GRETSA UNIVERSITY

ACADEMIC PROGRAMME: BSCS COMPUTER SCIENCE

COURSE CODE AND TITLE: BSCS 301: Computer organization and architecture

LECTURER'S NAME: DENIS WAPUKHA

LECTURER'S CONTACTS: Phone No.: 0725710895 **Email**: dwapukha@gretsauniversity.ac.ke

Topic 1: Introduction to Computer Systems

The objective of such historical review is to understand the factors affecting computing as we know it today and hopefully to forecast the future of computation. A great majority of the computers of our daily use are known as general purpose machines. These are machines that are built with no specific application in mind, but rather are capable of performing computation needed by a diversity of applications. These machines are to be distinguished from those built to serve (tailored to) specific applications. The latter are know **as special purpose machines.**

Based on the interface between different levels of the system, a number of computer architectures can be defined. The interface between the application programs and a high-level language is referred to as a **language architecture**. **The instruction set architecture** defines the interface between the basic machine instruction set and the runtime and I/O control.

A different definition of **computer architecture** is built on four basic viewpoints. These **are the structure, the organization, the implementation, and the performance**. In this definition, the structure defines the interconnection of various hardware components, the organization defines the dynamic interplay and management of the various components, the implementation defines the detailed design of hardware components, and the performance specifies the behavior of the computer system.

HISTORICAL BACKGROUND

It is probably fair to say that the first program-controlled (mechanical) computer ever build was **the Z1 (1938**). This was followed in **1939 by the Z2** as the first operational program-controlled computer with fixed-point arithmetic. However, the first recorded university-based attempt to build a computer originated on Iowa State University campus in the early 1940s. Researchers on that campus were able to build a **small-scale special-purpose electronic computer**. However, that computer was never completely operational. Just about the same time a complete design of a fully functional programmable special-purpose machine, **the Z3,** was reported in Germany in 1941.

It appears that the lack of funding prevented such design from being implemented. History recorded that while these two attempts were in progress, researchers from different parts of the world had opportunities to gain first-hand experience through their visits to the laboratories and institutes carrying out the work. It is assumed that such first-hand visits and interchange of ideas enabled the visitors to embark on similar projects in their own laboratories back home.

As far as **general-purpose machines** are concerned, the University of Pennsylvania is recorded to have hosted the building of the **Electronic Numerical Integrator and Calculator (ENIAC)** machine in 1944. It was the first operational general-purpose machine built using vacuum tubes. The machine was primarily built to help compute artillery firing tables during World War II. It was programmable through manual setting of switches and plugging of cables. The machine was slow by today's standard, with a limited amount of storage and primitive programmability. An improved version of the ENIAC was proposed on the same campus. The improved version of the ENIAC, called the Electronic Discrete Variable Automatic Computer (EDVAC), was an attempt to improve the way programs are entered and explore the concept of stored programs. It was not **until 1952 that the EDVAC** project was completed. Inspired by the ideas implemented in the ENIAC, researchers at the Institute for Advanced Study (IAS) at Princeton built (in 1946) the IAS machine, which was about 10 times faster than the ENIAC.

The first general-purpose commercial computer, the UNIVersal Automatic Computer (UNIVAC I), was on the market by the middle of 1951. It represented an improvement over the BINAC, which was built in 1949. IBM announced its first computer, the IBM701, in 1952. The early 1950s witnessed a slowdown in the computer industry. In 1964 IBM announced a line of products under the name IBM 360 series. The series included a number of models that varied in price and performance. This led Digital Equipment Corporation (DEC) to introduce the first minicomputer, the PDP-8.
It was considered a remarkably low-cost machine. Intel introduced the first microprocessor, the Intel 4004, in 1971. The world witnessed the birth of the first personal computer (PC) in 1977 when Apple computer series were first introduced. In 1977 the world also witnessed the introduction of the VAX-11/780 by DEC. Intel followed suit by introducing the first of the most popular microprocessor, the 80 86 series.

One of the **clear trends in computing** is the substitution of centralized servers by networks of computers. These networks connect inexpensive, powerful desktop machines to form unequaled computing power. Local area networks (LAN) of powerful personal computers and workstations began to replace mainframes and minis by 1990. These individual desktop computers were soon to be connected into larger complexes of computing by wide area networks (WAN).

ARCHITECTURAL DEVELOPMENT AND STYLES

Computer architects have always been striving to increase the performance of their architectures. This has taken a number of forms. Among these is the philosophy that by doing more in a single instruction, one can use a smaller number of instructions to perform the same job.

The huge number of addressing modes considered (more than 20 in the VAX machine) further adds to the complexity of instructions. Machines following this philosophy have been referred to as complex instructions set computers (CISCs). Examples of CISC machines include the Intel PentiumTM, the Motorola MC68000TM, and the IBM & Macintosh PowerPCTM. It should be noted that as more capabilities were added to their processors, manufacturers realized that it was increasingly difficult to support higher clock rates that would have been possible otherwise.

TECHNOLOGICAL DEVELOPMENT

Computer technology has shown an unprecedented rate of improvement. This includes the development of processors and memories. Indeed, it is the advances in technology that have fueled the computer industry.

de possible by the advances in the fabrication technology of transistors. The scale of integration has grown from small-scale (SSI) to medium-scale (MSI) to large-scale (LSI) to very large-scale integration (VLSI), and currently to waferscale integration (WSI). Table 1.2 shows the typical numbers of devices per chip in each of these technologies.

Topic 2: PERFORMANCE MEASURES

In this section, we consider the important issue of assessing the performance of a computer. In particular, we focus our discussion on a number of performance measures that are used to assess computers.

Performance analysis should help answering questions such as how fast can a program be executed using a given computer? In order to answer such a question, we need to determine the time taken by a computer to execute a given job. We define the clock cycle time as the time between two consecutive rising (trailing) edges of a periodic clock signal. The time required to execute a job by a computer is often expressed in terms of clock cycles.

We denote the number of CPU clock cycles for executing a job to be the cycle count (CC), the cycle time by CT, and the clock frequency by f ¼ 1/CT. The time taken by the CPU to execute a job can be expressed as CPU time ¼ **CC CT ¼ CC=f**

CPU time = CC x CT = CC/f

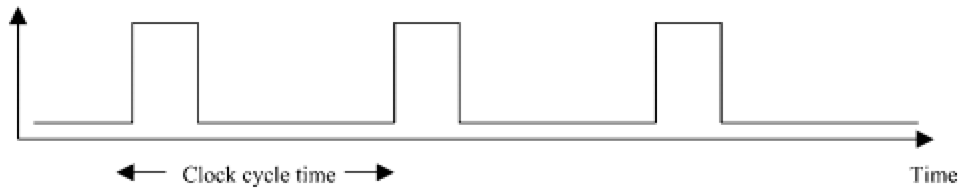


Figure 1.1 Clock signal

It may be easier to count the number of instructions executed in a given program as compared to counting the number of CPU clock cycles needed for executing that program. Therefore, the average number of clock cycles per instruction (CPI) has been used as an alternate performance measure. The following equation shows how to compute the CPI.

$$CPI = \frac{\text{CPU clock cycles for the program}}{\text{Instruction count}}$$

$$\text{CPU time} = \text{Instruction count} \times CPI \times \text{Clock cycle time}$$

$$= \frac{\text{Instruction count} \times CPI}{\text{Clock rate}}$$

Question

Compare uniprocessor systems with multiprocessor systems in the following aspects:
(a) Ease of programming

(b) The need for synchronization

(c) Performance evaluation

(d) Run time system

2. Consider having a program that runs in 50 s on computer A, which has a 500 MHz clock. We would like to run the same program on another machine, B, in 20 s. If machine B requires 2.5 times as many clock cycles as machine A for the same program, what clock rate must machine B have in MHz?
3. Suppose that we have two implementations of the same instruction set architecture. Machine A has a clock cycle time of 50 ns and a CPI of 4.0 for some program, and machine B has a clock cycle of 65 ns and a CPI of 2.5 for the same program. Which machine is faster and by how much?

TOPIC 3: Instruction Set Architecture and Design

Introduction

In this chapter, we consider the basic principles involved in instruction set architecture and design. Our discussion starts with a consideration of memory locations and addresses. We present an abstract model of the main memory in which it is considered as a sequence of cells each capable of storing n bits. We then address the issue of storing and retrieving information into and from the memory.

The information stored and/or retrieved from the memory needs to be addressed.

A discussion on a number of different ways to address memory locations (addressing modes) is the next topic to be discussed in the chapter. A program consists of a number of instructions that have to be accessed in a certain order. That motivates us to explain the issue of instruction execution and sequencing in some detail. We then show the application of the presented addressing modes and instruction characteristics in writing sample segment codes for performing a number of simple programming tasks.

MEMORY LOCATIONS AND OPERATIONS

The (main) memory can be modeled as an array of millions of adjacent cells, each capable of storing a binary digit (bit), having value of 1 or 0. These cells are organized in the form of groups of fixed number, say n, of cells that can be dealt with as an atomic entity.

An entity consisting of 8 bits is called a byte. In many systems, the entity consisting of n bits that can be stored and retrieved in and out of the memory using one basic memory operation is called a word (the smallest addressable entity). Typical size of a word ranges from 16 to 64 bits. It is, however, customary to express the size of the memory in terms of bytes. For example, the size of a typical memory of a personal computer is 256 Mbytes, that is, $256 \times 2^{20} = 228$ bytes.

In order to be able to move a word in and out of the memory, a distinct address has to be assigned to each word. This address will be used to determine the location in the memory in which a given word is to be stored. This is called a memory write operation. Similarly, the address will be used to determine the memory location from which a word is to be retrieved from the memory. This is called a memory read operation.

Three basic steps are needed in order for the CPU to perform a write operation into a specified memory location:

1. The word to be stored into the memory location is first loaded by the CPU into a specified register, called the memory data register (MDR).

2. The address of the location into which the word is to be stored is loaded by the CPU into a specified register, called the memory address register (MAR).

3. A signal, called write, is issued by the CPU indicating that the word stored in the MDR is to be stored in the memory location whose address in loaded in the MAR.

It is worth mentioning that the MDR and the MAR are registers used exclusively

by the CPU and are not accessible to the programmer.

Similar to the write operation, three basic steps are needed in order to perform a

memory read operation:

1. The address of the location from which the word is to be read is loaded into the MAR.

2. A signal, called read, is issued by the CPU indicating that the word whose address is in the MAR is to be read into the MDR.

3. After some time, corresponding to the memory delay in reading the specified word, the required word will be loaded by the memory into the MDR ready for use by the CPU.

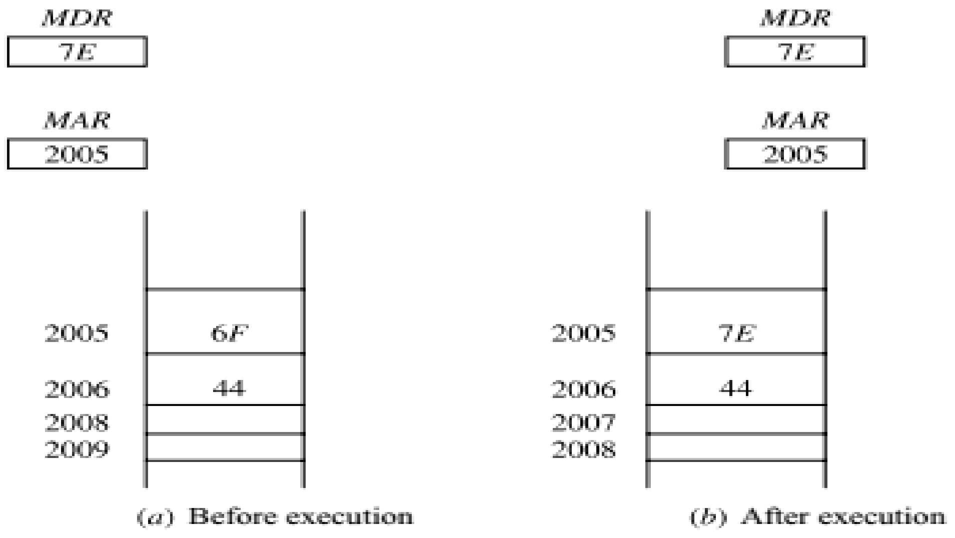


Figure 2.2 Illustration of the memory write operation

ADDRESSING MODES

Information involved in any operation performed by the CPU needs to be addressed. In computer terminology, such information is called the operand. Therefore, any instruction issued by the processor must carry at least two types of information.

These are the operation to be performed, encoded in what is called the op-code field, and the address information of the operand on which the operation is to be performed, encoded in what is called the address field.

Instructions can be classified based on the number of operands as: three-address, two-address, one-and-half-address, one-address, and zero-address. We explain these classes together with simple examples in the following paragraphs. It should be noted that in presenting these examples, we would use the convention operation, source, destination to express any instruction. In that convention, operation represents the operation to be performed, for example, add, subtract, write, or read.

The source field represents the source operand(s). The source operand can be a constant, a value stored in a register, or a value stored in the memory. The destination field represents the place where the result of the operation is to be stored, for example, a register or a memory location.

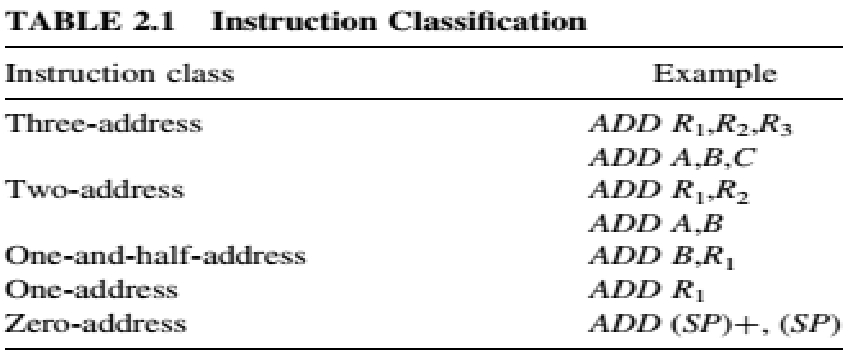


TABLE 2.1 Instruction Classification

Instruction class	Example
Three-address	$ADD\ R_1,R_2,R_3$
	$ADD\ A,B,C$
Two-address	$ADD\ R_1,R_2$
	$ADD\ A,B$
One-and-half-address	$ADD\ B,R_1$
One-address	$ADD\ R_1$
Zero-address	$ADD\ (SP)+,\ (SP)$

A **three-address instruction** takes the form operation add-1, add-2, add-3. In this form, each of add-1, add-2, and add-3 refers to a register or to a memory location. Consider, for example, the instruction ADD R1,R2,R3.

A **two-address instruction** takes the form operation add-1, add-2. In this form, each of add-1 and add-2 refers to a register or to a memory location. Consider, for example, the instruction ADD R1, R2. This instruction adds the contents of register R1 to the contents of register R2 and stores the results in register R2.

A **one-address instruction** takes the form ADD R1. In this case the instruction implicitly refers to a register, called the Accumulator Racc, such that the contents of the accumulator is added to the contents of the register R1 and the results are stored back into the accumulator Racc.

It is interesting to indicate that there exist **zero-address instructions**. These are the instructions that use stack operation. A stack is a data organization mechanism in which the last data item stored is the first data item retrieved. Two specific operations can be performed on a stack. These are the push and the pop operations.

Types of addressing modes

The different ways in which operands can be addressed are called the addressing modes. Addressing modes differ in the way the address information of operands is specified. The simplest addressing mode is to include the operand itself in the instruction, that is, no address information is needed. This is called **immediate addressing**. A more involved addressing mode is to compute the address of the operand by adding a constant value to the content of a register. This is called **indexed addressing**. Between these two addressing modes there exist a number of other addressing modes including **absolute addressing, direct addressing, and indirect addressing**. A number of different addressing modes are explained below.

Immediate Mode

According to this addressing mode, the value of the operand is (immediately) available in the instruction itself. Consider, for example, the case of loading the decimal value 1000 into a register Ri. This operation can be performed using an instruction such as the following: LOAD #1000, Ri. In this instruction, the operation to be performed is to load a value into a register. The source operand is (immediately) given as 1000, and the destination is the register Ri. It should be noted that in order to indicate that the value 1000 mentioned in the instruction is the operand itself and not its address (immediate mode), it is customary to prefix the operand by the special character (#).

Direct (Absolute) Mode

According to this addressing mode, the address of the memory location that holds the operand is included in the instruction. Consider, for example, the case of loading the value of the operand stored in memory location 1000 into register Ri. This operation can be performed using an instruction such as LOAD 1000, Ri. In this instruction, the source operand is the value stored in the memory location whose address is 1000, and the destination is the register Ri.

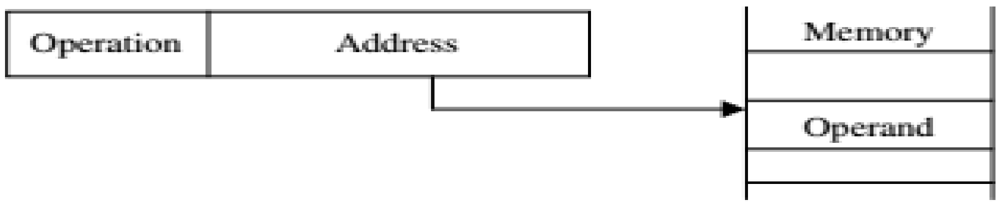


Figure 2.6 Illustration of the direct addressing mode

Direct (absolute) addressing mode provides more flexibility compared to the immediate mode. However, it requires the explicit inclusion of the operand address in the instruction. A more flexible addressing mechanism is provided through the use of the indirect addressing mode.

Indirect Mode

In the indirect mode, what is included in the instruction is not the address of the operand, but rather a name of a register or a memory location that holds the (effective) address of the operand. In order to indicate the use of indirection in the instruction, it is customary to include the name of the register or the memory location in parentheses. Consider, for example, the instruction LOAD (1000), Ri.

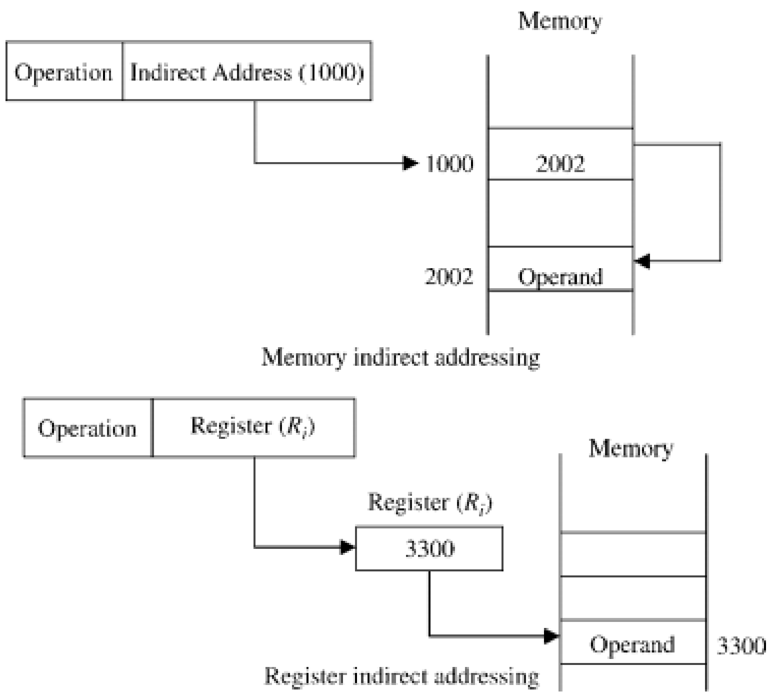


Figure 2.7 Illustration of the indirect addressing mode

This instruction has the memory location 1000 enclosed in parentheses, thus indicating indirection. The meaning of this instruction is to load register Ri with the contents of the memory location whose address is stored at memory address 1000. Because indirection can be made through either a register or a memory location, therefore, we can identify two types of indirect addressing. These are register indirect addressing, if a register is used to hold the address of the operand, and memory indirect addressing, if a memory location is used to hold the address of the operand.

Indexed Mode

In this addressing mode, the address of the operand is obtained by adding a constant to the content of a register, called the index register. Consider, for example, the instruction LOAD X(Rind), Ri. This instruction loads register Ri with the contents of the memory location whose address is the sum of the contents of register Rind and the value X. Index addressing is indicated in the instruction by including the name of the index register in parentheses and using the symbol X to indicate the constant to be added.

Other Modes

The addressing modes presented above represent the most commonly used modes in most processors. They provide the programmer with sufficient means to handle most general programming tasks. However, a number of other addressing modes have been used in a number of processors to facilitate execution of specific programming tasks.

Relative Mode

Recall that in indexed addressing, an index register, Rind, is used. Relative addressing is the same as indexed addressing except that the program counter (PC) replaces the index register. For example, the instruction LOAD X(PC), Ri loads register Ri with the contents of the memory location whose address is the sum of the contents of the program counter (PC) and the value X.

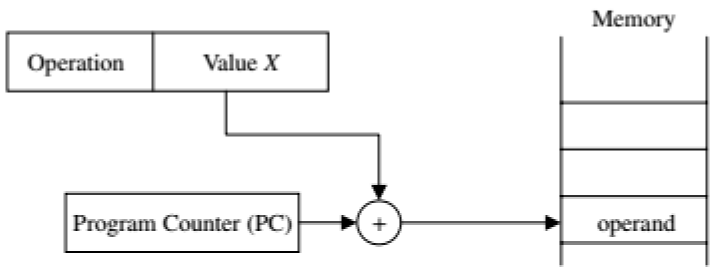


Figure 2.9 Illustration of relative addressing mode

Autoincrement Mode

This addressing mode is similar to the register indirect addressing mode in the sense that the effective address of the operand is the content of a register, call it the autoincrement register, that is included in the instruction.

However, with autoincrement, the content of the autoincrement register is automatically incremented after accessing the operand. As before, indirection is indicated by including the autoincrement register in parentheses.

INSTRUCTION TYPES

The type of instructions forming the instruction set of a machine is an indication of the power of the underlying architecture of the machine.

Data Movement Instructions

Data movement instructions are used to move data among the different units of the machine. Most notably among these are instructions that are used to move data among the different registers in the CPU. A simple register to register movement of data can be made through the instruction

$MOVE\ R_i,R_j$

TABLE 2.3 Some Common Data Movement Operations

Data movement operation	Meaning
MOVE	Move data (a word or a block) from a given source (a register or a memory) to a given destination
LOAD	Load data from memory to a register
STORE	Store data into memory from a register
PUSH	Store data from a register to stack
POP	Retrieve data from stack into a register

This instruction moves the content of register Ri to register Rj. The effect of the instruction is to override the contents of the (destination) register Rj without changing the contents of the (source) register Ri. Data movement instructions include those used to move data to (from) registers from (to) memory.

Arithmetic and Logical Instructions

Arithmetic and logical instructions are those used to perform arithmetic and logical manipulation of registers and memory contents. Examples of arithmetic instructions include the ADD and SUBTRACT instructions. These are:

$$ADD\ \ R_1, R_2, R_0$$
$$SUBTRACT\ \ R_1, R_2, R_0$$

The first instruction adds the contents of source registers R1 and R2 and stores the result in destination register R0. The second instruction subtracts the contents of the source registers R1 and R2 and stores the result in the destination register R0.

TABLE 2.4 Some Common Arithmetic Operations

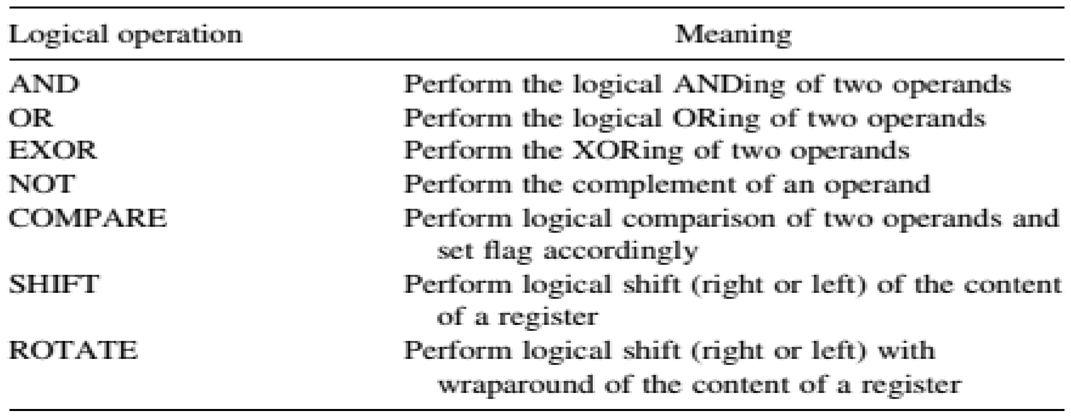
Arithmetic operations	Meaning
ADD	Perform the arithmetic sum of two operands
SUBTRACT	Perform the arithmetic difference of two operands
MULTIPLY	Perform the product of two operands
DIVIDE	Perform the division of two operands
INCREMENT	Add one to the contents of a register
DECREMENT	Subtract one from the contents of a register

The contents of the source registers are unchanged by the ADD and the SUBTRACT instructions. In addition to the ADD and SUBTRACT instructions, some machines have MULTIPLY and DIVIDE instructions. These two instructions are expensive to implement and could be substituted by the use of repeated addition or repeated subtraction. Therefore, most modern architectures do not have MULTIPLY or DIVIDE.

Sequencing Instructions

Control (sequencing) instructions are used to change the sequence in which instructions are executed. They take the form of CONDITIONAL BRANCHING (CONDITIONAL JUMP), UNCONDITIONAL BRANCHING (JUMP), or CALL instructions. A common characteristic among these instructions is that their execution changes the program counter (PC) value.

TABLE 2.5 Some Common Logical Operations

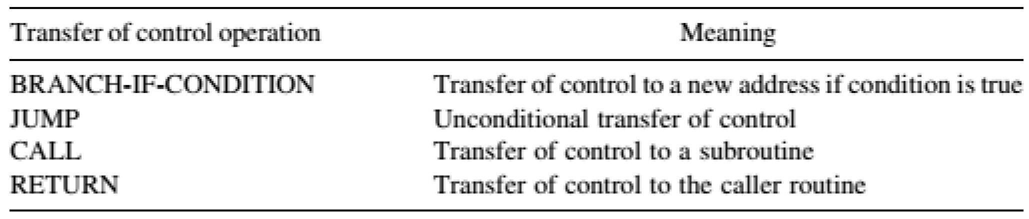


Logical operation	Meaning
AND	Perform the logical ANDing of two operands
OR	Perform the logical ORing of two operands
EXOR	Perform the XORing of two operands
NOT	Perform the complement of an operand
COMPARE	Perform logical comparison of two operands and set flag accordingly
SHIFT	Perform logical shift (right or left) of the content of a register
ROTATE	Perform logical shift (right or left) with wraparound of the content of a register

Input/Output Instructions

Input and output instructions (I/O instructions) are used to transfer data between thecomputer and peripheral devices. The two basic I/O instructions used are the INPUT and OUTPUT instructions. The INPUT instruction is used to transfer data from an input device to the processor. Examples of input devices include a keyboard or a mouse. Input devices are interfaced with a computer through dedicated input ports. Computers can use dedicated addresses to address these ports. Suppose that the input port through which a keyboard is connected to a computer carries the unique address 1000.

TABLE 2.7 Some Transfer of Control Operations



Transfer of control operation	Meaning
BRANCH-IF-CONDITION	Transfer of control to a new address if condition is true
JUMP	Unconditional transfer of control
CALL	Transfer of control to a subroutine
RETURN	Transfer of control to the caller routine

Summary

SUMMARY

In this topic we considered the main issues relating to instruction set design and characteristics. We presented a model of the main memory in which the memory is abstracted as a sequence of cells, each capable of storing n bits. A number of addressing modes were presented. These include immediate, direct, indirect, indexed, autoincrement, and autodecrement. Examples showing how to use these addressing modes were then presented. We also presented a discussion on instruction types, which include data movement, arithmetic/logical, instruction sequencing, and Input/Output.

Exercise

1. Consider a computer that has a number of registers such that the three registers R0 ¼ 1500, R1 ¼ 4500, and R2 ¼ 1000. Show the effective address of memory and the registers' contents in each of the following instructions (assume that all numbers are decimal).

(a) ADD (R0)þ, R2

(b) SUBTRACT 2 (R1), R2

(c) MOVE 500(R0), R2

(d) LOAD #5000, R2

(e) STORE R0, 100(R2)

2. Assume that the top of the stack in a program is pointed to by the register SP. You are required to write program segments to perform each of the following tasks (assume that only the following addressing modes are available: indexed, autoincrement, and autodecrement).

(a) Pop the top three elements of the stack, add them, and push the result back onto the stack.

(b) Pop the top two elements of the stack, subtract them, and push the results back onto the stack.

(c) Push five elements (one at a time) onto the stack.

(d) Remove the top five elements from the top of the stack.

(e) Copy the third element from the top of the stack into register R0

Topic 4: Computer Arithmetic

This chapter is dedicated to a discussion on computer arithmetic. Our goal is to introduce the reader to the fundamental issues related to the arithmetic operations and circuits used to support computation in computers. Our coverage starts with an introduction to number systems. In particular, we introduce issues such as number representations and base conversion. This is followed by a discussion on integer arithmetic. In this regard, we introduce a number of algorithms together with hardware schemes that are used in performing integer addition, subtraction, multiplication, and division. We end this chapter with a discussion on floating point arithmetic.

A number system uses a specific radix (base). Radices that are power of 2 are widely used in digital systems. These radices include binary (base 2), quaternary (base 4), octagonal (base 8), and hexagonal (base 16). The base 2 binary system is dominant in computer systems.

Negative Integer Representation

There exist a number of methods for representation of negative integers. These include the sign-magnitude, radix complement, and diminished radix complement. These are briefly explained below.

Sign-Magnitude
According to this representation, the most significant bit (of the n bits used to represent the number) is used to represent the sign of the number such that a "1" in the most significant bit position indicates a negative number while a "0" in the most significant bit position indicates a positive number. The remaining (n 1)bits are used to represent the magnitude of the number.

Radix Complement

According to this system, a positive number is represented the same way as in the sign-magnitude. However, a negative number is represented using the b's complement (for base b numbers). Consider, for example, the representation of the number (219) using 2's complement. In this case, the number 19 is first represented as (010011). Then each digit is complemented, hence the name radix complement to produce (101100). Finally a "1" is added at the least significant bit position to result in (101101).

Now, consider the **2's complement** representation of the number (þ18). Since the number is positive, then it is represented as (010010), the same as in the sign-magnitude case. Now, consider the addition of these two numbers. In this case, we add the corresponding bits without giving special treatment to the sign bit. The results of adding the two numbers produces (111111). This is the 2's complement representation of a (21), as expected. The main advantage of the 2's complement representation is that no special treatment is needed for the sign of the numbers.

SUMMARY

In this topic, we have discussed a number of issues related to computer arithmetic. Our discussion started with an introduction to number representation and radix conversion techniques. We then discussed integer arithmetic and, in particular, we discussed the four main operations, that is, addition, subtraction, multiplication, and division. In each case, we have shown basic architectures and organization. The last topic discussed in the chapter has been floating-point representation and arithmetic. We have also shown the basic architectures needed to perform basic floating-point operations such as addition, subtraction, multiplication, and division.

EXERCISES

1. Represent the decimal values 26, 2123 as signed, 10-bit numbers using each of the following binary formats:

(a) Sign-and-magnitude;

(b) 2’s complement.

2. Compute the decimal value of the binary number 1011 1101 0101 0110 if the given number represents unsigned integer. Repeat if the number represents 2’s complement. Repeat if the number represents sign-magnitude integer.

Topic 5: Processing Unit Design

Introduction

In previous chapters, we studied the history of computer systems and the fundamental issues related to memory locations, addressing modes, assembly language, and computer arithmetic. In this chapter, we focus our attention on the main component of any computer system, the central processing unit (CPU). The primary function of the CPU is to execute a set of instructions stored in the computer's memory. A simple CPU consists of a set of registers, an arithmetic logic unit (ALU), and a control unit (CU).

CPU BASICS

A typical CPU has three major components: (1) register set, (2) arithmetic logic unit (ALU), and (3) control unit (CU). The register set differs from one computer architecture to another. It is usually a combination of general-purpose and special purpose registers. General-purpose registers are used for any purpose, hence the name general purpose. Special-purpose registers have specific functions within the CPU. For example, the program counter (PC) is a special-purpose register that is used to hold the address of the instruction to be executed next. Another example of special-purpose registers is the instruction register (IR), which is used to hold the instruction that is currently executed. The ALU provides the circuitry needed to perform the arithmetic, logical and shift operations demanded of the instruction set.

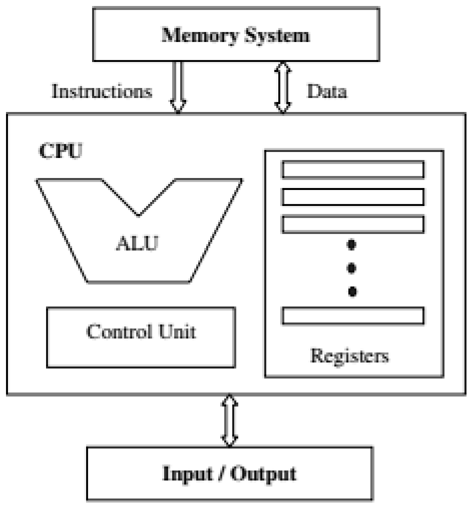


Figure 5.1 : Central processing unit main components and interactions with the memory and I/O

The CPU fetches instructions from memory, reads and writes data from and to memory, and transfers data from and to input/output

Simple execution cycle can be summarized as follows:

1. The next instruction to be executed, whose address is obtained from the PC, is fetched from the memory and stored in the IR.

2. The instruction is decoded.

3. Operands are fetched from the memory and stored in CPU registers, if needed.

4. The instruction is executed.

5. Results are transferred from CPU registers to the memory, if needed.

REGISTER SET

Registers are essentially extremely fast memory locations within the CPU that areused to create and store the results of CPU operations and other calculations. Different computers have different register sets. They differ in the number of registers, register types, and the length of each register. They also differ in the usage of each register. General-purpose registers can be used for multiple purposes and assigned to a variety of functions by the programmer. Special-purpose registers are restricted to only specific functions. In some cases, some registers are used only to hold data and cannot be used in the calculations of operand addresses.

Memory Access Registers

Two registers are essential in memory write and read operations: the memory data register (MDR) and memory address register (MAR). The MDR and MAR are used exclusively by the CPU and are not directly accessible to programmers.

In order to perform a write operation into a specified memory location, the MDR and MAR are used as follows:

1. The word to be stored into the memory location is first loaded by the CPU into MDR.

2. The address of the location into which the word is to be stored is loaded by the CPU into a MAR.

3. A write signal is issued by the CPU.

Similarly, to perform a memory read operation, the MDR and MAR are used as follows:

1. The address of the location from which the word is to be read is loaded into the MAR.

2. A read signal is issued by the CPU.

3. The required word will be loaded by the memory into the MDR ready for use by the CPU.

Instruction Fetching Registers

Two main registers are involved in fetching an instruction for execution: the program counter (PC) and the instruction register (IR). The PC is the register that contains the address of the next instruction to be fetched. The fetched instruction is loaded in the IR for execution. After a successful instruction fetch, the PC is updated to point to the next instruction to be executed.

Condition Registers

Condition registers, or flags, are used to maintain status information. Some architectures contain a special program status word (PSW) register. The PSW contains bits that are set by the CPU to indicate the current status of an executing program. These indicators are typically for arithmetic operations, interrupts, memory protection information, or processor status.

Special-Purpose Address Registers

Index Register :- As covered in Chapter 2, in index addressing, the address of the operand is obtained by adding a constant to the content of a register, called the index register. The index register holds an address displacement.

Segment Pointers :- As we will discuss in Chapter 6, in order to support segmentation, the address issued by the processor should consist of a segment number (base)

and a displacement (or an offset) within the segment.

Stack Pointer :- As shown in Chapter 2, a stack is a data organization mechanism in which the last data item stored is the first data item retrieved. Two specific operations can be performed on a stack. These are the Push and the Pop operations. A specific register, called the stack pointer (SP), is used to indicate the stack location that can be addressed.

80386 Registers :-

As discussed in Chapter 3, the Intel basic programming model of the 386, 486, and the Pentium consists of three register groups. These are the general-purpose registers, the segment registers, and the instruction pointer (program counter) and the flag register.

DATAPATH

The CPU can be divided into a data section and a control section. The data section, which is also called the datapath, contains the registers and the ALU. The datapath is capable of performing certain operations on data items. The control section is basically the control unit, which issues control signals to the datapath. Internal to the CPU, data move from one register to another and between ALU and registers. Internal data movements are performed via local buses, which may carry data, instructions, and addresses. Externally, data move from registers to memory and I/O devices, often by means of a system bus. Internal data movement among registers and between the ALU and registers may be carried out using different organizations including one-bus, two-bus, or three-bus organizations.

Type of data paths

One-Bus

Organization Using one bus, the CPU registers and the ALU use a single bus to move outgoing and incoming data. Since a bus can handle only a single data movement within one clock cycle, two-operand operations will need two cycles to fetch the operands for the ALU. Additional registers may also be needed to buffer data for the ALU. This bus organization is the simplest and least expensive, but it limits the amount of data transfer that can be done in the same clock cycle, which will slow down the overall performance.

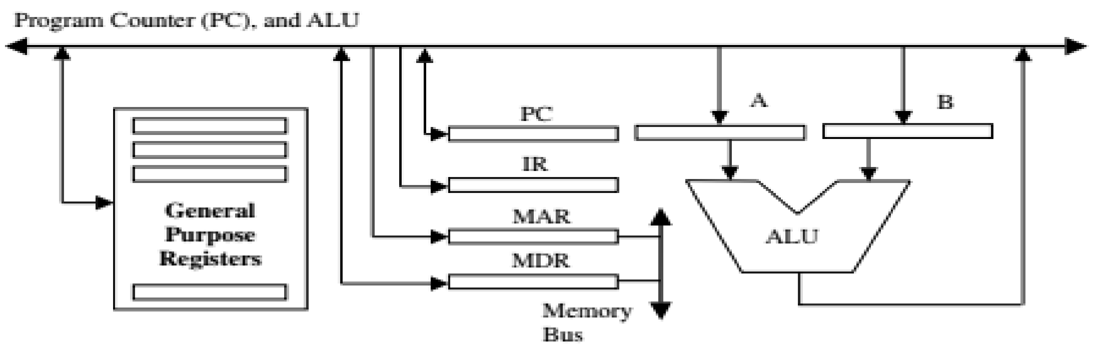
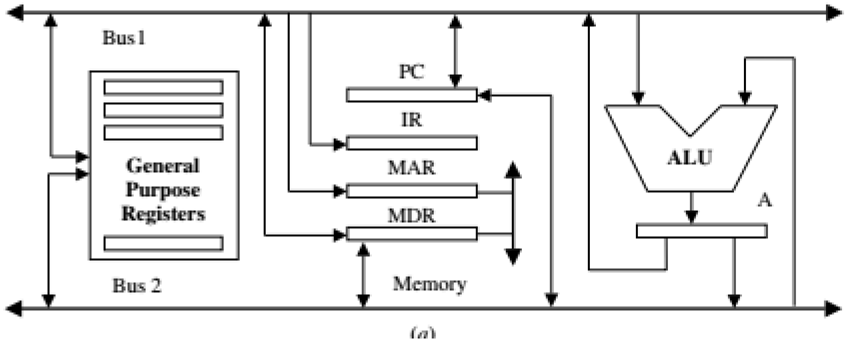


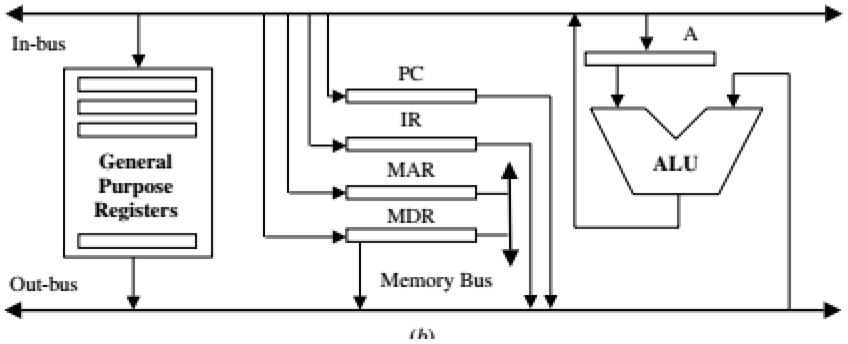
Figure 5.3 One-bus datapath

Two-Bus Organization

Using two buses is a faster solution than the one-bus organization. In this case, general-purpose registers are connected to both buses. Data can be transferred from two different registers to the input point of the ALU at the same time. Therefore, a two operand operation can fetch both operands in the same clock cycle. An additional buffer register may be needed to hold the output of the ALU when the two buses are busy carrying the two operands



In this case, the additional buffer register may be used, as one of the ALU inputs, to hold one of the operands. The ALU output can be connected directly to the in-bus, which will transfer the result into one of the registers.



Three-Bus Organization

In a three-bus organization, two buses may be used as source buses while the third is used as destination. The source buses move data out of registers (out-bus), and the destination bus may move data into a register (in-bus). Each of the two out-buses is connected to an ALU input point.

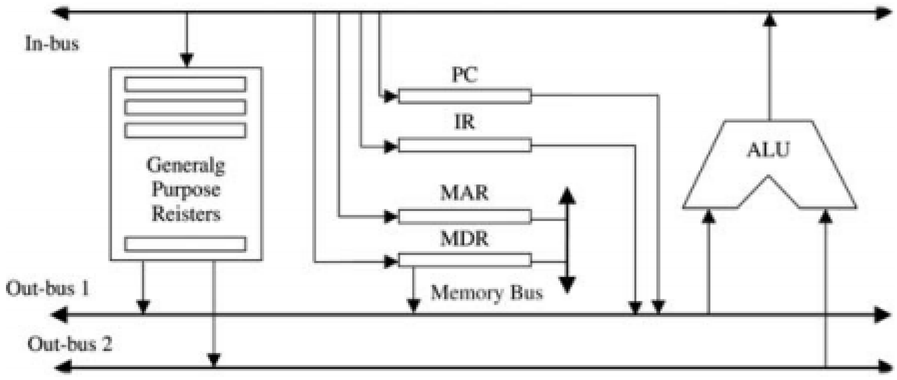


Figure 5.5 Three-bus datapath

The output of the ALU is connected directly to the in-bus. As can be expected, the more buses we have, the more data we can move within a single clock cycle. However, increasing the number of buses will also increase the complexity of the hardware.

CPU INSTRUCTION CYCLE

The sequence of operations performed by the CPU during its execution of instructions is presented in Fig. 5.6. As long as there are instructions to execute, the next instruction is fetched from main memory. The instruction is executed based on the operation specified in the opcode field of the instruction. At the completion of the instruction execution, a test is made to determine whether an interrupt has occurred. An interrupt handling routine needs to be invoked in case of an interrupt.

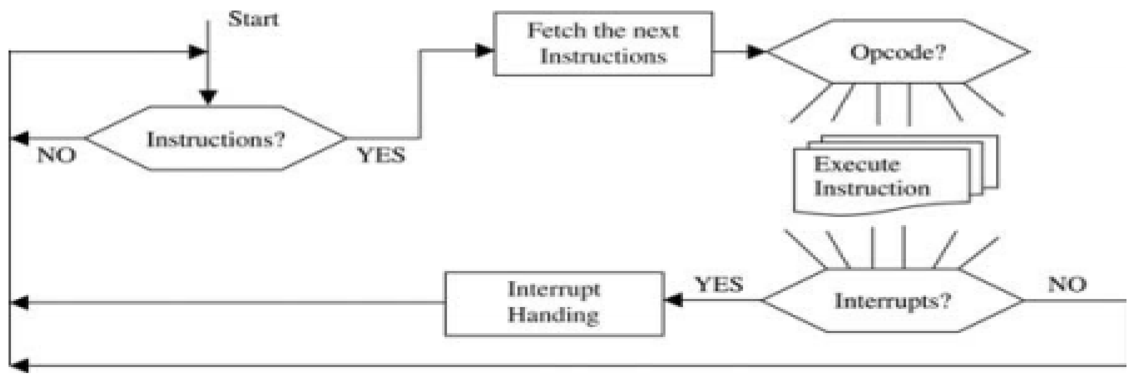


Figure 5.6 CPU functions

The basic actions during fetching an instruction, executing an instruction, or handling an interrupt are defined by a sequence of micro-operations. A group of control signals must be enabled in a prescribed sequence to trigger the execution of a microoperation.

Fetch Instructions

The sequence of events in fetching an instruction can be summarized as follows:
1. The contents of the PC are loaded into the MAR.

2. The value in the PC is incremented. (This operation can be done in parallel with a memory access.)

3. As a result of a memory read operation, the instruction is loaded into the MDR.

4. The contents of the MDR are loaded into the IR.

Let us consider the one-bus datapath organization shown in Fig. 5.3. We will see that the fetch operation can be accomplished in three steps as shown in the table below, where $t_0 < t_1 , < t_2$. Note that multiple operations separated by ";" imply that they are accomplished in parallel.

Step	Micro-operation
t_0	MAR $\leftarrow$ (PC); A $\leftarrow$ (PC)
t_1	MDR $\leftarrow$ Mem[MAR]; PC $\leftarrow$ (A) + 4
t_2	IR $\leftarrow$ (MDR)

Using the three-bus datapath shown in Figure 5.5, the following table shows the steps needed.

Step	Micro-operation
t_0	MAR $\leftarrow$ (PC); PC $\leftarrow$ (PC) + 4
t_1	MDR $\leftarrow$ Mem[MAR]
t_2	IR $\leftarrow$ (MDR)

Execute Simple Arithmetic Operation

Add R_1, R_2, R_0 This instruction adds the contents of source registers R_1 and R_2, and stores the results in destination register R_0. This addition can be executed as follows: 1. The registers R_0, R_1, R_2, are extracted from the IR. 2. The contents of R_1 and R_2 are passed to the ALU for addition. 3. The output of the ALU is transferred to R_0.

Using the one-bus datapath shown in Figure 5.3, this addition will take three steps as shown in the following table, where $t_0 > t_1 > t_2$.

Step	Micro-operation
t_0	$A \longleftarrow (R_1)$
t_1	$B \longleftarrow (R_2)$
t_2	$R_0 \longleftarrow (A) + (B)$

Using the two-bus datapath shown in Figure 5.4a, this addition will take two steps as shown in the following table, where $t_0 < t_1$.

Step	Micro-operation
t_0	$A \longleftarrow (R_1) + (R_2)$
t_1	$R_0 \longleftarrow (A)$

Using the two-bus datapath with in-bus and out-bus shown in Figure 5.4b, this addition will take two steps as shown below, where $t_0 < t_1$.

Interrupt Handling

After the execution of an instruction, a test is performed to check for pending interrupts. If there is an interrupt request waiting, the following steps take place:

1. The contents of PC are loaded into MDR (to be saved).

2. The MAR is loaded with the address at which the PC contents are to be saved.

3. The PC is loaded with the address of the first instruction of the interrupt handling routine.

4. The contents of MDR (old value of the PC) are stored in memory. The following table shows the sequence of events, where $t_1 < t_2 < t_3$.

Step	Micro-operation
t_1	MDR $\longleftarrow$ (PC)
t_2	MAR $\longleftarrow$ address1 (where to save old PC);
	PC $\longleftarrow$ address2 (interrupt handling routine)
t_3	Mem[MAR] $\longleftarrow$ (MDR)

CONTROL UNIT

The control unit is the main component that directs the system operations by sending control signals to the datapath. These signals control the flow of data within the CPU and between the CPU and external units such as memory and I/O. Control buses generally carry signals between the control unit and other computer components in a clock-driven manner. The system clock produces a continuous sequence of pulses in a specified duration and frequency. A sequence of steps t_0, t_1, t_2, . . . ,

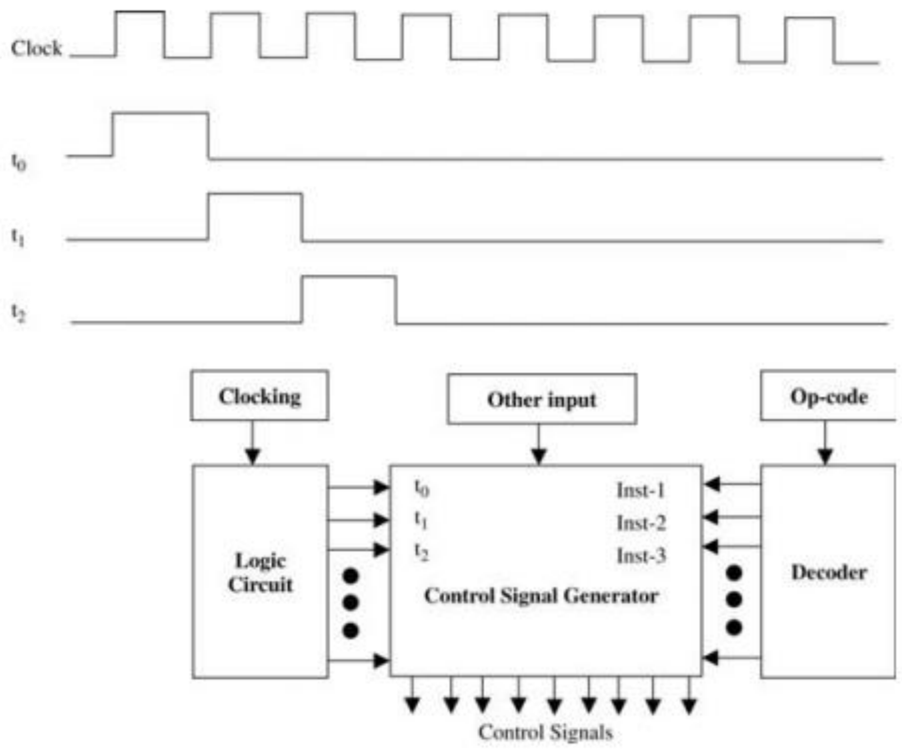


Figure 5.7 Timing of control signals

SUMMARY

The CPU is the part of a computer that interprets and carries out the instructions contained in the programs we write. The CPU's main components are the register file, ALU, and the control unit. The register file contains general-purpose and special registers. General-purpose registers may be used to hold operands and intermediate results. The special registers may be used for memory access, sequencing, status information, or to hold the fetched instruction during decoding and execution. Arithmetic and logical operations are performed in the ALU. Internal to the CPU, data may move from one register to another or between registers and ALU. Data may also move between the CPU and external components such as memory and I/O. The control unit is the component that controls the state of the instruction cycle. As long as there are instructions to execute, the next instruction is fetched from main memory. The instruction is executed based on the operation specified in the op-code field of the instruction. The control unit generates signals that control the flow of data within the CPU and between the CPU and external units such as memory and I/O.

Exercise

1. Show the micro-operations of the load, store, and jump instructions using:
   (a) One-bus system
   (b) Two-bus system
   (c) Three-bus system

Topic 6: Memory System Design I

In this chapter, we study the computer memory system. It was stated in previous topic that without a memory no information can be stored or retrieved in a computer. It is interesting to observe that as early as 1946 it was recognized by Burks, Goldstine, and Von Neumann that a computer memory has to be organized in a hierarchy. In such a hierarchy, larger and slower memories are used to supplement smaller and faster ones. This observation has since then proven essential in constructing a computer memory.

BASIC CONCEPTS

In this section, we introduce a number of fundamental concepts that relate to the memory hierarchy of a computer.

Memory Hierarchy

As mentioned above, a typical memory hierarchy starts with a small, expensive, and relatively fast unit, called the cache, followed by a larger, less expensive, and relatively slow main memory unit. Cache and main memory are built using solid-state semiconductor material (typically CMOS transistors). It is customary to call the fast memory level the primary memory. The solid-state memory is followed by larger, less expensive, and far slower magnetic memories that consist typically of the (hard) disk and the tape. It is customary to call the disk the secondary memory, while the tape is conventionally called the tertiary memory.

Thememoryhierarchycanbecharacterizedbyanumberofparameters.Amongtheseparameters are the access type, capacity, cycle time, latency, bandwidth, and cost. The term access refers to the action that physically takes place during a read or write operation. The capacity of a memory level is usually measured in bytes. The cycle time is defined as the time elapsed from the start of a read operation to the start of a subsequent read. The latency is defined as the time interval between the request for information and the access to the first bit of that information. The bandwidth provides a measure of the number of bits per second that can be accessed.

The term random access refers to the fact that any access to any memory location takes the same fixed amount of time regardless of the actual memory location and/or the sequence of accesses that takes place. For example, if a write operation to memory location 100 takes 15 ns and if this operation is followed by a read operation to memory location 3000, then the latter operation will also take 15 ns.

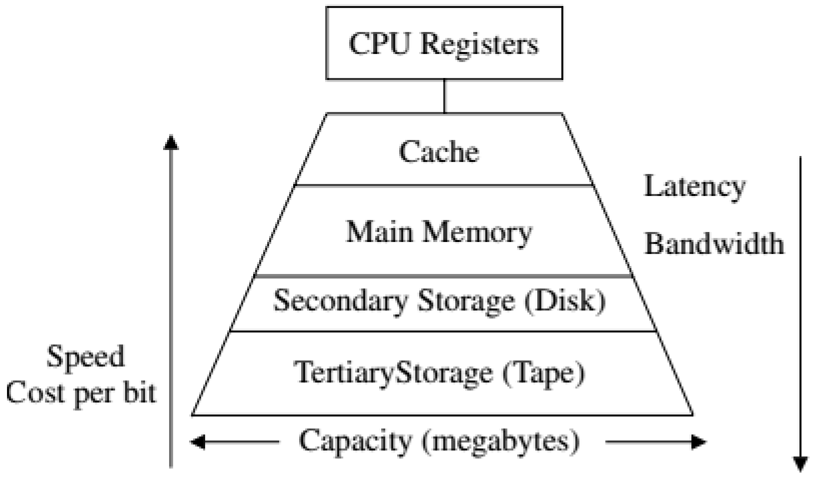


Figure 6.1 Typical memory hierarchy

CACHE MEMORY

Cache memory owes its introduction to Wilkes back in 1965. At that time, Wilkes distinguished between two types of main memory: The conventional and the slave memory. In Wilkes terminology, a slave memory is a second level of unconventional high-speed memory, which nowadays corresponds to what is called cache memory (the term cache means a safe place for hiding or storing things). The idea behind using a cache as the first level of the memory hierarchy is to keep the information expected to be used more frequently by the CPU in the cache memory.

If the request corresponds to an element that is currently residing in the cache, we call that a cache hit. On the other hand, if the request corresponds to an element that is not currently in the cache, we call that a cache miss. A cache hit ratio, h_c, is defined as the probability of finding the requested element in the cache. A cache miss ratio ($1h_c$) is defined as the probability of not finding the requested element in the cache.

SUMMARY
In this chapter, we consider the design and analysis of the first level of a memory hierarchy, that is, the cache memory. In this context, the locality issues were discussed and their effect on the average access time was explained. Three cache mapping techniques, namely direct, associative, and set-associative mappings were analyzed and their performance measures compared. We have also introduced three replacement techniques: Random, FIFO, and LRU replacement. The impact of the three techniques on the cache hit ratio was analyzed. Cache writing policies were also introduced and analyzed. Our discussion on cache ended with a presentation of the cache memory organization and characteristics of three real-life examples: Pentium IV, PowerPC, and PMC-Sierra RM7000, processors.

Topic 7: Pipelining Design Techniques

Introduction

There exist two basic techniques to increase the instruction execution rate of a processor. These are to increase the clock rate, thus decreasing the instruction execution time, or alternatively to increase the number of instructions that can be executed simultaneously. Pipelining and instruction-level parallelism are examples of the latter technique. Pipelining owes its origin to car assembly lines. The idea is to have more than one instruction being processed by the processor at the same time. Similar to the assembly line, the success of a pipeline depends upon dividing the execution of an instruction among a number of subunits (stages), each performing part of the required operations. A possible division is to consider instruction fetch (F), instruction decode (D), operand fetch (F), instruction execution (E), and store of results (S) as the subtasks needed for the execution of an instruction.

GENERAL CONCEPTS

Pipelining refers to the technique in which a given task is divided into a number of subtasks that need to be performed in sequence. Each subtask is performed by a given functional unit. The units are connected in a serial fashion and all of them operate simultaneously. The use of pipelining improves the performance compared to the traditional sequential execution of tasks.

INSTRUCTION PIPELINE

The simple analysis made in Section 9.1 ignores an important aspect that can affect the performance of a pipeline, that is, pipeline stall. A pipeline operation is said to have been stalled if one unit (stage) requires more time to perform its function, thus forcing other stages to become idle. Consider, for example, the case of an instruction fetch that incurs a cache miss. Assume also that a cache miss requires three extra time units. Figure 9.3 illustrates the effect of having instruction I_2 incurring a cache miss (assuming the execution of ten instructions I_1 to I_{10}).

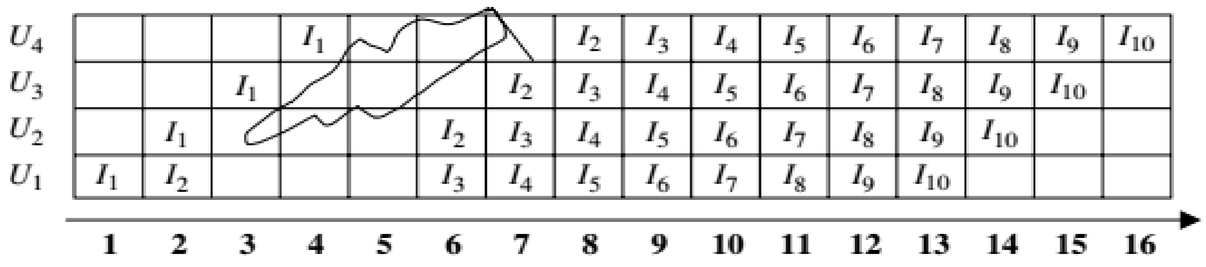

U_4: I_1 (4), I_2 … I_{10} (8–16)
U_3: I_1 (3), I_2 … I_{10} (7–15)
U_2: I_1 (2), I_2 … I_{10} (6–14)
U_1: I_1 (1), I_2 (2), I_3 … I_{10} (6–13)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Figure 9.3 Effect of a cache miss on the pipeline

The figure shows that due to the extra time units needed for instruction I_2 to be fetched, the pipeline stalls, that is, fetching of instruction I_3 and subsequent instructions are delayed. Such situations create what is known as pipeline bubble (or pipeline hazards). The creation of a pipeline bubble leads to wasted unit times, thus leading to an overall increase in the number of time units needed to finish executing a given number of instructions. The number of time units needed to execute the 10 instructions shown in Figure 9.3 is now 16 time units, compared to 13 time units if there were no cache misses. Pipeline hazards can take place for a number of other reasons. Among these are instruction dependency and data dependency.

Pipeline hazards can take place for a number of other reasons. Among these are instruction dependency and data dependency. These are explained below.

Pipeline “Stall” Due to Instruction Dependency Correct operation of a pipeline requires that operation performed by a stage MUST NOT depend on the operation(s) performed by other stage(s). Instruction dependency refers to the case whereby fetching of an instruction depends on the results of executing a previous instruction. Instruction dependency manifests itself in the execution of a conditional branch instruction.

Pipeline “Stall” Due to Data Dependency

Data dependency in a pipeline occurs when a source operand of instruction I_i depends on the results of executing a preceding instruction, I_j, $i > j$. It should be noted that although instruction I_i can be fetched, its operand(s) may not be available until the results of instruction I_j are stored.