



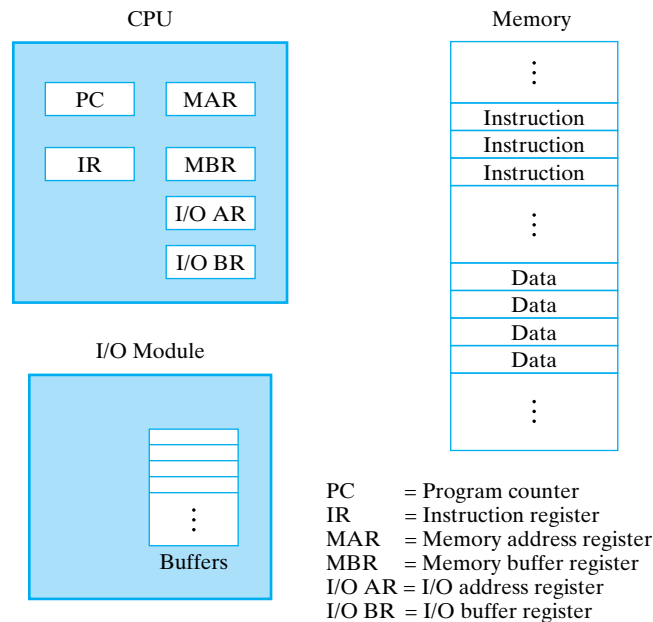
## PART OF THE PICTURE: Computer Architecture

BY WILLIAM STALLINGS

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. Thus, there are four main structural elements:

- **Processor:** 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)*.
- **Main Memory:** Stores data and programs. This memory is typically volatile; it is also referred to as *real memory or primary memory*.
- **I/O Modules:** Move data between the computer and its external environment. The external environment consists of a variety of external devices, including secondary memory devices, communications equipment, and printers.
- **System Interconnection:** Some structure and mechanisms that provide for communication among processors, main memory, and I/O modules.

Figure 1 depicts these top-level components. The processor controls operations. One of its 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. Similarly, an **I/O address register (IOAR)** specifies a particular I/O device. An **I/O buffer register (IOBR)** is used for the exchange of data between an I/O module and the processor.



### POP 6-1 Computer Components: Top-Level View

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 processor and memory, and vice versa. It contains internal buffers for temporarily holding this data until it can be sent on.

## PROCESSOR REGISTERS

Within the processor, there is a set of registers that provide a level of memory that is faster and smaller than main memory. The registers in the processor serve two functions:

- **User-visible registers:** These enable the assembly-language programmer to minimize main memory references by optimizing the use of registers. 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 and C++, allow the programmer to suggest to the compiler which variables should be held in registers.
- **Control and status registers:** These are used by the processor to control the operation of the processor and by privileged, operating-system routines to control the execution of programs.

There is not a clean separation of registers into these two categories. For example, on some machines the program counter is user visible, but on many it is not. For purposes of the following discussion, however, it is convenient to use these categories.

**User-Visible Registers.** A user-visible register is one that may be referenced by means of the machine language that the processor executes and that is generally available to all programs, including application programs as well as system programs. The following types of registers are typically available: data, address, and condition codes.

**Data registers** can be assigned to a variety of functions by the programmer. In some cases, they are general purpose in nature and can be used with any machine instruction that performs operations on data. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point 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 address. These registers may themselves be somewhat general purpose, or they may be devoted to a particular addressing mode. 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 variable-length blocks of words called *segments*. A memory reference consists of a reference to a particular segment and an offset within the segment; this mode of addressing is important in memory management. In this mode of addressing, a register is used to hold the address of the base (starting location) of the segment. There may be multiple registers; for example, one for the operating system (i.e., when operating-system code is executing on the processor) and one for the currently executing application.
- **Stack pointer:** If there is user-visible stack addressing, then typically the stack is in main memory and 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.

A final category of registers, which is at least partially visible to the user, holds **condition codes** (also referred to as *flags*). Condition codes are bits set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions allow these bits to be read by implicit reference, but they cannot be altered by the programmer.

**Control and Status Registers.** There are a variety of processor registers that are employed to control the operation of the processor. Most of these, on most machines, are not visible to the user. Some of them may be accessible by machine instructions executed in a control

or operating-system mode.

Of course, different machines will have different register organizations and use different terminology. We will list common register types, with a brief description.

In addition to the MAR, MBR, IOAR, and IOBR registers mentioned earlier, the following are essential to instruction execution:

- **Program counter (PC):** Contains the address of an instruction to be fetched.
- **Instruction register (IR):** Contains the instruction most recently fetched.

All processor designs also include a register or set of registers, often known as the **program status word (PSW)**, which contains status information. The PSW typically contains condition codes plus other status information. Common fields and flags include the following:

- **Sign:** Contains the sign bit of the last arithmetic operation.
- **Zero:** Set when the result of an arithmetic operation is zero.
- **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit. Used for multi-word arithmetic operations.
- **Equal:** Set if a logical compare result is equality.
- **Overflow:** Used to indicate arithmetic overflow.
- **Interrupt Enable/Disable:** Used to disable or enable interrupts. When interrupts are disabled, the processor ignores them. This is often desirable when the operating system is in the midst of dealing with another interrupt.
- **Supervisor:** Indicates whether the processor is executing in supervisor or user mode. Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

There are a number of other registers related to status and control that might be found in a particular processor design. In addition to the PSW, there may be a pointer to a block of memory containing additional status information. In machines using multiple types of interrupts, a set of registers may be provided, with one pointer to each interrupt-handling routine. If a stack is used to implement certain functions (e.g., procedure call), then a system stack pointer is needed. Memory management hardware requires dedicated registers. Finally, registers may be used in the control of I/O operations.

A number of factors go into the design of the control and status register organization. One key issue is operating system support. Certain types of control information are of specific utility to the operating system. If the processor designer has a functional understanding of the operating system to be used, then the register organization can to some extent be tailored to the operating system.

Another key design decision is the allocation of control information between registers and memory. It is common to dedicate the first (lowest) few hundred or thousand words of memory for control purposes. The designer must decide how much control information should be in more expensive, faster registers and how much in less expensive, slower main memory.

## INSTRUCTION EXECUTION

The basic function performed by a computer is program execution. The program to be executed consists of a set of instructions stored in memory. The processor does the actual work by executing instructions specified in the program.

The simplest point of view is to consider instruction processing as consisting 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*. In simple terms, the instruction cycle consists of a *fetch cycle*, in which the processor reads an instruction from memory, and the *execute cycle*, in which the processor executes the instruction. This instruction cycle is performed repeatedly.

## 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)** is used to keep track of which instruction is 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). So, for example, consider a computer in which each instruction occupies one 16-bit word of memory. Assume that the program counter is set to word location 300. 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 is in the form of a binary code that specifies what 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.

Of course, an instruction's execution may involve a combination of these actions.

Let us consider a simple example using a hypothetical machine that includes the characteristics listed in Figure 2. 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 locations, or words. The instruction format provides four bits for the opcode; thus there can be as many as  $2^4 = 16$  different opcodes, and up to  $2^{12} = 4,096$  (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

### POP 6-2 Characteristics of a Hypothetical Machine

Figure 3 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 cycles, are required:

1. The program counter (PC) contains 300, the address of the first instruction. This address is loaded into the instruction register (IR). Note that this process would involve the use of a memory address register (MAR) and a memory buffer register (MBR). For simplicity, these intermediate registers are ignored.
2. The first 4 bits in the IR indicate that the accumulator (AC) is to be loaded. The remaining 12 bits specify the address, which is 940.
3. After the load is complete, the PC is incremented and the next instruction is fetched.
4. The old contents of the AC and the contents of location 941 are added and the result is stored in the

AC.

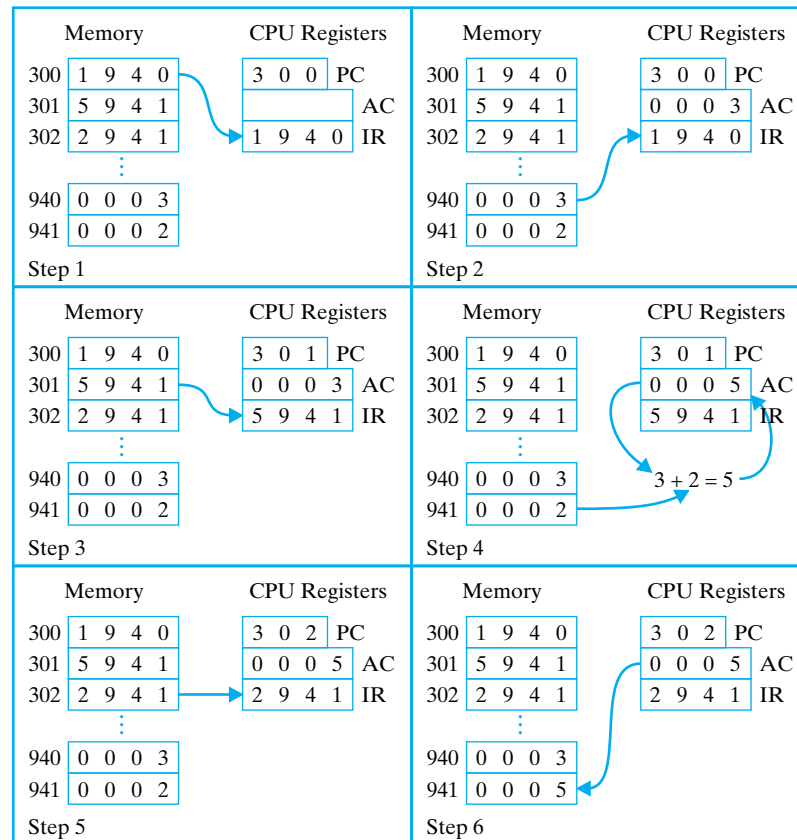
5. The PC is incremented and the next instruction is fetched.

The contents of the AC are stored in location 941.

In this example, three instruction cycles, each consisting of a fetch cycle and an execute cycle, are needed to add the contents of location 940 to the contents of 941. With a more complex set of instructions, fewer cycles would be needed. Most modern processors include instructions that contain more than one address. Thus 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.

## I/O FUNCTION

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 similar in form to that of Figure 3 could occur, with I/O instructions rather than memory-referencing instructions.



### POP 6-3 Example of Program Execution

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)**.

## THE MEMORY HIERARCHY

The design constraints on a computer's memory can be summed up by three questions: How much? How fast? How expensive?

The question of how much is somewhat open-ended. If the capacity is there, applications will likely be



developed to use it. The question of how fast is, in a sense, easier to answer. To achieve greatest performance, the memory must be able to keep up with the processor. That is, as the processor is executing instructions, we would not want it to have to pause waiting for instructions or operands. The final question must also be considered. For a practical system, the cost of memory must be reasonable in relationship to other components.

As might be expected, there is a trade-off among the three key characteristics of memory: namely cost, capacity, and access time. At any given time, a variety of technologies are used to implement memory systems. Across this spectrum of technologies, the following relationships hold:

- Smaller access time, greater cost per bit
- Greater capacity, smaller cost per bit
- Greater capacity, greater access time

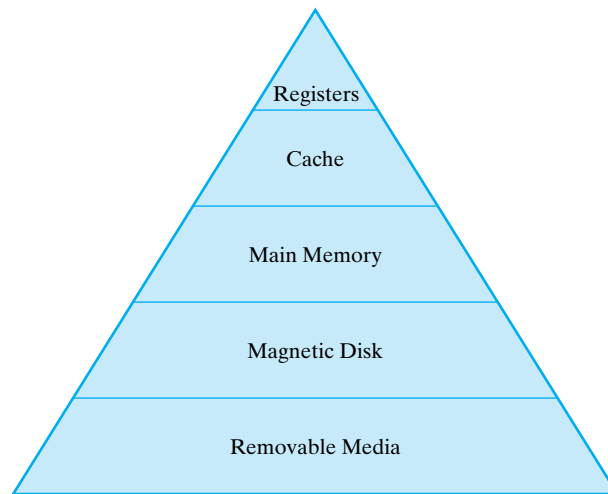
The dilemma facing the designer is clear. The designer would like to use memory technologies that provide for large-capacity memory, both because the capacity is needed and because the cost per bit is low. However, to meet performance requirements, the designer needs to use expensive, relatively lower-capacity memories with fast access times.

The way out of this dilemma is not to rely on a single memory component or technology, but to employ a **memory hierarchy**. A traditional hierarchy is illustrated in Figure 4. As one goes down the hierarchy, the following occur:

- (a) Decreasing cost/bit
- (b) Increasing capacity
- (c) Increasing access time
- (d) Decreasing frequency of access of the memory by the processor

Thus, smaller, more expensive, faster memories are supplemented by larger, cheaper, slower memories. By employing a variety of technologies, a spectrum of memory systems exist that satisfy conditions (a) through (c). Fortunately, condition (d) is also generally valid.

The basis for the validity of condition (d) is a principle known as *locality of reference*. During the course of execution of a program, memory references by the processor, for both instructions and data, tend to cluster. Programs typically contain a number of iterative loops and subroutines. Once a loop or subroutine is entered, there are repeated references to a small set of instructions. Similarly, operations on tables and arrays involve access to a clustered set of data words. Over a long period of time, the clusters in use change, but over a short period of time, the processor is primarily working with fixed clusters of memory references.



#### POP 6-4 The Memory Hierarchy

Accordingly, it is possible to organize data across the hierarchy such that the percentage of accesses to each successively lower level is substantially less than that of the level above. Consider a simple two-level memory. Let Level 2 memory contain all program instructions and data. The current clusters can be temporarily placed in Level 1. From time to time, one of the clusters in Level 1 will have to be swapped back to Level 2 to make room for a new cluster coming in to Level 1. On average, however, most references will be to instructions and data contained in Level 1.

This principle can be applied across more than two levels of memory. Consider the hierarchy shown in Figure 4. The fastest, smallest, and most expensive type of memory consists of the registers internal to the processor. Typically, a processor will contain a few dozen such registers, although some machines contain hundreds of registers. Skipping down two levels, main memory, also referred to as real memory, is the principal internal memory system of the computer. Each location in main memory has a unique address, and most machine instructions refer to one or more main memory addresses. Main memory is usually extended with a higher-speed, smaller cache. The cache is not usually visible to the programmer or, indeed, to the processor. It is a device for staging the movement of data between main memory and processor registers to improve performance.

The three forms of memory just described are, typically, volatile and employ semiconductor technology. The use of three levels exploits the fact that semiconductor memory comes in a variety of types, which differ in speed and cost. Data are stored more permanently on external mass storage devices, of which the most common are hard disk and removable media, such as removable disk, tape, and optical storage. External, nonvolatile memory is also referred to as secondary 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. Disk is also used to provide an extension to main memory known as *virtual storage or virtual memory* .

## INPUT/OUTPUT ORGANIZATION

In addition to one or more processors and a set of memory modules, the third key element of a computer system is a set of I/O modules. Each module interfaces to the system bus or other interconnection structure and controls one or more external devices. An I/O module is not simply mechanical connectors that wire a device into the system bus. Rather, the I/O module contains some “intelligence,” that is, it contains logic for controlling the flow of data between the external device and the bus.

An I/O module has two major functions:

- Interface to the processor and memory via the system bus or other interconnection structure.
- Interface to one or more external devices by tailored data links.

## I/O MODULE FUNCTION

The major functions or requirements for an I/O module fall into the following categories:

- Control and timing
- Communication with processor
- Communication with external device
- Data buffering
- Error detection

During any period of time, the processor may communicate with one or more external devices in unpredictable patterns, depending on the program’s need for I/O. The internal resources, such as main memory and the system bus, must be shared among a number of activities, including I/O. Thus, the I/O function includes a **control and timing** requirement, to coordinate the flow of traffic between internal resources and external devices. For example, the control of the transfer of data from an external device to the processor might involve the following sequence of steps:

1. The processor interrogates the I/O module to check the status of the attached device.
2. The I/O module returns the device status.
3. If the device is operational and ready to transmit, the processor requests the transfer of data, by means of a command to the I/O module.
4. The I/O module obtains a unit of data (e.g., 8 or 16 bits) from the external device.

The data are transferred from the I/O module to the processor.

If the system employs a bus, then each of the interactions between the processor and the I/O module

involves one or more bus events.

The preceding simplified scenario also illustrates that the I/O module must have the capability to engage in **communication with the processor** and with the external device. Communication with the processor involves:

- **Command decoding:** The I/O module accepts commands from the processor. These commands are generally sent as signals on the *control bus*. For example, an I/O module for a disk drive might accept the following commands: READ SECTOR, WRITE SECTOR, SEEK track number, and SCAN record ID. The latter two commands each include a parameter that is sent on the *data bus*.
- **Data:** Data are exchanged between the processor and the I/O module over the data bus.
- **Status reporting:** Because peripherals are so slow, it is important to know the status of the I/O module. For example, if an I/O module is asked to send data to the processor (read), it may not be ready to do so because it is still working on the previous command. This fact can be reported with a status signal. Common status signals are BUSY and READY. There may also be signals to report various error conditions.
- **Address recognition:** Just as each word of memory has an address, so does each I/O device. Thus, an I/O module must recognize one unique address for each peripheral it controls.

On the other side, the I/O module must be able to **communicate with external devices**. This communication also involves commands, status information, and data.

An essential task of an I/O module is **data buffering**. Whereas the transfer rate into and out of main memory or the processor is quite high, the rate is orders of magnitude lower for most peripheral devices. Data coming from main memory are sent to an I/O module in a rapid burst. The data are buffered in the I/O module and then sent to the external device at its data rate. In the opposite direction, data are buffered so as not to tie up the memory in a slow transfer operation. Thus, the I/O module must be able to operate at both device and memory speeds.

Finally, an I/O module is often responsible for **error detection** and for subsequently reporting errors to the processor. One class of errors includes mechanical and electrical malfunctions reported by the device (e.g., paper jam, bad disk track). Another class consists of unintentional changes to the bit pattern as it is transmitted from device to I/O module. Some form of error-detecting code is often used to detect transmission errors. A common example is the use of a *parity bit* on each character of data. For example, the ASCII character occupies 7 bits of a byte. The eighth bit is set so that the total number of 1s in the byte is even (even parity) or odd (odd parity). When a byte is received, the I/O module checks the parity to determine whether an error has occurred.

## TO PROBE FURTHER

The topics in this section are covered in detail in *Computer Organization and Architecture: Designing for Performance, Fourth Edition*, by William Stallings (Prentice Hall, 1996). Links to web sites with further information can be found at <http://www.shore.net/~ws/COA4e>.