I/O Organizations and Peripherals

6.1 INPUT/ OUTPUT SYSTEMS

In computing, input/output or I/O is the communication between an information processing system (such as a computer) and the outside world, possibly a human or another information processing system.
Inputs are the signals or data received by the system and outputs are the signals or data sent from it.I/O devices are used by a human (or other system) to communicate with a computer.For instance, a keyboard or mouse is an input device for a computer, while monitors and printers are output devices.
Devices for communication between computers, such as modems and network cards, typically perform both input and output operations.The designation of a device as either input or output depends on perspective.
Mouse and keyboards take physical movements that the human user outputs and convert them into input signals that a computer can understand; the output from these devices is the computer's input.
Similarly, printers and monitors take signals that a computer outputs as input, and they convert these signals into a representation that human users can understand.

The input/output module (I/O module) is normally connected to the computer system on one end and one or more input/output devices on the other.

6.2 Need For I/O module

An I/O module is needed because :
(a) Diversity of I/O devices makes it difficult to include all the peripheral device logic(i.e. its control commands, data format etc.) into CPU.

(b) The I/O devices are usually slower than the memory and CPU. Therefore, it is not advisable to use them on high speed system bus directly for communication purpose.

(c) The data format and word length used by the peripheral may be quite different than that of a CPU.

Thus,
(i) An I/O module is a mediator between the processor and an I/O device/devices.
(ii) It controls the data exchange between the external devices and main memory or CPU registers.

(iii)An I/O module provide an interface internal to the computer which connects it to CPU and main memory and an interface external to the computer connecting it to external device or peripheral.

(iv)The I/O module should not only communicate the information from CPU to I/O device, but it should also coordinate these two.

(v)In addition since there are speed differences between CPU and I/O devices, the I/O module should have facilities like buffer (storage area) and error detection mechanism.

6.3 Functions of I/O Module:
The major functions of an I/O module are:
1. Processor communication -- this involves the following tasks:
a. exchange of data between processor and I/O module

b. command decoding - I/O module accepts commands sent from the processor. E.g., the I/O module for a disk drive may accept the following commands from the processor: READ SECTOR, WRITE SECTOR, SEEK TRACK, etc.

c. status reporting – The device must be able to report its status to the processor, e.g., disk drive busy, ready etc. Status reporting may also involve reporting various errors.

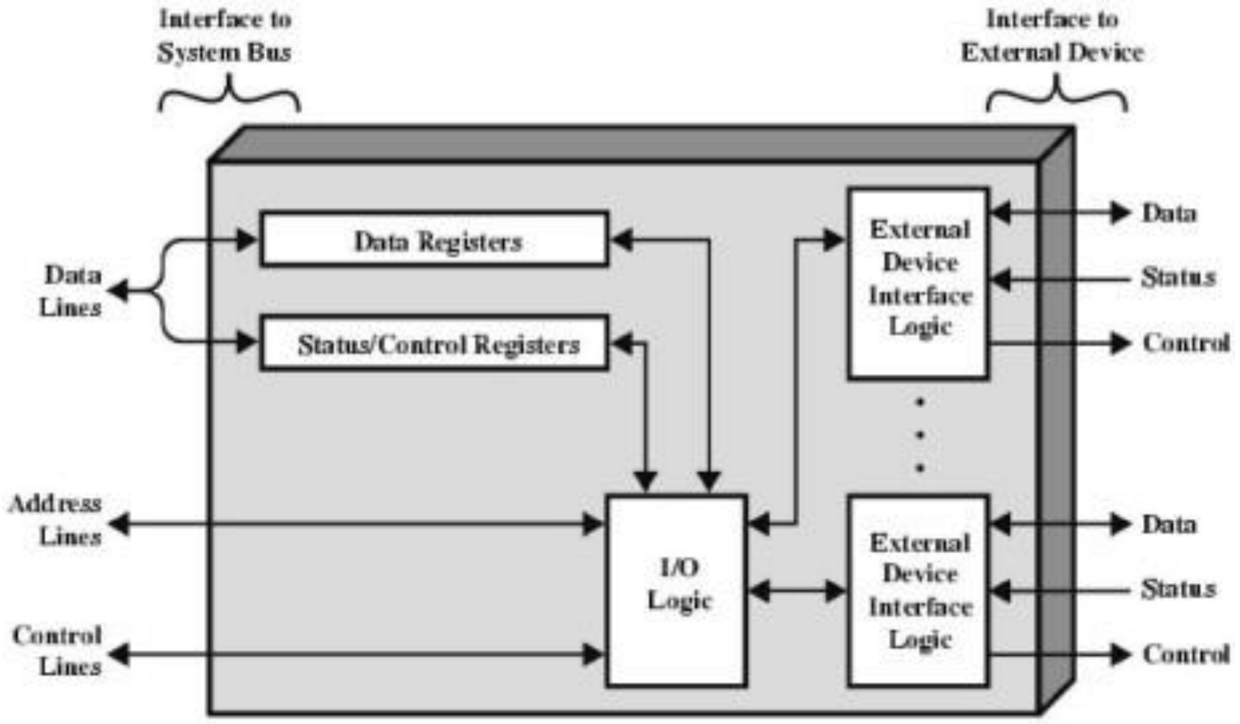
d. Address recognition – Each I/O device has a unique address and the I/O module must recognize this address.

2.Device communication – The I/O module must be able to perform device communication such as status reporting.

3.Control & timing – The I/O module must be able to co-ordinate the flow of data between the internal resources (such as processor, memory) and external devices.

4.Data buffering – This is necessary as there is a speed mismatch between speed of data transfer between processor and memory and external devices. Data coming from the main memory are sent to an I/O module in a rapid burst. The data is buffered in the I/O module and then sent to the peripheral device at its rate.

5. Error detection – The I/O module must also be able to detect errors and report them to the processor. These errors may be mechanical errors (such as paper jam in a printer)



6.4 Types of Data Transfer Techniques

Three techniques are possible for I/O operations.

With programmed I/O, data are exchanged between the processor and the I/O module. The processor executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command, and transferring the data. When the processor issues a command to the I/O module, it must wait until the I/O operation is complete.

If the processor is faster than the I/O module, this is wasteful of processor time. With interrupt-driven I/O, the processor issues an I/O command, continues to execute other instructions, and is interrupted by the I/O module when the latter has completed its work. With both programmed and interrupt I/O, the processor is responsible for extracting data from main memory for output and storing data in main memory for input.

The alternative is known as direct memory access (DMA). In this mode, the I/O module and main memory exchange data directly, without processor involvement.

.

6.4.1 PROGRAMMED I/O

Overview of Programmed I/O

When the processor is executing a program and encounters an instruction relating to I/O, it executes that instruction by issuing a command to the appropriate I/O module. With programmed I/O, the I/O module will perform the requested action and then set the appropriate bits in the I/O status register (Figure 7.3).

The I/O module takes no further action to alert the processor.

In particular, it does not interrupt the processor.Thus, it is the responsibility of the processor periodically to check the status of the I/O module until it finds that the operation is complete.To explain the programmed I/O technique, we view it first from the point of view of the I/O commands issued by the processor to the I/O module, and then from the point of view of the I/O instructions executed by the processor.

I/O Commands

To execute an I/O-related instruction, the processor issues an address, specifying the particular I/O module and external device, and an I/O command.

There are four types of I/O commands that an I/O module may receive when it is addressed by a processor:

• **Control:** Used to activate a peripheral and tell it what to do. For example, a magnetic-tape unit may be instructed to rewind or to move forward one record. These commands are tailored to the particular type of peripheral device.

• **Test:** Used to test various status conditions associated with an I/O module and its peripherals. The processor will want to know that the peripheral of interest is powered on and available for use. It will also want to know if the most recent I/O operation is completed and if any errors occurred.

• **Read:** Causes the I/O module to obtain an item of data from the peripheral and place it in an internal buffer. The processor can then obtain the data item by requesting that the I/O module place it on the data bus.

• **Write:** Causes the I/O module to take an item of data (byte or word) from the data bus and subsequently transmit that data item to the peripheral.

When the processor, main memory, and I/O share a common bus, two modes of addressing are possible: memory mapped and isolated

memory mapped.

With memory-mapped I/O, there is a single address space for memory locations and I/O devices.

The processor treats the status and data registers of I/O modules as memory locations and uses the same machine instructions to access both memory and I/O devices. So, for example, with 10 address lines, a combined total of 2^{10} 1024 memory locations and I/O addresses can be supported, in any combination.

Isolated I/O

With memory-mapped I/O, a single read line and a single write line are needed on the bus. Alternatively, the bus may be equipped with memory read and write plus input and output command lines.

Now, the command line specifies whether the address refers to a memory location or an I/O device.

The full range of addresses may be available for both.

Again, with 10 address lines, the system may now support both 1024 memory locations and 1024 I/O addresses. Because the address space for I/O is isolated from that for memory, this is referred to as isolated I/O.

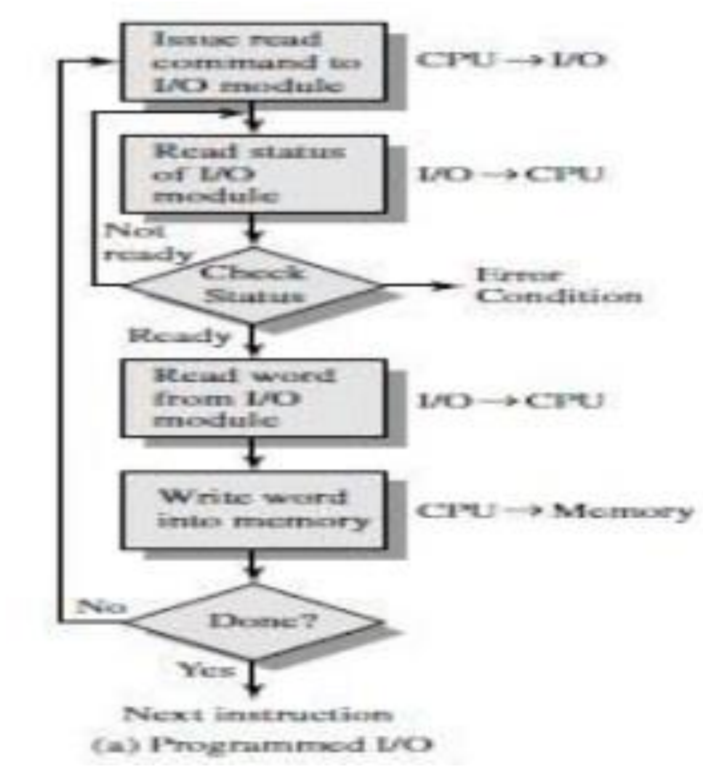


Figure 7.4a gives an example of the use of programmed I/O to read in a block of data from a peripheral device (e.g., a record from tape) into memory.

Data are read in one word (e.g., 16 bits) at a time.For each word that is read in, the processor must remain in a status-checking cycle until it determines that the word is available in the I/O module's data.

This flowchart highlights the main disadvantage of this technique:

it is a time-consuming process that keeps the processor busy needlessly.

INTERRUPT-DRIVEN I/O

The problem with programmed I/O is that the processor has to wait a long time for the I/O module of concern to be ready for either reception or transmission of data. The processor, while waiting, must repeatedly interrogate the status of the I/O module. As a result, the level of the performance of the entire system is severely degraded. An alternative is for the processor to issue an I/O command to a module and then go on to do some other useful work.

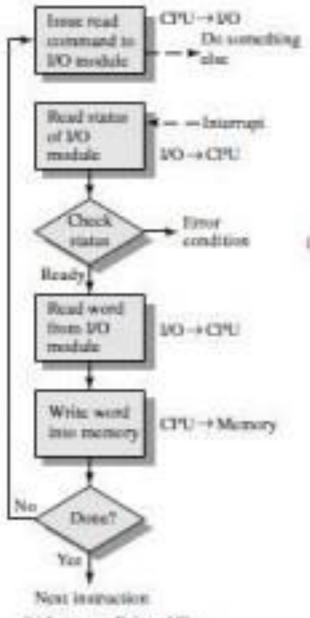
The I/O module will then interrupt the processor to request service when it is ready to exchange data with the processor.

The processor then executes the data transfer, as before, and then resumes its former processing.

Let us consider how this works, first from the point of view of the I/O module. For input, the I/O

module receives a READ command from the processor. The I/O module then proceeds to read data in from an associated peripheral. Once the data are in the module's data register, the module signals an interrupt to the processor over a control line.
The module then waits until its data are requested by the processor.
When the request is made, the module places its data on the data bus and is then ready for another I/O operation.

I/O Instructions
With programmed I/O, there is a close correspondence between the I/O-related instructions that the processor fetches from memory and the I/O commands that the processor issues to an I/O module to execute the instructions.
That is, the instructions are easily mapped into I/O commands, and there is often a simple one-to-one relationship.
The form of the instruction depends on the way in which external devices are addressed.
Typically, there will be many I/O devices connected through I/O modules to the system.
Each device is given a unique identifier or address.
When the processor issues an I/O command, the command contains the address of the desired device.
Thus, each I/O module must interpret the address lines to determine if the command is for itself



(b) Interrupt-Driven I/O

From the processor's point of view, the action for input is as follows.
The processor issues a READ command. It then goes off and does something else (e.g., the processor may be working on several different programs at the same time). At the end of each instruction cycle, the processor checks for interrupts.

When the interrupt from the I/O module occurs, the processor saves the context (e.g., program counter and processor registers) of the current program and processes the interrupt. In this case, the processor reads the word of data from the I/O module and stores it in memory. It then restores the context of the program it was working on (or some other program) and resumes execution.

Figure 7.4b shows the use of interrupt I/O for reading in a block of data.

Compare this with Figure 7.4a.

Interrupt I/O is more efficient than programmed I/O because it eliminates needless waiting. However, interrupt I/O still consumes a lot of processor time, because every word of data that goes from memory to I/O module or from I/O module to memory must pass through the processor.

Interrupt Processing

Let us consider the role of the processor in interrupt-driven I/O in more detail. The occurrence of an interrupt triggers a number of events, both in the processor hardware and in software.

Figure 7.6 shows a typical sequence. When an I/O device completes an I/O operation, the following sequence of hardware events occurs:

1. The device issues an interrupt signal to the processor.
2. The processor finishes execution of the current instruction before responding
to the interrupt, as indicated in Figure 3.9.
3. The processor tests for an interrupt, determines that there is one, and sends an acknowledgment signal to the device that issued the interrupt. The acknowledgment allows the device to remove its interrupt signal.
4. The processor now needs to prepare to transfer control to the interrupt routine. To begin, it needs to save information needed to resume the current program at the point of interrupt.The minimum information required is (a) the status of the processor, which is contained in a register called the program status word (PSW), and (b) the location of the next instruction to be executed, which is contained in the program counter.These can be pushed onto the system control stack.2
5. The processor now loads the program counter with the entry location of the interrupt-handling program that will respond to this interrupt. Depending on

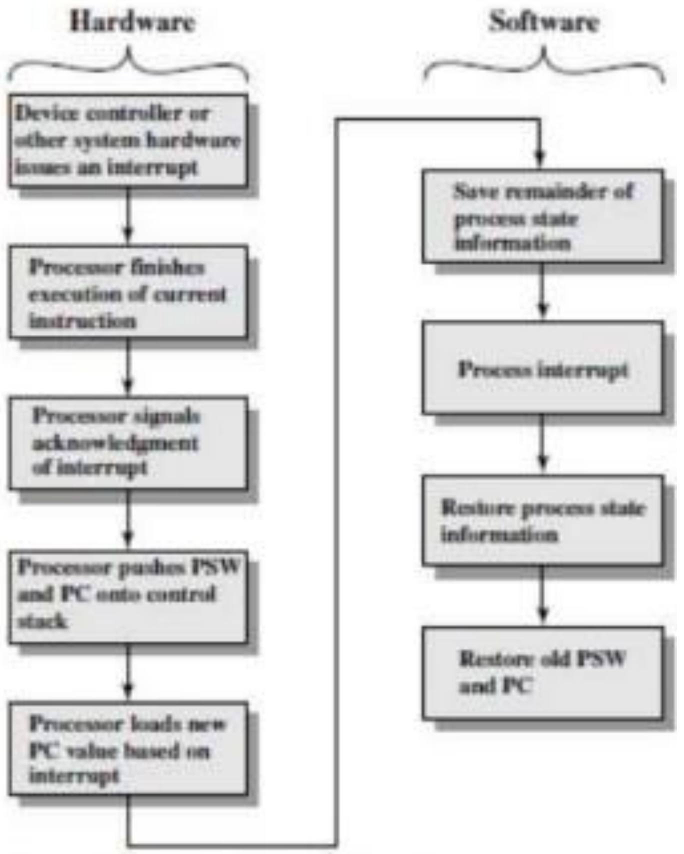

Hardware
Software
Device controller or other system hardware issues an interrupt
Processor finishes execution of current instruction
Processor signals acknowledgment of interrupt
Processor pushes PSW and PC onto control stack
Processor loads new PC value based on interrupt
Save remainder of process state information
Process interrupt
Restore process state information
Restore old PSW and PC

Figure 7.6 Simple Interrupt Processing

6.4.3 DIRECT MEMORY ACCESS

Drawbacks of Programmed and Interrupt-Driven I/O

Interrupt-driven I/O, though more efficient than simple programmed I/O, still requires the active intervention of the processor to transfer data between memory and an I/O module, and any data transfer must traverse a path through the processor.

Thus, both these forms of I/O suffer from two inherent drawbacks:

1. The I/O transfer rate is limited by the speed with which the processor can test and service a device.
2. The processor is tied up in managing an I/O transfer; a number of instructions must be executed for each I/O transfer.

There is somewhat of a trade-off between these two drawbacks. Consider the transfer of a block of data.

Using simple programmed I/O, the processor is dedicated to the task of I/O and can move data at a rather high rate, at the cost of doing nothing else.

Interrupt I/O frees up the processor to some extent at the expense of the I/O transfer rate. Nevertheless, both methods have an adverse impact on both processor activity and I/O transfer

rate.
When large volumes of data are to be moved, a more efficient technique is required: direct memory access (DMA).

DMA Function
DMA involves an additional module on the system bus.
The DMA module (Figure 7.11) is capable of mimicking the processor and, indeed, of taking over control of the system from the processor.
It needs to do this to transfer data to and from memory over the system bus. For this purpose, the DMA module must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily.
The latter technique is more common and is referred to as cycle stealing, because the DMA module in effect steals a bus cycle.
When the processor wishes to read or write a block of data, it issues a command to the DMA module, by sending to the DMA module the following information:
• Whether a read or write is requested, using the read or write control line between the processor and the DMA module
• The address of the I/O device involved, communicated on the data lines • The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register
• The number of words to be read or written, again communicated via the data lines and stored in the data count register

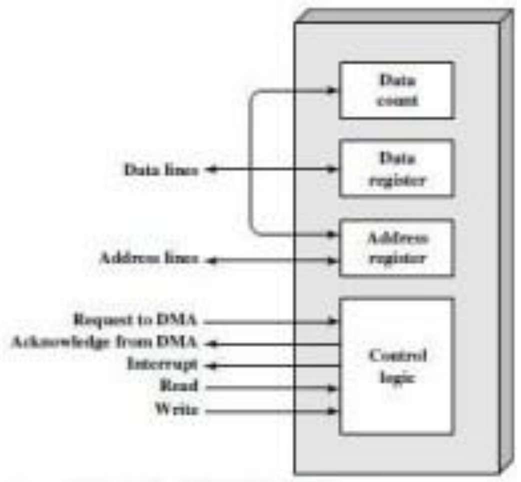
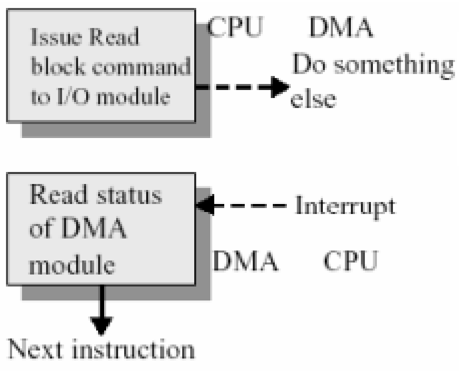


Figure 7.11 Typical DMA Block Diagram

The processor then continues with other work. It has delegated this I/O operation to the DMA module.

The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor.
When the transfer is complete, the DMA module sends an interrupt signal to the processor.
Thus, the processor is involved only at the beginning and end of the transfer (Figure 7.4c).


Issue Read block command to I/O module
CPU
DMA
Do something else
Read status of DMA module
Interrupt
DMA
CPU
Next instruction

(c) Direct memory access

Figure 7.12 shows where in the instruction cycle the processor may be suspended. In each case, the processor is suspended just before it needs to use the bus.
The DMA module then transfers one word and returns control to the processor. Note that this is not an interrupt; the processor does not save a context and do something else. Rather, the processor pauses for one bus cycle.
The overall effect is to cause the processor to execute more slowly.
Nevertheless, for a multiple-word I/O transfer, DMA is far more efficient than interrupt-driven or programmed I/O.

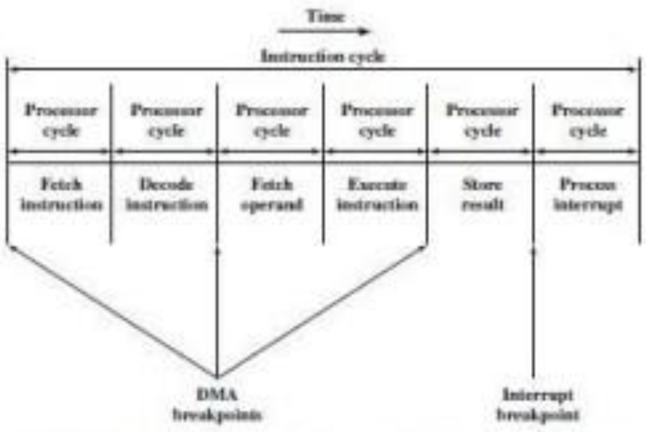


Figure 7.12 DMA and Interrupt Breakpoints during an Instruction Cycle

The DMA mechanism can be configured in a variety of ways.
Some possibilities are shown in Figure 7.13.
In the first example, all modules share the same system bus.
The DMA module, acting as a surrogate processor, uses programmed I/O to exchange data between memory and an I/O module through the DMA module.This configuration, while it may be inexpensive, is clearly inefficient.As with processor-controlled programmed I/O, each transfer of a word consumes two buscycles.The number of required bus cycles can be cut substantially by integrating the DMA and I/O functions.
As Figure 7.13b indicates, this means that there is a path between the DMA module and one or more I/O modules that does not include the system bus.The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules.This concept can be taken one step further by connecting I/O modules to the DMA module using an I/O bus (Figure 7.13c).This reduces the number of I/O interfaces in the DMA module to one and provides for an easily expandable configuration.
In both of these cases (Figures 7.13b and c), the system bus that the DMA module shares with the processor and memory is used by the DMA module only to exchange data with memory. The exchange of data between the DMA and I/O modules takes place off the system bus.

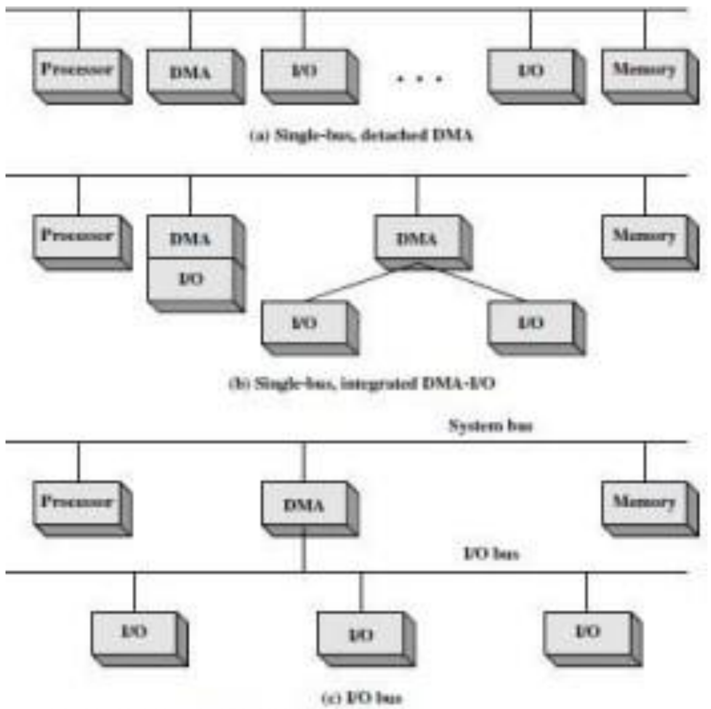


Figure 7.13 Alternative DMA Configurations

Introduction to Parallel Processing Systems

Traditionally, the computer has been viewed as a sequential machine.

Most computer programming languages require the programmer to specify algorithms as sequences of instructions.

Processors execute programs by executing machine instructions in a sequence and one at a time.Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results).This view of the computer has never been entirely true.At the micro-operation level, multiple control signals are generated at the same time.

Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time.Both of these are examples of performing functions in parallel.

This approach is taken further with superscalar organization, which exploits instruction-level parallelism.With a superscalar machine, there are multiple execution units within a single processor, and these may execute multiple instructions from the same program in parallel.
As computer technology has evolved, and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to enhance performance and, in some cases, to increase availability

Flynn's Classification

Types of Parallel Processor Systems

A taxonomy first introduced by Flynn is still the most common way of categorizing systems with parallel processing capability.
Flynn proposed the following categories of computer systems:

• ***Single instruction, single data (SISD) stream:*** A single processor executes a single instruction stream to operate on data stored in a single memory. Uniprocessors fall into this category.

• ***Single instruction, multiple data (SIMD) stream:*** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so that each instruction is executed on a different set of data by the different processors. Vector and array processors fall into this category

• ***Multiple instruction, single data (MISD) stream:*** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure is not commercially implemented.

Multiple instruction, multiple data (MIMD) stream: A set of processors simultaneously execute different instruction sequences on different data sets. SMPs (symmetric multiprocessor), clusters, and NUMA (non-uniform memory access) systems fit into this category.

With the MIMD organization, the processors are general purpose; each is able to process all of the instructions necessary to perform the appropriate data transformation.

MIMDs can be further subdivided by the means in which the processors communicate (Figure 17.1).

If the processors share a common memory, then each processor accesses programs and data stored in the shared memory, and processors communicate with each other via that memory.The most common form of such system is known as a **symmetric multiprocessor (SMP).**

In an SMP, multiple processors share a single memory or pool of memory by means of a shared bus or other interconnection mechanism; a distinguishing feature is that the memory access time to any region of memory is approximately the same for each processor.

A more recent development is the non-uniform memory access (NUMA) organization.As the name suggests, the memory access time to different regions of memory may differ for a NUMA processor.A collection of independent uniprocessors or SMPs may be interconnected to form a cluster. Communication among the computers is either via fixed paths or via some network facility.

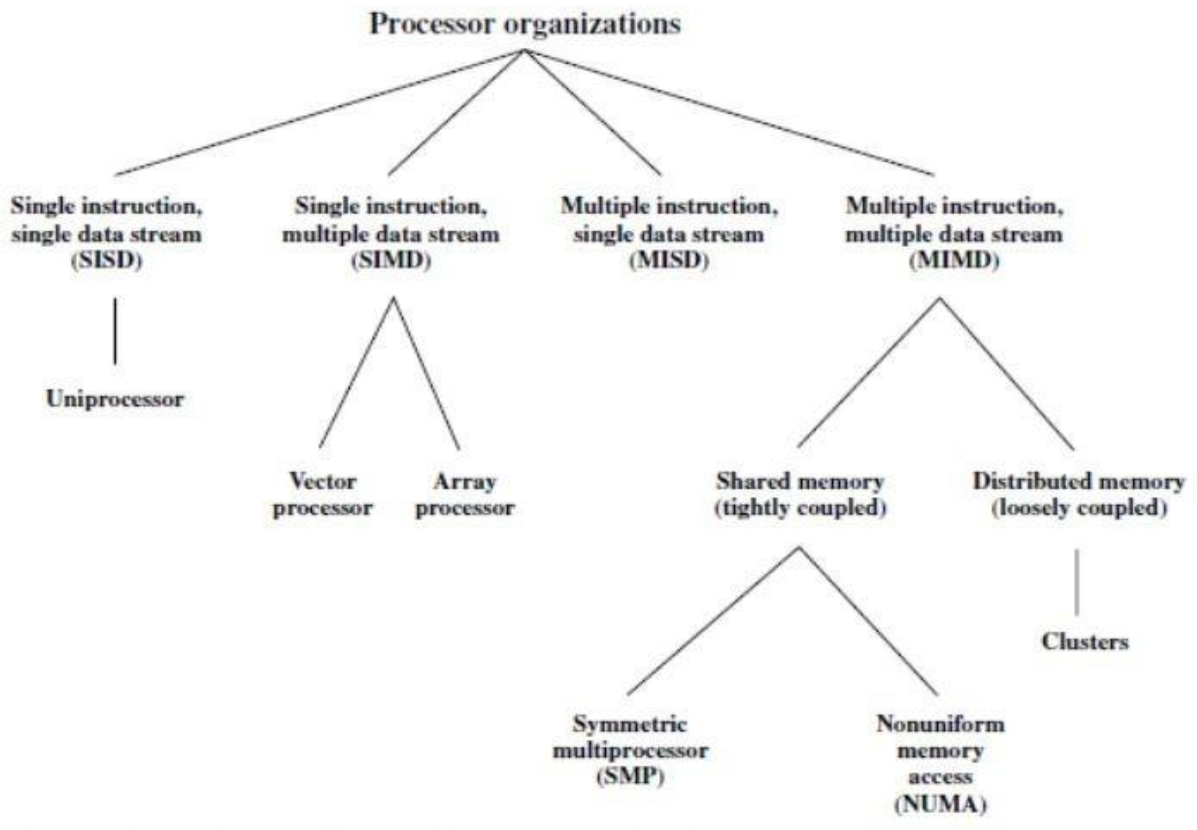

Processor organizations
Single instruction, single data stream (SISD)
Single instruction, multiple data stream (SIMD)
Multiple instruction, single data stream (MISD)
Multiple instruction, multiple data stream (MIMD)
Uniprocessor
Vector processor
Array processor
Shared memory (tightly coupled)
Distributed memory (loosely coupled)
Clusters
Symmetric multiprocessor (SMP)
Nonuniform memory access (NUMA)

Figure 17.1 A Taxonomy of Parallel Processor Architectures

Figure 17.2 illustrates the general organization of the taxonomy of Figure 17.1.

Figure 17.2a shows the structure of an SISD.

There is some sort of control unit (CU) that provides an instruction stream (IS) to a processing unit (PU).

The processing unit operates on a single data stream (DS) from a memory unit (MU).

With an SIMD, there is still a single control unit, now feeding a single instruction stream to multiple PUs.

Each PU may have its own dedicated memory (illustrated in Figure 17.2b), or there may be a shared memory.

Finally, with the MIMD, there are multiple control units, each feeding a separate instruction stream to its own PU.

The MIMD may be a shared-memory multiprocessor (Figure 17.2c) or a distributed-memory multicomputer (Figure 17.2d).

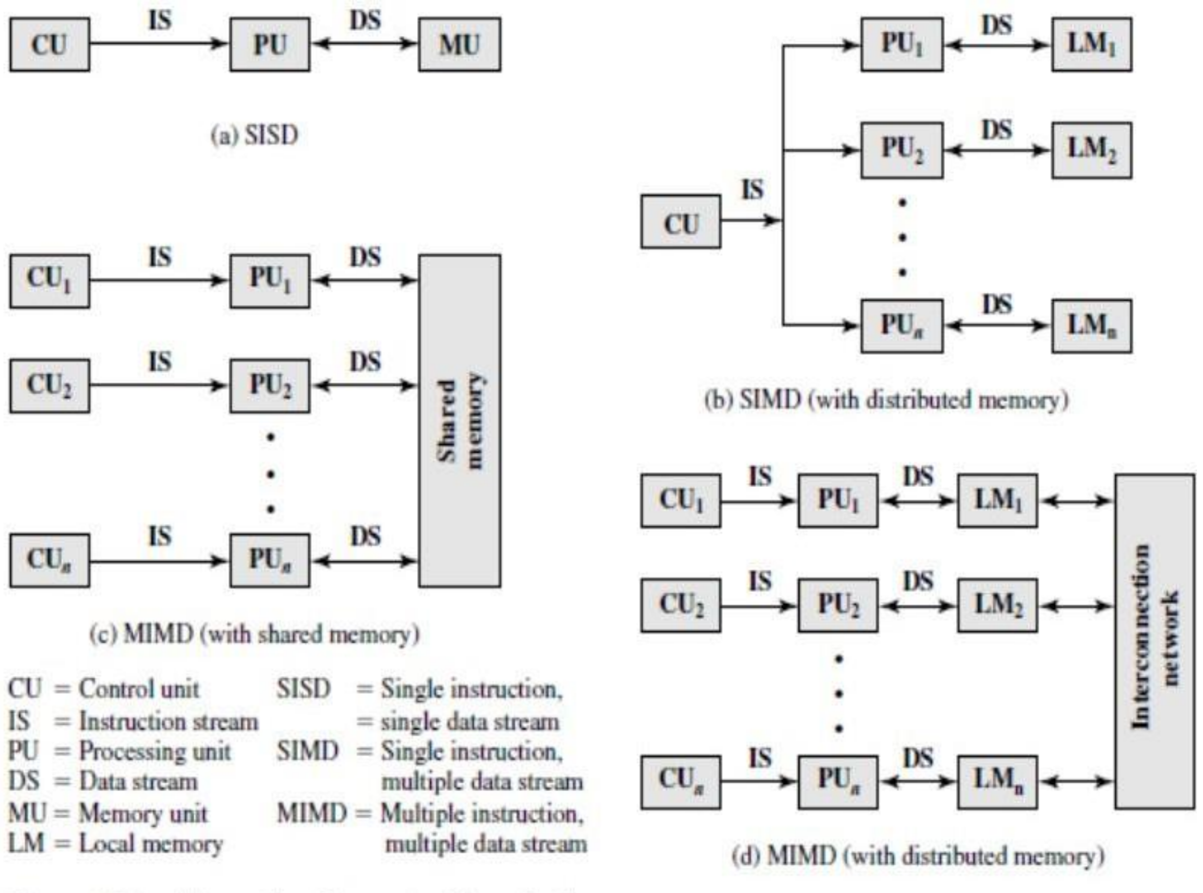

CU
IS
PU
DS
MU
(a) SISD
PU₁
LM₁
PU₂
LM₂
PUₙ
LMₙ
(b) SIMD (with distributed memory)
CU₁
CU₂
CUₙ
Shared memory
(c) MIMD (with shared memory)
Interconnection network
(d) MIMD (with distributed memory)
CU = Control unit
IS = Instruction stream
PU = Processing unit
DS = Data stream
MU = Memory unit
LM = Local memory
SISD = Single instruction, = single data stream
SIMD = Single instruction, multiple data stream
MIMD = Multiple instruction, multiple data stream

Figure 17.2 Alternative Computer Organizations

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.

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 12.5, 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 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.

Note that this approach, which involves instruction buffering, requires more registers.
In general, pipelining requires registers to store data between stages.

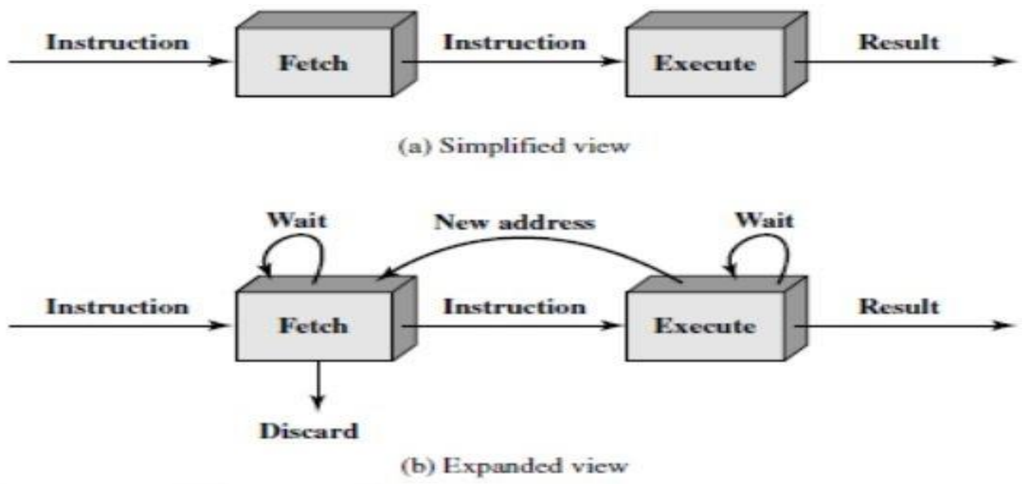
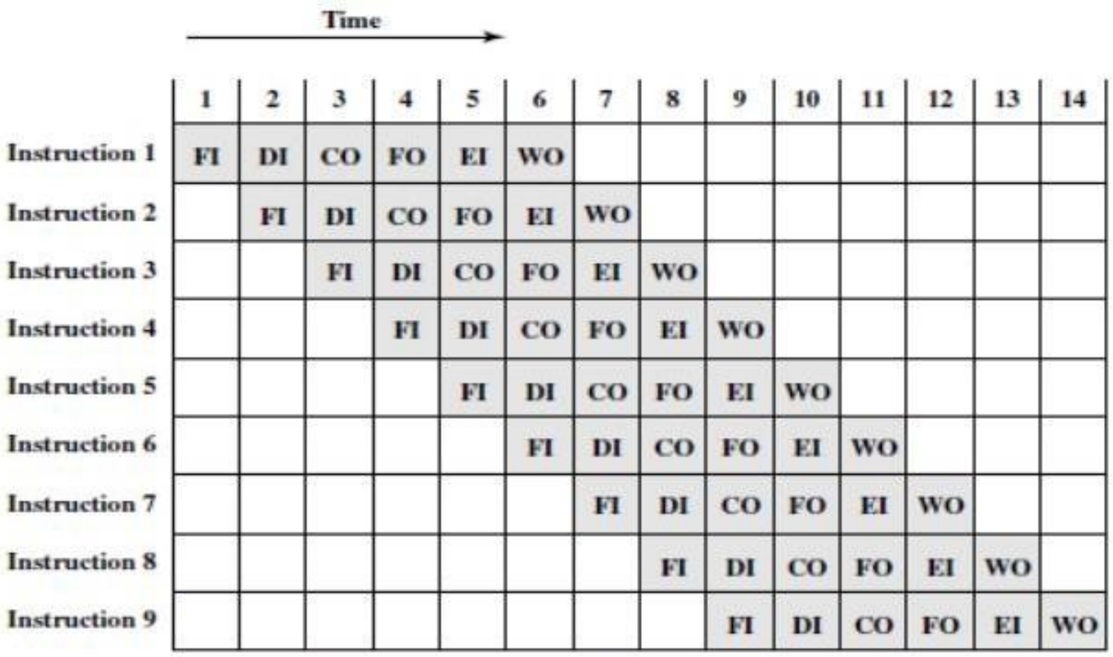


Figure 12.9 Two-Stage Instruction Pipeline

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.



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 12.10 Timing Diagram for Instruction Pipeline Operation

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.

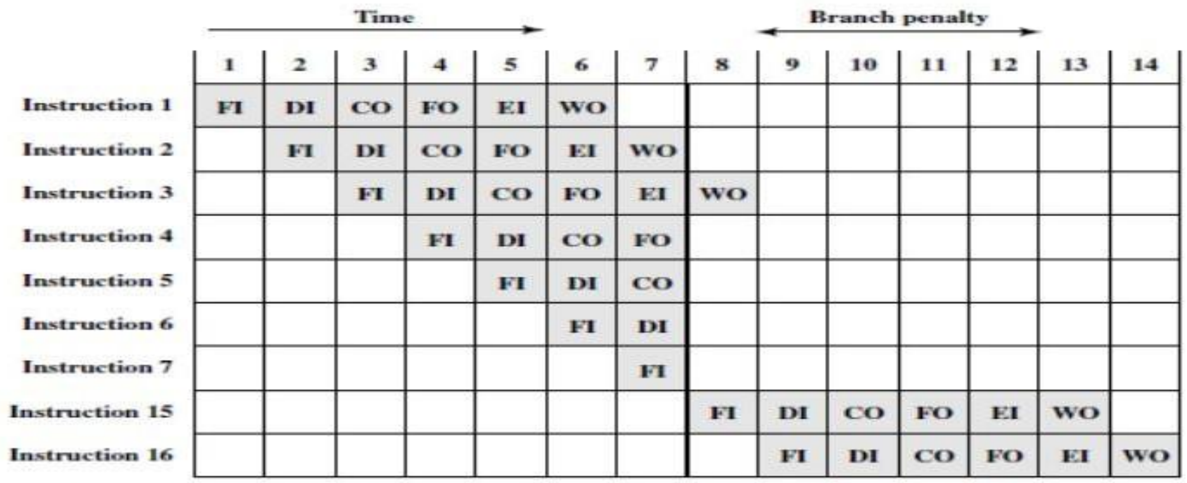
Thus, much of the time, memory conflicts will not slow down the pipeline.

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 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 we could not anticipate the branch.


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

Figure 12.12 indicates the logic needed for pipelining to account for branches and interrupts.

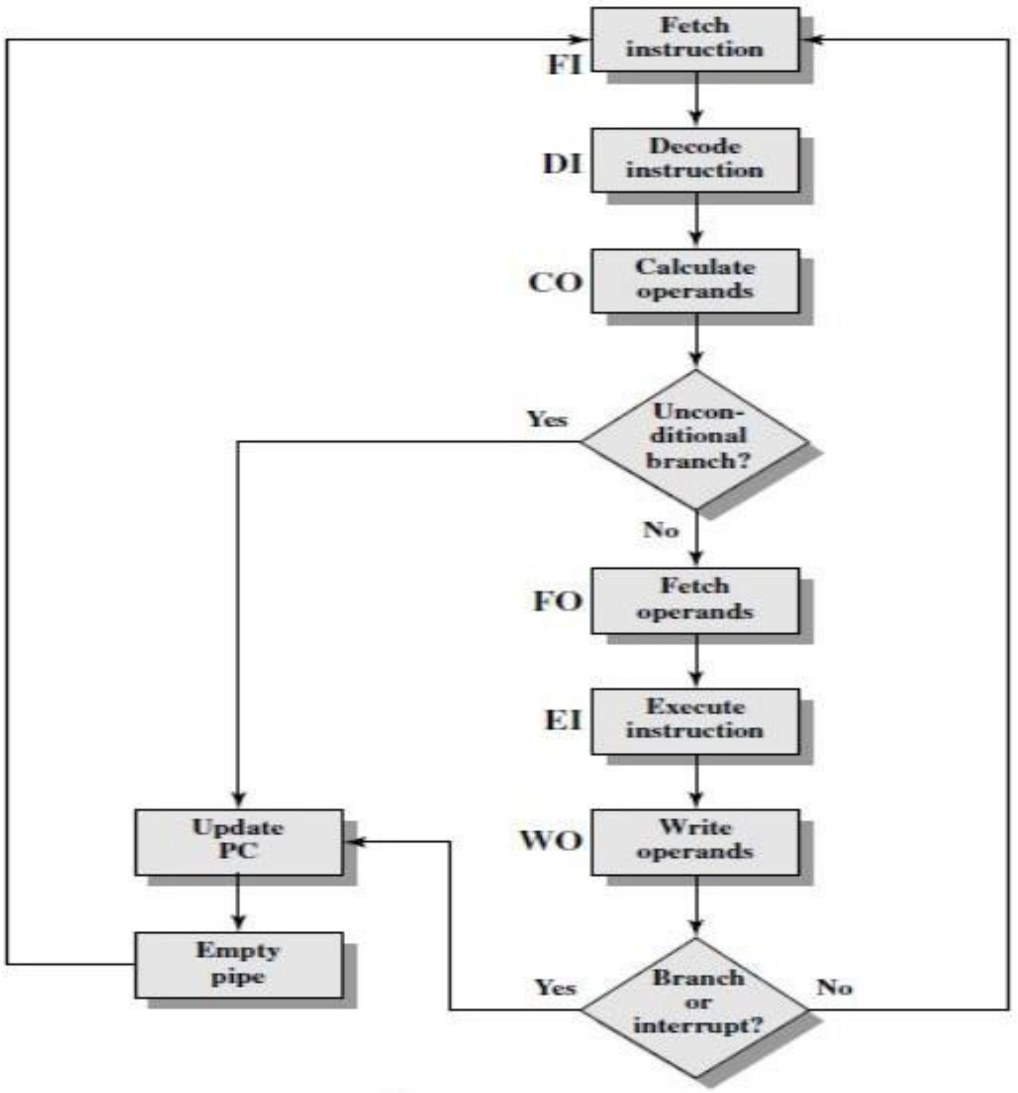


Figure 12.12 Six-Stage CPU Instruction Pipeline

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.

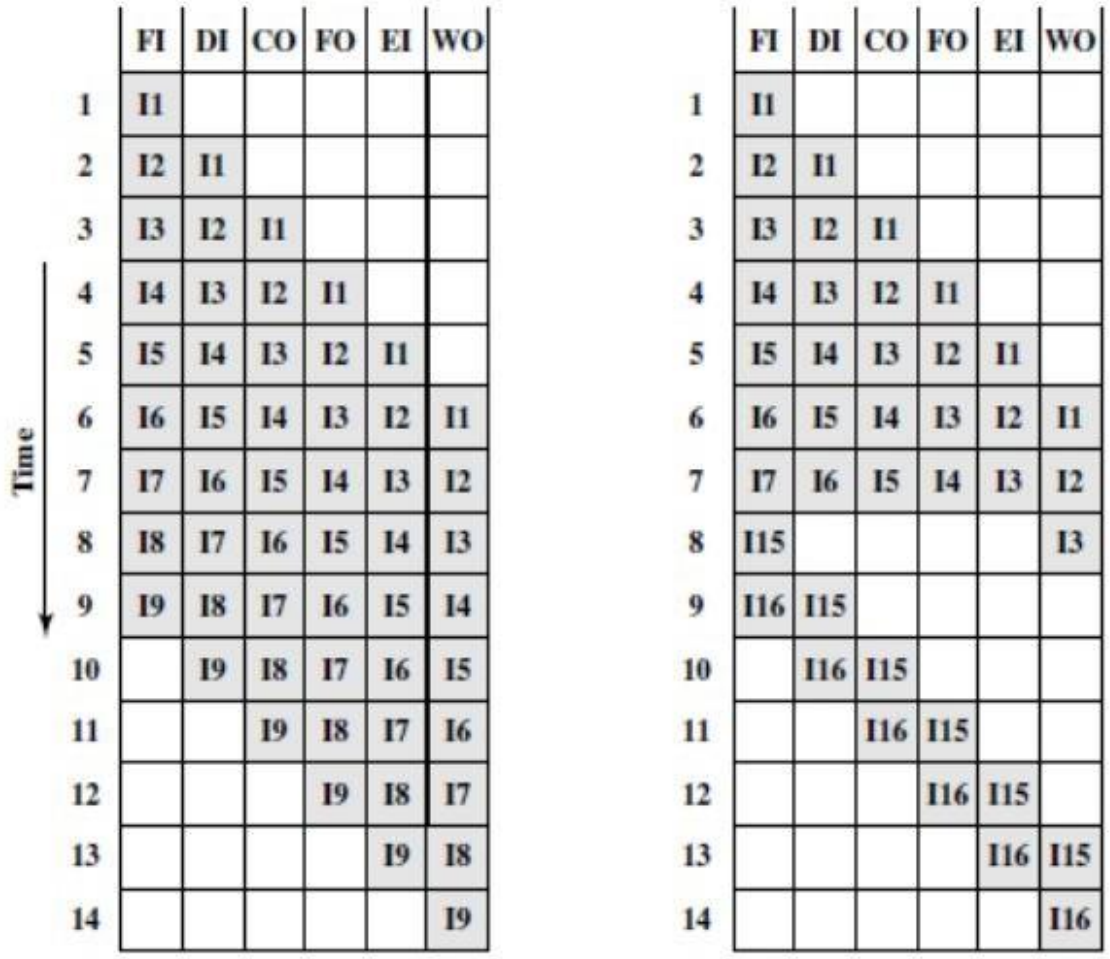
Figures 12.10 and 12.11 show the progression of time horizontally across the figures, with each row showing the progress of an individual instruction.

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 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.



	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I8	I7	I6	I5	I4	I3
9	I9	I8	I7	I6	I5	I4
10		I9	I8	I7	I6	I5
11			I9	I8	I7	I6
12				I9	I8	I7
13					I9	I8
14						I9

(a) No branches

	FI	DI	CO	FO	EI	WO
1	I1					
2	I2	I1				
3	I3	I2	I1			
4	I4	I3	I2	I1		
5	I5	I4	I3	I2	I1	
6	I6	I5	I4	I3	I2	I1
7	I7	I6	I5	I4	I3	I2
8	I15					I3
9	I16	I15				
10		I16	I15			
11			I16	I15		
12				I16	I15	
13					I16	I15
14						I16

(b) With conditional branch

Figure 12.13 An Alternative Pipeline Depiction

PIPELINE HAZARDS

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

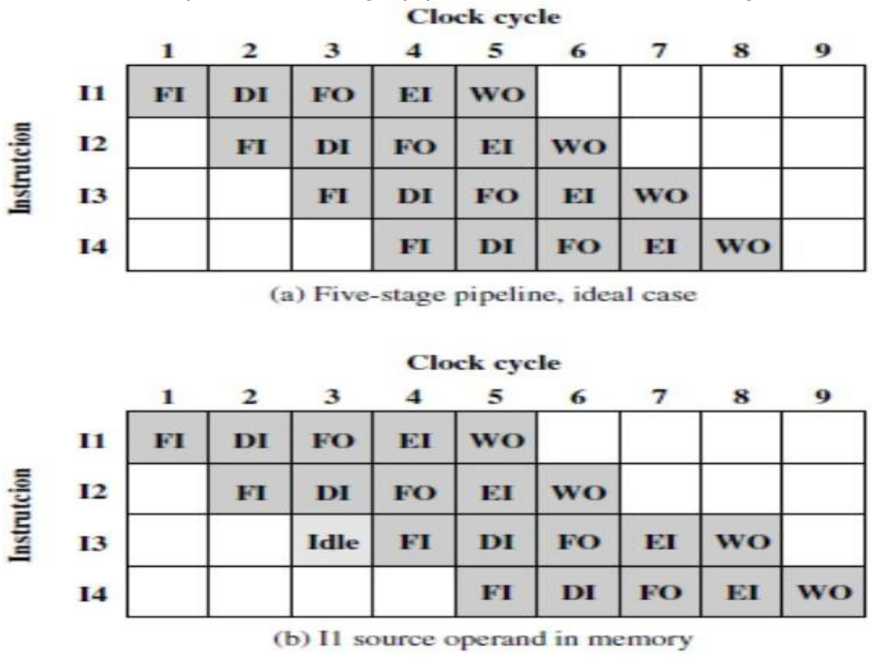
A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource.

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.



Clock cycle

Instrutcion	1	2	3	4	5	6	7	8	9
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

Instrutcion	1	2	3	4	5	6	7	8	9
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 12.15 Example of Resource Hazard

Figure 12.15a 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.

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 12.15b, 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.

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.

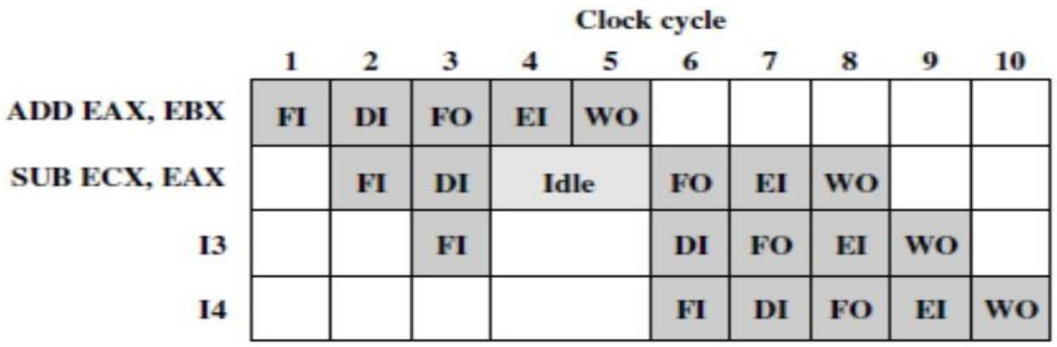
Figure 12.16 shows the pipeline behavior

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 clocks cycles.

Thus, in the absence of special hardware and specific avoidance algorithms, such a data hazard results in inefficient pipeline usage.



	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		
I3			FI			DI	FO	EI	WO	
I4						FI	DI	FO	EI	WO

Figure 12.16 Example of Data Hazard

There are three types of data hazards;

• <u>Read after write (RAW), or true dependency:</u>

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.

• <u>Write after read (WAR), or anti-dependency:</u>

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.

• <u>Write after write (WAW), or output dependency:</u>

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.

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.