruction is 4 bytes) to the PC.

2. *Instruction decode/register fetch cycle (ID):*

Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers as they are read, for a possible branch. Sign-extend the offset field of the instruction in case it is needed. Compute the possible branch target address by adding the sign-extended offset to the incremented PC. In an aggressive implementation, which we explore later, the branch can be completed at the end of this stage by storing the branch-target address into the PC, if the condition test yielded true.

Decoding is done in parallel with reading registers, which is possible because the register specifiers are at a fixed location in a RISC architecture.

This technique is known as *fixed-field decoding*. Note that we may read a register we don't use, which doesn't help but also doesn't hurt performance. (It does waste energy to read an unneeded register, and power-sensitive designs might avoid this.) Because the immediate portion of an instruction is also located in an identical place, the sign-extended immediate is also calculated during this cycle in case it is needed.

3. *Execution/effective address cycle (EX):*

The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.

- Memory reference—The ALU adds the base register and the offset to form the effective address.
- Register-Register ALU instruction—The ALU performs the operation specified by the ALU opcode on the values read from the register file.
- Register-Immediate ALU instruction—The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.

In a load-store architecture the effective address and execution cycles can be combined into a single clock cycle, since no instruction needs to simultaneously calculate a data address and perform an operation on the data.

4. *Memory access (MEM):*

If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.

5. *Write-back cycle (WB):*

- Register-Register ALU instruction or load instruction:

Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

In this implementation, branch instructions require 2 cycles, store instructions require 4 cycles, and all other instructions require 5 cycles. Assuming a branch frequency of 12% and a store frequency of 10%, a typical instruction distribution leads to an overall CPI of 4.54. This implementation, however, is not optimal either in achieving the best performance or in using the minimal amount of hardware given the performance level; we leave the improvement of this design as an exercise for you and instead focus on pipelining this version.

The Classic Five-Stage Pipeline for a RISC Processor

We can pipeline the execution described above with almost no changes by simply starting a new instruction on each clock cycle. (See why we chose this design?)

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

Figure C.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write-back.

Each of the clock cycles from the previous section becomes a *pipe stage*—a cycle in the pipeline. This results in the execution pattern shown in [Figure C.1](#), which is the typical way a pipeline structure is drawn. Although each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions.

You may find it hard to believe that pipelining is as simple as this; it's not. In this and the following sections, we will make our RISC pipeline “real” by dealing with problems that pipelining introduces.

To start with, we have to determine what happens on every clock cycle of the processor and make sure we don't try to perform two different operations with the same data path resource on the same clock cycle. For example, a single ALU cannot be asked to compute an effective address and perform a subtract operation at the same time. Thus, we must ensure that the overlap of instructions in the pipeline cannot cause such a conflict. Fortunately, the simplicity of a RISC instruction set makes resource evaluation relatively easy. [Figure C.2](#) shows a simplified version of a RISC data path drawn in pipeline fashion. As you can see, the major functional units are used in different cycles, and hence overlapping the execution of multiple instructions introduces relatively few conflicts. There are three observations on which this fact rests.

First, we use separate instruction and data memories, which we would typically implement with separate instruction and data caches (discussed in [Chapter 2](#)). The use of separate caches eliminates a conflict for a single memory that would arise between instruction fetch and data memory access. Notice that if our pipelined processor has a clock cycle that is equal to that of the unpipelined version, the memory system must deliver five times the bandwidth. This increased demand is one cost of higher performance.

Second, the register file is used in the two stages: one for reading in ID and one for writing in WB. These uses are distinct, so we simply show the register file in two places. Hence, we need to perform two reads and one write every

clock cycle. To handle reads and a write to the same register (and for another reason, which will become obvious shortly), we perform the register write in the first half of the clock cycle and the read in the second half.

Third, Figure C.2 does not deal with the PC. To start a new instruction every clock, we must increment and store the PC every clock, and this must be done during the IF stage in preparation for the next instruction. Furthermore, we must also have an adder to compute the potential branch target during ID. One further problem is that a branch does not change the PC until the ID stage. This causes a problem, which we ignore for now, but will handle shortly.

Although it is critical to ensure that instructions in the pipeline do not attempt to use the hardware resources at the same time, we must also ensure that instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing *pipeline registers* between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle. Figure C.3 shows the pipeline drawn with these pipeline registers.

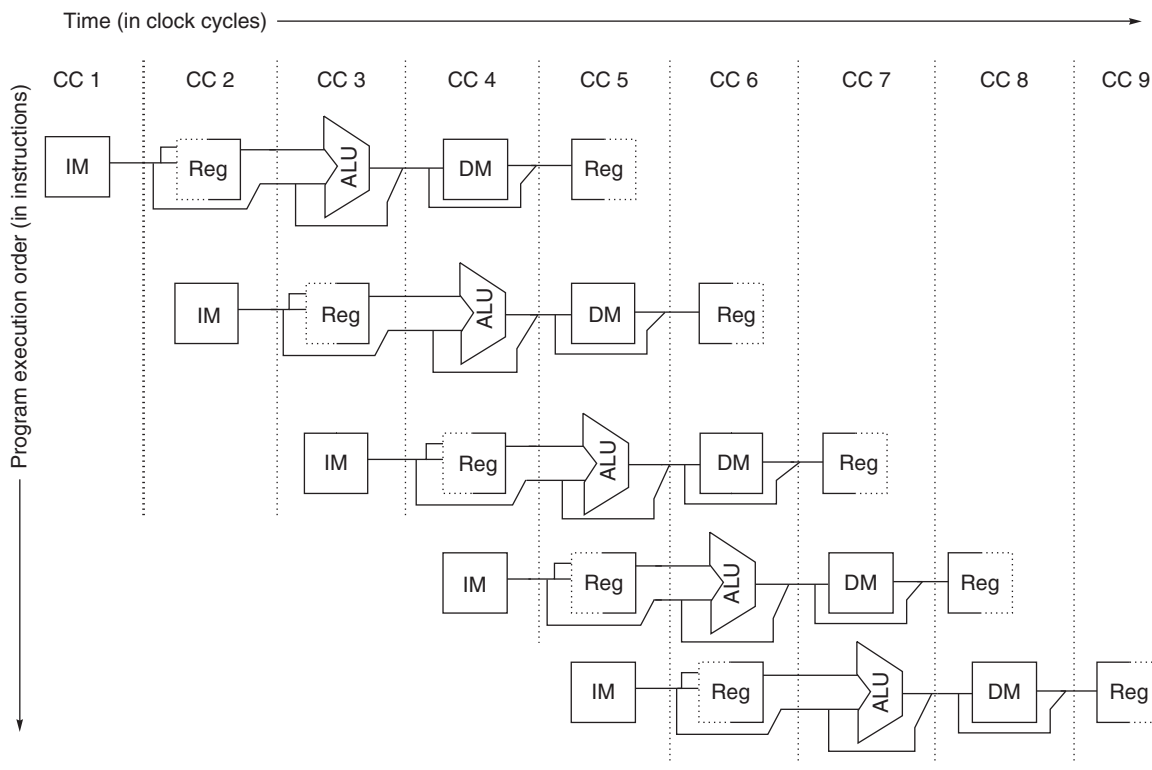


Figure C.2 The pipeline can be thought of as a series of data paths shifted in time. This shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on the other side. The abbreviation IM is used for instruction memory, DM for data memory, and CC for clock cycle.

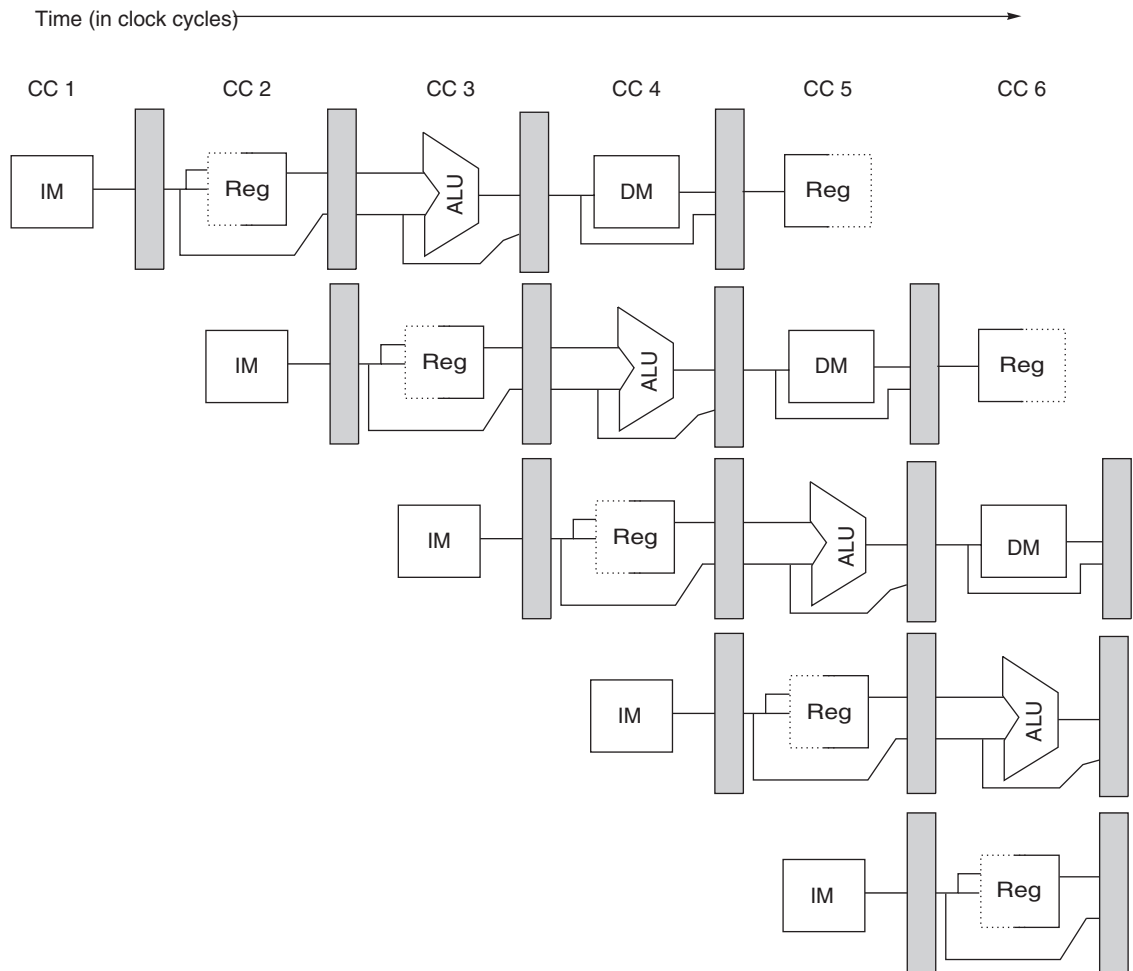


Figure C.3 A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers prevent interference between two different instructions in adjacent stages in the pipeline. The registers also play the critical role of carrying data for a given instruction from one stage to the other. The edge-triggered property of registers—that is, that the values change instantaneously on a clock edge—is critical. Otherwise, the data from one instruction could interfere with the execution of another!

Although many figures will omit such registers for simplicity, they are required to make the pipeline operate properly and must be present. Of course, similar registers would be needed even in a multicycle data path that had no pipelining (since only values in registers are preserved across clock boundaries). In the case of a pipelined processor, the pipeline registers also play the key role of carrying intermediate results from one stage to another where the source and destination may not be directly adjacent. For example, the register value to be stored

during a store instruction is read during ID, but not actually used until MEM; it is passed through two pipeline registers to reach the data memory during the MEM stage. Likewise, the result of an ALU instruction is computed during EX, but not actually stored until WB; it arrives there by passing through two pipeline registers. It is sometimes useful to name the pipeline registers, and we follow the convention of naming them by the pipeline stages they connect, so that the registers are called IF/ID, ID/EX, EX/MEM, and MEM/WB.

Basic Performance Issues in Pipelining

Pipelining increases the CPU instruction throughput—the number of instructions completed per unit of time—but it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the control of the pipeline. The increase in instruction throughput means that a program runs faster and has lower total execution time, even though no single instruction runs faster!

The fact that the execution time of each instruction does not decrease puts limits on the practical depth of a pipeline, as we will see in the next section. In addition to limitations arising from pipeline latency, limits arise from imbalance among the pipe stages and from pipelining overhead. Imbalance among the pipe stages reduces performance since the clock can run no faster than the time needed for the slowest pipeline stage. Pipeline overhead arises from the combination of pipeline register delay and clock skew. The pipeline registers add setup time, which is the time that a register input must be stable before the clock signal that triggers a write occurs, plus propagation delay to the clock cycle. Clock skew, which is maximum delay between when the clock arrives at any two registers, also contributes to the lower limit on the clock cycle. Once the clock cycle is as small as the sum of the clock skew and latch overhead, no further pipelining is useful, since there is no time left in the cycle for useful work. The interested reader should see Kunkel and Smith [1986]. As we saw in [Chapter 3](#), this overhead affected the performance gains achieved by the Pentium 4 versus the Pentium III.

Example Consider the unpipelined processor in the previous section. Assume that it has a 1 ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Answer The average instruction execution time on the unpipelined processor is

$$\begin{aligned}
 \text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI} \\
 &= 1 \text{ ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5] \\
 &= 1 \text{ ns} \times 4.4 \\
 &= 4.4 \text{ ns}
 \end{aligned}$$