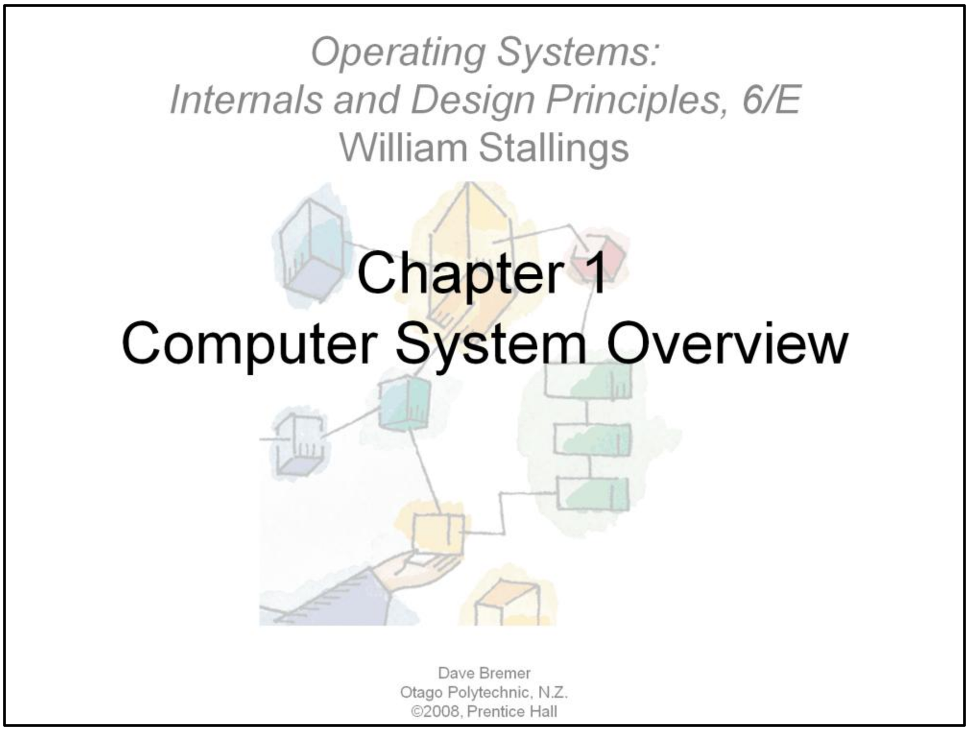
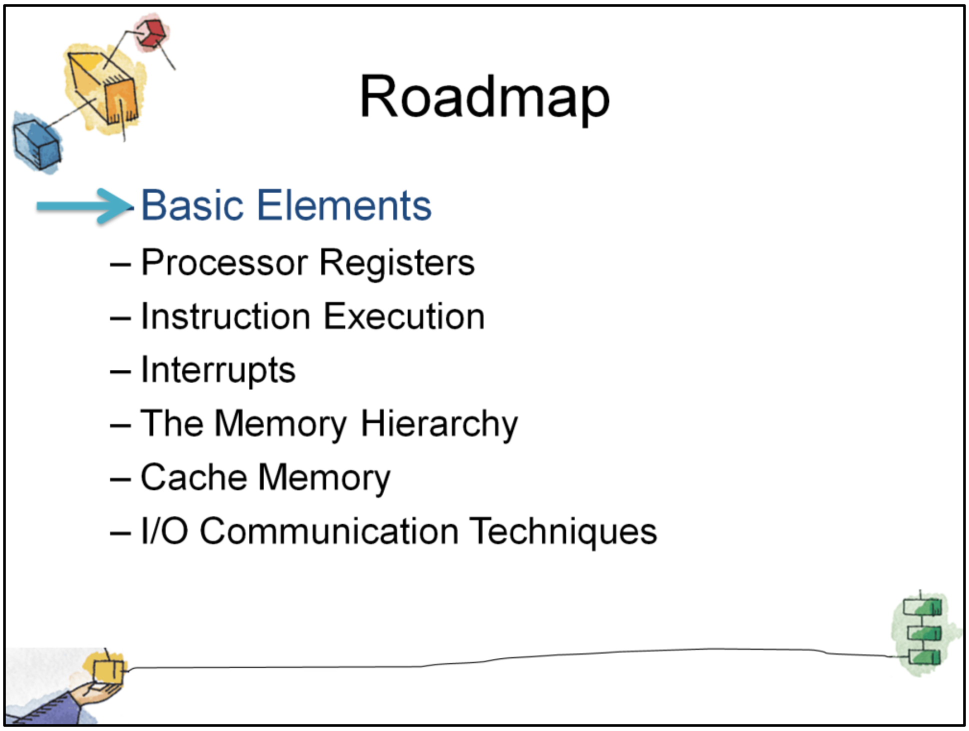


Operating Systems:
Internals and Design Principles, 6/E
William Stallings

Chapter 1
Computer System Overview

Dave Bremer
Otago Polytechnic, N.Z.
©2008, Prentice Hall

These slides are intended to help a teacher develop a presentation. This PowerPoint covers the entire chapter and includes too many slides for a single delivery. Professors are encouraged to adapt this presentation in ways which are best suited for their students and environment.



Roadmap

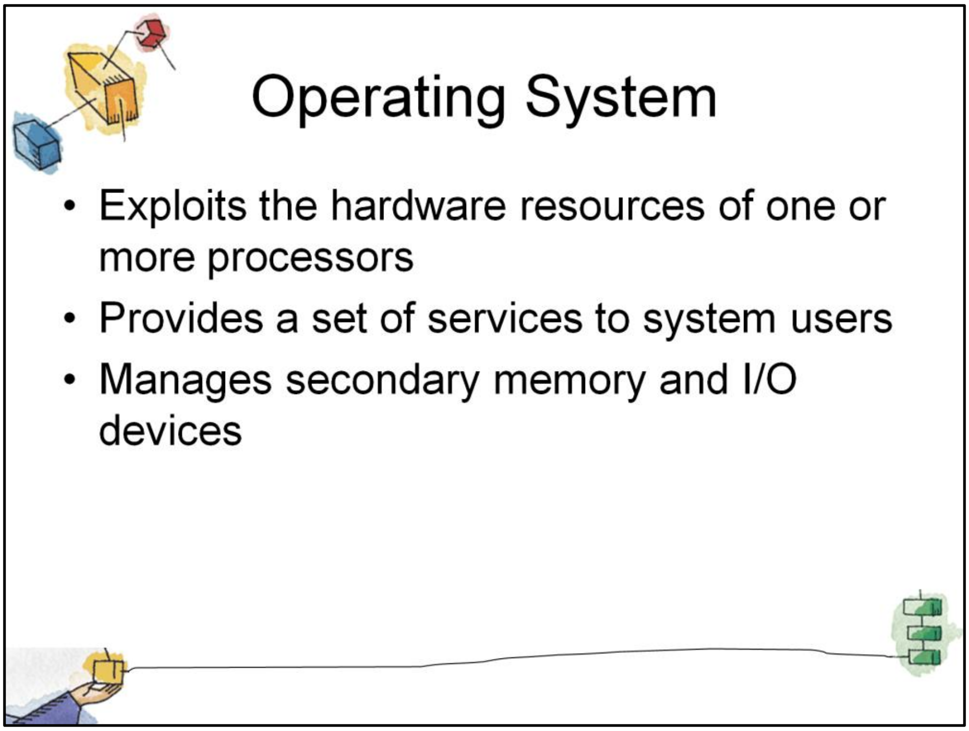
- Basic Elements
  - – Processor Registers
  - – Instruction Execution
  - – Interrupts
  - – The Memory Hierarchy
  - – Cache Memory
  - – I/O Communication Techniques

An operating system mediates among application programs, utilities, and users, on the one hand, and the computer system hardware on the other.

To appreciate the functionality of the operating system and the design issues involved, one must have some appreciation for computer organization and architecture.

This chapter provides a brief survey of the basic elements of a computer system including

- the processor,
- memory, and
- Input/Output (I/O)



Operating System

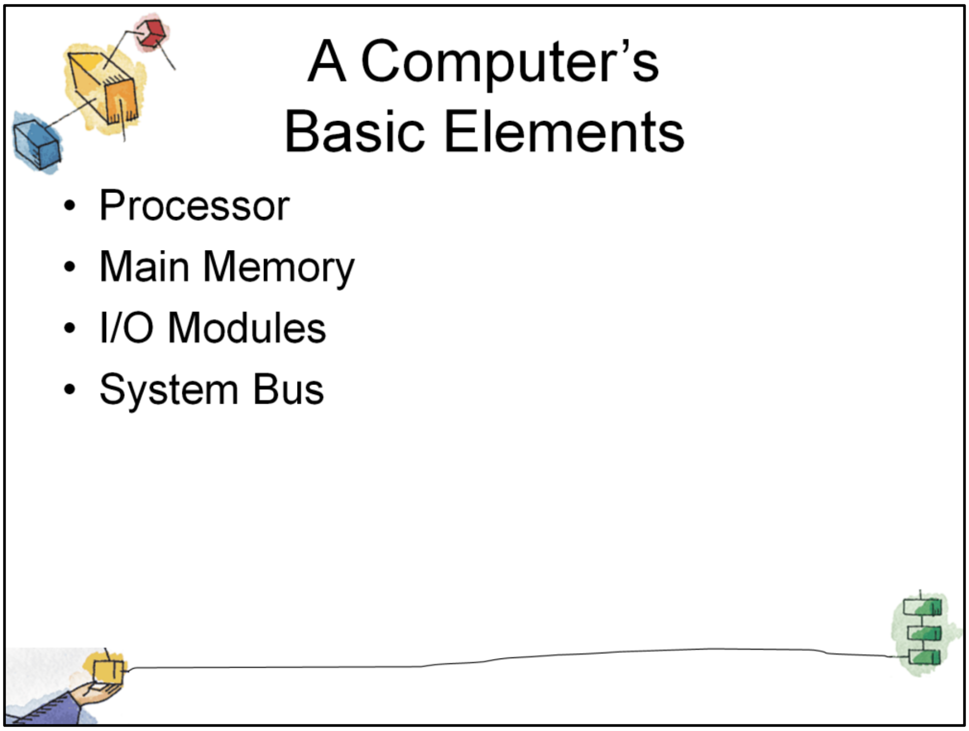
- Exploits the hardware resources of one or more processors
- Provides a set of services to system users
- Manages secondary memory and I/O devices

An operating system (OS) exploits the hardware resources of one or more processors to provide a set of services to system users.

The OS also manages secondary memory and I/O (input/output) devices on behalf of its users.

Accordingly, it is important to have some understanding of the underlying computer system hardware before we begin our examination of operating systems.

Many areas introduced in this chapter are covered in more depth later.

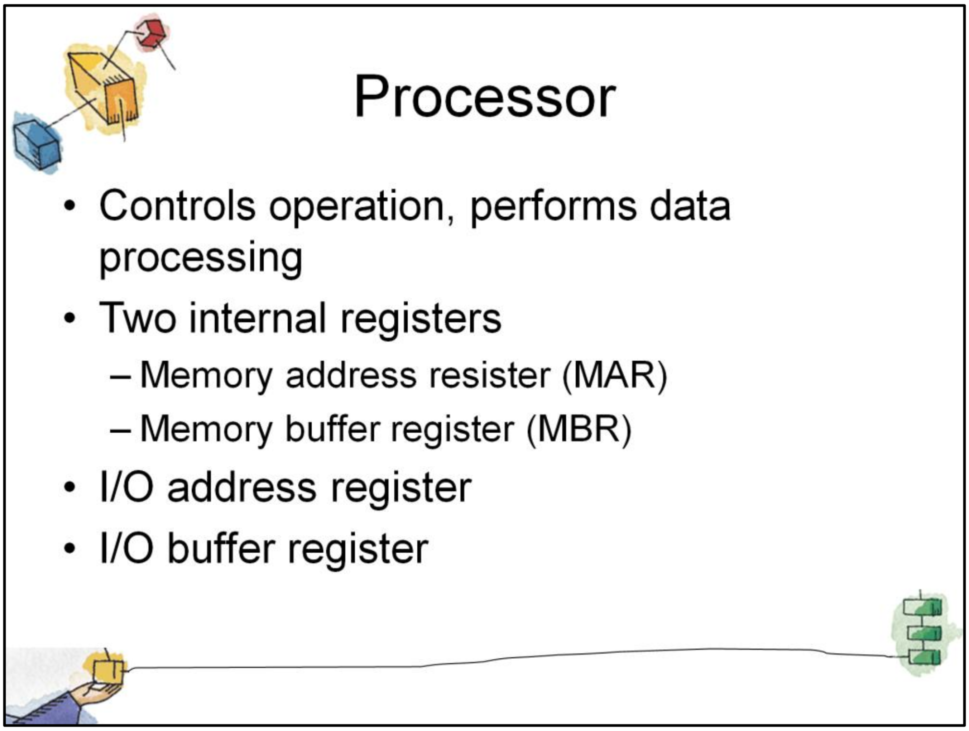


A Computer's Basic Elements

- Processor
- Main Memory
- I/O Modules
- System Bus

At a top level, a computer consists of processor, memory, and I/O components, with one or more modules of each type.

These components are interconnected in some fashion to achieve the main function of the computer, which is to execute programs.



Processor

- Controls operation, performs data processing
- Two internal registers
  - – Memory address resister (MAR)
  - – Memory buffer register (MBR)
- I/O address register
- I/O buffer register

Controls the operation of the computer and performs its data processing functions.

When there is only one processor, it is often referred to as the central processing unit (CPU).

One of the processor’s functions is to exchange data with memory.

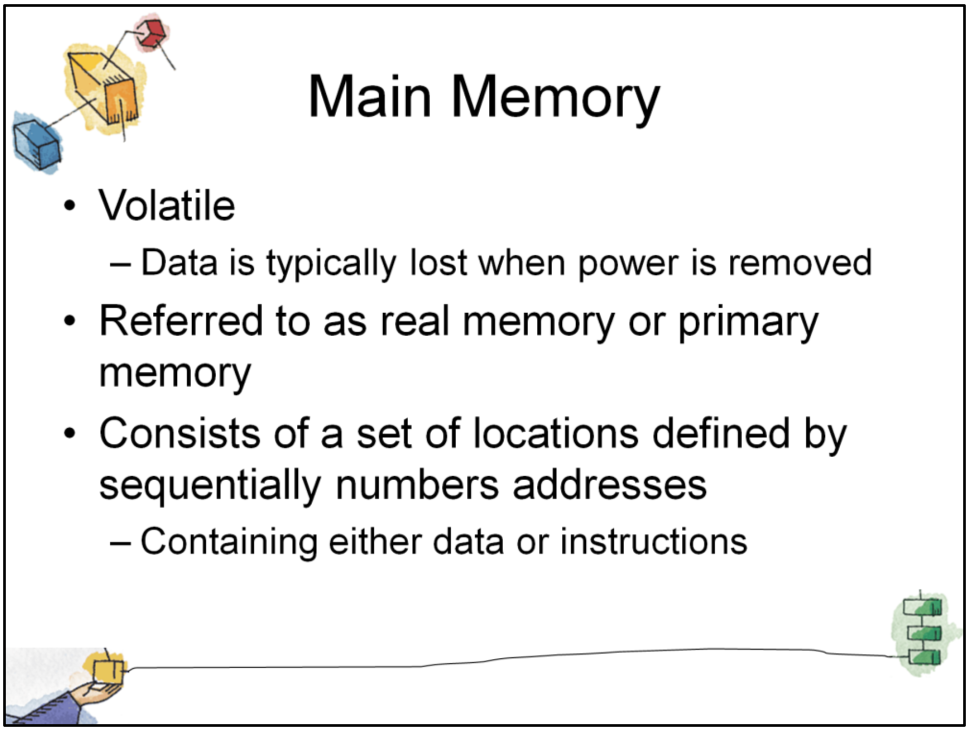
For this purpose, it typically makes use of two internal (to the processor) registers:

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 which receives the data read from memory.



Main Memory

- Volatile
  - Data is typically lost when power is removed
- Referred to as real memory or primary memory
- Consists of a set of locations defined by sequentially numbers addresses
  - Containing either data or instructions

Stores data and programs.

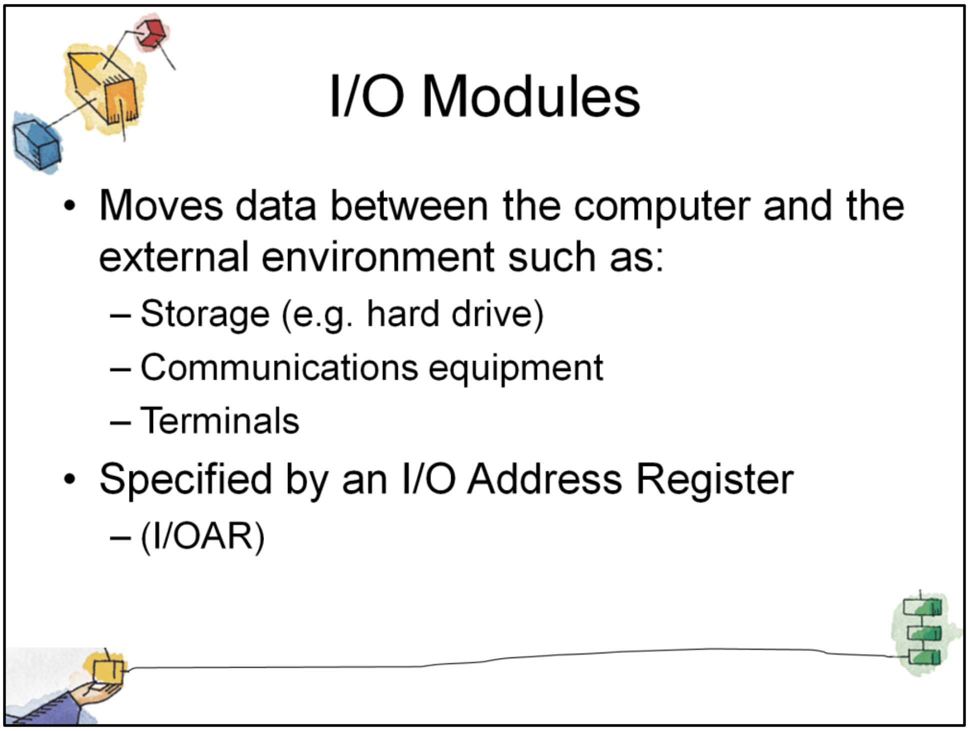
- typically volatile;
- ie, when the computer is shut down, the contents of the memory are lost.

In contrast, the contents of disk memory are retained even when the computer system is shut down.

- Main memory is also referred to as real memory or primary memory.

A memory module consists of a set of locations, defined by sequentially numbered addresses.

- Each location contains a bit pattern that can be interpreted as either an instruction or data.
-



I/O Modules

- Moves data between the computer and the external environment such as:
  - Storage (e.g. hard drive)
  - Communications equipment
  - Terminals
- Specified by an I/O Address Register
  - (I/OAR)

Move data between the computer and its external environment. The external environment consists of a variety of devices,

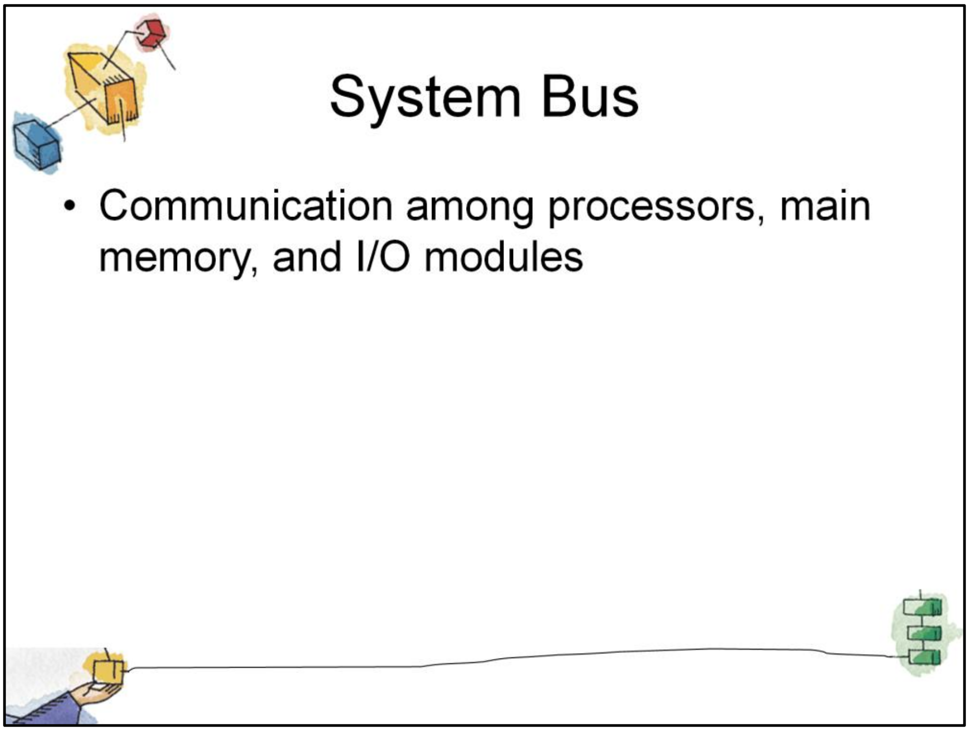
- including secondary memory devices (e. g., disks), communications equipment, and terminals.

An I/O module transfers data from external devices to processor and memory, and vice versa.

- It contains internal buffers for temporarily holding data until they can be sent on.

Similarly, an I/O address register (I/OAR) specifies a particular I/O device.

- An I/O buffer register (I/OBR) is used for the exchange of data between an I/O module and the processor.

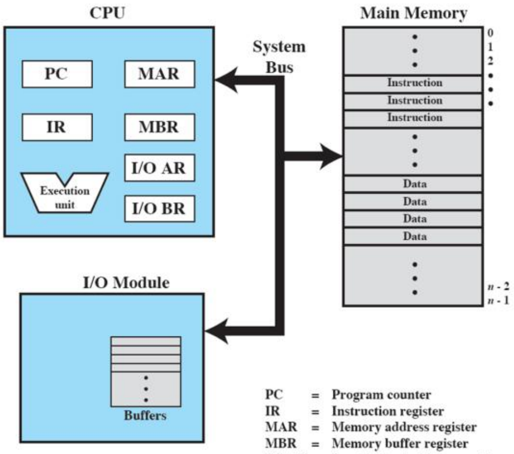


System Bus

- Communication among processors, main memory, and I/O modules

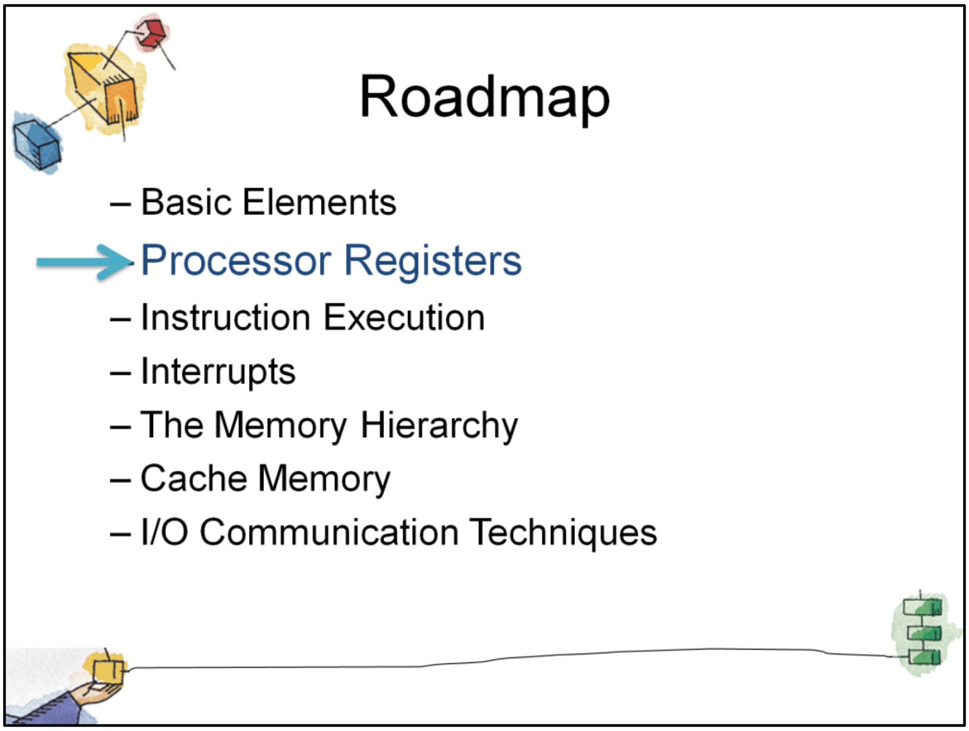
Provides for communication among processors, main memory, and I/O modules.

Top-Level View



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

Figure 1.1 Computer Components: Top-Level View



Roadmap

– Basic Elements

→ Processor Registers

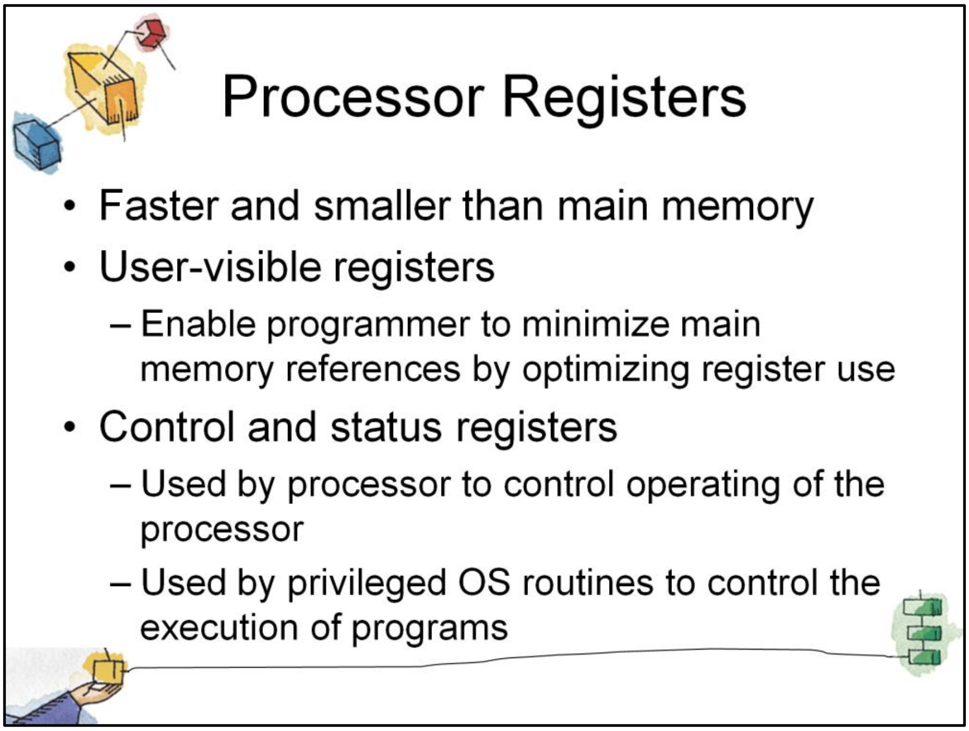
– Instruction Execution

– Interrupts

– The Memory Hierarchy

– Cache Memory

– I/O Communication Techniques



Processor Registers

- Faster and smaller than main memory
- User-visible registers
  – Enable programmer to minimize main memory references by optimizing register use
- Control and status registers
  – Used by processor to control operating of the processor
  – Used by privileged OS routines to control the execution of programs

A processor includes a set of registers that provide memory that is faster and smaller than main memory.

Processor registers serve two functions:

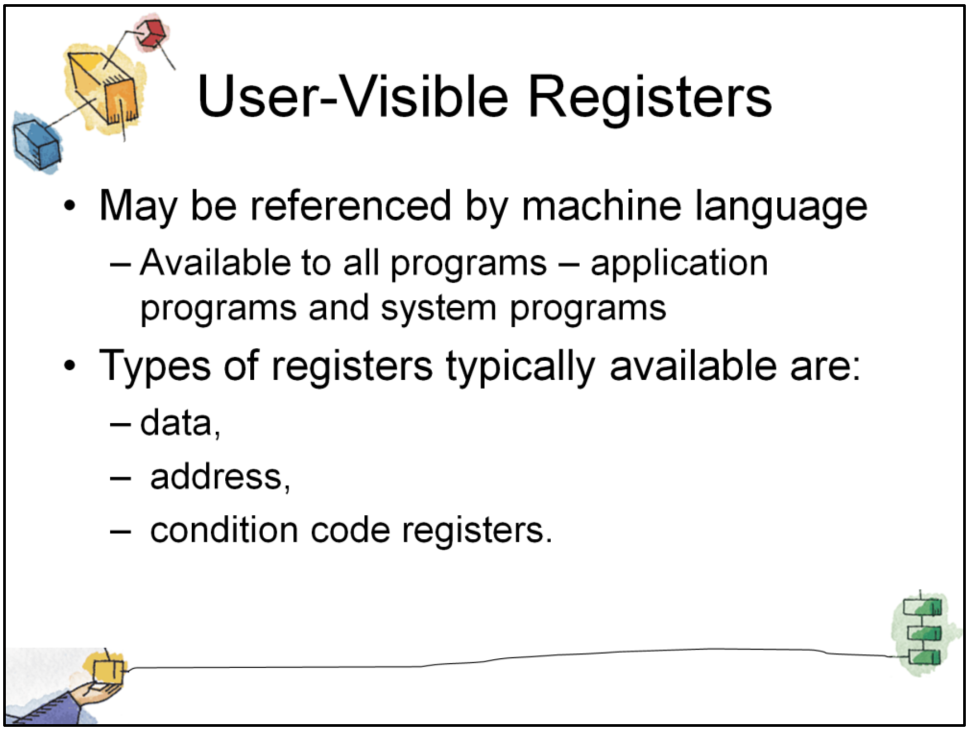
User-visible registers: Enable the machine or assembly language programmer to minimize main memory references by optimizing register use.

- For high-level languages, an optimizing compiler will attempt to make intelligent choices of which variables to assign to registers and which to main memory locations.
- Some high-level languages, such as C, allow the programmer to suggest to the compiler which variables should be held in registers.

Control and status registers: Used by the processor to control the operation of the processor and by privileged OS routines to control the execution of programs.

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

- For example, on some processors, the program counter is user visible, but on many it is not.
- For purposes of the following discussion, however, it is convenient to use these categories.



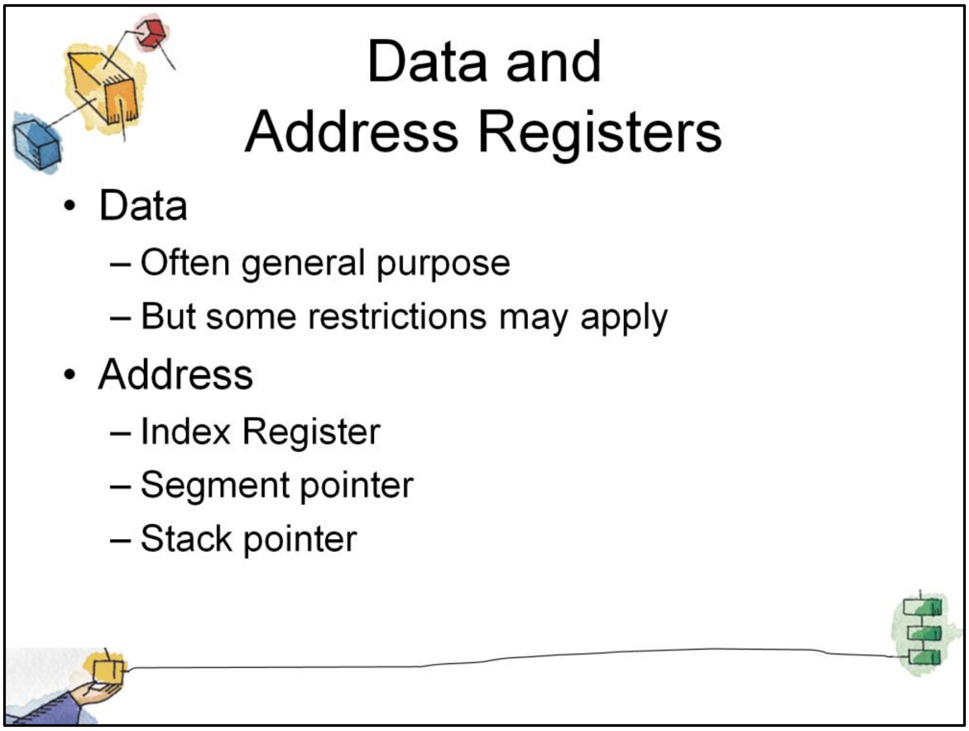
User-Visible Registers

- May be referenced by machine language
  - Available to all programs – application programs and system programs
- Types of registers typically available are:
  - data,
  - address,
  - condition code registers.

A user-visible register may be referenced by means of the machine language that the processor executes and is generally available to all programs, including application programs as well as system programs.

Types of registers that are typically available are

- data,
- address, and
- condition code registers



Data and Address Registers

- Data
  - – Often general purpose
  - – But some restrictions may apply
- Address
  - – Index Register
  - – Segment pointer
  - – Stack pointer

Data registers can be assigned to a variety of functions by the programmer.

- Usually they are general purpose in nature and can be used with any machine instruction that performs operations on data.
- Often, however, there are restrictions. E.g. there may be dedicated registers for floating-point operations and others for integer operations.

Address registers contain:

- main memory addresses of data and instructions,
- or they contain a portion of the address that is used in the calculation of the complete or effective address.

These registers may themselves be general purpose, or may be devoted to a particular way, or mode, of addressing memory.

Examples include:

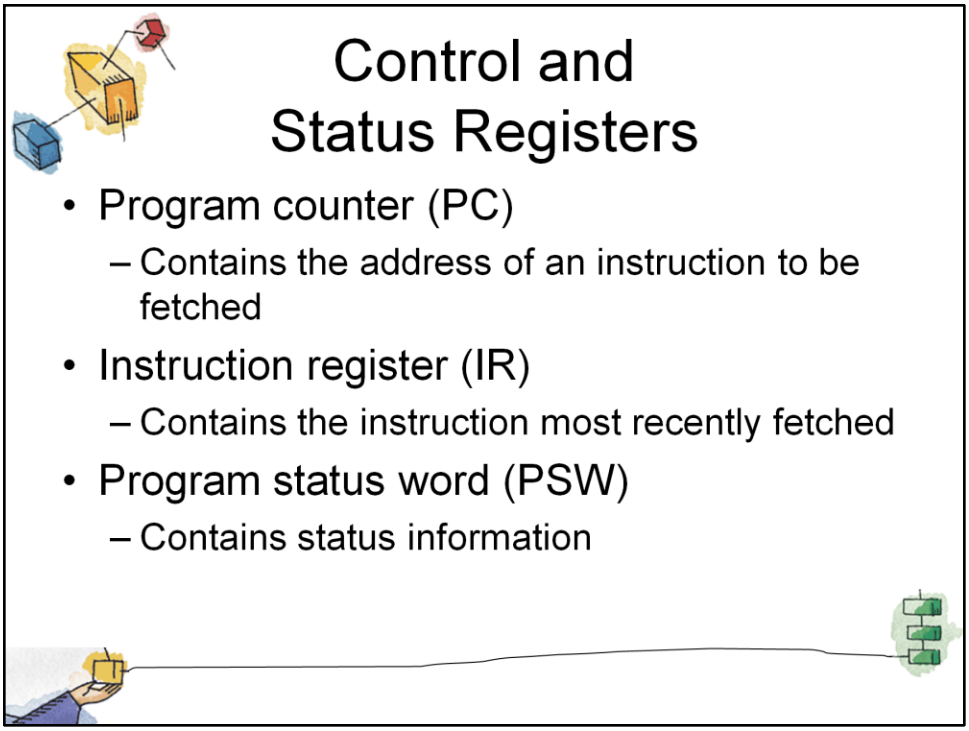
Index register: Indexed addressing is a common mode of addressing that involves adding an index to a base value to get the effective address.

Segment pointer: With segmented addressing, memory is divided into segments, which are variable-length blocks of words.

- A memory reference consists of a reference to a particular segment and an offset within the segment.
- In this mode of addressing, a register is used to hold the base address (starting location) of the segment. \
- There may be multiple registers; for example, one for the OS (i.e., when OS code is executing on the processor) and one for the currently executing application.

Stack pointer: If there is user-visible stack addressing, then there is a dedicated register that points to the top of the stack.

- This allows the use of instructions that contain no address field, such as push and pop.



Control and Status Registers

- Program counter (PC)
  - Contains the address of an instruction to be fetched
- Instruction register (IR)
  - Contains the instruction most recently fetched
- Program status word (PSW)
  - Contains status information

A variety of processor registers are employed to control the operation of the processor.

On most processors, most of these are not visible to the user.

- Some of them may be accessible by machine instructions executed in what is referred to as a control or kernel mode.

Different processors will have different register organizations and use different terminology.

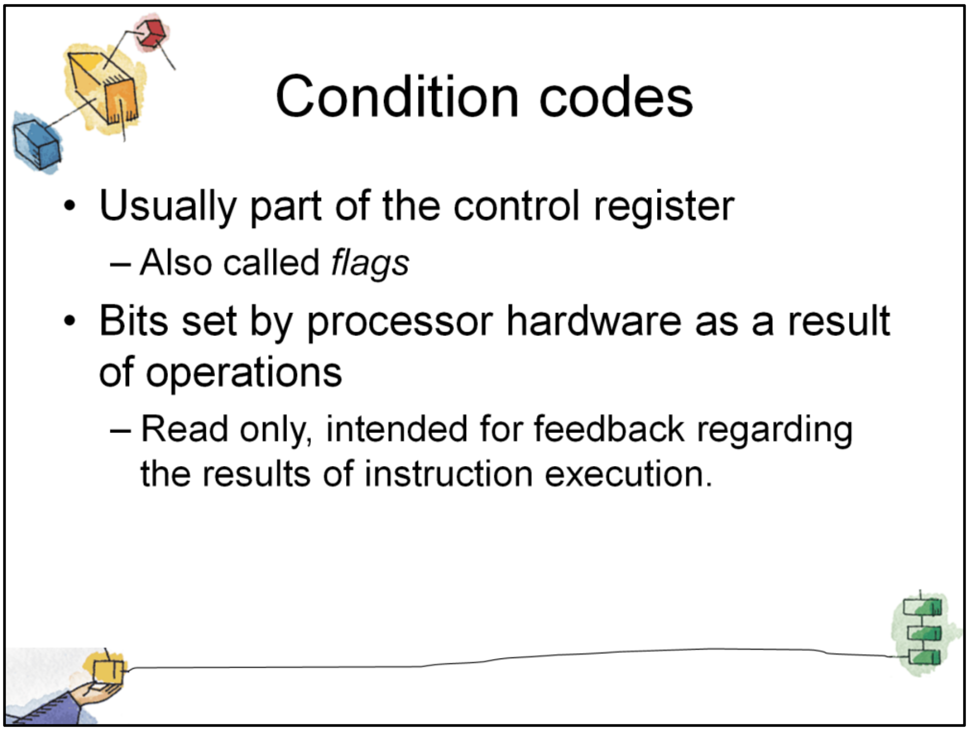
In addition to the MAR,MBR, I/OAR, and I/OBR the following are essential to instruction execution:

Program counter (PC): Contains the address of the next instruction to be fetched

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

The program status word (PSW), contains status information.

- The PSW typically contains condition codes plus other status information, such as an interrupt enable/disable bit and a kernel/user mode bit.



Condition codes

- Usually part of the control register
  - Also called *flags*
- Bits set by processor hardware as a result of operations
  - Read only, intended for feedback regarding the results of instruction execution.

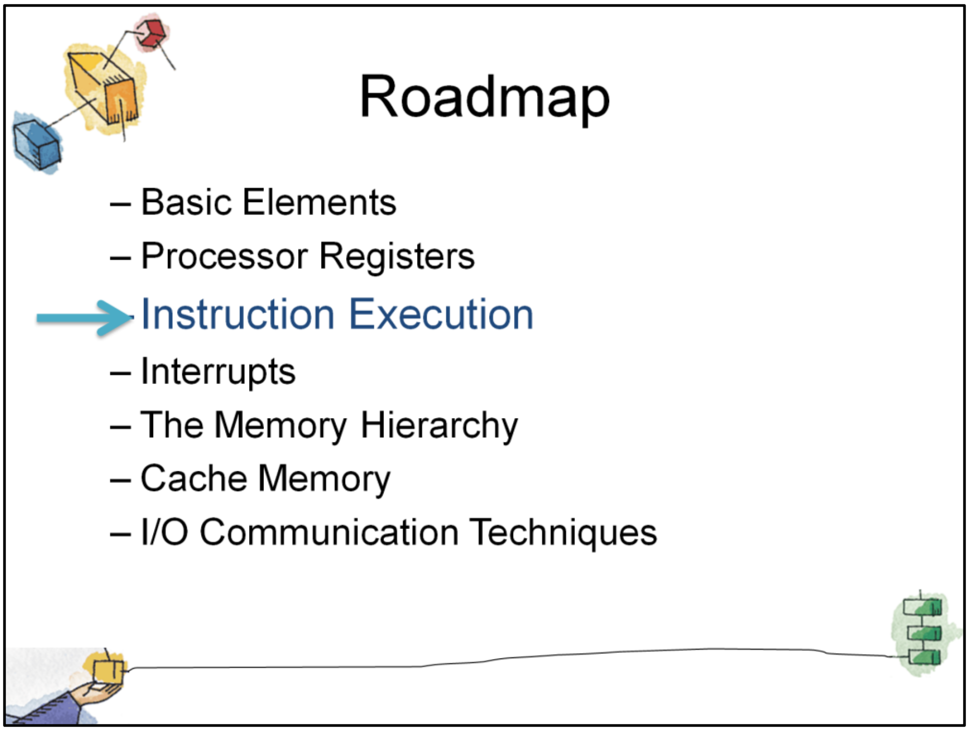
Bits typically set by the processor hardware as the result of operations.

For example, an arithmetic operation may produce a positive, negative, zero, or overflow result.

- In addition to the result itself being stored in a register or memory, a condition code is also set following the execution of the arithmetic instruction.
- The condition code may subsequently be tested as part of a conditional branch operation.
- Condition code bits are collected into one or more registers.

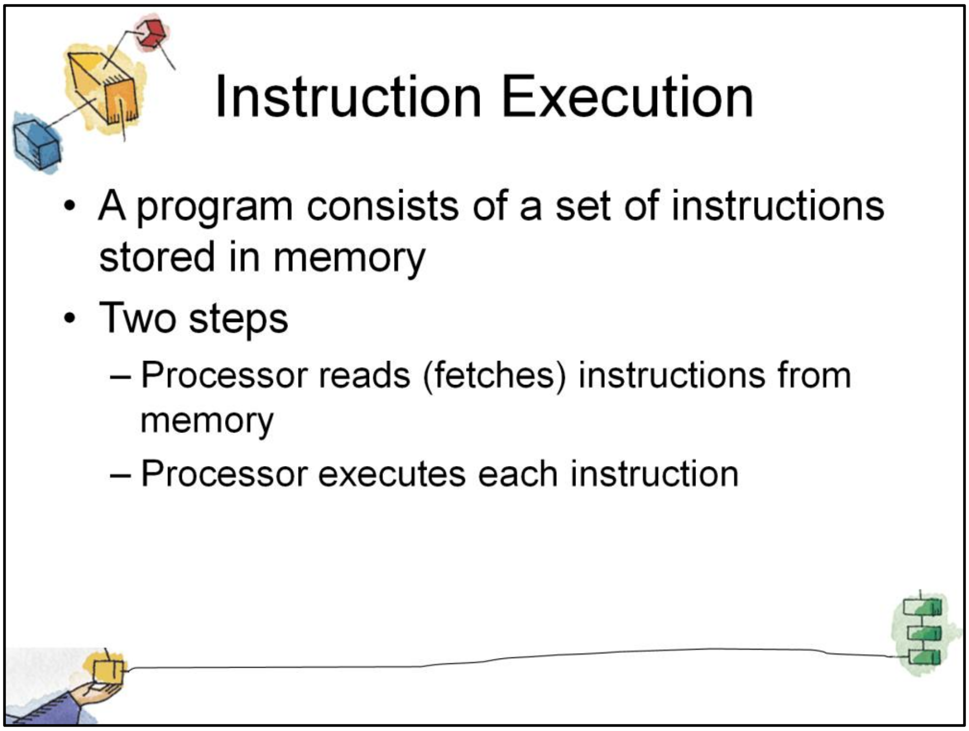
Usually, they form part of a control register.

Generally, machine instructions allow these bits to be read by implicit reference, but they cannot be altered by explicit reference because they are intended for feedback regarding the results of instruction execution.



Roadmap

- – Basic Elements
- – Processor Registers
- → Instruction Execution
- – Interrupts
- – The Memory Hierarchy
- – Cache Memory
- – I/O Communication Techniques



Instruction Execution

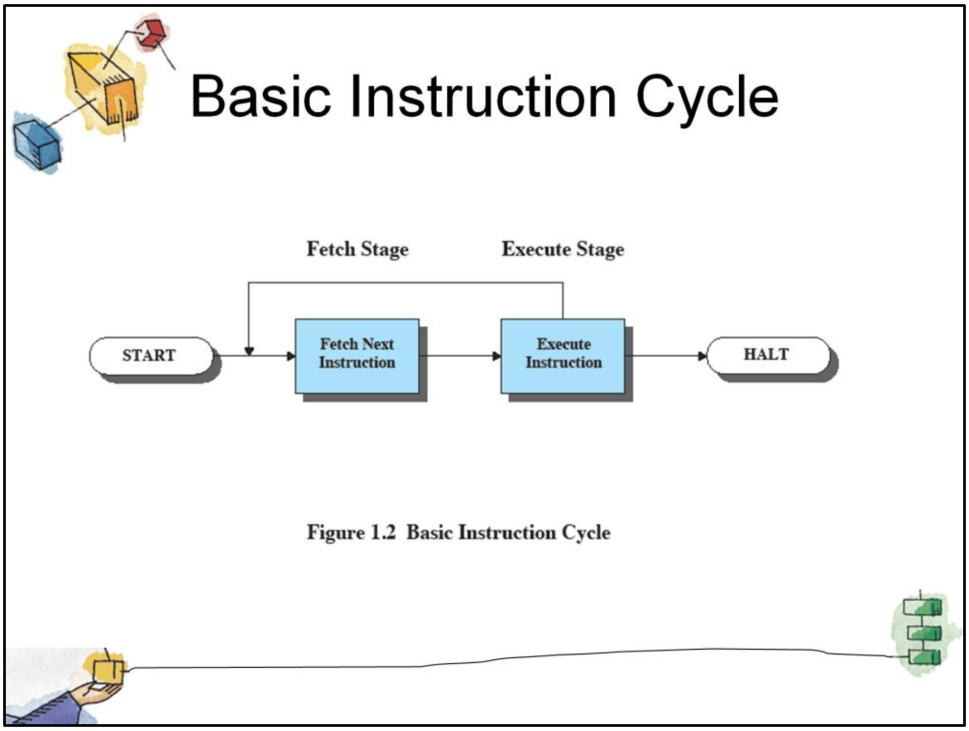
- A program consists of a set of instructions stored in memory
- Two steps
  - Processor reads (fetches) instructions from memory
  - Processor executes each instruction

A program to be executed by a processor 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.



Basic Instruction Cycle

Fetch Stage

Execute Stage

START

Fetch Next Instruction

Execute Instruction

HALT

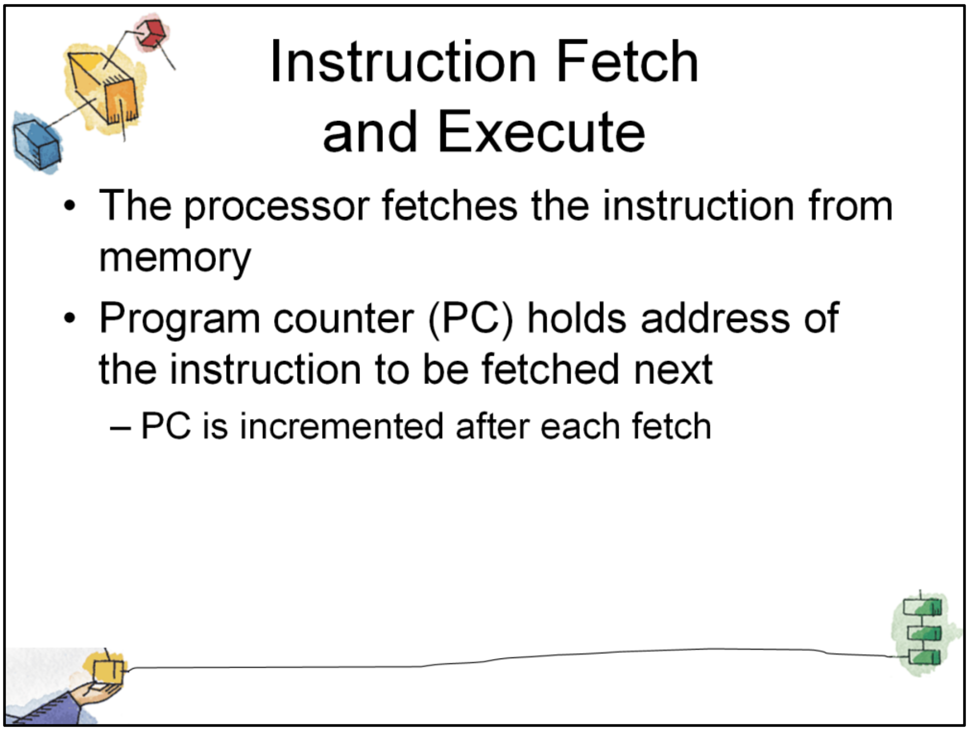
Figure 1.2 Basic Instruction Cycle

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

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.

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



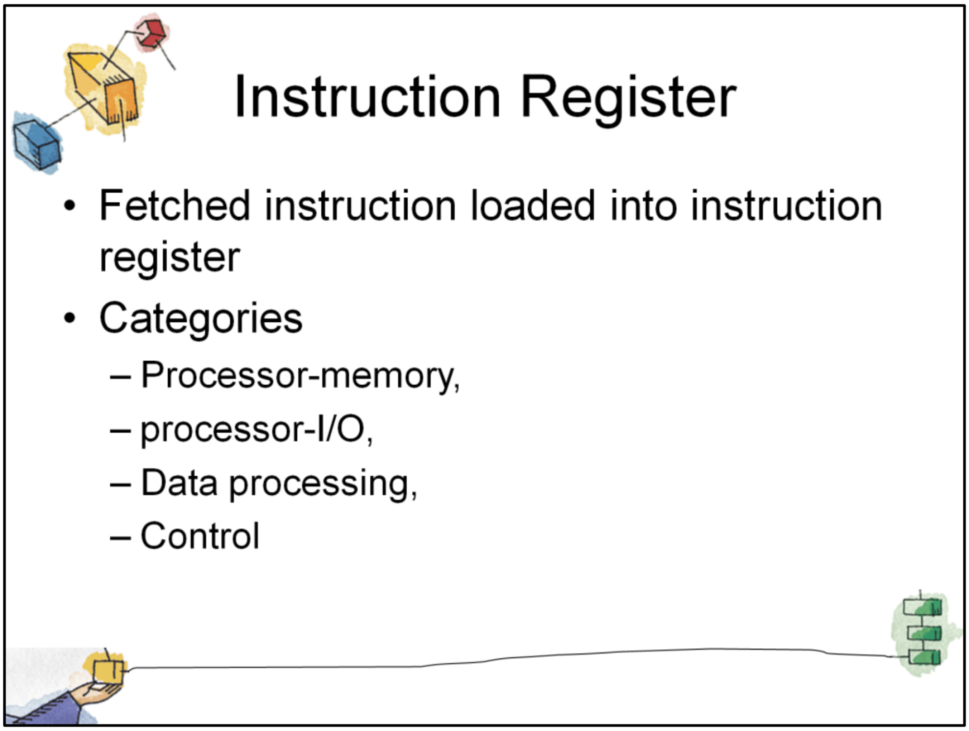
Instruction Fetch and Execute

- The processor fetches the instruction from memory
- Program counter (PC) holds address of the instruction to be fetched next
  - PC is incremented after each fetch

At the beginning of each instruction cycle, the processor fetches an instruction from memory.

Typically, the program counter (PC) holds the address of the next instruction to be fetched.

- Unless instructed 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).



Instruction Register

- Fetched instruction loaded into instruction register
- Categories
  - Processor-memory,
  - processor-I/O,
  - Data processing,
  - Control

The fetched instruction is loaded into the instruction register (IR).

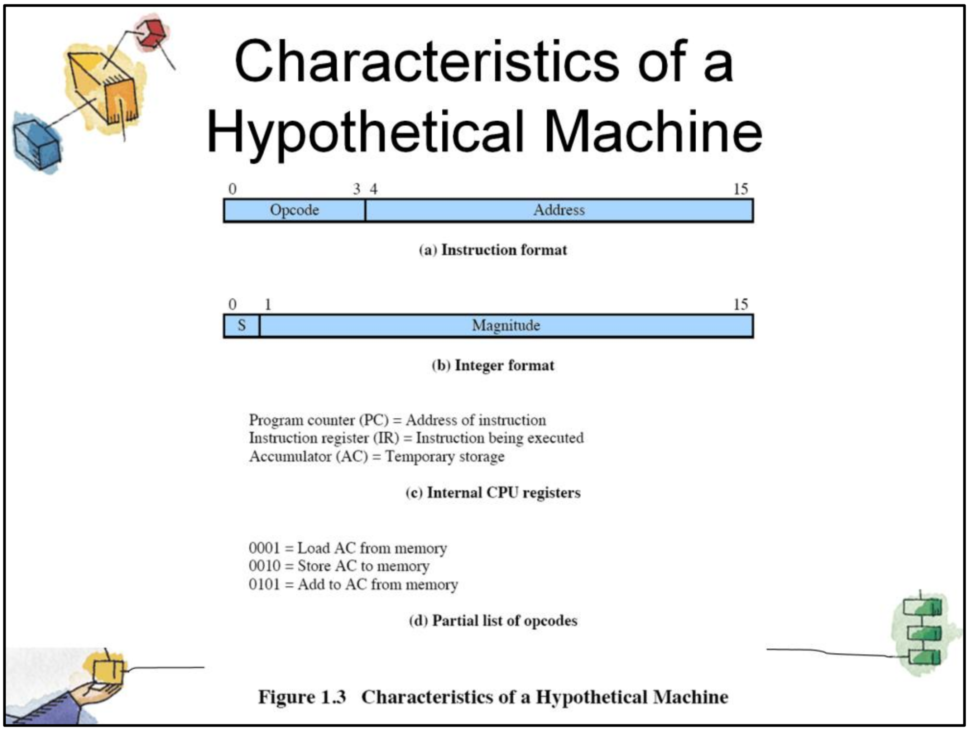
The instruction contains bits that specify the action the processor is to take.

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 opera- tion on data.

Control: An instruction may specify that the sequence of execution be altered.



Characteristics of a Hypothetical Machine

(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

Figure 1.3 Characteristics of a Hypothetical Machine

Consider a simple example using a hypothetical processor

The processor contains a single data register, called the accumulator (AC).

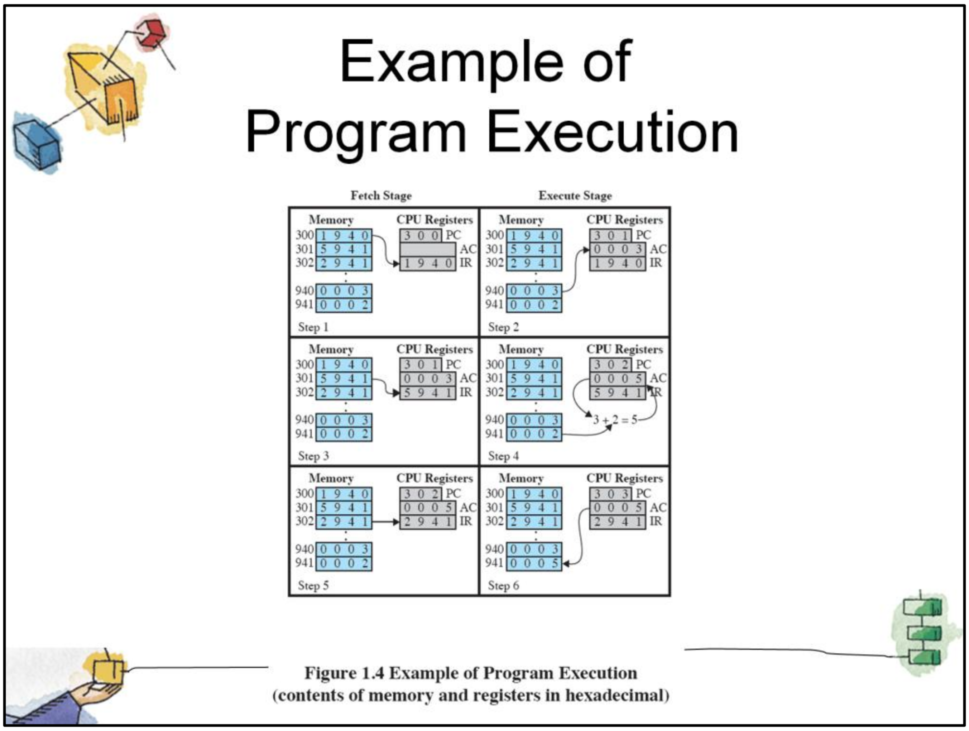
Both instructions and data are 16 bits long, and memory is organized as a sequence of 16-bit words.

- The instruction format provides 4 bits for the opcode, allowing as many as 2^4 = 16 different opcodes
- represented by a single hexadecimal digit.

The opcode defines the operation the processor is to perform.

With the remaining 12 bits of the instruction format, up to 2^{12} = 4096 (4 K) words of memory can be directly addressed.

- denoted by three hexadecimal digits.



Example of Program Execution

Figure 1.4 Example of Program Execution
(contents of memory and registers in hexadecimal)

This figure illustrates a partial program execution, showing the relevant portions of memory and processor registers.

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.

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

1. The PC contains 300, the address of the first instruction.

- This instruction (the value 1940 in hexadecimal) is loaded into the IR and the PC is incremented.
- Note that this process involves the use of a memory address register (MAR) and a memory buffer register (MBR).
- For simplicity, these intermediate registers are not shown.

2. The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded from memory.

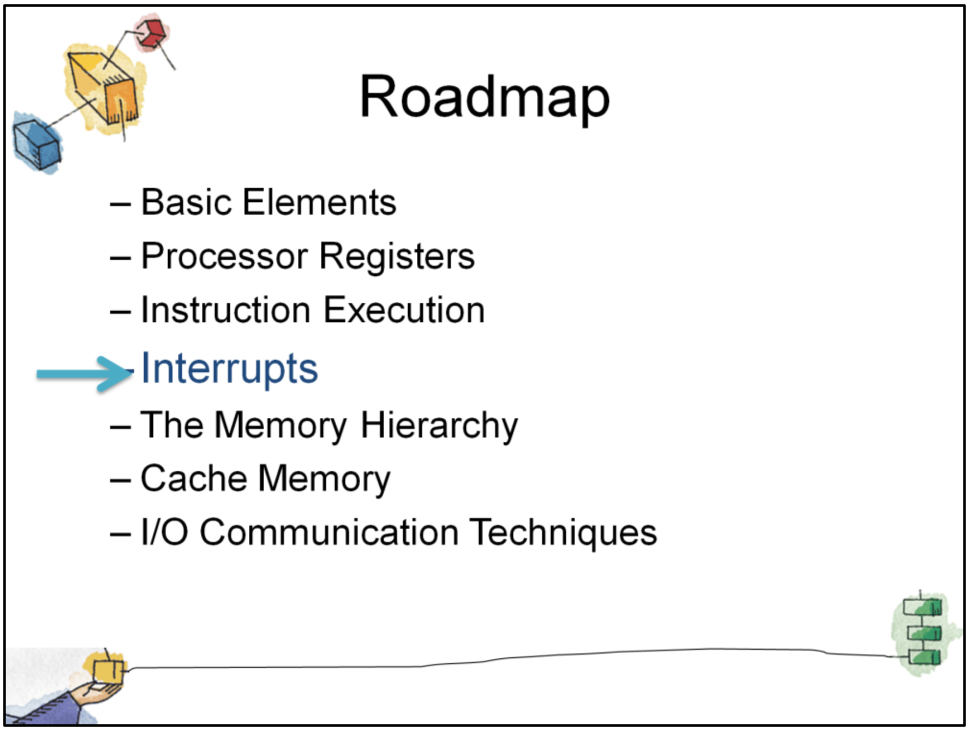
- The remaining 12 bits (three hexadecimal digits) specify the address, which is 940.

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.

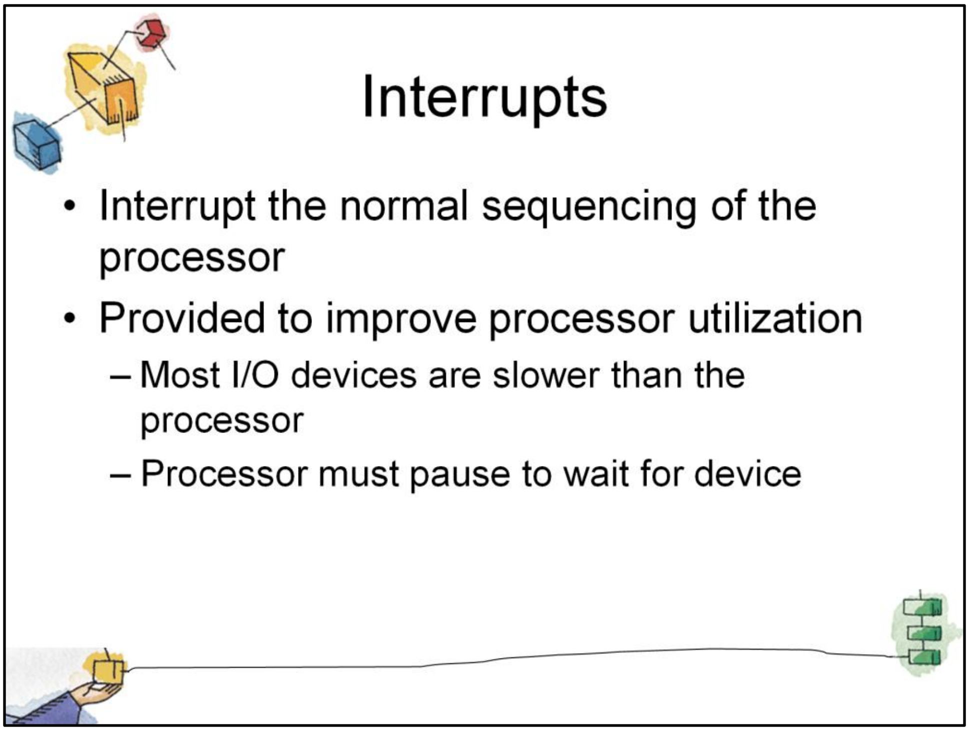
5. The next instruction (2941) is fetched from location 302 and the PC is incremented.

6. The contents of the AC are stored in location 941.



Roadmap

- – Basic Elements
- – Processor Registers
- – Instruction Execution
- – Interrupts
- – The Memory Hierarchy
- – Cache Memory
- – I/O Communication Techniques



Interrupts

- Interrupt the normal sequencing of the processor
- Provided to improve processor utilization
  - Most I/O devices are slower than the processor
  - Processor must pause to wait for device

Virtually all computers provide a mechanism by which other modules (I/O, memory) may interrupt the normal sequencing of the processor.

Interrupts are provided primarily as a way to improve processor utilization.

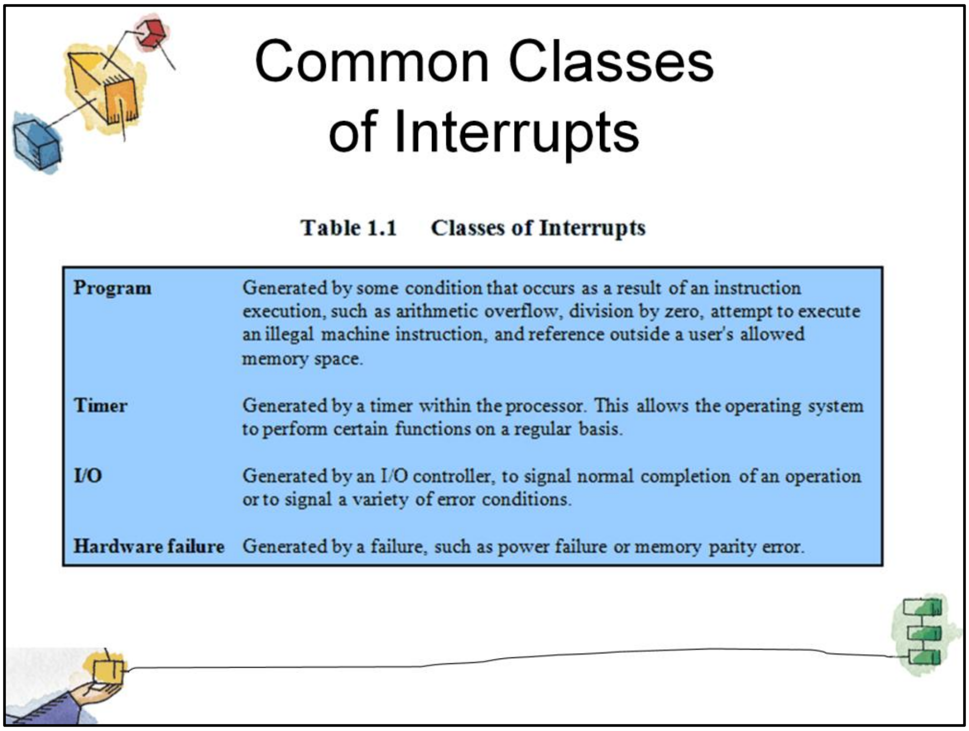
•For example,most I/O devices are much slower than the processor.

Suppose that the processor is transferring data to a printer using the instruction cycle scheme described earlier.

• 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 thousands or even millions of instruction cycles.

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

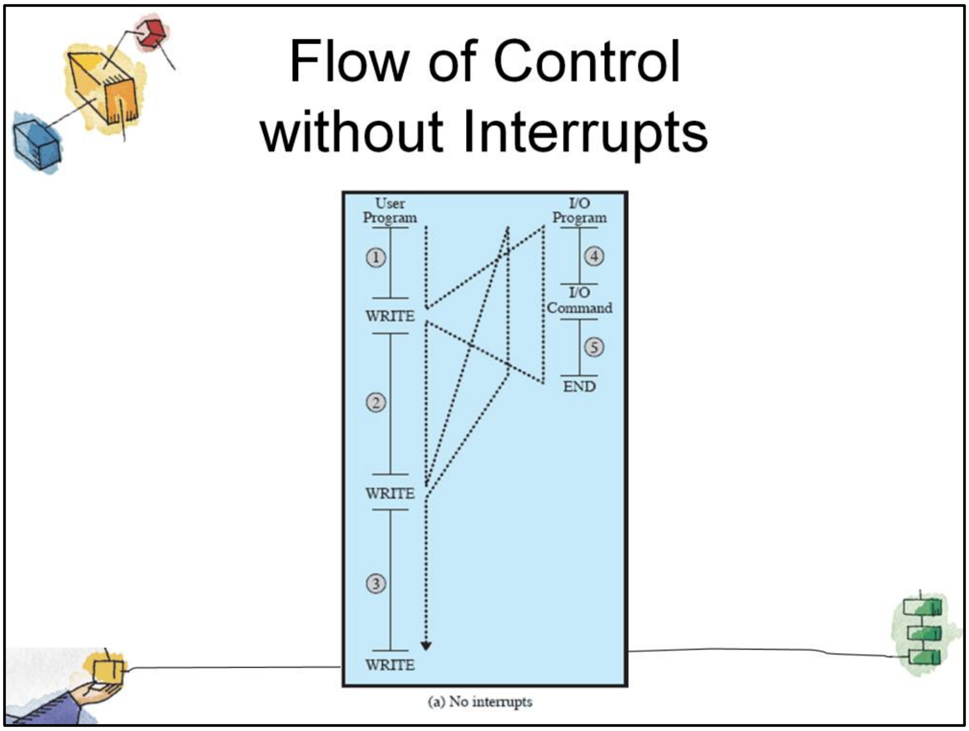


Common Classes of Interrupts

Table 1.1 Classes of Interrupts

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, and 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 or to signal a variety of error conditions.
Hardware failure	Generated by a failure, such as power failure or memory parity error.

Table 1.1 lists the most common classes of interrupts.



Flow of Control without Interrupts

(a) No interrupts

NB: Animated slide showing each stage one at a time.

A) No Interrupts

Figure 1.5a illustrates this previous printer example.

- The user program performs a series of WRITE calls interleaved with processing.
- The solid vertical lines represent segments of code in a program.
- 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 routine that is a system utility and that will perform the actual I/O operation.

The I/O program consists of three sections:

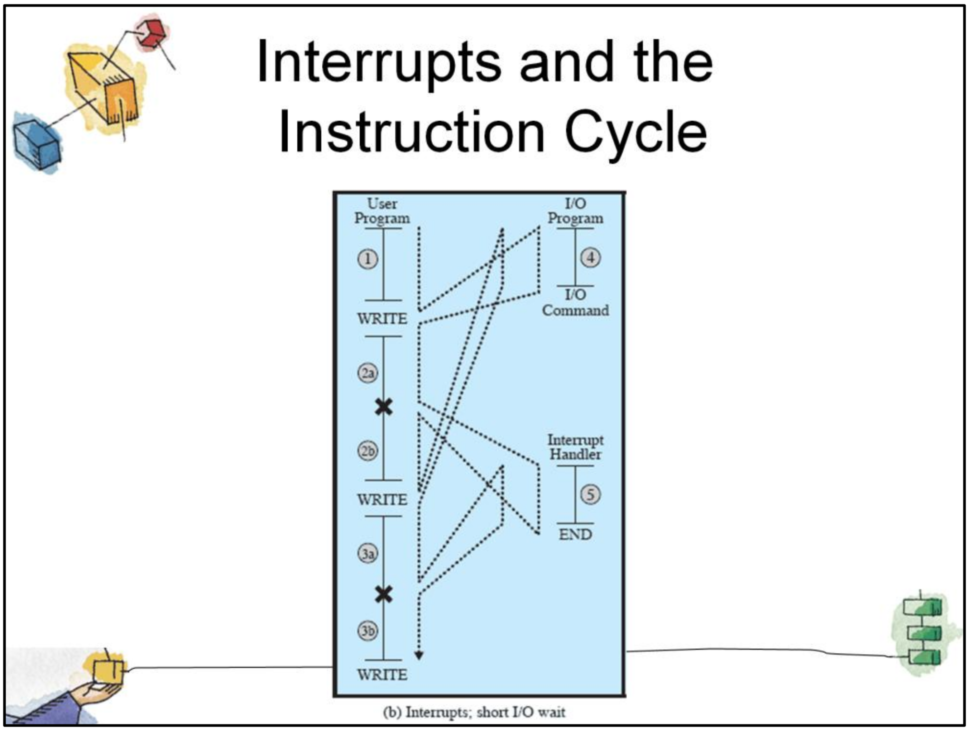
- A sequence of instructions, labeled 4 in the figure, to prepare for the actual I/O operation.
  - This may include copying the data to be output into a special buffer and preparing the parameters for a device command.
- The actual I/O command.
  - Without the use of interrupts, once this command is issued, the program must wait for the I/O device to perform the requested function (or periodically check the status, or poll, the I/O 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.

The dashed line represents the path of execution followed by the processor;

- i.e. , this line shows the sequence in which instructions are executed.
- Thus, after the first WRITE instruction is encountered, the user program is interrupted and execution continues with the I/O program.
- After the I/O program execution is complete, execution resumes in the user program immediately following the WRITE instruction.

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

(b) Interrupts; short I/O wait

With interrupts, the processor can be engaged in executing other instructions while an I/O operation is in progress.

As before, the user program reaches a point at which it makes a system call in the form of a WRITE call.

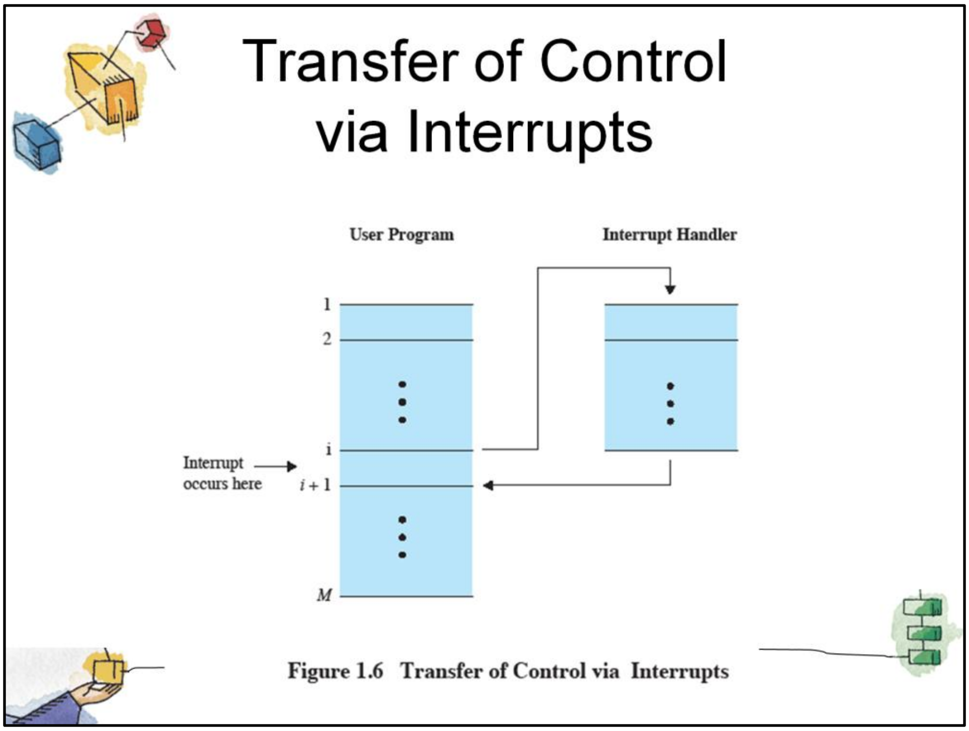
- The I/O program that is invoked in this case consists only of the preparation code and the actual I/O command.
- After these few instructions have been executed, control returns to the user program.
- Meanwhile, the external device is busy accepting data from computer memory and printing it.

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 routine 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 **X** in Figure 1.5b.

- **Note** that an interrupt can occur at any point in the main program, not just at one specific instruction.

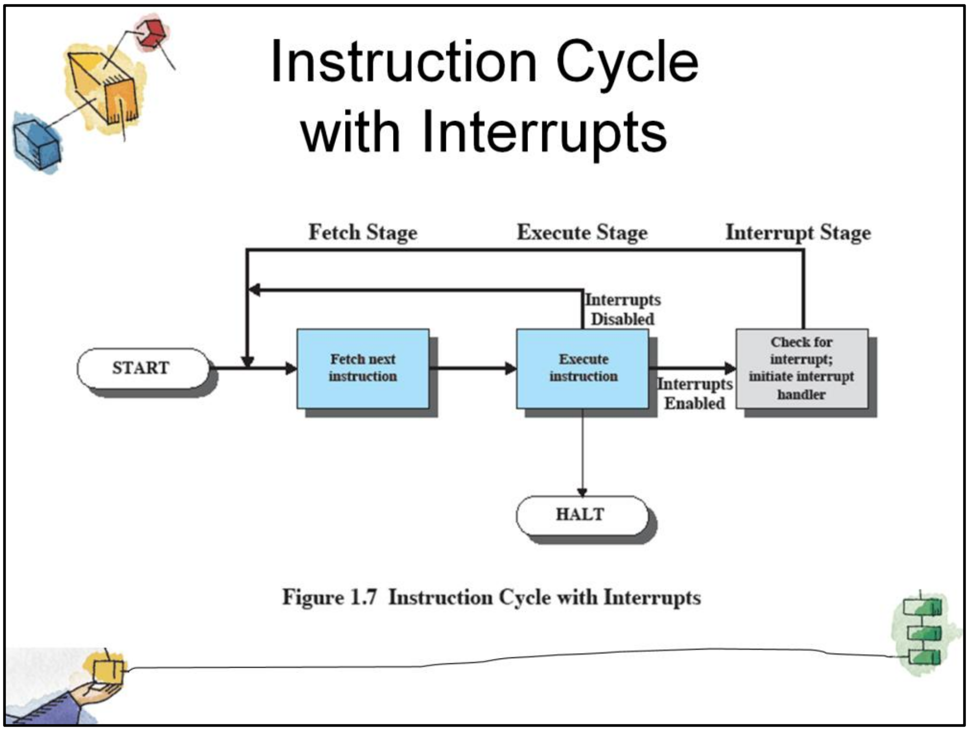


Transfer of Control via Interrupts

Figure 1.6 Transfer of Control via Interrupts

For the user program, an interrupt suspends 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 OS are responsible for suspending the user program and then resuming it at the same point.



Instruction Cycle with Interrupts

Figure 1.7 Instruction Cycle with Interrupts

To accommodate interrupts, an interrupt stage is added to the instruction cycle, as shown here (compare Figure 1.2 earlier).

In the interrupt stage, the processor checks to see if any interrupts have occurred, indicated by the presence of an interrupt signal.

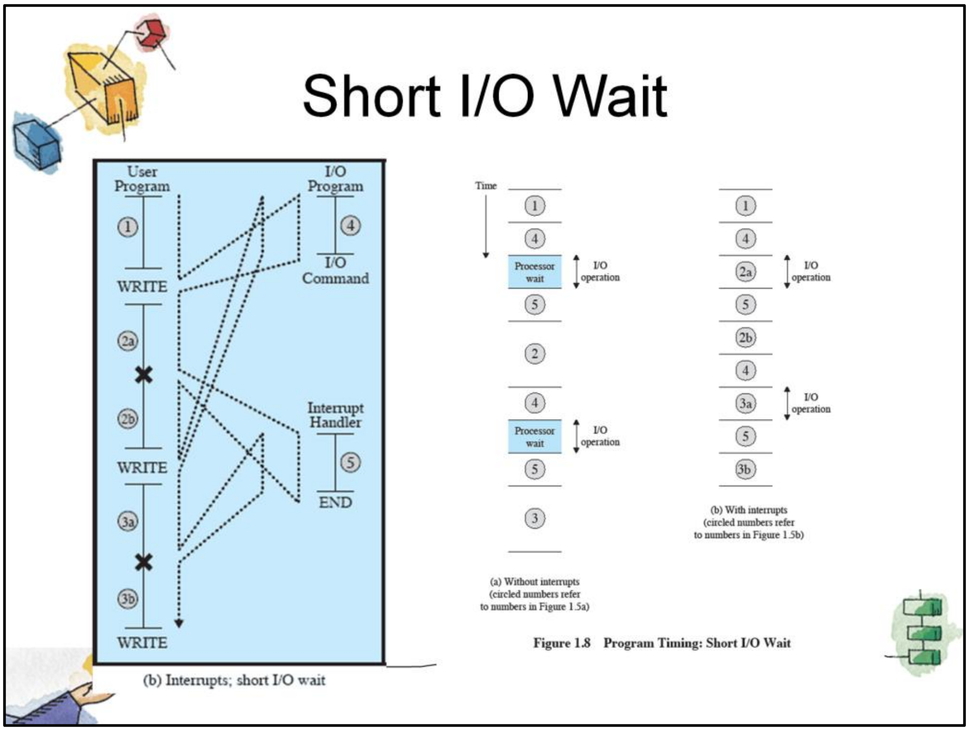
- If no interrupts are pending, the processor proceeds to the fetch stage and fetches the next instruction of the current program.
- If an interrupt is pending, the processor suspends execution of the current program and executes an interrupt-handler routine.

The interrupt-handler routine is generally part of the OS.

- Typically, this routine 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.



Short I/O Wait

(b) Interrupts; short I/O wait

(a) Without interrupts (circled numbers refer to numbers in Figure 1.5a)

(b) With interrupts (circled numbers refer to numbers in Figure 1.5b)

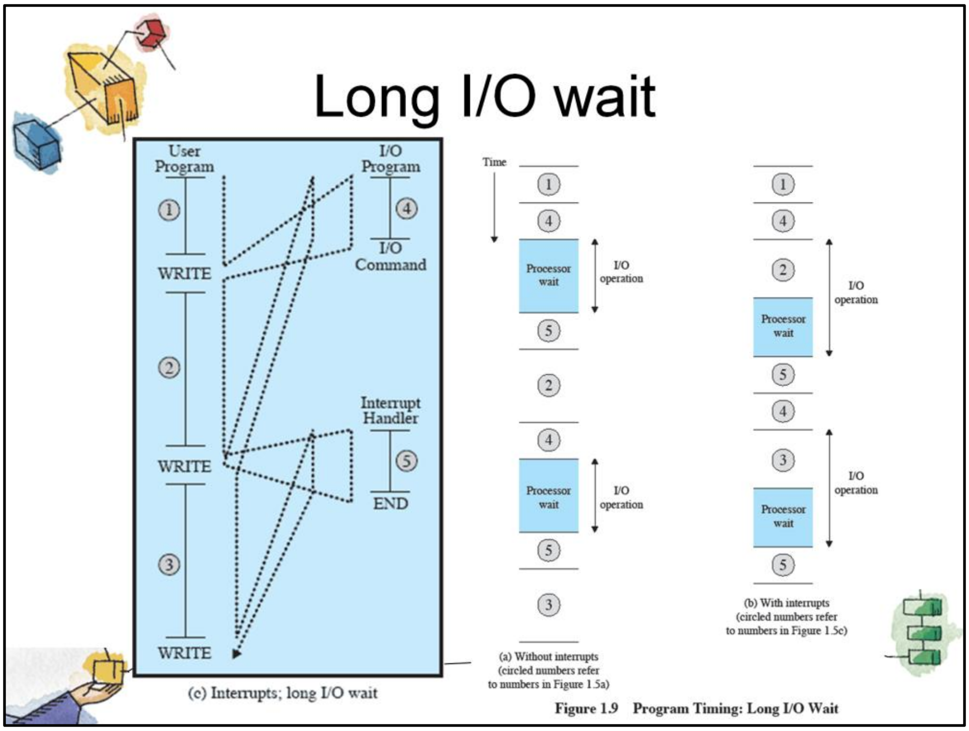
Figure 1.8 Program Timing: Short I/O Wait

To appreciate the gain in efficiency, consider Figure 1.8, which is a timing diagram based on the flow of control in Figures 1.5 a and 1.5b.

Figures 1.5b and 1.8 assume 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.

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.



Long I/O wait

(c) Interrupts; long I/O wait

(a) Without interrupts (circled numbers refer to numbers in Figure 1.5a)

(b) With interrupts (circled numbers refer to numbers in Figure 1.5c)

Figure 1.9 Program Timing: Long I/O Wait

Figure 1.5 c indicates a more typical state of affairs.

In this case, the user program reaches the second WRITE call before the I/O operation spawned by the first call is

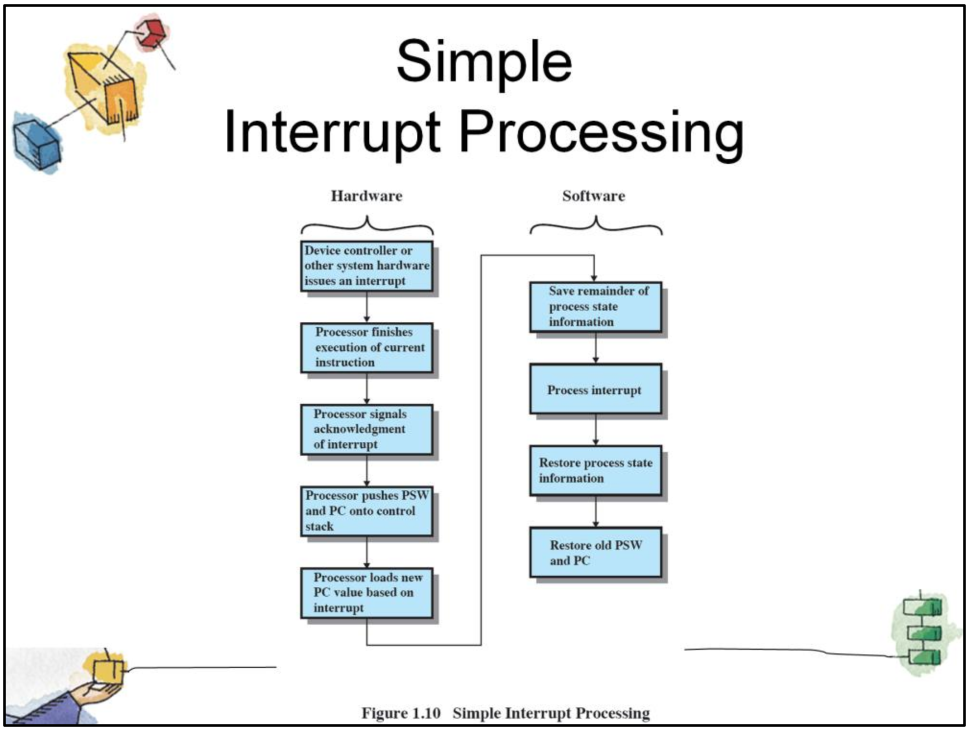
complete.

- 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.

Figure 1.9 shows the timing for this situation with and without the use of interrupts.

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



Simple Interrupt Processing

Hardware

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

Software

Save remainder of process state information

Process interrupt

Restore process state information

Restore old PSW and PC

Figure 1.10 Simple Interrupt Processing

An interrupt triggers a number of events, both in the processor hardware and in software.
This figure 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.

3. The processor tests for a pending interrupt request, 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 next needs to prepare to transfer control to the interrupt routine.

5. The processor then loads the program counter with the entry location of the interrupt-handling routine that will respond to this interrupt.

6. At this point, the program counter and PSW relating to the interrupted program have been saved on the control stack.

- Next slide shows more detail on this step

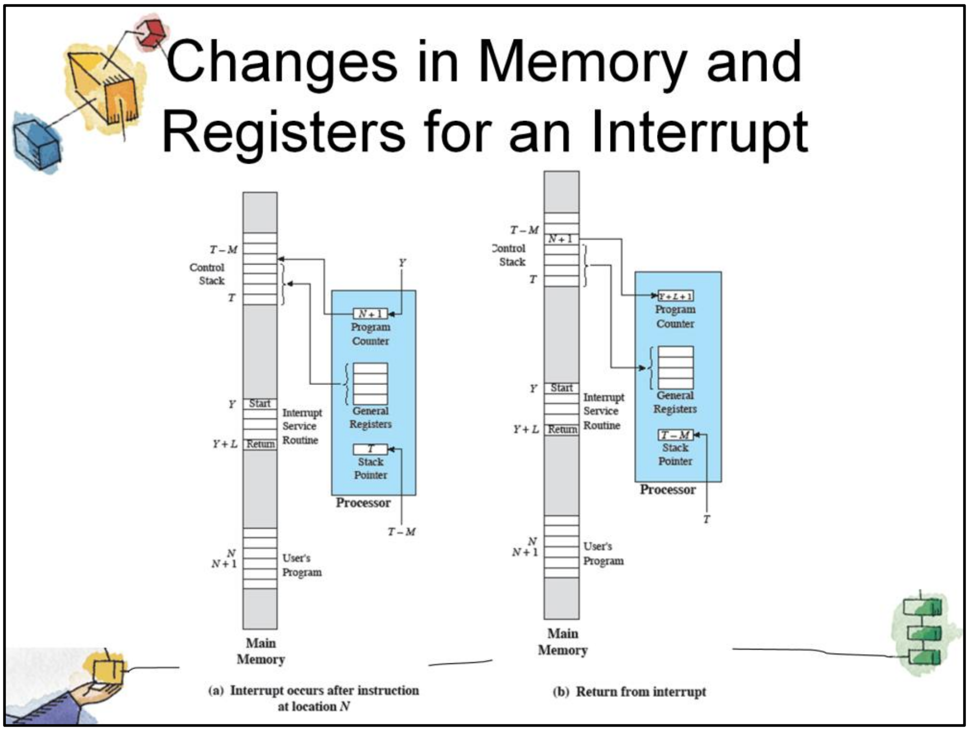
7. The interrupt handler may now proceed to process the interrupt.

8. The saved register values are retrieved from the stack and restored to the registers

9. The final act is to restore the PSW and program counter values from the stack.

It is important to save all of the state information about the interrupted program for later resumption.

- Because the interrupt is not a routine called from the program.
- Rather, the interrupt can occur at any time and therefore at any point in the execution of a user program.
- Its occurrence is unpredictable.



Changes in Memory and Registers for an Interrupt

(a) Interrupt occurs after instruction at location N

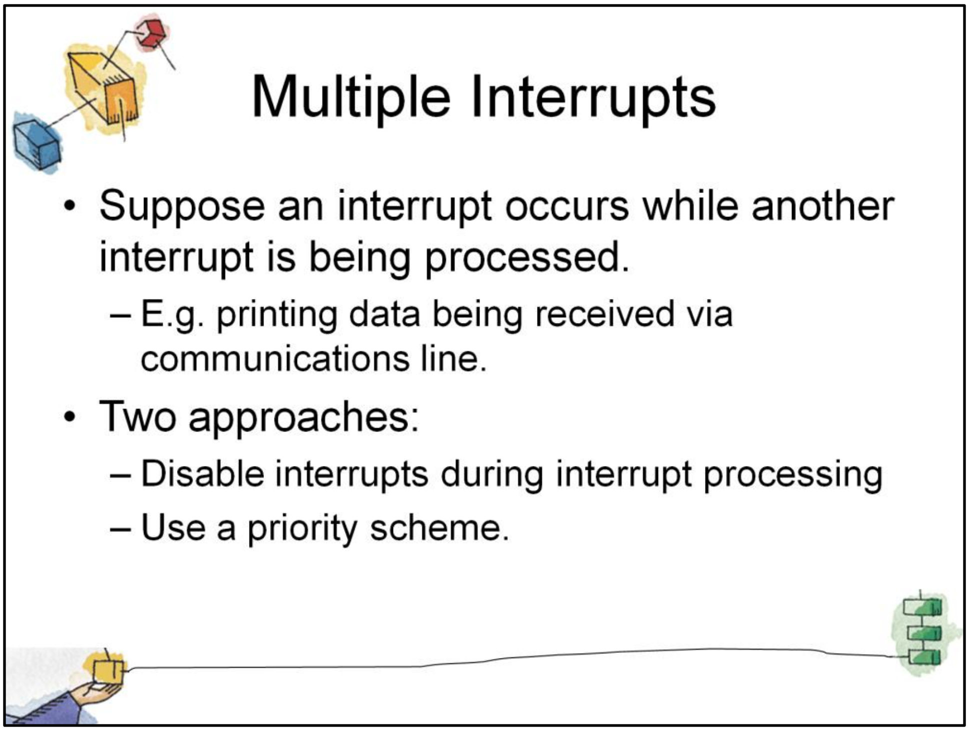
(b) Return from interrupt

From step 6 in previous diagram

In this case, a user program is interrupted after the instruction at location N.

The contents of all of the registers plus the address of the next instruction (N + 1), a total of M words, are pushed onto the control stack.

- The stack pointer is updated to point to the new top of stack, and the program counter is updated to point to the beginning of the interrupt service routine.



Multiple Interrupts

- Suppose an interrupt occurs while another interrupt is being processed.
  - E.g. printing data being received via communications line.
- Two approaches:
  - Disable interrupts during interrupt processing
  - Use a priority scheme.

Suppose that one or more interrupts can occur while an interrupt is being processed.

E.G. , a program may be receiving data from a communications line and printing results at the same time.

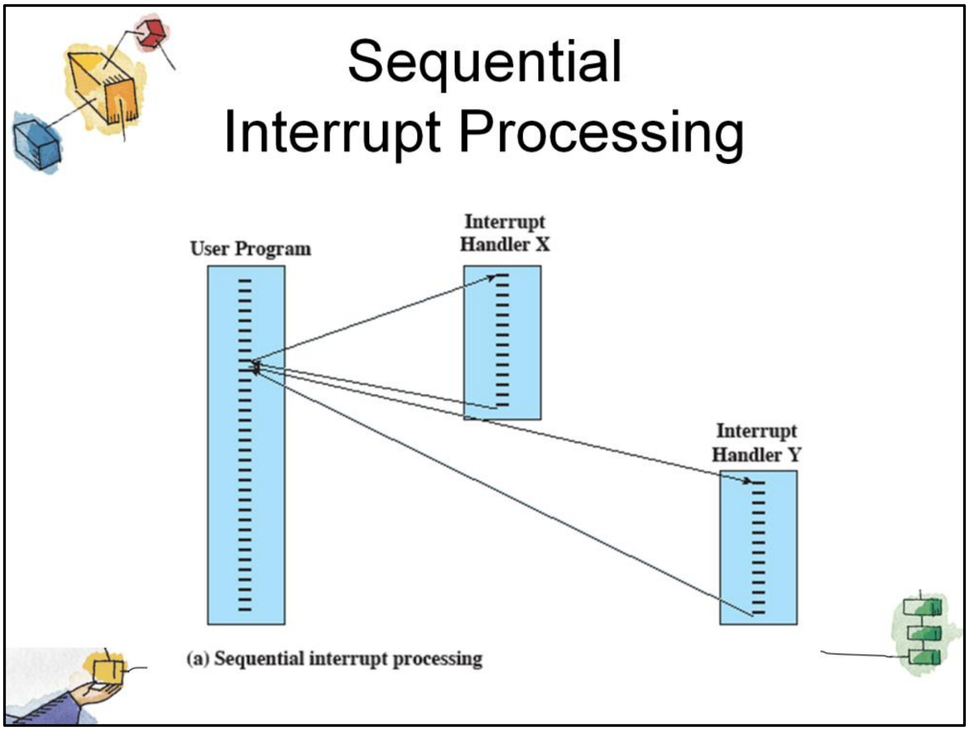
- The printer will generate an interrupt every time that it completes a print operation.
- The communication line controller will generate an interrupt every time a unit of data arrives.

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 ignores any new 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 re-enabled interrupts.

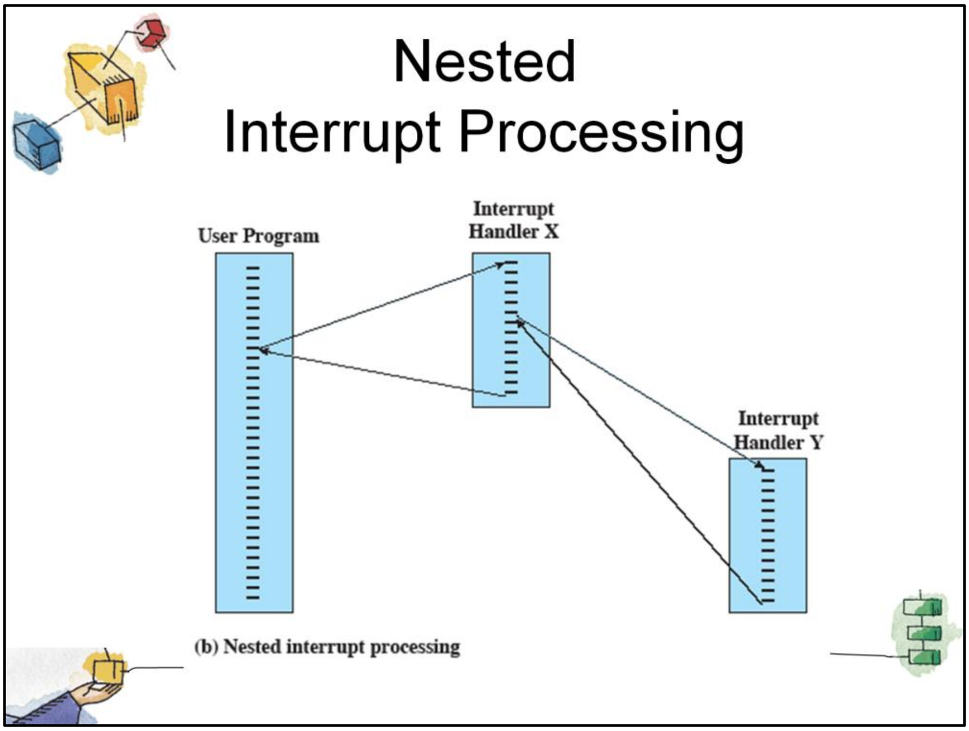
A second approach is to define priorities for interrupts and to allow an interrupt of higher priority to cause a lower-priority interrupt handler to be interrupted.



Sequential Interrupt Processing

(a) Sequential interrupt processing

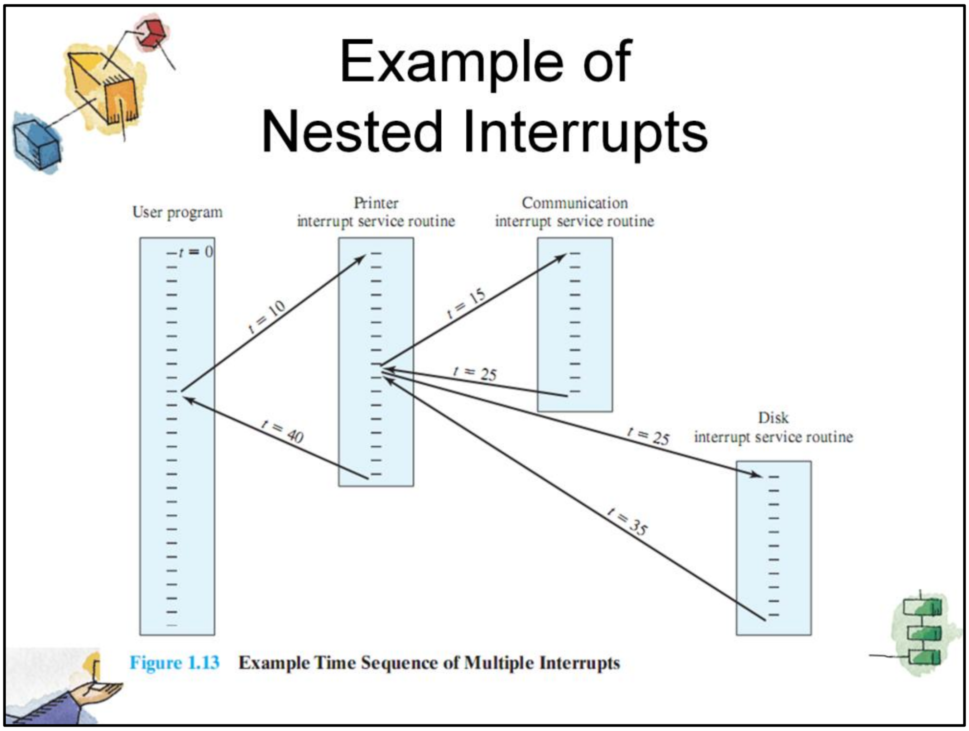
Simple sequential approach to multiple interrupts



Nested Interrupt Processing

(b) Nested interrupt processing

Using priorities to “nest” interrupt processing



Example of Nested Interrupts

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

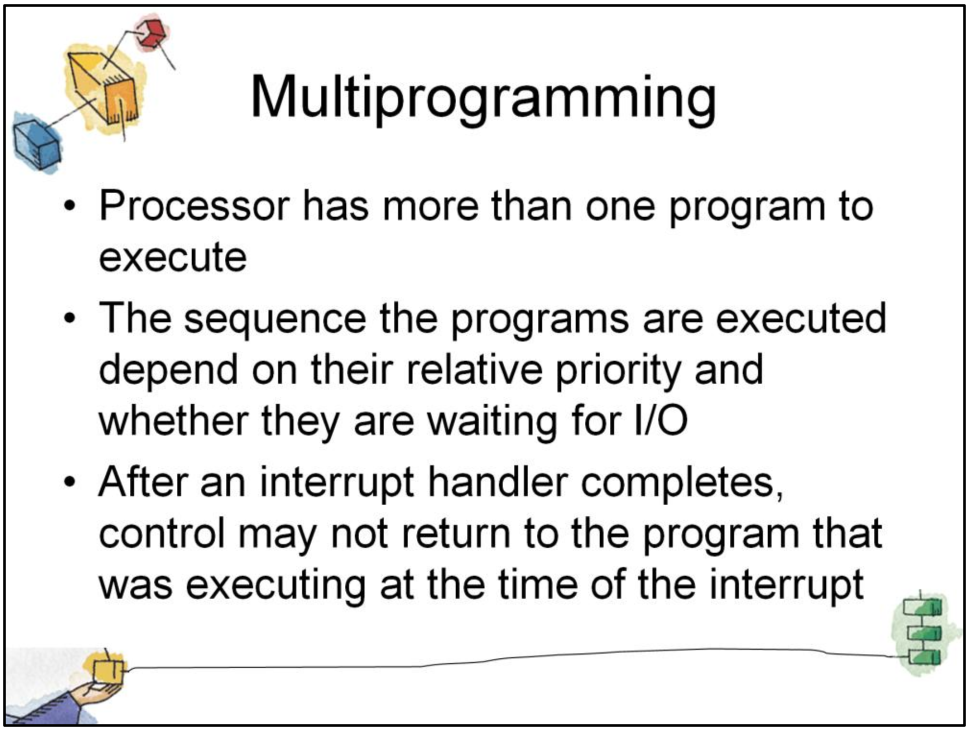
Figure 1.13 **Example Time Sequence of Multiple Interrupts**

As an example of this second approach, consider a system with three I/O devices:

- a printer (priority 2),
- a disk (priority 4), and
- a communications line (priority 5).

This figure illustrates a possible sequence.

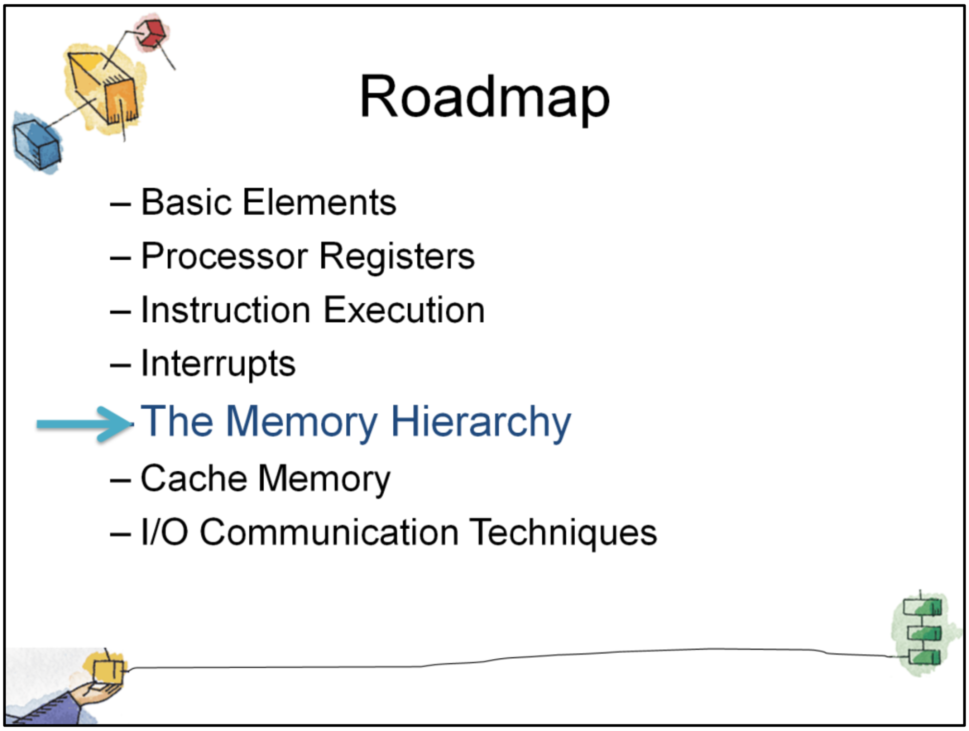
1. A user program begins at *t*= 0.
2. At *t*=10, a printer interrupt occurs;
   - user information is placed on the control stack and execution continues at the printer interrupt service routine (ISR).
3. While this routine is still executing, at *t* =15 a communications interrupt occurs.
   - Because the communications line has higher priority than the printer, the interrupt request is honored.
4. The printer ISR is interrupted, its state is pushed onto the stack, and execution continues at the communications ISR.
5. 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.
6. When the communications ISR is complete (*t*=25), the previous processor state is restored, which is the execution of the printer ISR.
7. However, before even a single instruction in that routine can be executed, the processor honors the higher-priority disk interrupt and transfers control to the disk ISR.
8. Only when that routine is complete (*t*=35) is the printer ISR resumed.
9. When that routine completes (*t*=40), control finally returns to the user program.



Multiprogramming

- Processor has more than one program to execute
- The sequence the programs are executed depend on their relative priority and whether they are waiting for I/O
- After an interrupt handler completes, control may not return to the program that was executing at the time of the interrupt

Even with the use of interrupts, a processor may not be used very efficiently.

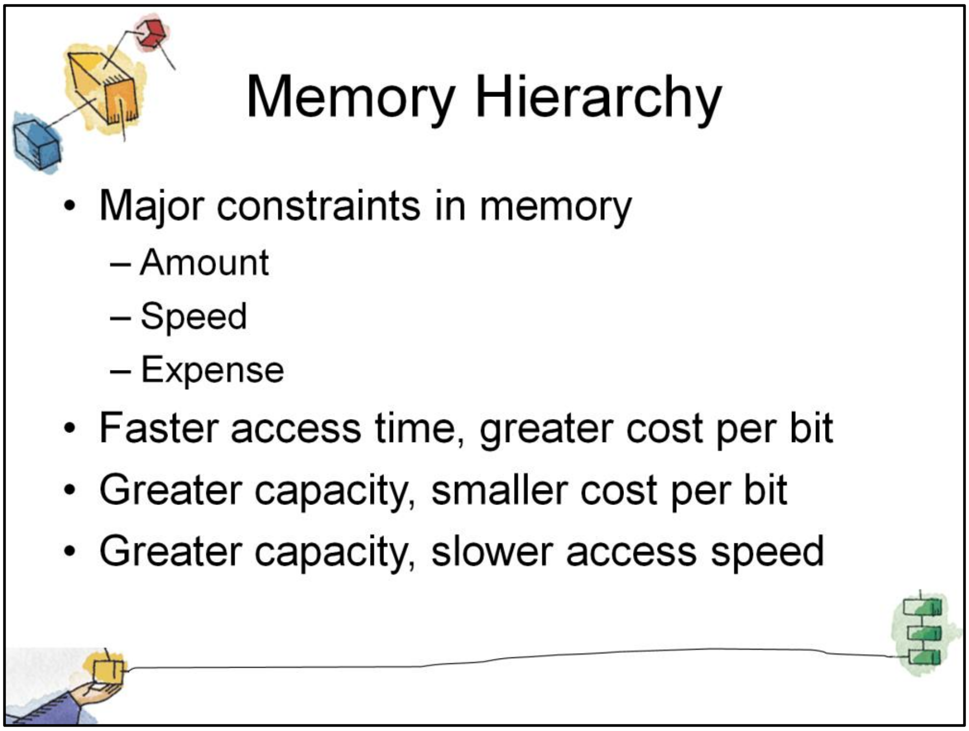


Roadmap

- Basic Elements
- Processor Registers
- Instruction Execution
- Interrupts

→ The Memory Hierarchy

- Cache Memory
- I/O Communication Techniques



Memory Hierarchy

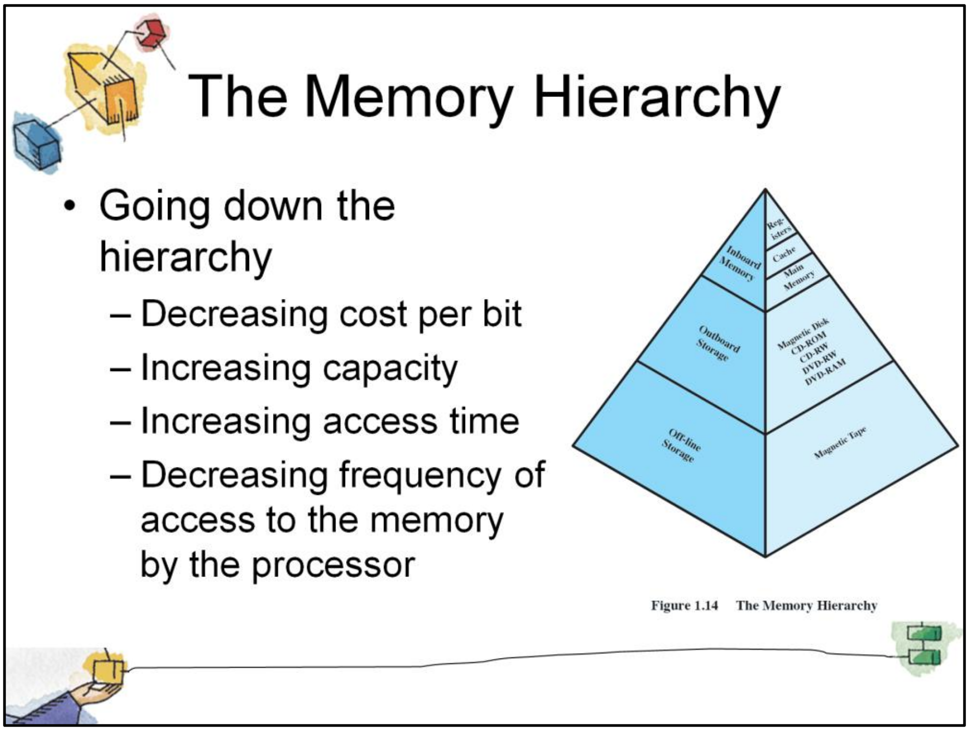
- Major constraints in memory
  - Amount
  - Speed
  - Expense
- Faster access time, greater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity, slower access speed

There is a tradeoff among the three key characteristics of memory:

- namely, capacity, access time, and cost.

A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

- Faster access time, greater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity, slower access speed



The Memory Hierarchy

- Going down the hierarchy
  - Decreasing cost per bit
  - Increasing capacity
  - Increasing access time
  - Decreasing frequency of access to the memory by the processor

Figure 1.14 The Memory Hierarchy

NOTE: Onclick, Animated diagram slides to centre and grows

A typical hierarchy is illustrated in this figure.

As one goes down the hierarchy, the following occur:

a. Decreasing cost per bit

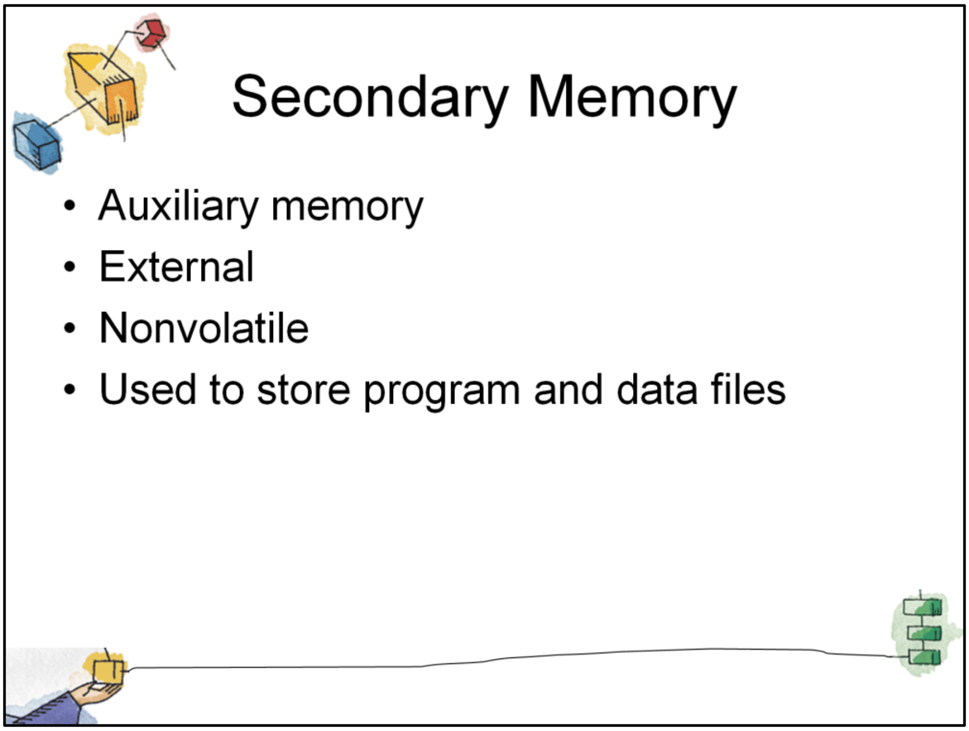
b. Increasing capacity

c. Increasing access time

d. Decreasing frequency of access to the memory by the processor

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories.

The key to the success of this organization decreasing frequency of access at lower levels.

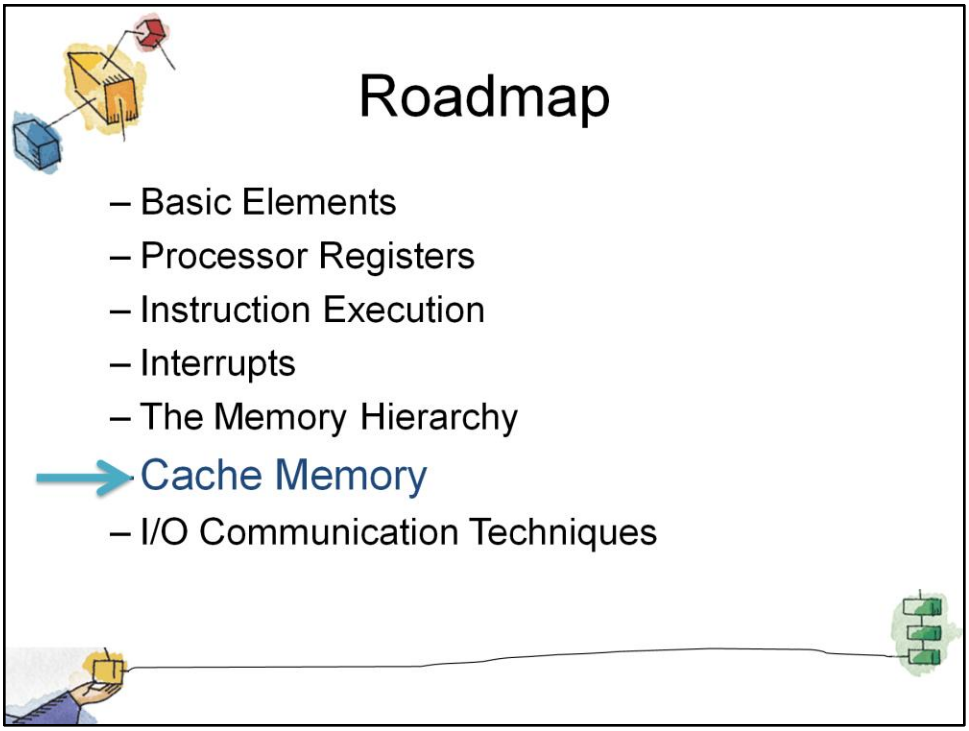


Secondary Memory

- Auxiliary memory
- External
- Nonvolatile
- Used to store program and data files

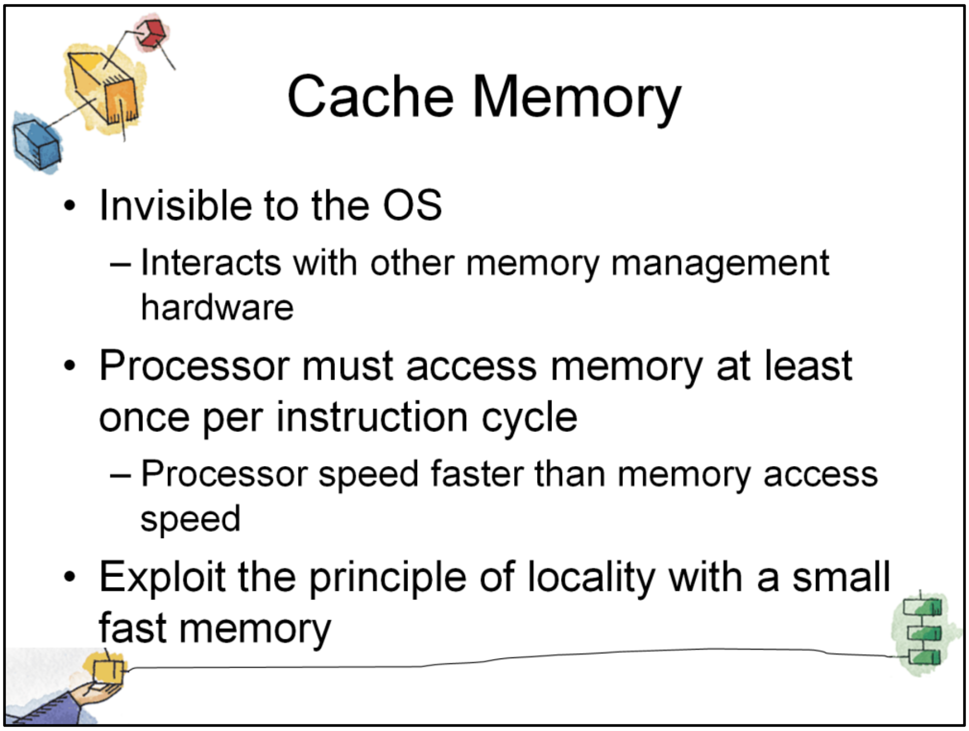
External, nonvolatile memory is also referred to as secondary memory or auxiliary memory.

These are used to store program and data files and are usually visible to the programmer only in terms of files and records, as opposed to individual bytes or words.



Roadmap

- Basic Elements
- Processor Registers
- Instruction Execution
- Interrupts
- The Memory Hierarchy
- Cache Memory
- I/O Communication Techniques



Cache Memory

- Invisible to the OS
  - Interacts with other memory management hardware
- Processor must access memory at least once per instruction cycle
  - Processor speed faster than memory access speed
- Exploit the principle of locality with a small fast memory

Although cache memory is invisible to the OS, it interacts with other memory management hardware.

On all instruction cycles, the processor accesses memory at least once,

- to fetch the instruction, and often one or more additional times, to fetch operands and/or store results.

The rate at which the processor can execute instructions is clearly limited by the memory cycle time

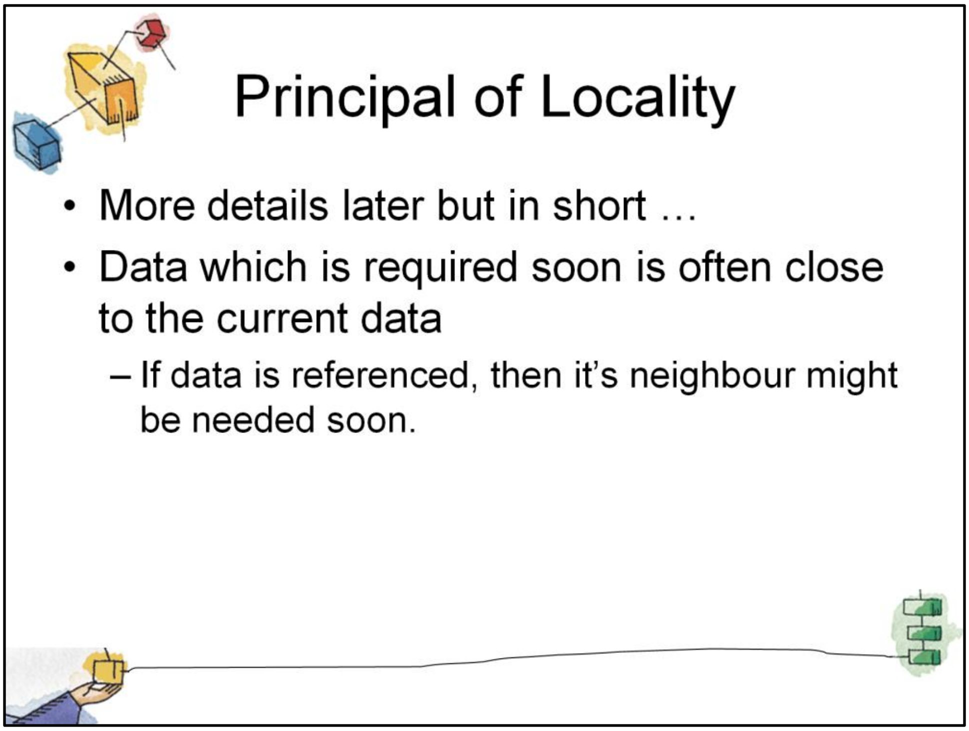
- i.e. the time it takes to read one word from or write one word to memory.

This limitation has been a significant problem because of the persistent mismatch between processor and main memory speeds:

- Over the years, processor speed has consistently increased more rapidly than memory access speed.
- We are faced with a tradeoff among speed, cost, and size.

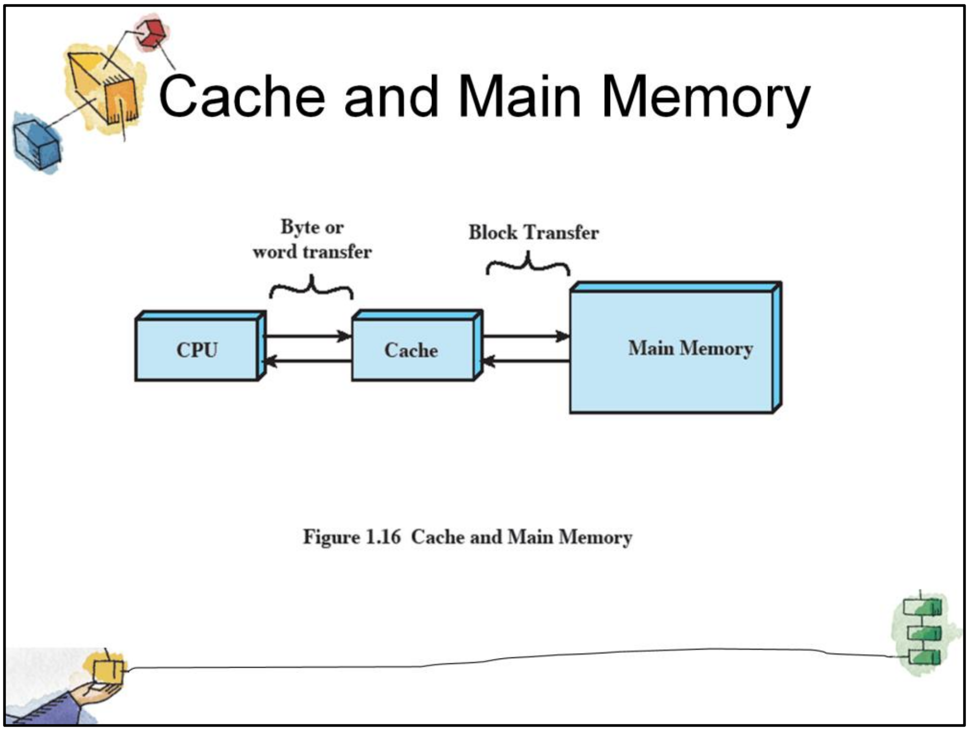
Ideally, main memory should be built with the same technology as that of the processor registers, giving memory cycle times comparable to processor cycle times.

- This has always been too expensive a strategy.
- The solution is to exploit the principle of locality by providing a small, fast memory between the processor and main memory, namely the cache.



Principal of Locality

- More details later but in short …
- Data which is required soon is often close to the current data
  - If data is referenced, then it’s neighbour might be needed soon.



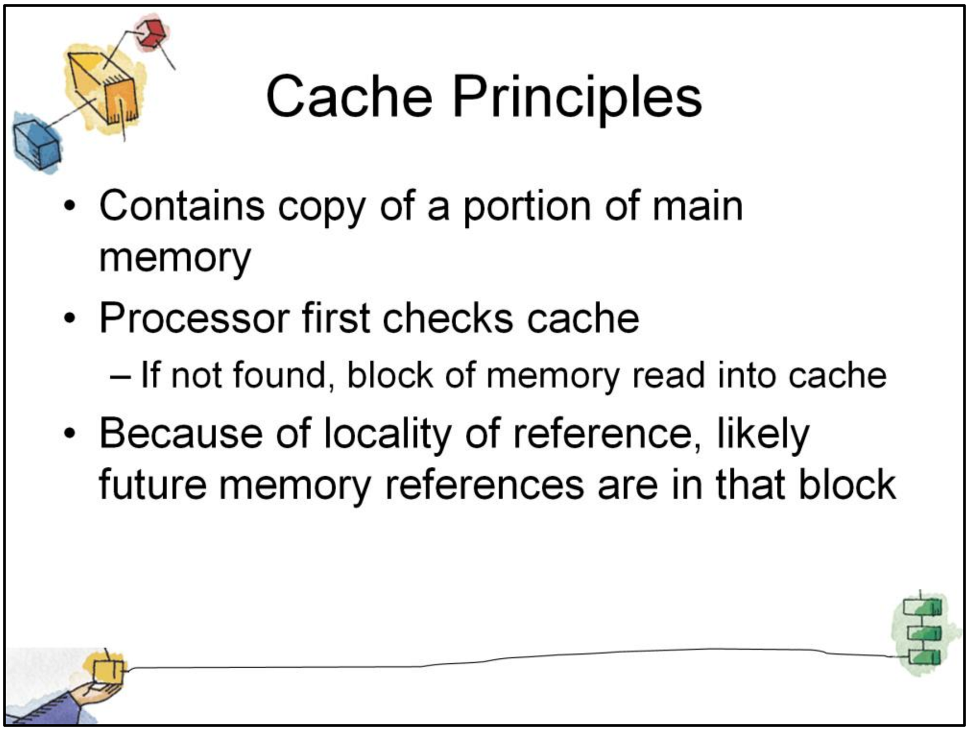
Cache and Main Memory

Figure 1.16 Cache and Main Memory

Cache memory is intended to provide memory access time approaching that of the fastest memories available and at the same time support a large memory size that has the price of less expensive types of semiconductor memories.

There is a relatively large and slow main memory together with a smaller, faster cache memory.

- The cache contains a copy of a portion of main memory.



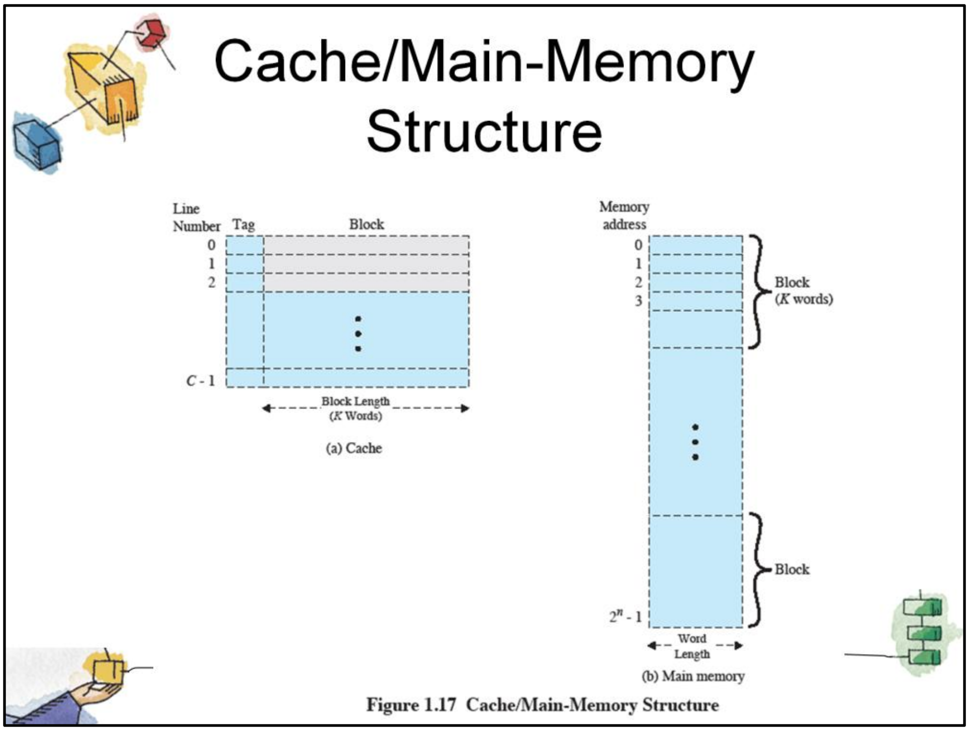
Cache Principles

- Contains copy of a portion of main memory
- Processor first checks cache
  - If not found, block of memory read into cache
- Because of locality of reference, likely future memory references are in that block

When the processor attempts to read a byte or word of memory, a check is made to determine if the byte or word is in the cache.

- If so, the byte or word is delivered to the processor.
- If not, a block of main memory, consisting of some fixed number of bytes, is read into the cache and then the byte or word is delivered to the processor.

Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single memory reference, it is likely that many of the near-future memory references will be to other bytes in the block.



Cache/Main-Memory Structure

Figure 1.17 Cache/Main-Memory Structure

Figure 1.17 depicts the structure of a cache/main memory system.

Main memory consists of up to 2^n addressable words, with each word having a unique n-bit address.

For mapping purposes, this memory is considered to consist of a number of fixed-length blocks of K words each.

- •i.e., there are $M=2^n/K$ blocks.

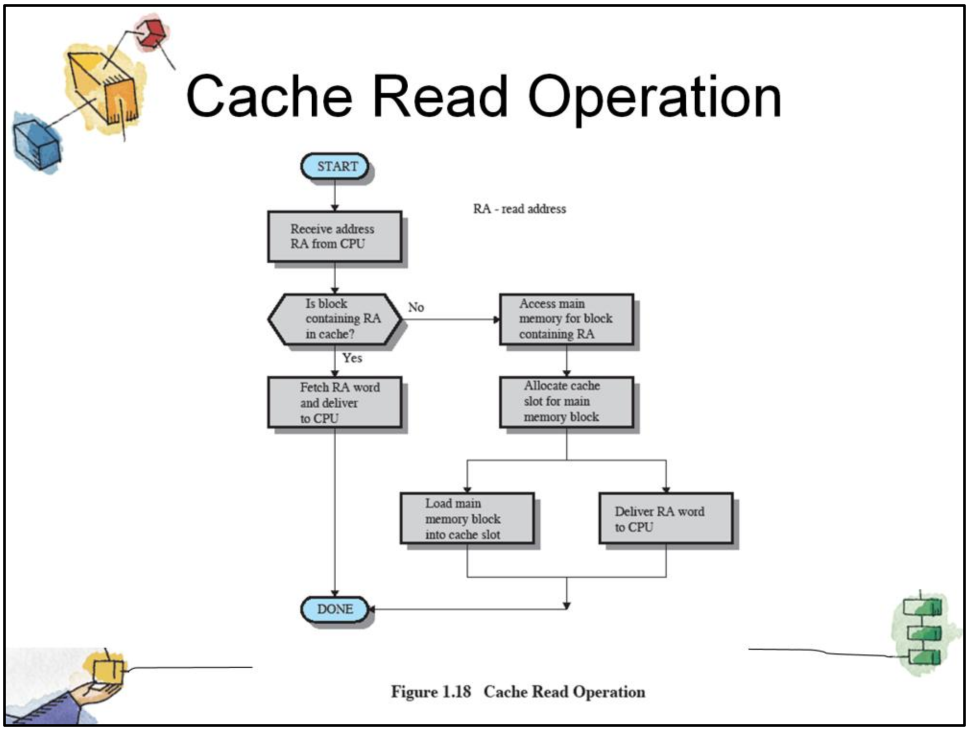
Cache consists of *C* slots (also referred to as lines) of *K* words each, and the number of slots is considerably less than the number of main memory blocks (*C* << *M*).

Some subset of the blocks of main memory resides in the slots of the cache.

- If a word in a block of memory that is not in the cache is read, that block is transferred to one of the slots of the cache.

Because there are more blocks than slots, an individual slot cannot be uniquely and permanently dedicated to a particular block.

- Therefore, each slot includes a tag that identifies which particular block is currently being stored.
- The tag is usually some number of higher-order bits of the address and refers to all addresses that begin with that sequence of bits.



Cache Read Operation

START
Receive address RA from CPU
RA - read address
Is block containing RA in cache?
No
Yes
Access main memory for block containing RA
Fetch RA word and deliver to CPU
Allocate cache slot for main memory block
Load main memory block into cache slot
Deliver RA word to CPU
DONE

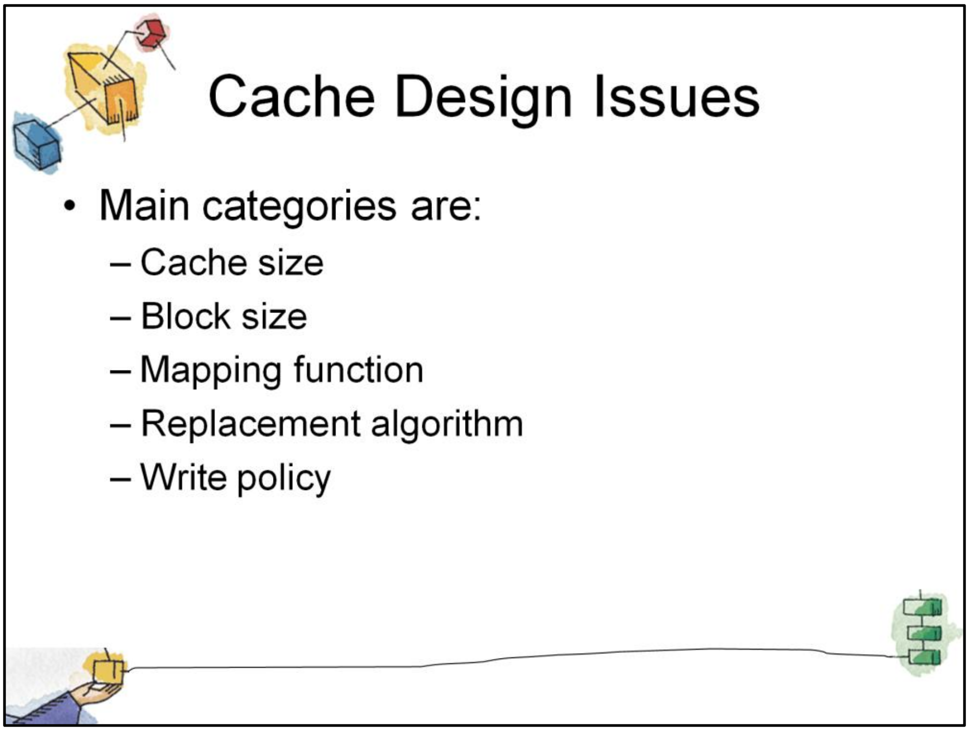
Figure 1.18 Cache Read Operation

Figure 1.18 illustrates the read operation.

The processor generates the address, RA, of a word to be read.

If the word is contained in the cache,

- it is delivered to the processor.
- Otherwise, the block containing that word is loaded into the cache and the word is delivered to the processor.



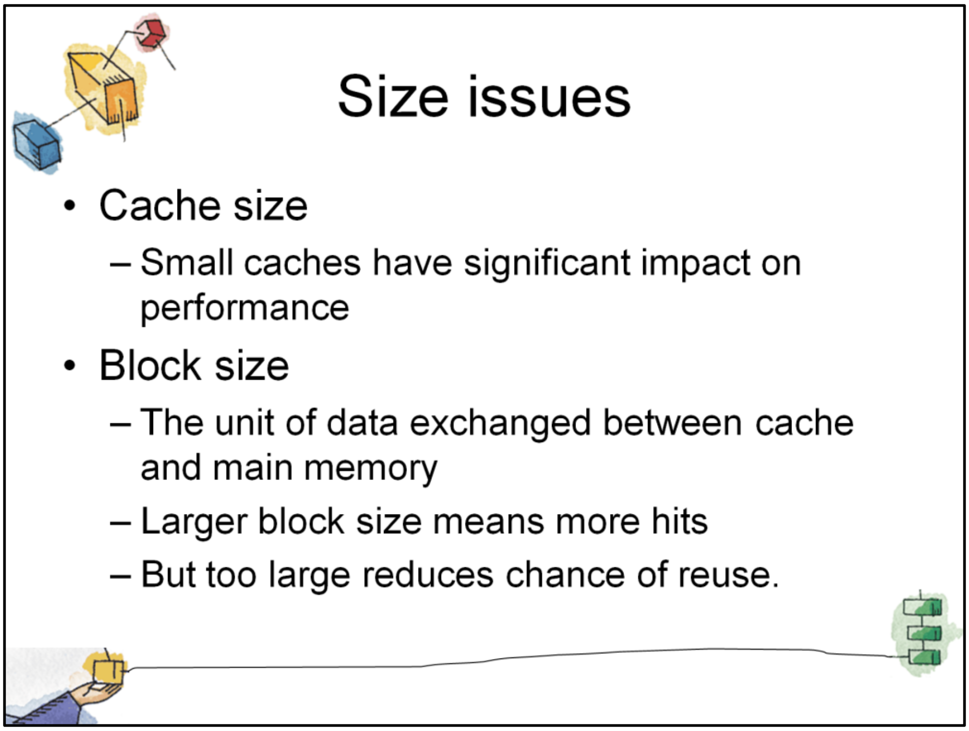
Cache Design Issues

- Main categories are:
  - Cache size
  - Block size
  - Mapping function
  - Replacement algorithm
  - Write policy

We will see that similar design issues must be addressed in dealing with virtual memory and disk cache design.

They fall into the following categories:

- Cache size
- Block size
- Mapping function
- Replacement algorithm
- Write policy



Size issues

- Cache size
  - Small caches have significant impact on performance
- Block size
  - The unit of data exchanged between cache and main memory
  - Larger block size means more hits
  - But too large reduces chance of reuse.

We have already dealt with the issue of **cache size**.

- It turns out that reasonably small caches can have a significant impact on performance.

Another size issue is that of **block size**:

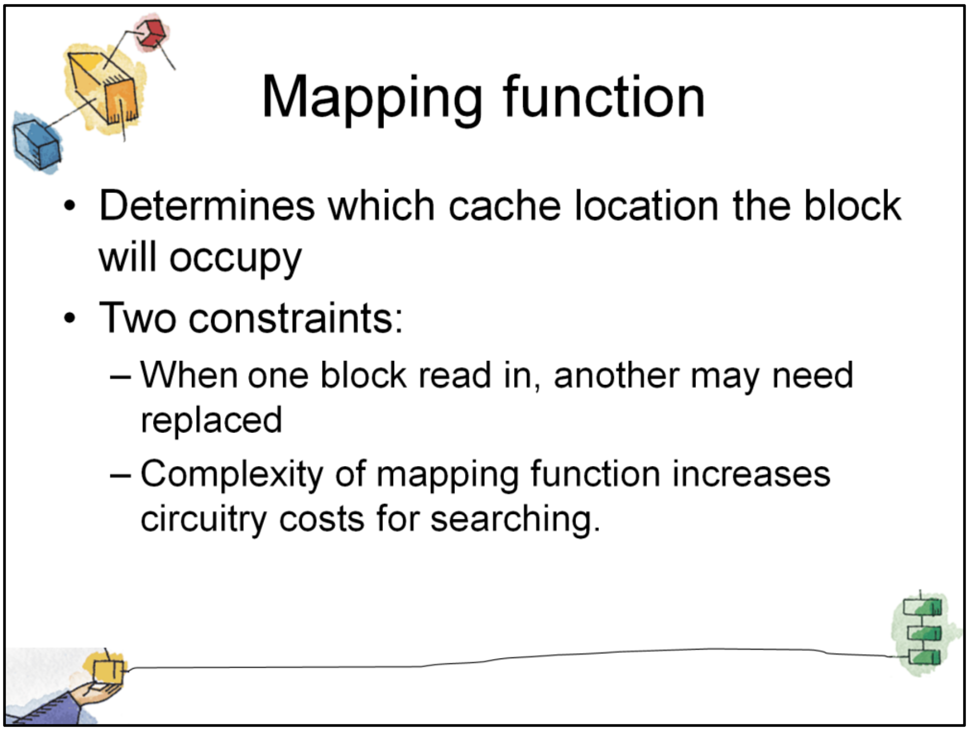
- the unit of data exchanged between cache and main memory.

As the block size increases from very small to larger sizes, the hit ratio will at first increase because of the principle of locality:

- the high probability that data in the vicinity of a referenced word are likely to be referenced in the near future.

As the block size increases, more useful data are brought into the cache.

- The hit ratio will begin to decrease.
- However, as the block becomes even bigger and the probability of using the newly fetched data becomes less than the probability of reusing the data that have to be moved out of the cache to make room for the new block.



Mapping function

- Determines which cache location the block will occupy
- Two constraints:
  - – When one block read in, another may need replaced
  - – Complexity of mapping function increases circuitry costs for searching.

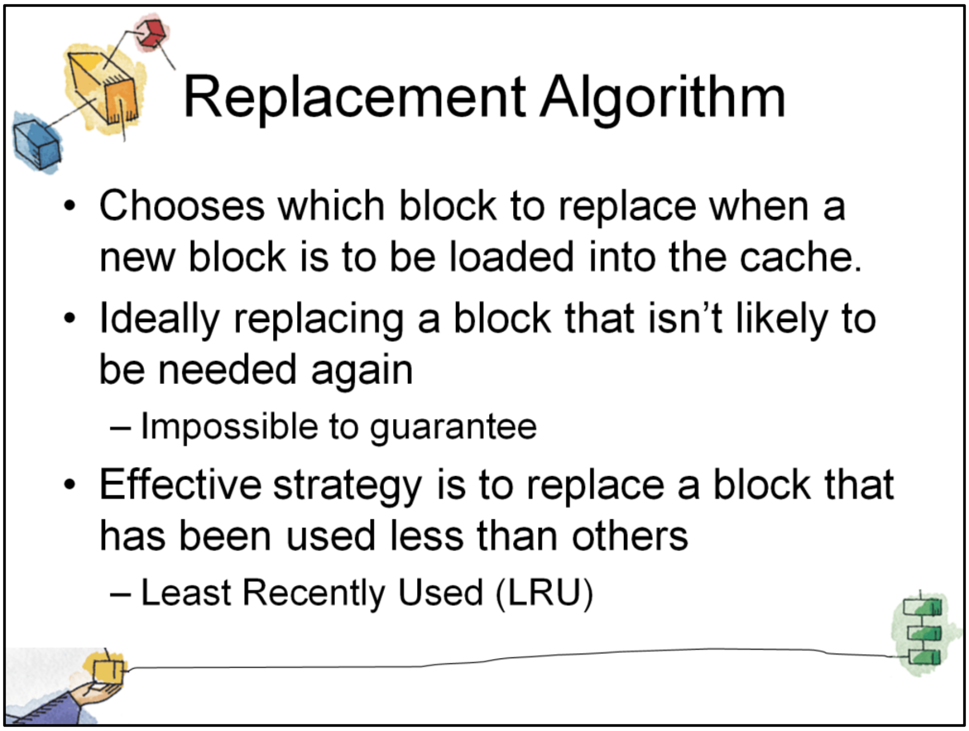
When a new block of data is read into the cache, the mapping function determines which cache location the block will occupy.

Two constraints affect the design of the mapping function.

First, when one block is read in, another may have to be replaced.

- We would like to do this in such a way as to minimize the probability that we will replace a block that will be needed in the near future.
- The more flexible the mapping function, the more scope we have to design a replacement algorithm to maximize the hit ratio.

Second, the more flexible the mapping function, the more complex is the circuitry required to search the cache to determine if a given block is in the cache.



Replacement Algorithm

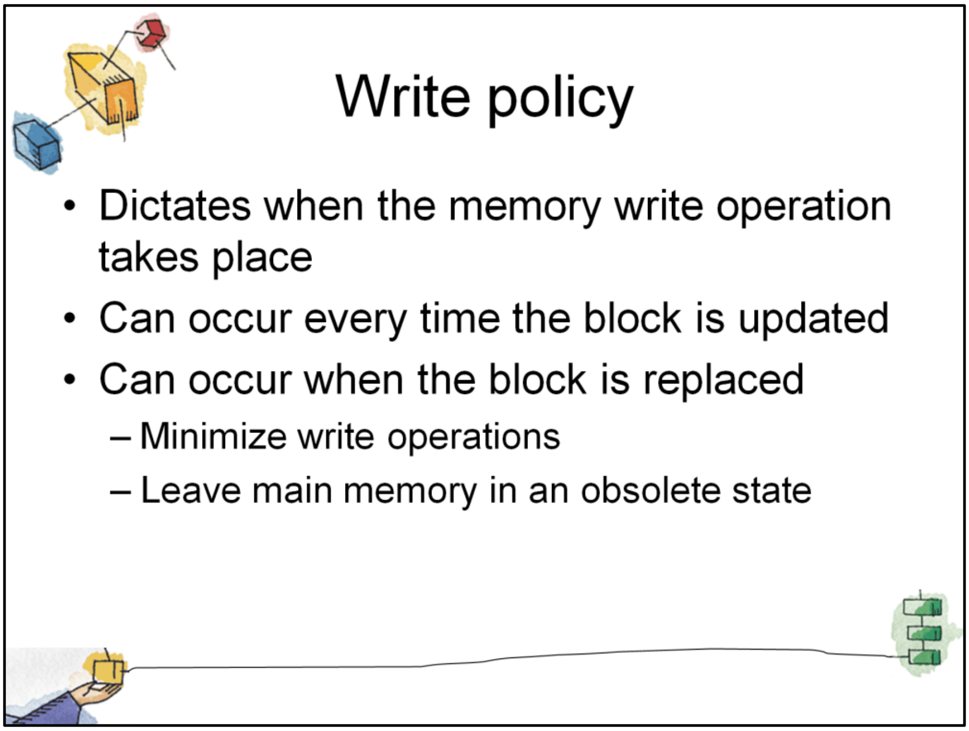
- Chooses which block to replace when a new block is to be loaded into the cache.
- Ideally replacing a block that isn’t likely to be needed again
  - Impossible to guarantee
- Effective strategy is to replace a block that has been used less than others
  - Least Recently Used (LRU)

The replacement algorithm chooses, within the constraints of the mapping function, which block to replace when a new block is to be loaded into the cache and the cache already has all slots filled with other blocks.

We would like to replace the block that is least likely to be needed again in the near future.

- Although it is impossible to identify such a block,
- a reasonably effective strategy is to replace the block that has been in the cache longest with no reference to it.
- This policy is referred to as the least-recently-used (LRU) algorithm.

Hardware mechanisms are needed to identify the least-recently-used block.



Write policy

- Dictates when the memory write operation takes place
- Can occur every time the block is updated
- Can occur when the block is replaced
  - Minimize write operations
  - Leave main memory in an obsolete state

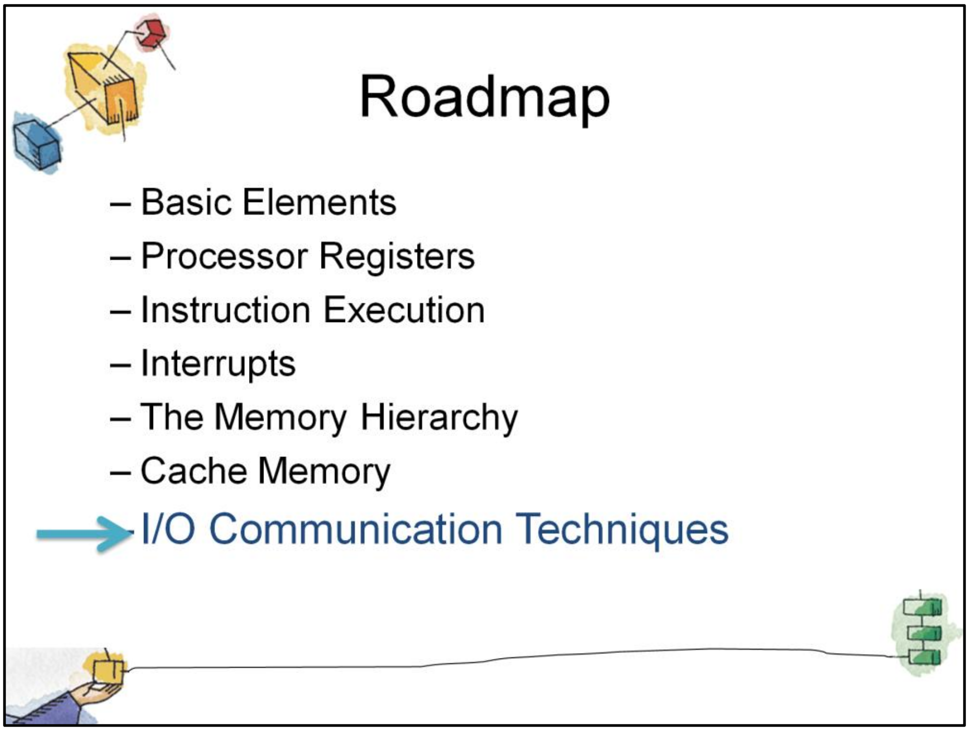
If the contents of a block in the cache are altered, then it is necessary to write it back to main memory before replacing it.

The write policy dictates when the memory write operation takes place.

- At one extreme, the writing can occur every time that the block is updated.
- At the other extreme, the writing occurs only when the block is replaced.

The latter policy minimizes memory write operations but leaves main memory in an obsolete state.

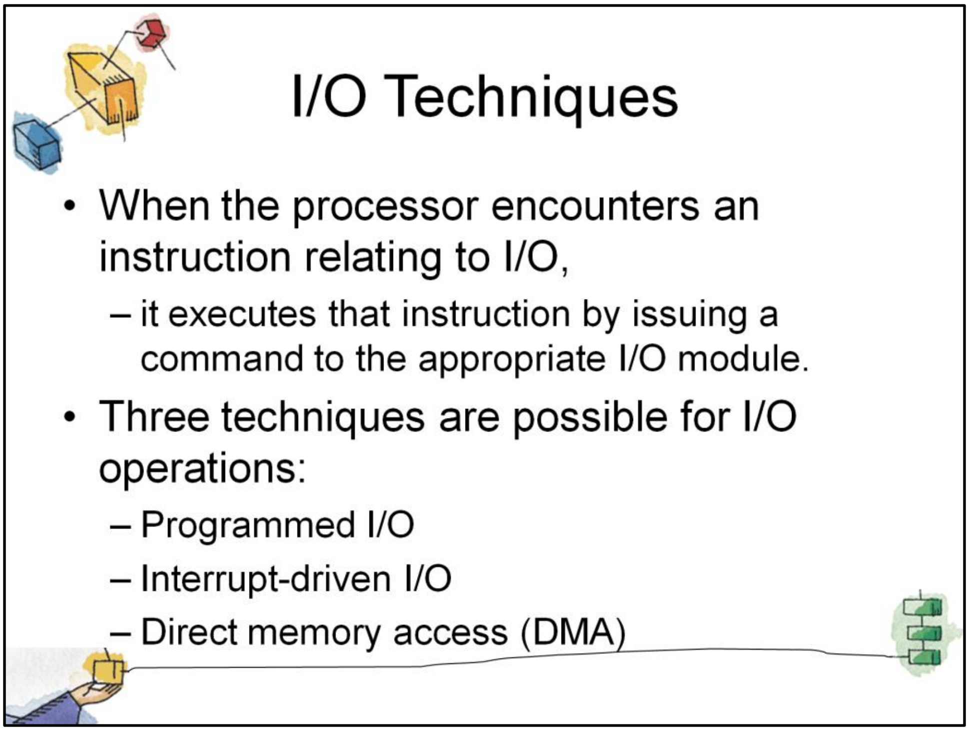
- This can interfere with multiple-processor operation and with direct memory access by I/O hardware modules.



Roadmap

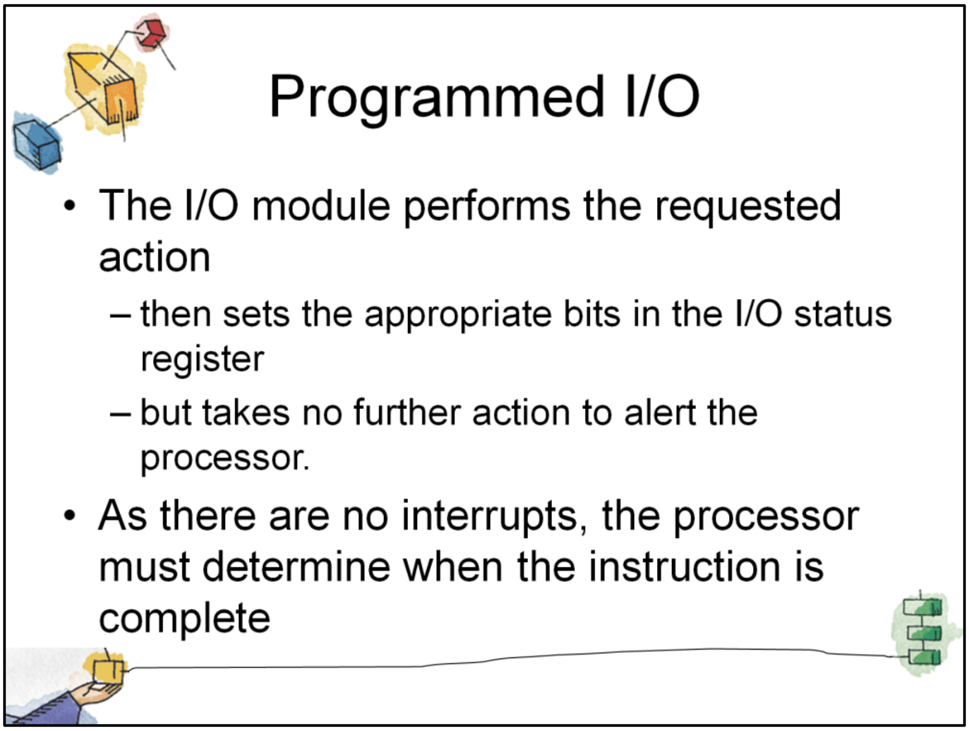
- Basic Elements
- Processor Registers
- Instruction Execution
- Interrupts
- The Memory Hierarchy
- Cache Memory

→ I/O Communication Techniques



I/O Techniques

- When the processor encounters an instruction relating to I/O,
  - it executes that instruction by issuing a command to the appropriate I/O module.
- Three techniques are possible for I/O operations:
  - Programmed I/O
  - Interrupt-driven I/O
  - Direct memory access (DMA)



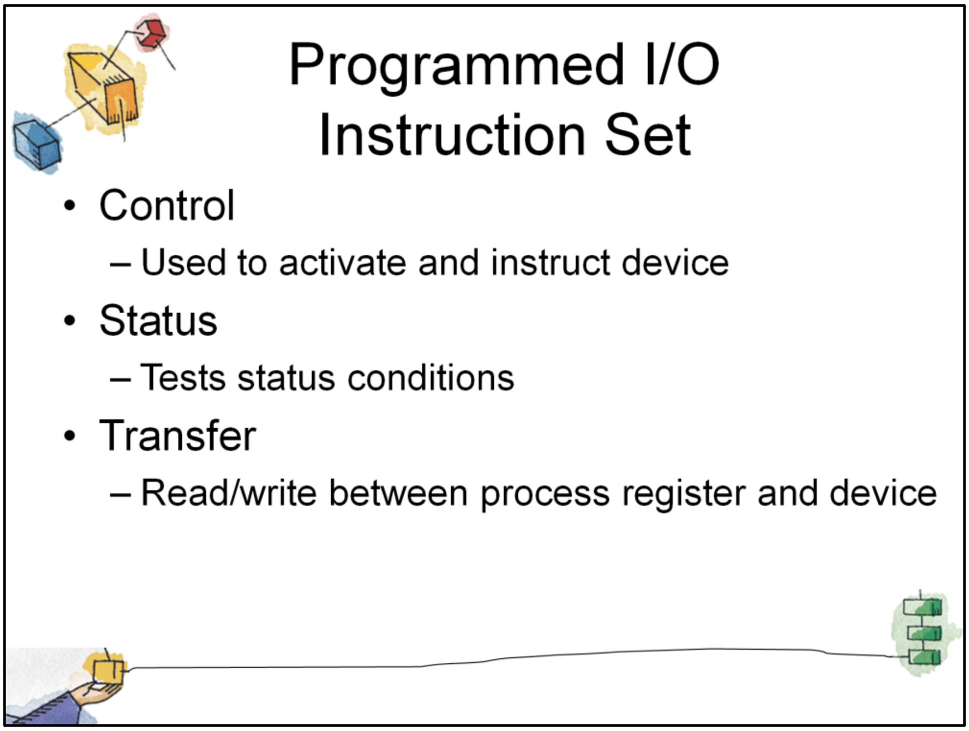
Programmed I/O

- The I/O module performs the requested action
  - then sets the appropriate bits in the I/O status register
  - but takes no further action to alert the processor.
- As there are no interrupts, the processor must determine when the instruction is complete

The I/O module performs the requested action and then sets the appropriate bits in the I/O status register but takes no further action to alert the processor.

In particular, it does not interrupt the processor.

- Thus, after the I/O instruction is invoked, the processor must take some active role in determining when the I/O instruction is completed.
- So, the processor periodically checks the status of the I/O module until it finds that the operation is complete.



Programmed I/O Instruction Set

- Control
  - Used to activate and instruct device
- Status
  - Tests status conditions
- Transfer
  - Read/write between process register and device

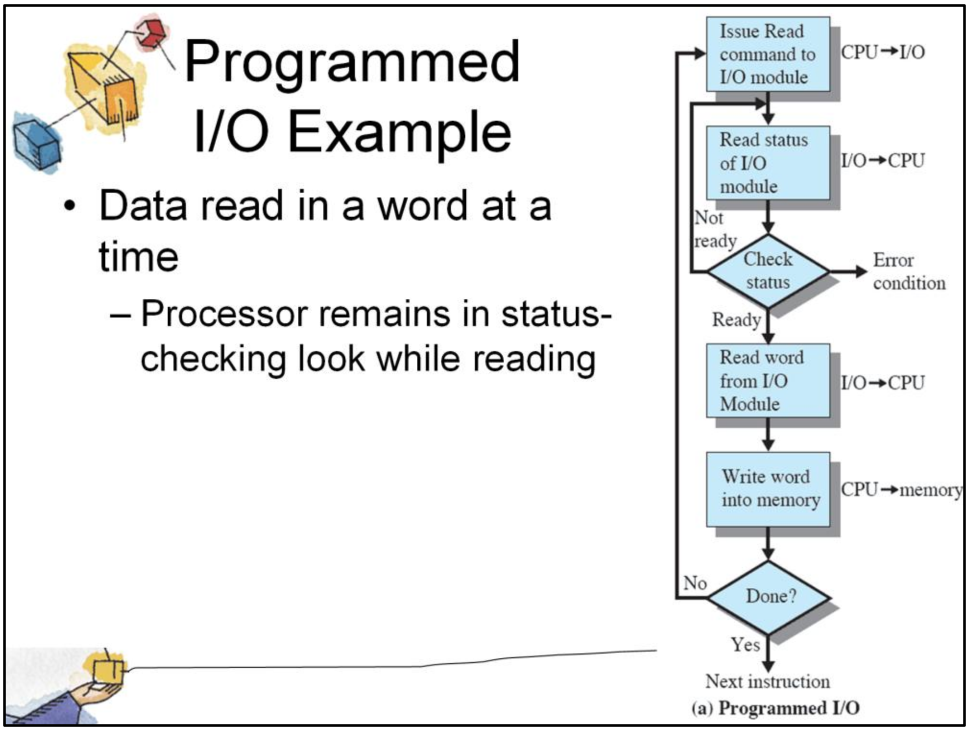
With this technique, the processor is responsible for extracting data from main memory for output and storing data in main memory for input.

The instruction set includes I/O instructions in the following categories:

- Control:Used to activate an external device and tell it what to do.
  - e.g. a magnetic-tape unit may be instructed to rewind or to move forward one record.

- Status: Used to test various status conditions associated with an I/O module and its peripherals.

- Transfer:Used to read and/or write data between processor registers and external devices.



Programmed I/O Example

- Data read in a word at a time
  - – Processor remains in status-checking look while reading

Issue Read command to I/O module

CPU→I/O

Read status of I/O module

I/O→CPU

Not ready

Check status

Error condition

Ready

Read word from I/O Module

I/O→CPU

Write word into memory

CPU→memory

No

Done?

Yes

Next instruction

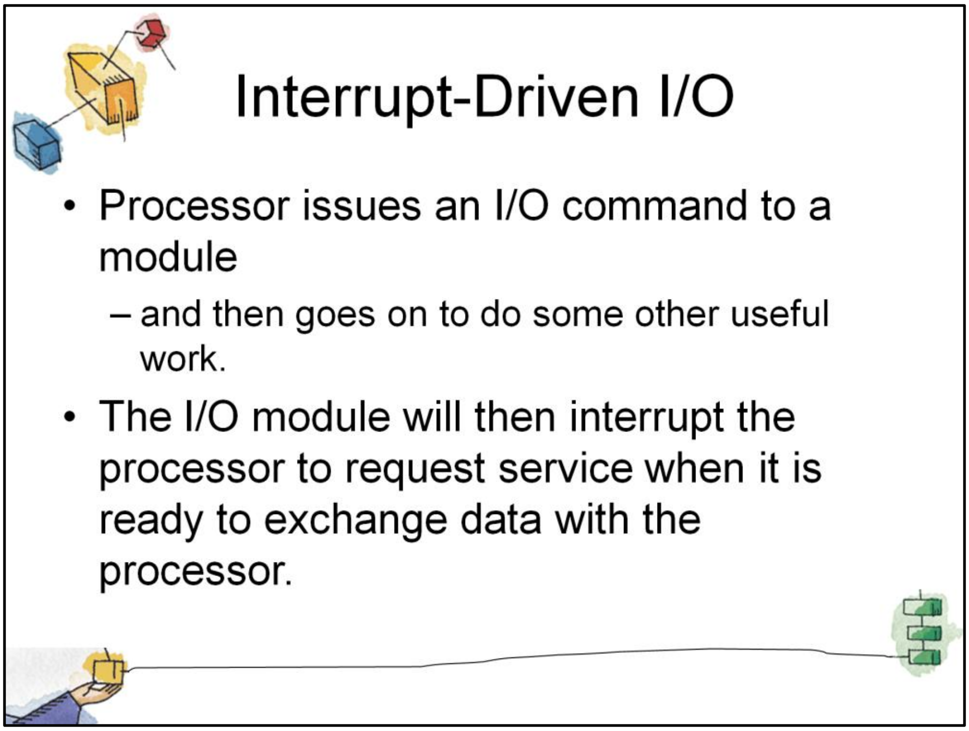
(a) Programmed I/O

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 loop until it determines that the word is available in the I/O module’s data register.

This flowchart highlights the main disadvantage of this technique:

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



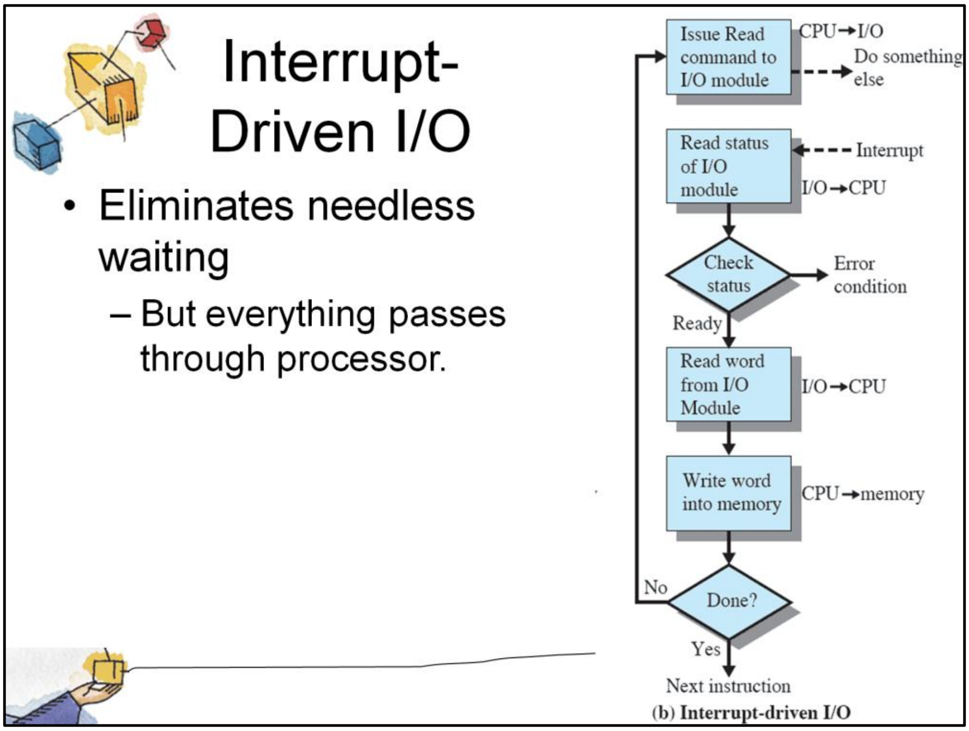
Interrupt-Driven I/O

- Processor issues an I/O command to a module
  - and then goes 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 to issues 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.



Interrupt-Driven I/O

- Eliminates needless waiting
  - But everything passes through processor.

Issue Read command to I/O module

CPU → I/O

Do something else

Read status of I/O module

Interrupt

I/O → CPU

Check status

Error condition

Ready

Read word from I/O Module

I/O → CPU

Write word into memory

CPU → memory

Done?

No

Yes

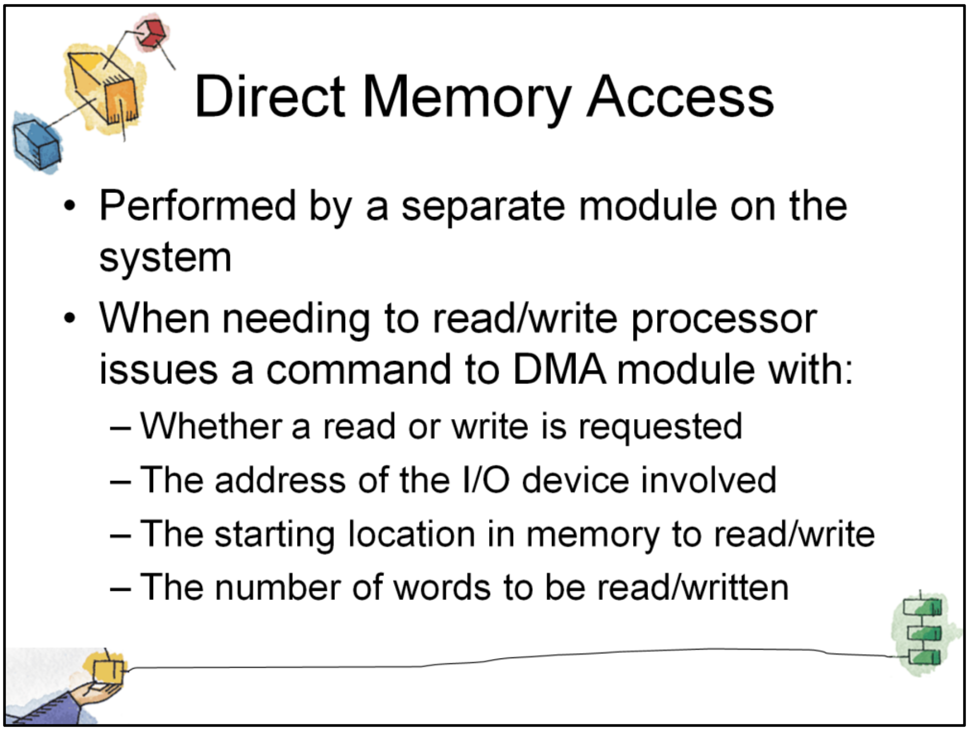
Next instruction

(b) Interrupt-driven I/O

This figure shows the use of interrupt-driven I/O for reading in a block of data.

Interrupt-driven I/O is more efficient than programmed I/O because it eliminates needless waiting.

- However, interrupt-driven 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.



Direct Memory Access

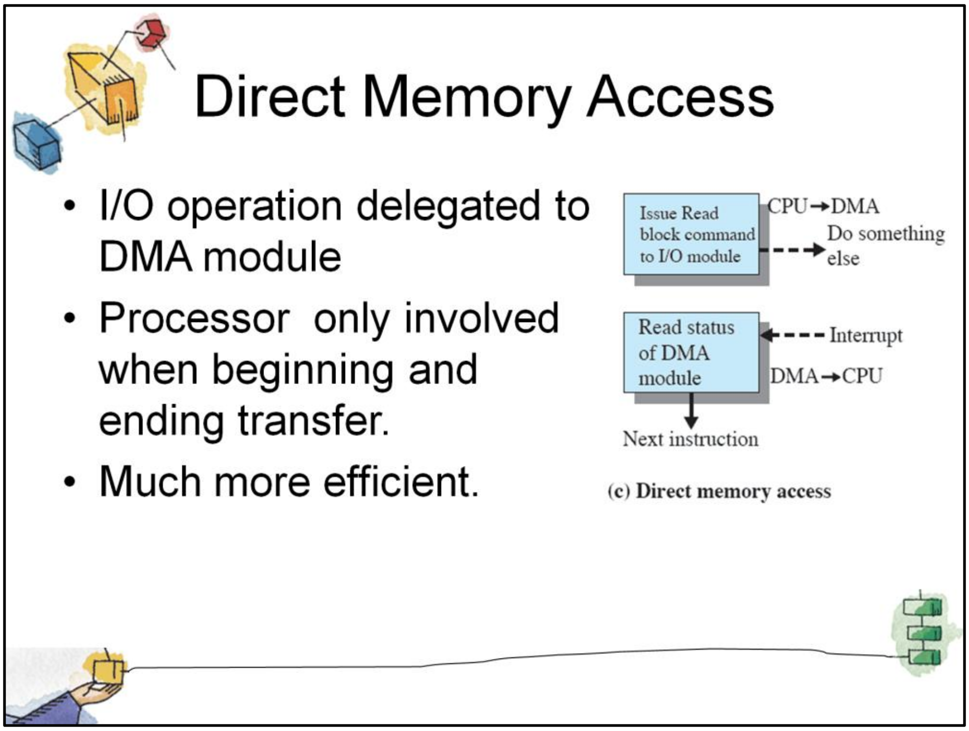
- Performed by a separate module on the system
- When needing to read/write processor issues a command to DMA module with:
  - Whether a read or write is requested
  - The address of the I/O device involved
  - The starting location in memory to read/write
  - The number of words to be read/written

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

The DMA function can be performed by a separate module on the system bus or it can be incorporated into an I/O module.

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
- The address of the I/O device involved
- The starting location in memory to read data from or write data to
- The number of words to be read or written



Direct Memory Access

- I/O operation delegated to DMA module
- Processor only involved when beginning and ending transfer.
- Much more efficient.

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

The processor can continue with other work.

It has delegated this I/O operation to the DMA module, and that module will take care of it.

- 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