branch being mispredicted, but it does not cause an error in execution. Misprediction only introduces a small delay in execution time. An alternative approach is to store the history bits as a tag associated with branch instructions in the instruction cache. We will see in Section 8.7 how this information is handled in the SPARC processor.

8.4 Influence on Instruction Sets

We have seen that some instructions are much better suited to pipelined execution than others. For example, instruction side effects can lead to undesirable data dependencies. In this section, we examine the relationship between pipelined execution and machine instruction features. We discuss two key aspects of machine instructions — addressing modes and condition code flags.

8.4.1 Addressing Modes

Addressing modes should provide the means for accessing a variety of data structures simply and efficiently. Useful addressing modes include index, indirect, autoincrement, and autodecrement. Many processors provide various combinations of these modes to increase the flexibility of their instruction sets. Complex addressing modes, such as those involving double indexing, are often encountered.

In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline. Two important considerations in this regard are the side effects of modes such as autoincrement and autodecrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers.

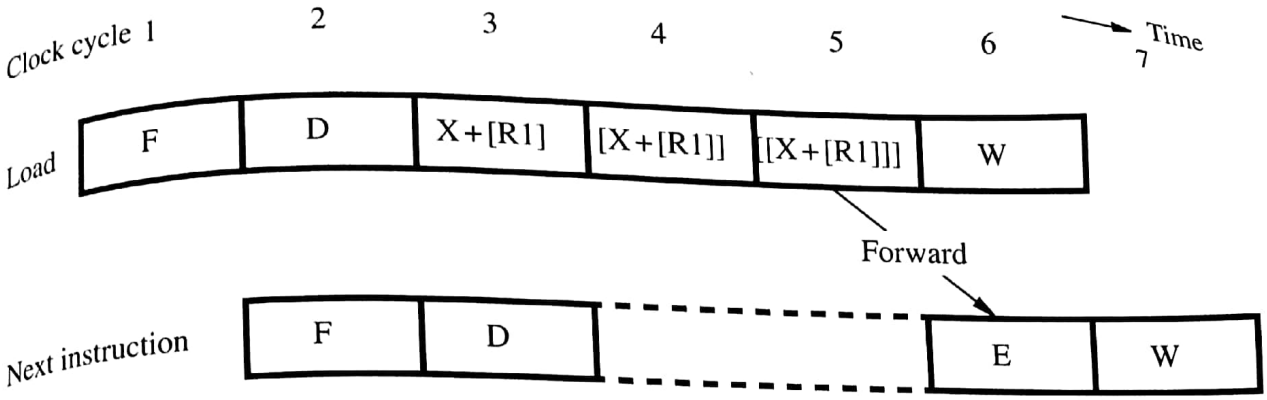
To compare various approaches, we assume a simple model for accessing operands in the memory. The load instruction Load X(R1),R2 takes five cycles to complete execution, as indicated in Figure 8.5. However, the instruction

```
Load    (R1),R2
```

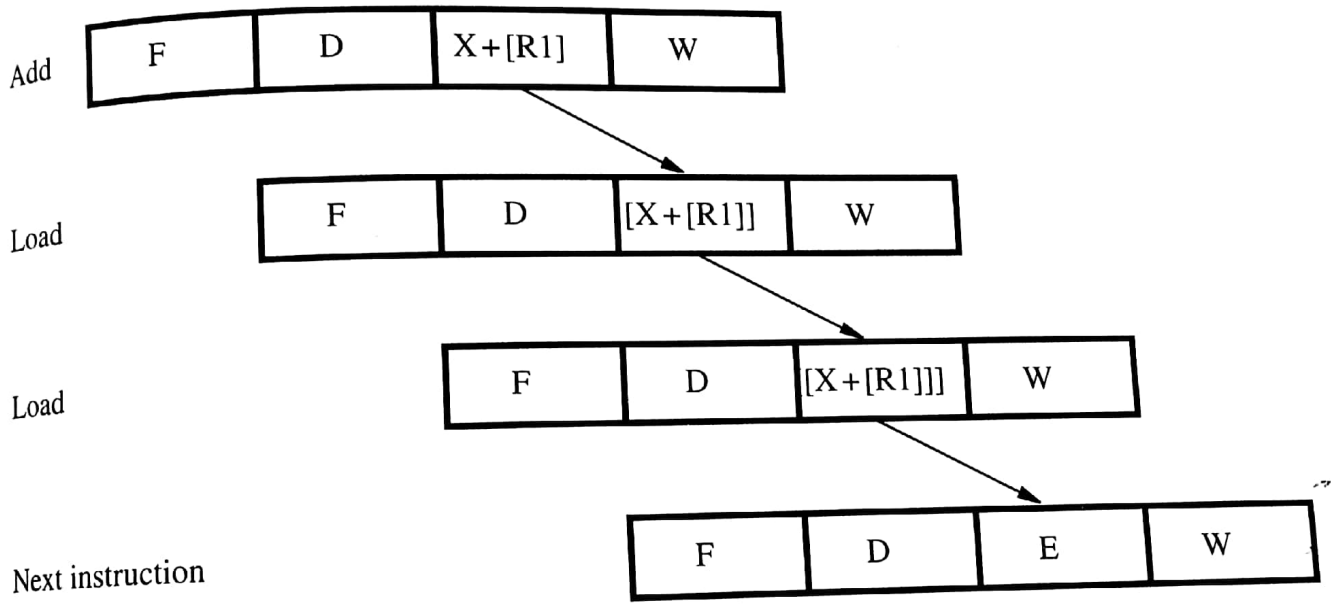
can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, the instruction

```
Load    (X(R1)),R2
```

may be executed as shown in Figure 8.16*a*, assuming that the index offset, X, is given in the instruction word. After computing the address in cycle 3, the processor needs to access memory twice — first to read location X+[R1] in clock cycle 4 and then to read location [X+[R1]] in cycle 5. If R2 is a source operand in the next instruction, that instruction would be stalled for three cycles, which can be reduced to two cycles with operand forwarding, as shown.



(a) Complex addressing mode



(b) Simple addressing mode

Figure 8.16 Equivalent operations using complex and simple addressing modes.

To implement the same Load operation using only simple addressing modes requires several instructions. For example, on a computer that allows three operand addresses, we can use

```
Add    #X,R1,R2
Load   (R2),R2
Load   (R2),R2
```

The Add instruction performs the operation R2 ← X+ [R1]. The two Load instructions fetch the address and then the operand from the memory. This sequence of instructions takes exactly the same number of clock cycles as the original, single Load instruction, as shown in Figure 8.16*b*.

This example indicates that, in a pipelined processor, complex addressing modes that involve several accesses to the memory do not necessarily lead to faster execution. The main advantage of such modes is that they reduce the number of instructions needed to perform a given task and thereby reduce the program space needed in the main memory. Their main disadvantage is that their long execution times cause the pipeline to stall, thus reducing its effectiveness. They require more complex hardware to decode and execute them. Also, they are not convenient for compilers to work with.

The instruction sets of modern processors are designed to take maximum advantage of pipelined hardware. Because complex addressing modes are not suitable for pipelined execution, they should be avoided. The addressing modes used in modern processors often have the following features:

- Access to an operand does not require more than one access to the memory.
- Only load and store instructions access memory operands.
- The addressing modes used do not have side effects.

Three basic addressing modes that have these features are register, register indirect, and index. The first two require no address computation. In the index mode, the address can be computed in one cycle, whether the index value is given in the instruction or in a register. Memory is accessed in the following cycle. None of these modes has any side effects, with one possible exception. Some architectures, such as ARM, allow the address computed in the index mode to be written back into the index register. This is a side effect that would not be allowed under the guidelines above. Note also that relative addressing can be used; this is a special case of indexed addressing in which the program counter is used as the index register.

The three features just listed were first emphasized as part of the concept of RISC processors. The SPARC processor architecture, which adheres to these guidelines, is presented in Section 8.7.

8.4.2 CONDITION CODES

In many processors, such as those described in Chapter 3, the condition code flags are stored in the processor status register. They are either set or cleared by many instructions, so that they can be tested by subsequent conditional branch instructions to change the flow of program execution. An optimizing compiler for a pipelined processor attempts to reorder instructions to avoid stalling the pipeline when branches or data dependencies between successive instructions occur. In doing so, the compiler must ensure that reordering does not cause a change in the outcome of a computation. The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

Consider the sequence of instructions in Figure 8.17*a*, and assume that the execution of the Compare and Branch=0 instructions proceeds as in Figure 8.14. The branch decision takes place in step E_2 rather than D_2 because it must await the result of the Compare instruction. The execution time of the Branch instruction can be reduced

Add	R1,R2
Compare	R3,R4
Branch=0	...

(a) A program fragment

Compare	R3,R4
Add	R1,R2
Branch=0	...

(b) Instructions reordered

Figure 8.17 Instruction reordering.

by interchanging the Add and Compare instructions, as shown in Figure 8.17*b*. This will delay the branch instruction by one cycle relative to the Compare instruction. As a result, at the time the Branch instruction is being decoded the result of the Compare instruction will be available and a correct branch decision will be made. There would be no need for branch prediction. However, interchanging the Add and Compare instructions can be done only if the Add instruction does not affect the condition codes.

These observations lead to two important conclusions about the way condition codes should be handled. First, to provide flexibility in reordering instructions, the condition-code flags should be affected by as few instructions as possible. Second, the compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not. An instruction set designed with pipelining in mind usually provides the desired flexibility. Figure 8.17*b* shows the instructions reordered assuming that the condition code flags are affected only when this is explicitly stated as part of the instruction OP code. The SPARC and ARM architectures provide this flexibility.

8.5 DATAPATH AND CONTROL CONSIDERATIONS

Organization of the internal datapath of a processor was introduced in Chapter 7. Consider the three-bus structure presented in Figure 7.8. To make it suitable for pipelined execution, it can be modified as shown in Figure 8.18 to support a 4-stage pipeline. The resources involved in stages F and E are shown in blue and those used in stages D and W in black. Operations in the data cache may happen during stage E or at a later stage, depending on the addressing mode and the implementation details. This section

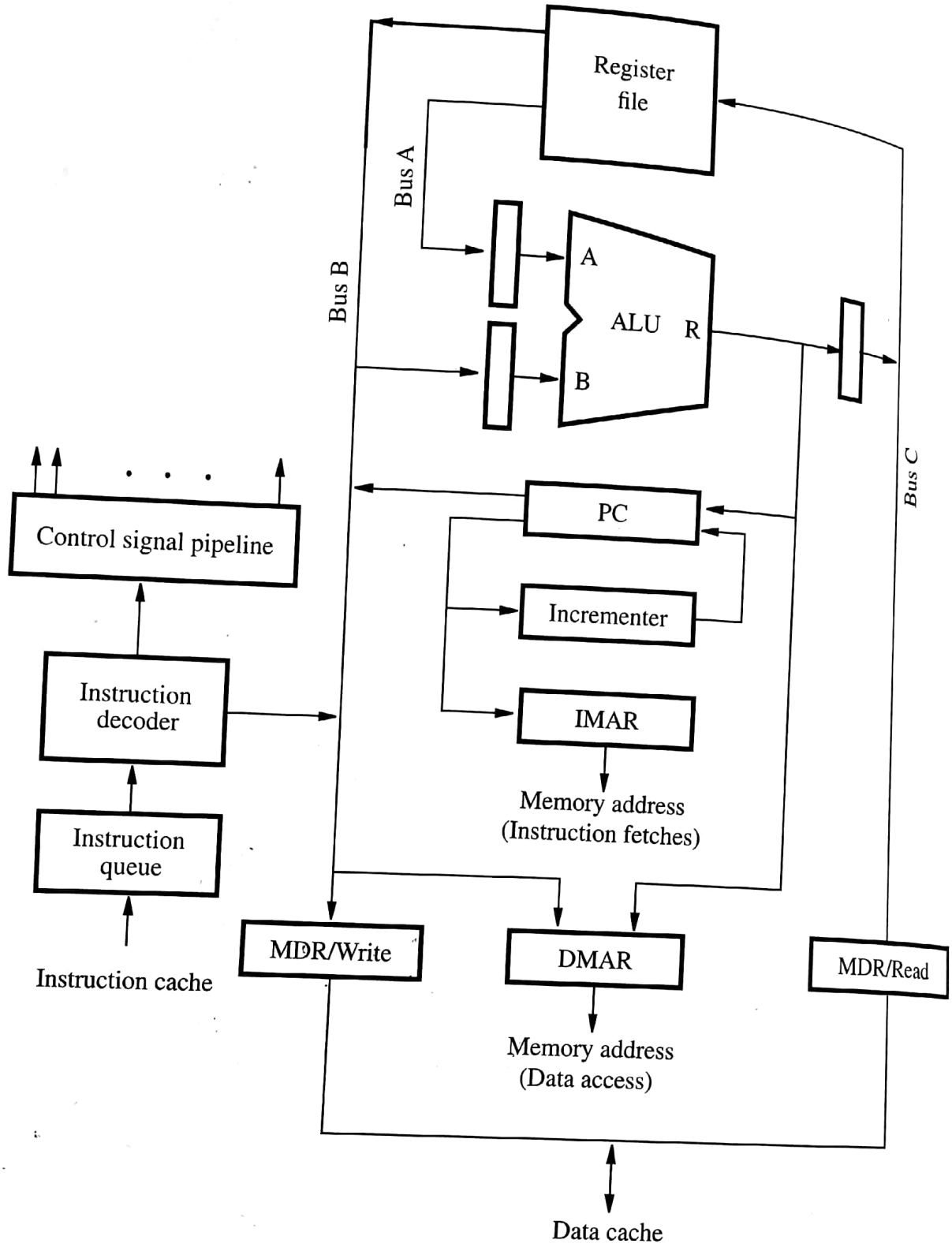


Figure 8.18 Datapath modified for pipelined execution with interstage buffers at the input and output of the ALU.

is shown in blue. Several important changes to Figure 7.8 should be noted:

1. There are separate instruction and data caches that use separate address and data connections to the processor. This requires two versions of the MAR register, IMAR for accessing the instruction cache and DMAR for accessing the data cache.
2. The PC is connected directly to the IMAR, so that the contents of the PC can be transferred to IMAR at the same time that an independent ALU operation is taking place.

3. The data address in DMAR can be obtained directly from the register file or from the ALU to support the register indirect and indexed addressing modes.
4. Separate MDR registers are provided for read and write operations. Data can be transferred directly between these registers and the register file during load and store operations without the need to pass through the ALU.
5. Buffer registers have been introduced at the inputs and output of the ALU. These are registers SRC1, SRC2, and RSLT in Figure 8.7. Forwarding connections are not included in Figure 8.18. They may be added if desired.
6. The instruction register has been replaced with an instruction queue, which is loaded from the instruction cache.
7. The output of the instruction decoder is connected to the control signal pipeline. The need for buffering control signals and passing them from one stage to the next along with the instruction is discussed in Section 8.1. This pipeline holds the control signals in buffers B2 and B3 in Figure 8.2*a*.

The following operations can be performed independently in the processor of Figure 8.18:

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two registers from the register file
- Writing into one register in the register file
- Performing an ALU operation

Because these operations do not use any shared resources, they can be performed simultaneously in any combination. The structure provides the flexibility required to implement the four-stage pipeline in Figure 8.2. For example, let I_1, I_2, I_3, and I_4 be a sequence of four instructions. As shown in Figure 8.2*a*, the following actions all happen during clock cycle 4:

- Write the result of instruction I_1 into the register file
- Read the operands of instruction I_2 from the register file
- Decode instruction I_3
- Fetch instruction I_4 and increment the PC.

8.6 SUPERSCALAR OPERATION

Pipelining makes it possible to execute instructions concurrently. Several instructions are present in the pipeline at the same time, but they are in different stages of their execution. While one instruction is performing an ALU operation, another instruction is being decoded and yet another is being fetched from the memory. Instructions enter the pipeline in strict program order. In the absence of hazards, one instruction enters the pipeline and one instruction completes execution in each clock cycle. This means that the maximum throughput of a pipelined processor is one instruction per clock cycle.