

## MPCA UNIT 2 NOTES:

UNIT 2 Pipelining ( 12 hours)	
<b>4.1,4.2 of Text T2</b>  <b>Appendix C-1, C-2, Sec 1.1 , 1.4, 1.5 of T1</b>	Introduction to Pipelining,
	3 stage pipelining
	5 stage pipelining
	Pipeline hazards
	Structural Hazards
	Data Hazards .
	Data Hazards continued..
	Data hazards
	Control hazards- what is a control hazard, prediction algorithms- static, dynamic
	Static branch prediction
	Dynamic branch prediction
	Dynamic branch prediction
	Dynamic Branch predication

### 3-stage pipeline ARM organization:

The organization of an ARM with a 3-stage pipeline is illustrated in Figure 4.1 on page 76. The principal components are:

- The register bank, which stores the processor state. It has two read ports and one write port which can each be used to access any register, plus an additional read port and an additional write port that give special access to r15, the program counter. (The additional write port on r15 allows it to be updated as the instruction fetch address is incremented and the read port allows instruction fetch to resume after a data address has been issued.)
- The barrel shifter, which can shift or rotate one operand by any number of bits.
- The ALU, which performs the arithmetic and logic functions required by the instruction set.
- The address register and incrementer, which select and hold all memory addresses and generate sequential addresses when required.
- The data registers, which hold data passing to and from memory.
- The instruction decoder and associated control logic.

In a single-cycle data processing instruction, two register operands are accessed, the value on the B bus is shifted and combined with the value on the A bus in the ALU, then the result is written back into the register bank. The program counter value is in the address register, from where it is fed into

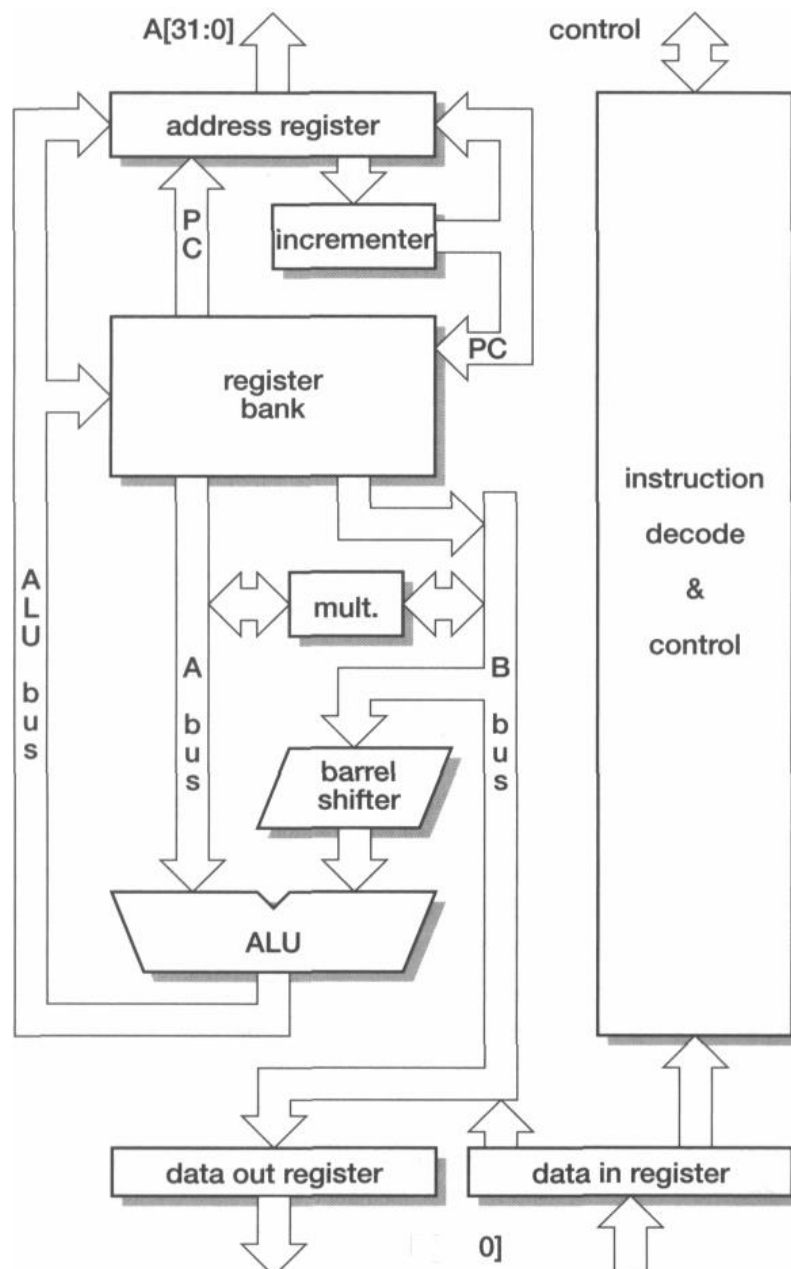
the incrementer, then the incremented value is copied back into r15 in the register bank and also into the address register to be used as the address for the next instruction fetch.

### The 3-stage pipeline

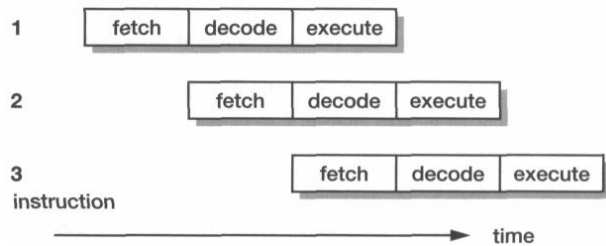
ARM processors up to the ARM7 employ a simple 3-stage pipeline with the following pipeline stages:

- Fetch; the instruction is fetched from memory and placed in the instruction pipeline.
- Decode; the instruction is decoded and the datapath control signals prepared for the next cycle. In this stage the instruction 'owns' the decode logic but not the datapath.
- Execute; the instruction 'owns' the datapath; the register bank is read, an operand shifted, the ALU result generated and written back into a destination register.

At any one time, three different instructions may occupy each of these stages, so the hardware in each stage has to be capable of independent operation.

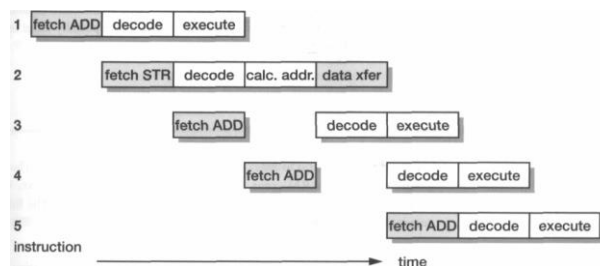


When the processor is executing simple data processing instructions the pipeline enables one instruction to be completed every clock cycle. An individual instruction takes three clock cycles to complete, so it has a three-cycle latency, but the throughput is one instruction per cycle.



When a multi-cycle instruction is executed the flow is less regular, as illustrated in Figure 4.3. This shows a sequence of single-cycle ADD instructions with a data store instruction, STR, occurring after the first ADD. The cycles that access main memory are shown with light shading so it can be seen that memory is used in every cycle.

The datapath is likewise used in every cycle, being involved in all the execute cycles, the address calculation and the data transfer. The decode logic is always generating the control signals for the datapath to use in the next cycle, so in addition to the explicit decode cycles it is also generating the control for the data transfer during the address calculation cycle of the STR.



Thus, in this instruction sequence, all parts of the processor are active in every cycle and the memory is the limiting factor, denning the number of cycles the sequence must take.

The simplest way to view breaks in the ARM pipeline is to observe that:

- All instructions occupy the datapath for one or more adjacent cycles.
- For each cycle that an instruction occupies the datapath, it occupies the decode logic in the immediately preceding cycle.
- During the first datapath cycle each instruction issues a fetch for the next instruction but one.
- Branch instructions flush and refill the instruction pipeline

One consequence of the pipelined execution model used on the ARM is that the program counter, which is visible to the user as r!5, must run ahead of the current instruction. If, as noted above, instructions fetch the next instruction but one during their first cycle, this suggests that the PC must point eight bytes (two instructions) ahead of the current instruction.

This is, indeed, what happens, and the programmer who attempts to access the PC directly through r!5 must take account of the exposure of the pipeline here. However, for most normal purposes the assembler or compiler handles all the details.

Even more complex behaviour is exposed if r15 is used later than the first cycle of an instruction, since the instruction will itself have incremented the PC during its first cycle. Such use of the PC is not often beneficial so the ARM architecture definition specifies the result as 'unpredictable' and it should be avoided, especially since later ARMs do not have the same behaviour in these cases.

## 5-stage pipeline ARM organization

All processors have to develop to meet the demand for higher performance. The 3-stage pipeline used in the ARM cores up to the ARM7 is very cost-effective, but higher performance requires the processor organization to be rethought. The time,  $T$ , required to execute a given program is given by:

$$T_{prog} = \frac{N_{inst} \times CPI}{f_{clk}}, \quad \text{Equation 11}$$

where  $N_{inst}$  is the number of ARM instructions executed in the course of the program,  $CPI$  is the average number of clock cycles per instruction and  $f_{clk}$  is the processor's clock frequency. Since  $N_{inst}$  is constant for a given program

There are only two ways to increase performance:

- Increase the clock rate,  $f_{clk}$ : This requires the logic in each pipeline stage to be simplified and, therefore, the number of pipeline stages to be increased.
- Reduce the average number of clock cycles per instruction,  $CPI$ : This requires either that instructions which occupy more than one pipeline slot in a 3-stage pipeline ARM are re-implemented to occupy fewer slots, or that pipeline stalls caused by dependencies between instructions are reduced, or a combination of both.

The fundamental problem with reducing the CPI relative to a 3-stage core is related to the von Neumann bottleneck - any stored-program computer with a single instruction and data memory will have its performance limited by the available memory bandwidth. A 3-stage ARM core accesses memory on (almost) every clock cycle either to fetch an instruction or to transfer data. Simply tightening up on the few cycles where the memory is not used will yield only a small performance gain.

To get a significantly better CPI the memory system must deliver more than one value in each clock cycle either by delivering more than 32 bits per cycle from a single memory or by having separate memories for instruction and data accesses.

As a result of the above issues, higher performance ARM cores employ a 5-stage pipeline and have separate instruction and data memories. Breaking instruction execution down into five components rather than three reduces the maximum work which must be completed in a clock cycle, and hence allows a higher clock frequency to be used (provided that other system components, and particularly the instruction memory, are also redesigned to operate at this higher clock rate). The separate instruction and data memories (which may be separate caches connected to a unified instruction and data main memory) allow a significant reduction in the core's CPI.

The ARM processors which use a 5-stage pipeline have the following pipeline stages:

- Fetch;  
the instruction is fetched from memory and placed in the instruction pipeline.
- Decode;  
the instruction is decoded and register operands read from the register file. There are three operand read ports in the register file, so most ARM instructions can source all their operands in one cycle.

Execute;

an operand is shifted and the ALU result generated. If the instruction is a load or store the memory address is computed in the ALU.

- Buffer/data;

data memory is accessed if required. Otherwise the ALU result is simply buffered for one clock cycle to give the same pipeline flow for all instructions.

- Write-back;

the results generated by the instruction are written back to the register file, including any data loaded from memory.

This 5-stage pipeline has been used for many RISC processors and is considered to be the 'classic' way to design such a processor. Although the ARM instruction set was not designed with such a pipeline in mind, it maps onto it relatively simply. The principal concessions to the ARM instruction set architecture in the organization shown in Figure 4.4 on page 81 are the three source operand read ports and two write ports in the register file (where a 'classic' RISC has two read ports and one write port), and the inclusion of address incrementing hardware in the execute stage to support load and store multiple instructions

Complexity in the 5-stage pipeline (compared to the 3-stage pipeline) is that, because instruction execution is spread across three pipeline stages, the only way to resolve data dependencies without stalling the pipeline is to introduce *forwarding* paths.

Data dependencies arise when an instruction needs to use the result of one of its predecessors before that result has returned to the register file. Forwarding paths allow results to be passed between stages as soon as they are available, and the 5-stage ARM pipeline requires each of the three source operands to be forwarded from any of three intermediate result registers.

There is one case where, even with forwarding, it is not possible to avoid a pipeline stall. Consider the following code sequence:

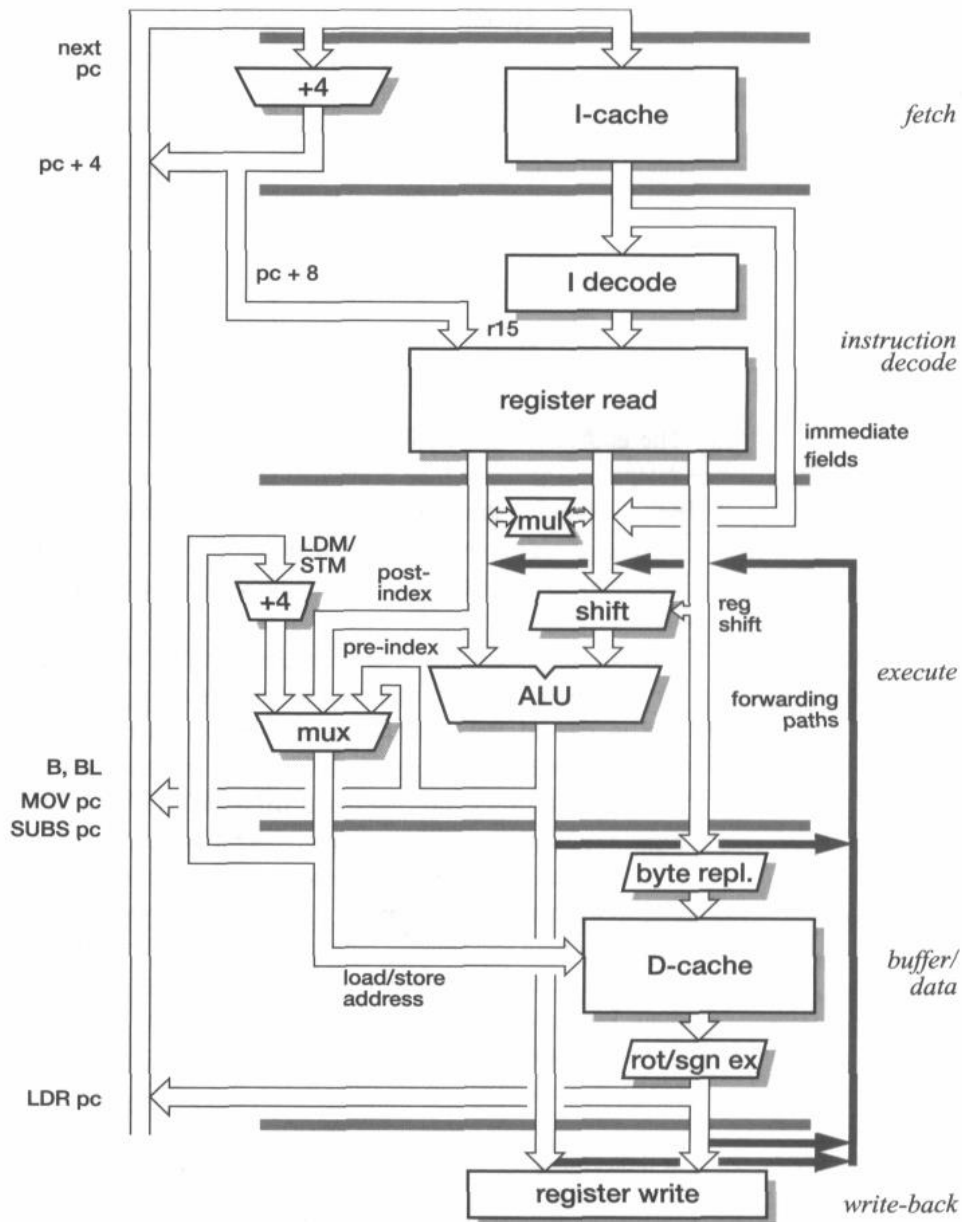
```
LDR  rN, [ . . ]           ; load rN from somewhere
ADD  r2, r1, rN            ; and use it immediately
```

The processor cannot avoid a one-cycle stall as the value loaded into rN only enters the processor at the end of the buffer/data stage and it is needed by the following instruction at the start of the execute stage. The only way to avoid this stall is to encourage the compiler (or assembly language programmer) not to put a dependent instruction immediately after a load instruction.

Since the 3-stage pipeline ARM cores are not adversely affected by this code sequence, existing ARM programs will often use it. Such programs will run correctly on 5-stage ARM cores, but could probably be rewritten to run faster by simply reordering the instructions to remove these dependencies.

The behaviour of r15, as seen by the programmer and described in 'PC behaviour' on , is based on the operational characteristics of the 3-stage ARM pipeline.

The 5-stage pipeline reads the instruction operands one stage earlier in the pipeline, and would naturally get a different value (PC+4 rather than PC+8). As this would lead to unacceptable code incompatibilities, however, the 5-stage pipeline ARMs all 'emulate' the behaviour of the older 3-stage designs. Referring to Figure 4.4, the incremented PC value from the fetch stage is fed directly to the register file in the decode stage, bypassing the pipeline register between the two stages. PC+4 for the next instruction is equal to PC+8 for the current instruction, so the correct r15 value is obtained without additional hardware.



# Pipelining: Basic and Intermediate Concepts

## 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, 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.

## 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.

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

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. In all RISC architectures, the branch destination is obtained by adding a sign-extended offset (16 bits in MIPS) to the current PC.

## 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.

## 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

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.

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).

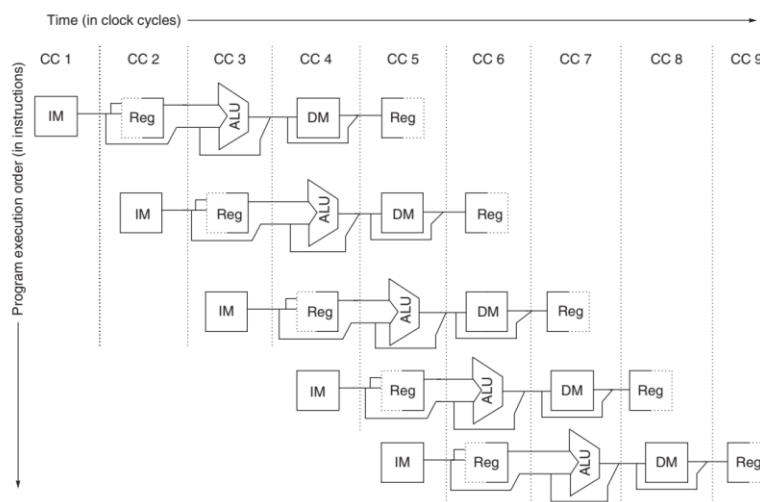
The use of separate caches eliminates a conflict for a single memory that would arise between instruction fetch and data memory access. Notice that if our pipelined processor has a clock cycle that is equal to that of the unpipelined version, the memory system must deliver five times the bandwidth. This 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.

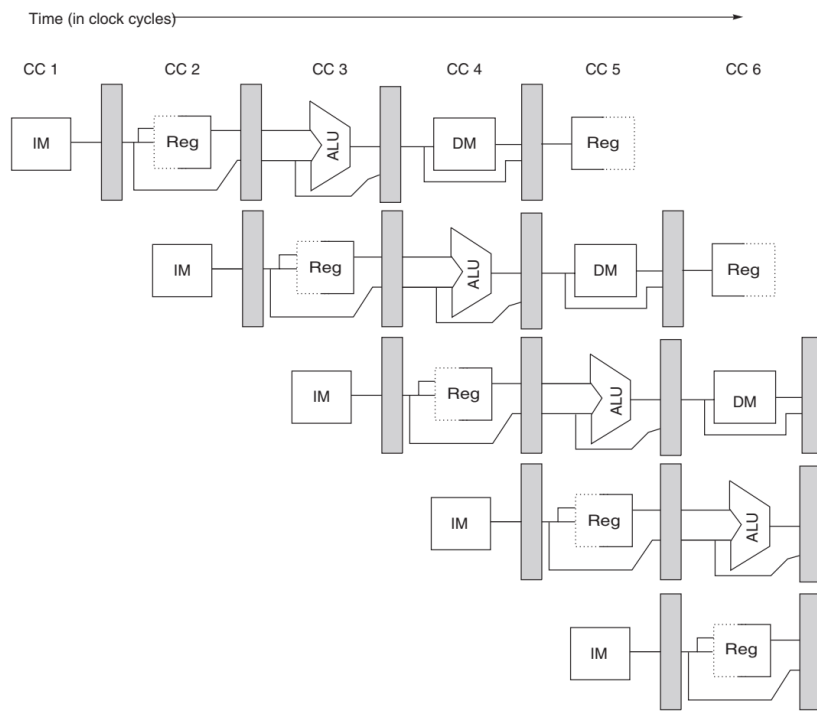
Third, Figure C.2 does not deal with the PC. To start a new instruction every clock, we must increment and store the PC every clock, and this must be done during the IF stage in preparation for the next instruction.

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.

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

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

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.

## 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, 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:

$$\text{Clock cycle pipelined} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}}$$

$$\text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

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

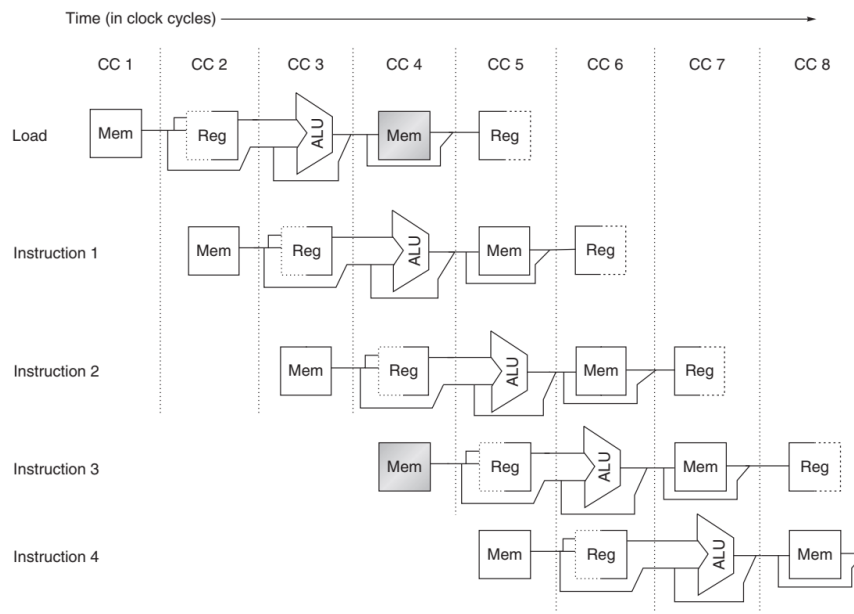
When a processor is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions cannot be accommodated because of resource conflicts, the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



**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.

## 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<sub>ideal</sub>. 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.

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.

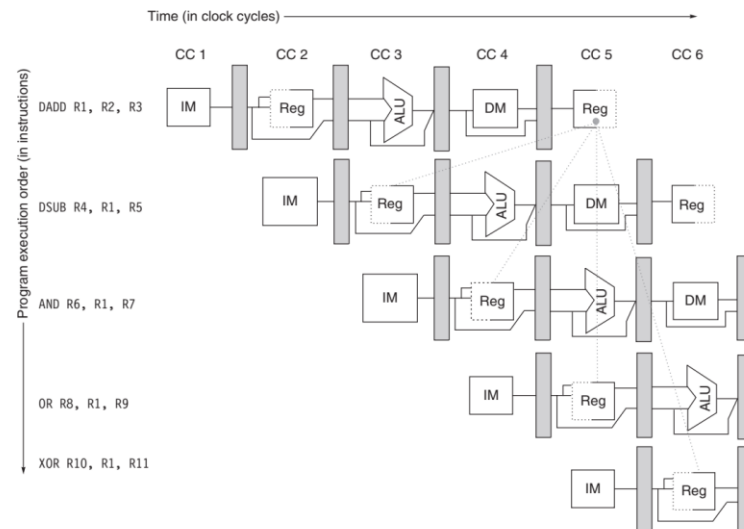
The XOR instruction operates properly because its register read occurs in clock cycle 6, after the register write. The OR instruction also operates without incurring a hazard because we perform the register file reads in the second half of the cycle and the writes in the first half.



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



**Figure C.6** The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

by the DSUB until

after the DADD actually produces it. If the result can be moved from the pipeline register where the DADD stores it to where the DSUB needs it, then the need for a stall can be avoided. Using this observation, forwarding works as follows:

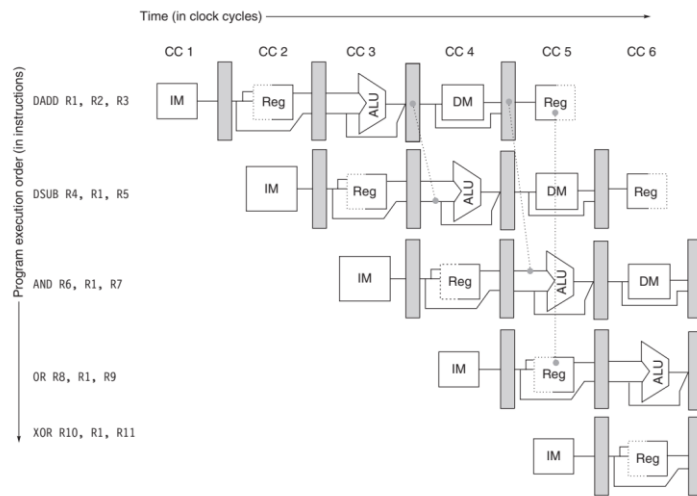
1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.
2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Notice that with forwarding, if the DSUB is stalled, the DADD will be completed and the bypass will not be activated. This relationship is also true for the case of an interrupt between the two instructions.

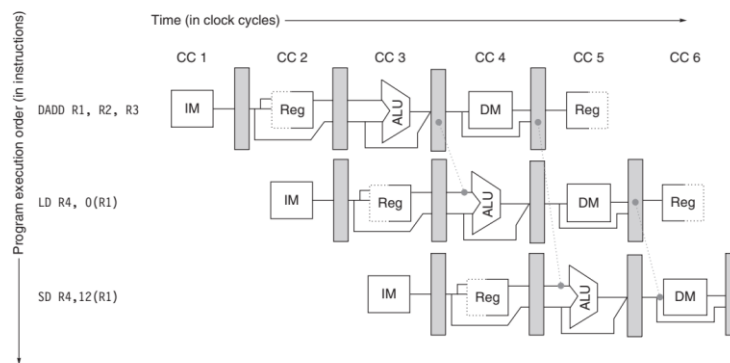
As the example in [Figure C.6](#) shows, we need to forward results not only from the immediately previous instruction but also possibly from an instruction that started 2 cycles earlier. [Figure C.7](#) shows our example with the bypass paths in place and highlighting the timing of the register read and writes. This code sequence can be executed without stalls.

Forwarding can be generalized to include passing a result directly to the functional unit that requires it: A result is forwarded from the pipeline register corresponding to the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit. Take, for example, the following sequence:

```
DADD    R1,R2,R3
LD      R4,0(R1)
SD      R4,12(R1)
```



**Figure C.7** A set of instructions that depends on the DADD result uses forwarding paths to avoid the data hazard. The inputs for the DSUB and AND instructions forward from the pipeline registers to the first ALU input. The OR receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the AND instruction was AND R6, R1, R4.



**Figure C.8** Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown above), the result would need to be forwarded to prevent a stall.

To prevent a stall in this sequence, we would need to forward the values of the ALU output and memory unit output from the pipeline registers to the ALU and data memory inputs. [Figure C.8](#) shows all the forwarding paths for this example.

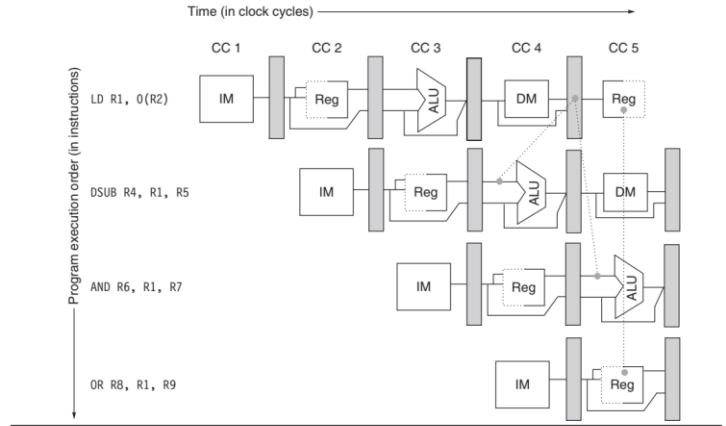
### Data Hazards Requiring Stalls

Unfortunately, not all potential data hazards can be handled by bypassing. Consider the following sequence of instructions:

```
LD      R1, 0(R2)
DSUB    R4, R1, R5
AND     R6, R1, R7
OR      R8, R1, R9
```

The pipelined data path with the bypass paths for this example is shown in Figure C.9. This case is different from the situation with back-to-back ALU operations. The LD instruction does not have the data until the end of clock cycle 4 (its MEM cycle), while the DSUB instruction needs to have the data by the beginning of that clock cycle. Thus, the data hazard from using the result of a load instruction cannot be completely eliminated with simple hardware.

As Figure C.9 shows, such a forwarding path would have to operate backward



**Figure C.9** The load instruction can bypass its results to the AND and OR instructions, but not to the DSUB, since that would mean forwarding the result in “negative time.”

in time—a capability not yet available to computer designers! We *can* forward the result immediately to the ALU from the pipeline registers for use in the AND operation, which begins 2 clock cycles after the load. Likewise, the OR instruction has no problem, since it receives the value through the register file.

For the DSUB instruction, the forwarded result arrives too late—at the end of a clock cycle, when it is needed at the beginning. The load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, we need to add hardware, called a *pipeline interlock*, to preserve the correct execution pattern. In general, a pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared. In this case, the interlock stalls the pipeline, beginning with the instruction that wants to use the data until the source instruction produces it. This pipeline interlock introduces a stall or bubble, just as it did for the structural hazard. The CPI for the stalled instruction increases by the length of the stall (1 clock cycle in this case). Figure C.10 shows the pipeline before and after the stall using the names of the pipeline stages. Because the stall causes the instructions starting with the DSUB to move 1 cycle later in time, the forwarding to the AND instruction now goes through the register file, and no forwarding at all is needed for the OR instruction.

The insertion of the bubble causes the number of cycles to complete this sequence to grow by one. No instruction is started during clock cycle 4 (and none finishes during cycle 6)

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB			
AND	R6,R1,R7			IF	ID	EX	MEM	WB		
OR	R8,R1,R9				IF	ID	EX	MEM	WB	
LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND	R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR	R8,R1,R9				stall	IF	ID	EX	MEM	WB

**Figure C.10** In the top half, we can see why a stall is needed: The MEM cycle of the load produces a value that is needed in the EX cycle of the DSUB, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

## Branch Hazards

*Control hazards* can cause a greater performance loss for our MIPS pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4. Recall that if a branch changes the PC to its target address, it is a *taken* branch; if it falls through, it is *not taken*, or *untaken*. If instruction  $i$  is a taken branch, then the PC is normally not changed until the end of ID, after the completion of the address calculation and comparison. Figure C.11 shows that the simplest method of dealing with branches is to redo the fetch of the instruction following a branch, once we detect the branch during ID (when instructions are decoded). The first IF cycle is essentially a stall, because it never performs useful work. You may have noticed that if the branch is untaken, then the repetition of the IF stage is unnecessary since the correct instruction was indeed fetched. We will develop several schemes to take advantage of this fact shortly. One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency, so we will examine some techniques to deal with this loss.

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

**Figure C.11** A branch causes a one-cycle stall in the five-stage pipeline. The instruction after the branch is fetched, but the instruction is ignored, and the fetch is restarted once the branch target is known. It is probably obvious that if the branch is not taken, the second IF for branch successor is redundant. This will be addressed shortly.

### Reducing Pipeline Branch Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay; we discuss four simple compile time schemes in this subsection. In these four schemes the actions for a branch are static—they are fixed for each branch during the entire execution. The software can try to minimize the branch penalty using knowledge of the hardware scheme and of branch behavior.

The simplest scheme to handle branches is to *freeze* or *flush* the pipeline, holding or deleting any instructions after the branch until the branch destination is known. The attractiveness of this solution lies primarily in its simplicity both for hardware and software. It is the solution used earlier in the pipeline shown in Figure C.11. In this case, the branch penalty is fixed and cannot be reduced by software.

A higher-performance, and only slightly more complex, scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed. Here, care must be taken not to change the processor state until the branch outcome is definitely known. The complexity of this scheme arises from having to know when the state might be changed by an instruction and how to “back out” such a change.

In the simple five-stage pipeline, this *predicted-not-taken* or *predicted untaken* scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction. The pipeline looks as if nothing out of the ordinary is happening. If the branch is taken, however, we need to turn the fetched instruction into a no-op and restart the fetch at the target address. Figure C.12 shows both situations.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB
Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

**Figure C.12** The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

An alternative scheme is to treat every branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target. Because in our five-stage pipeline we don't know the target address any earlier than we know the branch outcome, there is no advantage in this approach for this pipeline. In some processors—especially those with implicitly set condition codes or more powerful (and hence slower) branch conditions—the branch target is known before the branch outcome, and a predicted-taken scheme might make sense. In either a predicted taken or predicted-not-taken scheme, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware's choice. Our fourth scheme provides more opportunities for the compiler to improve performance.

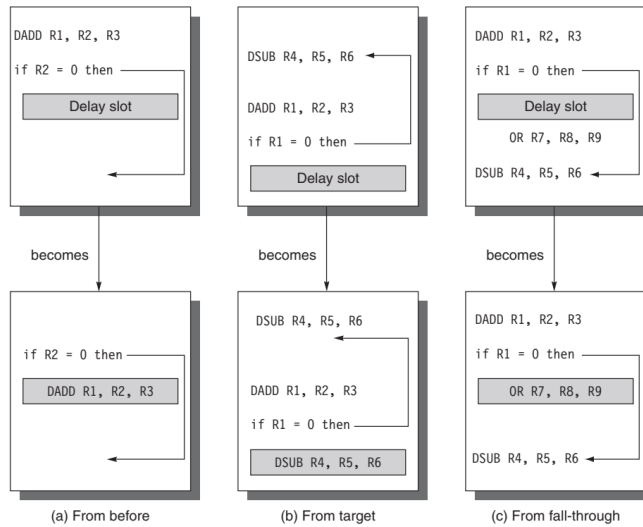
A fourth scheme in use in some processors is called *delayed branch*. This technique was heavily used in early RISC processors and works reasonably well in the five-stage pipeline. In a delayed branch, the execution cycle with a branch delay of one is

- branch instruction
- sequential successor
- branch target if taken

The sequential successor is in the *branch delay slot*. This instruction is executed whether or not the branch is taken. The pipeline behavior of the five-stage pipeline with a branch delay is shown in [Figure C.13](#). Although it is possible to have a branch delay longer than one, in practice almost all processors with delayed branch have a single instruction delay; other techniques are used if the pipeline has a longer potential branch penalty.

Untaken branch instruction	IF	ID	EX	MEM	WB			
Branch delay 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
Taken branch instruction	IF	ID	EX	MEM	WB			
Branch delay instruction ( $i + 1$ )		IF	ID	EX	MEM	WB		
Branch target			IF	ID	EX	MEM	WB	
Branch target + 1				IF	ID	EX	MEM	WB
Branch target + 2					IF	ID	EX	MEM WB

**Figure C.13** The behavior of a delayed branch is the same whether or not the branch is taken. The instructions in the delay slot (there is only one delay slot for MIPS) are executed. If the branch is untaken, execution continues with the instruction after the branch delay instruction; if the branch is taken, execution continues at the branch target. When the instruction in the branch delay slot is also a branch, the meaning is unclear: If the branch is not taken, what should happen to the branch in the branch delay slot? Because of this confusion, architectures with delay branches often disallow putting a branch in the delay slot.



**Figure C.14 Scheduling the branch delay slot.** The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of R1 in the branch condition prevents the DADD instruction (whose destination is R1) from being moved after the branch. In (b), the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the moved instruction when the branch goes in the unexpected direction. By OK we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, in (c) if R7 were an unused temporary register when the branch goes in the unexpected direction.

The job of the compiler is to make the successor instructions valid and useful. A number of optimizations are used. Figure C.14 shows the three ways in which the branch delay can be scheduled. The limitations on delayed-branch scheduling arise from:

- (1) the restrictions on the instructions that are scheduled into the delay slots, and
- (2) our ability to predict at compile time whether a branch is likely to be taken or not.

To improve the ability of the compiler to fill branch delay slots, most processors with conditional branches have introduced a *canceled* or *nullifying* branch. In a canceled branch, the instruction includes the delay slot is simply executed as it would direction that the branch was predicted. When the branch behaves as predicted, the instruction in the branch delay slot is simply executed as it would normally be with a delayed branch. When the branch is incorrectly predicted, the instruction in the branch delay slot is simply turned into a no-op.

### Performance of Branch Schemes

What is the effective performance of each of these schemes? The effective pipeline speedup with branch penalties, assuming an ideal CPI of 1, is

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Because of the following:

$$\text{Pipeline stall cycles from branches} = \text{Branch frequency} \times \text{Branch penalty}$$

we obtain:

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

The branch frequency and branch penalty can have a component from both unconditional and conditional branches. However, the latter dominate since they are more frequent.

**Example** For a deeper pipeline, such as that in a MIPS R4000, it takes at least three pipeline stages before the branch-target address is known and an additional cycle before the branch condition is evaluated, assuming no stalls on the registers in the conditional comparison. A three-stage delay leads to the branch penalties for the three simplest prediction schemes listed in Figure C.15.

Find the effective addition to the CPI arising from branches for this pipeline, assuming the following frequencies:

Unconditional branch	4%
Conditional branch, untaken	6%
Conditional branch, taken	10%

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

Figure C.15 Branch penalties for the three simplest prediction schemes for a deeper pipeline.

Branch scheme	Additions to the CPI from branch costs			
	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

Figure C.16 CPI penalties for three branch-prediction schemes and a deeper pipeline.

*Answer* We find the CPIs by multiplying the relative frequency of unconditional, conditional untaken, and conditional taken branches by the respective penalties. The results are shown in Figure C.16.

The differences among the schemes are substantially increased with this longer delay. If the base CPI were 1 and branches were the only source of stalls, the ideal pipeline would be 1.56 times faster than a pipeline that used the stall-pipeline scheme. The predicted-untaken scheme would be 1.13 times better than the stall-pipeline scheme under the same assumptions

## Reducing the Cost of Branches through Prediction

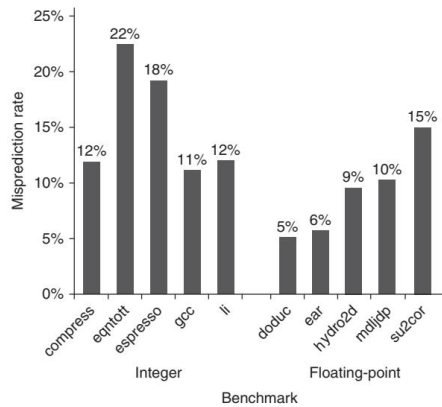
As pipelines get deeper and the potential penalty of branches increases, using delayed branches and similar schemes becomes insufficient. Instead, we need to turn to more aggressive means for predicting branches. Such schemes fall into two classes: low-cost static schemes that rely on information available at compile time and strategies that predict branches dynamically based on program behavior.

### Static Branch Prediction

A key way to improve compile-time branch prediction is to use profile information collected from earlier runs. The key observation that makes this worthwhile is that the behavior of branches is often bimodally distributed; that is, an individual branch is often highly biased toward taken or untaken. Figure C.17 shows the success of branch prediction using this strategy. The same input data were used for runs and for collecting the profile; other studies have shown that changing the input so that the profile is for a different run leads to only a small change in the accuracy of profile-based prediction.

The effectiveness of any branch prediction scheme depends both on the accuracy of the scheme and the frequency of conditional branches, which vary in SPEC from 3% to 24%. The fact that the misprediction rate for the integer programs is higher and such programs typically have a higher branch frequency is a major limitation for static branch prediction. In the next section, we consider dynamic branch predictors, which most recent processors have employed.





**Figure C.17** Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which vary from 3% to 24%.

## Dynamic Branch Prediction and Branch-Prediction Buffers

The simplest dynamic branch-prediction scheme is a *branch-prediction buffer* or *branch history table*. A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. This scheme is the simplest sort of buffer; it has no tags and is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs. With such a buffer, we don't know, in fact, if the prediction is correct—it may have been put there by another branch that has the same low-order address bits. But this doesn't matter. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.

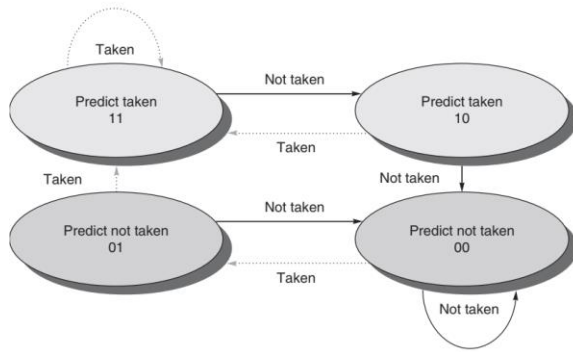
This buffer is effectively a cache where every access is a hit, and, as we will see, the performance of the buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches. Before we analyze the performance, it is useful to make a small, but important, improvement in the accuracy of the branch-prediction scheme.

This simple 1-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, since the misprediction causes the prediction bit to be flipped.

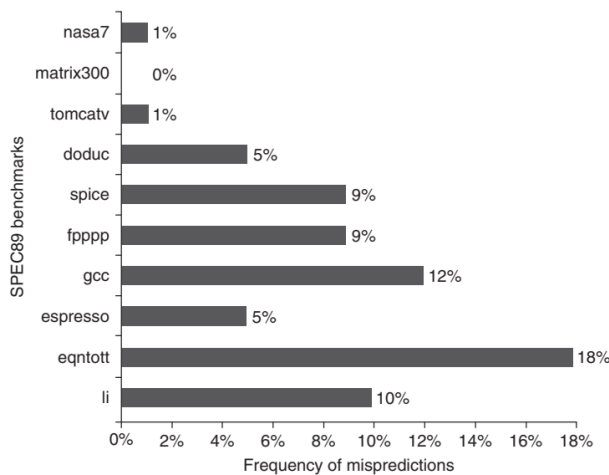
To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must miss twice before it is changed. [Figure C.18](#) shows the finite-state processor for a 2-bit prediction scheme.

A branch-prediction buffer can be implemented as a small, special “cache” accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue. As [Figure C.18](#) shows, if the prediction turns out to be wrong, the prediction bits are changed. What kind of accuracy can be expected from a branch-prediction buffer using 2 bits per entry on real applications? [Figure C.19](#) shows that for the SPEC89





**Figure C.18** The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an  $n$ -bit saturating counter for each entry in the prediction buffer. With an  $n$ -bit counter, the counter can take on values between 0 and  $2^n - 1$ : When the counter is greater than or equal to one-half of its maximum value ( $2^{n-1}$ ), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of  $n$ -bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general  $n$ -bit predictors.



**Figure C.19** Prediction accuracy of a 4096-entry 2-bit prediction buffer for the SPEC89 benchmarks. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the floating-point programs (average of 4%). Omitting the floating-point kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch-prediction study done using the IBM Power architecture and optimized code for that system. See Pan, So, and Rameh [1992]. Although these data are for an older version of a subset of the SPEC benchmarks, the newer benchmarks are larger and would show slightly worse behavior, especially for the integer benchmarks.

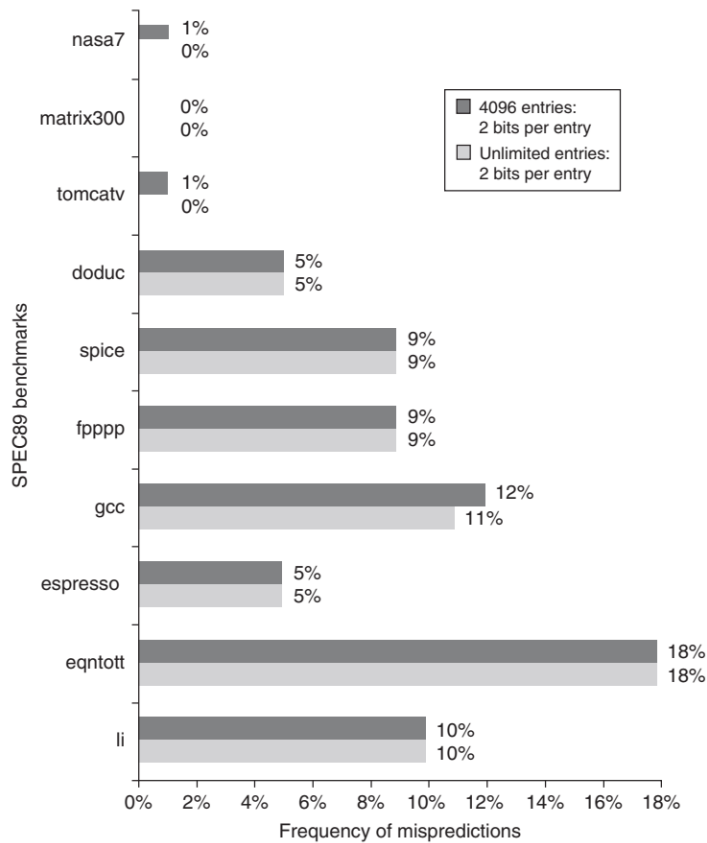
benchmarks a branch-prediction buffer with 4096 entries results in a prediction accuracy ranging from over 99% to 82%, or a *misprediction rate* of 1% to 18%. A 4K entry buffer, like that used for these results, is considered small by 2005 standards, and a larger buffer could produce somewhat better results.

As we try to exploit more ILP, the accuracy of our branch prediction becomes critical. As we can see in [Figure C.19](#), the accuracy of the predictors for integer programs, which typically also have higher branch frequencies, is lower than for the loop-intensive scientific programs.

We can attack this problem in two ways:

by increasing the size of the buffer and by increasing the accuracy of the scheme we use for each prediction. A buffer with 4K entries, however, as [Figure C.20](#) shows, performs quite comparably to an infinite buffer, at least for benchmarks like those in SPEC. The data in [Figure C.20](#) make it clear that the hit rate of the buffer is not the

major limiting factor. As we mentioned above, simply increasing



**Figure C.20** Prediction accuracy of a 4096-entry 2-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks. Although these data are for an older version of a subset of the SPEC benchmarks, the results would be comparable for newer versions with perhaps as many as 8K entries needed to match an infinite 2-bit predictor.

the number of bits per predictor without changing the predictor structure also has little impact. Instead, we need to look at how we might increase the accuracy of each predictor.