**CPU STRUCTURE AND FUNCTION UNIT-3-PT-2**

***14.1]PROCESSOR ORGANISATION***

To understand the organization of the processor, let us consider the requirements

placed on the processor, the things that it must do:

■ Fetch instruction: The processor reads an instruction from memory (register,

cache, main memory).

■ Interpret instruction: The instruction is decoded to determine what action is

required.

■ Fetch data: The execution of an instruction may require reading data from

memory or an I/O module.

■ Process data: The execution of an instruction may require performing some

arithmetic or logical operation on data.

■ Write data: The results of an execution may require writing data to memory

or an I/O module.

To do these things, it should be clear that the processor needs to store some

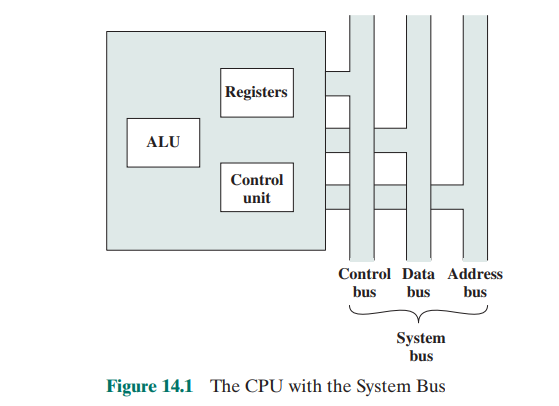
data temporarily. It must remember the location of the last instruction so that it can

know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the processor needs

a small internal memory.

Figure 14.1 is a simplified view of a processor, indicating its connection to the

rest of the system via the system bus. A similar interface would be needed for any

of the interconnection structures described in Chapter 3. The reader will recall that

the major components of the processor are an arithmetic and logic unit (ALU) and

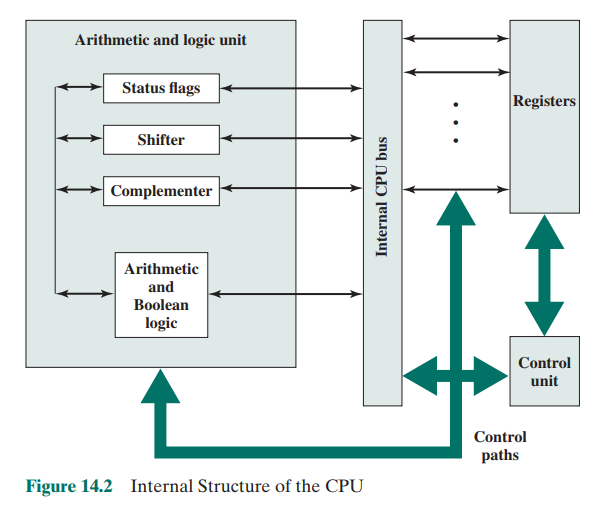
a control unit (CU). The ALU does the actual computation or processing of data.

The control unit controls the movement of data and instructions into and out of the

processor and controls the operation of the ALU. In addition, the

figure shows a

minimal internal memory, consisting of a set of storage locations, called registers.

Figure 14.2 is a slightly more detailed view of the processor. The data transfer and logic control paths are indicated, including an element labeled internal 

processor bus. This element is needed to transfer data between the various registers

and the ALU because the ALU in fact operates only on data in the internal processor memory. The figure also shows typical basic elements of the ALU. Note the

similarity between the internal structure of the computer as a whole and the internal

structure of the processor. In both cases, there is a small collection of major elements (computer: processor, I/O, memory; processor: control unit, ALU, registers)

connected by data paths.

***14.2]REGISTER ORGANIZATION***

As we discussed in Chapter 4, a computer system employs a memory hierarchy. At

higher levels of the hierarchy, memory is faster, smaller, and more expensive (per

bit). Within the processor, there is a set of registers that function as a level of memory above main memory and cache in the hierarchy. The registers in the processor

perform two roles:

■ User-visible registers: Enable the machine- or assembly language programmer

to minimize main memory references by optimizing use of registers.

■ Control and status registers: Used by the control unit to control the operation

of the processor and by privileged, operating system programs to control the

execution of programs.

There is not a clean separation of registers into these two categories. For

example, on some machines the program counter is user visible (e.g., x86), but on

many it is not. For purposes of the following discussion, however, we will use these

categories.

**User-Visible Registers**

A user-visible register is one that may be referenced by means of the machine

language that the processor executes. We can characterize these in the following

categories:

■ General purpose

■ Data

■ Address

■ Condition codes

General-purpose registers can be assigned to a variety of functions by the programmer. Sometimes their use within the instruction set is orthogonal to the operation. That is, any general-purpose register can contain the operand for any opcode.

This provides true general-purpose register use. Often, however, there are restrictions.

For example, there may be dedicated registers for floating-point and stack operations.

In some cases, general-purpose registers can be used for addressing functions

(e.g., register indirect, displacement). In other cases, there is a partial or clean separation between data registers and address registers. Data registers may be used

only to hold data and cannot be employed in the calculation of an operand address.

Address registers may themselves be somewhat general purpose, or they may be

devoted to a particular addressing mode. Examples include the following:

■ Segment pointers: In a machine with segmented addressing (see Section 8.3),

a segment register holds the address of the base of the segment. There may be

multiple registers: for example, one for the operating system and one for the

current process.

■ Index registers: These are used for indexed addressing and may be autoindexed.

■ Stack pointer: If there is user-visible stack addressing, then typically there is

a dedicated register that points to the top of the stack. This allows implicit

addressing; that is, push, pop, and other stack instructions need not contain an

explicit stack operand.

There are several design issues to be addressed here. An important issue

is whether to use completely general- purpose registers or to specialize their use.

We have already touched on this issue in the preceding chapter because it affects

instruction set design. With the use of specialized registers, it can generally be implicit in the opcode which type of register a certain operand specifier refers to. The

operand specifier must only identify one of a set of specialized registers rather than

one out of all the registers, thus saving bits. On the other hand, this specialization

limits the programmer’s flexibility.

Another design issue is the number of registers, either general purpose or data

plus address, to be provided. Again, this affects instruction set design because more

registers require more operand specifier bits. As we previously discussed, somewhere

between 8 and 32 registers appears optimum [LUND77]. Fewer registers result in more

memory references; more registers do not noticeably reduce memory references (e.g.,

see [WILL90]). However, a new approach, which finds advantage in the use of hundreds of registers, is exhibited in some RISC systems and is discussed in Chapter 15.

Finally, there is the issue of register length. Registers that must hold addresses

obviously must be at least long enough to hold the largest address. Data registers

should be able to hold values of most data types. Some machines allow two contiguous registers to be used as one for holding double-length values.

A final category of registers, which is at least partially visible to the user, holds

condition codes (also referred to as flags). Condition codes are bits set by the processor hardware as the result of operations. For example, an arithmetic operation

may produce a positive, negative, zero, or overflow result. In addition to the result

itself being stored in a register or memory, a condition code is also set. The code

may subsequently be tested as part of a conditional branch operation.

Condition code bits are collected into one or more registers. Usually, they

form part of a control register. Generally, machine instructions allow these bits to

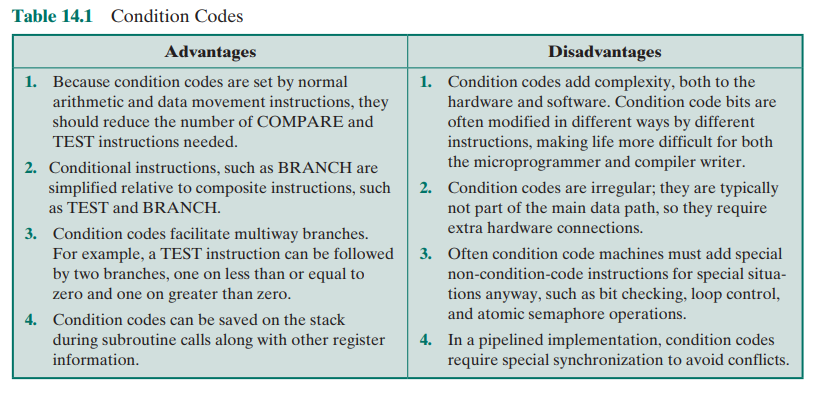
be read by implicit reference, but the programmer cannot alter them.

Many processors, including those based on the IA- 64 architecture and the

MIPS processors, do not use condition codes at all. Rather, conditional branch

instructions specify a comparison to be made and act on the result of the comparison, without storing a condition code. Table 14.1, based on [DERO87], lists key

advantages and disadvantages of condition codes.



In some machines, a subroutine call will result in the automatic saving of all

user-visible registers, to be restored on return. The processor performs the saving

and restoring as part of the execution of call and return instructions. This allows

each subroutine to use the user-visible registers independently. On other machines,

it is the responsibility of the programmer to save the contents of the relevant uservisible registers prior to a subroutine call, by including instructions for this purpose

in the program.

**Control and Status Registers**

There are a variety of processor registers that are employed to control the operation

of the processor. Most of these, on most machines, are not visible to the user. Some

of them may be visible to machine instructions executed in a control or operating

system mode.

Of course, different machines will have different register organizations and

use different terminology. We list here a reasonably complete list of register types,

with a brief description.

Four registers are essential to instruction execution:

■ Program counter (PC): Contains the address of an instruction to be fetched.

■ Instruction register (IR): Contains the instruction most recently fetched.

■ Memory address register (MAR): Contains the address of a location in

memory.

■ Memory buffer register (MBR): Contains a word of data to be written to

memory or the word most recently read.

Not all processors have internal registers designated as MAR and MBR, but

some equivalent buffering mechanism is needed whereby the bits to be transferred

to the system bus are staged and the bits to be read from the data bus are temporarily stored.

Typically, the processor updates the PC after each instruction fetch so that the

PC always points to the next instruction to be executed. A branch or skip instruction will also modify the contents of the PC. The fetched instruction is loaded into

an IR, where the opcode and operand specifiers are analyzed. Data are exchanged

with memory using the MAR and MBR. In a bus-organized system, the MAR connects directly to the address bus, and the MBR connects directly to the data bus.

User-visible registers, in turn, exchange data with the MBR.

The four registers just mentioned are used for the movement of data between

the processor and memory. Within the processor, data must be presented to the

ALU for processing. The ALU may have direct access to the MBR and user-visible

registers. Alternatively, there may be additional buffering registers at the boundary

to the ALU; these registers serve as input and output registers for the ALU and

exchange data with the MBR and user-visible registers.

Many processor designs include a register or set of registers, often known as

the program status word (PSW), that contain status information. The PSW typically contains condition codes plus other status information. Common fields or flags

include the following:

■ Sign: Contains the sign bit of the result of the last arithmetic operation.

■ Zero: Set when the result is 0.

■ Carry: Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit. Used for multiword arithmetic operations.

■ Equal: Set if a logical compare result is equality.

■ Overflow: Used to indicate arithmetic overflow.

■ Interrupt Enable/Disable: Used to enable or disable interrupts.

■ Supervisor: Indicates whether the processor is executing in supervisor or

user mode. Certain privileged instructions can be executed only in supervisor

mode, and certain areas of memory can be accessed only in supervisor mode.

A number of other registers related to status and control might be found in a

particular processor design. There may be a pointer to a block of memory containing additional status information (e.g., process control blocks). In machines using

vectored interrupts, an interrupt vector register may be provided. If a stack is used

to implement certain functions (e.g., subroutine call), then a system stack pointer is

needed. A page table pointer is used with a virtual memory system. Finally, registers may be used in the control of I/O operations.

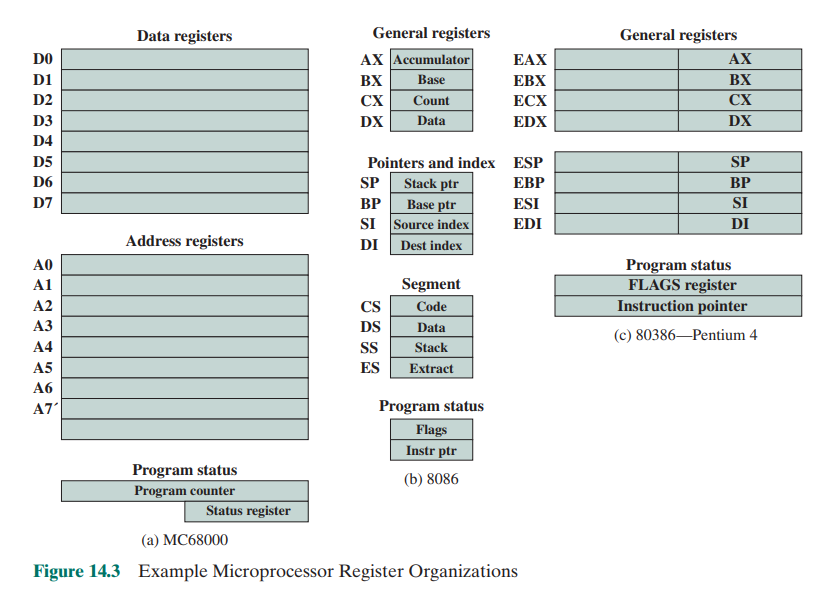
A number of factors go into the design of the control and status register organization. One key issue is operating system support. Certain types of control information are of specific utility to the operating system. If the processor designer has

a functional understanding of the operating system to be used, then the register

organization can to some extent be tailored to the operating system.

Another key design decision is the allocation of control information between

registers and memory. It is common to dedicate the first (lowest) few hundred or



thousand words of memory for control purposes. The designer must decide how much control information should be in registers and how much in memory. The usual trade-off of cost versus speed arises.

**Example Microprocessor Register Organizations**

It is instructive to examine and compare the register organization of comparable

systems. In this section, we look at two 16-bit microprocessors that were designed at

about the same time: the Motorola MC68000 [STRI79] and the Intel 8086 [MORS78].

Figures 14.3a and b depict the register organization of each; purely internal registers,

such as a memory address register, are not shown.

The MC68000 partitions its 32-bit registers into eight data registers and nine

address registers. The eight data registers are used primarily for data manipulation

and are also used in addressing as index registers. The width of the registers allows

8-, 16-, and 32-bit data operations, determined by opcode. The address registers contain 32-bit (no segmentation) addresses; two of these registers are also used as stack

pointers, one for users and one for the operating system, depending on the current

execution mode. Both registers are numbered 7, because only one can be used at a

time. The MC68000 also includes a 32-bit program counter and a 16-bit status register.

The Motorola team wanted a very regular instruction set, with no specialpurpose registers. A concern for code efficiency led them to divide the registers into two functional components, saving one bit on each register specifier. This seems a reasonable compromise between complete generality and code compaction.

The Intel 8086 takes a different approach to register organization. Every

register is special purpose, although some registers are also usable as general purpose. The 8086 contains four 16-bit data registers that are addressable on a byte

or 16-bit basis, and four 16-bit pointer and index registers. The data registers can

be used as general purpose in some instructions. In others, the registers are used

implicitly. For example, a multiply instruction always uses the accumulator. The

four pointer registers are also used implicitly in a number of operations; each

contains a segment offset. There are also four 16-bit segment registers. Three of

the four segment registers are used in a dedicated, implicit fashion, to point to

the segment of the current instruction (useful for branch instructions), a segment

containing data, and a segment containing a stack, respectively. These dedicated

and implicit uses provide for compact encoding at the cost of reduced flexibility.

The 8086 also includes an instruction pointer and a set of 1-bit status and control

flags.

The point of this comparison should be clear. There is no universally accepted

philosophy concerning the best way to organize processor registers [TOON81]. As

with overall instruction set design and so many other processor design issues, it is

still a matter of judgment and taste.

A second instructive point concerning register organization design is illustrated in Figure 14.3c. This figure shows the user- visible register organization for

the Intel 80386 [ELAY85], which is a 32-bit microprocessor designed as an extension of the 8086.1

The 80386 uses 32-bit registers. However, to provide upward

compatibility for programs written on the earlier machine, the 80386 retains the

original register organization embedded in the new organization. Given this design

constraint, the architects of the 32-bit processors had limited flexibility in designing

the register organization.

***14.3]INSTRUCTION CYCLE***

In Section 3.2, we described the processor’s instruction cycle (Figure 3.9). To recall,

an instruction cycle includes the following stages:

■ Fetch: Read the next instruction from memory into the processor.

■ Execute: Interpret the opcode and perform the indicated operation.

■ Interrupt: If interrupts are enabled and an interrupt has occurred, save the

current process state and service the interrupt.

We are now in a position to elaborate somewhat on the instruction cycle. First,

we must introduce one additional stage, known as the indirect cycle.

**The Indirect Cycle**

We have seen, in Chapter 13, that the execution of an instruction may involve one

or more operands in memory, each of which requires a memory access. Further, if

indirect addressing is used, then additional memory accesses are required.

We can think of the fetching of indirect addresses as one more instruction

stages. The result is shown in Figure 14.4. The main line of activity consists of alternating instruction fetch and instruction execution activities. After an instruction is

fetched, it is examined to determine if any indirect addressing is involved. If so, the

required operands are fetched using indirect addressing. Following execution, an

interrupt may be processed before the next instruction fetch.

Another way to view this process is shown in Figure 14.5, which is a revised

version of Figure 3.12. This illustrates more correctly the nature of the instruction

cycle. Once an instruction is fetched, its operand specifiers must be identified. Each

input operand in memory is then fetched, and this process may require indirect

addressing. Register-based operands need not be fetched. Once the opcode is executed, a similar process may be needed to store the result in main memory.

**Data Flow**

The exact sequence of events during an instruction cycle depends on the design of

the processor. We can, however, indicate in general terms what must happen. Let us

assume that a processor that employs a memory address register (MAR), a memory

buffer register (MBR), a program counter (PC), and an instruction register (IR).

During the fetch cycle, an instruction is read from memory. Figure 14.6 shows

the flow of data during this cycle. The PC contains the address of the next instruction to be fetched. This address is moved to the MAR and placed on the address

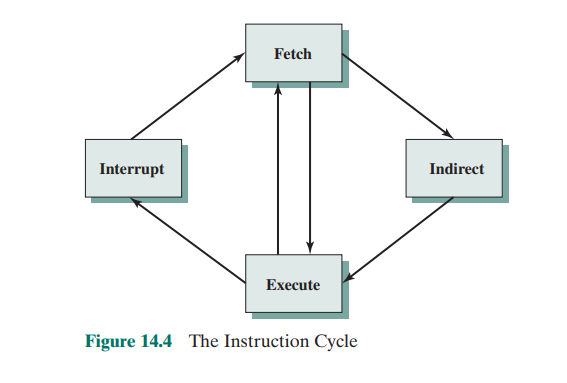
bus. The control unit requests a memory read, and the result is placed on the data

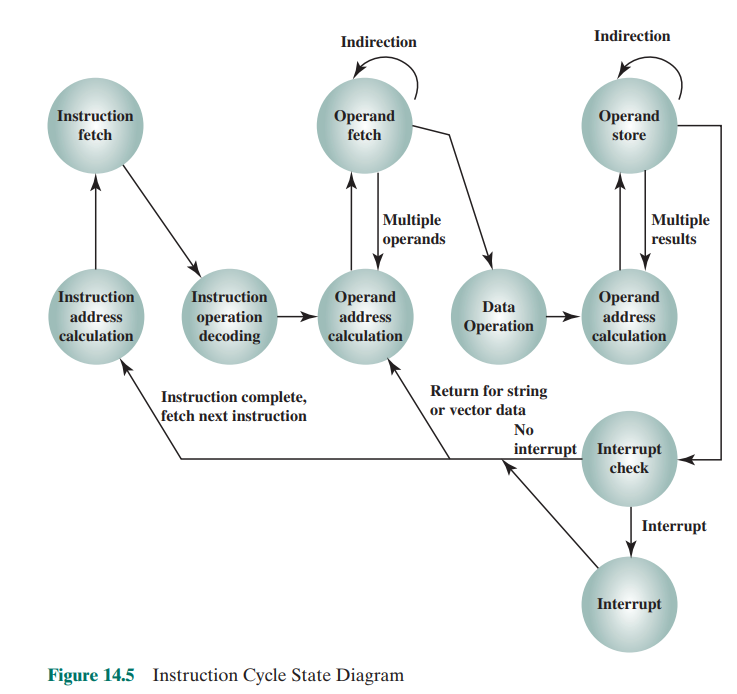
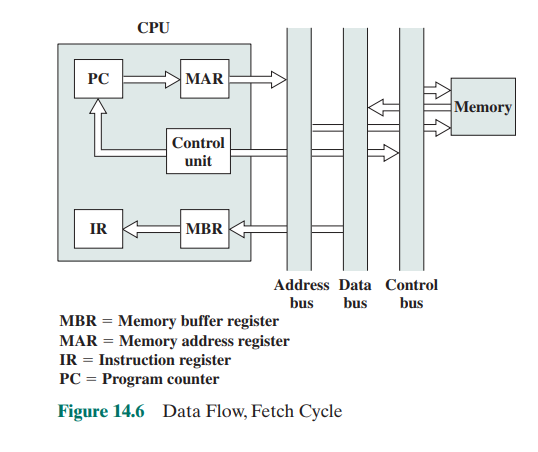
bus and copied into the MBR and then moved to the IR. Meanwhile, the PC is

incremented by 1, preparatory for the next fetch.

Once the fetch cycle is over, the control unit examines the contents of the IR

to determine if it contains an operand specifier using indirect addressing. If so, an





indirect cycle is performed. As shown in Figure 14.7, this is a simple cycle. The rightmost N bits of the MBR, which contain the address reference, are transferred to

the MAR. Then the control unit requests a memory read, to get the desired address

of the operand into the MBR.

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 14.8). 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

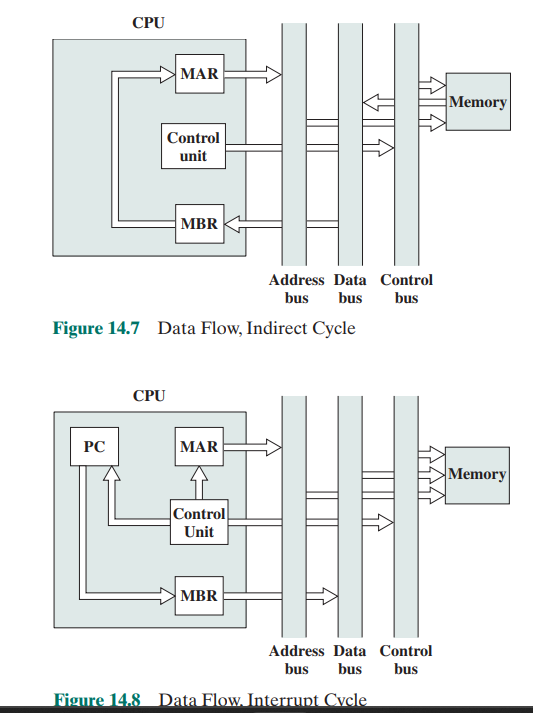
transferred to the MBR to be written into memory. The special memory location

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.



***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

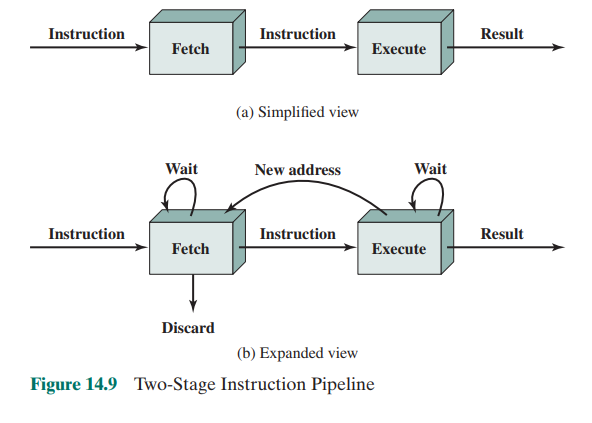
used to fetch the next instruction in parallel with the execution of the current one.

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

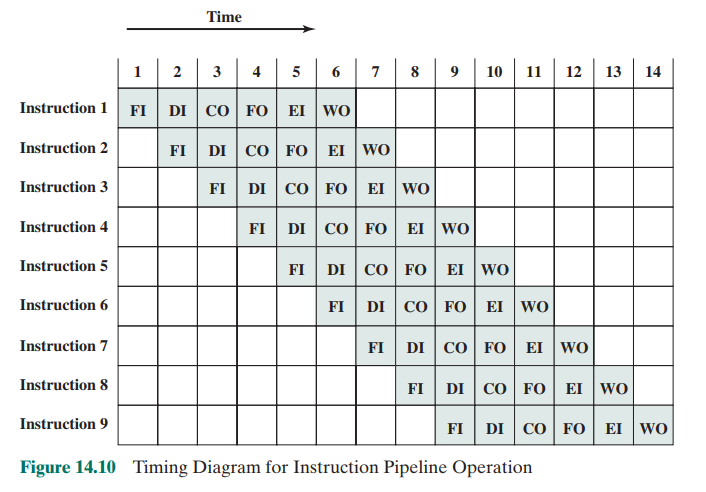
example, a load instruction does not need the WO stage. However, to simplify the

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.

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.

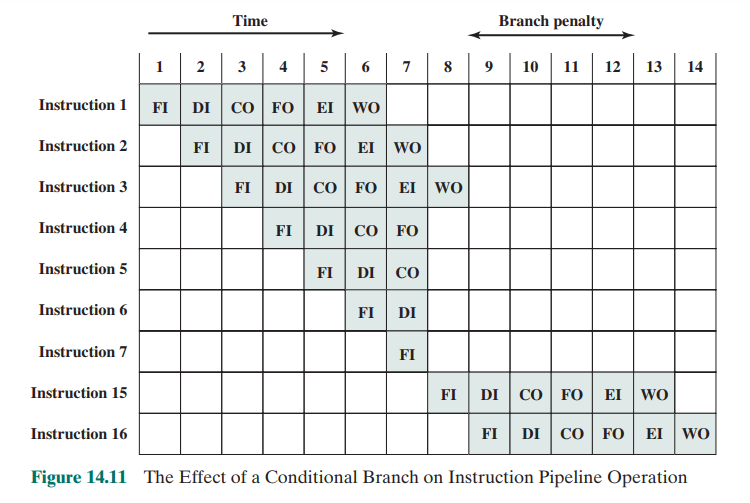
The CO stage may depend on the contents of a register that could be altered by a

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, 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

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.

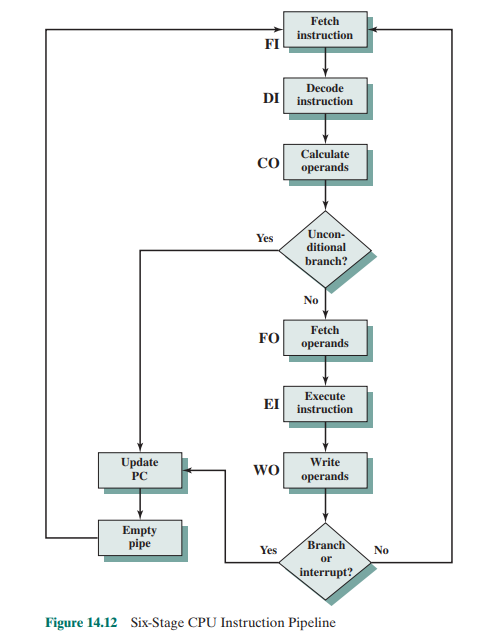
2. The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the

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

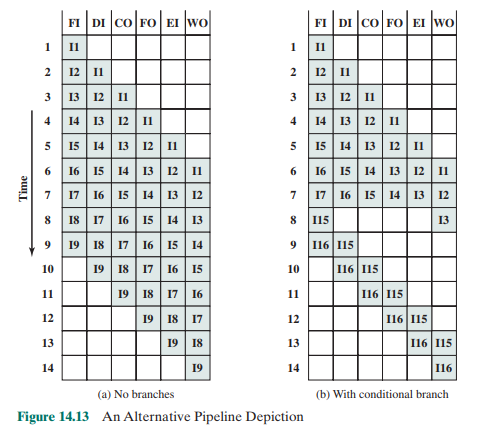
relative speedup (based on a discussion in [HWAN93]). The cycle time t of an

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





where

ti = time delay of the circuitry in the ith stage of the pipeline

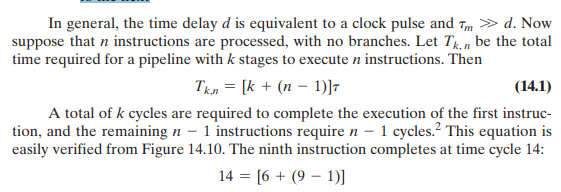
tm = maximum stage delay (delay through stage which experiences the largest

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



Now consider a processor with equivalent functions but no pipeline, and

assume that the instruction cycle time is kt. The speedup factor for the instruction

pipeline compared to execution without the pipeline is defined as



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 S ∞),

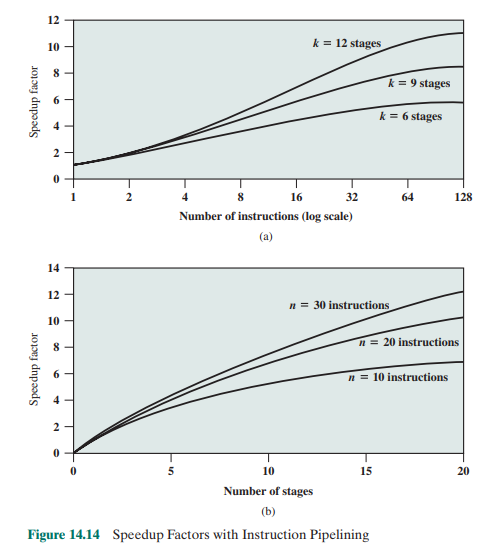
we have a k-fold speedup. Figure 14.14b shows the speedup factor as a function of

the number of stages in the instruction pipeline.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 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

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

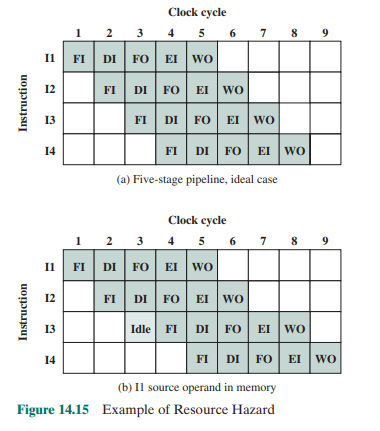
five-stage pipeline, in which each stage takes one clock cycle. Figure 14.15a shows

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

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.

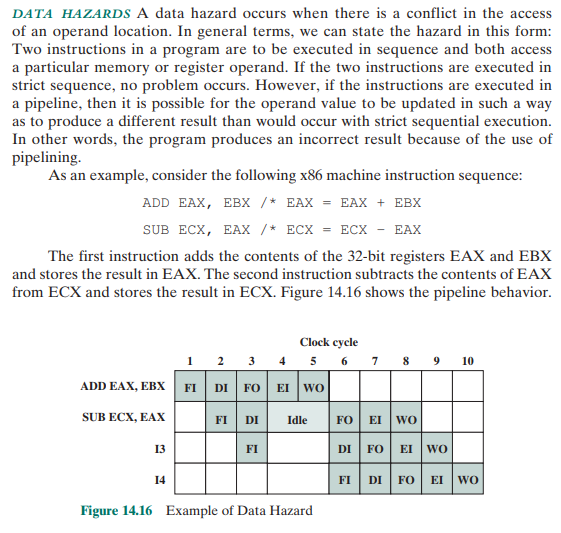
Another example of a resource conflict is a situation in which multiple instructions are ready to enter the execute instruction phase and there is a single ALU. One

solutions to such resource hazards is to increase available resources, such as having

multiple ports into main memory and multiple ALU units.

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

**Data hazards**



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 clocks 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.

**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.

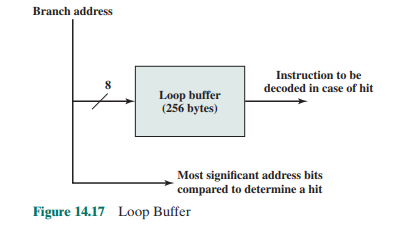
The loop buffer is similar in principle to a cache dedicated to instructions.

The differences are that the loop buffer only retains instructions in sequence and is

much smaller in size and hence lower in cost.

Figure 14.17 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 DBcc (decrement and branch on condition) instruction (see Problem 14.14).

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 predict-never-taken approach is the most popular of all the branch prediction methods.

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 highspeed 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

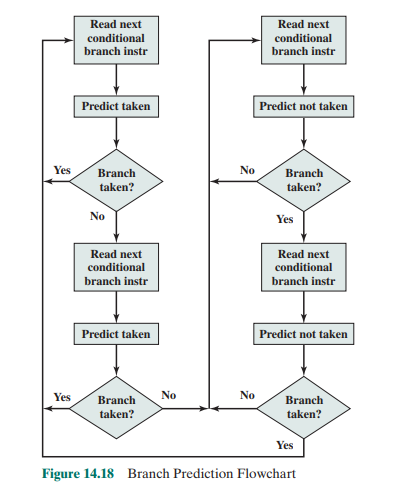
taken, such as a loop instruction. With only one bit of history, an error in prediction will occur twice for each use of the loop: once on entering the loop, and once

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 14.18 shows a typical approach (see Problem 14.13 for other



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 14.19. 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

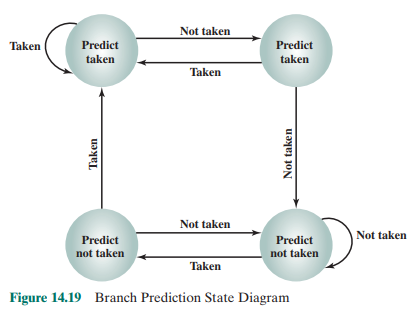
instruction. The trade-off is clear: Storing the target address yields a smaller table

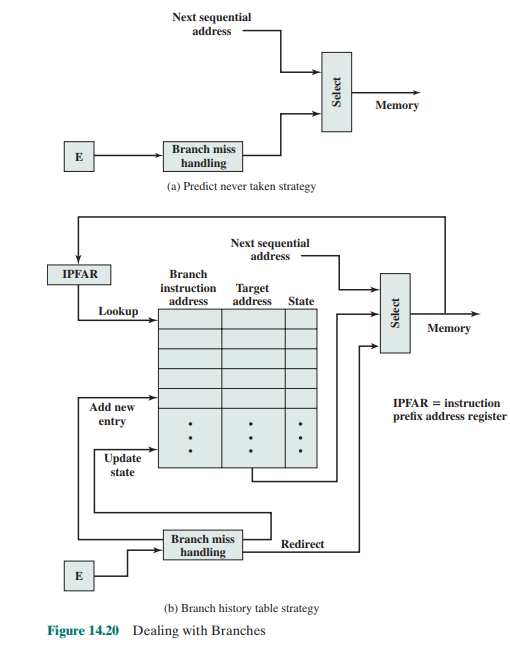
but a greater instruction fetch time compared with storing the target instruction

[RECH98].

Figure 14.20 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 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.

A refinement of the branch history approach is referred to as two- level or

correlation-based branch history [YEH91]. This approach is based on the assumption that whereas in loop-closing branches, the past history of a particular branch

instruction is a good predictor of future behavior, with more complex control-flow

structures, the direction of a branch is frequently correlated with the direction of

related branches. An example is an if-then-else or case structure. There are a number of strategies possible. Typically, recent global branch history (i.e., the history of

the most recent branches not just of this branch instruction) is used in addition to

the history of the current branch instruction. The general structure is defined as an

(m, n) correlator, which uses the behavior of the last m branches to choose from 2m

n-bit branch predictors for the current branch instruction. In other words, an n-bit

history is kept for a give branch for each possible combination of branches taken by

the most recent m branches.

**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 15.

INTEL 80468 PIPELINING?

FK IT:->

**Intel 80486 Pipelining**

An instructive example of an instruction pipeline is that of the Intel 80486. 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 14.21 shows examples of the operation of the pipeline. Figure 14.21a

shows that there is no delay introduced into the pipeline when a memory access is

required. However, as Figure 14.21b 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.

Figure 14.21c 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).

