Q.2 a. What are the different performance measures used to represent a computer system's performance?

Answer:

Metrics are criteria to compare the performances of a system. In general, the metrics are related to speed, accuracy, reliability and availability of services. The basic characteristics of a computer system that we typically need
to measure are:

- a count of how many times an event occurs,
- the duration of some time interval, and
- the size of some parameter.

From these types of measured values, we can derive the actual value that we wish to use to describe the system: the performance metric.

- Reliability A system A always outperforms a system B , the performance metric indicates that A always outperforms B.
- Repeatability The same value of the metric is measured each time the same experiments are performed.
- Consistency Units of the metrics and its precise definition are the same across different systems and different configurations of the same system.
- Linearity The value of the metric should be linearly proportional to the actual performance of the machine.
- Easiness of measurement If a metric is hard to measure, it is unlikely anyone will actually use it. Moreover it is more likely to be incorrectly determined. Independence Metrics should not be defined to favor particular systems.
- Time between the start and the end of an operation. Also called running time, elapsed time, wall-clock time, response time, latency, execution time, ... Most straightforward measure: "my program takes 12.5s on Pentium 3.5GHz" Can be normalized to some reference time .Must be measured on a "dedicated" machine

Other parameters are:

MIPS Millions of instructions / sec
But Instructions Set Architectures are not equivalent
1 CISC instruction = many RISC instructions

Programs use different instruction mixes
MFlops Millions of floating point operations /sec
=---very popular, but often misleading

e.g., A high MFlops rate in a stupid algorithm could have poor application performance

Application-specific

- Millions of frames rendered per second
- Millions of amino-acid compared per second
- Millions of HTTP requests served per seconds

Application-specific metrics are often preferable and others may be misleading

This is often the way in which people say that a computer Is better than another. More instruction per seconds for higher clock rate. Faces the same problems as MIPS

b. Explain Big-endian and Little-endian byte-addresses assignment with example.

Answer: Refer Page No 35 from Text Book

c. Describe in brief the different generations of computer.

Answer: Refer Page No 19 from Text Book

Q.3 a. Bring out the four key differences between subroutine and interrupt service routine.

Answer:

Subroutine runs when you call it. ISR runs whenever a certain signal occurs. (The signal can be generated by software or hardware.) The big difference is that you know where the subroutine runs (because you call it). But you do not know when the ISR will be executed. You code may run normally when a hardware interrupt occurs and your program jumps to the ISR. This can happen anywhere in your code (in between two statements or even in the middle of a complete statement, remember a statement is compiled into multiple assembly instructions).

Therefore, cares must be taken when ISR accesses global variables. Race condition may occur if ISR and a normal thread touch the same global variable at the same time.

The interrupt service routine (on the 14-bit core there is only one) is just like a subroutine. The difference is that when you call a subroutine, you call it when you decide, and you understand completely what will be changed by the subroutine, and your code can be prepared for, even welcome, those changes.

But the interrupt service routine gets called when you least expect it, kind of like that unwelcome call from the mother in law. No matter what you are doing you are unceremoniously ripped out of it to go answer the phone. Because of this, it becomes the

© IETE

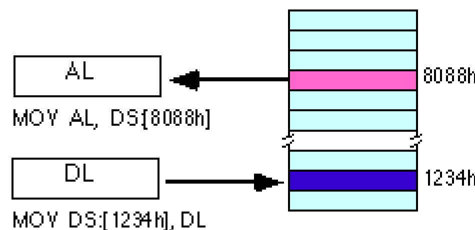
responsibility of the interrupt service routine to carefully put back anything it may have disturbed. This is probably the trickiest part of interrupt handling, not because it is so difficult, but because the bugs a mistake can introduce will be terribly difficult to sort out. On the other hand, by not having to poll for certain types of events, you code can sometimes be greatly simplified.

b. Define addressing mode. With the help of example explain different addressing modes.

Answer:

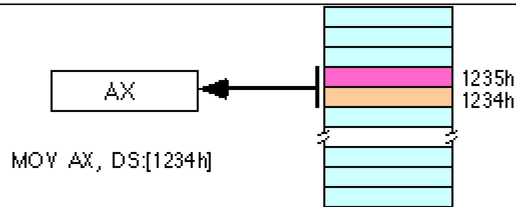
The Displacement Only Addressing Mode

The most common addressing mode, and the one that's easiest to understand, is the displacement-only (or direct) addressing mode. The displacement-only addressing mode consists of a 16 bit constant that specifies the address of the target location. The instruction `mov al,ds:[8088h]` loads the `al` register with a copy of the byte at memory location 8088h. Likewise, the instruction `mov ds:[1234h],dl` stores the value in the `dl` register to memory location 1234h:



The displacement-only addressing mode is perfect for accessing simple variables. Of course, you'd probably prefer using names like "I" or "J" rather than "DS:[1234h]" or "DS:[8088h]". Well, fear not, you'll soon see it's possible to do just that.

Intel named this the displacement-only addressing mode because a 16 bit constant (displacement) follows the `mov` opcode in memory. In that respect it is quite similar to the direct addressing mode on the x86 processors (see the previous chapter). There are some minor differences, however. First of all, a displacement is exactly that- some distance from some other point. On the x86, a direct address can be thought of as a displacement from address zero. On the 80x86 processors, this displacement is an offset from the beginning of a segment (the data segment in this example). Don't worry if this doesn't make a lot of sense right now. You'll get an opportunity to study segments to your heart's content a little later in this chapter. For now, you can think of the displacement-only addressing mode as a direct addressing mode. The examples in this chapter will typically access bytes in memory. Don't forget, however, that you can also access words on the 8086 processors :



By default, all displacement-only values provide offsets into the data segment. If you want to provide an offset into a different segment, you must use a segment override prefix before your address. For example, to access location 1234h in the extra segment (`es`) you would use an instruction of the form `mov ax,es:[1234h]`. Likewise, to access this location in the code segment you would use the instruction `mov ax, cs:[1234h]`. The `ds:` prefix in the previous examples is not a segment override. The CPU uses the data segment register by default. These specific examples require `ds:` because of MASM's syntactical limitations.

1.2.2 The Register Indirect Addressing Modes

The 80x86 CPUs let you access memory indirectly through a register using the register indirect addressing modes. There are four forms of this addressing mode on the 8086, best demonstrated by the following instructions:

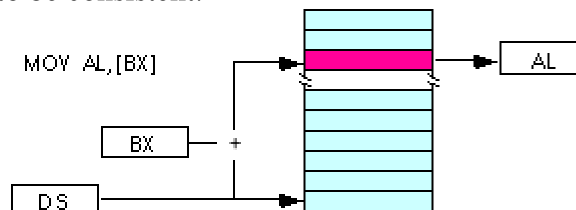
```
mov     al, [bx]
mov     al, [bp]
mov     al, [si]
mov     al, [di]
```

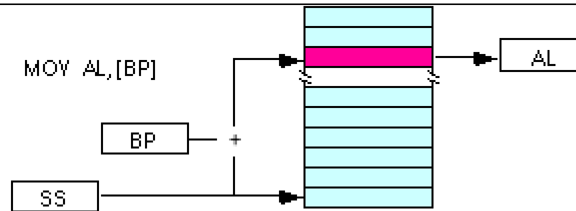
As with the x86 `[bx]` addressing mode, these four addressing modes reference the byte at the offset found in the `bx`, `bp`, `si`, or `di` register, respectively. The `[bx]`, `[si]`, and `[di]` modes use the `ds` segment by default. The `[bp]` addressing mode uses the stack segment (`ss`) by default.

You can use the segment override prefix symbols if you wish to access data in different segments. The following instructions demonstrate the use of these overrides:

```
mov     al, cs:[bx]
mov     al, ds:[bp]
mov     al, ss:[si]
mov     al, es:[di]
```

Intel refers to `[bx]` and `[bp]` as base addressing modes and `bx` and `bp` as base registers (in fact, `bp` stands for base pointer). Intel refers to the `[si]` and `[di]` addressing modes as indexed addressing modes (`si` stands for source index, `di` stands for destination index). However, these addressing modes are functionally equivalent. This text will call these forms register indirect modes to be consistent.





Note: the `[si]` and `[di]` addressing modes work exactly the same way, just substitute `si` and `di` for `bx` above.

1.2.3 Indexed Addressing Modes

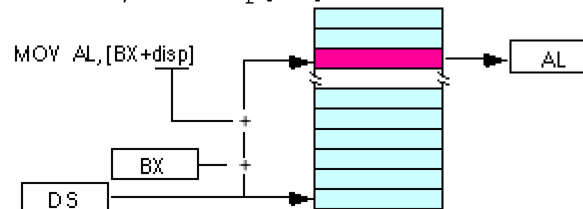
The indexed addressing modes use the following syntax:

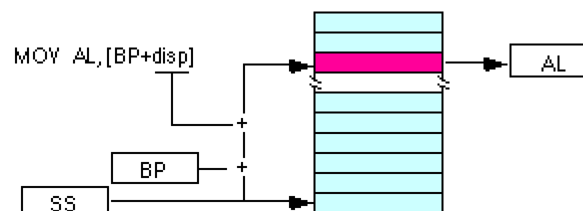
```
mov     al, disp[bx]
mov     al, disp[bp]
mov     al, disp[si]
mov     al, disp[di]
```

If `bx` contains 1000h, then the instruction `mov cl,20h[bx]` will load `cl` from memory location ds:1020h. Likewise, if `bp` contains 2020h, `mov dh,1000h[bp]` will load `dh` from location ss:3020.

The offsets generated by these addressing modes are the sum of the constant and the specified register. The addressing modes involving `bx, si,` and `di` all use the data segment, the `disp[bp]` addressing mode uses the stack segment by default. As with the register indirect addressing modes, you can use the segment override prefixes to specify a different segment:

```
mov     al, ss:disp[bx]
mov     al, es:disp[bp]
mov     al, cs:disp[si]
mov     al, ss:disp[di]
```





You may substitute `si` or `di` in the figure above to obtain the `[si+disp]` and `[di+disp]` addressing modes.

Note that Intel still refers to these addressing modes as based addressing and indexed addressing. Intel's literature does not differentiate between these modes with or without the constant. If you look at how the hardware works, this is a reasonable definition. From the

programmer's point of view, however, these addressing modes are useful for entirely different things. Which is why this text uses different terms to describe them. Unfortunately, there is very little consensus on the use of these terms in the 80x86 world.

Based Indexed Addressing Modes

The based indexed addressing modes are simply combinations of the register indirect addressing modes. These addressing modes form the offset by adding together a base register (bx or bp) and an index register (si or di). The allowable forms for these addressing modes are

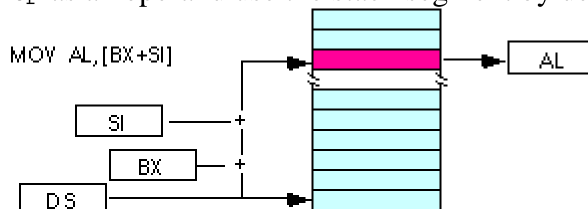
```
mov     al, [bx][si]
mov     al, [bx][di]
mov     al, [bp][si]
mov     al, [bp][di]
```

Suppose that bx contains 1000h and si contains 880h. Then the instruction

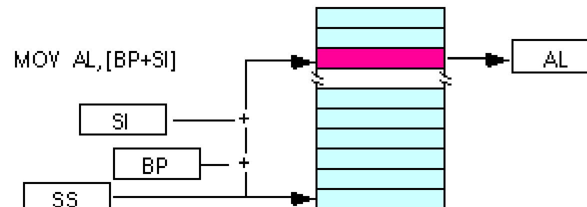
```
mov     al,[bx][si]
```

would load al from location DS:1880h. Likewise, if bp contains 1598h and di contains 1004, `mov ax,[bp+di]` will load the 16 bits in ax from locations SS:259C and SS:259D.

The addressing modes that do not involve bp use the data segment by default. Those that have bp as an operand use the stack segment by default.



You substitute di in the figure above to obtain the [bx+di] addressing mode.



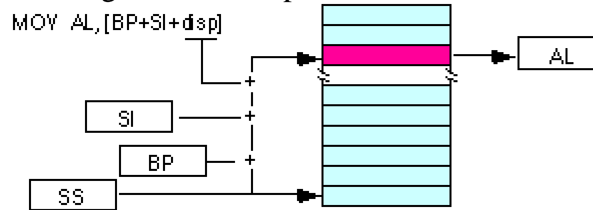
You substitute di in the figure above for the [bp+di] addressing mode.

1.2.5 Based Indexed Plus Displacement Addressing Mode

These addressing modes are a slight modification of the base/indexed addressing modes with the addition of an eight bit or sixteen bit constant. The following are some examples of these addressing modes:

```
mov     al, disp[bx][si]
mov     al, disp[bx+di]
mov     al, [bp+si+disp]
mov     al, [bp][di][disp]
```

You may substitute `di` in the figure above to produce the `[bx+di+disp]` addressing mode.



You may substitute `di` in the figure above to produce the `[bp+di+disp]` addressing mode.

c. Explain stack organisation used in processors. Differentiate between a register stack and a memory stack.

Answer:

Stack is a storage structure that stores information in such a way that the last item stored is the first item retrieved. It is based on the principle of LIFO (Last-in-first-out). The stack in digital computers is a group of memory locations with a register that holds the address of top of element. This register that holds the address of top of element of the stack is called ***Stack Pointer***.

Stack Operations

The two operations of a stack are:

1. **Push:** Inserts an item on top of stack.
2. **Pop:** Deletes an item from top of stack.

Implementation of Stack

In digital computers, stack can be implemented in two ways:

1. Register Stack
2. Memory Stack

Register Stack

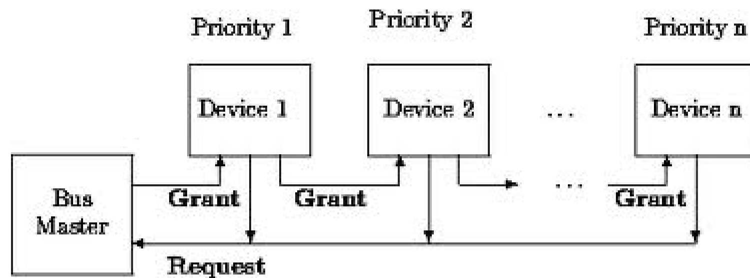
A stack can be organized as a collection of finite number of registers that are used to store temporary information during the execution of a program. The stack pointer (SP) is a register that holds the address of top of element of the stack.

Memory Stack

A stack can be implemented in a random access memory (RAM) attached to a CPU. The implementation of a stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer. The starting memory location of the stack is specified by the processor register as *stack pointer*.

Q.4 a. Explain in brief with the help of a diagram the working of daisy chaining with multiple priority levels and multiple devices in each level.

Answer:



b. Define and explain the following (Any TWO): (2×4)
i. Interrupt
ii. Vectored Interrupt
iii. Interrupt nesting
iv. An exception and its two examples

Answer:

Q.5 a. What are the different kinds of I/O communication techniques? Compare and contrast. In the above techniques, which is the most efficient? Justify your answer.

Answer:

- Devices communicate with the computer via signals sent over wires or through the air.
- Devices connect with the computer via ***ports***, e.g. a serial or parallel port.
- A common set of wires connecting multiple devices is termed a ***bus.***
  - o Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
  - o Figure 13.1 below illustrates three of the four bus types commonly found in a modern PC:
    1. The ***PCI bus*** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)
    2. The ***expansion bus*** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
    3. The ***SCSI bus*** connects a number of SCSI devices to a common SCSI controller.
    4. A ***daisy-chain bus,*** (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.

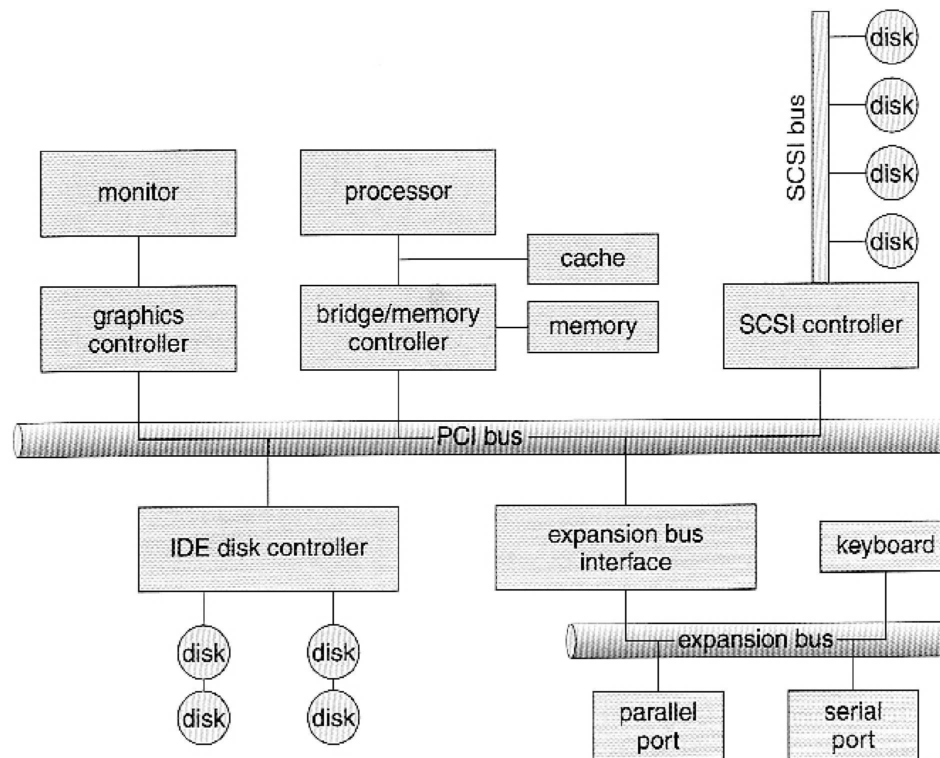


Figure 13.1 A typical PC bus structure.

- One way of communicating with devices is through ***registers*** associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:
  1. The ***data-in register*** is read by the host to get input from the device.
  2. The ***data-out register*** is written by the host to send output.
  3. The ***status register*** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
  4. The ***control register*** has bits written by the host to issue commands or to change settings of the device such as parity checking, word length, or full- versus half-duplex operation.
- Figure 13.2 shows some of the most common I/O port address ranges.

I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Figure 13.2 Device I/O port locations on PCs (partial).

- Another technique for communicating with devices is ***memory-mapped I/O.***
  - In this case a certain portion of the processor's address space is mapped to the device, and communications occur by reading and writing directly to/from those memory areas.
  - Memory-mapped I/O is suitable for devices which must move large quantities of data quickly, such as graphics cards.
  - Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.
  - A potential problem exists with memory-mapped I/O, if a process is allowed to write directly to the address space used by a memory-mapped I/O device.
  - (Note: Memory-mapped I/O is not the same thing as direct memory access, DMA. See section 13.2.3 below.)

Polling

- One simple means of device ***handshaking*** involves polling:
  1. The host repeatedly checks the ***busy bit*** on the device until it becomes clear.
  2. The host writes a byte of data into the data-out register, and sets the ***write bit*** in the command register (in either order.)
  3. The host sets the ***command ready bit*** in the command register to notify the device of the pending command.
  4. When the device controller sees the command-ready bit set, it first sets the busy bit.
  5. Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.

6. The device controller then clears the ***error bit*** in the status register, the command-ready bit, and finally clears the busy bit, signaling the completion of the operation.

- Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer. It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

Interrupts

- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- The CPU has an ***interrupt-request line*** that is sensed after every instruction.
    - A device's controller ***raises*** an interrupt by asserting a signal on the interrupt request line.
    - The CPU then performs a state save, and transfers control to the ***interrupt handler*** routine at a fixed address in memory. (The CPU ***catches*** the interrupt and ***dispatches*** the interrupt handler.)
    - The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a ***return from interrupt*** instruction to return control to the CPU. (The interrupt handler ***clears*** the interrupt by servicing the device.)
        - (Note that the state restored does not need to be the same state as the one that was saved when the interrupt went off. See below for an example involving time-slicing.)
- Figure 13.3 illustrates the interrupt-driven I/O procedure:

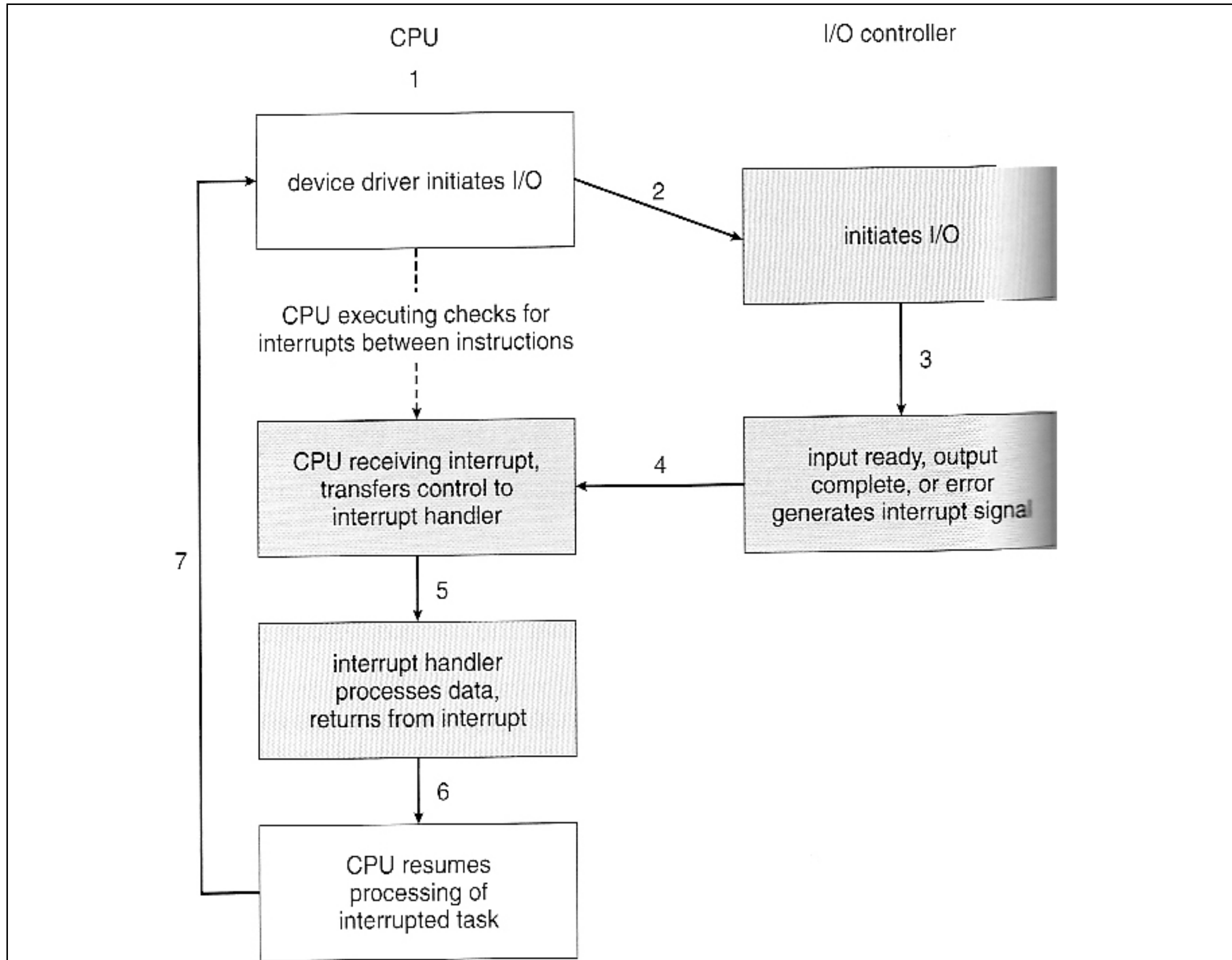


Figure 13.3 Interrupt-driven I/O cycle.

- The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:
  1. The need to defer interrupt handling during critical processing,
  2. The need to determine ***which*** interrupt handler to invoke, without having to poll all devices to see which one needs attention, and
  3. The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.
- These issues are handled in modern computer architectures with ***interrupt-controller*** hardware.
  - Most CPUs now have two interrupt-request lines: One that is ***non-maskable*** for critical error conditions and one that is ***maskable,*** that the CPU can temporarily ignore during critical processing.
  - The interrupt mechanism accepts an ***address,*** which is usually one of a small set of numbers for an offset into a table called the ***interrupt vector.*** This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.

- The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be ***interrupt chained***. Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
- Figure 13.4 shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are non-maskable and reserved for serious hardware and other errors. Maskable interrupts, including normal device I/O interrupts begin at interrupt 32.
- Modern interrupt hardware also supports ***interrupt priority levels***, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Figure 13.4 Intel Pentium processor event-vector table.

- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.
- During operation, devices signal errors or the completion of commands via interrupts.
- Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.

- Time slicing and context switches can also be implemented using the interrupt mechanism.
    - The scheduler sets a hardware timer before transferring control over to a user process.
    - When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
    - The scheduler does a state-restore of a ***different*** process before resetting the timer and issuing the return-from-interrupt instruction.
- A similar example involves the paging system for virtual memory - A page fault causes an interrupt, which in turn issues an I/O request and a context switch as described above, moving the interrupted process into the wait queue and selecting a different process to run. When the I/O request has completed (i.e. when the requested page has been loaded up into physical memory), then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue, (or depending on scheduling algorithms and policies, may go ahead and context switch it back onto the CPU.)
- System calls are implemented via ***software interrupts,*** a.k.a. ***traps.*** When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt. (E.g. 21 hex in DOS.) The system does a state save and then calls on the proper interrupt handler to process the request in kernel mode. Software interrupts generally have low priority, as they are not as urgent as devices with limited buffering space.
- Interrupts are also used to control kernel operations, and to schedule activities for optimal performance. For example, the completion of a disk read operation involves **two** interrupts:
    - A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.
    - A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.
- The Solaris OS uses a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers. This allows for the "simultaneous" handling of multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

Direct Memory Access

- For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.
- Instead this work can be off-loaded to a special processor, known as the ***Direct Memory Access, DMA, Controller.***
- The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.

- A simple DMA controller is a standard component in modern PCs, and many ***bus-mastering*** I/O cards contain their own DMA hardware.
- Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.
- While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.
- DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as ***Direct Virtual Memory Access, DVMA,*** and allows direct data transfer from one memory-mapped device to another without using the main memory chips.
- Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons. (I.e. DMA is a kernel-mode operation.)
- Figure 13.5 below illustrates the DMA process.

Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.

- DOS uses the colon separator to specify a particular device (e.g. C:, LPT:, etc.)
- UNIX uses a ***mount table*** to map filename prefixes (e.g. /usr) to specific mounted devices. Where multiple entries in the mount table match different prefixes of the filename the one that matches the longest prefix is chosen. (e.g. /usr/home instead of /usr where both exist in the mount table and both match the desired file.)
- UNIX uses special ***device files,*** usually located in /dev, to represent and access physical devices directly.
    - Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go.
    - The major number is an index into a table of device drivers, and indicates which device driver handles this device. (E.g. the disk drive handler.)
    - The minor number is a parameter passed to the device driver, and indicates which specific device is to be accessed, out of the many which may be handled by a particular device driver. (e.g. a particular disk drive or partition.)
- A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.
- Figure 13.13 illustrates the steps taken to process a (blocking) read request:

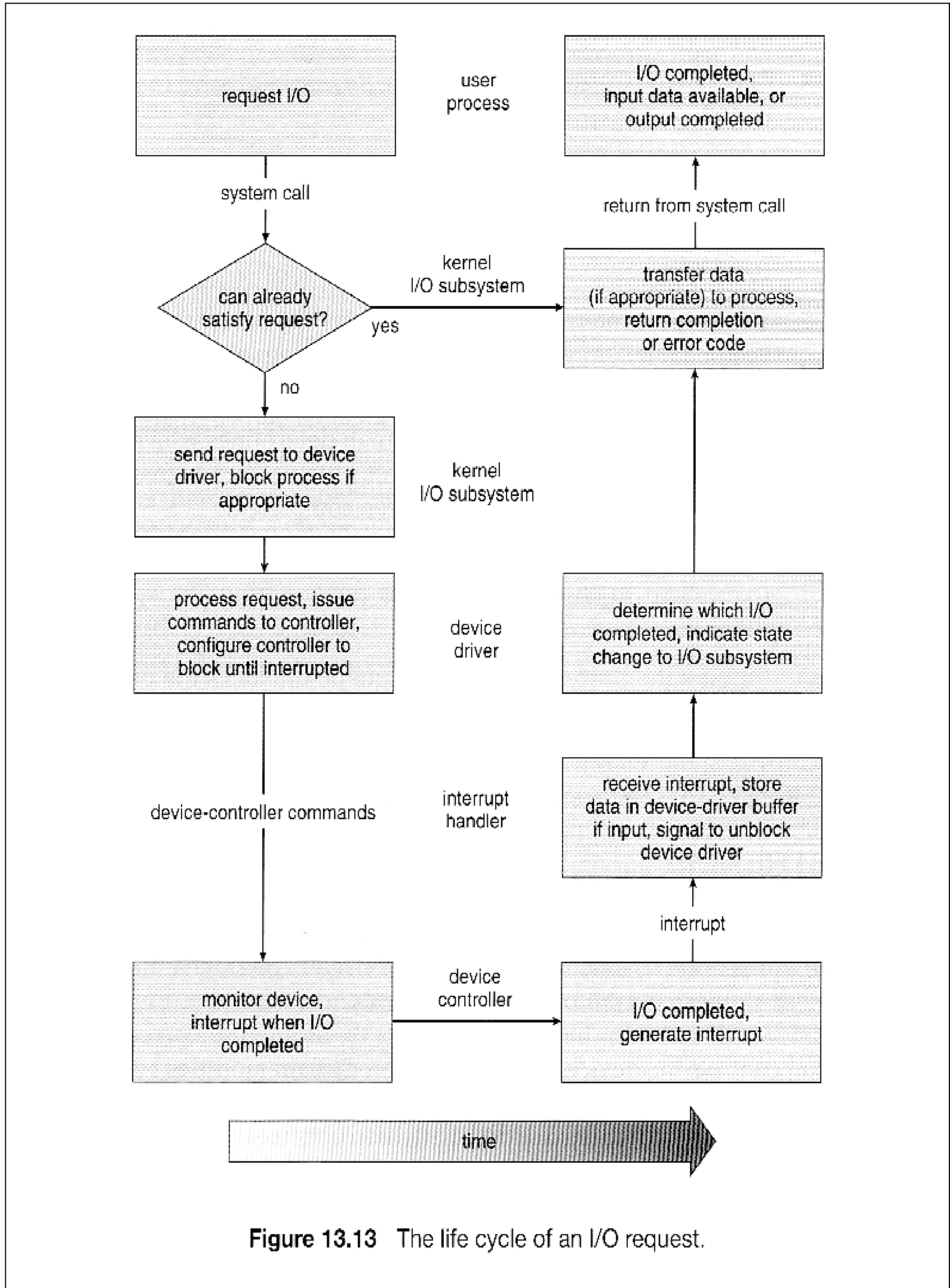


Figure 13.13 The life cycle of an I/O request.

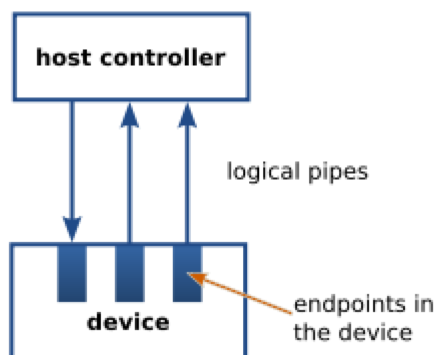
b. Draw the block diagram of universal bus (USB) structure connected to the host computer. Briefly explain all fields of packets that are used for communication between a host and a device connected to an USB port.

Answer:

The design architecture of USB is asymmetrical in its topology, consisting of a host, a multitude of downstream USB ports, and multiple peripheral devices connected in a tiered-star topology. Additional USB hubs may be included in the tiers, allowing branching into a tree structure with up to five tier levels. A USB host may implement multiple host controllers and each host controller may provide one or more USB ports. Up to 127 devices, including hub devices if present, may be connected to a single host controller.[18][19]

USB devices are linked in series through hubs. One hub is known as the root hub which is built into the host controller.

A physical USB device may consist of several logical sub-devices that are referred to as *device functions*. A single device may provide several functions, for example, a webcam (video device function) with a built-in microphone (audio device function). This kind of device is called *composite device*. An alternative for this is *compound device* in which each logical device is assigned a distinctive address by the host and all logical devices are connected to a built-in hub to which the physical USB wire is connected.



USB endpoints actually reside on the connected device: the channels to the host are referred to as pipes

USB device communication is based on *pipes* (logical channels). A pipe is a connection from the host controller to a logical entity, found on a device, and named an *endpoint*. Because pipes correspond 1-to-1 to endpoints, the terms are sometimes used interchangeably. A USB device can have up to 32 endpoints, though USB devices seldom have this many endpoints. An endpoint is built into the USB device by the manufacturer and therefore exists permanently, while a pipe may be opened and closed.

There are two types of pipes: stream and message pipes. A message pipe is bi-directional and is used for *control* transfers. Message pipes are typically used for short, simple commands to the device, and a status response, used, for example, by the bus control pipe number 0. A stream pipe is a uni-directional pipe connected to a uni-directional endpoint that transfers data using an *isochronous*, *interrupt*, or *bulk* transfer:

- *isochronous transfers*: at some guaranteed data rate (often, but not necessarily, as fast as possible) but with possible data loss (e.g., realtime audio or video).
- *interrupt transfers*: devices that need guaranteed quick responses (bounded latency) (e.g., pointing devices and keyboards).
- *bulk transfers*: large sporadic transfers using all remaining available bandwidth, but with no guarantees on bandwidth or latency (e.g., file transfers).

An endpoint of a pipe is addressable with a tuple *(device_address, endpoint_number)* as specified in a TOKEN packet that the host sends when it wants to start a data transfer session. If the direction of the data transfer is from the host to the endpoint, an OUT packet (a specialization of a TOKEN packet) having the desired device address and endpoint number is sent by the host. If the direction of the data transfer is from the device to the host, the host sends an IN packet instead. If the destination endpoint is a uni-directional endpoint whose manufacturer's designated direction does not match the TOKEN packet (e.g., the manufacturer's designated direction is IN while the TOKEN packet is an OUT packet), the TOKEN packet will be ignored. Otherwise, it will be accepted and the data transaction can start. A bi-directional endpoint, on the other hand, accepts both IN and OUT packets.

Two USB standard A receptacles on the front of a computer

Endpoints are grouped into *interfaces* and each interface is associated with a single device function. An exception to this is endpoint zero, which is used for device configuration and which is not associated with any interface. A single device function composed of independently controlled interfaces is called a *composite device*. A composite device only has a single device address because the host only assigns a device address to a function.

When a USB device is first connected to a USB host, the USB device enumeration process is started. The enumeration starts by sending a reset signal to the USB device. The data rate of the USB device is determined during the reset signaling. After reset, the USB device's information is read by the host and the device is assigned a unique 7-bit address. If the device is supported by the host, the device drivers needed for communicating with the device are loaded and the device is set to a configured state. If the USB host is restarted, the enumeration process is repeated for all connected devices.

The host controller directs traffic flow to devices, so no USB device can transfer any data on the bus without an explicit request from the host controller. In USB 2.0, the host controller polls the bus for traffic, usually in a round-robin fashion. The throughput of each USB port is determined by the slower speed of either the USB port or the USB device connected to the port.

High-speed USB 2.0 hubs contain devices called transaction translators that convert between high-speed USB 2.0 buses and full and low speed buses. When a high-speed USB 2.0 hub is plugged into a high-speed USB host or hub, it will operate in high-speed mode. The USB hub will then either use one transaction translator per hub to create a full/low-speed bus that is routed to all full and low speed devices on the hub, or will use one transaction translator per port to create an isolated full/low-speed bus per port on the hub.

Because there are two separate controllers in each USB 3.0 host, USB 3.0 devices will transmit and receive at USB 3.0 data rates regardless of USB 2.0 or earlier devices connected to that host. Operating data rates for them will be set in the legacy manner.

Q.6 a. Give the organization of a 2M X 32 memory module using 512K X 8 memory chips. Explain the organization.

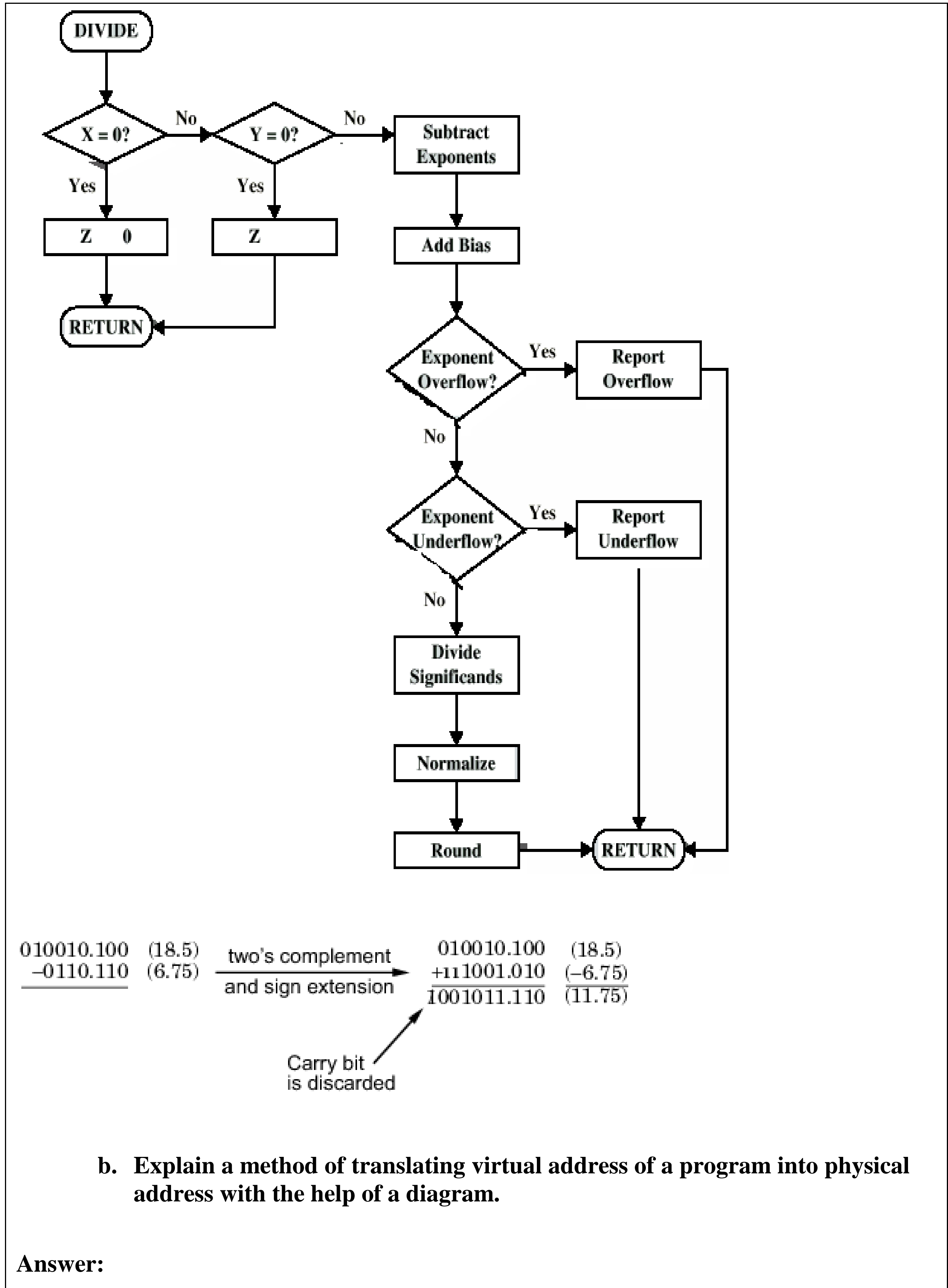
Answer: Refer Page No. 306 ad Fig 5.10 from Text Book

b. Explain the following mapping procedure:
(i) Direct mapping
(ii) Associative mapping (4+4)

Answer: Refer page nos 317 & 318 from Text Book

Q.7 a. Draw a flow chart to explain how addition and subtraction of two fixed point numbers can be done. Give an example to explain it.

Answer:



b. Explain a method of translating virtual address of a program into physical address with the help of a diagram.

Answer:

Virtual memory is easiest to comprehend if one thinks in terms of the VAS, and not the physical memory of the machine nor the size of its page file. Byte values in the VAS come *only* from byte values in a file. The OS manages the mapping between the VAS and the files that hold its values.

Each time an application is run on an operating system (OS), the OS creates a new process and a new VAS for this process.

Physical memory comes in various flavors: on-chip cache, off-chip cache, and system memory. As far as the process is concerned, system memory is just another level of cache used by the OS. System memory has a lot to do with performance, but nothing to do with the architecture of a process. The process architecture is based on the VAS. Physical memory is used by the OS to map values from file bytes to VAS addresses: process memory is VAS memory, not physical memory.

Example

In the following description, the terminology used will be particular to the Windows NT OS, but the concepts are applicable to other virtual memory operating systems.

When you run a new application on a 32-bit OS, the process has a 4 GiB VAS: each one of the memory addresses (from 0 to $2^{32}-1$) in that space can have a single byte as value. Initially, none of them have values ('-' represents no value). Using or setting values in such a VAS would cause a memory exception.

```
                0                                               4GB
VAS             |----------------------------------------------|
```

Then the application's EXE file is mapped into the VAS. Addresses in the process VAS are mapped to bytes in the exe file. The OS manages the mapping:

```
                0                                               4GB
```

```
VAS          |---vvvvvvv------------------------------------|
mapping          |-----|
file bytes       app.exe
```

The v's are values from bytes in the mapped file. Then, required DLL files are mapped (this includes custom libraries as well as system ones such as `kernel32.dll` and `user32.dll`):

```
             0                                             4GB
VAS          |---vvvvvvv----vvvvvv---vvvv-------------------|
mapping          |||||||    ||||||   ||||
file bytes       app.exe    kernel   user
```

The process then starts executing bytes in the exe file. However, the only way the process can use or set '-' values in its VAS is to ask the OS to map them to bytes from a file. A common way to use VAS memory in this way is to map it to the page file. The page file is a single file, but multiple distinct sets of contiguous bytes can be mapped into a VAS:

```
             0                                             4GB
VAS          |---vvvvvvv----vvvvvv---vvvv----vv---v----vvv--|
mapping          |||||||    ||||||   ||||    ||   |    |||
file bytes       app.exe    kernel   user   system_page_file
```

And different parts of the page file can map into the VAS of different processes:

```
             0                                             4GB
VAS 1        |---vvvv-------vvvvvv---vvvv----vv---v----vvv--|
mapping          ||||       ||||||   ||||    ||   |    |||
file bytes       app1 app2  kernel   user   system_page_file
mapping               ||||  ||||||   ||||       ||   |
VAS 2        |--------vvvv--vvvvvv---vvvv-------vv---v------|
```

On a 32-bit Microsoft Windows installation, by default, only 2 GiB are made available to processes for their own use. The other 2GB are used by the operating system. On later 32-bit editions of Microsoft Windows it is possible to extend the user-mode virtual address space to 3 GiB while only 1 GiB is left for kernel-mode virtual address space by marking the programs as IMAGE_FILE_LARGE_ADDRESS_AWARE and enabling the /3GB switch in the boot.ini file.

On 64-bit Microsoft Windows, processes running 32-bit executables that were linked with the /LARGEADDRESSAWARE:YES option have access to 4 GiB of virtual address space; without that option they are limited to 2GB. By default, 64-bit processes have 8TB of user-mode virtual address space; Linking with /LARGEADDRESSAWARE:NO artificially limits the user-mode virtual address space to 2 GB.

Allocating memory via system calls such as C's malloc implicitly maps bytes of the page file into the VAS. However, a process can also *explicitly* map file bytes.

Q.8 a. Perform signed multiplication of -3 and 7 using booth multiplication algorithm. Represent the numbers in 5 bits including sign bit. Give booth multiplier recording table that is used in the multiplication.

Answer:

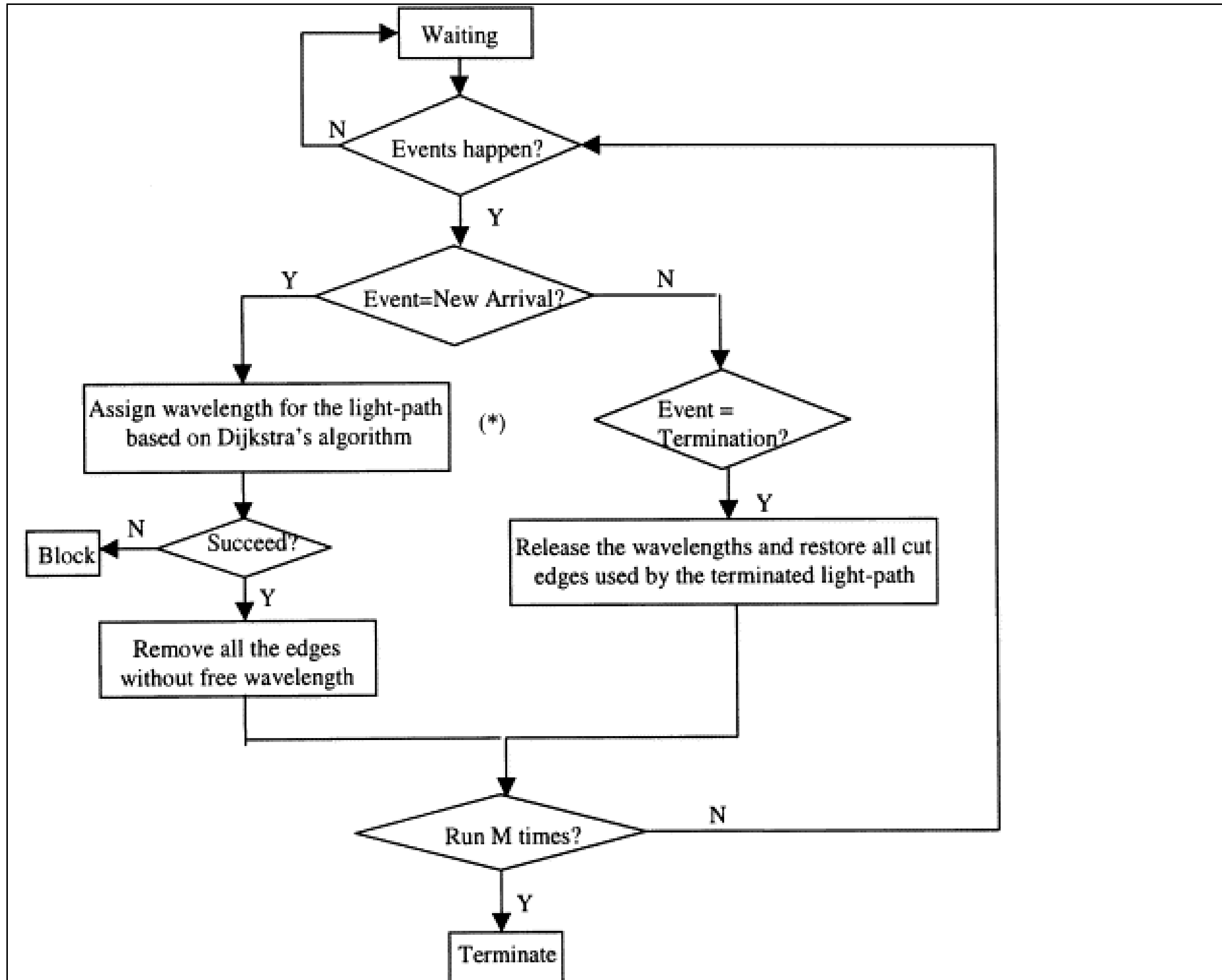
```
        A =  3    0 0 1 1              -A = 1 1 0 1
        B = -7 =  1 0 0 1
                 ---------
                  0 0 1 1       1      PP 1
          1 1 1 0 1 0          -2      PP 2
          --------------------
   -21 =  1 1 1 0 1 0 1 1              Final product


      A = -3   1 1 0 1             -A = 0 0 1 1
      B =  7 = 0 1 1 1
               ---------
               0 0 1 1     -1      PP 1
         0 0 1 1 0         -2      PP 2
       1 1 0 1              1      Extra partial product
       --------------------
 -21 = 1 1 1 0 1 0 1 1             Final product
```

Table 15

b. Explain restoring division algorithm with a diagram.

Answer:



Q.9 a. Hard-wired control unit is faster than micro programmed control unit. Justify this statement.

Answer: Hardwired vs. Micro-programmed Computers

It should be mentioned that most computers today are micro-programmed. The reason is basically one of flexibility. Once the control unit of a hard-wired computer is designed and built, it is virtually impossible to alter its architecture and instruction set. In the case of a micro-programmed computer, however, we can change the computer's instruction set simply by altering the microprogram stored in its control memory. In fact, taking our basic computer as an example, we notice that its four-bit op-code permits up to 16 instructions. Therefore, we could add seven more instructions to the instruction set by simply expanding its microprogram. To do this with the hard-wired version of our computer would require a complete redesign of the controller circuit hardware.

Another advantage to using micro-programmed control is the fact that the task of designing the computer in the first place is simplified. The process of specifying the architecture and instruction set is now one of software (micro-programming) as opposed to hardware design. Nevertheless, for certain applications hard-wired computers are still used. If speed is a

consideration, hard-wiring may be required since it is faster to have the hardware issue the required control signals than to have a "program" do it.

b. Explain the variety of techniques available for sequencing of microinstructions based on the format of the address information in the microinstruction.

Answer:

The two main variations of microprogramming are the horizontal and vertical methods. In the previous section, we already saw some distinction between horizontal and vertical next-state organizations. In *horizontal microprogramming*, there is one ROM output for each control point in the data-path. *Vertical microprogramming* is based on the observation that only a subset of these signals is ever asserted in a given state. Thus, the control outputs can be stored in the ROM in an encoded form, effectively reducing the width of the ROM word at the expense of some external decoding logic.

Encoding the control signals may limit the data-path operations that can take place in parallel. If this is the case, we may need multiple ROM words to encode the same data-path operations that could be performed in a single horizontal ROM word.

For example, consider a microprogrammed control for a machine with four general-purpose accumulators. Most computer instruction formats limit the destination of an operation to a single register. Realizing this, you may choose to encode the destination of a register transfer operation in 2 bits rather than 4. The destination register select line is driven by logic that decodes these 2 bits from the control. Thus, at any given time, only one of the registers is selected as a destination.

The art of engineering a microprogrammed control unit is to strike the correct balance between the parallelism of the horizontal approach and the ROM economy of a vertical encoding. For example, the encoded register enable lines eliminate the possibility of any state loading two registers at the same time, even if this was supported by the processor data-path. If a machine instruction must load two registers, it will require multiple control states（and ROM words）to implement its execution.

We begin our study with the horizontal approach to microprogramming. We will see that the instruction set and the data-path typically do not support the full parallelism implied by horizontal control, so we will examine methods of encoding the ROM word to reduce its size.

Horizontal Microprogramming

The horizontal next-state organization of Figure 12.22 offers the core of a horizontal microprogrammed controller. An extremely horizontal control word format would have 1 bit for each microoperation in the data-path. Let's develop such a format for the simple CPU's control.

Our sample processor supports 14 register transfer operations. We further decompose these into 22 discrete microoperations (ordered by destination):

PC --> ABUS
IR --> ABUS
MBR --> ABUS
RBUS --> AC
AC --> ALU *A*
MBUS --> ALU B
ALU ADD
ALU PASS B
MAR --> Address Bus
MBR --> Data Bus
ABUS --> IR
ABUS --> MAR
Data Bus --> MBR
RBUS --> MBR
MBR --> MBUS
0 --> PC
PC + 1 --> PC
ABUS --> PC
Read/$\overline{\text{Write}}$
Request
AC --> RBUS
ALU Result --> RBUS

A very long ROM word for a four-way branch sequencer would have a and b multiplexer bits, four 4-bit next states, and 22 microoperation bits. This yields a total ROM word length of 40 bits, as shown in Figure 12.23.

α mux
β mux
Next States
A_0 A_1 A_2 A_3
PC → ABUS
IR → ABUS
MBR → ABUS
RBUS → AC
AC → ALU A
MBUS → ALU B
ALU ADD
ALU PASS B
MAR → Address Bus
MBR → Data Bus
ABUS → IR
ABUS → MAR
Data Bus → MBR
RBUS → MBR
MBR → MBUS
0 → PC
PC + 1 → PC
ABUS → PC
Read/$\overline{\text{Write}}$
Request
AC → RBUS
ALU Result → RBUS

Figure 12.23 Extremely wide horizontal microcode format for four-way branch sequencer.

Current State (Address)	α mux β mux	Next State A_0	A_1	A_2	A_3	PC → ABUS	IR → ABUS	MBR → ABUS	RBUS → AC	AC → ALU A	MBUS → ALU B	ALU ADD	ALU PASS B	MAR → Address Bus	MBR → Data Bus	ABUS → IR	ABUS → MAR	Data Bus → MBR	RBUS → MBR	MBR → MBUS	0 → PC	PC + 1 → PC	ABUS → PC	Read/$\overline{Write}$	Request	AC → RBUS	ALU Result → RBUS
RES (0000)	00	0001	0001	0001	0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
IF_0 (0001)	00	0010	0010	0010	0010	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0
IF_1 (0010)	00	0010	0010	0011	0011	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
IF_2 (0011)	00	0100	0100	0011	0011	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0
IF_3 (0100)	00	0100	0100	0101	0101	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
OD (0101)	11	0110	1001	1011	1110	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LD_0 (0110)	00	0111	0111	0111	0111	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
LD_1 (0111)	00	1000	1000	0111	0111	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0
LD_2 (1000)	00	0001	0001	0001	0001	0	0	0	1	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	1
ST_0 (1001)	00	1010	1010	1010	1010	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0
ST_1 (1010)	00	0001	0001	1010	1010	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	1	0	0
AD_0 (1011)	00	1100	1100	1100	1100	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
AD_1 (1100)	00	1101	1101	1100	1100	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	0	0
AD_2 (1101)	00	0001	0001	0001	0001	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
BR_0 (1110)	01	0001	1111	0001	1111	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
BR_1 (1111)	00	0001	0001	0001	0001	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0

Figure 12.24 ROM contents for the Moore processor controller .

Figure 12.24 gives the ROM contents for the Moore controller of Figure 12.1（the branch sequencer of Figure 12.22 implements a Moore machine）. The a multiplexer inputs are Sel0 = Wait, Sel1 = IR<15> and the b inputs are Sel0 = AC<15>, Sel1 = IR<14>. We assume the next-state register is reset directly.

The multiplexers on the next state work just as in Figure 12.22. For example, consider state IF1. We stay in this state if Wait is unasserted. If Wait is asserted, we advance to state IF2. The a and b mux controls are set to examine Wait and AC<15>, respectively. Thus, the 0X next-state bits（*A*0, *A*1）are set to the encoding for IF1, 0010. Similarly, the 1X next states（*A*2, *A*3）are set to the encoding for IF2, 0011.

Reducing ROM Word Width Through Encoding The horizontal approach offers the most flexibility by providing access to all of the data-path control points at the same time. The disadvantage is the width of the ROM word, which can exceed a few hundred bits in complex controllers.

A good way to reduce the ROM size is by encoding its output. This need not lead to an inherent loss of parallelism. After all, certain control combinations may not make sense logically（for example, 0 --> PC and PC + 1 --> PC are logically exclusive）or might be ruled out by the data-path busing strategy（for example, PC --> ABUS and IR --> ABUS cannot take place simultaneously）.

Furthermore, the ROM contents of Figure 12.24 are very sparse. In any given state, very few of the control signals are ever asserted. This means we can group the control signals into mutually exclusive sets to encode them. We decode them outside the ROM with additional hardware.

For example, the three PC microoperations, 0 --> PC, PC + 1 --> PC, and ABUS --> PC, are never asserted in the same state. For the cost of an external 2-to-4 decoder, a ROM bit can be saved by encoding the signals as follows:

00	**No PC control**
01	0 --> PC
10	PC + 1 --> PC
11	ABUS --> PC

There are many other plausible encoding strategies for this controller. MAR --> Address Bus and Request are always asserted together, as are RBUS --> AC, MBUS --> ALU B, MBR --> MBUS, and ALU --> RBUS. If we have designed the ALU to pass its *A* input selectively, we can combine AC --> ALU *A* in state LD2 with this list of signals. As another example, we can combine MBR --> ABUS and ABUS --> IR. Taken together, these encodings save six ROM bits.

We can save additional ROM bits by finding unrelated signals that are never asserted at the same time. These are good candidates for encoding. For example, we can combine PC --> ABUS, IR --> ABUS, and Data Bus --> MBR, encoding them into 2 bits. Applying all of these encodings at the same time yields the encoded control unit in Figure 12.25. The direct ROM outputs have been reduced from 22 to 15.

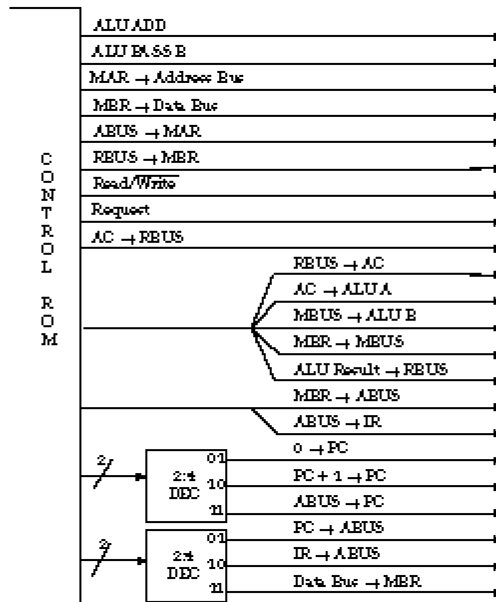


Figure 12.25 Encoded ROM outputs.

As more control signals are placed in the ROM in an encoded form, we move from a very horizontal format to one that is ever more vertical. We present a systematic approach to vertical microprogramming next.

Vertical Microprogramming

Vertical microprogramming makes more use of ROM encoding to reduce the length of the control word. To achieve this goal, we commonly use multiple microword formats. For example, many states require no conditional next-state branch; they simply advance to the

next state in sequence. Rather than having every microword contain a next state and a list of microoperations, we can shorten the ROM word by separating these two functions into individual microword formats: one for conditional "branch jumps" and another for register transfer operations/microoperations.

Shortening the ROM word does not come free. We may need several ROM words in a sequence to perform the same operations as a single horizontal microword. The combination of extra levels of decoding, multiple ROM accesses to execute a sequence of control operations, and sacrifice of the potential parallelism of the vertical approach leads to slower implementations. The basic machine cycle time increases, and the number of machine cycles to execute an instruction also increases.

Despite this inefficiency, designers prefer vertical microcode because it is much like coding in assembly language. So the trade-off between vertical and horizontal microcode is really a matter of ease of implementation versus performance.

Vertical Microcode Format for the Simple CPU Let's develop a simple vertical microcode format for our simple processor. We will introduce just two formats: a *branch jump* format and a *register transfer/operation* -format.

In a branch jump microword, we include a field to select a signal to be tested (Wait, AC<15>, IR<15>, IR<14>) and the value it should be tested against (0 or 1). If the signal matches the specified value, the rest of the microword contains the address of the next ROM word to be fetched. The condition selection field can be 2 bits in length; the condition comparison field can be 1 bit wide.

The register transfer/operation microword contains three fields: a register source, a register destination, and an operation field for instructing functional units like the ALU what to do. To start, let's arrange the microoperations according to these categories:

Sources:
PC --> ABUS
IR --> ABUS
MBR --> MBUS
AC --> ALU A
MAR --> Mem Address Bus
MBR --> Mem Data Bus
MBR --> MBUS
AC --> RBUS
ALU Result --> RBUS
Destinations:
RBUS --> AC
MBUS --> ALU B
MBUS --> IR
ABUS --> MAR
Mem Data Bus --> MBR

RBUS --> MBR
ABUS --> PC
Operations:
ALU ADD
ALU PASS B
0 --> PC
PC + 1 --> PC
Read (Read, Request)
Write ($\overline{\text{Write}}$, Request)
We can encode the nine sources in a 4-bit field, the seven destinations in 3, and the six operations also in 3 (we have combined Read/$\overline{\text{Write}}$ and Request in the operation format).

It would certainly be convenient to encode all the fields in the same number of bits. At the moment, we have several more sources than destinations. A close examination of the data-path of Figure 11.26 indicates that we can do better at encoding the destinations. We can assume that the AC is hardwired to the ALU *A* input, just as the MBUS is wired to the ALU *B* input. Also, the MBR is the only source on the MBUS, so we can eliminate the microoperation MBR --> MBUS. This gives us seven sources and six destinations, easily encoded in 3 bits each.

There is still one hitch. On writes to memory, such as during a store, the MAR must drive the memory address lines and the MBR must drive the data lines. But as listed above, these two microoperations are now mutually exclusive.
Fortunately, there is a reasonable solution. We can move the operation MBR --> Mem Data Bus from the sources to the destinations, simply by thinking of the memory as a destination rather than the MBR as a source. The encoding of the two formats can fit in a very compact 10 bits, as shown in Figure 12.26.

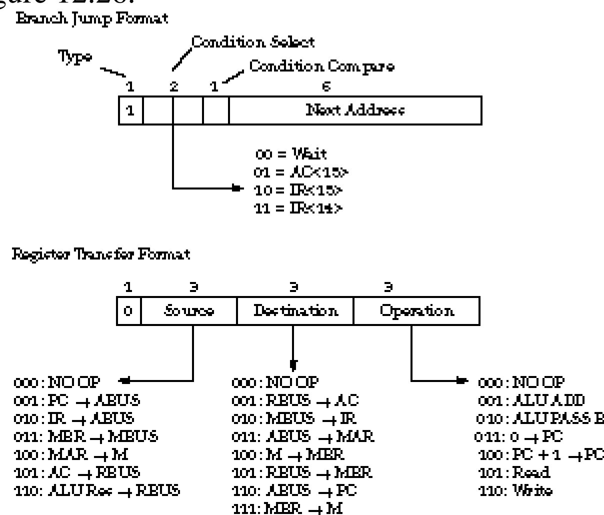
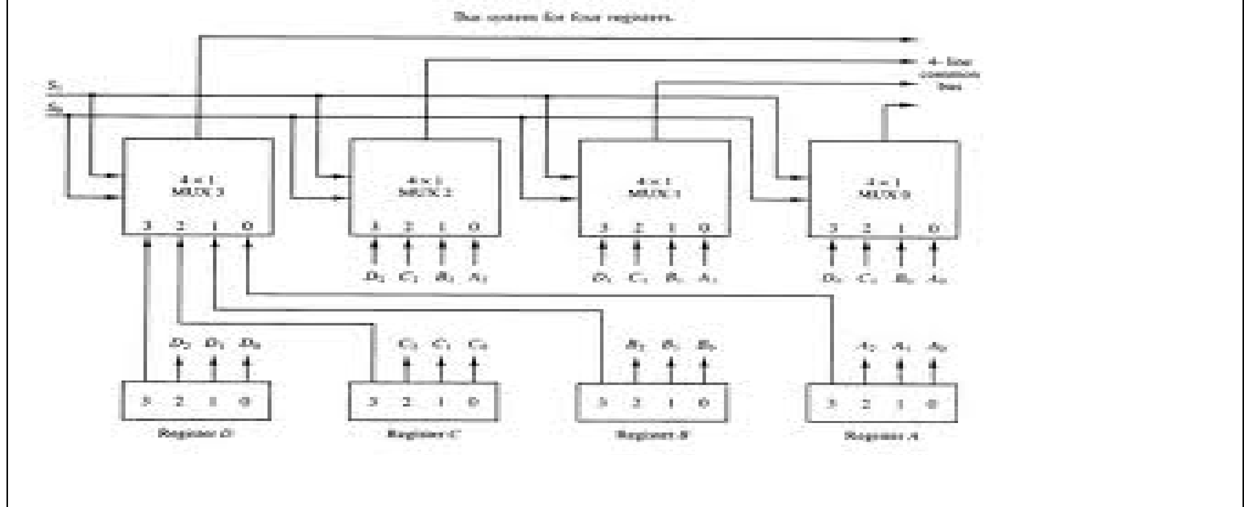


Figure 12.26 Vertical microcode format

c. Explain time-shared common bus organization.

Answer:



TEXT BOOK

I. Computer Organization, Carl Hamacher, Zvonko Vranesic, Safwat Zaky, 5th Edition, TMH, 2002