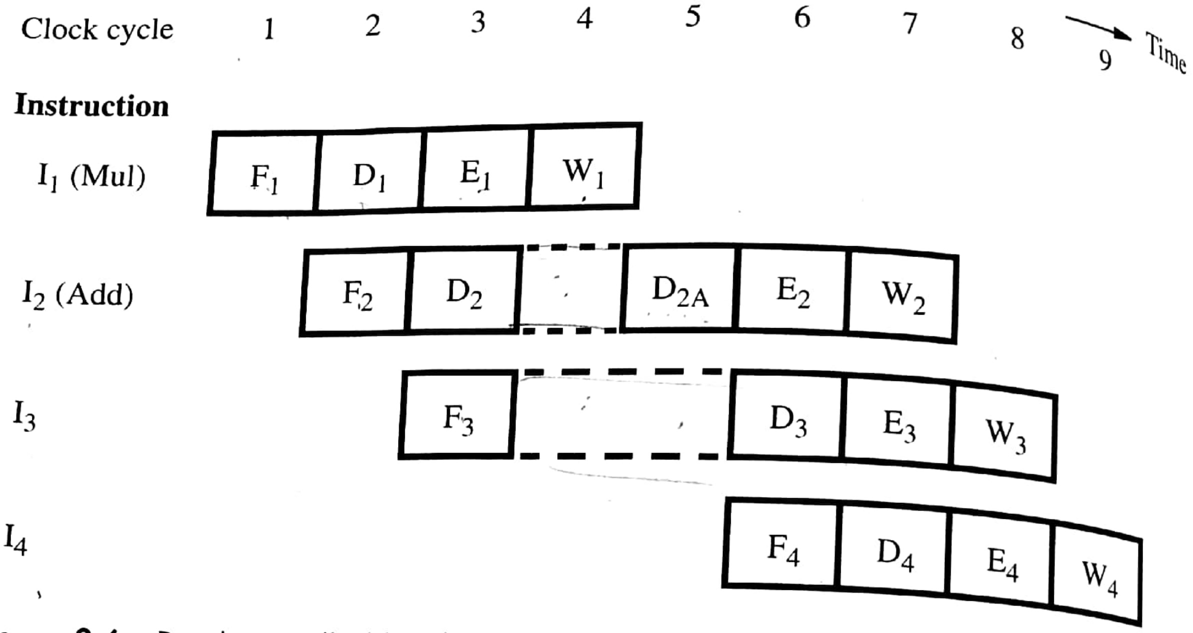


Figure 8.6 Pipeline stalled by data dependency between D_2 and W_1.

This example illustrates a basic constraint that must be enforced to guarantee correct results. When two operations depend on each other, they must be performed sequentially in the correct order. This rather obvious condition has far-reaching consequences. Understanding its implications is the key to understanding the variety of design alternatives and trade-offs encountered in pipelined computers.

Consider the pipeline in Figure 8.2. The data dependency just described arises when the destination of one instruction is used as a source in the next instruction. For example, the two instructions

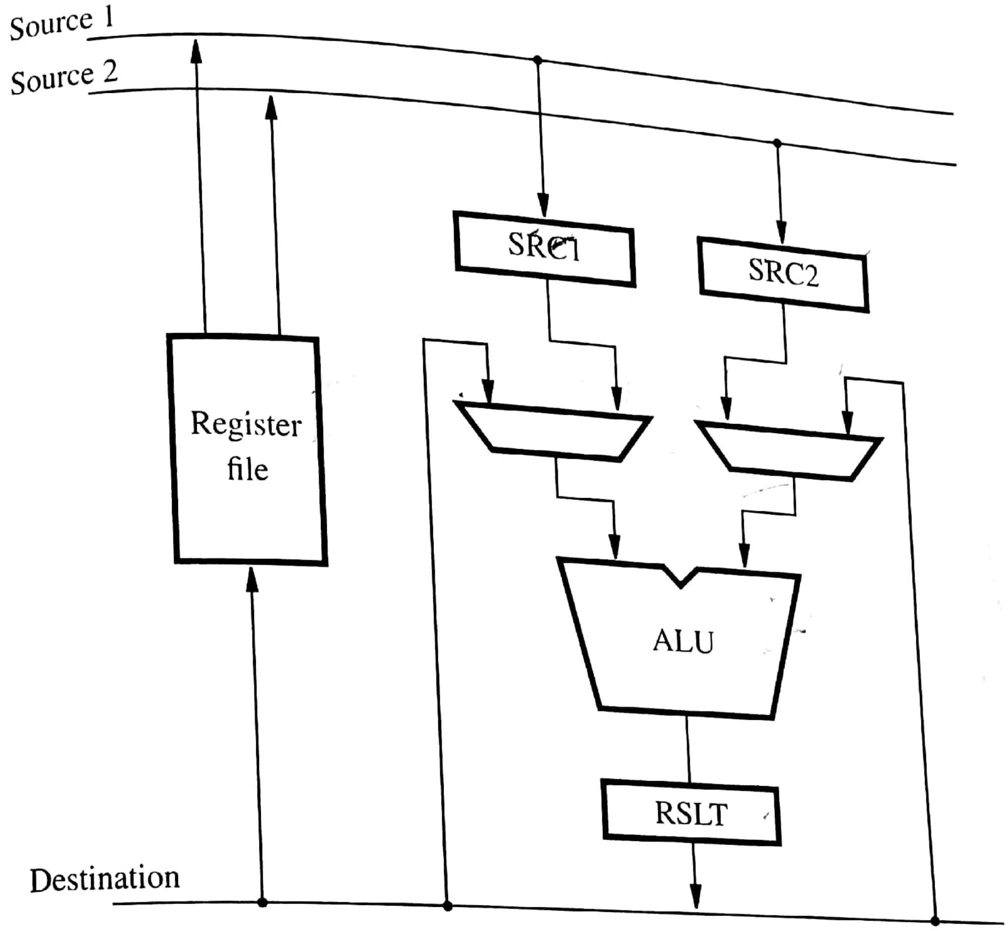
```
Mul  R2,R3,R4
Add  R5,R4,R6
```

give rise to a data dependency. The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction. Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in Figure 8.6. As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction cannot be completed until the W step of the multiply instruction has been completed. Completion of step D_2 must be delayed to clock cycle 5, and is shown as step D_{2A} in the figure. Instruction I_3 is fetched in cycle 3, but its decoding must be delayed because step D_3 cannot precede D_2. Hence, pipelined execution is stalled for two cycles.

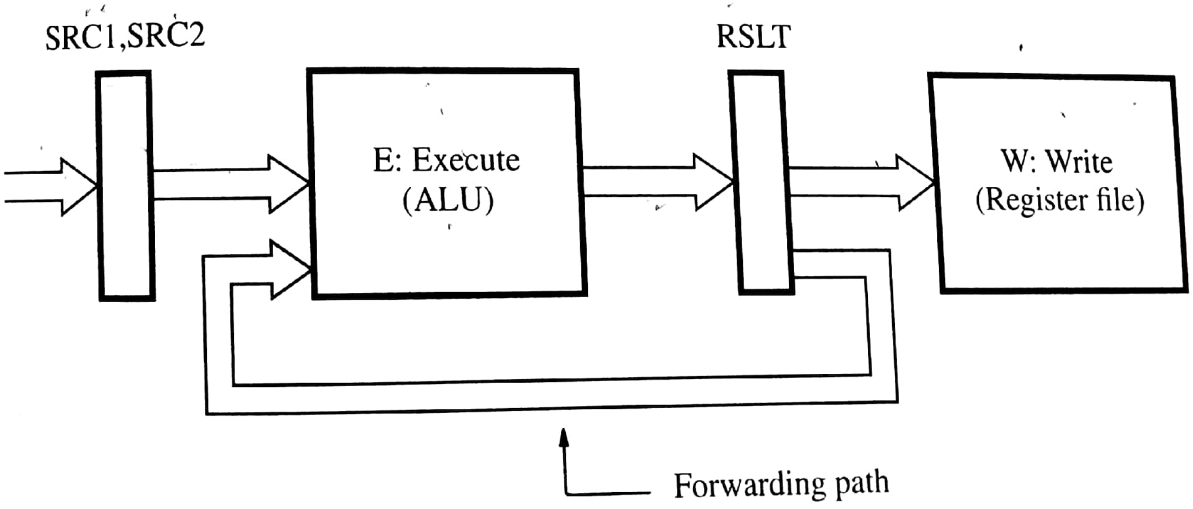
8.2.1 OPERAND FORWARDING

The data hazard just described arises because one instruction, instruction I_2 in Figure 8.6, is waiting for data to be written in the be reduced, or possibly eliminated, if we arrange for the result of instruction I_1 to be forwarded directly for use in step E_2.

Figure 8.7*a* shows a part of the processor datapath involving the ALU and the register file. This arrangement is similar to the three-bus structure in Figure 7.8, except that registers SRC1, SRC2, and RSLT have been added. These registers constitute the



(a) Datapath



(b) Position of the source and result registers in the processor pipeline

Figure 8.7 Operand forwarding in a pipelined processor.

interstage buffers needed for pipelined operation, as illustrated in Figure 8.7b. W reference to Figure 8.2*b*, registers SRC1 and SRC2 are part of buffer B2 and RSLT part of B3. The data forwarding mechanism is provided by the blue connection lines. T two multiplexers connected at the inputs to the ALU allow the data on the destinati bus to be selected instead of the contents of either the SRC1 or SRC2 register.

When the instructions in Figure 8.6 are executed in the datapath of Figure 8.7, th operations performed in each clock cycle are as follows. After decoding instructi I_2 and detecting the data dependency, a decision is made to use data forwarding. Th operand not involved in the dependency, register R2, is read and loaded in regist SRC1 in clock cycle 3. In the next clock cycle, the product produced by instruction I is available in register RSLT, and because of the forwarding connection, it can be use in step E_2. Hence, execution of I_2 proceeds without interruption.

8.2.2 HANDLING DATA HAZARDS IN SOFTWARE

In Figure 8.6, we assumed the data dependency is discovered by the hardware while the instruction is being decoded. The control hardware delays reading register R4 until cycle 5, thus introducing a 2-cycle stall unless operand forwarding is used. An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software. In this case, the compiler can introduce the two-cycle delay needed between instructions I_1 and I_2 by inserting NOP (No-operation) instructions, as follows:

```
I1:  Mul   R2,R3,R4
     NOP
     NOP
I2:  Add   R5,R4,R6
```

If the responsibility for detecting such dependencies is left entirely to the software, the compiler must insert the NOP instructions to obtain a correct result. This possibility illustrates the close link between the compiler and the hardware. A particular feature can be either implemented in hardware or left to the compiler. Leaving tasks such as inserting NOP instructions to the compiler leads to simpler hardware. Being aware of the need for a delay, the compiler can attempt to reorder instructions to perform useful tasks in the NOP slots, and thus achieve better performance. On the other hand, the insertion of NOP instructions leads to larger code size. Also, it is often the case that a given processor architecture has several hardware implementations, offering different features. NOP instructions inserted to satisfy the requirements of one implementation may not be needed and, hence, would lead to reduced performance on a different implementation.

8.2.3 SIDE EFFECTS

The data dependencies encountered in the preceding examples are explicit and easily detected because the register involved is named as the destination in instruction I_1 and as a source in I_2. Sometimes an instruction changes the contents of a register other