

**Figure 14.8** Data Flow, Interrupt Cycle

transferred to the MBR to be written into memory. The special memory location reserved for this purpose is loaded into the MAR from the control unit. It might, for example, be a stack pointer. The PC is loaded with the address of the interrupt routine. As a result, the next instruction cycle will begin by fetching the appropriate instruction.

## 14.4 INSTRUCTION PIPELINING

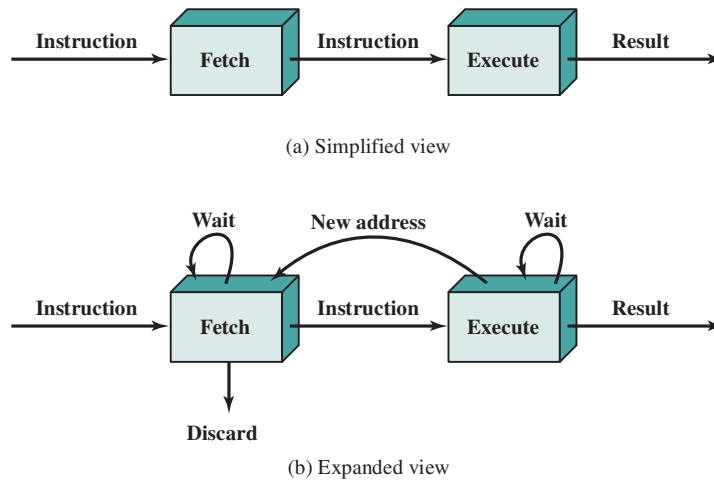
As computer systems evolve, greater performance can be achieved by taking advantage of improvements in technology, such as faster circuitry. In addition, organizational enhancements to the processor can improve performance. We have already seen some examples of this, such as the use of multiple registers rather than a single accumulator, and the use of a cache memory. Another organizational approach, which is quite common, is instruction pipelining.

### Pipelining Strategy

Instruction pipelining is similar to the use of an assembly line in a manufacturing plant. An assembly line takes advantage of the fact that a product goes through various stages of production. By laying the production process out in an assembly line, products at various stages can be worked on simultaneously. This process is also referred to as *pipelining*, because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply this concept to instruction execution, we must recognize that, in fact, an instruction has a number of stages. Figures 14.5, for example, breaks the instruction cycle up into 10 tasks, which occur in sequence. Clearly, there should be some opportunity for pipelining.

As a simple approach, consider subdividing instruction processing into two stages: fetch instruction and execute instruction. There are times during the execution of an instruction when main memory is not being accessed. This time could be used to fetch the next instruction in parallel with the execution of the current



**Figure 14.9** Two-Stage Instruction Pipeline

one. Figure 14.9a depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called instruction prefetch or *fetch overlap*. Note that this approach, which involves instruction buffering, requires more registers. In general, pipelining requires registers to store data between stages.

It should be clear that this process will speed up instruction execution. If the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 14.9b), we will see that this doubling of execution rate is unlikely for two reasons:

1. The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to wait for some time before it can empty its buffer.
2. A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

Guessing can reduce the time loss from the second reason. A simple rule is the following: When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost. If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

While these factors reduce the potential effectiveness of the two-stage pipeline, some speedup occurs. To gain further speedup, the pipeline must have more stages. **Let us consider the following decomposition of the instruction processing.**

- **Fetch instruction (FI):** Read the next expected instruction into a buffer.
- **Decode instruction (DI):** Determine the opcode and the operand specifiers.
- **Calculate operands (CO):** Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
- **Fetch operands (FO):** Fetch each operand from memory. Operands in registers need not be fetched.
- **Execute instruction (EI):** Perform the indicated operation and store the result, if any, in the specified destination operand location.
- **Write operand (WO):** Store the result in memory.

With this decomposition, the various stages will be of more nearly equal duration. For the sake of illustration, let us assume equal duration. Using this assumption, Figure 14.10 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

Several comments are in order: The diagram assumes that each instruction goes through all six stages of the pipeline. This will not always be the case. For example, a load instruction does not need the WO stage. However, to simplify the pipeline hardware, the timing is set up assuming that each instruction requires all six stages. Also, the diagram assumes that all of the stages can be performed in parallel. In particular, it is assumed that there are no memory conflicts. For example, the FI, FO, and WO stages involve a memory access. The diagram implies that all these accesses can occur simultaneously. Most memory systems will not permit that. However, the desired value may be in cache, or the FO or WO stage may be null. Thus, much of the time, memory conflicts will not slow down the pipeline.

Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

**Figure 14.10** Timing Diagram for Instruction Pipeline Operation

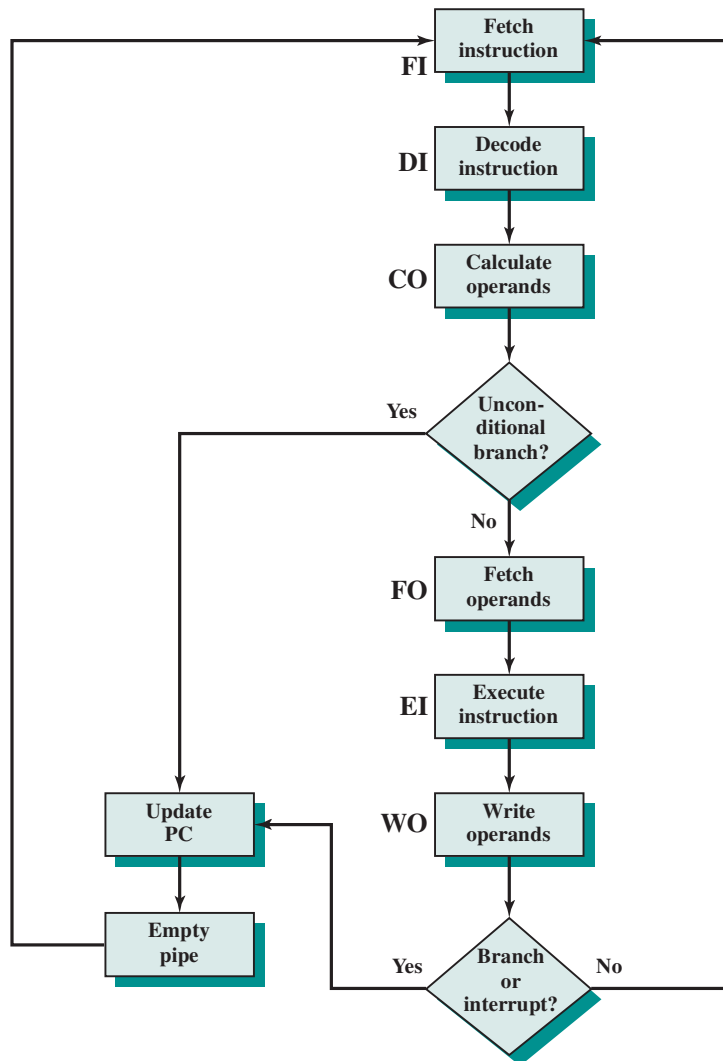
	Time →						← Branch penalty							
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO							
Instruction 5					FI	DI	CO							
Instruction 6						FI	DI							
Instruction 7							FI							
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16									FI	DI	CO	FO	EI	WO

**Figure 14.11** The Effect of a Conditional Branch on Instruction Pipeline Operation

Several other factors serve to limit the performance enhancement. If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages, as discussed before for the two-stage pipeline. Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches. A similar unpredictable event is an interrupt. Figure 14.11 illustrates the effects of the conditional branch, using the same program as Figure 14.10. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds. In Figure 14.10, the branch is not taken, and we get the full performance benefit of the enhancement. In Figure 14.11, the branch is taken. This is not determined until the end of time unit 7. At this point, the pipeline must be cleared of instructions that are not useful. During time unit 8, instruction 15 enters the pipeline. No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch. Figure 14.12 indicates the logic needed for pipelining to account for branches and interrupts.

Other problems arise that did not appear in our simple two-stage organization. The CO stage may depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline. Other such register and memory conflicts could occur. The system must contain logic to account for this type of conflict.

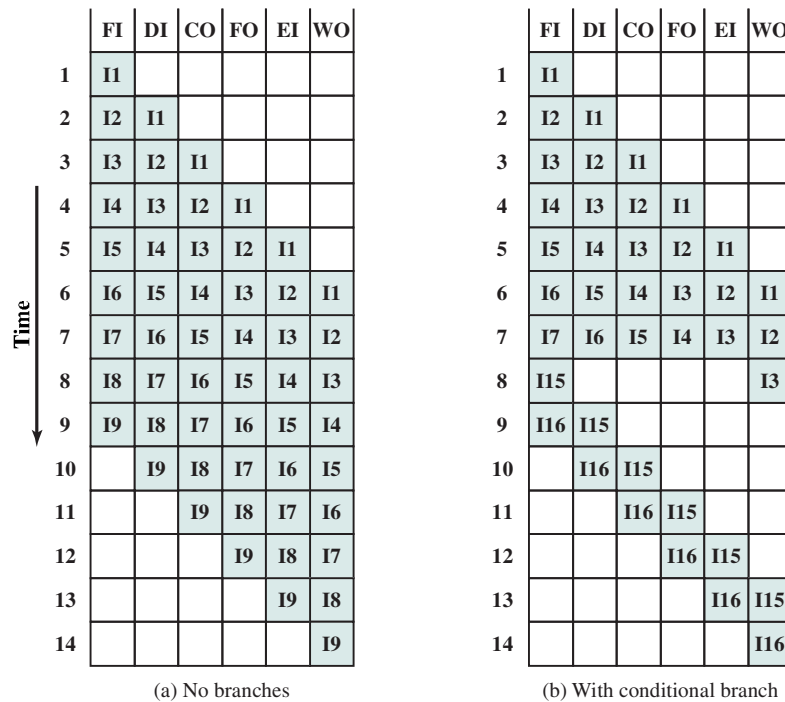
To clarify pipeline operation, it might be useful to look at an alternative depiction. Figures 14.10 and 14.11 show the progression of time horizontally across the figures, with each row showing the progress of an individual instruction. Figure 14.13 shows same sequence of events, with time progressing vertically down the figure,



**Figure 14.12** Six-Stage CPU Instruction Pipeline

and each row showing the state of the pipeline at a given point in time. In Figure 14.13a (which corresponds to Figure 14.10), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed. In Figure 14.13b, (which corresponds to Figure 14.11), the pipeline is full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15. At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.

From the preceding discussion, it might appear that the greater the number of stages in the pipeline, the faster the execution rate. Some of the IBM S/360 designers



**Figure 14.13** An Alternative Pipeline Depiction

pointed out two factors that frustrate this seemingly simple pattern for high-performance design [ANDE67a], and they remain elements that designer must still consider:

1. At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a single instruction. This is significant when sequential instructions are logically dependent, either through heavy use of branching or through memory access dependencies.
2. The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a situation where the logic controlling the gating between stages is more complex than the stages being controlled.

Another consideration is latching delay: It takes time for pipeline buffers to operate and this adds to instruction cycle time.

Instruction pipelining is a powerful technique for enhancing performance but requires careful design to achieve optimum results with reasonable complexity.

### Pipeline Performance

In this subsection, we develop some simple measures of pipeline performance and relative speedup (based on a discussion in [HWAN93]). The cycle time  $\tau$  of an **instruction pipeline** is the time needed to advance a set of instructions one stage

through the pipeline; each column in Figures 14.10 and 14.11 represents one cycle time. The cycle time can be determined as

$$\tau = \max_i[\tau_i] + d = \tau_m + d \quad 1 \leq i \leq k$$

where

$\tau_i$  = time delay of the circuitry in the  $i$ th stage of the pipeline

$\tau_m$  = maximum stage delay (delay through stage which experiences the largest delay)

$k$  = number of stages in the instruction pipeline

$d$  = time delay of a latch, needed to advance signals and data from one stage to the next

In general, the time delay  $d$  is equivalent to a clock pulse and  $\tau_m \gg d$ . Now suppose that  $n$  instructions are processed, with no branches. Let  $T_{k,n}$  be the total time required for a pipeline with  $k$  stages to execute  $n$  instructions. Then

$$T_{k,n} = [k + (n - 1)]\tau \quad (14.1)$$

A total of  $k$  cycles are required to complete the execution of the first instruction, and the remaining  $n - 1$  instructions require  $n - 1$  cycles.<sup>2</sup> This equation is easily verified from Figures 14.10. The ninth instruction completes at time cycle 14:

$$14 = [6 + (9 - 1)]$$

Now consider a processor with equivalent functions but no pipeline, and assume that the instruction cycle time is  $k\tau$ . The speedup factor for the instruction pipeline compared to execution without the pipeline is defined as

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)} \quad (14.2)$$

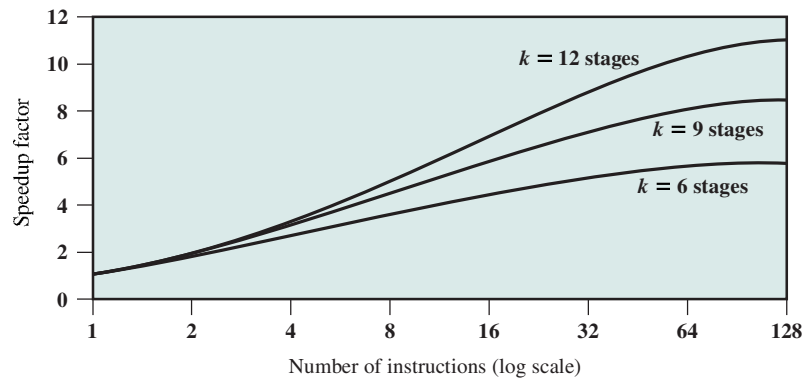
Figure 14.14a plots the speedup factor as a function of the number of instructions that are executed without a branch. As might be expected, at the limit ( $n \rightarrow \infty$ ), we have a  $k$ -fold speedup. Figure 14.14b shows the speedup factor as a function of the number of stages in the instruction pipeline.<sup>3</sup> In this case, the speedup factor approaches the number of instructions that can be fed into the pipeline without branches. Thus, the larger the number of pipeline stages, the greater the potential for speedup. However, as a practical matter, the potential gains of additional pipeline stages are countered by increases in cost, delays between stages, and the fact that branches will be encountered requiring the flushing of the pipeline.

### Pipeline Hazards

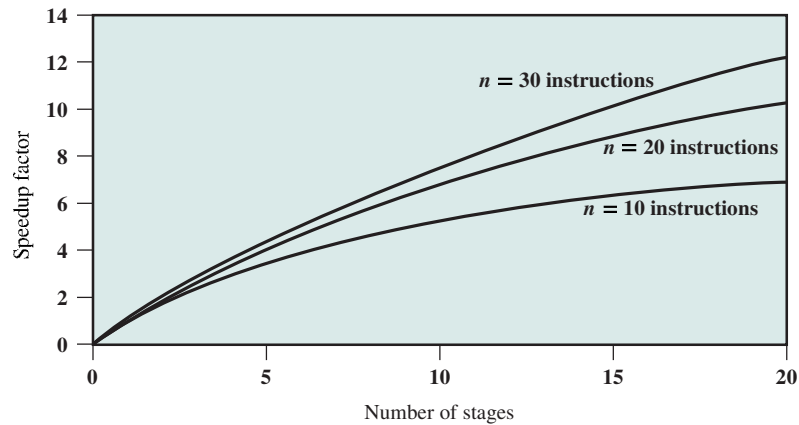
In the previous subsection, we mentioned some of the situations that can result in less than optimal pipeline performance. In this subsection, we examine this issue in

<sup>2</sup>We are being a bit sloppy here. The cycle time will only equal the maximum value of  $\tau$  when all the stages are full. At the beginning, the cycle time may be less for the first one or few cycles.

<sup>3</sup>Note that the  $x$ -axis is logarithmic in Figure 14.14a and linear in Figure 14.14b.



(a)



(b)

**Figure 14.14** Speedup Factors with Instruction Pipelining

a more systematic way. Chapter 16 revisits this issue, in more detail, after we have introduced the complexities found in superscalar pipeline organizations.

A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a *pipeline bubble*. There are three types of hazards: resource, data, and control.

**RESOURCE HAZARDS** A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline. A resource hazard is sometime referred to as a *structural hazard*.

Let us consider a simple example of a resource hazard. Assume a simplified five-stage pipeline, in which each stage takes one clock cycle. Figure 14.15a shows the ideal case, in which a new instruction enters the pipeline each clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. Further, ignore the cache. In this



		Clock cycle								
		1	2	3	4	5	6	7	8	9
Instruction	I1	FI	DI	FO	EI	WO				
	I2		FI	DI	FO	EI	WO			
	I3			FI	DI	FO	EI	WO		
	I4				FI	DI	FO	EI	WO	

(a) Five-stage pipeline, ideal case

		Clock cycle								
		1	2	3	4	5	6	7	8	9
Instruction	I1	FI	DI	FO	EI	WO				
	I2		FI	DI	FO	EI	WO			
	I3			Idle	FI	DI	FO	EI	WO	
	I4					FI	DI	FO	EI	WO

(b) I1 source operand in memory

**Figure 14.15** Example of Resource Hazard

case, an operand read to or write from memory cannot be performed in parallel with an instruction fetch. This is illustrated in Figure 14.15b, which assumes that the source operand for instruction I1 is in memory, rather than a register. Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3. The figure assumes that all other operands are in registers.

Another example of a resource conflict is a situation in which multiple instructions are ready to enter the execute instruction phase and there is a single ALU. One solutions to such resource hazards is to increase available resources, such as having multiple ports into main memory and multiple ALU units.



**Reservation Table Analyzer**

One approach to analyzing resource conflicts and aiding in the design of pipelines is the reservation table. We examine reservation tables in Appendix I.

**DATA HAZARDS** A data hazard occurs when there is a conflict in the access of an operand location. In general terms, we can state the hazard in this form: Two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence,

no problem occurs. However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution. In other words, the program produces an incorrect result because of the use of pipelining.

As an example, consider the following x86 machine instruction sequence:

```
ADD EAX,    EBX /* EAX = EAX + EBX
SUB ECX,    EAX /* ECX = ECX - EAX
```

The first instruction adds the contents of the 32-bit registers EAX and EBX and stores the result in EAX. The second instruction subtracts the contents of EAX from ECX and stores the result in ECX. Figure 14.16 shows the pipeline behavior. The ADD instruction does not update register EAX until the end of stage 5, which occurs at clock cycle 5. But the SUB instruction needs that value at the beginning of its stage 2, which occurs at clock cycle 4. To maintain correct operation, the pipeline must stall for two clock cycles. Thus, in the absence of special hardware and specific avoidance algorithms, such a data hazard results in inefficient pipeline usage.

There are three types of data hazards;

- **Read after write (RAW), or true dependency:** An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the read takes place before the write operation is complete.
- **Write after read (WAR), or antidependency:** An instruction reads a register or memory location and a succeeding instruction writes to the location. A hazard occurs if the write operation completes before the read operation takes place.
- **Write after write (WAW), or output dependency:** Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence.

The example of Figure 14.16 is a RAW hazard. The other two hazards are best discussed in the context of superscalar organization, discussed in Chapter 16.

**CONTROL HAZARDS** A control hazard, also known as a *branch hazard*, occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded. We discuss approaches to dealing with control hazards next.

	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
ADD EAX, EBX	FI	DI	FO	EI	WO					
SUB ECX, EAX		FI	DI	Idle		FO	EI	WO		
13			FI			DI	FO	EI	WO	
14						FI	DI	FO	EI	WO

**Figure 14.16** Example of Data Hazard

### Dealing with Branches

One of the major problems in designing an instruction pipeline is assuring a steady flow of instructions to the initial stages of the pipeline. The primary impediment, as we have seen, is the conditional branch instruction. Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

A variety of approaches have been taken for dealing with conditional branches:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Branch prediction
- Delayed branch

**MULTIPLE STREAMS** A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice. A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams. There are two problems with this approach:

- With multiple pipelines there are contention delays for access to the registers and to memory.
- Additional branch instructions may enter the pipeline (either stream) before the original branch decision is resolved. Each such instruction needs an additional stream.

Despite these drawbacks, this strategy can improve performance. Examples of machines with two or more pipeline streams are the IBM 370/168 and the IBM 3033.

**PREFETCH BRANCH TARGET** When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

The IBM 360/91 uses this approach.

**LOOP BUFFER** A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the  $n$  most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer. The loop buffer has three benefits:

1. With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address. Thus, instructions fetched in sequence will be available without the usual memory access time.
2. If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer. This is useful for the rather common occurrence of IF-THEN and IF-THEN-ELSE sequences.