

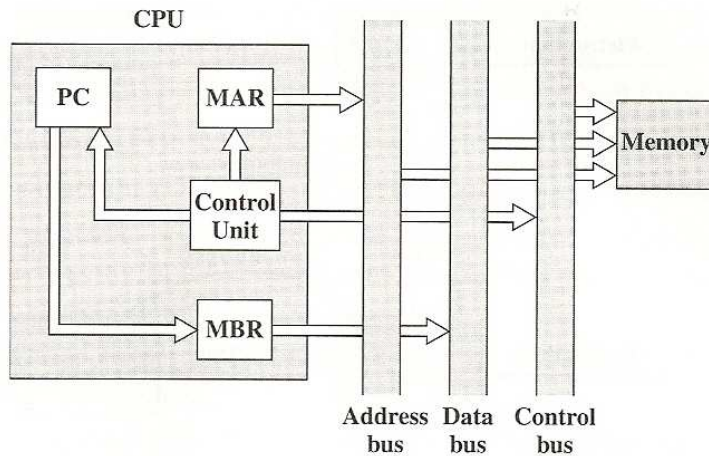


Figure 12.8 Data Flow, Interrupt Cycle

The fetch and indirect cycles are simple and predictable. The *execute cycle* takes many forms; the form depends on which of the various machine instructions is in the IR. This cycle may involve transferring data among registers, read or write from memory or I/O, and/or the invocation of the ALU.

Like the fetch and indirect cycles, the *interrupt cycle* is simple and predictable (Figure 12.8). The current contents of the PC must be saved so that the processor can resume normal activity after the interrupt. Thus, the contents of the PC are 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.

12.4 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. Figure 12.5, for example, breaks the

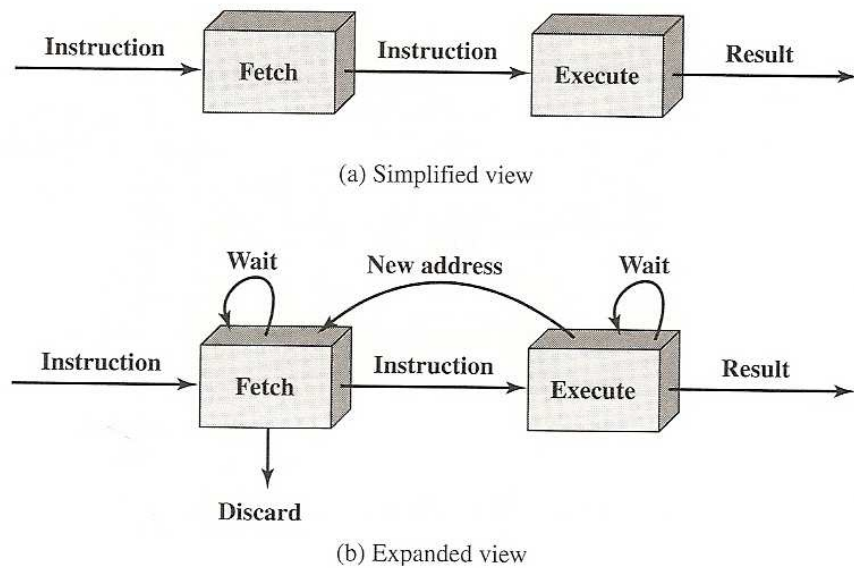


Figure 12.9 Two-Stage Instruction Pipeline

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 one. Figure 12.9a 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*.

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 12.9b), 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 12.10 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

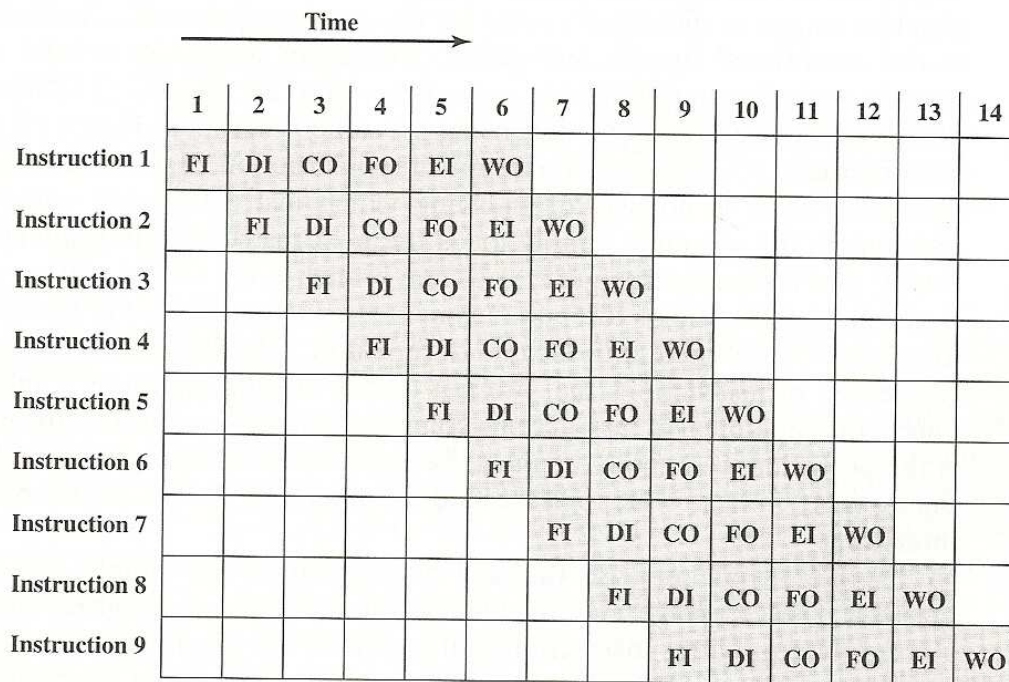


Figure 12.10 Timing Diagram for Instruction Pipeline Operation

	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 12.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

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.

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 12.11 illustrates the effects of the conditional branch, using the same program as Figure 12.10. 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 12.10, the branch is not taken, and we get the full performance benefit of the enhancement. In Figure 12.11, 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 12.12 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.

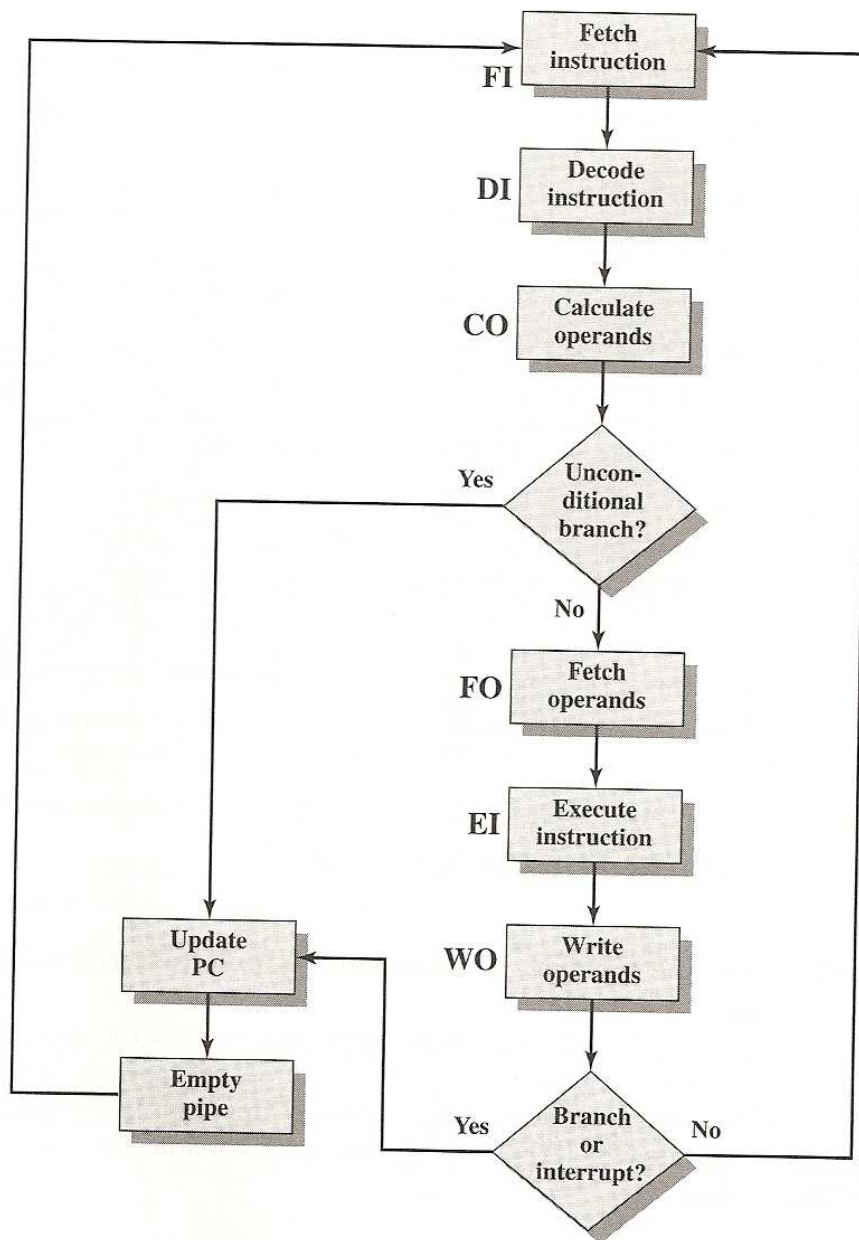


Figure 12.12 Six-Stage CPU Instruction Pipeline

To clarify pipeline operation, it might be useful to look at an alternative depiction. Figures 12.10 and 12.11 show the progression of time horizontally across the figures, with each row showing the progress of an individual instruction. Figure 12.13 shows same sequence of events, with time progressing vertically down the figure, and each row showing the state of the pipeline at a given point in time. In Figure 12.13a (which corresponds to Figure 12.10), 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 12.13b, (which corresponds to Figure 12.11), 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

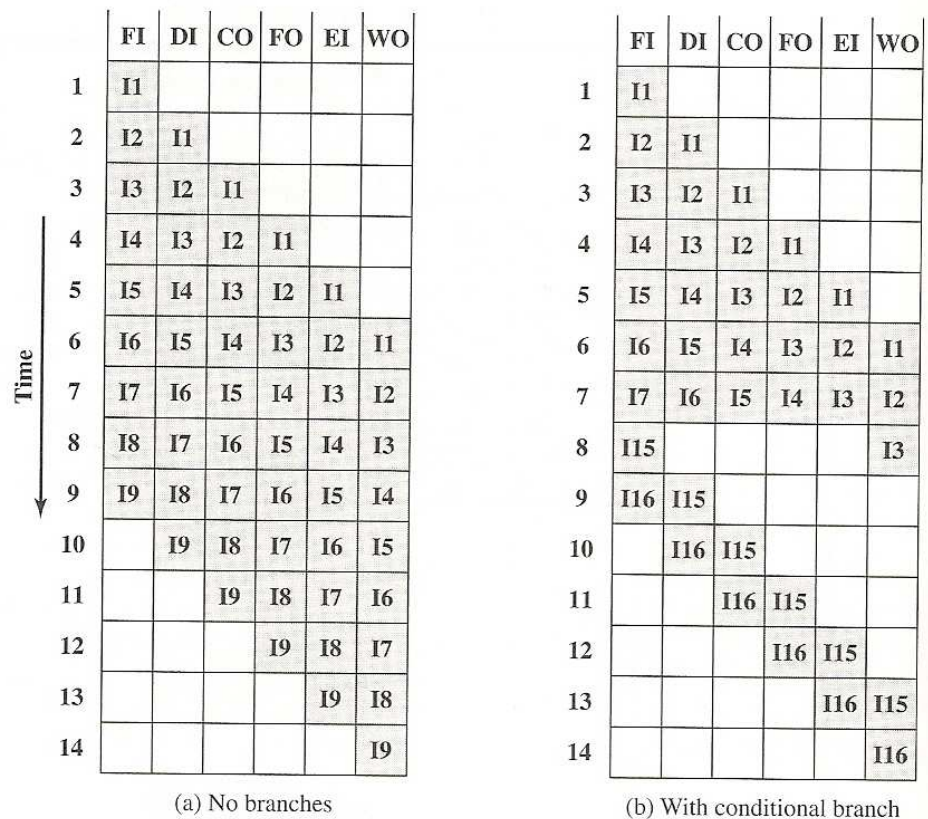


Figure 12.13 An Alternative Pipeline Depiction

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 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.

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 τ of an instruction pipeline is the time needed to advance a set of instructions one stage through the pipeline; each column in Figures 12.10 and 12.11 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

τ_i = time delay of the circuitry in the i th stage of the pipeline

τ_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

$$T_{k,n} = [k + (n - 1)]\tau \quad (12.1)$$

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.² This equation is easily verified from Figure 12.10. The ninth instruction completes at time cycle 14:

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

Now consider an 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

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)} \quad (12.2)$$

Figure 12.14a 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 12.14b shows the speedup factor as a function of the number of stages in the instruction pipeline.³ 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

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

³Note that the x -axis is logarithmic in Figure 12.14a and linear in Figure 12.14b.

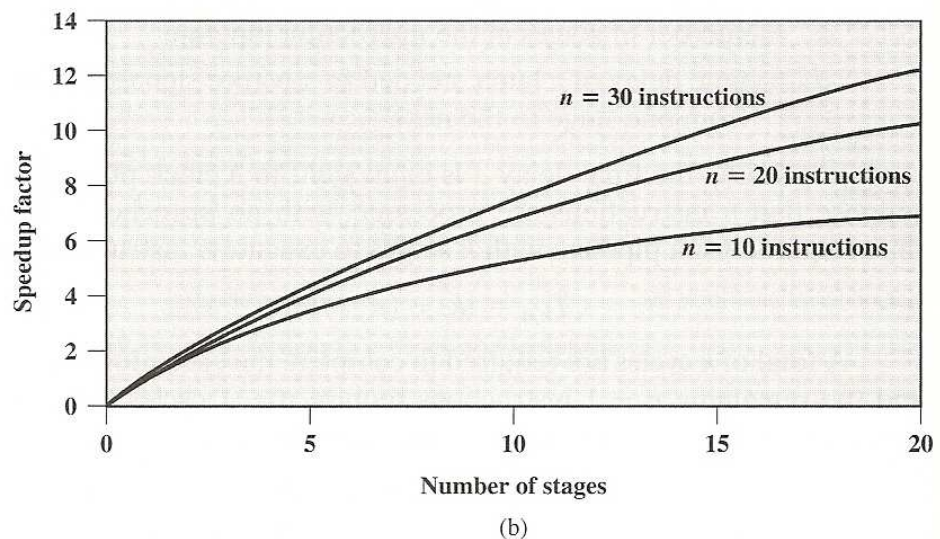
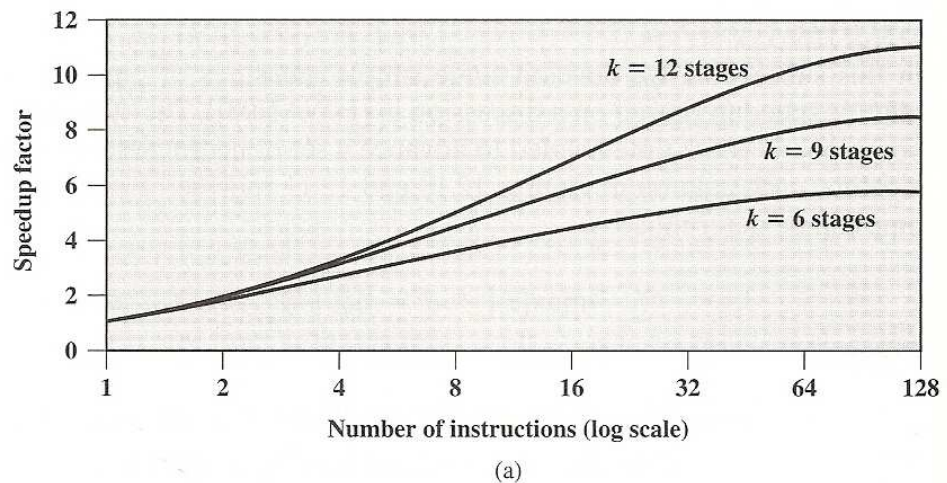


Figure 12.14 Speedup Factors with Instruction Pipelining

stages are countered by increases in cost, delays between stages, and the fact that branches will be encountered requiring the flushing of the pipeline.

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.
3. This strategy is particularly well suited to dealing with loops, or iterations; hence the name *loop buffer*. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

The loop buffer is similar in principle to a cache dedicated to instructions. The differences are that the loop buffer only retains instructions in sequence and is much smaller in size and hence lower in cost.

Figure 12.15 gives an example of a loop buffer. If the buffer contains 256 bytes, and byte addressing is used, then the least significant 8 bits are used to index the buffer. The remaining most significant bits are checked to determine if the branch target lies within the environment captured by the buffer.

Among the machines using a loop buffer are some of the CDC machines (Star-100, 6600, 7600) and the CRAY-1. A specialized form of loop buffer is available on the Motorola 68010, for executing a three-instruction loop involving the

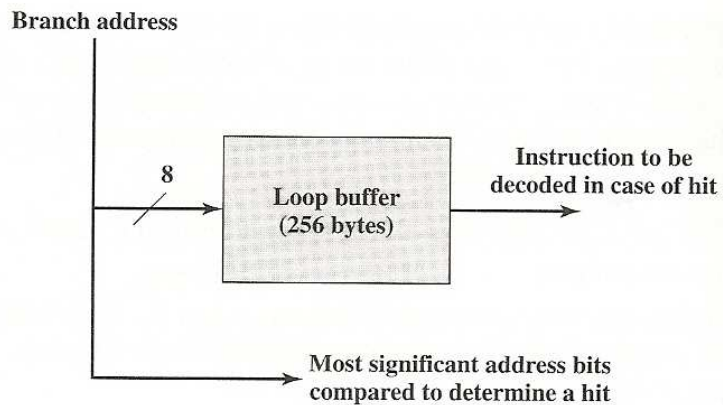


Figure 12.15 Loop Buffer

DBcc (decrement and branch on condition) instruction (see Problem 12.14). A three-word buffer is maintained, and the processor executes these instructions repeatedly until the loop condition is satisfied.

Branch Prediction Various techniques can be used to predict whether a branch will be taken. Among the more common are the following:

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table

The first three approaches are static: They do not depend on the execution history up to the time of the conditional branch instruction. The latter two approaches are dynamic: They depend on the execution history.

The first two approaches are the simplest. These either always assume that the branch will not be taken and continue to fetch instructions in sequence, or they always assume that the branch will be taken and always fetch from the branch target. The 68020 and the VAX 11/780 use the predict-never-taken approach. The VAX 11/780 also includes a feature to minimize the effect of a wrong decision. If the fetch of the instruction after the branch will cause a page fault or protection violation, the processor halts its prefetching until it is sure that the instruction should be fetched.

Studies analyzing program behavior have shown that conditional branches are taken more than 50% of the time [LILJ88], and so if the cost of prefetching from either path is the same, then always prefetching from the branch target address should give better performance than always prefetching from the sequential path. However, in a paged machine, prefetching the branch target is more likely to cause a page fault than prefetching the next instruction in sequence, and so this performance penalty should be taken into account. An avoidance mechanism may be employed to reduce this penalty.

The final static approach makes the decision based on the opcode of the branch instruction. The processor assumes that the branch will be taken for certain branch opcodes and not for others. [LILJ88] reports success rates of greater than 75% with this strategy.

Dynamic branch strategies attempt to improve the accuracy of prediction by recording the history of conditional branch instructions in a program. For example, one or more bits can be associated with each conditional branch instruction that reflect the recent history of the instruction. These bits are referred to as a taken/not taken switch that directs the processor to make a particular decision the next time the instruction is encountered. Typically, these history bits are not associated with the instruction in main memory. Rather, they are kept in temporary high-speed storage. One possibility is to associate these bits with any conditional branch instruction that is in a cache. When the instruction is replaced in the cache, its history is lost. Another possibility is to maintain a small table for recently executed branch instructions with one or more history bits in each entry. The processor could access the table associatively, like a cache, or by using the low-order bits of the branch instruction's address.

With a single bit, all that can be recorded is whether the last execution of this instruction resulted in a branch or not. A shortcoming of using a single bit appears in the case of a conditional branch instruction that is almost always taken, such as a loop instruction. With only one bit of history, an error in prediction will occur twice for each use of the loop: once on entering the loop, and once on exiting.

If two bits are used, they can be used to record the result of the last two instances of the execution of the associated instruction, or to record a state in some other fashion. Figure 12.16 shows a typical approach (see Problem 12.13 for other

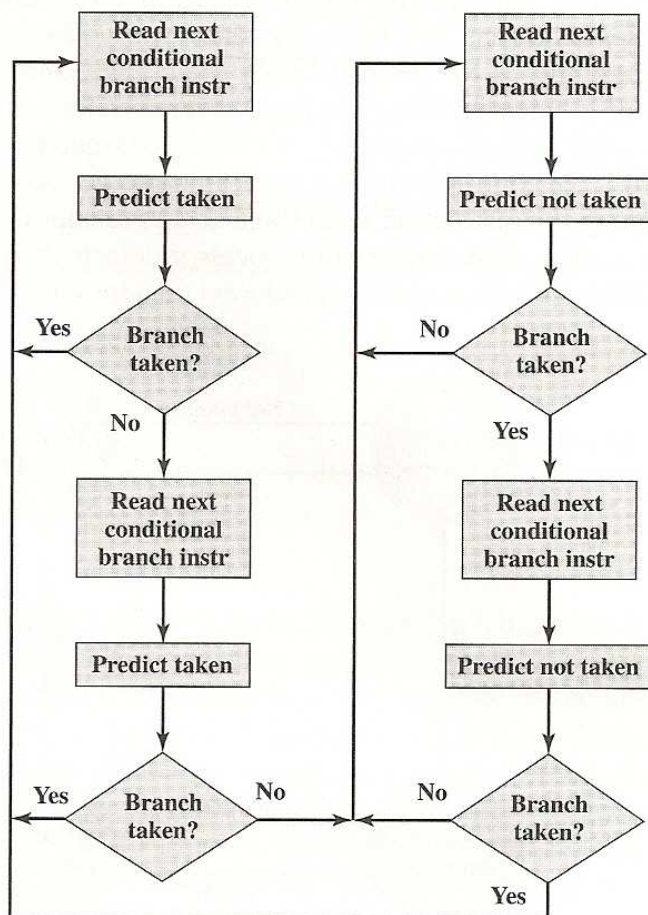


Figure 12.16 Branch Prediction Flowchart

possibilities). Assume that the algorithm starts at the upper left-hand corner of the flowchart. As long as each succeeding conditional branch instruction that is encountered is taken, the decision process predicts that the next branch will be taken. If a single prediction is wrong, the algorithm continues to predict that the next branch is taken. Only if two successive branches are not taken does the algorithm shift to the right-hand side of the flowchart. Subsequently, the algorithm will predict that branches are not taken until two branches in a row are taken. Thus, the algorithm requires two consecutive wrong predictions to change the prediction decision.

The decision process can be represented more compactly by a finite-state machine, shown in Figure 12.17. The finite-state machine representation is commonly used in the literature.

The use of history bits, as just described, has one drawback: If the decision is made to take the branch, the target instruction cannot be fetched until the target address, which is an operand in the conditional branch instruction, is decoded. Greater efficiency could be achieved if the instruction fetch could be initiated as soon as the branch decision is made. For this purpose, more information must be saved, in what is known as a branch target buffer, or a branch history table.

The branch history table is a small cache memory associated with the instruction fetch stage of the pipeline. Each entry in the table consists of three elements: the address of a branch instruction, some number of history bits that record the state of use of that instruction, and information about the target instruction. In most proposals and implementations, this third field contains the address of the target instruction. Another possibility is for the third field to actually contain the target instruction. The trade-off is clear: Storing the target address yields a smaller table but a greater instruction fetch time compared with storing the target instruction [RECH98].

Figure 12.18 contrasts this scheme with a predict-never-taken strategy. With the former strategy, the instruction fetch stage always fetches the next sequential address. If a branch is taken, some logic in the processor detects this and instructs that the next instruction be fetched from the target address (in addition to flushing the pipeline). The

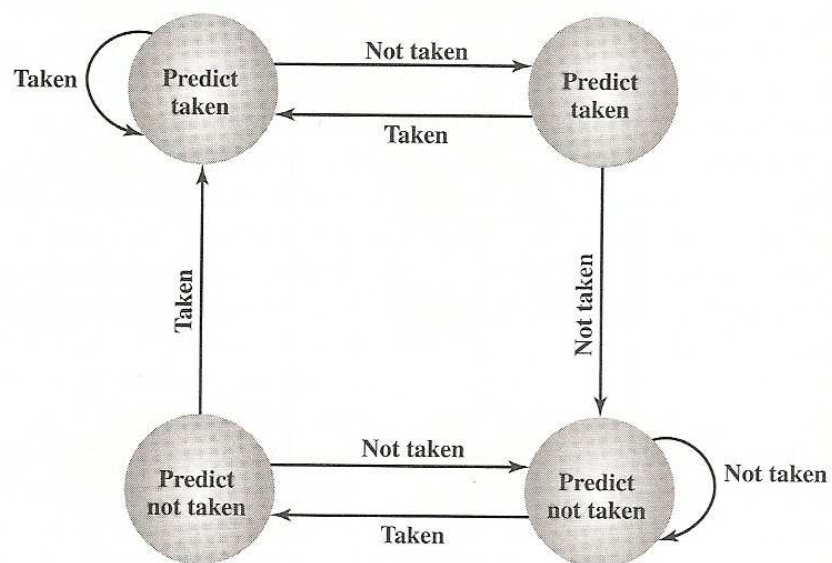
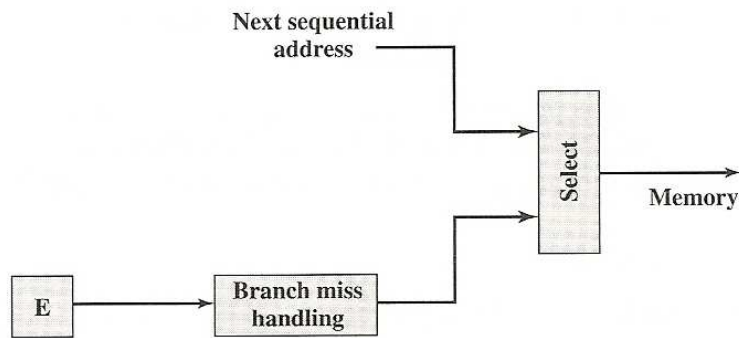
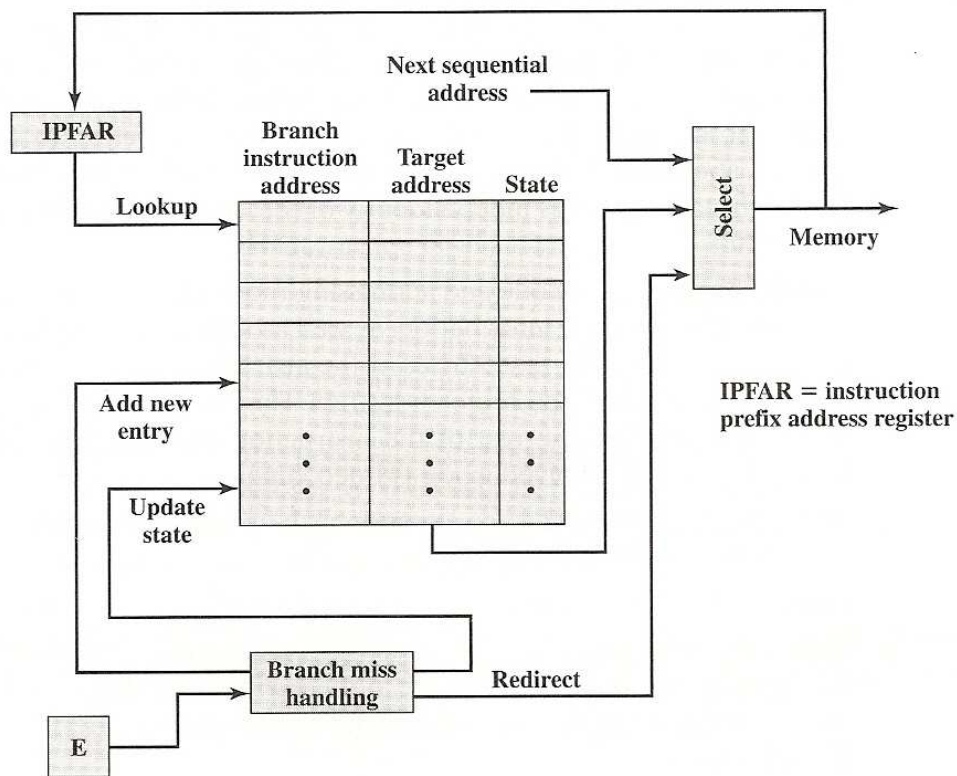


Figure 12.17 Branch Prediction State Diagram



(a) Predict never taken strategy



(b) Branch history table strategy

Figure 12.18 Dealing with Branches

branch history table is treated as a cache. Each prefetch triggers a lookup in the branch history table. If no match is found, the next sequential address is used for the fetch. If a match is found, a prediction is made based on the state of the instruction: Either the next sequential address or the branch target address is fed to the select logic.

When the branch instruction is executed, the execute stage signals the branch history table logic with the result. The state of the instruction is updated to reflect a correct or incorrect prediction. If the prediction is incorrect, the select logic is redirected to the correct address for the next fetch. When a conditional branch instruction is encountered that is not in the table, it is added to the table and one of the existing entries is discarded, using one of the cache replacement algorithms discussed in Chapter 4.

One example of an implementation of a branch history table is the Advanced Micro Device AMD29000 microprocessor.

Delayed Branch It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired. This intriguing approach is examined in Chapter 13.

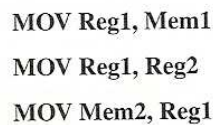
Intel 80486 Pipelining

The 80486 implements a five-stage pipeline:

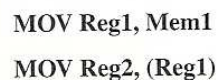
- **Fetch:** Instructions are fetched from the cache or from external memory and placed into one of the two 16-byte prefetch buffers. The objective of the fetch stage is to fill the prefetch buffers with new data as soon as the old data have been consumed by the instruction decoder. Because instructions are of variable length (from 1 to 11 bytes not counting prefixes), the status of the prefetcher relative to the other pipeline stages varies from instruction to instruction. On average, about five instructions are fetched with each 16-byte load [CRAW90]. The fetch stage operates independently of the other stages to keep the prefetch buffers full.
- **Decode stage 1:** All opcode and addressing-mode information is decoded in the D1 stage. The required information, as well as instruction-length information, is included in at most the first 3 bytes of the instruction. Hence, 3 bytes are passed to the D1 stage from the prefetch buffers. The D1 decoder can then direct the D2 stage to capture the rest of the instruction (displacement and immediate data), which is not involved in the D1 decoding.
- **Decode stage 2:** The D2 stage expands each opcode into control signals for the ALU. It also controls the computation of the more complex addressing modes.
- **Execute:** This stage includes ALU operations, cache access, and register update.
- **Write back:** This stage, if needed, updates registers and status flags modified during the preceding execute stage. If the current instruction updates memory, the computed value is sent to the cache and to the bus-interface write buffers at the same time.

With the use of two decode stages, the pipeline can sustain a throughput of close to one instruction per clock cycle. Complex instructions and conditional branches can slow down this rate.

Figure 12.19 shows examples of the operation of the pipeline. Part a shows that there is no delay introduced into the pipeline when a memory access is required. However, as part b shows, there can be a delay for values used to compute memory addresses. That is, if a value is loaded from memory into a register and that register is then used as a base register in the next instruction, the processor will stall for one cycle. In this example, the processor accesses the cache in the EX stage of the first instruction and stores the value retrieved in the register during the WB stage. However, the next instruction needs this register in its D2 stage. When the D2 stage lines up with the WB stage of the previous instruction, bypass signal paths allow the D2 stage to have access to the same data being used by the WB stage for writing, saving one pipeline stage.



(a) No data load delay in the pipeline



(b) Pointer load delay



(c) Branch instruction timing

Figure 12.19 80486 Instruction Pipeline Examples

Figure 12.19c illustrates the timing of a branch instruction, assuming that the branch is taken. The compare instruction updates condition codes in the WB stage, and bypass paths make this available to the EX stage of the jump instruction at the same time. In parallel, the processor runs a speculative fetch cycle to the target of the jump during the EX stage of the jump instruction. If the processor determines a false branch condition, it discards this prefetch and continues execution with the next sequential instruction (already fetched and decoded).

12.5 THE PENTIUM PROCESSOR

An overview of the Pentium 4 processor organization is depicted in Figure 4.13. In this section, we examine some of the details.

Register Organization

The register organization includes the following types of registers (Table 12.2):

- **General:** There are eight 32-bit general-purpose registers (see Figure 12.3c). These may be used for all types of Pentium instructions; they can also hold operands for address calculations. In addition, some of these registers also serve special purposes. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands without having to reference these registers explicitly in the instruction. As a result, a number of instructions can be encoded more compactly.
- **Segment:** The six 16-bit segment registers contain segment selectors, which index into segment tables, as discussed in Chapter 8. The code segment (CS) register references the segment containing the instruction being executed. The