Computer Organization and Architecture

Study Material for **MS-07/MCA/204**

Directorate of Distance Education
Guru Jambheshwar University of Science &Technology, Hisar

Study Material Prepared by Rajeev Jha & Preeti Bhardwaj

Published by Excel Books, A-45, Naraina, Phase I, New Delhi-110 028

Published by Anurag Jain for Excel Books, A-45, Naraina, Phase I, New Delhi-110 028 and printed by him at Excel Printers, C-206, Naraina, Phase I, New Delhi - 110 028

CONTENTS

Unit 1

Principles of Computer Design

Learning Objectives

After completion of this unit, you should be able to :

- describe software and hardware interaction layers in computer architecture
- Describe central processing unit
- Describe various machine language instructions
- Describe various addressing modes
- Describe various instruction types and Instruction cycle

Introduction

Copy from page-12, BSIT-301, PTU

Software

Software, or program enables a computer to perform specific tasks, as opposed to the physical components of the system (*hardware*). This includes application software such as a word processor, which enables a user to perform a task, and system software such as an operating system, which enables other software to run properly, by interfacing with hardware and with other software or custom software made to user specifications.

Types of Software

Practical computer systems divide software into three major classes: system software, programming software and application software, although the distinction is arbitrary, and often blurred.

- **System software** helps run the computer hardware and computer system. It includes operating systems, device drivers, diagnostic tools, servers, windowing systems, utilities and more. The purpose of systems software is to insulate the applications programmer as much as possible from the details of the particular computer complex being used, especially memory and other hardware features, and such accessory devices as communications, printers, readers, displays, keyboards, etc.
- **Programming software** usually provides tools to assist a programmer in writing computer programs and software using different programming languages in a more convenient way. The tools include text editors, compilers, interpreters,

linkers, debuggers, and so on. An Integrated development environment (IDE) merges those tools into a software bundle, and a programmer may not need to type multiple commands for compiling, interpreter, debugging, tracing, and etc., because the IDE usually has an advanced *graphical user interface*, or GUI.

- **Application software** allows end users to accomplish one or more specific (non-computer related) tasks. Typical applications include industrial automation, business software, educational software, medical software, databases, and computer games. Businesses are probably the biggest users of application software, but almost every field of human activity now uses some form of application software. It is used to automate all sorts of functions.

Operation

Computer software has to be "loaded" into the computer's storage (such as a *hard drive*, *memory*, or *RAM*). Once the software is loaded, the computer is able to execute the software. Computers operate by *executing* the computer program. This involves passing instructions from the application software, through the system software, to the hardware which ultimately receives the instruction as machine code. Each instruction causes the computer to carry out an operation -- moving data, carrying out a computation, or altering the control flow of instructions.
Data movement is typically from one place in memory to another. Sometimes it involves moving data between memory and registers which enable high-speed data access in the CPU. Moving data, especially large amounts of it, can be costly. So, this is sometimes avoided by using "pointers" to data instead. Computations include simple operations such as incrementing the value of a variable data element. More complex computations may involve many operations and data elements together.
Instructions may be performed sequentially, conditionally, or iteratively. Sequential instructions are those operations that are performed one after another. Conditional instructions are performed such that different sets of instructions execute depending on the value(s) of some data. In some languages this is known as an "if" statement. Iterative instructions are performed repetitively and may depend on some data value. This is sometimes called a "loop." Often, one instruction may "call" another set of instructions that are defined in some other program or module. When more than one computer processor is used, instructions may be executed simultaneously.
A simple example of the way software operates is what happens when a user selects an entry such as "Copy" from a menu. In this case, a conditional instruction is executed to copy text from data in a 'document' area residing in memory, perhaps to an intermediate storage

area known as a 'clipboard' data area. If a different menu entry such as "Paste" is chosen, the software may execute the instructions to copy the text from the clipboard data area to a specific location in the same or another document in memory.
Depending on the application, even the example above could become complicated. The field of software engineering endeavors to manage the complexity of how software operates. This is especially true for software that operates in the context of a large or powerful computer system.
Currently, almost the only limitations on the use of computer software in applications is the ingenuity of the designer/programmer. Consequently, large areas of activities (such as playing grand master level chess) formerly assumed to be incapable of software simulation are now routinely programmed. The only area that has so far proved reasonably secure from software simulation is the realm of human art- especially, pleasing music and literature.
Kinds of software by operation: computer program as executable, source code or script, configuration.

Hardware

Computer hardware is the physical part of a computer, including the digital circuitry, as distinguished from the computer software that executes within the hardware. The hardware of a computer is infrequently changed, in comparison with software and data, which are "soft" in the sense that they are readily created, modified or erased on the computer. Firmware is a special type of software that rarely, if ever, needs to be changed and so is stored on hardware devices such as read-only memory (ROM) where it is not readily changed (and is therefore "firm" rather than just "soft").

Most computer hardware is not seen by normal users. It is in embedded systems in automobiles, microwave ovens, electrocardiograph machines, compact disc players, and other devices. Personal computers, the computer hardware familiar to most people, form only a small minority of computers (about 0.2% of all new computers produced in 2003).

Personal computer hardware

A typical pc consists of a case or chassis in desktop or tower shape and the following parts:

Typical Motherboard found in a computer 

- Motherboard or system board with slots for expansion cards and holding parts
  - Central processing unit (**CPU**)
    - Computer fan - used to cool down the CPU
  - Random Access Memory (RAM) - for program execution and short term data storage, so the computer does not have to take the time to access the hard drive to find the file(s) it requires. More RAM will normally contribute to a faster PC. RAM is almost always removable as it sits in slots in the motherboard, attached with small clips. The RAM slots are normally located next to the CPU socket.
  - Basic Input-Output System (BIOS) or Extensible Firmware Interface (EFI) in some newer computers
  - Buses
- Power supply - a case that holds a transformer, voltage control, and (usually) a cooling fan
- Storage controllers of IDE, SATA, SCSI or other type, that control hard disk, floppy disk, CD-ROM and other drives; the controllers sit directly on the motherboard (on-board) or on expansion cards
- Video display controller that produces the output for the computer display. This will either be built into the motherboard or attached in its own separate slot (PCI, PCI-E or AGP), requiring a Graphics Card.
- Computer bus controllers (parallel, serial, USB, FireWire) to connect the computer to external peripheral devices such as printers or scanners
- Some type of a removable media writer:
  - CD - the most common type of removable media, cheap but fragile.
    - CD-ROM Drive
    - CD Writer
  - DVD
    - DVD-ROM Drive
    - DVD Writer
    - DVD-RAM Drive
  - Floppy disk
  - Zip drive
  - USB flash drive AKA a Pen Drive
  - Tape drive - mainly for backup and long-term storage
- Internal storage - keeps data inside the computer for later use.
  - Hard disk - for medium-term storage of data.
  - Disk array controller
- Sound card - translates signals from the system board into analog voltage levels, and has terminals to plug in speakers.

- Networking - to connect the computer to the Internet and/or other computers
    - Modem - for dial-up connections
    - Network card - for DSL/Cable internet, and/or connecting to other computers.
- Other peripherals

In addition, hardware can include external components of a computer system. The following are either standard or very common.

- Input devices
    - Text input devices
        - Keyboard
    - Pointing devices
        - Mouse
        - Trackball
    - Gaming devices
        - Joystick
        - Game pad
        - Game controller
    - Image, Video input devices
        - Image scanner
        - Webcam
    - Audio input devices
        - Microphone
- Output devices
    - Image, Video output devices
        - Printer: Peripheral device that produces a hard copy. (Inkjet, Laser)
        - Monitor: Device that takes signals and displays them. (CRT, LCD)
    - Audio output devices
        - Speakers: A device that converts analog audio signals into the equivalent air vibrations in order to make audible sound.
        - Headset: A device similar in functionality to that of a regular telephone handset but is worn on the head to keep the hands free.

Student-Activity

1. What is computer Software?
2. What is computer Hardware?
3. List various Input and Output devices.
4. Describe various Audio Output devices.
5. What is the function of RAM

Software-Hardware Interaction layers in Computer Architecture

In computer engineering, **computer architecture** is the conceptual design and fundamental operational structure of a computer system. It is a blueprint and functional description of requirements (especially speeds and interconnections) and design implementations for the various parts of a computer – focusing largely on the way by which the central processing unit (CPU) performs internally and accesses addresses in memory.

It may also be defined as the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals.

"Architecture" therefore typically refers to the fixed internal structure of the CPU (i.e. electronic switches to represent logic gates) to perform logical operations, and may also include the built-in interface (i.e. opcodes) by which hardware resources (i.e. CPU, memory, and also motherboard, peripherals) may be used by the software.

It is frequently confused with computer organization. But computer architecture is the *abstract* image of a computing system that is seen by a machine language (or assembly language) programmer, including the instruction set, memory address modes, processor registers, and address and data formats; whereas the computer organization is a lower level, more *concrete*, description of the system that involves how the constituent parts of the system are interconnected and how they interoperate in order to implement the architectural specification.

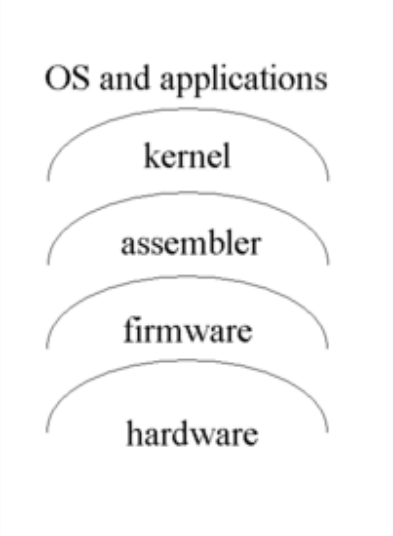

Fig : A typical vision of a computer architecture as a series of abstraction layers: hardware, firmware, assembler, kernel, operating system and applications

Abstraction Layer

An **abstraction layer** (or abstraction level) is a way of hiding the implementation details of a particular set of functionality. Perhaps the most well known software models which use layers of abstraction are the OSI 7 Layer model for computer protocols, OpenGL graphics drawing library, and the byte stream I/O model originated by Unix and adopted by MSDOS, Linux, and most other modern operating systems.

In computer science, an abstraction level is a generalization of a model or algorithm, away from any specific implementation. These generalizations arise from broad similarities that are best encapsulated by models that express similarities present in various specific implementations. The simplification provided by a good abstraction layer allows for easy reuse by distilling a useful concept or metaphor so that situations where it may be accurately applied can be quickly recognized.

A good abstraction will generalize that which can be made abstract; while allowing specificity where the abstraction breaks down and its successful application requires customization to each unique requirement or problem.

Firmware

In computing, **firmware** is software that is embedded in a hardware device. It is often provided on flash ROMs or as a binary image file that can be uploaded onto existing hardware by a user. Firmware is defined as:

- the computer program in a read-only memory (ROM) integrated circuit (a hardware part number or other configuration identifier is usually used to represent the software);
- the erasable programmable read-only memory (EPROM) chip, whose program may be modified by special external hardware, but not by [a general purpose] application program.
- the electrically erasable programmable read-only memory (EEPROM) chip, whose program may be modified by special electrical external hardware (not the usual optical light), but not by [a general purpose] application program.

Assembler

An assembly language program is translated into the target computer's machine code by a utility program called an **assembler**.Typically a modern **assembler** creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications.

Kernel

In computing, the **kernel** is the central component of most computer operating systems (OSs). Its responsibilities include managing the system's resources and the communication between hardware and software components. As a basic component of an operating system, a kernel provides the lowest-level abstraction layer for the resources (especially memory, processor and I/O devices) that applications must control to perform their function. It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls.

These tasks are done differently by different kernels, depending on their design and implementation. While monolithic kernels will try to achieve these goals by executing all the code in the same address space to increase the performance of the system, micro kernels run most of their services in user space, aiming to improve maintainability and

modularity of the code base. A range of possibilities exists between these two extremes.

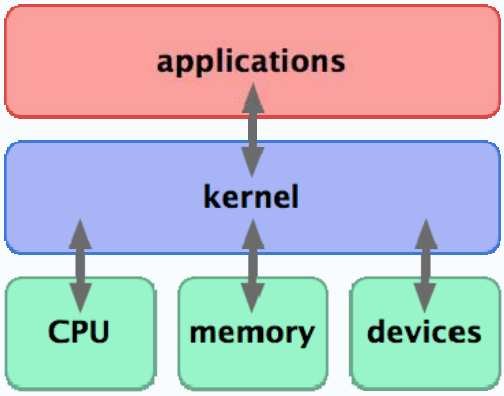

Fig : A kernel connects the application software to the hardware of a computer.

Operating System

An **operating system** (OS) is a computer program that manages the hardware and software resources of a computer. At the foundation of all system software, an operating system performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking, and managing files. It also may provide a graphical user interface for higher level functions. It forms a platform for other software.

Application Software

Application software is a subclass of computer software that employs the capabilities of a computer directly to a task that the user wishes to perform. This should be contrasted with system software which is involved in integrating a computer's various capabilities, but typically does not directly apply them in the performance of tasks that benefit the user. In this context the term application refers to both the *application software* and its implementation.

Central Processing Unit

Copy introduction from page-66, BSIT-301, PTU

Student Activity

1. Describe various software-hardware interaction layers in computer hardware.
2. Define CPU. Describe its various parts.

Machine Language Instructions

A computer executes machine language programs mechanically -- that is without understanding them or thinking about them -- simply because of the way it is physically put together. This is not an easy concept. A computer is a machine built of millions of tiny switches called transistors, which have the property that they can be wired together in such a way that an output from one switch can turn another switch on or off. As a computer computes, these switches turn each other on or off in a pattern determined both by the way they are wired together and by the program that the computer is executing.

Machine language instructions are expressed as binary numbers. A binary number is made up of just two possible digits, zero and one. So, a machine language instruction is just a sequence of zeros and ones. Each particular sequence encodes some particular instruction. The data that the computer manipulates is also encoded as binary numbers. A computer can work directly with binary numbers because switches can readily represent such numbers: Turn the switch on to represent a one; turn it off to represent a zero. Machine language instructions are stored in memory as patterns of switches turned on or off. When a machine language instruction is loaded into the CPU, all that happens is that certain switches are turned on or off in the pattern that encodes that particular instruction. The CPU is built to respond to this pattern by executing the instruction it encodes; it does this simply because of the way all the other switches in the CPU are wired together.

Addressing Modes

Copy from page-66 to page-69, upto student activity, BSIT-301, PTU

Instruction Types

The type of instruction is recognized by the computer control from the four bits in position 12 through 15 of the instruction. If the three opcode bits in positions 12 through 14 are not equal to 111, the instruction is a memory reference type and the bit in the position 15 is taken as the addressing mode. If the bit is 1, the instruction is an input-output instruction.

Only three bits of the instruction are used for the operation code. It may seem that the computer is restricted to a maximum of eight distinct operations. Since register reference and input output instructions use the remaining 12 bits as part of the total number of instruction chosen for the basic computer is equal to 25.

Data Movement Instructions:

Assembly Language Instruction:	Example:	Meaning:	Machine Language Instruction:
LOAD [REG] [MEM] STORE [MEM] [REG] MOVE [REG1] [REG2]	LOAD R2 13 STORE 8 R3 MOVE R2 R0	R2 = M[13] M[8] = R3 R2 = R0	1 000 0001 0 RR MMMMM 1 000 0010 0 RR MMMMM 1 001 0001 0000 RR RR

Arithmetic and Logic Instructions:

Instruction:	Example:	Meaning:	Machine Language Instruction:
ADD [REG1] [REG2] [REG3] SUB [REG1] [REG2] [REG3] AND [REG1] [REG2] [REG3] OR [REG1] [REG2] [REG3]	ADD R3 R2 R1 SUB R3 R1 R0 AND R0 R3 R1 OR R2 R2 R3	R3 = R2 + R1 R3 = R1 - R0 R0 = R3 & R1 R2 = R2 | R3	1 010 0001 00 RR RR RR 1 010 0010 00 RR RR RR 1 010 0011 00 RR RR RR 1 010 0100 00 RR RR RR

Branching Instructions:

Instruction:	Example:	Meaning:	Machine Language Instruction:
BRANCH [MEM] BZERO [MEM] BNEG [MEM]	BRANCH 10 BZERO 2 BNEG 7	PC = 10 PC = 2 IF ALU RESULT IS ZERO PC = 7 IF ALU RESULT IS NEGATIVE	0 000 0001 000 MMMMM 0 000 0010 000 MMMMM 0 000 0011 000 MMMMM
Other Instructions:			
Instruction:	Example:	Meaning:	Machine Language Instruction:
NOP HALT	NOP HALT	Do nothing. Halt the machine.	0000 0000 0000 0000 1111 1111 1111 1111

Student Activity

1. How will you express machine level instructions?
2. How does the computer control recognize the type of instruction?
3. Describe various types of machine level instructions.
4. Describe the addressing mode of computer instructions.

Instruction Set Selection

Copy from page-30, MCA-204, GJU
(While designing …….. upto 4 points, before timing and control)

Instructions on a RISC architecture are structured in a systematic way. They can be categorized into ALU operations, memory operations, and control operations. ALU operations always work on registers, and every register from the register set can be addressed by every ALU operation. Memory operations move values between the register set and memory. This structure makes instruction selection relatively easy, and the Java data types map 1:1 to the instruction architecture.

The optimal selection of instructions is more complex on x86 than it is on Alpha for the following reasons:

- *Different addressing modes:* Because a lot of operations exist in different addressing modes, unlike a RISC processor, the correct kind of instruction needs to be chosen in order to avoid any additional instructions for moving values. For example if the values for an addition operation are in a memory and in a register

the instruction selection algorithm always picks an add instruction that adds a memory and a register location.

- *Limited set of registers per instruction:* When picking the next instruction, the code generator always checks in which registers the current values are in and chooses the instruction appropriately. If the current register allocation doesn't fit to the instruction at all, values need to be moved. This scheme could be improved with more global analysis, but at the expense of a larger compile-time cost.
- *Efficient 64-bit operations:* The Java bytecode contains 64-bit integer and floating-point operations that the x86 platform needs to support. For each of these bytecode operations the number of temporary registers and the amount of memory accesses need to be minimized. For example, the following code is one possible implementation of the `add` (64-bit integer addition) bytecode instruction.
- `mov 0x0(%esp,1),%eax`
- `add 0x8(%esp,1),%eax`
- `mov 0x4(%esp,1),%ecx`
- `adc 0x10(%esp,1),%ecx`

Timing and control

Copy from page-30, MCA-204, GJU

Instruction Cycle

Copy from page-63-64, Instruction Cycle, MCA-301, PTU

Execution Cycle

Copy from page-50 to 54, (Instruction Execution), BSIT-301, PTU

Student Activity

1. Define an instruction cycle. Describe its various parts.
2. While designing the instruction set of a computer, what are the important things to be kept in mind? When is a set of instructions said to be complete?
3. Describe the execution cycle of an instruction.

Summary

- Our computer system consists of software and hardware. Software, or program enables a computer to perform specific tasks, as opposed to the physical components of the system (*hardware*).
- **Computer hardware** is the physical part of a computer, including the digital circuitry, as distinguished from the computer software that executes within the hardware.
- **Computer architecture** is defined as the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals.
- **A** computer architecture is considered as a series of abstraction layers: hardware, firmware, assembler, kernel, operating system and applications
- (copy summary from page-33, MCA-204, GJU)

- A program has a sequence of instructions which gets executed through a cycle , called instruction cycle. The basic parts of and instruction cycle include fetch, decode, read the effective address from memory and execute.

Keywords

Copy the following from page-33, MCA-204, GJU

- Instruction code
- Computer register
- System bus
- External bus
- Input/Output
- Interrupt

Copy the following from page-69, MCA-301, PTU

- Common Bus
- Fetch
- Control Flow Chart

Review Questions

1. Define software and hardware.
2. Describe the function of firmware.
3. What is the role of kernel.
4. What are applications?
5. Describe various machine language instructions.
6. Give different phases of instruction cycle.
7. What is the significance of instruction?
8. How are computer instructions identified.
9. What do you understand by the term instruction code?
10. Describe the time and control of instructions.

Further Readings

Copy from page-34, MCA-204, GJU

Unit-2
Control Unit and Microprogramming

Learning Objectives
After completion of this unit, you should be able to :

- describe control unit
- describe data path and control path design
- describe microprogramming
- comparing microprogramming and hardwired control
- comparing RISC and CISC architecture
- Describe pipelining in CPU Design
- Describe superscalar processors

Introduction
Copy from page-97-98, MCA-301, PTU

Control Unit

As the name suggests, a control unit is used to control something. In this case, the control unit provides instructions to the other CPU devices (previously listed) in a way that causes them to operate coherently to achieve some goal as shown in Fig. (2.1). Basically, there is one control unit, because two control units may cause conflict. The control unit of a simple CPU performs the FETCH / DECODE / EXECUTE / WRITEBACK von Neumann sequence.

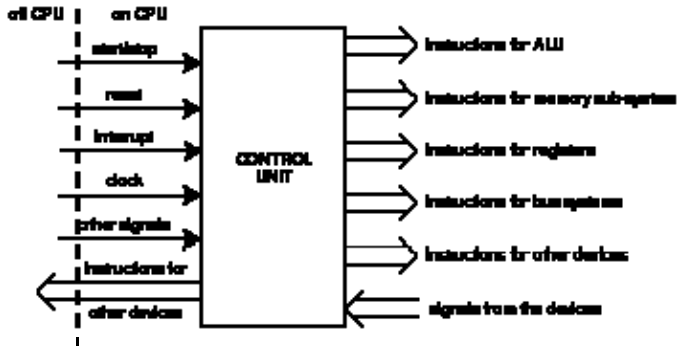

Figure 2.1: A general control unit.

To describe how the CPU works we may describe what signals the control unit issues and when. Clearly these instructions are more complicated than those that the control unit receives as input. Thus the control unit must store the instructions within itself, perhaps using a memory or perhaps in the form of a complicated network.

In either case, let us describe what the control unit does in terms of a program (for ease of understanding) called the *micro-program*, consisting naturally of *micro-instructions*. Let the micro-program be stored in a *micro-memory*.

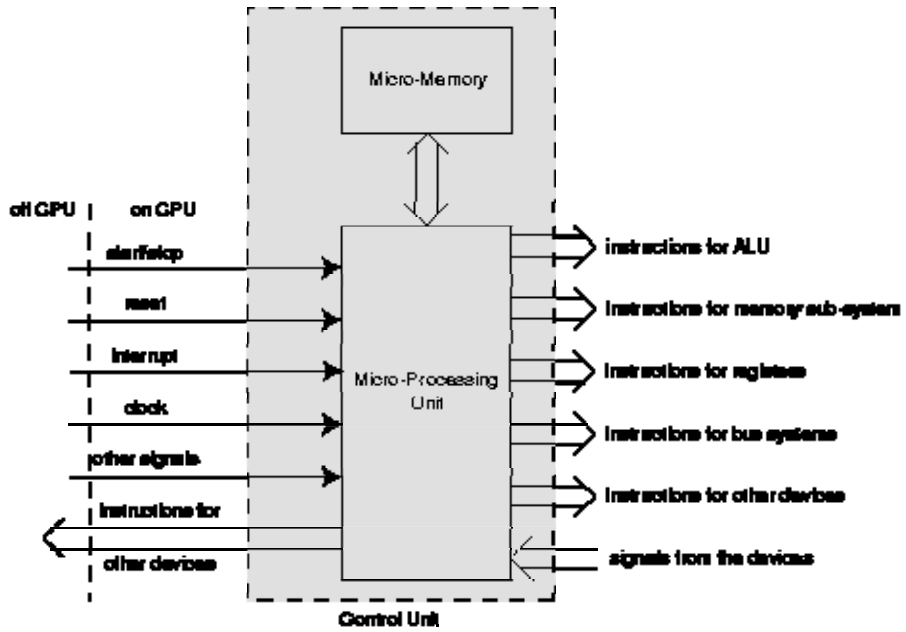

Figure 2.2: Micro-architectural view of the control unit.

The control unit may not be micro-programmed, however we can still use micro-instructions to indicate what the control unit is doing. In this case we take a logical view of the control unit. The possible instructions are dictated by the architecture of the CPU. Different architectures allow for different instructions and this is a major concept to consider when examining CPU design and operation. We are not interested in design in this subject, but we concentrate on operation.

Basic control unit operation

You should recall that the basic operation of the CPU is described by the FETCH / DECODE / EXECUTE / WRITEBACK sequence. The control unit is used to implement this sequence using a micro-program. Of the following registers, the first two are a normally only available to the control unit.

- *Instruction Register*, ***IR***: Stores the number that represents the *machine* instruction the Control Unit is to execute. (note that this is not the micro-instruction)

- *Program Counter,* PC: Stores the number that represents the address of the next instruction to execute (found in memory).
- *General Purpose Registers*: Examples are A, B, Acc, X, Y, to store intermediate results of execution.

Consider the example of a $PDP8$as shown in Fig. (2.3). The control unit is not shown.

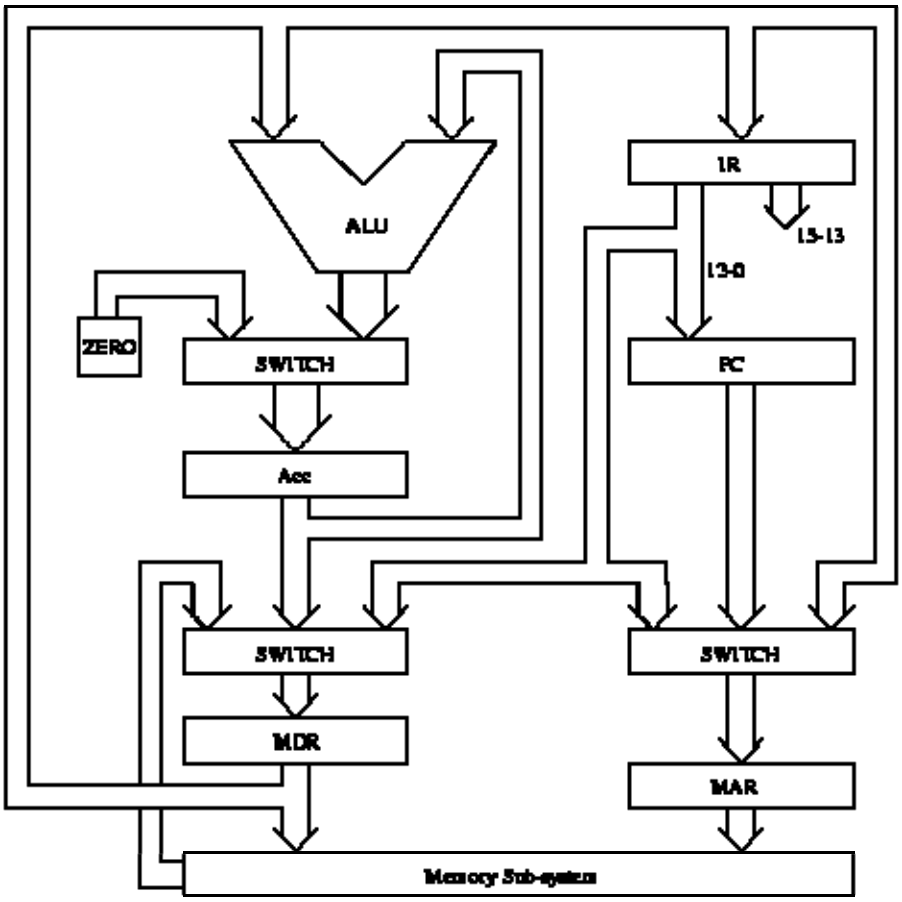

Figure 2.3: PDP8 micro-architecture.

Modern computers are more complex, but the operation is essentially the same. First consider the micro-program executed by the Control Unit to provide the FETCH:

1. $\{PC \rightarrow MAR\}$
2. $\{READ\}$
3. $\{MDR \rightarrow IR, PC + 1 \rightarrow PC\}$

Note the first line may be written $\{PC \rightarrow SWITCH, SWITCH \rightarrow MAR\}$, but in this case the path is inferred from the diagram. The second line instructs the memory sub-system to retrieve the contents of the memory at address given by MAR, the contents is put into MDR. The third line does two things at the same time, moves the MDR into the IR for DECODING and EXECUTING plus it instructs the PC to increase by 1, so as to point to the next instruction for execution. This increment instruction is available for some registers like the PC. The ALU does not have to be used in this case. Consider some more small examples of micro-programs which use the PDP8 micro-architecture.

Example zero into the Acc register:

1. $\{ZERO \rightarrow Acc\}$

□

Example the contents of the Acc to the MDR and put the result in the Acc.

1. $\{MDR \rightarrow ALU, Acc \rightarrow ALU, ADD\}$
2. $\{ALU \rightarrow Acc\}$

□

Example the two's complement of Acc.

1. $\{Acc \rightarrow ALU, COM\}$
2. $\{ALU \rightarrow Acc\}$

□

Example the Acc to obtain the next instruction.

1. $\{Acc \rightarrow MDR, MDR \rightarrow MAR\}$
2. $\{FETCH\}$

Where $FETCH$ is previously defined. □

Function of Control Unit

Copy from page-36, (third para after figure)
The function of a Control unit in a digital system …… upto page -37, Student activity-1, MCA-204, GJU

Data path and control path design

A stored program computer consists of a p*rocessing unit* and an attached *memory system*. Commands that instruct the processor to perform certain operations are placed in the memory along with the data items to be operated on. The processing unit consists of d*ata-path* and *control*. The data-path contains *registers* to hold data and *functional units*, such as arithmetic logic units and shifters, to operate on data. The control unit is little more than a finite state machine that sequences through its states to

(1) fetch the next instruction from memory,
(2) decode the instruction to interpret its meaning, and
(3) execute the instruction by moving and/or operating on data in the registers and functional units of the data-path.

The critical design issues for a data-path are how to "wire" the various components together to minimize hardware complexity and the number of control states to complete a typical operation. For control, the issue is how to organize the relatively complex "instruction interpretation" finite state machine.

Microprogramming

Copy from page-102 to page-111, upto student activity, MCA-301, PTU

RISC Vs. CISC

Copy overview of RISC/CISC from page-111

Reduced Instruction Set Computer (RISC)
Copy section 4.3, page-38, MCA-204, GJU

RISC Characteristics
Copy from page-112, MCA-301, PTU

RISC Design Philosophy

Copy section-7.4, page-60 to 62, upto student activity-2, MCA-204, GJU

Complex Instruction Set Computer (CISC)

CISC, which stands for **Complex Instruction Set Computer**, is a philosophy for designing chips that are easy to program and which make efficient use of memory. Each instruction in a CISC instruction set might perform a series of operations inside the processor. This reduces the number of instructions required to implement a given program, and allows the programmer to learn a small but flexible set of instructions.

Since the earliest machines were programmed in assembly language and memory was slow and expensive, the CISC philosophy made sense, and was commonly implemented in such large computers as the PDP-11 and the DECsystem 10 and 20 machines.

Most common microprocessor designs --- including the Intel(R) 80x86 and Motorola 68K series --- also follow the CISC philosophy.

The advantages of CISC

At the time of their initial development, CISC machines used available technologies to optimize computer performance.

- Microprogramming is as easy as assembly language to implement, and much less expensive than hardwiring a control unit.
- The ease of microcoding new instructions allowed designers to make CISC machines upwardly compatible: a new computer could run the same programs as earlier computers because the new computer would contain a superset of the instructions of the earlier computers.
- As each instruction became more capable, fewer instructions could be used to implement a given task. This made more efficient use of the relatively slow main memory.
- Because microprogram instruction sets can be written to match the constructs of high-level languages, the compiler does not have to be as complicated.

The disadvantages of CISC

Still, designers soon realized that the CISC philosophy had its own problems, including:

- Earlier generations of a processor family generally were contained as a subset in every new version --- so instruction set & chip hardware become more complex with each generation of computers.
- So that as many instructions as possible could be stored in memory with the least possible wasted space, individual instructions could be of almost any length---this means that different instructions will take different amounts of clock time to execute, slowing down the overall performance of the machine.
- Many specialized instructions aren't used frequently enough to justify their existence --- approximately 20% of the available instructions are used in a typical program.
- CISC instructions typically set the condition codes as a side effect of the instruction. Not only does setting the condition codes take time, but programmers have to remember to examine the condition code bits before a subsequent instruction changes them.

A CISC Microprocessor-the Motorola MC68000

Copy from page-62 to 67, upto student activity-4, MCA-204, GJU

THE Bridge from CISC to RISC

Copy from page-60, upto student activity-1, MCA-204, GJU

Pipelining in CPU Design

Copy section-6.2, page-50 to page-53, before student activity-1, MCA-204, GJU

A Typical Pipeline

Consider the steps necessary to do a generic operation:

- Fetch opcode.
- Decode opcode and (in parallel) prefetch a possible displacement or constant operand (or both)
- Compute complex addressing mode (e.g., [ebx+xxxx]), if applicable.
- Fetch the source value from memory (if a memory operand) and the destination register value (if applicable).
- Compute the result.
- Store result into destination register.

Assuming you're willing to pay for some extra silicon, you can build a little "mini-processor" to handle each of the above steps. The organization would look something like Figure 2.5.

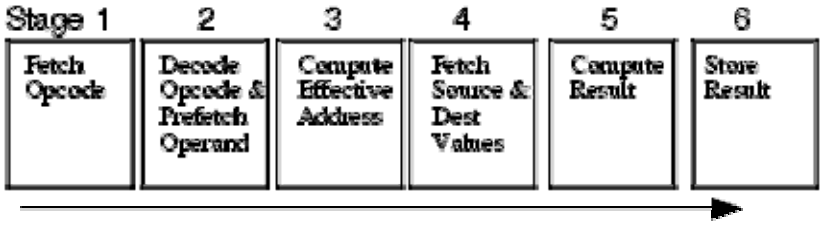

Figure 2.5 A Pipelined Implementation of Instruction Execution

Note how we've combined some stages from the previous section. For example, in stage four of Figure 2.5 the CPU fetches the source and destination operands in the same step. You can do this by putting multiple data paths inside the CPU (e.g., from the registers to

the ALU) and ensuring that no two operands ever compete for simultaneous use of the data bus (i.e., no memory-to-memory operations).

If you design a separate piece of hardware for each stage in the pipeline above, almost all these steps can take place in parallel. Of course, you cannot fetch and decode the opcode for more than one instruction at the same time, but you can fetch one opcode while decoding the previous instruction. If you have an n-stage pipeline, you will usually have n instructions executing concurrently.

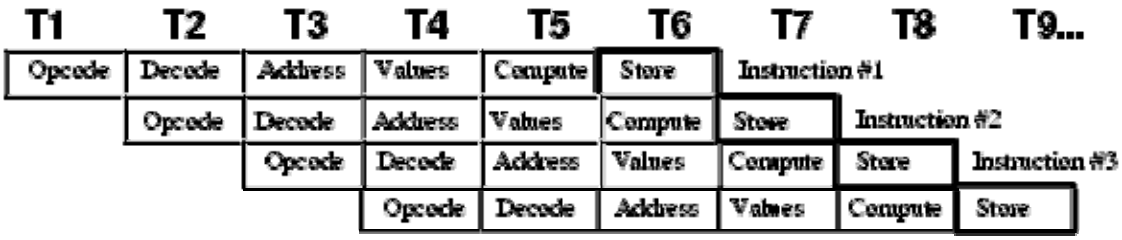

Figure 2.6 Instruction Execution in a Pipeline

Figure 2.6 shows pipelining in operatoin. T1, T2, T3, etc., represent consecutive "ticks" of the system clock. At T=T1 the CPU fetches the opcode byte for the first instruction.

At T=T2, the CPU begins decoding the opcode for the first instruction. In parallel, it fetches a block of bytes from the prefetch queue in the event the instruction has an operand. Since the first instruction no longer needs the opcode fetching circuitry, the CPU instructs it to fetch the opcode of the second instruction in parallel with the decoding of the first instruction. Note there is a minor conflict here. The CPU is attempting to fetch the next byte from the prefetch queue for use as an operand, at the same time it is fetching operand data from the prefetch queue for use as an opcode. How can it do both at once? You'll see the solution in a few moments.

At T=T3 the CPU computes an operand address for the first instruction, if any. The CPU does nothing on the first instruction if it does not use an addressing mode requiring such computation. During T3, the CPU also decodes the opcode of the second instruction and fetches any necessary operand. Finally the CPU also fetches the opcode for the third instruction. With each advancing tick of the clock, another step in the execution of each instruction in the pipeline completes, and the CPU fetches yet another instruction from memory.

This process continues until at T=T6 the CPU completes the execution of the first instruction, computes the result for the second, etc., and, finally, fetches the opcode for the sixth instruction in the pipeline. The important thing to see is that after T=T5 the CPU completes an instruction on every clock cycle. Once the CPU fills the pipeline, it completes one instruction on each cycle. Note that this is true even if there are complex addressing modes to be computed, memory operands to fetch, or other operations which use cycles on a non-pipelined processor. All you need to do is add more stages to the pipeline, and you can still effectively process each instruction in one clock cycle.

A bit earlier you saw a small conflict in the pipeline organization. At T=T2, for example, the CPU is attempting to prefetch a block of bytes for an operand and at the same time it is trying to fetch the next opcode byte. Until the CPU decodes the first instruction it doesn't know how many operands the instruction requires nor does it know their length. However, the CPU needs to know this information to determine the length of the instruction so it knows what byte to fetch as the opcode of the next instruction. So how can the pipeline fetch an instruction opcode in parallel with an address operand?

One solution is to disallow this. If an instruction as an address or constant operand, we simply delay the start of the next instruction (this is known as a *hazard* as you shall soon see). Unfortunately, many instructions have these additional operands, so this approach will have a substantial negative impact on the execution speed of the CPU.

The second solution is to throw (a lot) more hardware at the problem. Operand and constant sizes usually come in one, two, and four-byte lengths. Therefore, if we actually fetch three bytes from memory, at offsets one, three, and five, beyond the current opcode we are decoding, we know that one of these bytes will probably contain the opcode of the next instruction. Once we are through decoding the current instruction we know how long it will be and, therefore, we know the offset of the next opcode. We can use a simple data selector circuit to choose which of the three opcode bytes we want to use.

In actual practice, we have to select the next opcode byte from more than three candidates because 80x86 instructions take many different lengths. For example, an instruction that moves a 32-bit constant to a memory location can be ten or more bytes long. And there are instruction lengths for nearly every value between one and fifteen bytes. Also, some opcodes on the 80x86 are longer than one byte, so the CPU may have to fetch multiple bytes in order to properly decode the current instruction. Nevertheless, by throwing more hardware at the problem we can decode the current opcode at the same time we're fetching the next.

Stalls in a Pipeline

Unfortunately, the scenario presented in the previous section is a little too simplistic. There are two drawbacks to that simple pipeline: bus contention among instructions and non-sequential program execution. Both problems may increase the average execution time of the instructions in the pipeline.

Bus contention occurs whenever an instruction needs to access some item in memory. For example, if a "mov(reg, mem);" instruction needs to store data in memory and a "mov(mem, reg);" instruction is reading data from memory, contention for the address and data bus may develop since the CPU will be trying to simultaneously fetch data and write data in memory.

One simplistic way to handle bus contention is through a *pipeline stall*. The CPU, when faced with contention for the bus, gives priority to the instruction furthest along in the pipeline. The CPU suspends fetching opcodes until the current instruction fetches (or stores) its operand. This causes the new instruction in the pipeline to take two cycles to execute rather than one (see Figure 2.7).

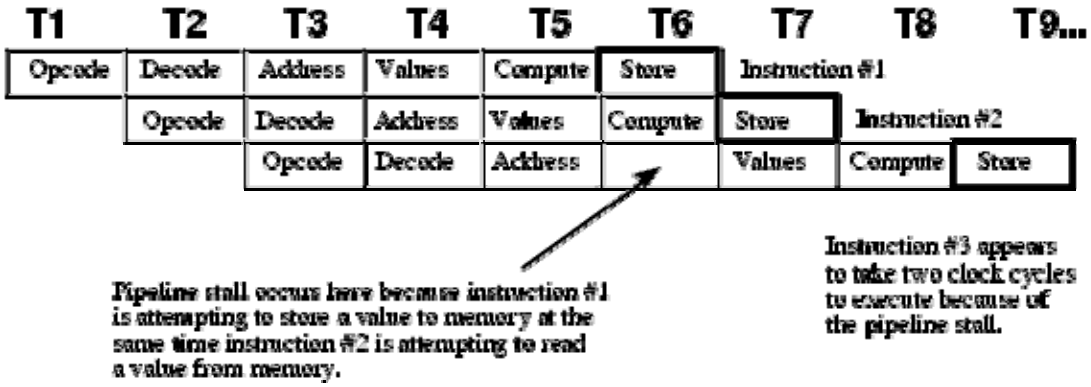

Figure 2.7 A Pipeline Stall

This example is but one case of bus contention. There are many others. For example, as noted earlier, fetching instruction operands requires access to the prefetch queue at the same time the CPU needs to fetch an opcode. Given the simple scheme above, it's unlikely that most instructions would execute at one clock per instruction (CPI).

Fortunately, the intelligent use of a cache system can eliminate many pipeline stalls like the ones discussed above. The next section on caching will describe how this is done. However, it is not always possible, even with a cache, to avoid stalling the pipeline. What you cannot fix in hardware, you can take care of with software. If you avoid using memory, you can reduce bus contention and your programs will execute faster. Likewise, using shorter instructions also reduces bus contention and the possibility of a pipeline stall.

What happens when an instruction *modifies* the EIP register? This, of course, implies that the next set of instructions to execute do not immediately follow the instruction that modifies EIP. By the time the instruction

```
JNZ             Label;
```

completes execution (assuming the zero flag is clear so the branch is taken), we've already started five other instructions and we're only one clock cycle away from the completion of the first of these. Obviously, the CPU must not execute those instructions or it will compute improper results.

The only reasonable solution is to *flush* the entire pipeline and begin fetching opcodes anew. However, doing so causes a severe execution time penalty. It will take six clock cycles (the length of the pipeline in our examples) before the next instruction completes execution. Clearly, you should avoid the use of instructions which interrupt the sequential execution of a program. This also shows another problem - pipeline length. The longer the pipeline is, the more you can accomplish per cycle in the system. However, lengthening a pipeline may slow a program if it jumps around quite a bit. Unfortunately, you cannot control the number of stages in the pipeline. You can, however, control the number of transfer instructions which appear in your programs. Obviously you should keep these to a minimum in a pipelined system.

Student Activity

Copy student activity from page-53, MCA-204, GJU

5. Describe various stalls of pipelining.

Superscalar processors

Superscalar is a term coined in the late 1980s. Superscalar processors arrived as the RISC movement gained widespread acceptance, and RISC processors are particularly suited to superscalar techniques. However, the approach can be used on non-RISC processors (e.g. Intel's P6-based processors, the Pentium 4, and AMD's IA32 clones.), with considerable effort. All current desktop and server market processors are now superscalar.

The idea, basically, is: as a logical development of pipelined design, where a new instruction starts on successive machine cycles, start up a number of machine instructions on the *same* machine cycle. By taking advantage of available memory bandwidth (*if* it is available), fetch a number of instructions simultaneously, and start executing them. [In practice, instructions need to be in a dedicated high-speed cache to be available at a high enough rate for this to work.]

The development of superscalar from 'vanilla' pipelining came about via more sophisticated, though still non-superscalar, pipelining technologies - most of which were seen on supercomputers from the mid 60s onwards. However, true superscalar processors are a concept that is unique to microprocessor-based systems.

The first developmental step is to observe that as resources grew in availability, high-performance computer systems (we are really talking about the most powerful machines of their day - Control Data 6600 and 7600, Cray-1, top-of-the-range IBM mainframes) started to acquire dedicated hardware to execute different types of instruction. So for example there may be separate execution units (EX) for floating point and integer operations. See fig 2.11.

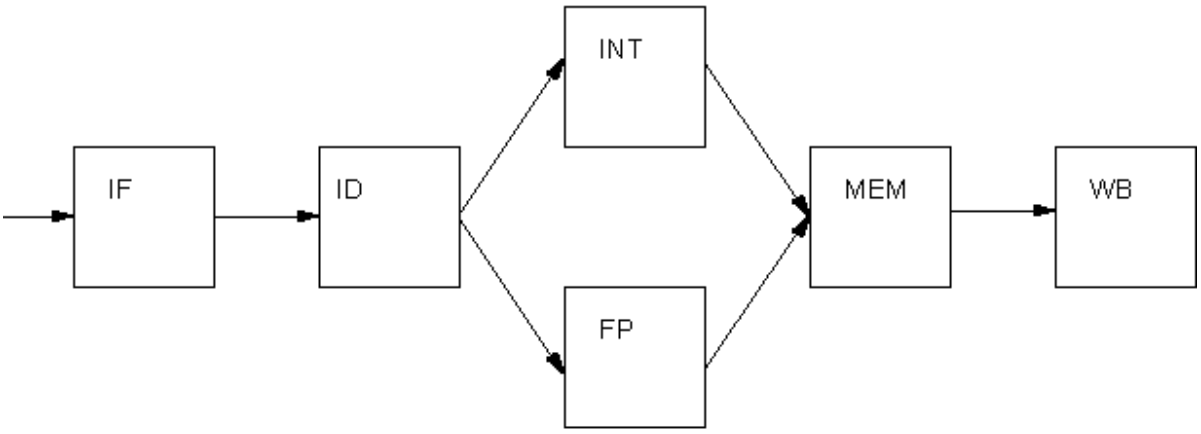

Fig.2.11. Multiple Execution Units

Once instructions had been decoded, they would be sent to the appropriate execution unit. In general, floating point operations are more complex than integer ones (though integer multiply/divide can also be time consuming), and an FP unit would be broken down into further pipeline stages. Suppose we have an FP operation followed by an integer one. Our basic pipeline would simply halt execution until the FP operation had exited the EX phase, effectively stalling the pipeline because the FP EX stage takes several cycles. However, there is no real need to [always] do that. We could allow the following integer operation to enter the Integer EX unit. Since integer operations typically take quite a bit less time than FP ones, this means that the integer result may well be available *before* the FP one that *preceeded* it in program

order. This is known as *out-of-order completion*, and brings potential performance benefits as well as big problems, as we will see.

A further step would be to observe that now we have admitted the possibility of at least partially executing instructions out of program order, we have a potential mechanism for reducing the impact of data dependencies (i.e. pipeline stalling). Instead of just leaving a pipeline idle while we are waiting for a dependency to resolve, we could pick other, later instructions and execute them instead. We have moved from a situation, with the simple pipeline, where instructions are *statically scheduled* or ordered, by the compiler, to one where they may be *dynamically scheduled* by the hardware. (Actually, the situation is a bit more complex than this and we will return to it below.)

Superscalar Concept

As resources continued to grow, yet more execution units could be made available. What is more, these would often be *duplicates*: there might be two integer units, and two FP units, and perhaps others (for example, to handle memory accesses). To justify the existence of such a large number of execution units, it may no longer sufficient to simply attempt to fetch and decode a single instruction at a time. It is necessary to also attempt to fetch, decode and write back multiple instructions as well - see fig 2.

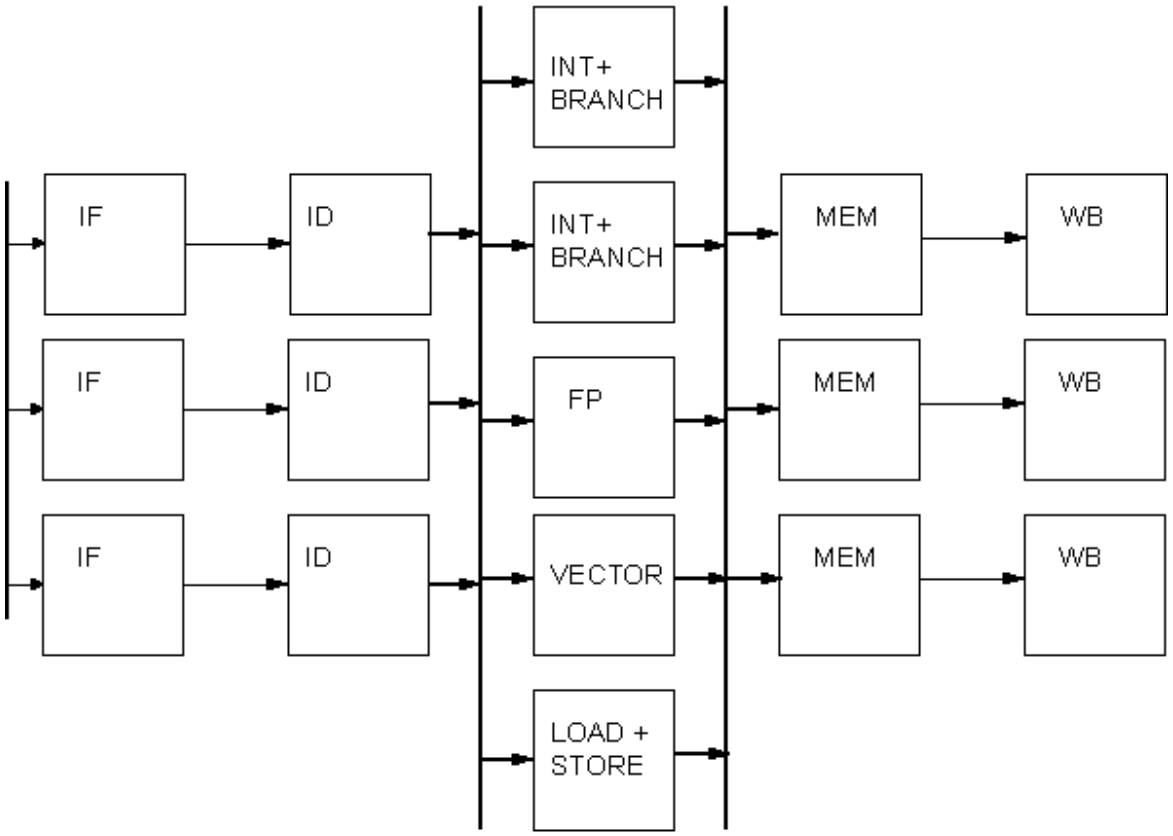

Fig.2.12. Basic Superscalar Structure

Note however that it is *not* the case that we would typically be able to fetch, decode etc. as many instructions as there are execution units - for example, in fig 2.12 most of the pipeline can handle three instructions, but there are five execution units.

We can ask ourselves: how many instructions can we profitably attempt to execute in parallel? Like less sophisticated pipelined machines, there seem to be limits on the number of instructions that can effectively be executed in parallel, before speed-up advantages become small - typically, between 2 and 8. All current superscalar machines lie within this band (for example, P6-derived processors and the Pentium 4 have three parallel pipelines). However, techniques are being looked at - and some workers are optimistic - that *may* increase this dramatically. It has to be said though that other workers are very pessimistic. A processor that attempts to execute n instructions simultaneously is said to be of degree n, or n-way. Of course, a superscalar processor will be pipelined as well: for example, a Pentium 3 has 14 pipeline stages. as well as being degree 3. Notice that a key word is 'attempts': there is

no guarantee that a superscalar processor will succeed in executing as many instructions as is, in principle, possible.

Before proceeding, we will define some useful terminology and concepts. The key problems we have to solve when building superscalar machines concern the various data dependencies, resource conflicts, and branches (especially conditional ones). However, not all phases of instruction execution are equally affected. For example, we can happily fetch and decode instructions regardless of any dependencies. Even though we may be fetching/decoding the *wrong* instructions, doing so will not affect the state of a program. However, other phases are more critical.

- **Scheduing** - this refers to the process of deciding in what order to execute instructions. The choices are *static*, when the hardware has no control, and *dynamic*, when the hardware is permitted to re-order instructions to at least some degree.
- **Issue** - this is essentially when we *fetch operands* for an instruction. Before we do this, we must be sure that they are either actually available (that is, they have already been computed and stored - either in their final destination or at least somewhere we can access them) or we at least know where they will be available in the future. In practice, resource constraints can also stop instruction issue. In our simple five-stage pipeline, the issue phase occurs when instructions leave the decode stage and enter the execute stage. Instruction issue is generally in program order.
- **Execution** - once issued, instructions can be executed once all their operands are actually available. Execution may in program order (*in-order execution*) or out of program order (*out-of-order execution*), depending on the machine.
- **Speculation and Committal** - because we are predicting the outcome of branches (and maybe other things), we do not necessarily know for sure that when we execute an instruction we will actually end up using the result. This process is known as *speculation*. Once we for sure an instruction is going to be executed then it is *committed*, or enters the *committal phase*. The next stage is to store the result.
- **Write Back** - ultimately we must store instruction results in their final destinations. (Actually, it is possible to not do this in all cases - for example, if a result actually represents a temporary, intermediate value that will be soon overwritten. However, as we will see, this raises awkward questions when dealing with exceptions, so in general all results are stored.) This phase is known as *write back* and before proceeding we must be sure that (a) the instruction will actually be committed, and (b) the destination is free to be overwritten - that is, no other instruction is waiting to use its current value as a source operand. Note in some machines, (a) does not necessarily hold - they are prepared to write uncommitted results and then 'back out' of them later on. However, we will not deal with such machines in this module. It is relatively easy to establish (b) when the destination

is a register, but difficult when it is a memory word because of the possibility of pointers creating *aliases*.
- **Retirement or Completion** - finally, an instruction finishes and leaves the pipeline. Typically this happens immediately after write back and we say the instruction is *completed* or *retired*.

Classification

We can divided superscalar processors into a number of classes of varying complexity.

- **Static Superscalar** - these processors issue and execute instructions in program order. So for example, in a degree 2 machine, it is possible to issue and two instructions simultaneously: given instructions `i1` and `i2`, we may choose to issue both, or only `i1` (depending on the presence of hazards). We may *not* just issue `i2`. To complicate and confuse matters, because the hardware has a choice (albeit limited) about issuing instructions, we say that *instruction issue* is *dynamic*. However, the actual execution of instructions is in-order we say that scheduling is *static*.
- **Dynamic Superscalar** - these machines permit out-of-order program execution, but they generally still *issue* instructions in program order. Because we can potentially re-order execution, we now say scheduling is dynamic.
- **Dynamic with Speculation** - these machines add the ability to speculate beyond branches.

Superscalar Operation- Executing Instructions in Parallel

With the pipelined architecture we could achieve, at best, execution times of one CPI (clock per instruction). Is it possible to execute instructions faster than this? At first glance you might think, "Of course not, we can do at most one operation per clock cycle. So there is no way we can execute more than one instruction per clock cycle." Keep in mind however, that a single instruction is *not* a single operation. In the examples presented earlier each instruction has taken between six and eight operations to complete. By adding seven or eight separate units to the CPU, we could effectively execute these eight operations in one clock cycle, yielding one CPI. If we add more hardware and execute, say, 16 operations at once, can we achieve 0.5 CPI? The answer is a qualified "yes." A CPU including this additional hardware is a *superscalar* CPU and can execute more than one instruction during a single clock cycle. The 80x86 family began supporting superscalar execution with the introduction of the Pentium processor.

A superscalar CPU has, essentially, several execution units (see Figure 2.12). If it encounters two or more instructions in the instruction stream (i.e., the prefetch queue) which can execute independently, it will do so.

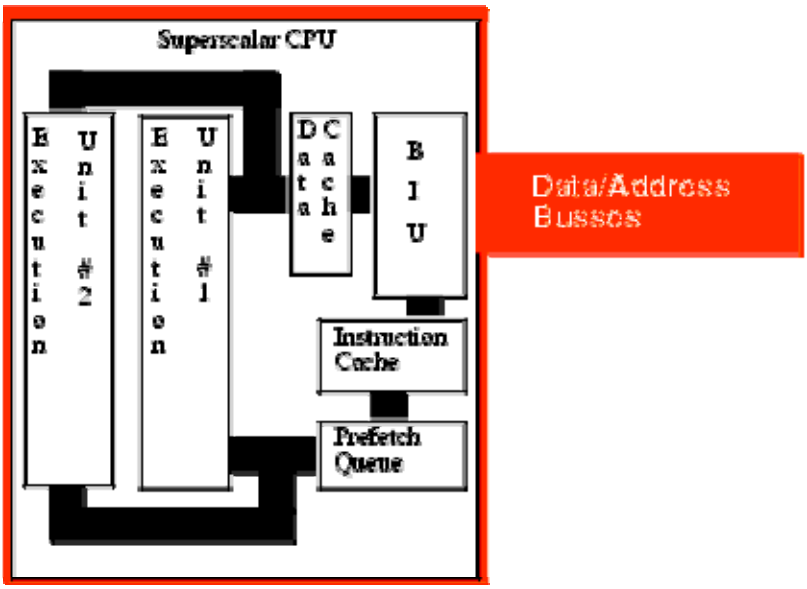

Figure 2.12 A CPU that Supports Superscalar Operation

There are a couple of advantages to going superscalar. Suppose you have the following instructions in the instruction stream:

```
mov( 1000, eax );

mov( 2000, ebx );
```

If there are no other problems or hazards in the surrounding code, and all six bytes for these two instructions are currently in the prefetch queue, there is no reason why the CPU cannot fetch and execute both instructions in parallel. All it takes is extra silicon on the CPU chip to implement two execution units.

Besides speeding up independent instructions, a superscalar CPU can also speed up program sequences that have hazards. One limitation of superscalar CPU is that once a hazard occurs, the offending instruction will completely stall the pipeline. Every instruction which follows will also have to wait for the CPU to synchronize the execution of the instructions. With a superscalar CPU, however, instructions following the hazard may continue execution through the pipeline as long as they don't have hazards of their own. This alleviates (though does not eliminate) some of the need for careful instruction scheduling.

As an assembly language programmer, the way you write software for a superscalar CPU can dramatically affect its performance. First and foremost is that rule you're probably sick of by now: *use short instructions*. The shorter your instructions are, the more instructions the CPU can fetch in a single operation and, therefore, the more likely the CPU will execute faster than one CPI. Most superscalar CPUs do not completely duplicate the execution unit. There might be multiple ALUs, floating point units, etc. This means that certain instruction sequences can execute very quickly while others won't. You have to study the exact composition of your CPU to decide which instruction sequences produce the best performance.

Out of Order Execution

In a standard superscalar CPU it is the programmer's (or compiler's) responsibility to schedule (arrange) the instructions to avoid hazards and pipeline stalls. Fancier CPUs can actually remove some of this burden and improve performance by automatically rescheduling instructions while the program executes. To understand how this is possible, consider the following instruction sequence:

```
mov( SomeVar, ebx );

mov( [ebx], eax );

mov( 2000, ecx );
```

A data hazard exists between the first and second instructions above. The second instruction must delay until the first instruction completes execution. This introduces a pipeline stall and increases the running time of the program. Typically, the stall affects every instruction that follows. However, note that the third instruction's execution does not depend on the result from either of the first two instructions. Therefore, there is no reason to stall the execution of the "mov(2000, ecx);" instruction. It may continue executing while the second instruction waits for the first to complete. This technique, appearing in later members of the Pentium line, is called "out of order execution" because the CPU completes the execution of some instruction prior to the execution of previous instructions appearing in the code stream.

Clearly, the CPU may only execute instruction out of sequence if doing so produces exactly the same results as in-order execution. While there a lots of little technical issues that make this problem a little more difficult than it seems, with enough engineering effort it is quite possible to implement this feature.

Although you might think that this extra effort is not worth it (why not make it the programmer's or compiler's responsibility to schedule the instructions) there are some situations where out of order execution will improve performance that static scheduling could not handle.

Register Renaming

One problem that hampers the effectiveness of superscalar operation on the 80x86 CPU is the 80x86's limited number of general purpose registers. Suppose, for example, that the CPU had four different pipelines and, therefore, was capable of executing four instructions simultaneously. Actually achieving four instructions per clock cycle would be very difficult because most instructions (that can execute simultaneously with other instructions) operate on two register operands. For four instructions to execute concurrently, you'd need four separate destination registers and four source registers (and the two sets of registers must be disjoint, that is, a destination register for one instruction cannot be the source of another). CPUs that have lots of registers can handle this task quite easily, but the limited register set of the 80x86 makes this difficult. Fortunately, there is a way to alleviate part of the problem: through *register renaming*.

Register renaming is a sneaky way to give a CPU more registers than it actually has. Programmers will not have direct access to these extra registers, but the CPU can use these additional register to prevent hazards in certain cases. For example, consider the following short instruction sequence:

```
mov( 0, eax );

mov( eax, i );

mov( 50, eax );

mov( eax, j );
```

Clearly a data hazard exists between the first and second instructions and, likewise, a data hazard exists between the third and fourth instructions in this sequence. Out of order execution in a superscalar CPU would normally allow the first and third instructions to execute concurrently and then the second and fourth instructions could also execute concurrently. However, a data hazard, of sorts, also exists between the first and third instructions since they use the same register. The programmer could have easily solved this problem by using a different register (say EBX) for the third and fourth instructions. However, let's assume that the programmer was unable to do this because the other registers are all holding important values. Is this sequence doomed to executing in four cycles on a superscalar CPU that should only require two?

One advanced trick a CPU can employ is to create a bank of registers for each of the general purpose registers on the CPU. That is, rather than having a single EAX register, the CPU could support an array of EAX registers; let's call these registers EAX[0], EAX[1], EAX[2], etc. Similarly, you could have an array of each of the registers, so we could also have EBX[0]..EBX[n], ECX[0]..ECX[n], etc. Now the instruction set does not give the programmer the ability to select one of these specific register array elements for a given instruction, but the CPU can automatically choose a different register array

element if doing so would not change the overall computation and doing so could speed up the execution of the program. For example, consider the following sequence (with register array elements automatically chosen by the CPU):

```
mov( 0, eax[0] );
mov( eax[0], i );
mov( 50, eax[1] );
mov( eax[1], j );
```

Since EAX[0] and EAX[1] are different registers, the CPU can execute the first and third instructions concurrently. Likewise, the CPU can execute the second and fourth instructions concurrently.

The code above provides an example of *register renaming*. Dynamically, the CPU automatically selects one of several different elements from a register array in order to prevent data hazards. Although this is a simple example, and different CPUs implement register renaming in many different ways, this example does demonstrate how the CPU can improve performance in certain instances through the use of this technique.

Very Long Instruction Word Architecture (VLIW)

Superscalar operation attempts to schedule, in hardware, the execution of multiple instructions simultaneously. Another technique that Intel is using in their IA-64 architecture is the use of very long instruction words, or VLIW. In a VLIW computer system, the CPU fetches a large block of bytes (41 in the case of the IA-64 Itanium CPU) and decodes and executes this block all at once. This block of bytes usually contains two or more instructions (three in the case of the IA-64). VLIW computing requires the programmer or compiler to properly schedule the instructions in each block (so there are no hazards or other conflicts), but if properly scheduled, the CPU can execute three or more instructions per clock cycle.

The Intel IA-64 Architecture is not the only computer system to employ a VLIW architecture. Transmeta's Crusoe processor family also uses a VLIW architecture. The Crusoe processor is different than the IA-64 architecture insofar as it does not support native execution of IA-32 instructions. Instead, the Crusoe processor dynamically translates 80x86 instructions to Crusoe's VLIW instructions. This "code morphing" technology results in code running about 50% slower than native code, though the Crusoe processor has other advantages.

We will not consider VLIW computing any further since the IA-32 architecture does not support it. But keep this architectural advance in mind if you move towards the IA-64 family or the Crusoe family.

Student Activity

1. What are superscalar processors?
2. Describe various types of superscalar processors.
3. What are the problems associated with superscalar processors?

Summary

Copy section 5.6 from page-48, MCA-204, GJU
Copy first two points of summary from page-57, MCA-204,GJU

Keywords

Copy from page-48-49, MCA-204, GJU
Copy Pipelining Processing from page-57, MCA-204, GJU

Review Questions

Copy Q-1 to 5 from page-49, MCA-204, GJU

6. Describe pipeline processing.
7. Describe Superscalar processors, their functioning and classification

Further Readings

Copy from page-58, MCA-204, GJU

Unit-3

Memory Organization

Learning Objectives

After completion of this unit, you should be able to:

- describe memory subsystem
- describe various storage technologies
- describe memory array organization
- understand memory hierarchy and interleaving
- describe cache and virtual memory
- describe various architectural aids to implement cache and virtual memory.

Introduction

Copy from page-149, MCA-301, PTU

Memory Subsystem

Copy from page-150 (Memory System), MCA-301, PTU

Storage Technologies

Copy from page-132-133 (upto Student Activity)

Memory Array Organization

A typical 80x86 processor addresses a maximum of 2^n different memory locations, where *n* is the number of bits on the address bus, as 80x86 processors have 20, 24, 32, and 36 bit address buses (with 64 bits on the way).

Of course, the first question you should ask is, "What exactly is a memory location?" The 80x86 supports *byte addressable memory*. Therefore, the basic memory unit is a byte. So with 20, 24, 32, and 36 address lines, the 80x86 processors can address one megabyte, 16 megabytes, four gigabytes, and 64 gigabytes of memory, respectively.

Think of memory as a linear array of bytes. The address of the first byte is zero and the address of the last byte is 2^n-1. For an 8088 with a 20 bit address bus, the following pseudo-Pascal array declaration is a good approximation of memory:

Memory: array [0..1048575] of byte;

To execute the equivalent of the Pascal statement "Memory [125] := 0;" the CPU places the value zero on the data bus, the address 125 on the address bus, and asserts the write line (since the CPU is writing data to memory), see Figure 1.2.

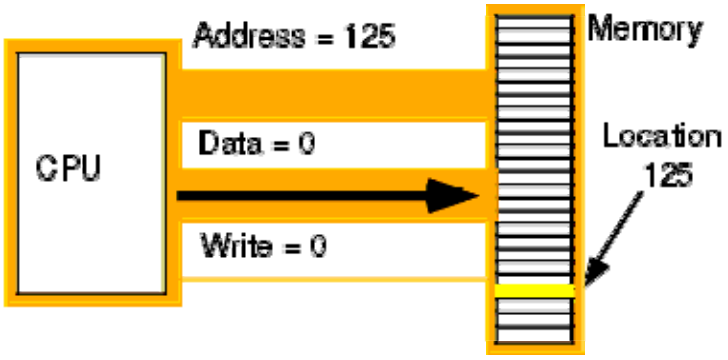

Figure 1.2 Memory Write Operation

To execute the equivalent of "CPU := Memory [125];" the CPU places the address 125 on the address bus, asserts the read line (since the CPU is reading data from memory), and then reads the resulting data from the data bus (see Figure 1.3).

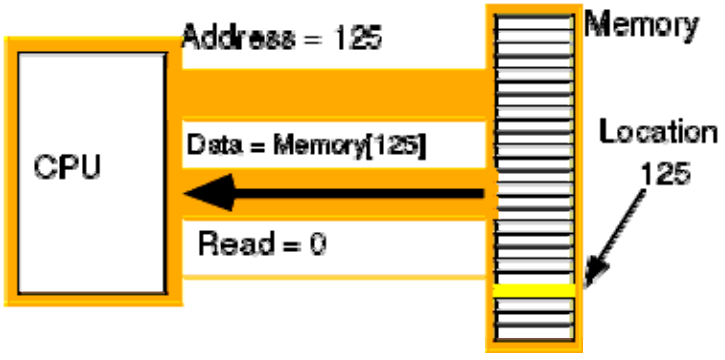

Figure 1.3 Memory Read Operation

The above discussion applies *only* when accessing a single byte in memory. So what happens when the processor accesses a word or a double word? Since memory consists of an array of bytes, how can we possibly deal with values larger than eight bits?

Different computer systems have different solutions to this problem. The 80x86 family deals with this problem by storing the L.O. byte of a word at the address specified and the H.O. byte at the next location. Therefore, a word consumes two consecutive memory addresses (as you would expect, since a word consists of two bytes). Similarly, a double word consumes four consecutive memory locations. The address for the double word is the address of its L.O. byte. The remaining three bytes follow this L.O. byte, with the H.O. byte appearing at the address of the double word *plus three* (see Figure 1.4). Bytes, words, and double words may begin at *any* valid address in memory. We will soon see, however, that starting larger objects at an arbitrary address is not a good idea.

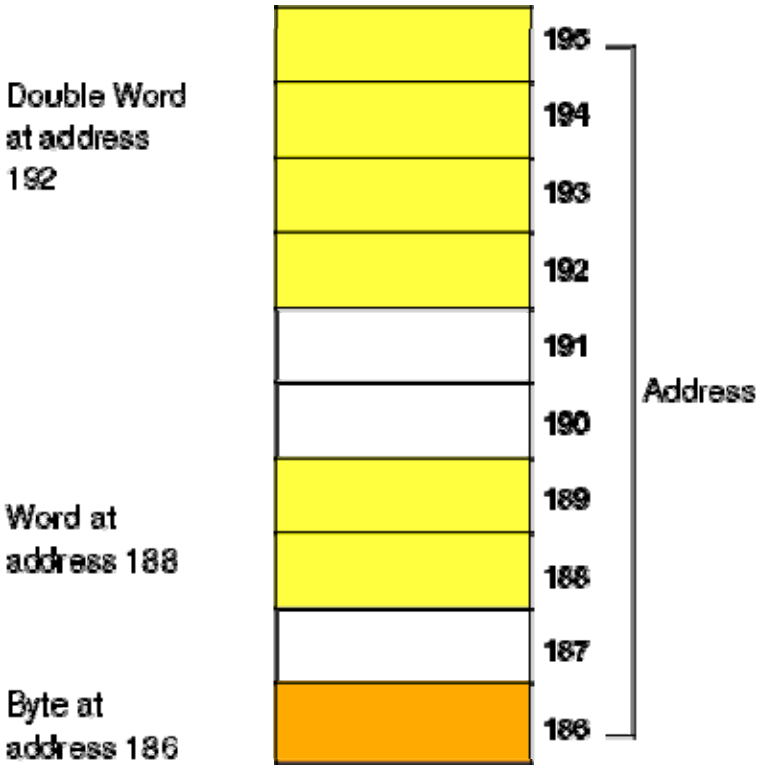

Figure 1.4 Byte, Word, and DWord Storage in Memory

Note that it is quite possible for byte, word, and double word values to overlap in memory. For example, in Figure 1.4 you could have a word variable beginning at address 193, a byte variable at address 194, and a double word value beginning at address 192. These variables would all overlap.

By carefully arranging how you use memory, you can improve the speed of your program on these CPUs.

Memory Management

Copy section 8.6 from page 74 to 87, upto student activity-2, MCA-204, GJU

Memory Hierarchy

Copy section 8.10 onwards (from page-91 to page-105), upto student activity-5, MCA-204, GJU

Cache Memory

Copy from page – 159 to 161, upto set associative mapping, MCA-301, PTU

Summary

Copy from page-105 (delete first two points), MCA-204, GJU

Key Words

Copy from page-105-106 (delete first two definitions), MCA-204, GJU

Review questions

copy question 1 to 8, from page-165, MCA-301, PTU

Further Readings

Copy from page-106, MCA-204, GJU

Unit-4
Input-Output Devices and Characteristics

Learning Objectives
After completion of this unit, you should be able to :

- describe input-output processing
- describe bus interface
- describe I/O Interrupt channels
- Describe performance evaluation-SPEC-MARKS
- Describe various benchmarks of transaction processing

Introduction
Copy from page-117 to 118 (Introduction and I/O and their brief description), MCA-301, PTU

Input-Output Processing
Copy from page-118 (I/O Processing), MCA-301, PTU

Input-Output Processor
Copy from page-139 to page-144 (upto figure-5.23), MCA-301, PTU

Bus Interface
Copy from page-118 (Bus Interface) to Student Activity, Page-129, MCA-301, PTU

I/O Interrupts Channels
Copy from page-129(Interrupt-Initiated I/O) , MCA-301, PTU

Communication between the CPU and the channel
Copy from page-73-74, MCA-204, GJU

I/O Interrupts
Copy from page-130 to 136 (before DMA), MCA-301, PTU

Student Activity
1. What is an Interrupt?
2. What is the purpose of having a channel?
3. The CPU and the channel are usually in a master-slave relationship. Explain.
4. Explain the CPU I/O instructions executed by the CPU.
5. (Copy question-1 to 5 of student activity, page-144, MCA-301, PTU)

Performance Evaluation- Benchmarks
copy from section-10.2 to section-10.4, page-114 to 118, MCA-204, GJU

Student Activity

1. Describe the following :
    (a) Spec-Mark
    (b) TPC-H
    (c) TPC-R
    (d) TPC-W

Summary

Copy from page-145, Summary (copy para 1,2 and 4 only), Page-145, MCA-301, PTU
Copy from page-118, summary, MCA-204, GJU

Keywords

Copy keywords from page-145, (Delete DMS), MCA-301, PTU
Copy keywords from page-118, MCA-204, GJU

Review Questions

Copy Q-1,2,3,5,6,7 from page-145-146, MCA-301, PTU
Copy Q-1to5 from page-118-119, MCA-204, GJU

Further Readings

Copy from page-146, MCA-301, PTU and page-119, MCA-204, GJU