	Additions to the CPI from branch costs			
Branch scheme	**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 A.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 A.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.

A.3 How Is Pipelining Implemented?

Before we proceed to basic pipelining, we need to review a simple implementation of an unpipelined version of MIPS.

A Simple Implementation of MIPS

In this section we follow the style of Section A.1, showing first a simple unpipelined implementation and then the pipelined implementation. This time, however, our example is specific to the MIPS architecture.

In this subsection we focus on a pipeline for an integer subset of MIPS that consists of load-store word, branch equal zero, and integer ALU operations. Later in this appendix, we will incorporate the basic floating-point operations. Although we discuss only a subset of MIPS, the basic principles can be extended to handle all the instructions. We initially used a less aggressive implementation of a branch instruction. We show how to implement the more aggressive version at the end of this section.

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

1. *Instruction fetch cycle* (IF):

```
IR ← Mem[PC];
NPC ← PC + 4;
```

Operation: Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction. The IR is used to hold the instruction that will be needed on subsequent clock cycles; likewise the register NPC is used to hold the next sequential PC.

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

```
A ← Regs[rs];
B ← Regs[rt];
Imm ← sign-extended immediate field of IR;
```

Operation: Decode the instruction and access the register file to read the registers (rs and rt are the register specifiers). The outputs of the general-purpose registers are read into two temporary registers (A and B) for use in later clock cycles. The lower 16 bits of the IR are also sign extended and stored into the temporary register Imm, for use in the next cycle.

Decoding is done in parallel with reading registers, which is possible because these fields are at a fixed location in the MIPS instruction format (see Figure B.22 on page B-35). Because the immediate portion of an instruction is located in an identical place in every MIPS format, the sign-extended immediate is also calculated during this cycle in case it is needed in the next cycle.

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

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

- Memory reference:

```
ALUOutput ← A + Imm;
```

Operation: The ALU adds the operands to form the effective address and places the result into the register ALUOutput.

- Register-Register ALU instruction:

```
ALUOutput ← A func B;
```

Operation: The ALU performs the operation specified by the function code on the value in register A and on the value in register B. The result is placed in the temporary register ALUOutput.

- Register-Immediate ALU instruction:

```
ALUOutput ← A op Imm;
```

Operation: The ALU performs the operation specified by the opcode on the value in register A and on the value in register Imm. The result is placed in the temporary register ALUOutput.

- Branch:

```
ALUOutput ← NPC + (Imm << 2);
Cond ← (A == 0)
```

Operation: The ALU adds the NPC to the sign-extended immediate value in Imm, which is shifted left by 2 bits to create a word offset, to compute the address of the branch target. Register A, which has been read in the prior cycle, is checked to determine whether the branch is taken. Since we are considering only one form of branch (BEQZ), the comparison is against 0. Note that BEQZ is actually a pseudoinstruction that translates to a BEQ with R0 as an operand. For simplicity, this is the only form of branch we consider.

The load-store architecture of MIPS means that effective address and execution cycles can be combined into a single clock cycle, since no instruction needs to simultaneously calculate a data address, calculate an instruction target address, and perform an operation on the data. The other integer instructions not included above are jumps of various forms, which are similar to branches.

4. *Memory access/branch completion cycle* (MEM):

   The PC is updated for all instructions: `PC ← NPC;`

   - Memory reference:

     ```
     LMD ← Mem[ALUOutput] or
     Mem[ALUOutput] ← B;
     ```

   Operation: Access memory if needed. If instruction is a load, data returns from memory and is placed in the LMD (load memory data) register; if it is a store, then the data from the B register is written into memory. In either case the address used is the one computed during the prior cycle and stored in the register ALUOutput.

   - Branch:

     ```
     if (cond) PC ← ALUOutput
     ```

   Operation: If the instruction branches, the PC is replaced with the branch destination address in the register ALUOutput.

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

   - Register-Register ALU instruction:

     ```
     Regs[rd] ← ALUOutput;
     ```

   - Register-Immediate ALU instruction:

     ```
     Regs[rt] ← ALUOutput;
     ```

   - Load instruction:

     ```
     Regs[rt] ← LMD;
     ```

   Operation: Write the result into the register file, whether it comes from the memory system (which is in LMD) or from the ALU (which is in ALUOutput); the register destination field is also in one of two positions (rd or rt) depending on the effective opcode.

Figure A.17 shows how an instruction flows through the data path. At the end of each clock cycle, every value computed during that clock cycle and required on a later clock cycle (whether for this instruction or the next) is written into a storage device, which may be memory, a general-purpose register, the PC, or a temporary register (i.e., LMD, Imm, A, B, IR, NPC, ALUOutput, or Cond). The temporary registers hold values between clock cycles for one instruction, while the other storage elements are visible parts of the state and hold values between successive instructions.

Although all processors today are pipelined, this multicycle implementation is a reasonable approximation of how most processors would have been implemented in earlier times. A simple finite-state machine could be used to implement the control following the 5-cycle structure shown above. For a much more complex processor, microcode control could be used. In either event, an instruction sequence like that above would determine the structure of the control.

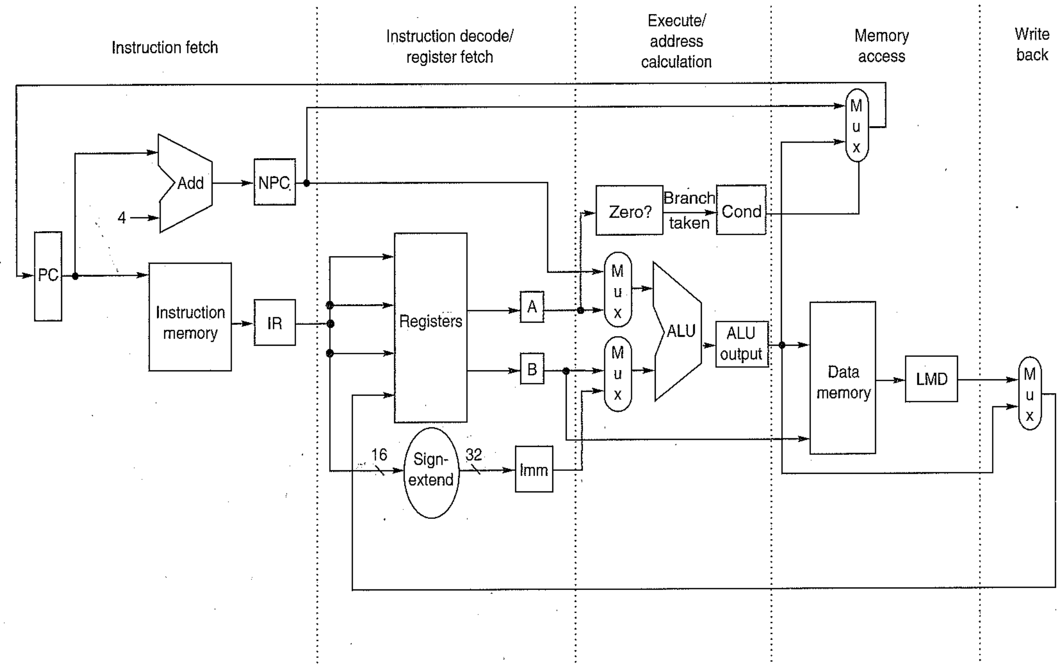

Figure A.17 The implementation of the MIPS data path allows every instruction to be executed in 4 or 5 clock cycles. Although the PC is shown in the portion of the data path that is used in instruction fetch and the registers are shown in the portion of the data path that is used in instruction decode/register fetch, both of these functional units are read as well as written by an instruction. Although we show these functional units in the cycle corresponding to where they are read, the PC is written during the memory access clock cycle and the registers are written during the write-back clock cycle. In both cases, the writes in later pipe stages are indicated by the multiplexer output (in memory access or write back), which carries a value back to the PC or registers. These backward-flowing signals introduce much of the complexity of pipelining, since they indicate the possibility of hazards.

There are some hardware redundancies that could be eliminated in this multicycle implementation. For example, there are two ALUs: one to increment the PC and one used for effective address and ALU computation. Since they are not needed on the same clock cycle, we could merge them by adding additional multiplexers and sharing the same ALU. Likewise, instructions and data could be stored in the same memory, since the data and instruction accesses happen on different clock cycles.

Rather than optimize this simple implementation, we will leave the design as it is in Figure A.17, since this provides us with a better base for the pipelined implementation.

As an alternative to the multicycle design discussed in this section, we could also have implemented the CPU so that every instruction takes 1 long clock cycle. In such cases, the temporary registers would be deleted, since there would not be any communication across clock cycles within an instruction. Every instruction would execute in 1 long clock cycle, writing the result into the data memory, registers, or PC at the end of the clock cycle. The CPI would be one for such a processor. The clock cycle, however, would be roughly equal to five times the clock cycle of the multicycle processor, since every instruction would need to traverse all the functional units. Designers would never use this single-cycle implementation for two reasons. First, a single-cycle implementation would be very inefficient for most CPUs that have a reasonable variation among the amount of work, and hence in the clock cycle time, needed for different instructions. Second, a single-cycle implementation requires the duplication of functional units that could be shared in a multicycle implementation. Nonetheless, this single-cycle data path allows us to illustrate how pipelining can improve the clock cycle time, as opposed to the CPI, of a processor.

A Basic Pipeline for MIPS

As before, we can pipeline the data path of Figure A.17 with almost no changes by starting a new instruction on each clock cycle. Because every pipe stage is active on every clock cycle, all operations in a pipe stage must complete in 1 clock cycle and any combination of operations must be able to occur at once. Furthermore, pipelining the data path requires that values passed from one pipe stage to the next must be placed in registers. Figure A.18 shows the MIPS pipeline with the appropriate registers, called *pipeline registers* or *pipeline latches,* between each pipeline stage. The registers are labeled with the names of the stages they connect. Figure A.18 is drawn so that connections through the pipeline registers from one stage to another are clear.

All of the registers needed to hold values temporarily between clock cycles within one instruction are subsumed into these pipeline registers. The fields of the instruction register (IR), which is part of the IF/ID register, are labeled when they are used to supply register names. The pipeline registers carry both data and control from one pipeline stage to the next. Any value needed on a later pipeline stage must be placed in such a register and copied from one pipeline register to