# Computer Architecture

## Chapter 3 A Top-down View of Computer Function and Interconnection

Asst. Prof. Dr. Gazi Erkan BOSTANCI

ebostanci@ankara.edu.tr

Slides are mainly based on

Computer Organization and Architecture: Designing for Performance by William Stallings, 9th Edition, Prentice Hall

## Outline

1. Computer Components
2. Computer Function
3. Interconnection Structures
4. Bus Interconnection
5. Point-to-Point Interconnection

At a top level, a computer consists of CPU (central processing unit), memory, and I/O components, with one or more modules of each type. These components are interconnected in some fashion to achieve the basic function of the computer, which is to execute programs. Thus, at a top level, we can characterize a computer system by describing

1. the external behavior of each component, that is, the data and control signals that it exchanges with other components and
2. the interconnection structure and the controls required to manage the use of the interconnection structure.

## Computer Components

The design we are going to focus is referred to as *von Neumann architecture* and is based on three key concepts:

- Data and instructions are stored in a single read–write memory.
- The contents of this memory are addressable by location, without regard to the type of data contained there.
- Execution occurs in a sequential fashion (unless explicitly modified) from one instruction to the next.

The reasoning behind these concepts was previously discussed.

There is a small set of basic logic components that can be combined in various ways to store binary data and perform arithmetic and logical operations on that data.

If there is a particular computation to be performed, a configuration of logic components designed specifically for that computation could be constructed.

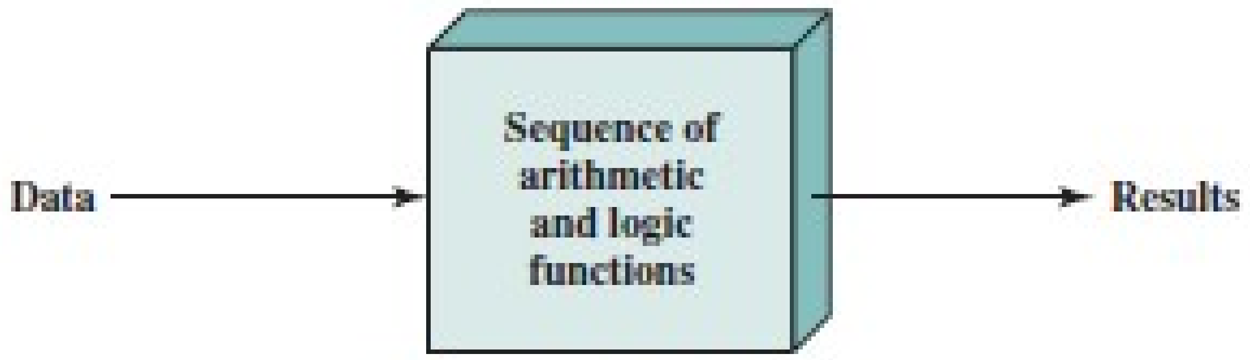
We can think of the process of connecting the various components in the desired configuration as a form of programming. The resulting “program” is in the form of hardware and is termed a *hardwired program*.

Here is an alternative:

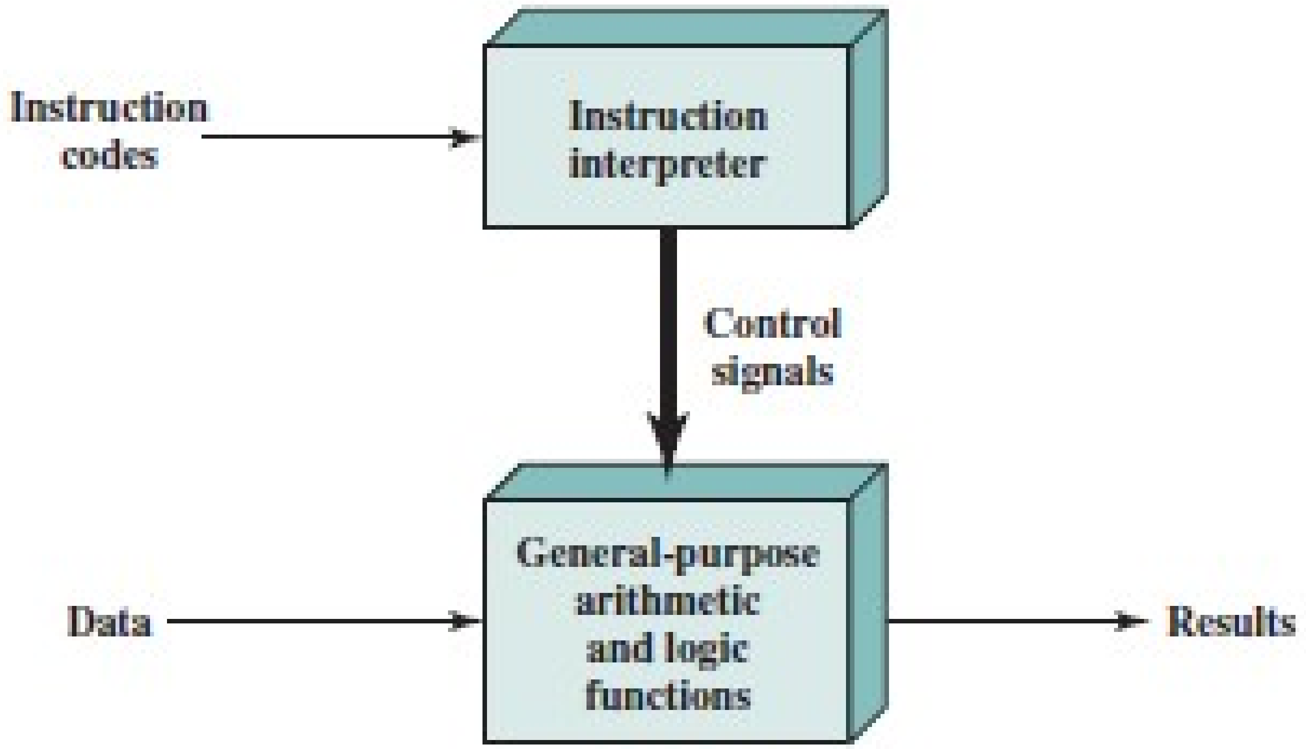
Suppose we construct a general-purpose configuration of arithmetic and logic functions. This set of hardware will perform various functions on data depending on control signals applied to the hardware.

In the original case of customized hardware, the system accepts data and produces results .

With general-purpose hardware, the system accepts data and control signals and produces results. Thus, instead of rewiring the hardware for each new program, the programmer merely needs to supply a new set of control signals.



(a) Programming in hardware



(b) Programming in software

- Then the problem is how the control signals will be supplied.
- The entire program is actually a sequence of steps.
- At each step, some arithmetic or logical operation is performed on some data. For each step, a new set of control signals is needed.
- Let us provide a unique code for each possible set of control signals, and let us add to the general-purpose hardware a segment that can accept a code and generate control signals

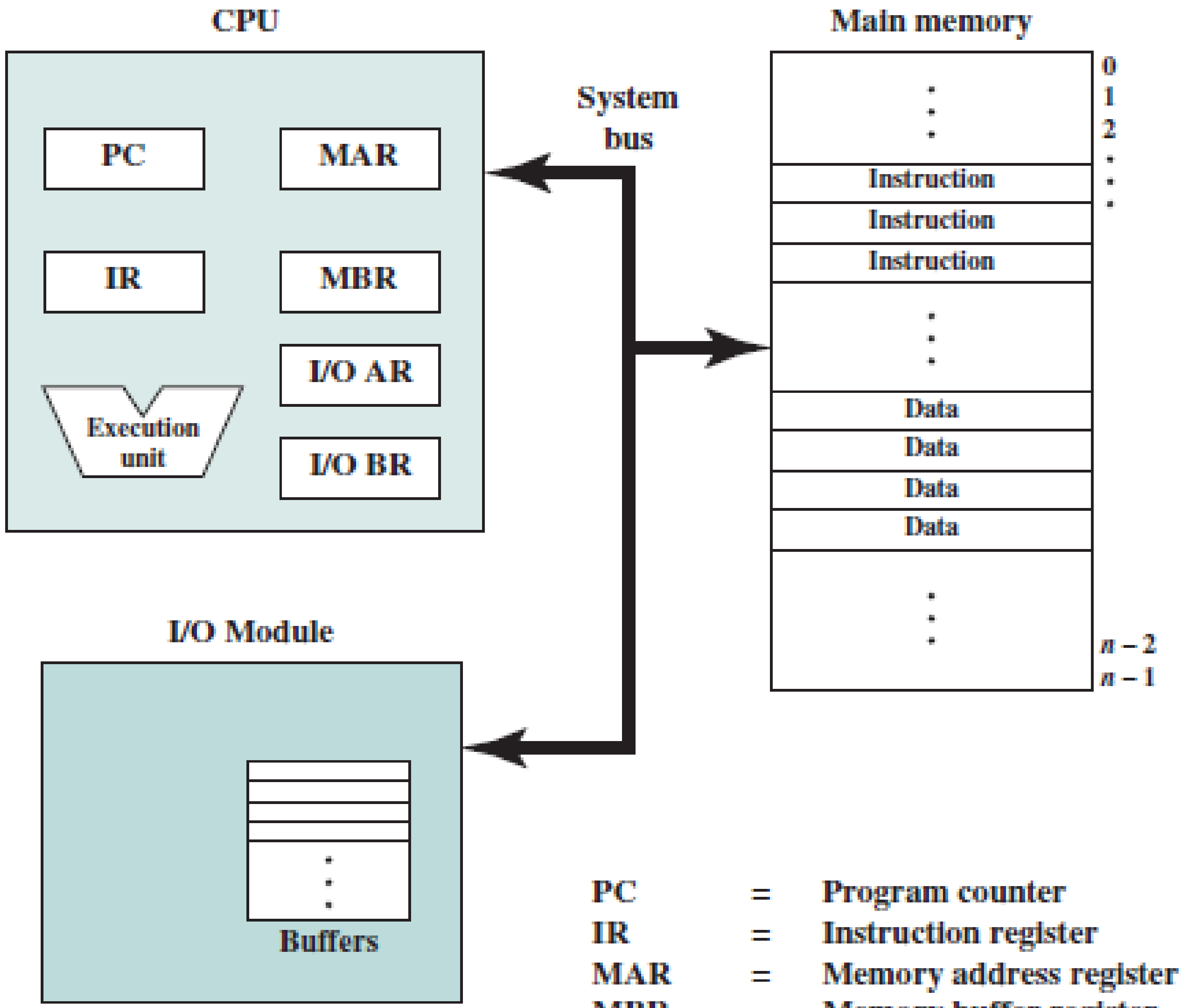
- Programming is now much easier. Instead of rewiring the hardware for each new program, all we need to do is provide a new sequence of codes.
- Each code is, in effect, an instruction, and part of the hardware interprets each instruction and generates control signals. To distinguish this new method of programming, a  sequence of codes or instructions is called *software*.

- We have two major components of the system: an instruction interpreter and a module of general-purpose arithmetic and logic functions. These two constitute the CPU.
- Several other components are needed to yield a functioning computer.
  - Data and instructions must be put into the system. For this we need some sort of input module. This module contains basic components for accepting data and instructions in some form and converting them into an internal form of signals usable by the system.
  - A means of reporting results is needed, and this is in the form of an output module. Taken together, these are referred to as *I/O components*.

- One more component is needed. An input device will bring instructions and data in sequentially. But a program is not invariably executed sequentially; it may jump around (e.g., the jump instruction). Similarly, operations on data may require access to more than just one element at a time in a predetermined sequence.
- Thus, there must be a place to store temporarily both instructions and data. That module is called *memory*, or *main memory*, to distinguish it from external storage or peripheral devices.
- Von Neumann pointed out that the same memory could be used to store both instructions and data.

- The CPU exchanges data with memory. For this purpose, it typically makes use of two internal (to the CPU) registers: a **memory address register (MAR)**, which specifies the address in memory for the next read or write, and a **memory buffer register (MBR)**, which contains the data to be written into memory or receives the data read from memory.
- Similarly, an I/O address register (I/OAR) specifies a particular I/O device. An I/O buffer (I/OBR) register is used for the exchange of data between an I/O module and the CPU.

- A memory module consists of a set of locations, defined by sequentially  numbered addresses. Each location contains a binary number that can be interpreted as either an instruction or data.
- An I/O module transfers data from external devices to CPU and memory, and vice versa. It contains internal buffers for temporarily holding these data until they can be sent on.



| | | |
|---|---|---|
| PC | = | Program counter |
| IR | = | Instruction register |
| MAR | = | Memory address register |
| MBR | = | Memory buffer register |
| I/O AR | = | Input/output address register |
| I/O BR | = | Input/output buffer register |

## Computer Function

The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory.

In its simplest form, instruction processing consists of two steps:

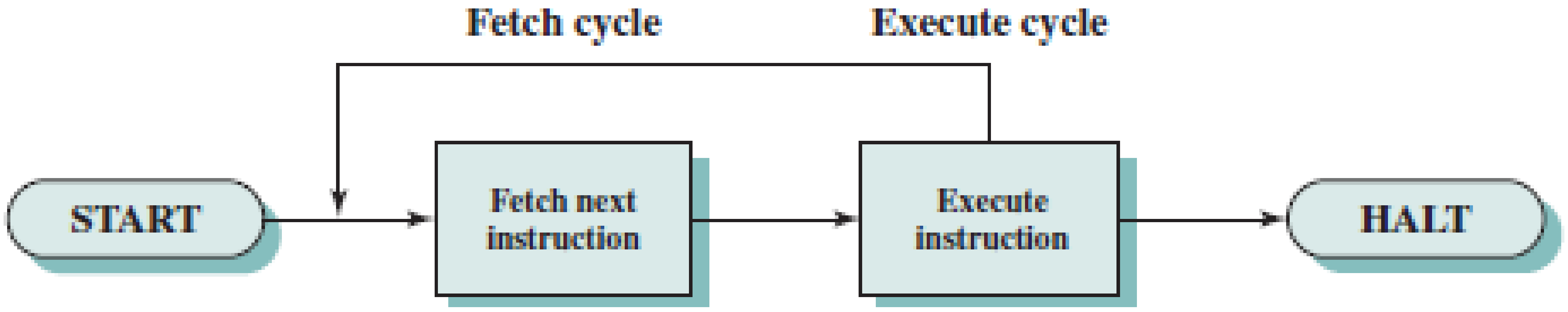
- The processor reads (*fetches*) instructions from memory one at a time and executes each instruction.
- Program execution consists of repeating the  process of instruction fetch and instruction execution. The instruction execution may involve several operations and depends on the nature of the instruction.

The processing required for a single instruction is called an **instruction cycle**.

The two steps are referred to as the **fetch cycle** and the **execute  cycle**.

Program execution halts only if the machine is turned off, some sort of unrecoverable error occurs, or a program instruction that halts the computer is encountered.

# Basic Instruction Cycle



## Instruction Fetch and Execute

- At the beginning of each instruction cycle, the processor fetches an instruction from memory. In a typical processor, a register called the program counter (PC) holds the address of the instruction to be fetched next. Unless told otherwise, the processor always increments the PC after each instruction fetch so that it will fetch the next instruction in sequence (i.e., the instruction located at the next higher memory address).

- Consider a computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to memory location 300, where the location address refers to a 16-bit word. The processor will next fetch the instruction at location 300. On succeeding instruction cycles, it will fetch instructions from locations 301, 302, 303, and so on. This sequence may be altered, as explained presently.
- The fetched instruction is loaded into a register in the processor known as the instruction register (IR). The instruction contains bits that specify the  action the processor is to take. The processor interprets the instruction and performs the required action.

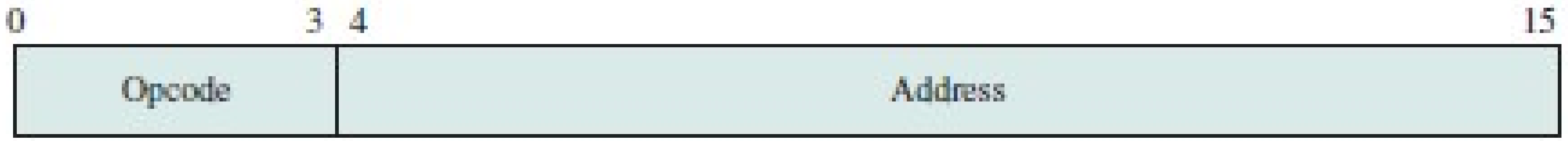
In general, these actions fall into four categories:

- **Processor-memory:** Data may be transferred from processor to memory or from memory to processor.
- **Processor-I/O:** Data may be transferred to or from a peripheral device by transferring between the processor and an I/O module.
- **Data processing:** The processor may perform some arithmetic or logic operation on data.
- **Control:** An instruction may specify that the sequence of execution be altered. For example, the processor may fetch an instruction from location 149, which specifies that the next instruction be from location 182. The processor will remember this fact by setting the program counter to 182. Thus, on the next fetch cycle, the instruction will be fetched from location 182 rather than 150.

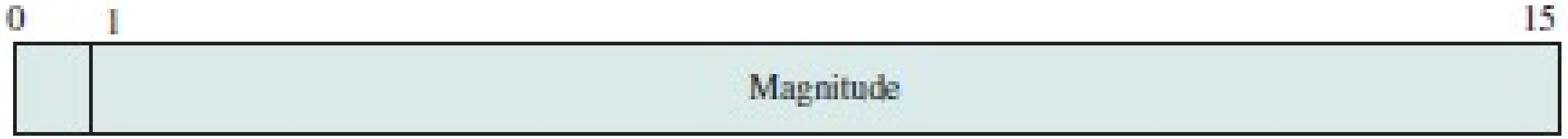
Consider a simple example using a hypothetical machine that includes the characteristics listed in the next slide.

The processor contains a single data register, called an accumulator (AC). Both instructions and data are 16 bits long.

Thus, it is convenient to organize memory using 16-bit words. The instruction format provides 4 bits for the opcode, so that there can be as many as $2^4 = 16$ different opcodes, and up to $2^{12} = 4096$ (4K) words of memory can be directly addressed.



(a) Instruction format



(b) Integer format

Program counter (PC) = Address of instruction
Instruction register (IR) = Instruction being executed
Accumulator (AC) = Temporary storage

(c) Internal CPU registers

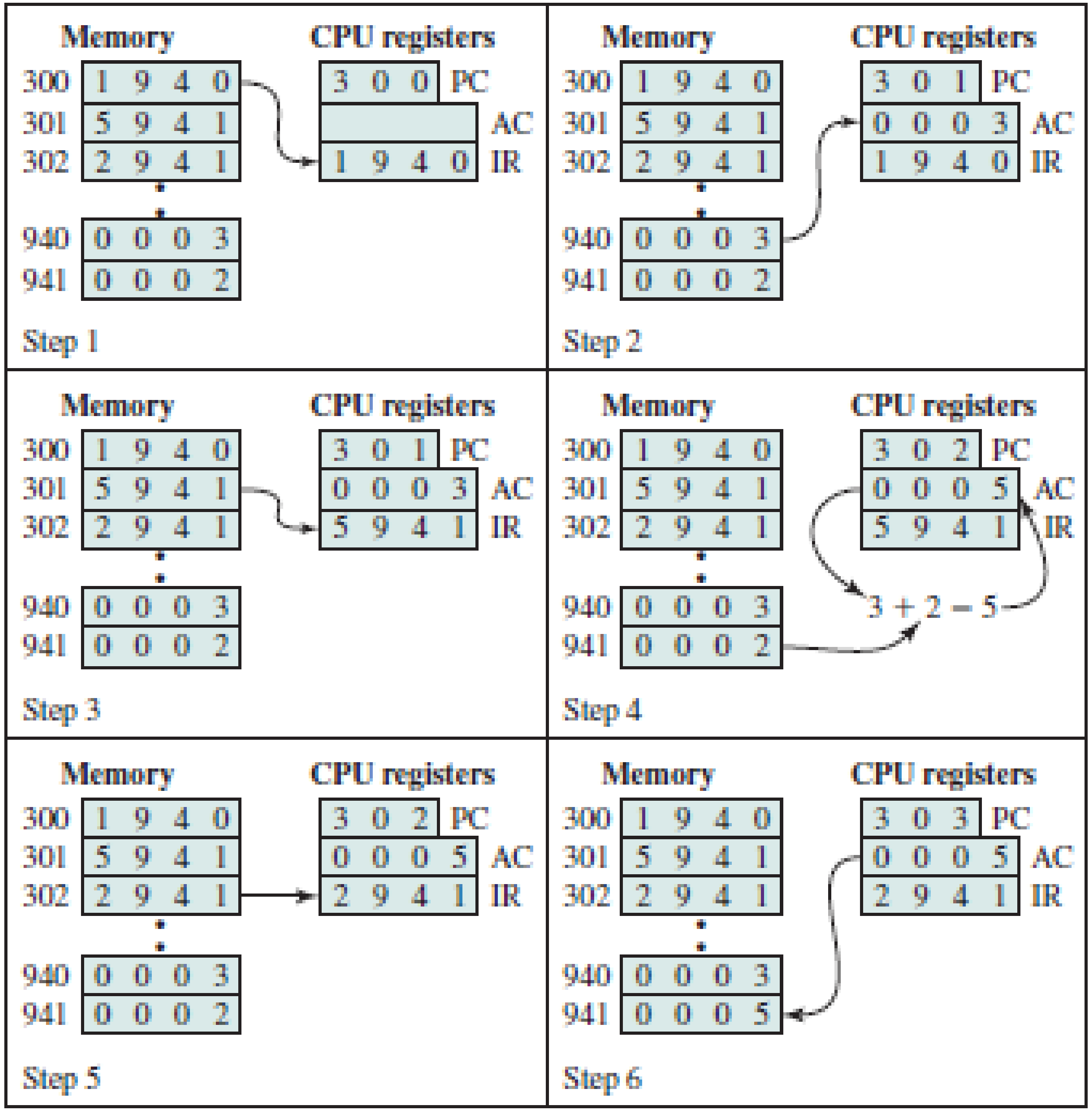
0001 = Load AC from memory
0010 = Store AC to memory
0101 = Add to AC from memory

(d) Partial list of opcodes

Now we will illustrate a partial program execution, showing the relevant portions of memory and processor registers.1 The program fragment shown adds the contents of the memory word at address 940 to the contents of the memory word at address 941 and stores the result in the latter location.

*Remember, we use hexadeciamal notation to indicate addresses.

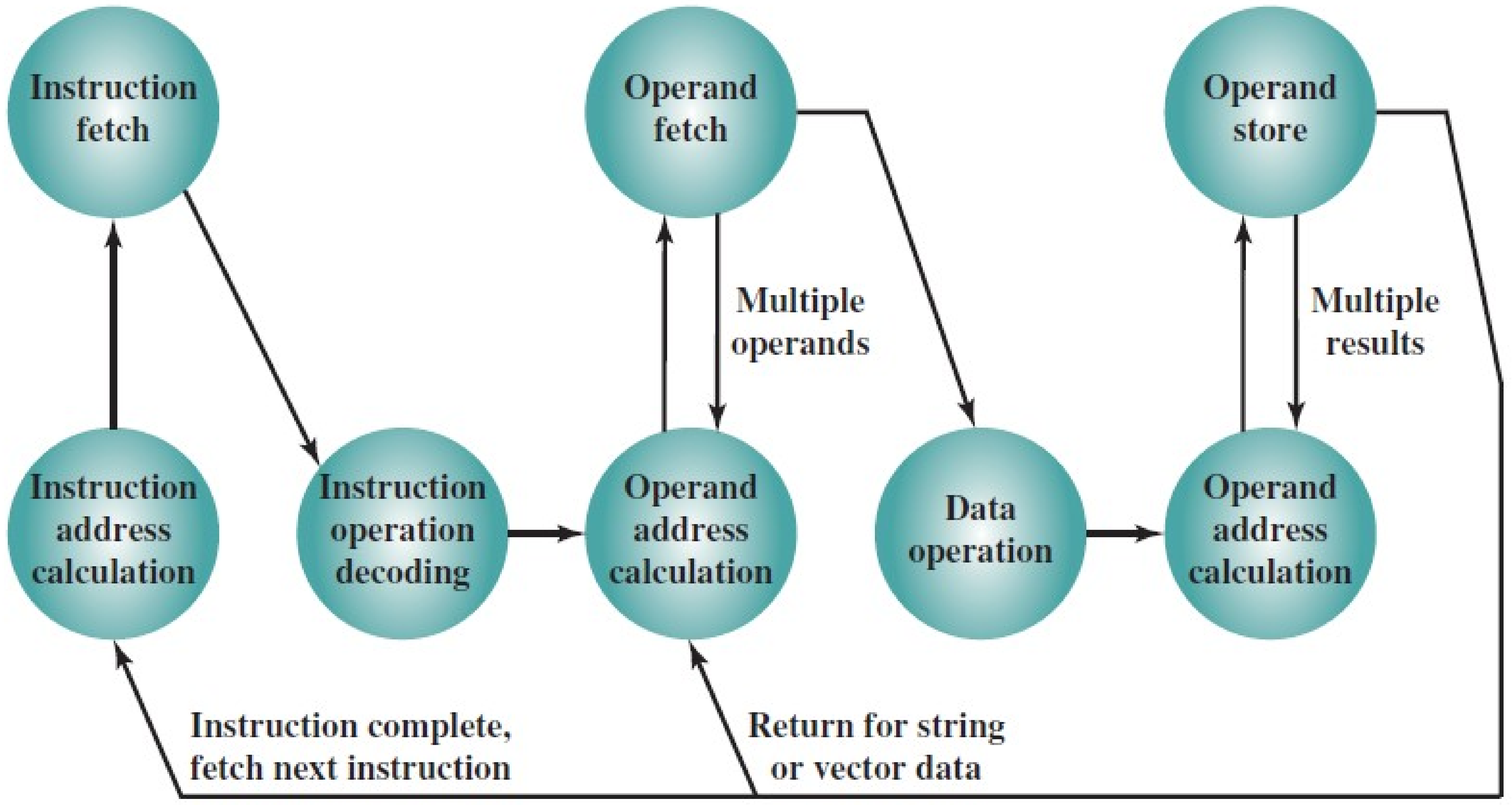
Three instructions, which can be described as three fetch and three execute cycles, are required:


Memory
CPU registers
300 1 9 4 0
301 5 9 4 1
302 2 9 4 1
940 0 0 0 3
941 0 0 0 2
PC
AC
IR
Step 1
Step 2
Step 3
Step 4
3 + 2 = 5
Step 5
Step 6
941 0 0 0 5

1. The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IR, and the PC is incremented. Note that this process involves the use of a memory address register and a memory buffer register. For simplicity, these intermediate registers are ignored.
2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded.
3. The next instruction (5941) is fetched from location 301, and the PC is incremented.
4. The old contents of the AC and the contents of location 941 are added, and the result is stored in the AC.
5. The next instruction (2941) is fetched from location 302, and the PC is incremented.
6. The contents of the AC are stored in location 941.

The execution cycle for a particular instruction may involve more than one reference to memory. Also, instead of memory references, an instruction may specify an I/O operation. With these additional considerations in mind,  we now give a more detailed look at the basic instruction cycle.

The figure is in the form of a state diagram. For any given instruction cycle, some states may be null and others may be visited more than once.


Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data

- The states can be described as follows:
  - **Instruction address calculation (iac):** Determine the address of the next instruction to be executed. Usually, this involves adding a fixed number to the address of the previous instruction. For example, if each instruction is 16 bits long and memory is organized into 16-bit words, then add 1 to the previous address. If, instead, memory is organized as individually addressable 8-bit bytes, then add 2 to the previous address.
  - **Instruction fetch (if):** Read instruction from its memory location into the processor.
  - **Instruction operation decoding (iod):** Analyze instruction to determine type of operation to be performed and operand(s) to be used.
  - **Operand address calculation (oac):** If the operation involves reference to an operand in memory or available via I/O, then determine the address of the operand.
  - **Operand fetch (of):** Fetch the operand from memory or read it in from I/O.
  - **Data operation (do):** Perform the operation indicated in the instruction.
  - **Operand store (os):** Write the result into memory or out to I/O.

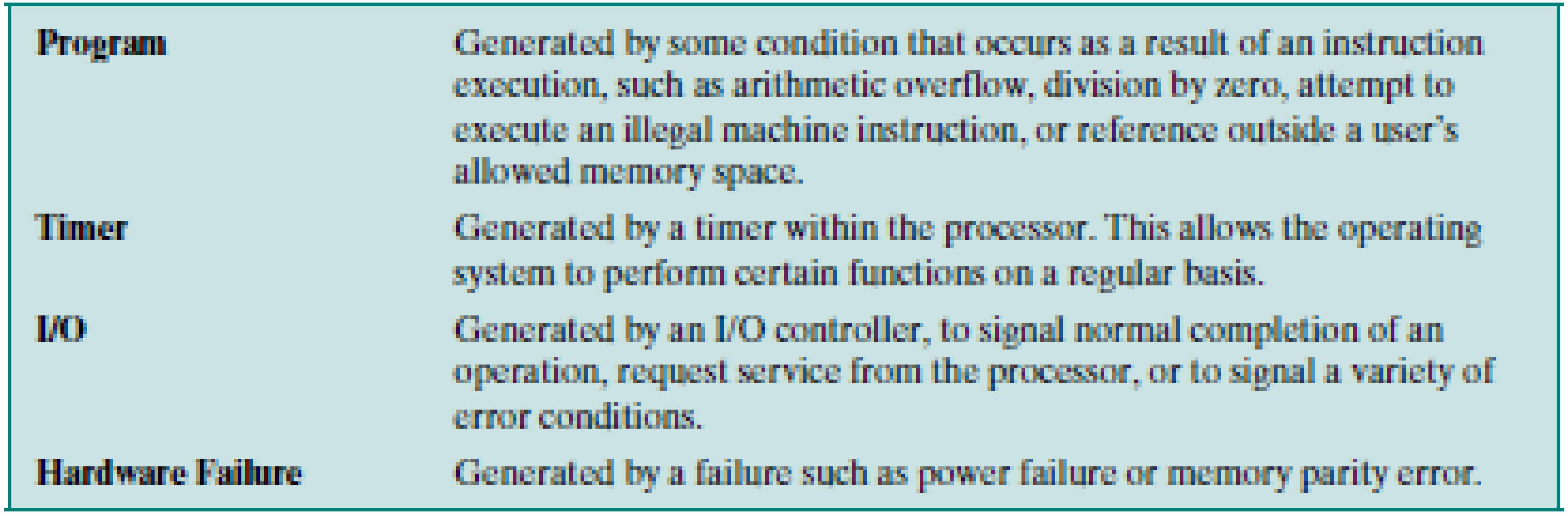
States in the upper part of the diagram involve an exchange between the processor and either memory or an I/O module. States in the lower part of the diagram involve only internal processor operations.

The oac state appears twice, because an instruction may involve a read, a write, or both. However, the action performed during that state is fundamentally the same in both cases, and so only a single state identifier is needed.

Finally, on some machines, a single instruction can specify an operation to be performed on a vector (one-dimensional array) of numbers or a string (one-dimensional array) of characters. As the diagram indicates, this would involve repetitive operand fetch and/or store operations.

## Interrupts

Virtually all computers provide a mechanism by which other modules (I/O, memory) may **interrupt** the normal processing of the processor. The most common classes of interrupts are given in the table.

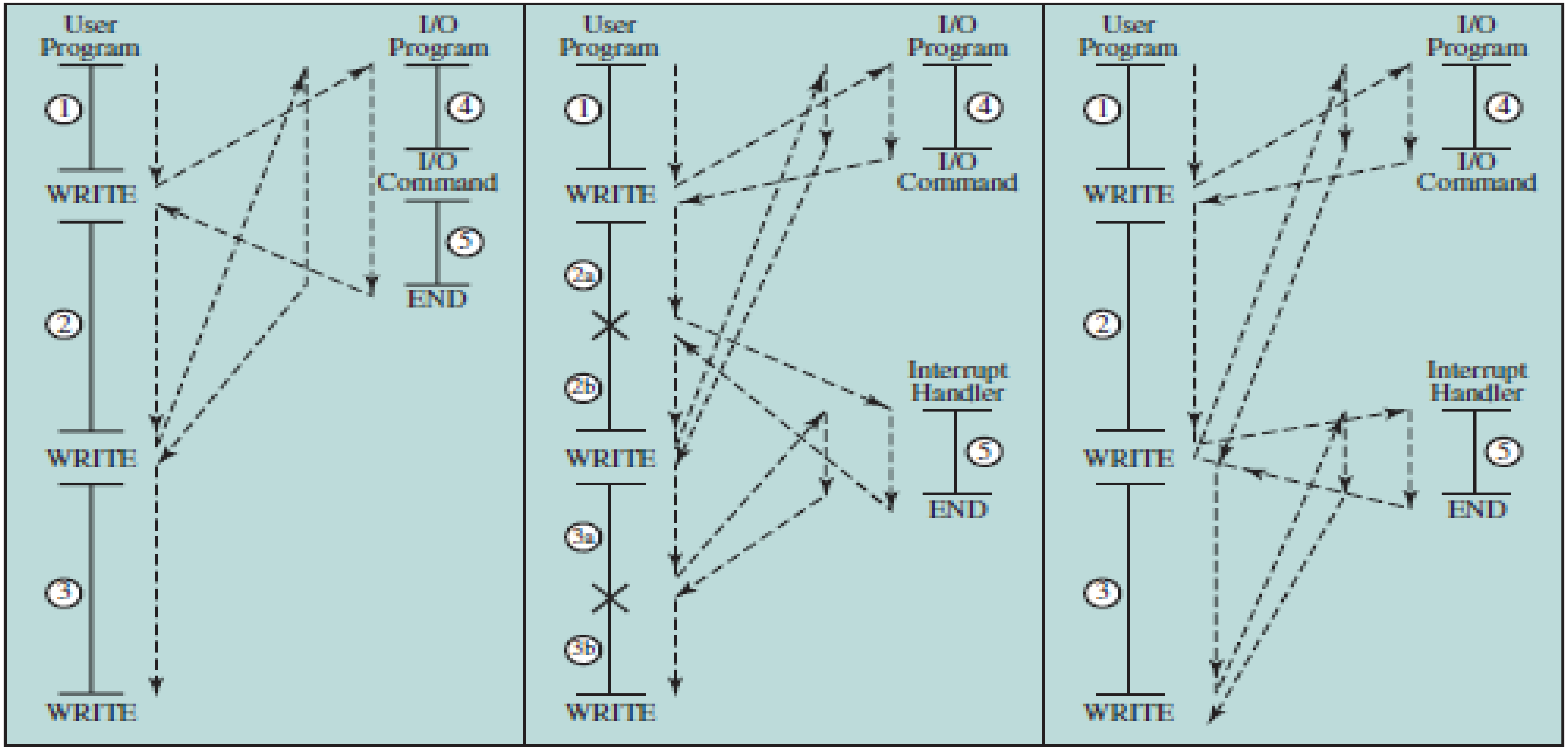


| | |
|---|---|
| **Program** | Generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space. |
| **Timer** | Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis. |
| **I/O** | Generated by an I/O controller, to signal normal completion of an operation, request service from the processor, or to signal a variety of error conditions. |
| **Hardware Failure** | Generated by a failure such as power failure or memory parity error. |

Interrupts are provided primarily as a way to improve processing efficiency.

For example, most external devices are much slower than the processor. Suppose that the processor is transferring data to a printer using the instruction cycle scheme. After each write operation, the processor must pause and remain idle until the printer catches up. The length of this pause may be on the order of many hundreds or even thousands of instruction cycles that do not involve memory.

Clearly, this is a very wasteful use of the processor.



(a) No interrupts (b) Interrupts; short I/O wait (c) Interrupts; long I/O wait

Interrupt occurs during the execution of user program

- The user program performs a series of WRITE calls interleaved with processing. Code segments 1, 2, and 3 refer to sequences of instructions that do not involve I/O. The WRITE calls are to an I/O program that is a system utility and that will perform the actual I/O operation. The I/O program consists of three sections:
  - A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation. This may include copying the data to be output into a special buffer and preparing the parameters for a device command.
  - The actual I/O command. Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically poll the device). The program might wait by simply repeatedly performing a test operation to determine if the I/O operation is done.
  - A sequence of instructions, labeled 5 in the figure, to complete the operation. This may include setting a flag indicating the success or failure of the operation.

- Because the I/O operation may take a relatively long time to complete, the I/O program is hung up waiting for the operation to complete; hence, the user program is stopped at the point of the WRITE call for some considerable period of time.

## Interrupts and The Instruction Cycle

- With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress. Consider the flow of control in (b), the short I/O wait. As before, the user program reaches a point at which it makes a system call in the form of a WRITE call. The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command.
- After these few instructions have been executed, control returns to the user program.
- Meanwhile, the external device is busy accepting data from computer memory and printing it. This I/O operation is conducted concurrently with the execution of instructions in the user program.

When the external device becomes ready to be serviced—that is, when it is ready to accept more data from the processor—the I/O module for that external device sends an *interrupt request signal to the processor.*

*The processor responds by* suspending operation of the current program, branching off to a program to service that particular I/O device, known as an **interrupt handler, and resuming the original** execution after the device is serviced.

The points at which such interrupts occur are indicated by an asterisk.

- We have a user program that contains two WRITE commands. There is a segment of code at the beginning, then one WRITE command, then a second segment of code, then a second WRITE command, then a third and final segment of code. The WRITE command invokes the I/O program provided by the OS.
- Similarly, the I/O program consists of a segment of code, followed by an I/O command, followed by another segment of code. The I/O command invokes a hardware I/O operation.

```
USER PROGRAM

⟨statement⟩ ⎫
⟨statement⟩ ⎪
     ⋮      ⎬ Code segment 1
⟨statement⟩ ⎭

WRITE

⟨statement⟩ ⎫
⟨statement⟩ ⎪
     ⋮      ⎬ Code segment 2
⟨statement⟩ ⎭

WRITE

⟨statement⟩ ⎫
⟨statement⟩ ⎪
     ⋮      ⎬ Code segment 3
⟨statement⟩ ⎭
```

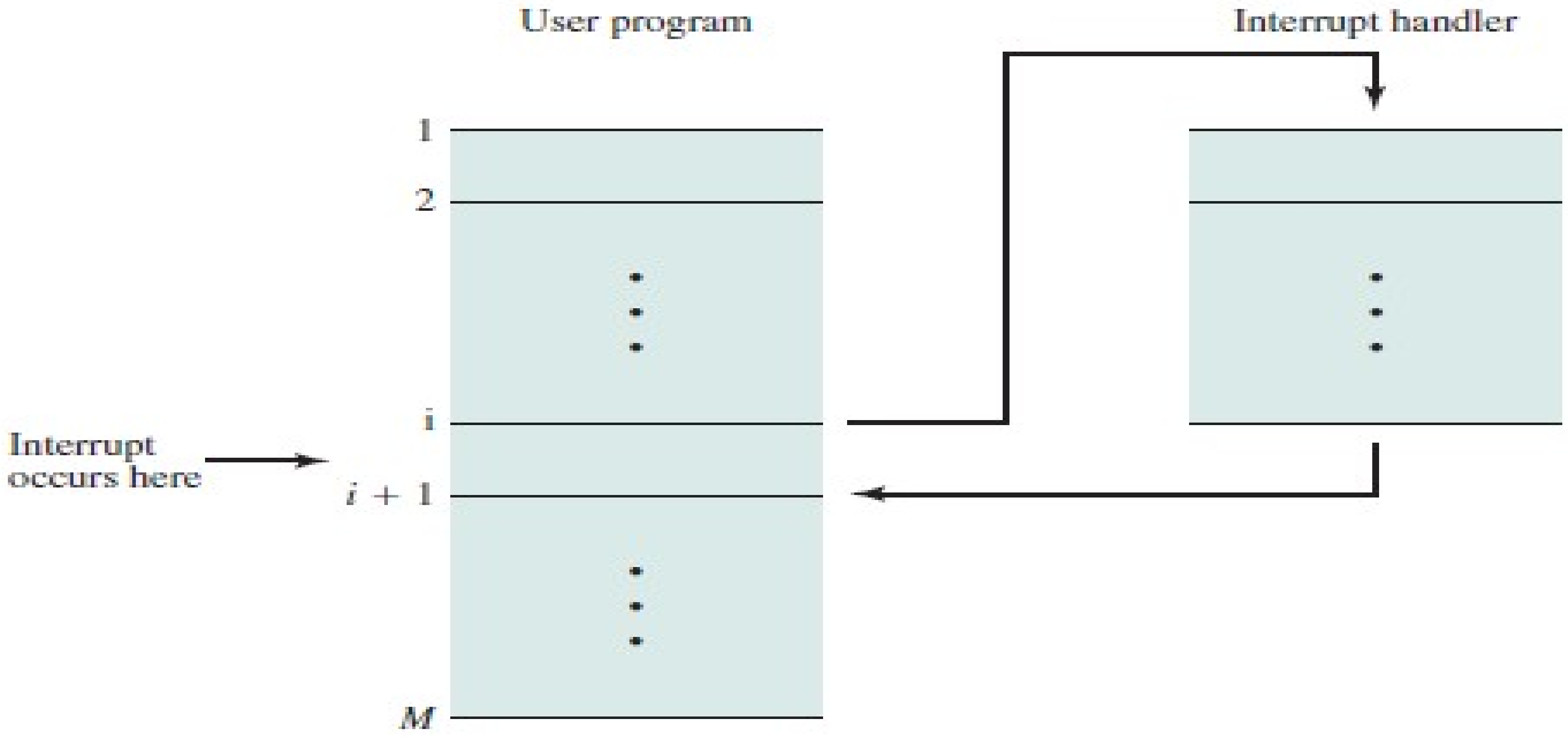
```
I/O PROGRAM

⟨statement⟩ ⎫
⟨statement⟩ ⎪
     ⋮      ⎬ Code segment 4
⟨statement⟩ ⎭

I/O command

⟨statement⟩ ⎫
⟨statement⟩ ⎪
     ⋮      ⎬ Code segment 5
⟨statement⟩ ⎭
```

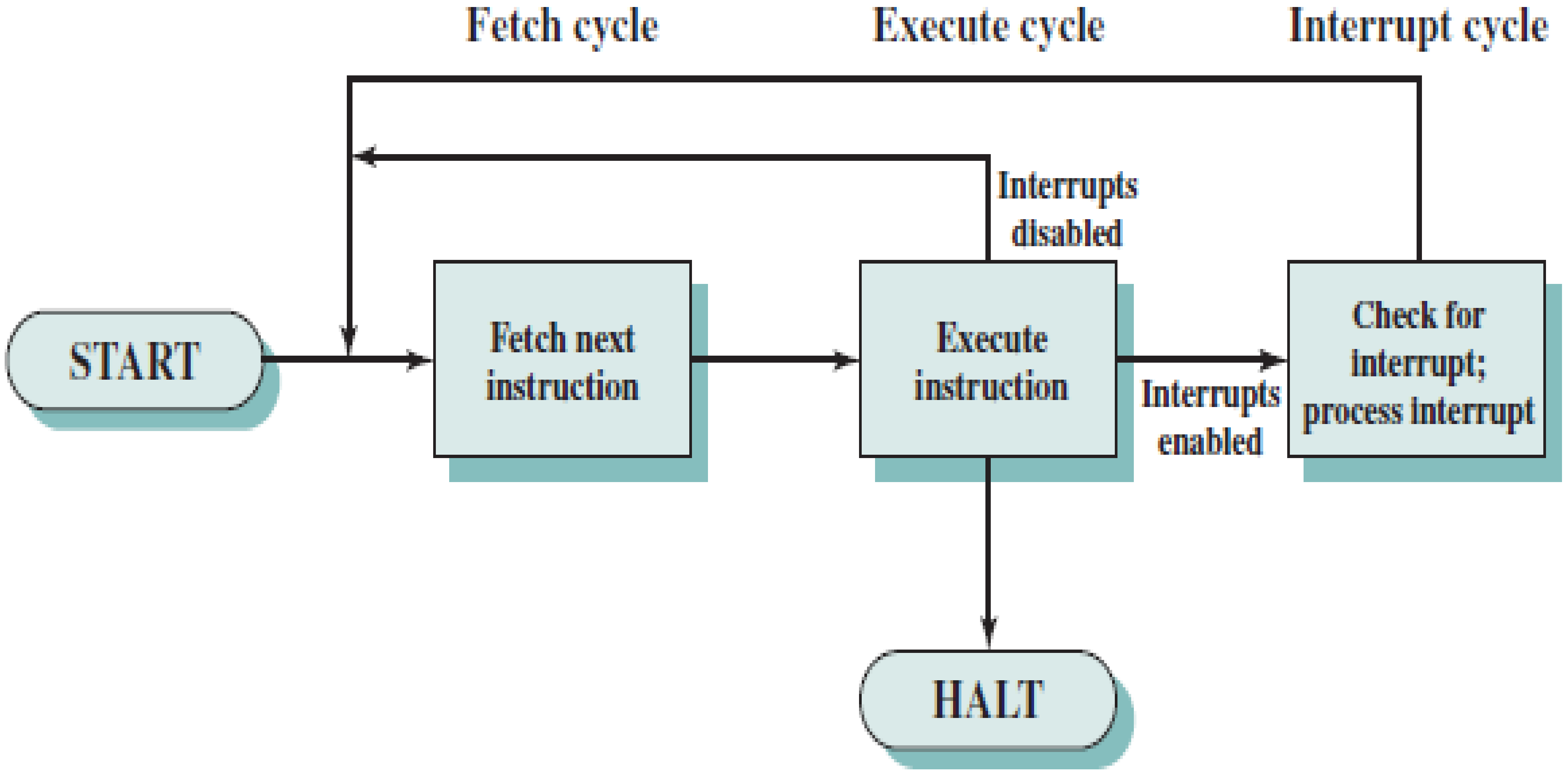
# Transfer of Control via Interrupts



From the point of view of the user program, an interrupt is just that: an interruption of the normal sequence of execution. When the interrupt processing is completed, execution resumes .

Thus, the user program does not have to contain any special code to accommodate interrupts; the processor and the operating system are responsible for suspending the user program and then resuming it at the same point.

To accommodate interrupts, an *interrupt cycle is added to the instruction* cycle


Fetch cycle
Execute cycle
Interrupt cycle
Interrupts disabled
START
Fetch next instruction
Execute instruction
Interrupts enabled
Check for interrupt; process interrupt
HALT

In the interrupt cycle, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal. If no interrupts are pending, the processor proceeds to the fetch cycle and fetches the next instruction of the current program. If an interrupt is pending, the processor does the following:

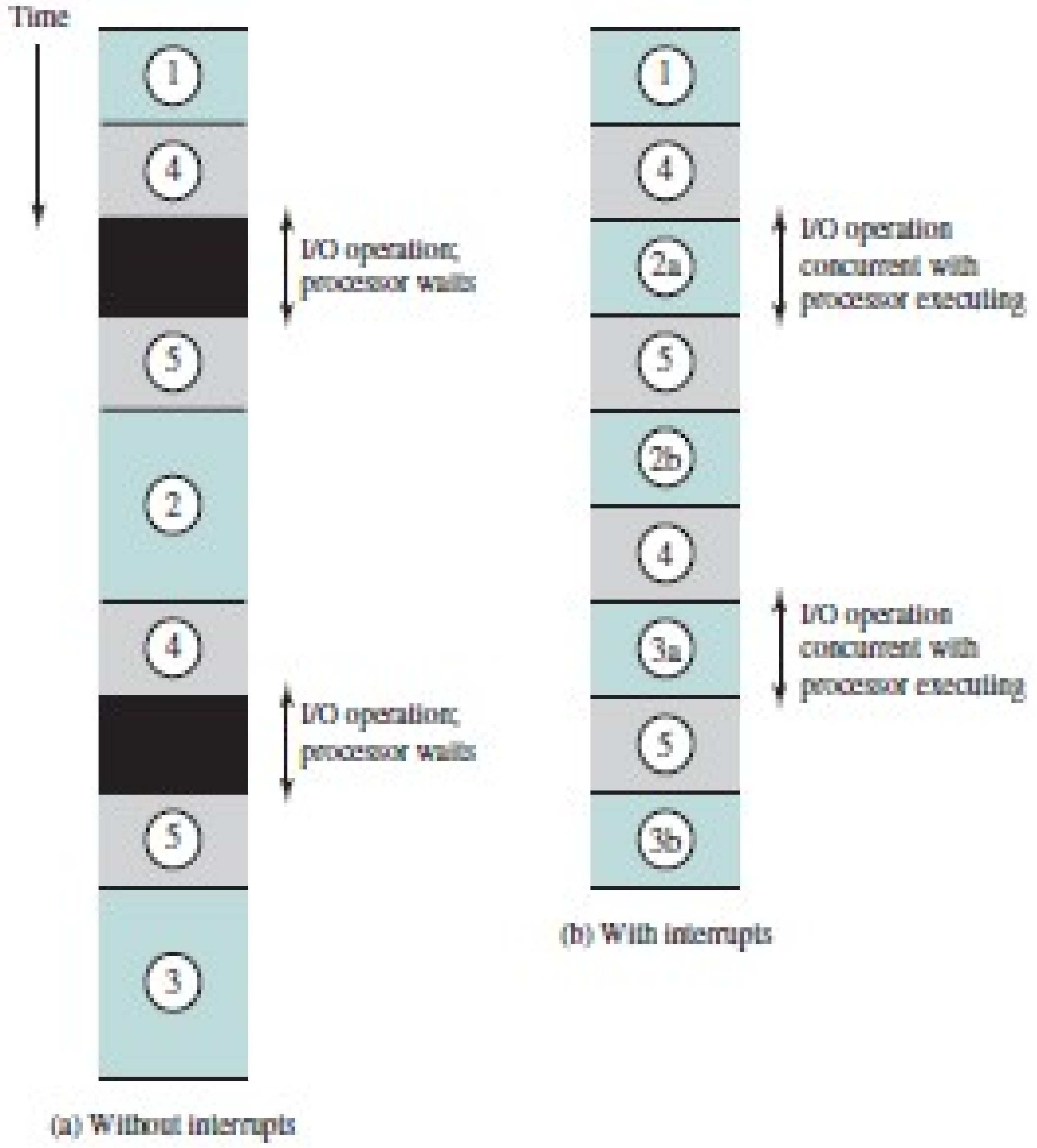
- It suspends execution of the current program being executed and saves its context. This means saving the address of the next instruction to be executed current contents of the program counter) and any other data relevant to the processor’s current activity.
- It sets the program counter to the starting address of an *interrupt handler routine.*

- The processor now proceeds to the fetch cycle and fetches the first instruction in the interrupt handler program, which will service the interrupt. The interrupt handler program is generally part of the operating system. Typically, this   program determines the nature of the interrupt and performs whatever actions are needed.
- In the example we have been using, the handler determines which I/O module generated the interrupt and may branch to a program that will write more data out to that I/O module. When the interrupt handler routine is completed, the processor can resume execution of the user program at the point of interruption.

- It is clear that there is some overhead involved in this process. Extra instructions must be executed (in the interrupt handler) to determine the nature of the interrupt  and to decide on the appropriate action.
- Nevertheless, because of the relatively large amount of time that would be wasted by simply waiting on an I/O operation, the processor can be employed much more efficiently with the use of interrupts.

## Short I/O Wait

- To appreciate the gain in efficiency, consider the figure on the right, which is a timing diagram based on the flow of control in previous figures.
- Here, user program code segments are shaded green, and I/O program code segments are shaded gray.



Time
1
4
I/O operation; processor waits
5
2
4
I/O operation; processor waits
5
3
(a) Without interrupts
1
4
I/O operation concurrent with processor executing
2a
5
2b
4
I/O operation concurrent with processor executing
3a
5
3b
(b) With interrupts

This figure assumes that the time required for the I/O operation is relatively short: less than the time to complete the execution of instructions between write operations in the user program.

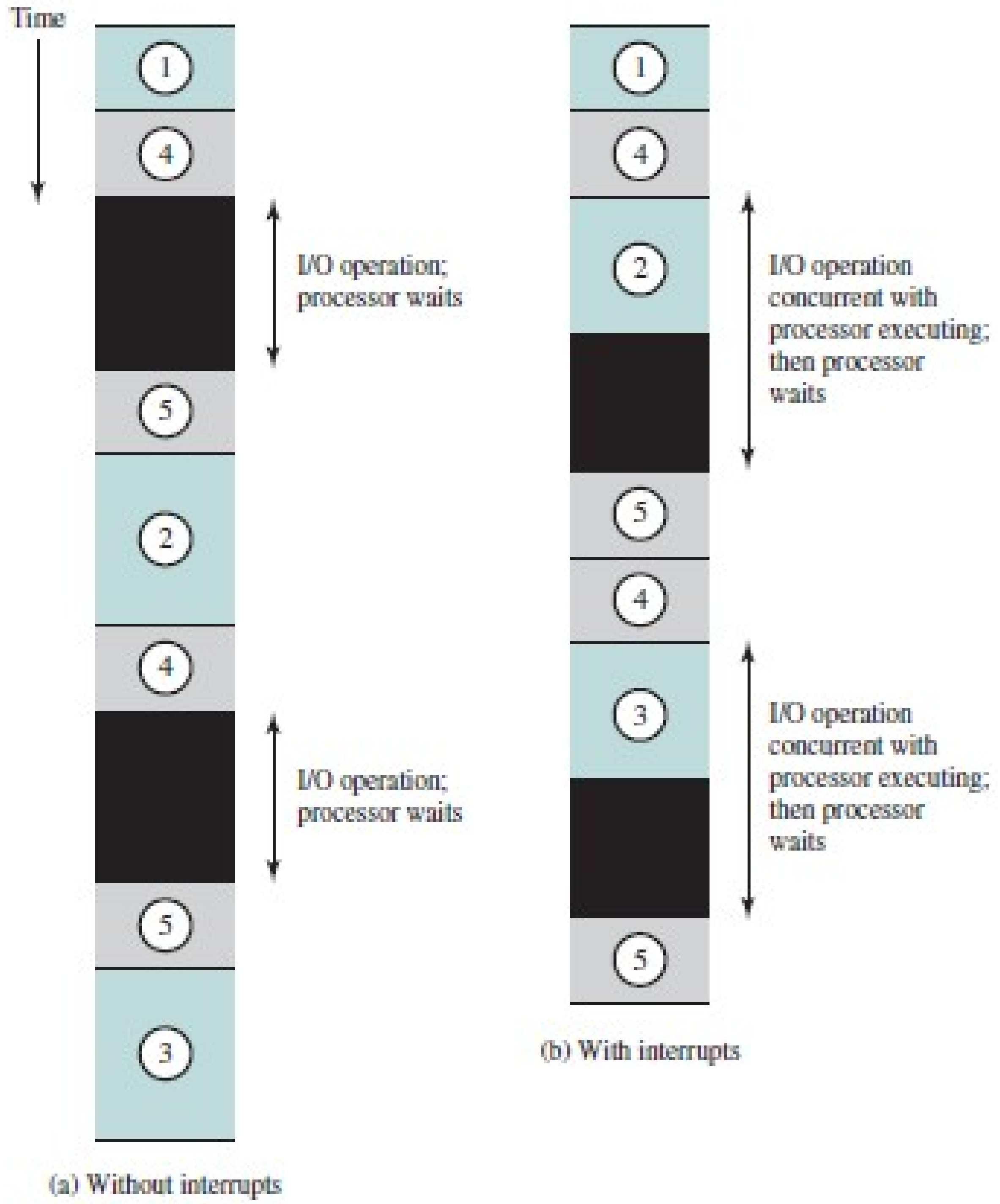
In this case, the segment of code labeled code segment 2 is interrupted. A portion of the code (2a) executes (while the I/O operation is performed) and then the interrupt occurs (upon the completion of the I/O operation). After the interrupt is serviced, execution resumes with the remainder of code segment 2 (2b).

The more typical case, especially for a slow device such as a printer, is that the I/O operation will take much more time than executing a sequence of user instructions.

In this case, the user program reaches the second WRITE call before the I/O operation spawned by the first call is complete. The result is that the user program is hung up at that point. When the preceding I/O operation is completed, this new WRITE call may be processed, and a new I/O operation may be started.

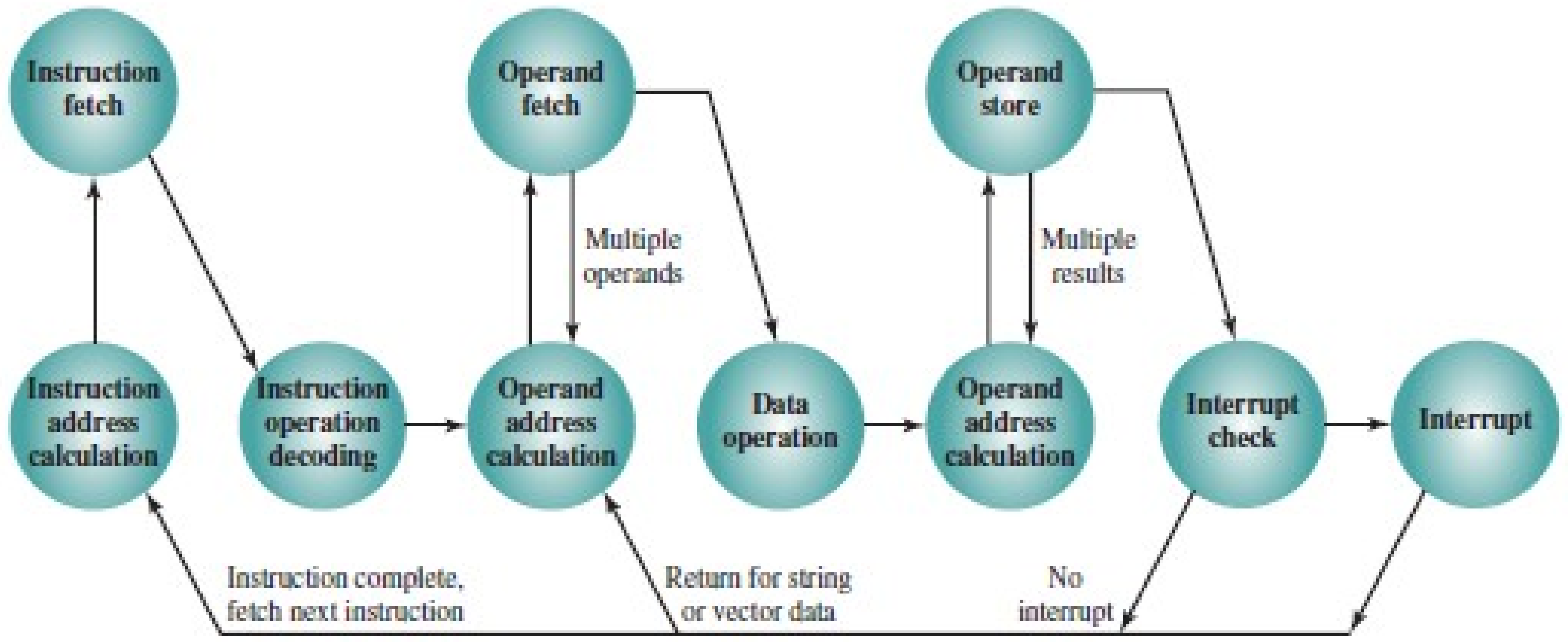
## Long I/O Wait

- We can see that there is still a gain in efficiency because part of the time during which the I/O operation is under way overlaps with the execution of user instructions.



Time
1
4
I/O operation; processor waits
5
2
4
I/O operation; processor waits
5
3
(a) Without interrupts
1
4
2
I/O operation concurrent with processor executing; then processor waits
5
4
3
I/O operation concurrent with processor executing; then processor waits
5
(b) With interrupts

# Instruction Cycle State Diagram with Interrupts


Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data operation
Operand address calculation
Interrupt check
Interrupt
Instruction complete, fetch next instruction
Return for string or vector data
No interrupt

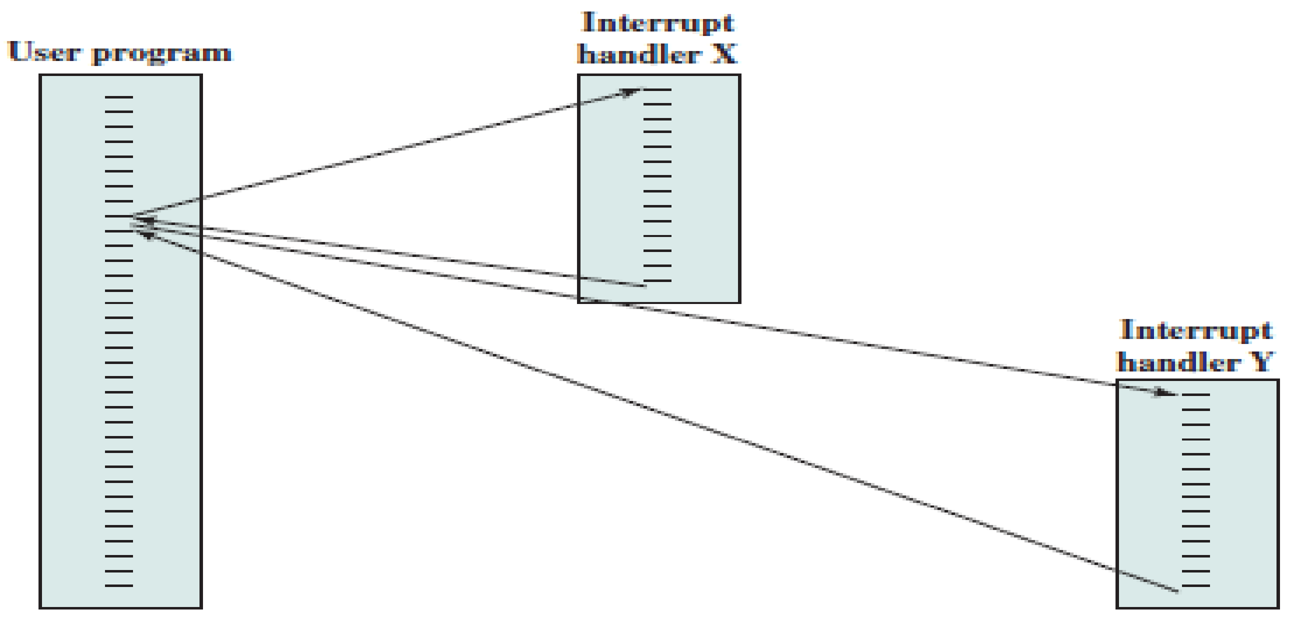
**Multiple Interrupts**

The discussion so far has focused only on the occurrence of a single interrupt. Suppose, however, that multiple interrupts can occur. For example, a program may be receiving data from a communications line and printing results. The printer will generate an interrupt every time it completes a print operation. The communication line controller will generate an interrupt every time a unit of data arrives.

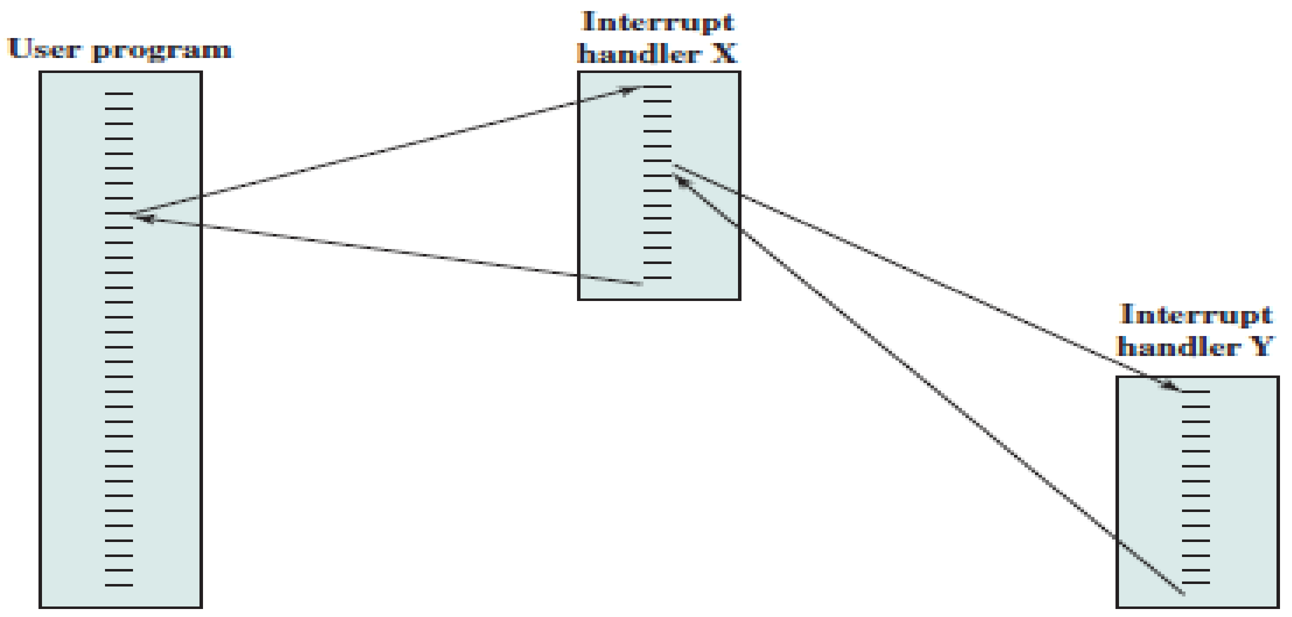
The unit could either be a single character or a block, depending on the nature of the communications discipline. In any case, it is possible for a communications interrupt to occur while a printer interrupt is being processed.

Two approaches can be taken to dealing with multiple interrupts. The first is to disable interrupts while an interrupt is being processed. A **disabled interrupt** simply means that the processor can and will ignore that interrupt request signal. If an interrupt occurs during this time, it generally remains pending and will be checked by the processor after the processor has enabled interrupts. Thus, when a user program is executing and an interrupt occurs, interrupts are disabled immediately. After the interrupt handler routine completes, interrupts are enabled before resuming the user program, and the processor checks to see if additional interrupts have occurred.

This approach is nice and simple, as interrupts are handled in strict sequential order as in (a) in the figure below:



(a) Sequential interrupt processing



(b) Nested interrupt processing

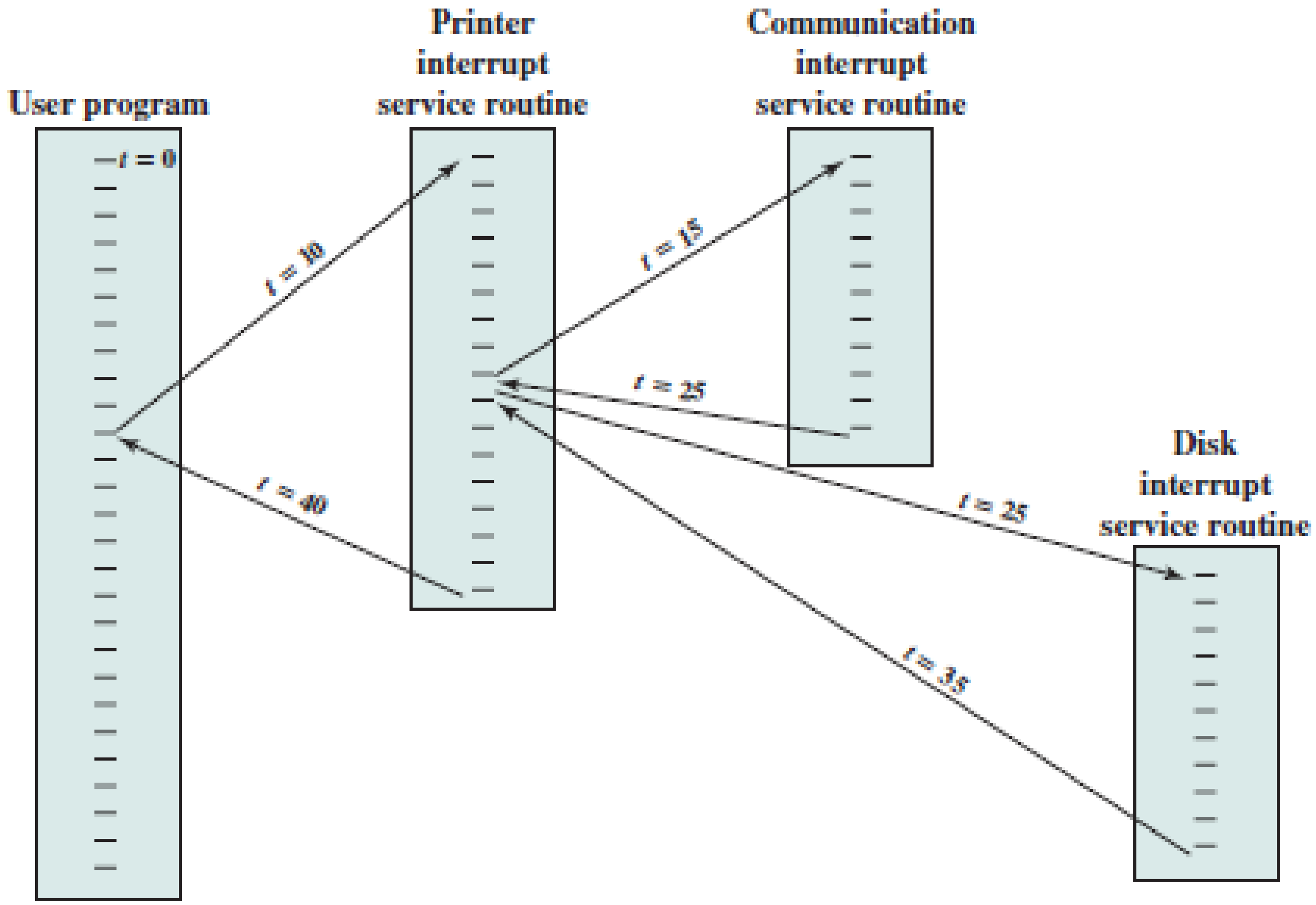
The drawback to the preceding approach is that it does not take into account relative priority or time-critical needs.

For example, when input arrives from the communications line, it may need to be absorbed rapidly to make room for more input. If the first batch of input has not been processed before the second batch arrives, data may be lost.

A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be itself interrupted as in (b) in the figure above.

As an example of this second approach, consider a system with three I/O devices: a printer, a disk, and a communications line, with increasing priorities of 2, 4, and 5, respectively. Figure below illustrates a possible sequence.

- Example timing sequence of multiple interrupts


User program
Printer interrupt service routine
Communication interrupt service routine
Disk interrupt service routine
t = 0
t = 10
t = 15
t = 25
t = 25
t = 35
t = 40

A user program begins at *t* = 0. At *t* = 10, a printer interrupt occurs; user information is placed on the system stack and execution continues at the printer **interrupt service routine (ISR)**.

While this routine is still executing, at *t* = 15, a communications interrupt occurs. Because the communications line has higher priority than the printer, the interrupt is honored. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR. While this routine is executing, a disk interrupt occurs (*t* = 20). Because this interrupt is of lower priority, it is simply held, and the communications ISR runs to completion.

When the communications ISR is complete (*t* = 25), the previous processor state is restored, which is the execution of the printer ISR. However, before even a single instruction in that routine can be executed, the processor honors the higher priority disk interrupt and control transfers to the disk ISR.

Only when that routine is complete  (*t* = 35) is the printer ISR resumed.

When that routine completes (*t* = 40), control finally returns to the user program.

**I/O Function**

Thus far, we have discussed the operation of the computer as controlled by the processor, and we have looked primarily at the interaction of processor and memory. The discussion has only alluded to the role of the I/O component.

An I/O module (e.g., a disk controller) can exchange data directly with the processor. Just as the processor can initiate a read or write with memory, designating the address of a specific location, the processor can also read data from or write data to an I/O module.

In this latter case, the processor identifies a specific device that is controlled by a particular I/O module. Thus, an instruction sequence can with I/O instructions rather than memory-referencing instructions.

In some cases, it is desirable to allow I/O exchanges to occur directly with memory. In such a case, the processor grants to an I/O module the authority to read from or write to memory, so that the I/O-memory transfer can occur without tying up the processor. During such a transfer, the I/O module issues read or write commands to memory, relieving the processor of responsibility for the exchange.

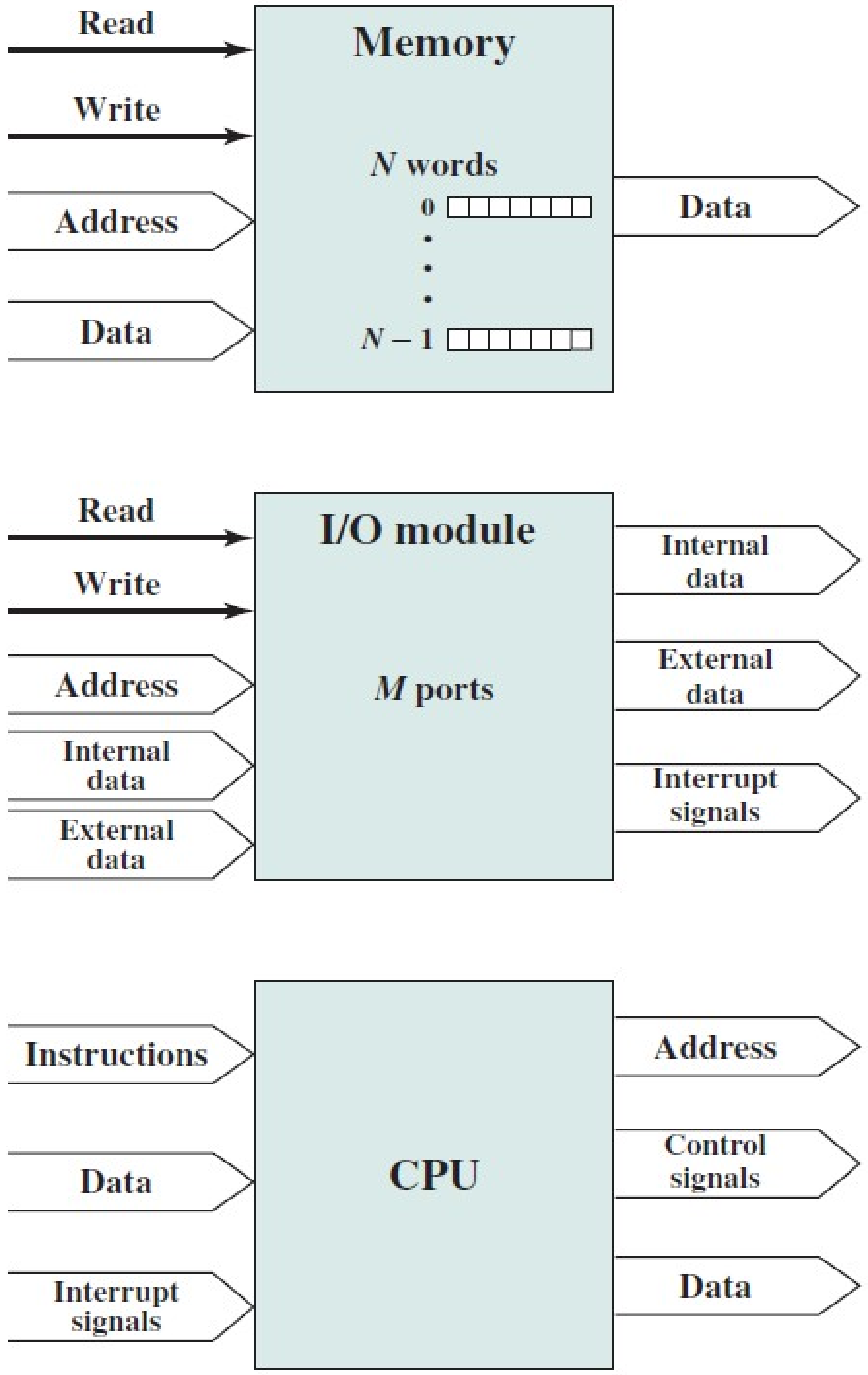
This operation is known as **direct memory access (DMA)** and is examined in later chapters.

## Interconnection Structures

- A computer consists of a set of components or modules of three basic types (processor, memory, I/O) that communicate with each other. In effect, a computer is a network of basic modules. Thus, there must be paths for connecting the modules.
- The collection of paths connecting the various modules is called the *interconnection structure*. The design of this structure will depend on the exchanges that must be made among modules.

# Computer Modules

- The wide arrows represent multiple signal lines carrying multiple bits of information in parallel.
- Each narrow arrow represents a single signal line.



**Memory:** Typically, a memory module will consist of *N* words of equal length. Each word is assigned a unique numerical address (0, 1, …, *N* - 1). A word of data can be read from or written into the memory. The nature of the operation is indicated by read and write control signals. The location for the operation is  specified by an address.

**I/O module:** From an internal (to the computer system) point of view, I/O is functionally similar to memory. There are two operations, read and write. Further, an I/O module may control more than one external device.

We can refer to each of the interfaces to an external device as a *port* and give each a unique address (e.g., 0, 1, …, *M* - 1). In addition, there are external data paths for the input and output of data with an external device. Finally, an I/O module may be able to send interrupt signals to the processor.

**Processor:** The processor reads in instructions and data, writes out data after processing, and uses control signals to control the overall operation of the system. It also receives interrupt signals.

The interconnection structure must support the following types of transfers:

- **Memory to processor:** The processor reads an instruction or a unit of data from memory.
- **Processor to memory:** The processor writes a unit of data to memory.
- **I/O to processor:** The processor reads data from an I/O device via an I/O module.
- **Processor to I/O:** The processor sends data to the I/O device.
- **I/O to or from memory:** For these two cases, an I/O module is allowed to exchange data directly with memory, without going through the processor, using direct memory access.

Over the years, a number of interconnection structures have been tried. By far the most common are

1. the bus and various multiple-bus structures, and
2. point-to-point interconnection structures with packetized data transfer.

## Bus Interconnection

A bus is a communication pathway connecting two or more devices. A key characteristic of a bus is that it is a shared transmission medium.

Multiple devices connect to the bus, and a signal transmitted by any one device is available for reception by all other devices attached to the bus.

If two devices transmit during the same time period, their signals will overlap and become garbled. Thus, only one device at a time can successfully transmit.

Typically, a bus consists of multiple communication pathways, or lines. Each line is capable of transmitting signals representing binary 1 and binary 0. Over time, a sequence of binary digits can be transmitted across a single line.

Taken together, several lines of a bus can be used to transmit binary digits simultaneously (in parallel). For example, an 8-bit unit of data can be transmitted over eight bus lines.

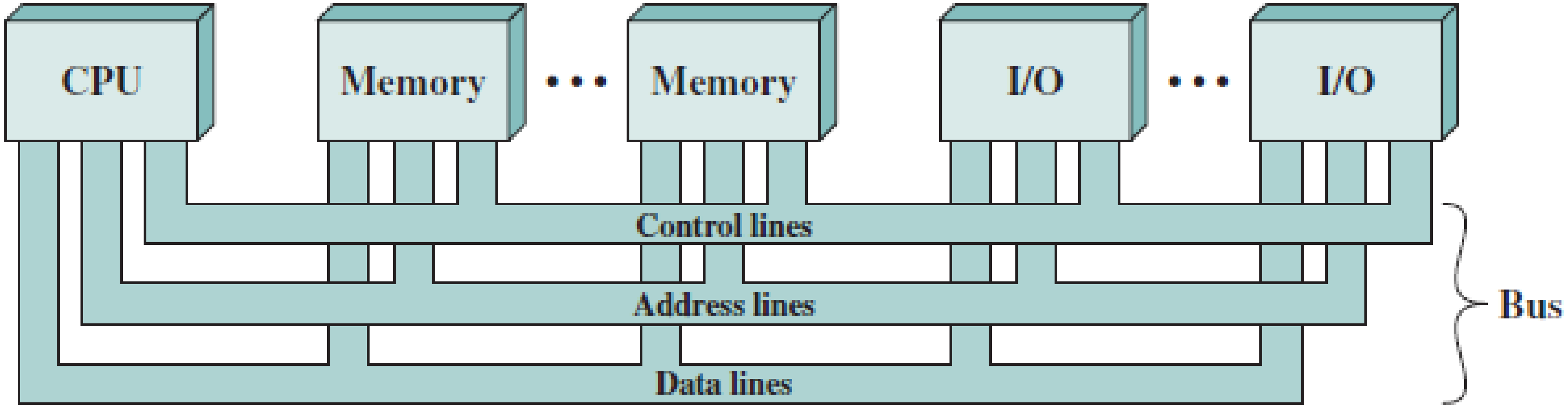
Computer systems contain a number of different buses that provide pathways between components at various levels of the computer system hierarchy. A bus that connects major computer components (processor, memory, I/O) is called a **system bus**. The most common computer interconnection structures are based on the use of one or more system buses.

## Bus Structure

A system bus consists, typically, of from about fifty to hundreds of separate lines. Each line is assigned a particular meaning or function. Although there are many different bus designs, on any bus the lines can be classified into three functional groups :

- data,
- address,
- and control lines.

In addition, there may be power distribution lines that supply power to the attached modules.


CPU
Memory
Memory
I/O
I/O
Control lines
Address lines
Data lines
Bus

The **data lines** provide a path for moving data among system modules. These lines, collectively, are called the **data bus**. The data bus may consist of 32, 64, 128, or even more separate lines.

The number of lines being referred to as the *width* of the data bus. Because each line can carry only 1 bit at a time, the number of lines determines how many bits can be transferred at a time.

The width of the data bus is a key factor in determining overall system performance.  For example, if the data bus is 32 bits wide and each instruction is 64 bits long, then the processor must access the memory module twice during each instruction cycle.

The **address lines** are used to designate the source or destination of the data on the data bus. For example, if the processor wishes to read a word (8, 16, or 32 bits) of data from memory, it puts the address of the desired word on the address lines. Clearly, the width of the **address bus** determines the maximum possible memory capacity of the system. Furthermore, the address lines are generally also used to address I/O ports.

Typically, the higher-order bits are used to select a particular module on the bus, and the lower-order bits select a memory location or I/O port within the module.

For example, on an 8-bit address bus, address 01111111 and below might reference locations in a memory module (module 0) with 128 words of memory, and address 10000000 and above refer to devices attached to an I/O module (module 1).

The **control lines** are used to control the access to and the use of the data and address lines. Because the data and address lines are shared by all components, there must be a means of controlling their use. Control signals transmit both command and timing information among system modules. Timing signals indicate the validity of data and address information. Command signals specify operations to be performed.

Typical control lines include:

- **Memory write:** causes data on the bus to be written into the addressed location
- **Memory read:** causes data from the addressed location to be placed on the bus
- **I/O write:** causes data on the bus to be output to the addressed I/O port
- **I/O read:** causes data from the addressed I/O port to be placed on the bus
- **Transfer ACK:** indicates that data have been accepted from or placed on the bus
- **Bus request:** indicates that a module needs to gain control of the bus
- **Bus grant:** indicates that a requesting module has been granted control of the bus
- **Interrupt request:** indicates that an interrupt is pending
- **Interrupt ACK:** acknowledges that the pending interrupt has been recognized
- **Clock:** is used to synchronize operations
- **Reset:** initializes all modules.

The operation of the bus is as follows. If one module wishes to send data to another, it must do two things:

1. obtain the use of the bus, and
2. transfer data via the bus.

If one module wishes to request data from another module, it must

3. obtain the use of the bus, and
4. transfer a request to the other module over the appropriate control and address lines. It must then wait for that second module to send the data.

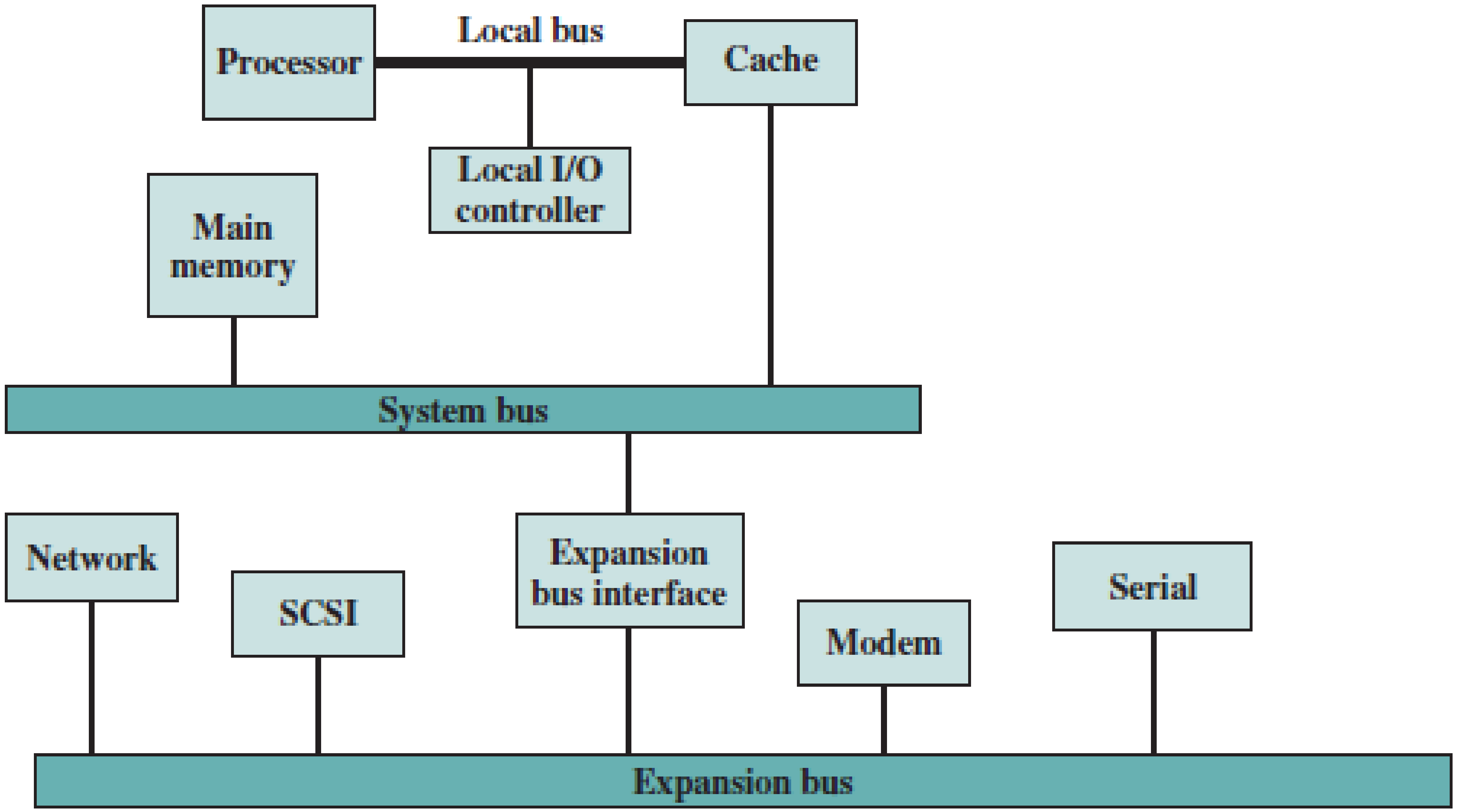
**Multiple-Bus Hierarchies**

If a great number of devices are connected to the bus, performance will suffer. There are two main causes:

1. In general, the more devices attached to the bus, the greater the bus length and hence the greater the propagation delay. This delay determines the time it takes for devices to coordinate the use of the bus. When control of the bus passes from one device to another frequently, these propagation delays can noticeably affect performance.
2. The bus may become a bottleneck as the aggregate data transfer demand approaches the capacity of the bus. This problem can be countered to some extent by increasing the data rate that the bus can carry and by using wider buses (e.g., increasing the data bus from 32 to 64 bits). However, because the data rates generated by attached devices (e.g., graphics and video controllers, network interfaces) are growing rapidly, this is a race that a single bus is ultimately destined to lose.

Accordingly, most bus-based computer systems use multiple buses, generally  laid out in a hierarchy. A typical traditional structure is shown below.

There is a local bus that connects the processor to a cache memory and that may support one or more local devices. The cache memory controller connects the cache not only to this local bus, but to a system bus to which are attached all of the main memory modules. In contemporary systems, the cache is in the same chip as the processor, and so an external bus or other interconnect scheme is not needed, although there may also be an external cache.



(a) Traditional bus architecture

The use of a cache structure insulates the processor from a requirement to access main memory frequently.

Hence, main memory can be moved off of the local bus onto a system bus. In this way, I/O transfers to and from the main memory across the system bus do not interfere with the processor's activity.

It is possible to connect I/O controllers directly onto the system bus. A more efficient solution is to make use of one or more expansion buses for this purpose. An expansion bus interface buffers data transfers between the system bus and the I/O controllers on the expansion bus. This arrangement allows the system to support a wide variety of I/O devices and at the same time insulate memory-to-processor traffic from I/O traffic.

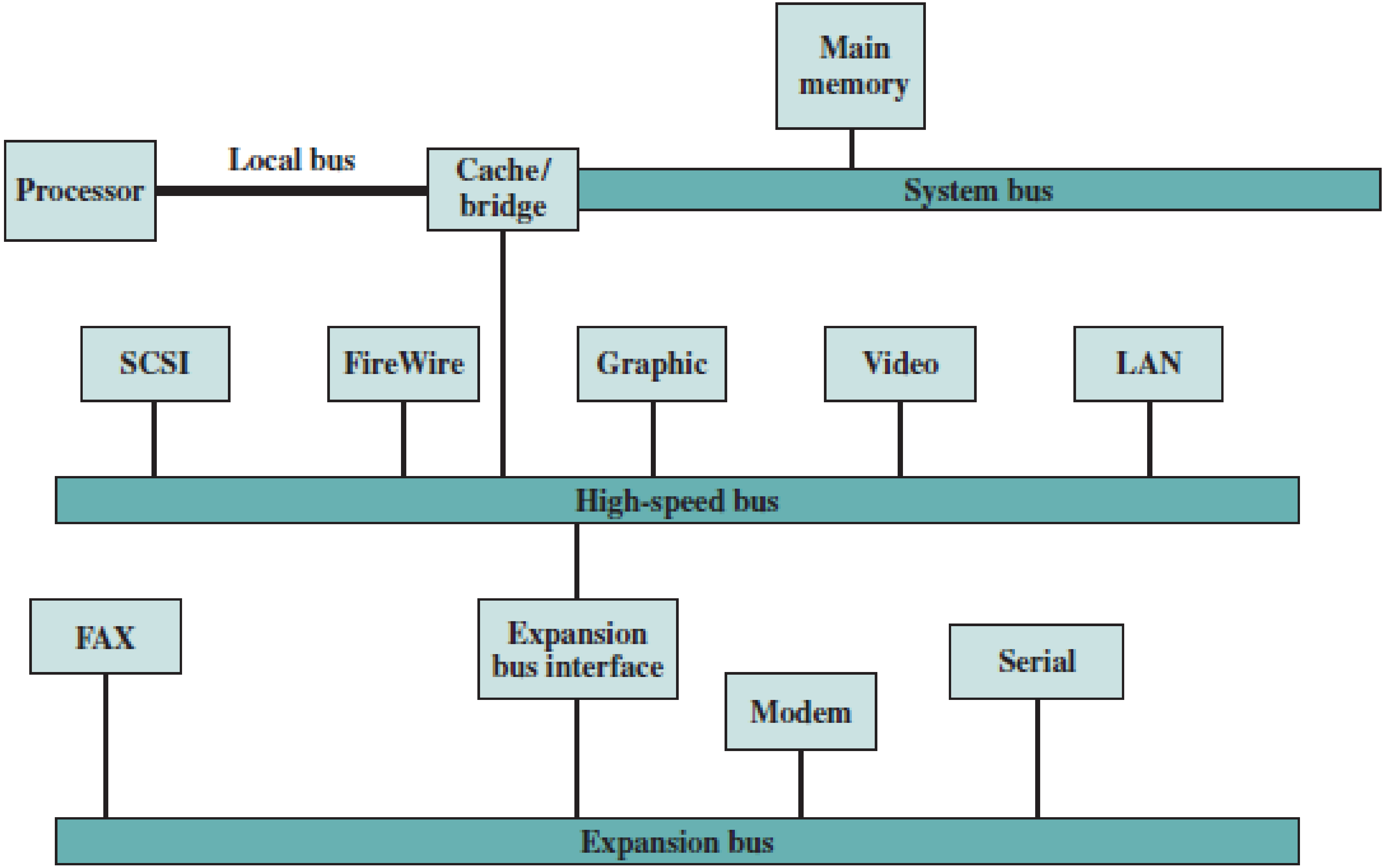
Network connections include local area networks (LANs) such as a 10-Mbps Ethernet and connections to wide area networks (WANs) such as a packet-switching network.

SCSI (small computer system interface) is itself a type of bus used to support local disk drives and other peripherals.

A serial port could be used to support a printer or scanner.

This traditional bus architecture is reasonably efficient but begins to break down as higher and higher performance is seen in the I/O devices.

In response to these growing demands, a common approach taken by industry is to build a high speed bus that is closely integrated with the rest of the system, requiring only a bridge between the processor’s bus and the high-speed bus. This arrangement is sometimes known as a mezzanine architecture.



(b) High-performance architecture

Again, there is a local bus that connects the processor to a cache controller, which is in turn connected to a system bus that supports main memory. The cache controller is integrated into a bridge, or buffering device, that connects to the high-speed bus.

This bus supports connections to high-speed LANs, such as Fast Ethernet at 100 Mbps, video and graphics workstation controllers, as well as interface controllers to local peripheral buses, including SCSI and FireWire. The latter is a high-speed bus arrangement specifically designed to support high-capacity I/O devices. Lower-speed devices are still supported off an expansion bus, with an interface buffering traffic between the expansion bus and the high-speed bus.

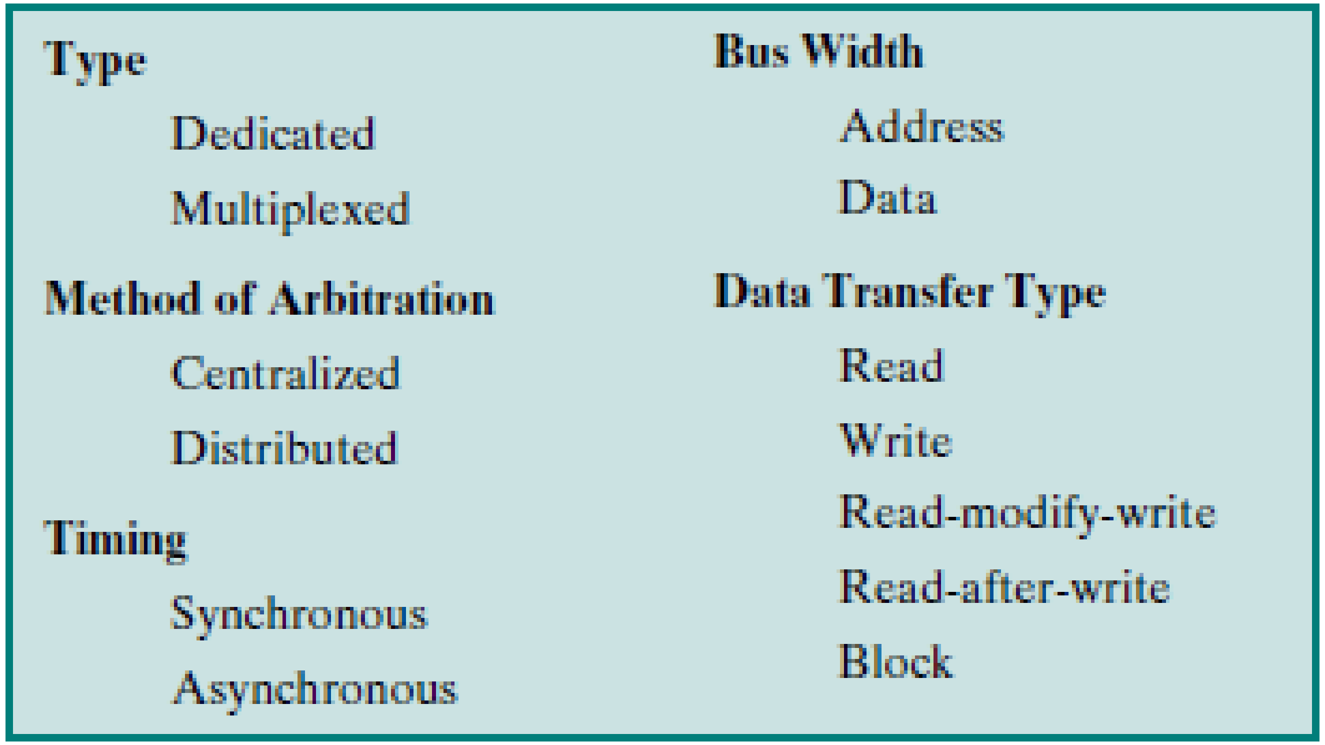
The advantage of this arrangement is that the high-speed bus brings high demand devices into closer integration with the processor and at the same time is independent of the processor.

Thus, differences in processor and high-speed bus  speeds and signal line definitions are tolerated.

Changes in processor architecture do not affect the high-speed bus, and vice versa.

## Elements of Bus Design

Although a variety of different bus implementations exist, there are a few basic parameters or design elements that serve to classify and differentiate buses.



**Type**
- Dedicated
- Multiplexed

**Method of Arbitration**
- Centralized
- Distributed

**Timing**
- Synchronous
- Asynchronous

**Bus Width**
- Address
- Data

**Data Transfer Type**
- Read
- Write
- Read-modify-write
- Read-after-write
- Block

## Bus Types

Bus lines can be separated into two generic types: dedicated and multiplexed. A dedicated bus line is permanently assigned either to one function or to a physical subset of computer components.

An example of functional dedication is the use of separate dedicated address and data lines, which is common on many buses. However, it is not essential.

For example, address and data information may be transmitted over the same set of lines using an Address Valid control line. At the beginning of a data transfer, the address is placed on the bus and the Address Valid line is activated. At this point, each module has a specified period of time to copy the address and determine if it is the addressed module.

The address is then removed from the bus, and the same bus connections are used for the subsequent read or write data transfer. This method of using the same lines for multiple purposes is known as *time multiplexing*.

The advantage of time multiplexing is the use of fewer lines, which saves space and, usually, cost. The disadvantage is that more complex circuitry is needed within each module. Also, there is a potential reduction in performance because certain events that share the same lines cannot take place in parallel.

*Physical dedication* refers to the use of multiple buses, each of which connects only a subset of modules. A typical example is the use of an I/O bus to interconnect all I/O modules; this bus is then connected to the main bus through some type of I/O adapter module. The potential advantage of physical dedication is high throughput, because there is less bus contention. A disadvantage is the increased size and cost of the system.

## Method of Arbitration

In all but the simplest systems, more than one module may need control of the bus. For example, an I/O module may need to read or write directly to memory, without sending the data to the processor. Because only one unit at a time can successfully transmit over the bus, some method of **arbitration** is needed.

The various methods can be roughly classified as being either **centralized arbitration** or **distributed arbitration**.

In a centralized scheme, a single hardware device, referred to as a *bus controller* or *arbiter*, is responsible for allocating time on the bus. The device may be a separate module or part of the processor.

In a distributed scheme, there is no central controller. Rather, each module contains access control logic and the modules act together to share the bus.

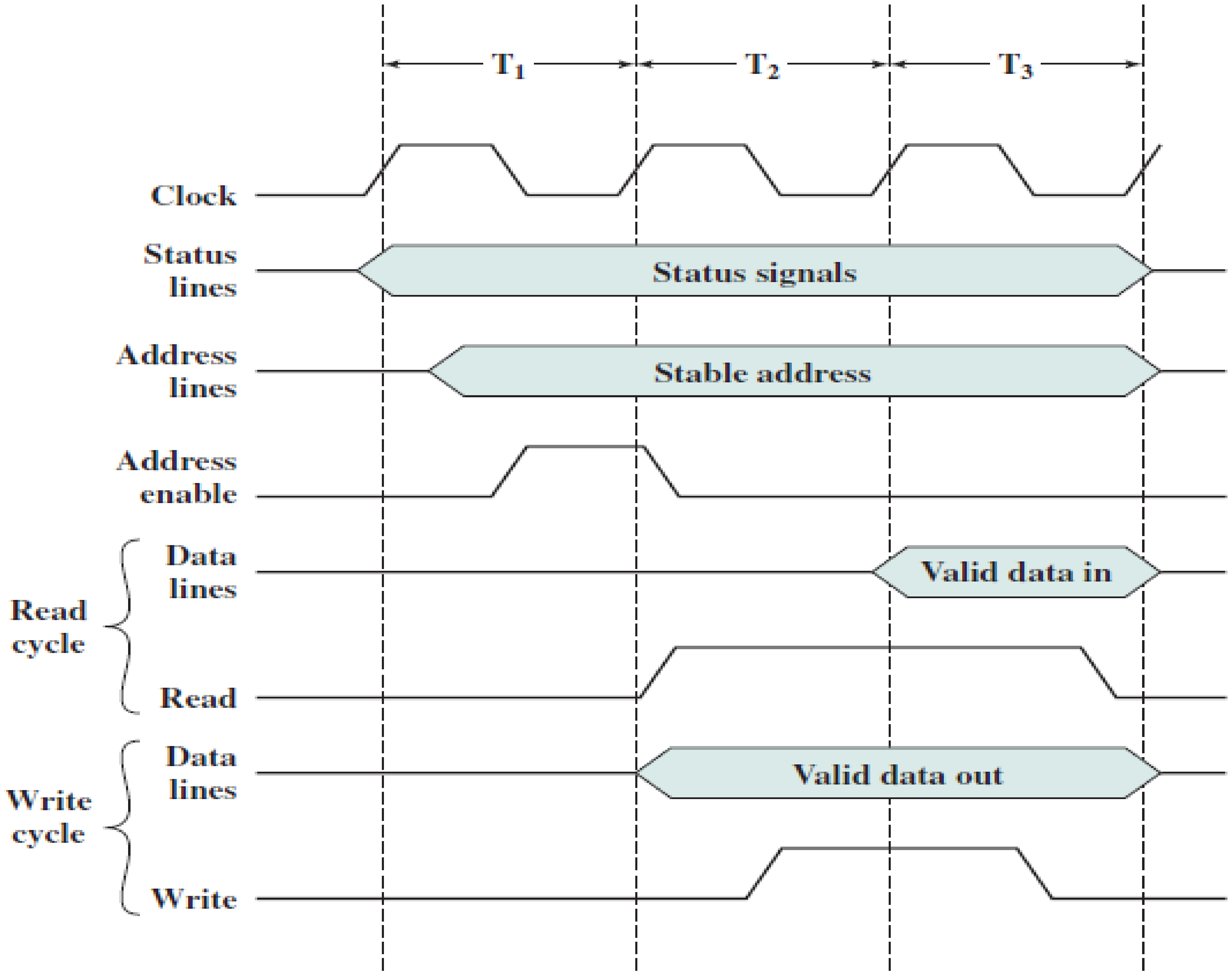
With both methods of arbitration, the purpose is to designate one device, either the processor or an I/O module, as master. The master may then initiate a data transfer (e.g., read or write) with some other device, which acts as slave for this particular exchange.

## Timing

Timing refers to the way in which events are coordinated on the bus. Buses use either synchronous timing or asynchronous timing.

With **synchronous timing**, the occurrence of events on the bus is determined by a clock. The bus includes a clock line upon which a clock transmits a regular sequence of alternating 1s and 0s of equal duration.

A single 1–0 transmission is referred to as a *clock cycle* or *bus cycle* and defines a time slot. All other devices on the bus can read the clock line, and all events start at the beginning of a clock cycle.



T1
T2
T3
Clock
Status lines
Status signals
Address lines
Stable address
Address enable
Read cycle
Data lines
Valid data in
Read
Write cycle
Data lines
Valid data out
Write

In this simple example, the processor places a memory address on the address lines during the first clock cycle and may assert various status lines. Once the address lines have stabilized, the processor issues an address enable signal.

For a read operation, the processor issues a read command at the start of the second cycle. A memory module recognizes the address and, after a delay of one cycle, places the data on the data lines. The processor reads the data from the data lines and drops the read signal.

For a write operation, the processor puts the data on the data lines at the start of the second cycle and issues a write command after the data lines have stabilized. The memory module copies the information from the data lines during the third clock cycle.

With **asynchronous timing**, the occurrence of one event on a bus follows and depends on the occurrence of a previous event. In the simple read example of figure  below, the processor places address and status signals on the bus.

After pausing for these signals to stabilize, it issues a read command, indicating the presence of valid address and control signals. The appropriate memory decodes the address and responds by placing the data on the data line.

Once the data lines have stabilized, the memory module asserts the acknowledged line to signal the processor that the data are available. Once the master has read the data from the data lines, it deasserts the read signal. This causes the memory module to drop the data and acknowledge lines. Finally, once the acknowledge line is dropped, the master removes the address information.

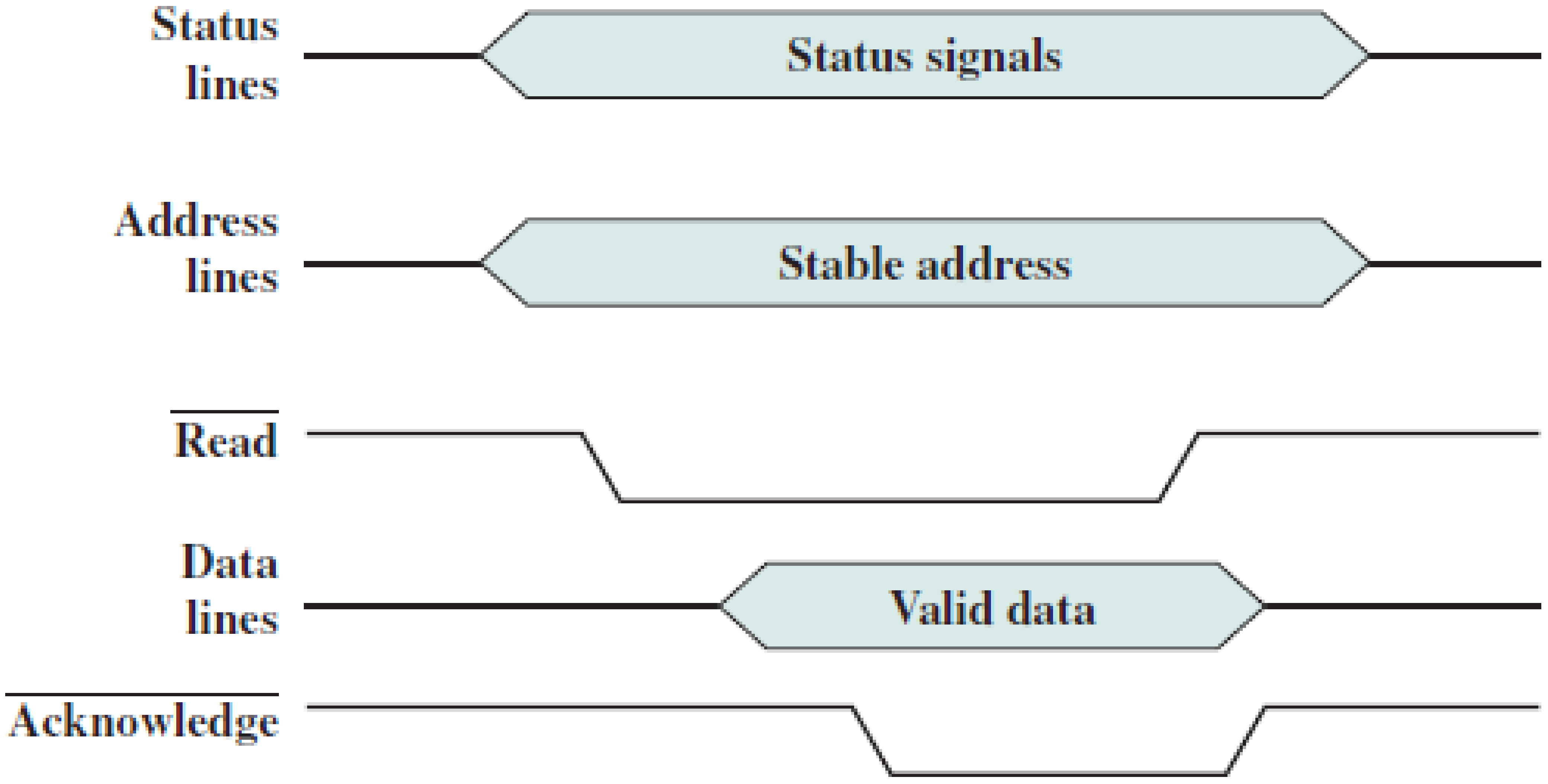
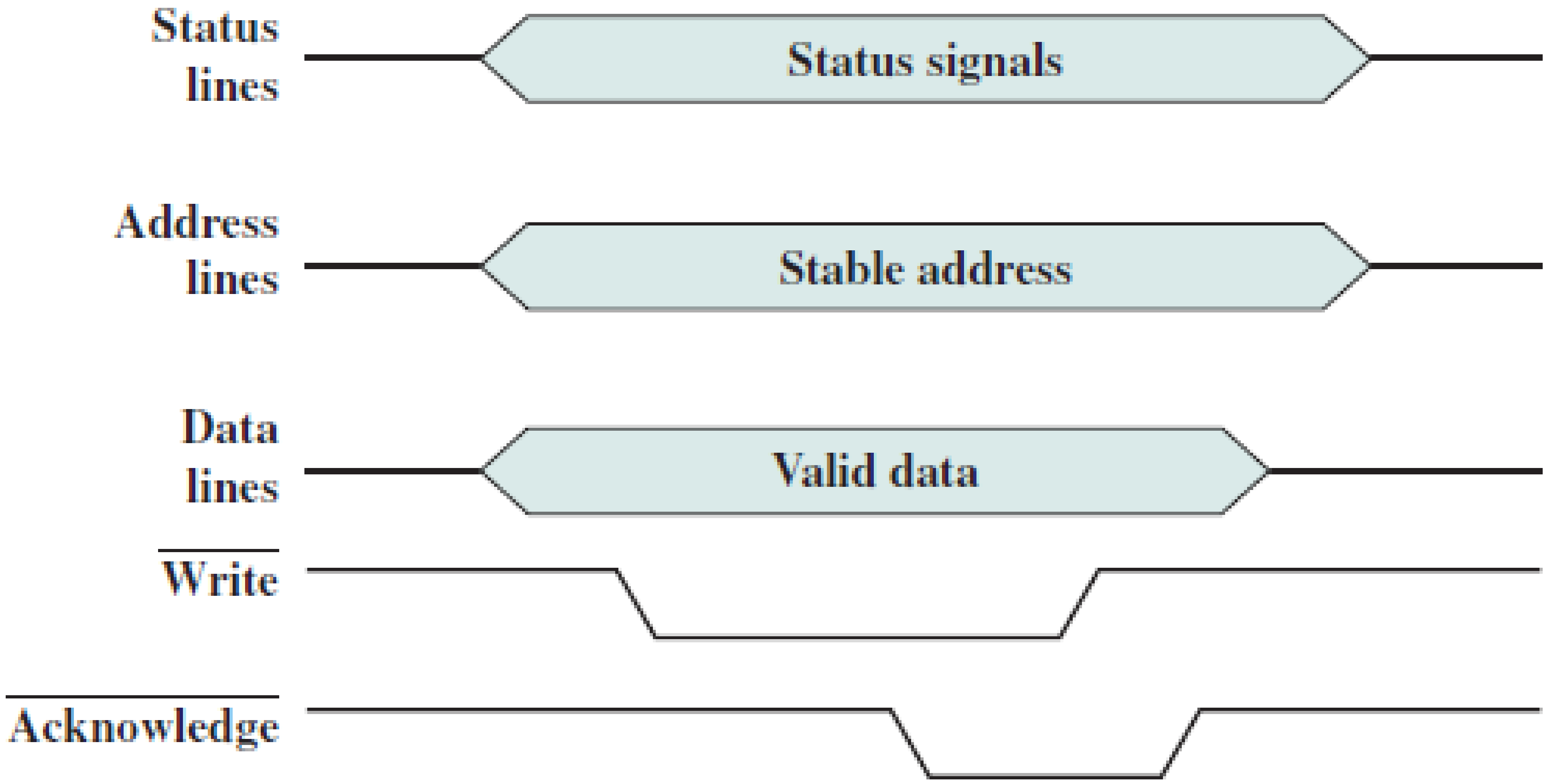

Status lines
Status signals
Address lines
Stable address
$\overline{\text{Read}}$
Data lines
Valid data
$\overline{\text{Acknowledge}}$

Figure below shows a simple asynchronous write operation. In this case, the master places the data on the data line at the same time that it puts signals on the status and address lines.

The memory module responds to the write command by copying the data from the data lines and then asserting the acknowledge line.

The master then drops the write signal and the memory module drops the acknowledge signal.


Status lines
Status signals
Address lines
Stable address
Data lines
Valid data
$\overline{\text{Write}}$
$\overline{\text{Acknowledge}}$

Synchronous timing is simpler to implement and test. However, it is less flexible than asynchronous timing. Because all devices on a synchronous bus are tied to a fixed clock rate, the system cannot take advantage of advances in device performance.

With asynchronous timing, a mixture of slow and fast devices, using older and newer technology, can share a bus.

# Point-to-Point Interconnect

The shared bus architecture was the standard approach to interconnection between the processor and other components (memory, I/O, and so on) for decades. But contemporary systems increasingly rely on point-to-point interconnection rather than shared buses.

The principal reason driving the change from bus to point-to-point interconnect was the electrical constraints encountered with increasing the frequency of wide synchronous buses. At higher and higher data rates, it becomes increasingly difficult to perform the synchronization and arbitration functions in a timely fashion.

Further, with the advent of multicore chips, with multiple processors and significant memory on a single chip, it was found that the use of a conventional shared bus on the same chip magnified the difficulties of increasing bus data rate and reducing bus latency to keep up with the processors. Compared to the shared bus, the point-to-point interconnect has lower latency, higher data rate, and better scalability.

Here, we look at an important and representative example of the point-to-point interconnect approach: Intel’s **QuickPath Interconnect (QPI)**, which was introduced in 2008.

The following are significant characteristics of QPI and other point-to-point interconnect schemes:

- **Multiple direct connections:** Multiple components within the system enjoy direct pairwise connections to other components. This eliminates the need for arbitration found in shared transmission systems.
- **Layered protocol architecture:** As found in network environments, such as TCP/IP-based data networks, these processor-level interconnects use a layered protocol architecture, rather than the simple use of control signals found in shared bus arrangements.
- **Packetized data transfer:** Data are not sent as a raw bit stream. Rather, data are sent as a sequence of packets, each of which includes control headers and error control codes.