UNIT 1
INTRODUCTION

FUNCTIONAL UNITS OF A COMPUTER SYSTEM

Digital computer systems consist of three distinct units. These units are as follows: Input unit Central Processing unit Output unit these units are interconnected by electrical cables to permit communication between them. This allows the computer to function as a system. Input Unit A computer must receive both data and program statements to function properly and be able to solve problems. The method of feeding data and programs to a computer is accomplished by an input device. Computer input devices read data from a source, such as magnetic disks, and translate that data into electronic impulses for transfer into the CPU. Some typical input devices are a keyboard, a mouse, or a scanner. Central Processing Unit The brain of a computer system is the central processing unit (CPU). The CPU processes data transferred to it from one of the various input devices. It then transfers either an intermediate or final result of the CPU to one or more output devices. A central control section and work areas are required to perform calculations or manipulate data. The CPU is the computing center of the system. It consists of a control section, an arithmetic-logic section, and an internal storage section (main memory). Each section within the CPU serves a specific function and has a particular relationship with the other sections within the CPU.

CONTROL SECTION

The control section directs the flow of traffic (operations) and data. It also maintains order within the computer. The control section selects one program statement at a time from the program storage area, interprets the statement, and sends the appropriate electronic impulses to the arithmetic-logic and storage sections so they can carry out the instructions. The control section does not perform actual processing operations on the data. The control section instructs the input device on when to start and stop transferring data to the input storage area. It also tells the output device when to start and stop receiving data from the output storage area.

ARITHMETIC-LOGIC SECTION.—

The arithmetic-logic section performs arithmetic operations, such as addition, subtraction, multiplication, and division. Through internal logic capability, it tests various conditions encountered during processing and takes action based on the result. At no time does processing take place in the storage section. Data maybe transferred back and forth between these two sections several times before processing is completed.

Computer architecture topics

Sub-definitions

Some practitioners of computer architecture at companies such as Intel and AMD use more fine distinctions:

- Macroarchitecture - architectural layers that are more abstract than microarchitecture, e.g. ISA

- ISA (Instruction Set Architecture) - as defined above
- Assembly ISA - a smart assembler may convert an abstract assembly language common to a group of machines into slightly different machine language for different implementations
- Programmer Visible Macroarchitecture - higher level language tools such as compilers may define a consistent interface or contract to programmers using them, abstracting differences between underlying ISA, UISA, and micro architectures. E.g. the C, C++, or Java standards define different Programmer Visible Macro architecture - although in practice the C micro architecture for a particular computer includes

- UISA (Microcode Instruction Set Architecture) - a family of machines with different hardware level micro architectures may share a common microcode architecture, and hence a UISA.

- Pin Architecture - the set of functions that a microprocessor is expected to provide, from the point of view of a hardware platform. E.g. the x86 A20M, FERR/IGNNE or FLUSH pins, and the messages that the processor is expected to emit after completing a cache invalidation so that external caches can be invalidated. Pin architecture functions are more flexible than ISA functions - external hardware can adapt to changing encodings, or changing from a pin to a message - but the functions are expected to be provided in successive implementations even if the manner of encoding them changes.

Design goals

The exact form of a computer system depends on the constraints and goals for which it was optimized. Computer architectures usually trade off standards, cost, memory capacity, latency and throughput. Sometimes other considerations, such as features, size, weight, reliability, expandability and power consumption are factors as well.

The most common scheme carefully chooses the bottleneck that most reduces the computer's speed. Ideally, the cost is allocated proportionally to assure that the data rate is nearly the same for all parts of the computer, with the most costly part being the slowest. This is how skillful commercial integrators optimize personal computers.

Performance

Computer performance is often described in terms of clock speed (usually in MHz or GHz). This refers to the cycles per second of the main clock of the CPU. However, this metric is somewhat misleading, as a machine with a higher clock rate may not necessarily have higher performance. As a result manufacturers have moved away from clock speed as a measure of performance.

Computer performance can also be measured with the amount of cache a processor has. If the speed, MHz or GHz, were to be a car then the cache is like the gas tank. No matter how fast the car goes, it will still need to get gas. The higher the speed, and the greater the cache, the faster a processor runs.

Modern CPUs can execute multiple instructions per clock cycle, which dramatically speeds up a program. Other factors influence speed, such as the mix of functional units, bus speeds, available memory, and the type and order of instructions in the programs being run.

There are two main types of speed, latency and throughput. Latency is the time between the start of a process and its completion. Throughput is the amount of work done per unit time. Interrupt latency is the guaranteed maximum response time of the system to an electronic event (*e.g.* when the disk drive finishes moving some data). Performance is affected by a very wide range of design choices — for example, pipelining a processor usually makes latency worse (slower) but makes throughput better. Computers that control machinery usually need low interrupt latencies. These computers operate in a real-time environment and fail if an operation is not completed in a specified amount of time. For example, computer-controlled anti-lock brakes must begin braking almost immediately after they have been instructed to brake.

The performance of a computer can be measured using other metrics, depending upon its application domain. A system may be CPU bound (as in numerical calculation), I/O bound (as in a webserving application) or memory bound (as in video editing). Power consumption has become important in servers and portable devices like laptops.

Benchmarking tries to take all these factors into account by measuring the time a computer takes to run through a series of test programs. Although benchmarking shows strengths, it may not help one to choose a computer. Often the measured machines split on different measures. For example, one system might handle scientific applications quickly, while another might play popular video games more smoothly. Furthermore, designers have been known to add special features to their products, whether in hardware or software, which permit a specific benchmark to execute quickly but which do not offer similar advantages to other, more general tasks.

A Functional Unit is defined as a collection of computer systems and network infrastructure components which, when abstracted, can be more easily and obviously linked to the goals and objectives of the enterprise, ultimately supporting the success of the enterprise's mission.

From a technological perspective, a Functional Unit is an entity that consists of computer systems and network infrastructure components that deliver critical information assets,1 through network-based services, to constituencies that are authenticated to that Functional Unit.

Central processing unit (CPU) —

The part of the computer that executes program instructions is known as the processor or central processing unit (CPU). In a microcomputer, the CPU is on a single electronic component, the microprocessor chip, within the system unit or system cabinet. The system unit also includes circuit boards, memory chips, ports and other components. A microcomputer system cabinet will also house disk drives, hard disks, etc., but these are considered separate from the CPU. This is principal part of any digital computer system, generally composed of control unit, and arithmetic-logic unit the 'heart" of the computer. It constitutes the physical heart of the entire computer system; to it is linked various peripheral equipment, including input/output devices and auxiliary storage units

o **Control Unit** is the part of a CPU or other device that directs its operation. The control unit tells the rest of the computer system how to carry out a program's instructions. It directs the movement of electronic signals between memory—which temporarily holds data, instructions and processed information—and the ALU. It also directs these control signals between the CPU and input/output devices. The control unit is the circuitry that controls the flow of information through the processor, and coordinates the activities of the other units within it. In a way, it is the "brain", as it controls what happens inside the processor, which in turn controls the rest of the PC.

o **Arithmetic-Logic** Unit usually called the ALU is a digital circuit that performs two types of operations— arithmetic and logical. Arithmetic operations are the fundamental mathematical operations consisting of addition, subtraction, multiplication and division. Logical operations consist of comparisons. That is, two pieces of data are compared to see whether one is equal to, less than, or greater than the other. The ALU is a fundamental building block of the central processing unit of a computer. Memory — Memory enables a computer to store, at least temporarily, data and programs. Memory—also known as the primary storage or main memory—is a part of the microcomputer that holds data for processing, instructions for processing the data (the program) and information (processed data). Part of the contents of the memory is held only temporarily, that is, it is stored only as long as the microcomputer is turned on. When you turn the machine off, the contents are lost. The capacity of the memory to hold data and program instructions varies in different computers. The original IBM PC could hold approximately 6,40,000 characters of data or instructions only. But modern microcomputers can hold millions, even billions of characters in their memory.

Input device :

An input device is usually a keyboard or mouse, the input device is the conduit through which data and instructions enter a computer. A personal computer would be useless if you could not interact with it because the machine could not receive instructions or deliver the results of its work. Input devices accept data and instructions from the user or from another computer system (such as a computer on the Internet). Output devices return processed data to the user or to another computer system.

The most common input device is the keyboard, which accepts letters, numbers, and commands from the user. Another important type of input device is the mouse, which lets you select options from on-screen menus. You use a mouse by moving it across a flat surface and pressing its buttons.

A variety of other input devices work with personal computers, too: The trackball and touchpad are variations of the mouse and enable you to draw or point on the screen.
The joystick is a swiveling lever mounted on a stationary base that is well suited for playing video games.

Basic Operational Concepts of a Computer

- Most computer operations are executed in the ALU (arithmetic and logic unit) of a processor.
- Example: to add two numbers that are both located in memory.
  - Each number is brought into the processor, and the actual addition is carried out by the ALU.
  - The sum then may be stored in memory or retained in the processor for immediate use.

Registers

- When operands are brought into the processor, they are stored in high-speed storage elements (registers).
- A register can store one piece of data (8-bit registers, 16-bit registers, 32-bit registers, 64-bit registers, etc…)
- Access times to registers are faster than access times to the fastest cache unit in the memory hierarchy.

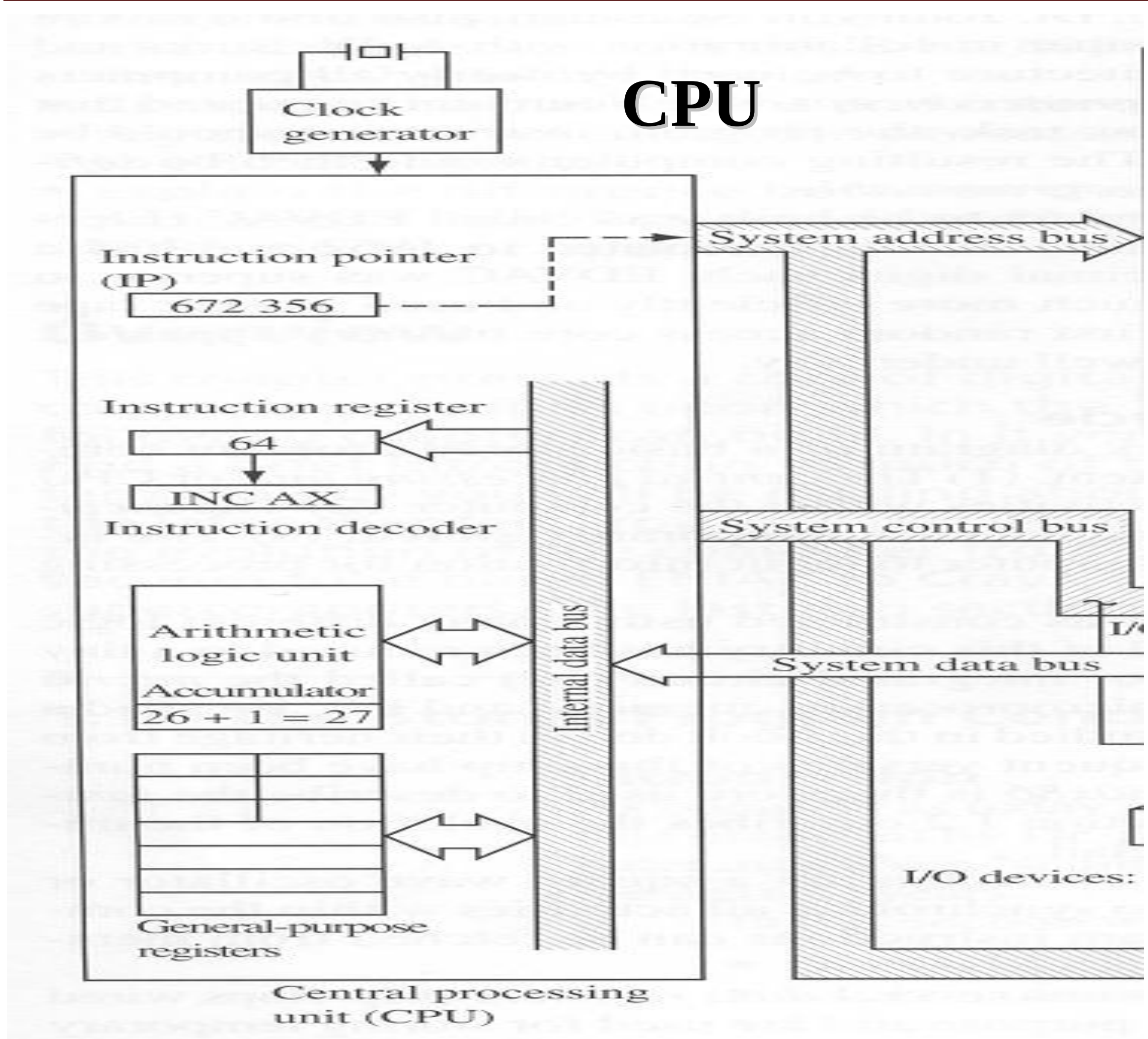
Instructions

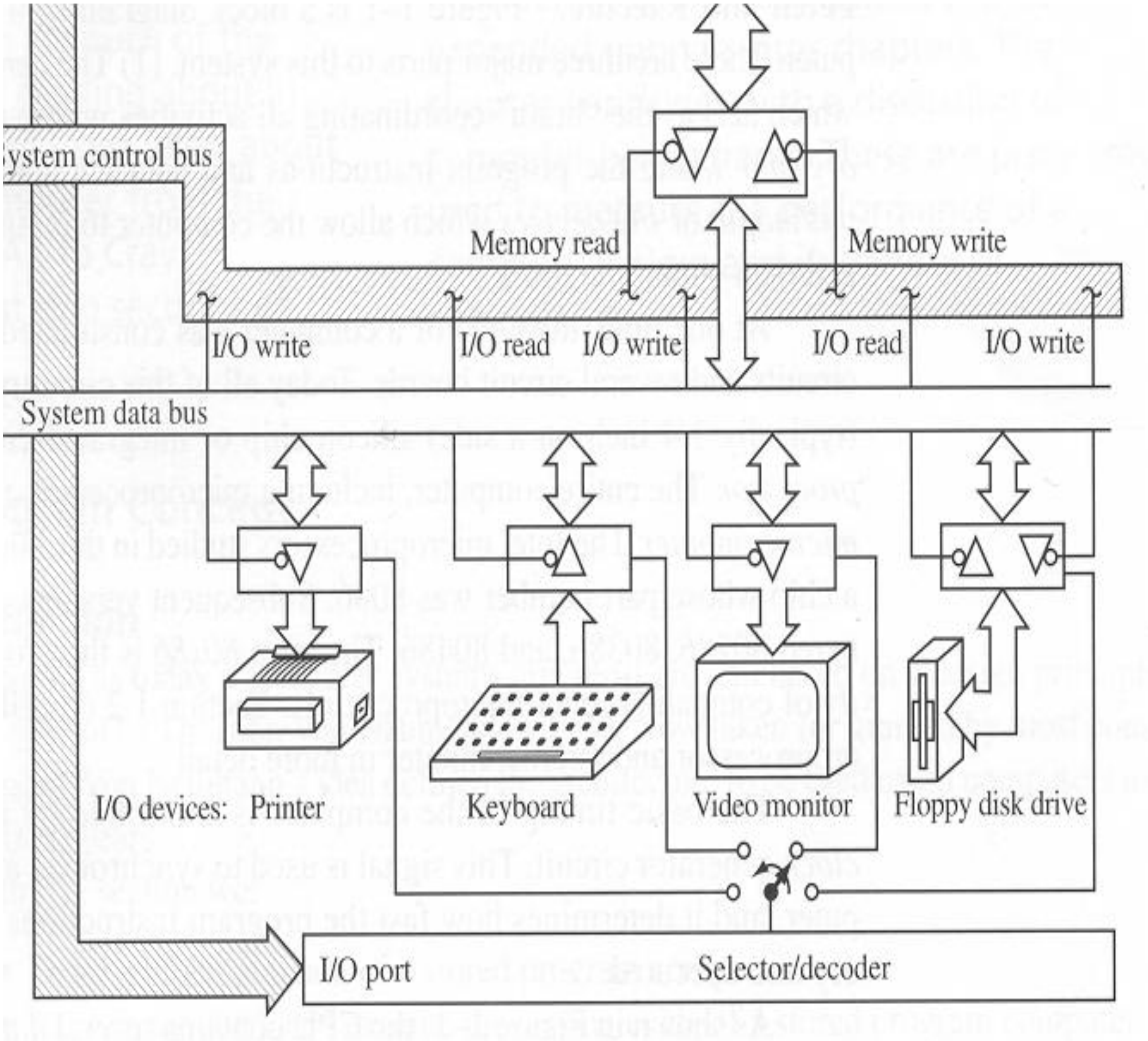
- **Instructions for a processor are defined in the ISA (Instruction Set Architecture) – Level 2**
- **Typical instructions include:**
  - **Mov BX, LocA**
    - **Fetch the instruction**
    - **Fetch the contents of memory location LocA**
    - **Store the contents in general purpose register BX**
  - **Add AX,BX**
    - **Fetch the instruction**
    - **Add the contents of registers BX and AX**
    - **Place the sum in register AX**

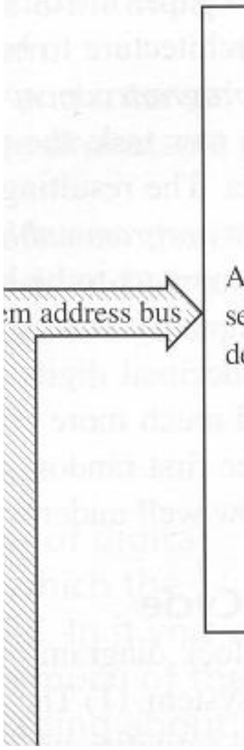
How are instructions sent between memory and the processor

- **The program counter (PC) or instruction pointer (IP) contains the memory address of the next instruction to be fetched and executed.**
- **Send the address of the memory location to be accessed to the memory unit and issue the appropriate control signals (*memory read*).**
- **The instruction register (IR) holds the instruction that is currently being executed.**
- **Timing is crucial and is handled by the control unit within the processor.**

CPU


Clock generator
Instruction pointer (IP)
672 356
System address bus
Instruction register
64
INC AX
Instruction decoder
System control bus
I/
Arithmetic logic unit
Accumulator
26 + 1 = 27
Internal data bus
System data bus
General-purpose registers
I/O devices:
Central processing unit (CPU)


ystem control bus
Memory read
Memory write
I/O write
I/O read
I/O write
I/O read
I/O write
System data bus
I/O devices: Printer
Keyboard
Video monitor
Floppy disk drive
I/O port
Selector/decoder


em address bus
A
se
de

Single BUS STRUCTURES:

Bus structure and multiple bus structures are types of bus or computing. A bus is basically a subsystem which transfers data between the components of a Computer components either within a computer or between two computers. It connects peripheral devices at the same time.

- A multiple Bus Structure has multiple inter connected service integration buses and for each bus the other buses are its foreign buses. A Single bus structure is very simple and consists of a single server.

- A bus can not span multiple cells. And each cell can have more than one buses.
- Published messages are printed on it. There is no messaging engine on Single bus structure

I)In single bus structure all units are connected in the same bus than connecting different buses as multiple bus structure.

Ii)multiple bus structure's performance is better than single bus structure.
Iii)single bus structure's cost is cheap than multiple bus structure.

Computer software, or just **software** is a general term used to describe the role that computer programs, procedures and documentation play in a computer system.

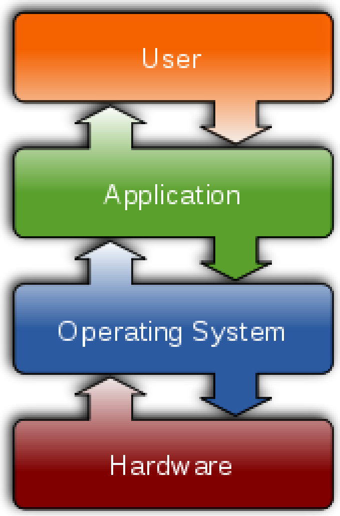
The term includes:

- Application software, such as word processors which perform productive tasks for users.
- Firmware, which is software programmed resident to electrically programmable memory devices on board mainboards or other types of integrated hardware carriers.
- Middleware, which controls and co-ordinates distributed systems.
- System software such as operating systems, which interface with hardware to provide the necessary services for application software.
- Software testing is a domain dependent of development and programming. Software testing consists of various methods to test and declare a software product fit before it can be launched for use by either an individual or a group.
- Testware, which is an umbrella term or container term for all utilities and application software that serve in combination for testing a software package but not necessarily may optionally contribute to operational purposes. As such, testware is not a standing configuration but merely a working environment for application software or subsets thereof.

Software Characteristics

- Software is developed and engineered.
- Software doesn't "wear-out".
- Most software continues to be custom built.

Types of software



A layer structure showing where Operating System is located on generally used software systems on desktops

System software

System software helps run the computer hardware and computer system. It includes a combination of the following:

- device drivers
- operating systems
- servers
- utilities
- windowing systems

The purpose of systems software is to unburden the applications programmer from the often complex details of the particular computer being used, including such accessories as communications devices, printers, device readers, displays and keyboards, and also to partition the computer's resources such as memory and processor time in a safe and stable manner. Examples are- Windows XP, Linux and Mac.

Programming software

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:

- compilers
- debuggers
- interpreters
- linkers
- text editors

Application software

Application software allows end users to accomplish one or more specific (not directly computer development related) tasks. Typical applications include:

- industrial automation
- business software
- computer games
- quantum chemistry and solid state physics software
- telecommunications (i.e., the internet and everything that flows on it)
- databases
- educational software
- medical software
- military software
- molecular modeling software
- image editing
- spreadsheet
- simulation software
- Word processing
- Decision making software

Application software exists for and has impacted a wide variety of topics.

Assembler

Typically a modern **assembler** creates object code by translating assembly instruction mnemonics into op codes, 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. Most assemblers also include macro facilities for performing textual substitution—e.g., to generate common short sequences of instructions to run inline, instead of in a subroutine.

There are two types of assemblers based on how many passes through the source are needed to produce the executable program. One-pass assemblers go through the source code once and assumes that all symbols will be defined before any instruction that references them. Two-pass assemblers (and multi-pass assemblers) create a table with all unresolved symbols in the first pass, then use the 2nd pass to resolve these addresses. The advantage in one-pass assemblers is speed, which is not as important as it once was with advances in computer speed and capabilities. The advantage of the two-pass assembler is that symbols can be defined anywhere in the program source. As a result, the program can be defined in a more logical and meaningful way. This makes two-pass assembler programs easier to read and maintain.

More sophisticated high-level assemblers provide language abstractions such as:

- Advanced control structures
- High-level procedure/function declarations and invocations
- High-level abstract data types, including structures/records, unions, classes, and sets
- Sophisticated macro processing
- Object-Oriented features such as encapsulation, polymorphism, inheritance, interfaces

Assembly language

A program written in assembly language consists of a series of *instructions*--mnemonics that correspond to a stream of executable instructions, when translated by an assembler, that can be loaded into memory and executed.

For example, an x86/IA-32 processor can execute the following binary instruction as expressed in machine language (see x86 assembly language):

- Binary: 10110000 01100001 (Hexadecimal: B0 61)

The equivalent assembly language representation is easier to remember (example in Intel syntax, more *mnemonic*):

- `MOV AL, 61h`

This instruction means:

- Move the value 61h (or 97 decimal; the h-suffix means hexadecimal; into the processor register named "AL".

The mnemonic "mov" represents the opcode **1011** which *moves* the value in the second operand into the register indicated by the first operand. The mnemonic was chosen by the instruction set designer to abbreviate "move", making it easier for the programmer to remember. A comma-separated list of arguments or parameters follows the opcode; this is a typical assembly language statement.

In practice many programmers drop the word *mnemonic* and, technically incorrectly, call "mov" an *op code*. When they do this they are referring to the underlying binary code which it represents. To put it another way, a mnemonic such as "mov" is not an op code, but as it symbolizes an op code, one might refer to "the op code mov" for example when one intends to refer to the binary op code it symbolizes rather than to the symbol -- the mnemonic -- itself. As few modern programmers have need to be mindful of actually what binary patterns are the op codes for specific instructions, the distinction has in practice become a bit blurred among programmers but not among processor designers.

Transforming assembly into machine language is accomplished by an assembler, and the reverse by a disassembler. Unlike in high-level languages, there is usually a one-to-one correspondence between simple assembly statements and machine language instructions. However, in some cases, an assembler may provide *pseudoinstructions* which expand into several machine language instructions to provide commonly needed functionality. For example, for a machine that lacks a "branch if greater or equal" instruction, an assembler may provide a pseudoinstruction that expands to the machine's "set if less than" and "branch if zero (on the result of the set instruction)". Most full-featured assemblers also provide a rich macro language (discussed below) which is used by vendors and programmers to generate more complex code and data sequences.

Each computer architecture and processor architecture has its own machine language. On this level, each instruction is simple enough to be executed using a relatively small number of electronic circuits. Computers differ by the number and type of operations they support. For example, a new 64-bit machine would have different circuitry from a 32-bit machine. They may also have different sizes and numbers of registers, and different representations of data types in storage. While most general-purpose computers are able to carry out essentially the same functionality, the ways they do so differ; the corresponding assembly languages reflect these differences.

Multiple sets of mnemonics or assembly-language syntax may exist for a single instruction set, typically instantiated in different assembler programs. In these cases, the most popular one is usually that supplied by the manufacturer and used in its documentation.

Basic elements

Any Assembly language consists of 3 types of instruction statements which are used to define the program operations:

- Op code mnemonics
- Data sections
- Assembly directives

Opcode mnemonics

Instructions (statements) in assembly language are generally very simple, unlike those in high-level languages. Generally, an opcode is a symbolic name for a single executable machine language instruction, and there is at least one opcode mnemonic defined for each machine language instruction. Each instruction typically consists of an *operation* or *opcode* plus zero or more *operands*. Most instructions refer to a single value, or a pair of values. Operands can be either immediate (typically one byte values, coded in the instruction itself) or the addresses of data located elsewhere in storage. This is determined by the underlying processor architecture: the assembler merely reflects how this architecture works.

Data sections

There are instructions used to define data elements to hold data and variables. They define what type of data, length and alignment of data. These instructions can also define whether the data is available to outside programs (programs assembled separately) or only to the program in which the data section is defined.

Assembly directives and pseudo-ops

Assembly directives are instructions that are executed by the assembler at assembly time, not by the CPU at run time. They can make the assembly of the program dependent on parameters input by the programmer, so that one program can be assembled different ways, perhaps for different applications. They also can be used to manipulate presentation of the program to make it easier for the programmer to read and maintain.

(For example, pseudo-ops would be used to reserve storage areas and optionally their initial contents.) The names of pseudo-ops often start with a dot to distinguish them from machine instructions.

Some assemblers also support *pseudo-instructions*, which generate two or more machine instructions.

Symbolic assemblers allow programmers to associate arbitrary names (*labels* or *symbols*) with memory locations. Usually, every constant and variable is given a name so instructions can reference those locations by name, thus promoting self-documenting code. In executable code, the name of each subroutine is associated with its entry point, so any calls to a subroutine can use its name. Inside subroutines, GOTO destinations are given labels. Some assemblers support *local symbols* which are lexically distinct from normal symbols (e.g., the use of "10$" as a GOTO destination).

Most assemblers provide flexible symbol management, allowing programmers to manage different namespaces, automatically calculate offsets within data structures, and assign labels that refer to literal values or the result of simple computations performed by the assembler. Labels can also be used to initialize constants and variables with relocatable addresses.

Assembly languages, like most other computer languages, allow comments to be added to assembly source code that are ignored by the assembler. Good use of comments is even more important with assembly code than with higher-level languages, as the meaning and purpose of a sequence of instructions is harder to decipher from the code itself.

Wise use of these facilities can greatly simplify the problems of coding and maintaining low-level code. *Raw* assembly source code as generated by compilers or disassemblers—code without any comments, meaningful symbols, or data definitions—is quite difficult to read when changes must be made.

Types of Addressing Modes

Each instruction of a computer specifies an operation on certain data. The are various ways of specifying address of the data to be operated on. These different ways of specifying data are called the addressing modes. The most common addressing modes are:

- Immediate addressing mode
- Direct addressing mode
- Indirect addressing mode
- Register addressing mode
- Register indirect addressing mode
- Displacement addressing mode
- Stack addressing mode

To specify the addressing mode of an instruction several methods are used. Most often used are :

a) Different operands will use different addressing modes.
b) One or more bits in the instruction format can be used as mode field. The value of the mode field determines which addressing mode is to be used.

The effective address will be either main memory address of a register.

Immediate Addressing:

This is the simplest form of addressing. Here, the operand is given in the instruction itself. This mode is used to define a constant or set initial values of variables. The advantage of this mode is that no memory reference other than instruction fetch is required to obtain operand. The disadvantage is that the size of the number is limited to the size of the address field, which most instruction sets is small compared to word length.

Direct Addressing:

In direct addressing mode, effective address of the operand is given in the address field of the instruction. It requires one memory reference to read the operand from the given location and provides only a limited address space. Length of the address field is usually less than the word length.

Ex : Move P, Ro, Add Q, Ro P and Q are the address of operand.

Indirect Addressing:

Indirect addressing mode, the address field of the instruction refers to the address of a word in memory, which in turn contains the full length address of the operand. The advantage of this mode is that for the word length of N, an address space of 2N can be addressed. He disadvantage is that instruction execution requires two memory reference to fetch the operand Multilevel or cascaded indirect addressing can also be used.

Register Addressing:

Register addressing mode is similar to direct addressing. The only difference is that the address field of the instruction refers to a register rather than a memory location 3 or 4 bits are used as address field to reference 8 to 16 generate purpose registers. The advantages of register addressing are Small address field is needed in the instruction.

Register Indirect Addressing:

This mode is similar to indirect addressing. The address field of the instruction refers to a register. The register contains the effective address of the operand. This mode uses one memory reference to obtain the operand. The address space is limited to the width of the registers available to store the effective address.

Displacement Addressing:

In displacement addressing mode there are 3 types of addressing mode. They are :

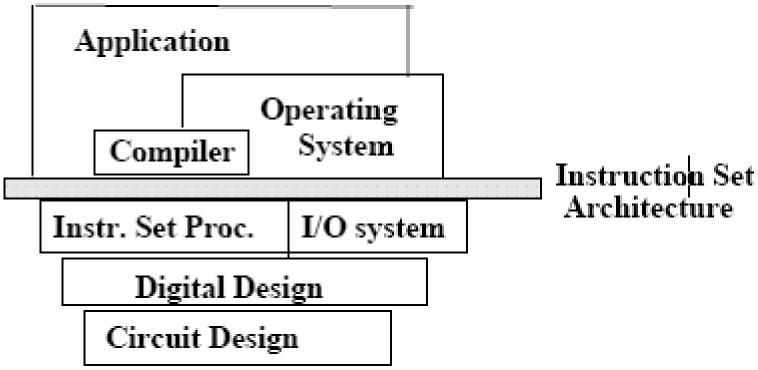
1) Relative addressing
2) Base register addressing
3) Indexing addressing.

This is a combination of direct addressing and register indirect addressing. The value contained in one address field. A is used directly and the other address refers to a register whose contents are added to A to produce the effective address.

Stack Addressing:

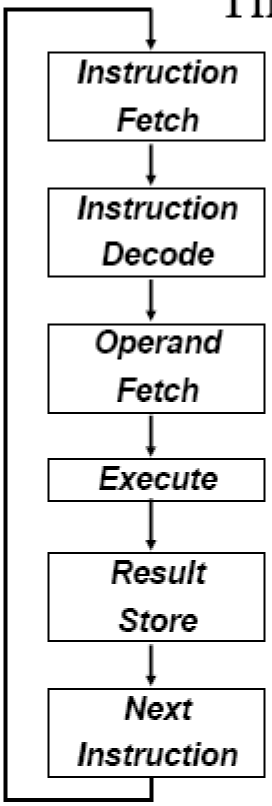
Stack is a linear array of locations referred to as last-in first out queue. The stack is a reserved block of location, appended or deleted only at the top of the stack. Stack pointer is a register which stores the address of top of stack location. This mode of addressing is also known as implicit addressing.

The Instruction Set Architecture



•superscalar processor --can execute more than one instructions per cycle.
•cycle--smallest unit of time in a processor.
•parallelism--the ability to do more than one thingat once.
•pipelining--overlapping parts of a large task to increase throughput without decreasing latency

The Instruction Execution Cycle



Instruction Fetch — Obtain instruction from program storage

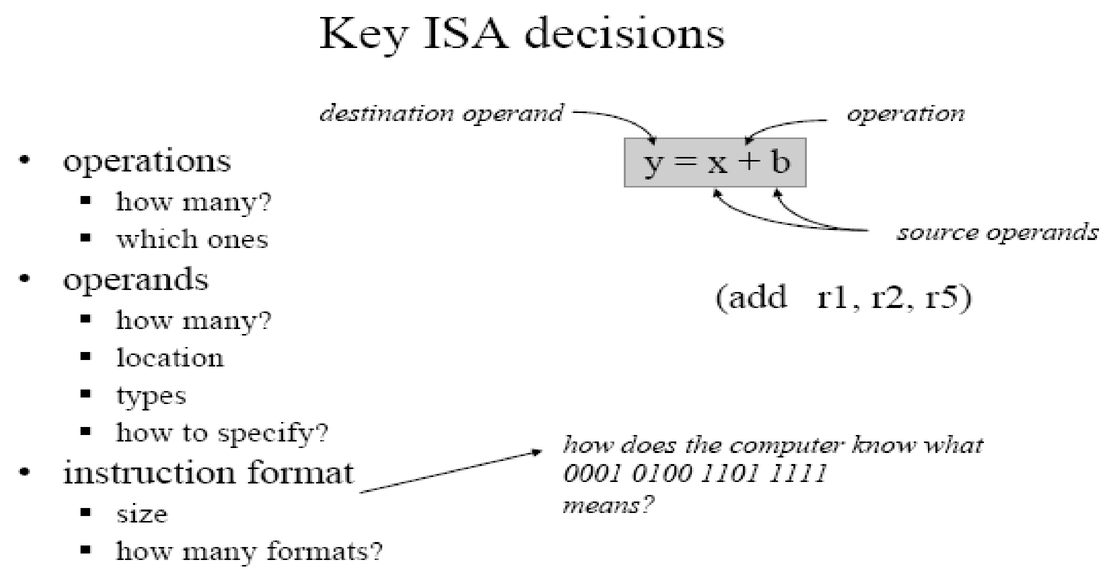
Instruction Decode — Determine required actions and instruction size

Operand Fetch — Locate and obtain operand data

Execute — Compute result value or status

Result Store — Deposit results in storage for later use

Next Instruction — Determine successor instruction



Key ISA decisions

- operations
  - how many?
  - which ones
- operands
  - how many?
  - location
  - types
  - how to specify?
- instruction format
  - size
  - how many formats?

destination operand

operation

y = x + b

source operands

(add r1, r2, r5)

how does the computer know what 0001 0100 1101 1111 means?

Crafting an ISA

•We’ll look at some of the decisions facing an instruction set architect, and

•how those decisions were made in the design of the MIPS instruction set.

•MIPS, like SPARC, PowerPC, and Alpha AXP, is a RISC (Reduced Instruction Set Computer) ISA.

–fixed instruction length

–few instruction formats

–load/store architecture

•RISC architectures worked because they enabled pipelining. They continue to thrive because they enable parallelism.

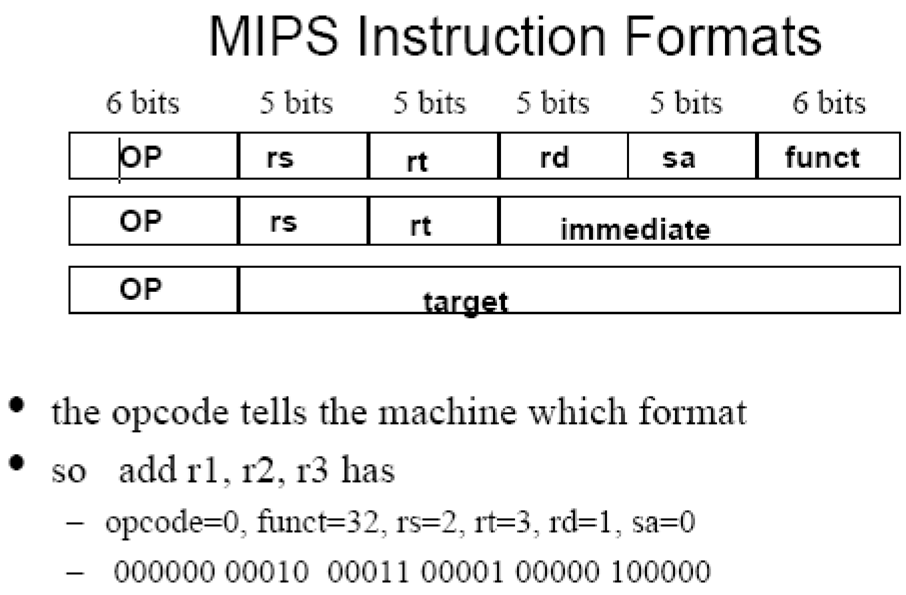
Instruction Length

•Variable-length instructions (Intel 80x86, VAX) require multi-step fetch and decode, but allow for a much more flexible and compact instruction set.

•Fixed-length instructions allow easy fetch and decode, and simplify pipelining and parallelism.

All MIPS instructions are 32 bits long.

–this decision impacts every other ISA decision we make because it makes instruction bits scarce.



MIPS Instruction Formats

6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
OP	rs	rt	rd	sa	funct
OP	rs	rt	immediate		
OP	target				

- the opcode tells the machine which format
- so add r1, r2, r3 has
  - opcode=0, funct=32, rs=2, rt=3, rd=1, sa=0
  - 000000 00010 00011 00001 00000 100000

Accessing the Operands

•operands are generally in one of two places:
–registers (32 int, 32 fp)
–memory (232locations)
•registers are
–easy to specify
–close to the processor (fast access)
•the idea that we want to access registers whenever possible led to load-store architectures.
–normal arithmetic instructions only access registers
–only access memory with explicit loads and stores.

Load-store architectures

can do:
add r1=r2+r3
and
load r3, M(address)
⇒forces heavy dependence on registers, which is exactly what you want in today’s CPUs

can’t do

add r1 = r2 + M(address)
-more instructions
+ fast implementation (e.g., easy pipelining)

How Many Operands?

•Most instructions have three operands (e.g., z = x + y).
•Well-known ISAsspecify 0-3 (explicit) operands per instruction.
•Operands can be specified implicitly or explicity.

How Many Operands?
Basic ISA Classes

Accumulator:

1 addressadd Aacc ←acc + mem[A]

Stack:
0 addressaddtos←tos+ next

General Purpose Register:

2 addressadd A BEA(A) ←EA(A) + EA(B)
3 addressadd A B CEA(A) ←EA(B) + EA(C)

Load/Store:

3 addressadd Ra RbRcRa ←Rb+ Rc
load Ra RbRa ←mem[Rb]
store Ra Rbmem[Rb] ←Ra

Comparing the Number of Instructions

Code sequence for C = A + B for four classes of instruction sets:

Stack	Accumulator	Register	Register
		(register-memory)	(load-store)
Push A	**Load A**	**ADD C, A, B**	**Load R1,A**
Push B	**Add B**		**Load R2,B**
Add	**Store C**		**Add R3,R1,R2**
Pop C			**Store C,R3**

Addressing Modes

how do we specify the operand we want?

-	**Register direct**	**R3**
-	**Immediate (literal)**	**#25**
-	**Direct (absolute)**	**M[10000]**
-	**Register indirect**	**M[R3]**
-	**Base+Displacement**	**M[R3 + 10000]**
	if register is the program counter, this is *PC-relative*	
-	**Base+Index**	**M[R3 + R4]**
-	**Scaled Index**	**M[R3 + R4*d + 10000]**
-	**Autoincrement**	**M[R3++]**
-	**Autodecrement**	**M[R3 - -]**
-	**Memory Indirect**	**M[M[R3]]**

MIPS addressing modes

register direct

OP	rs	rt	rd	sa	funct

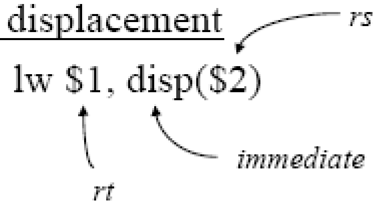
add $1, $2, $3

immediate

OP	rs	rt	immediate

add $1, $2, #35

base + displacement


lw $1, disp($2)

register indirect

⇨ disp = 0

absolute

⇨ (rs) = 0

Memory Organization

- Viewed as a large, single-dimension array, with an address.
- A memory address is an index into the array
- "Byte addressing" means that the index points to a byte of memory.

0	8 bits of data
1	8 bits of data
2	8 bits of data
3	8 bits of data
4	8 bits of data
5	8 bits of data
6	8 bits of data
...	

Memory Organization

- Bytes are nice, but most data items use larger "words"
- For MIPS, a word is 32 bits or 4 bytes.

0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

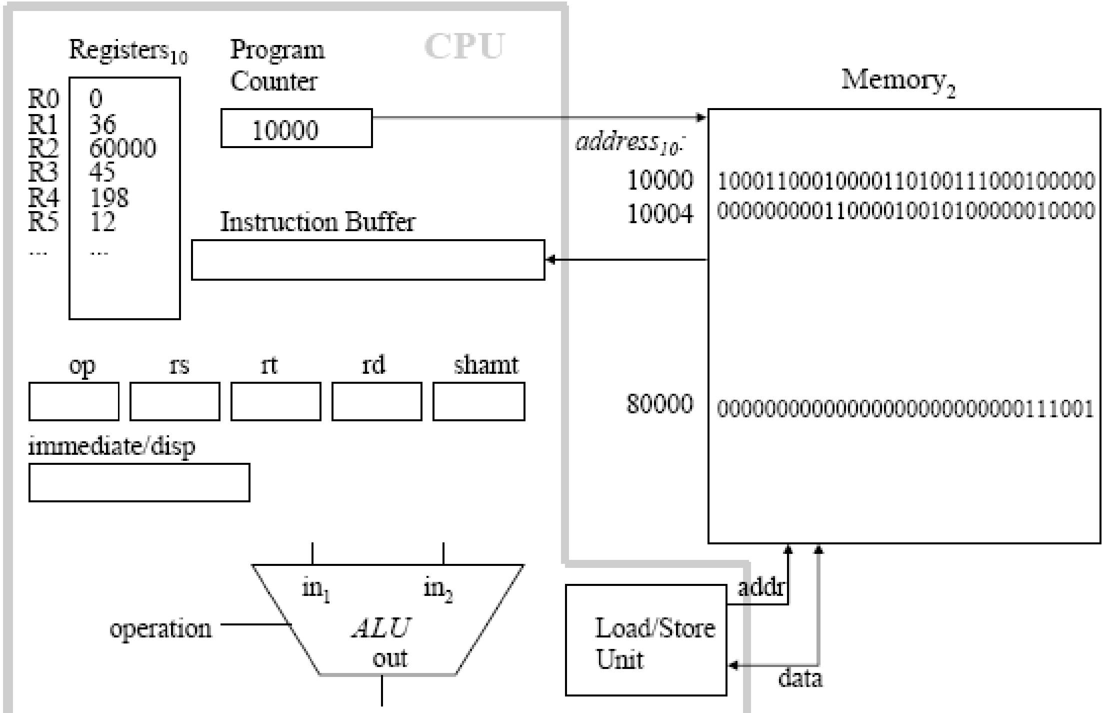
Registers hold 32 bits of data

- 2^{32} bytes with byte addresses from 0 to 232-1
- 2^{30} words with byte addresses 0, 4, 8, ... 232-4
- Words are aligned
  i.e., what are the least 2 significant bits of a word address?

The MIPS ISA, so far

- fixed 32-bit instructions
- 3 instruction formats
- 3-operand, load-store architecture
- 32 general-purpose registers (integer, floating point)
  - R0 always equals 0.
- 2 special-purpose integer registers, HI and LO, because multiply and divide produce more than 32 bits.
- registers are 32-bits wide (word)
- register, immediate, and base+displacement addressing modes

Review -- Instruction Execution in a CPU



Four principles of IS architecture

–simplicity favors regularity
–smaller is faster
–good design demands compromise
–make the common case fast

Instruction Set Architecture (ISA)

The *Instruction Set Architecture* (ISA) is the part of the processor that is visible to the programmer or compiler writer. The ISA serves as the boundary between software and hardware. We will briefly describe the instruction sets found in many of the microprocessors used today. The ISA of a processor can be described using 5 catagories:

Operand Storage in the CPU
Where are the operands kept other than in memory?
Number of explicit named operands
How many operands are named in a typical instruction.

Operand location

Can any ALU instruction operand be located in memory? Or must all operands be kept internaly in the CPU?

Operations

What operations are provided in the ISA.

Type and size of operands

What is the type and size of each operand and how is it specified?

Of all the above the most distinguishing factor is the first.

The 3 most common types of ISAs are:

1. ***Stack*** - The operands are implicitly on top of the stack.
2. ***Accumulator*** - One operand is implicitly the accumulator.
3. *General Purpose Register (GPR)* - All operands are explicitely mentioned, they are either registers or memory locations.

Lets look at the assembly code of

```
A = B + C;
```

in all 3 architectures:

Stack	Accumulator	GPR
PUSH A	LOAD A	LOAD R1,A
PUSH B	ADD B	ADD R1,B
ADD	STORE C	STORE R1,C
POP C	-	-

Not all processors can be neatly tagged into one of the above catagories. The i8086 has many instructions that use implicit operands although it has a general register set. The i8051 is another example, it has 4 banks of GPRs but most instructions must have the A register as one of its operands.

What are the advantages and disadvantages of each of these approachs?

Stack

Advantages: Simple Model of expression evaluation (reverse polish). Short instructions.
Disadvantages: A stack can't be randomly accessed This makes it hard to generate eficient code. The stack itself is accessed every operation and becomes a bottleneck.

Accumulator

Advantages: Short instructions.
Disadvantages: The accumulator is only temporary storage so memory traffic is the highest for this approach.

GPR

Advantages: Makes code generation easy. Data can be stored for long periods in registers.
Disadvantages: All operands must be named leading to longer instructions.

Earlier CPUs were of the first 2 types but in the last 15 years all CPUs made are GPR processors. The 2 major reasons are that registers are faster than memory, the more data that can be kept internaly in the CPU the faster the program wil run. The other reason is that registers are easier for a compiler to use.

Reduced Instruction Set Computer (RISC)

As we mentioned before most modern CPUs are of the GPR (General Purpose Register) type. A few examples of such CPUs are the IBM 360, DEC VAX, Intel 80x86 and Motorola 68xxx. But while these CPUS were clearly better than previous stack and accumulator based CPUs they were still lacking in several areas:

1. Instructions were of varying length from 1 byte to 6-8 bytes. This causes problems with the pre-fetching and pipelining of instructions.
2. ALU (Arithmetic Logical Unit) instructions could have operands that were memory locations. Because the number of cycles it takes to access memory varies so does the whole instruction. This isn't good for compiler writers, pipelining and multiple issue.
3. Most ALU instruction had only 2 operands where one of the operands is also the destination. This means this operand is destroyed during the operation or it must be saved before somewhere.

Thus in the early 80's the idea of RISC was introduced. The SPARC project was started at Berkeley and the MIPS project at Stanford. RISC stands for Reduced Instruction Set Computer. The ISA is composed of instructions that all have exactly the same size, usualy 32 bits. Thus they can be pre-fetched and pipelined succesfuly. All ALU instructions have 3 operands which are only registers. The only memory access is through explicit LOAD/STORE instructions.
Thus A = B + C will be assembled as:

```
LOAD  R1,A
LOAD  R2,B
ADD   R3,R1,R2
STORE C,R3
```

Although it takes 4 instructions we can reuse the values in the registers.

Why is this architecture called RISC?

What is Reduced about it?
The answer is that to make all instructions the same length the number of bits that are used for the opcode is reduced. Thus less instructions are provided. The instructions that were thrown out are the less important string and BCD (binary-coded decimal) operations. In fact, now that memory access is restricted there aren't several kinds of MOV or ADD instructions. Thus the older architecture is called CISC (Complete Instruction Set Computer). RISC architectures are also called *LOAD/STORE* architectures.

The number of registers in RISC is usualy 32 or more. The first RISC CPU the MIPS 2000 has 32 GPRs as opposed to 16 in the 68xxx architecture and 8 in the 80x86 architecture. The only disadvantage of RISC is its code size. Usualy more instructions are needed and there is a waste in short instructions (POP, PUSH).

So why are there still CISC CPUs being developed?

Why is Intel spending time and money to manufacture the Pentium II and the Pentium III?

The answer is simple, backward compatibility. The IBM compatible PC is the most common computer in the world. Intel wanted a CPU that would run all the applications that are in the hands of more than 100 million users. On the other hand Motorola which builds the 68xxx series which was used in the Macintosh made the transition and together with IBM and Apple built the Power PC (PPC) a RISC CPU which is installed in the new Power Macs. As of now Intel and the PC manufacturers are making more money but with Microsoft playing in the RISC field as well (Windows NT runs on Compaq's Alpha) and with the promise of Java the future of CISC isn't clear at all.

An important lesson that can be learnt here is that superior technology is a factor in the computer industry, but so are marketing and price as well (if not more).

The CISC Approach

The primary goal of CISC architecture is to complete a task in as few lines of assembly as possible. This is achieved by building processor hardware that is capable of understanding and executing a series of operations. For this particular task, a CISC processor would come prepared with a specific instruction (we'll call it "MULT"). When executed, this instruction loads the two values into separate registers, multiplies the operands in the execution unit, and then stores the product in the appropriate register. Thus, the entire task of multiplying two numbers can be completed with one instruction:

```
MULT 2:3, 5:2
```

MULT is what is known as a "complex instruction." It operates directly on the computer's memory banks and does not require the programmer to explicitly call any loading or storing functions. It closely resembles a command in a higher level language. For instance, if we let "a" represent the value of 2:3 and "b" represent the value of 5:2, then this command is identical to the C statement "a = a * b."

One of the primary advantages of this system is that the compiler has to do very little work to translate a high-level language statement into assembly. Because the length of the code is relatively short, very little RAM is required to store instructions. The emphasis is put on building complex instructions directly into the hardware.

The RISC Approach

RISC processors only use simple instructions that can be executed within one clock cycle. Thus, the "MULT" command described above could be divided into three separate commands: "LOAD," which moves data from the memory bank to a register, "PROD," which finds the product of two operands located within the registers, and "STORE," which moves data from a register to the memory banks. In order to perform the exact series of steps described in the CISC approach, a programmer would need to code four lines of assembly:

```
LOAD A, 2:3
LOAD B, 5:2
PROD A, B
STORE 2:3, A
```

At first, this may seem like a much less efficient way of completing the operation. Because there are more lines of code, more RAM is needed to store the assembly level instructions. The compiler must also perform more work to convert a high-level language statement into code of this form.

However, the RISC strategy also brings some very important advantages. Because each instruction requires only one clock cycle to execute, the entire program will execute in approximately the same amount of time as the multi-cycle "MULT" command. These RISC "reduced instructions" require less transistors of hardware space than the complex instructions, leaving more room for general purpose registers. Because all of the instructions execute in a uniform amount of time (i.e. one clock), pipelining is possible.

CISC	**RISC**
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per second	Low cycles per second, large code sizes
Transistors used for storing complex instructions	Spends more transistors on memory registers

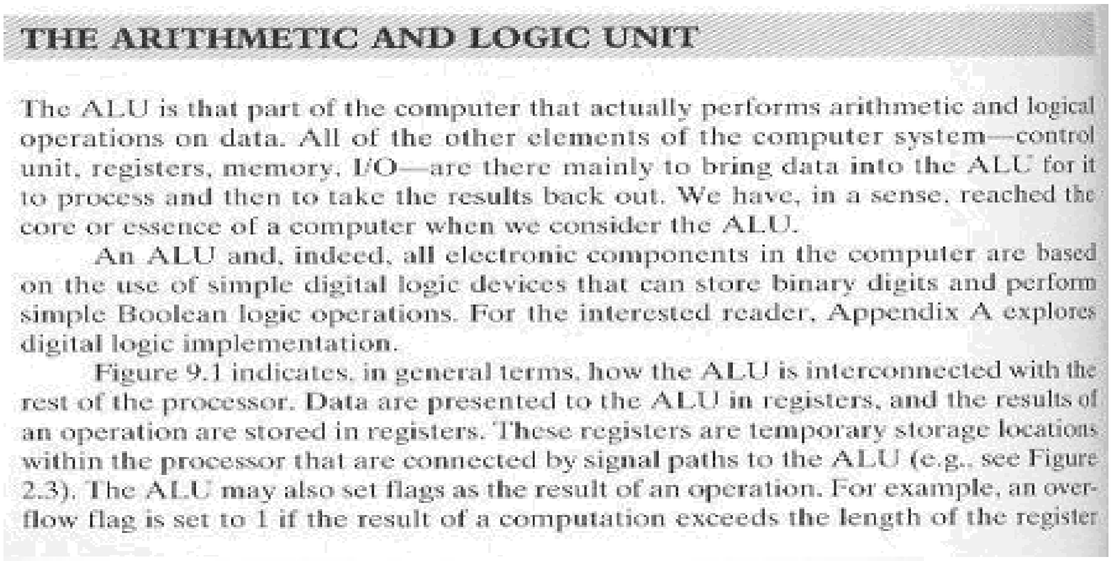
Separating the "LOAD" and "STORE" instructions actually reduces the amount of work that the computer must perform. After a CISC-style "MULT" command is executed, the processor automatically erases the registers. If one of the operands needs to be used for another computation, the processor must re-load the data from the memory bank into a register. In RISC, the operand will remain in the register until another value is loaded in its place.

The Performance Equation
The following equation is commonly used for expressing a computer's performance ability:

$$\frac{\text{time}}{\text{program}} = \frac{\text{time}}{\text{cycle}} \text{ x } \frac{\text{cycles}}{\text{instruction}} \text{ x } \frac{\text{instructions}}{\text{program}}$$

The CISC approach attempts to minimize the number of instructions per program, sacrificing the number of cycles per instruction. RISC does the opposite, reducing the cycles per instruction at the cost of the number of instructions per program.

UNIT – II
DATA PATH DESIGN



THE ARITHMETIC AND LOGIC UNIT

The ALU is that part of the computer that actually performs arithmetic and logical operations on data. All of the other elements of the computer system—control unit, registers, memory, I/O—are there mainly to bring data into the ALU for it to process and then to take the results back out. We have, in a sense, reached the core or essence of a computer when we consider the ALU.

An ALU and, indeed, all electronic components in the computer are based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations. For the interested reader, Appendix A explores digital logic implementation.

Figure 9.1 indicates, in general terms, how the ALU is interconnected with the rest of the processor. Data are presented to the ALU in registers, and the results of an operation are stored in registers. These registers are temporary storage locations within the processor that are connected by signal paths to the ALU (e.g., see Figure 2.3). The ALU may also set flags as the result of an operation. For example, an overflow flag is set to 1 if the result of a computation exceeds the length of the register into which it is to be stored. The flag values are also stored in registers within the processor. The control unit provides signals that control the operation of the ALU and the movement of the data into and out of the ALU.

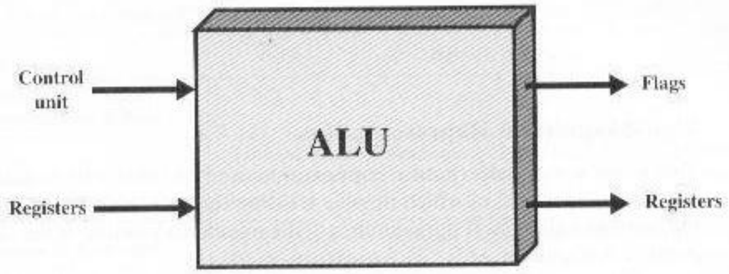


Figure 9.1 ALU Inputs and Outputs

INTEGER REPRESENTATION

In the binary number system,[1] arbitrary numbers can be represented with just the digits zero and one, the minus sign, and the period, or **radix point**.

$$-1101.0101_2 = -13.3125_{10}$$

For purposes of computer storage and processing, however, we do not have the benefit of minus signs and periods. Only binary digits (0 and 1) may be used to represent numbers. If we are limited to nonnegative integers, the representation is straightforward.

An 8-bit word can represent the numbers from 0 to 255, including

$$\begin{aligned} 00000000 &= 0 \\ 00000001 &= 1 \\ 00101001 &= 41 \\ 10000000 &= 128 \\ 11111111 &= 255 \end{aligned}$$

In general, if an n-bit sequence of binary digits $a_{n-1}a_{n-2} \ldots a_1 a_0$ is interpreted as an unsigned integer A, its value is

$$A = \sum_{i=0}^{n-1} 2^i a_i$$

Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers, all of which involve treating the most significant (leftmost) bit in the word as a sign bit. If the sign bit is 0, the number is positive; if the sign bit is 1, the number is negative.

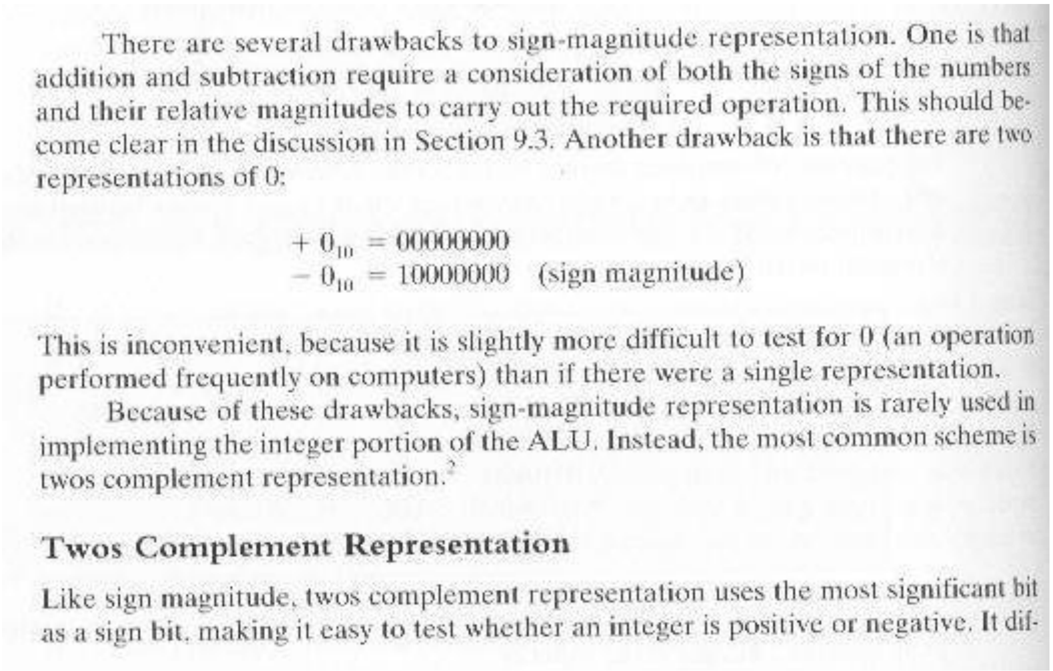
The simplest form of representation that employs a sign bit is the sign-magnitude representation. In an n-bit word, the rightmost $n - 1$ bits hold the magnitude of the integer.

$$\begin{aligned} +18 &= 00010010 \\ -18 &= 10010010 \quad \text{(sign magnitude)} \end{aligned}$$

The general case can be expressed as follows:

Sign Magnitude

$$A = \begin{cases} \displaystyle\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 0 \\ \displaystyle -\sum_{i=0}^{n-2} 2^i a_i & \text{if } a_{n-1} = 1 \end{cases} \tag{9.1}$$



There are several drawbacks to sign-magnitude representation. One is that addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation. This should become clear in the discussion in Section 9.3. Another drawback is that there are two representations of 0:

$$+0_{10} = 00000000$$
$$-0_{10} = 10000000 \quad \text{(sign magnitude)}$$

This is inconvenient, because it is slightly more difficult to test for 0 (an operation performed frequently on computers) than if there were a single representation.

Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU. Instead, the most common scheme is twos complement representation.[2]

Twos Complement Representation

Like sign magnitude, twos complement representation uses the most significant bit as a sign bit, making it easy to test whether an integer is positive or negative. It dif-

Multiplication

More complicated than addition

- Accomplished via shifting and addition

More time and more area

Let's look at 3 versions based on grade school algorithm

01010010 (multiplicand)

x 01101101 (multiplier)

Negative numbers: convert and multiply

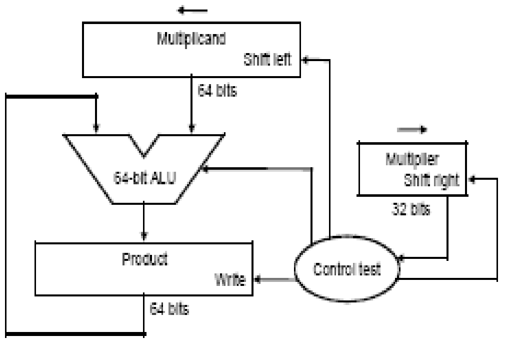
Use other better techniques like Booth’s encoding

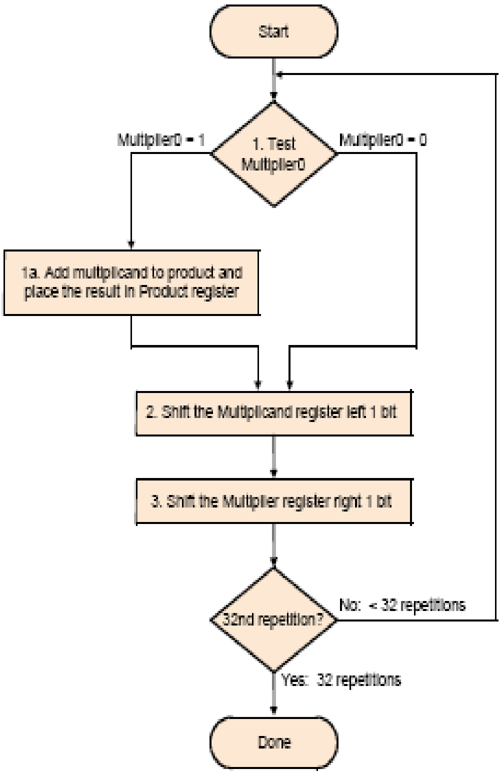
Multiplication

```
        01010010(multiplicand)
      × 01101101 (multiplier)
        00000000
        01010010 ×1
        01010010
       000000000 ×0
       001010010
      0101001000 ×1
      0110011010
     01010010000 ×1
     10000101010
    000000000000 ×0
    010000101010
   0101001000000 ×1
   0111001101010
  01010010000000 ×1
  10001011101010
 000000000000000 ×0
0010001011101010
```

```
  01010010  (multiplicand)
× 01101101  (multiplier)
  00000000
  01010010  ×1
  00101001 0          (add & shr)
  00000000 0 ×0
  00010100 10         (add & shr)
  01010010 00 ×1
  00110011 010        (add & shr)
  01010010 000 ×1
  01000010 1010       (add & shr)
  00000000 0000 ×0
  00100001 01010      (add & shr)
  01010010 00000 ×1
  00111001 101010     (add & shr)
  01010010 000000 ×1
  01000101 1101010    (add & shr)
  00000000 0000000 ×0
  00100010 11101010   (add & shr)
```

Multiplication: Implementation





Start

1. Test Multiplier0

Multiplier0 = 1

Multiplier0 = 0

1a. Add multiplicand to product and place the result in Product register

2. Shift the Multiplicand register left 1 bit

3. Shift the Multiplier register right 1 bit

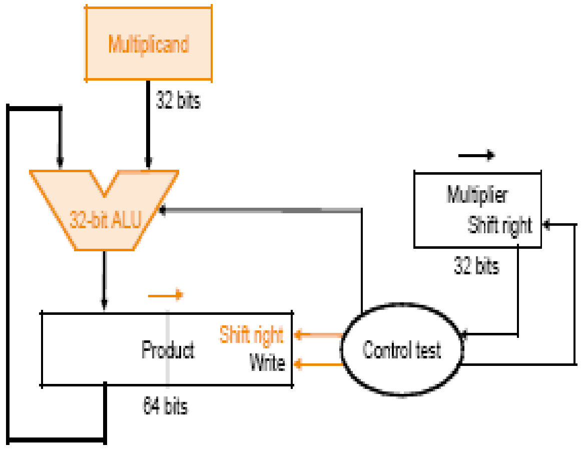
32nd repetition?

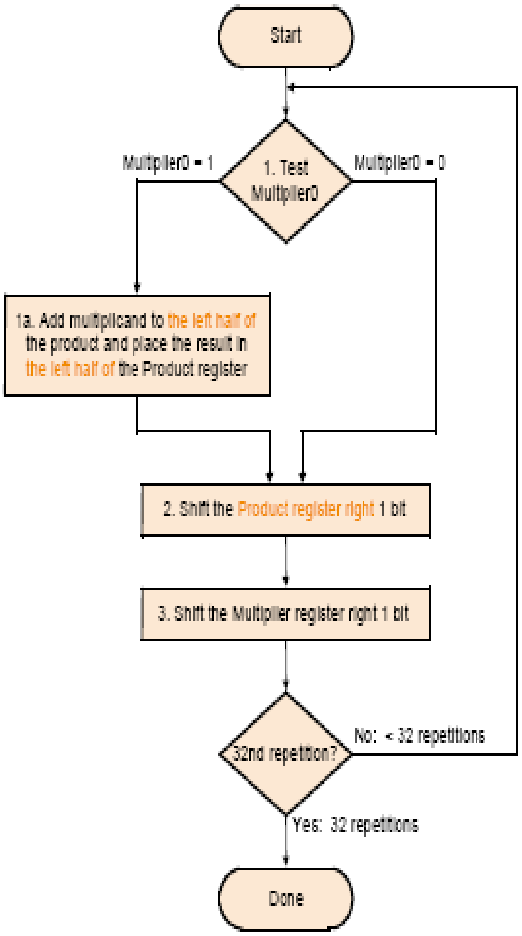
No: < 32 repetitions

Yes: 32 repetitions

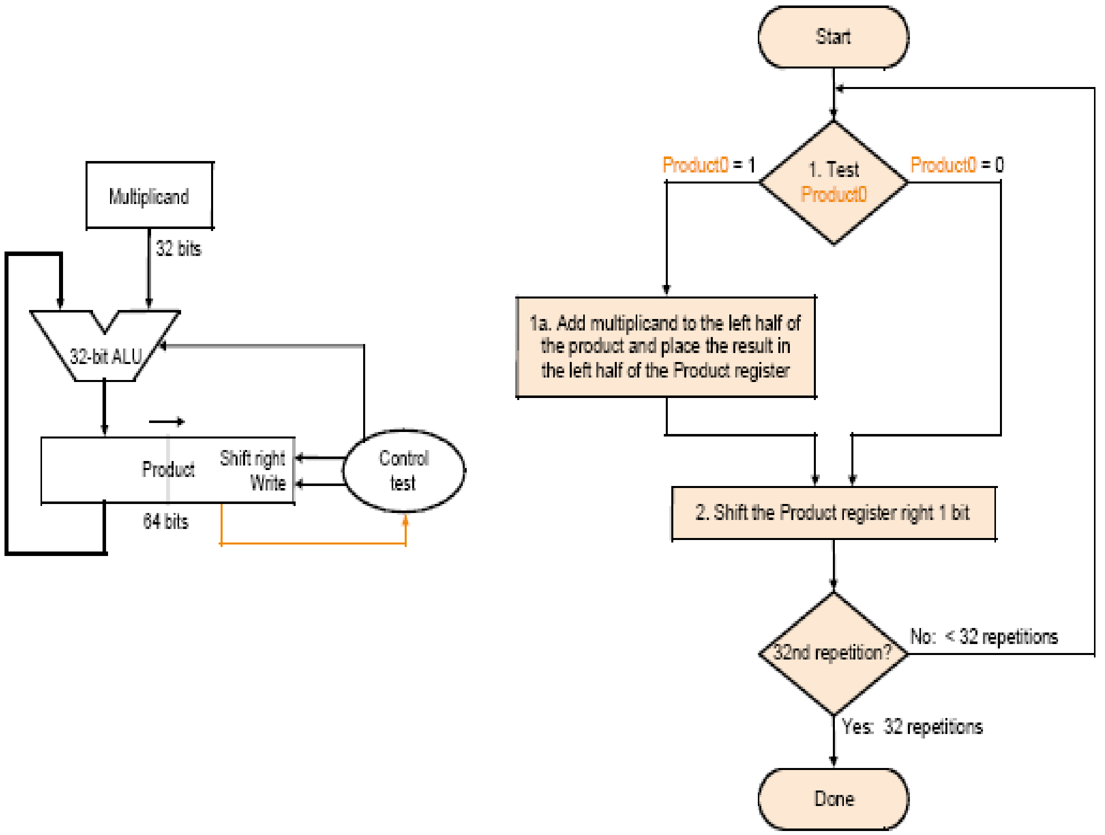
Done

Second version




Start
Multiplier0 = 1
1. Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Final version



Multiplication example: 0010 x 0110

Iteration	Multiplicand	Original Algorithm	
		Step	**Product**
0	0010	Initial values	0000 011**0**
1	0010	1: **0** -> No operation	0000 0110
		2: Shift right	0000 001**1**
2	0010	1a: **1** -> Product = Product + Multiplicand	**0010** 0011
		2: Shift right	0001 0001
3	0010	1a: 1 -> Product = Product + Multiplicand	**0011** 0001
		2: Shift right	0001 100**0**
4	0010	1: **0** -> No operation	0001 1000
		2: Shift right	0000 1100

Signed Multiplication

The easiest way to deal with signed numbers is to first convert the multiplier and multiplicand to positive numbers and then remember the original sign. It turns out that the last algorithm will work with signed numbers provided that when we do the shifting steps we extend the sign of the product.

Speeding up multiplication (Booth's Algorithm)

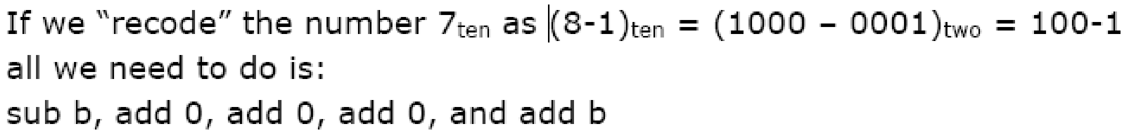
The way we have done multiplication so far consisted of repeatedly scanning the multiplier, adding the multiplicand (or zeros) and shifting the result accumulated.

Observation:
if we could reduce the number of times we have to add the multiplicand that would make the all process faster.
Let say we want to do:

Bxa where a=7_{ten}=0111_{two}

With the algorithm used so far we successively:

add b, add b, add b, and add 0



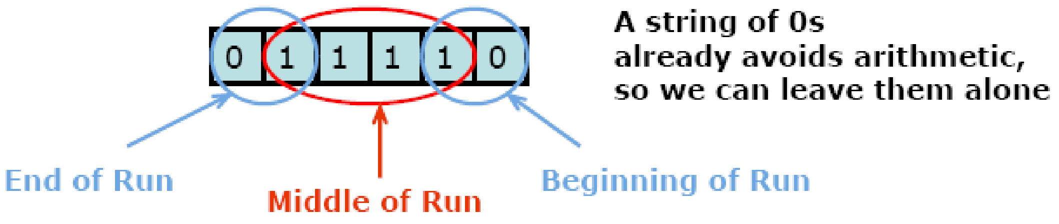
If we "recode" the number 7_{ten} as $(8-1)_{ten}$ = $(1000 - 0001)_{two}$ = 100-1
all we need to do is:
sub b, add 0, add 0, add 0, and add b

Booth's Algorithm

Observation: If besides addition we also use subtraction, we can reduce the number of consecutives additions and therefore we can make the multiplication faster.

This requires to "recode" the multiplier in such a way that the number of consecutive 1s in the multiplier (indeed the number of consecutive additions we should have done) are reduced.

The key to Booth's algorithm is to scan the multiplier and classify group of bits into the beginning, the middle and the end of a run of 1s

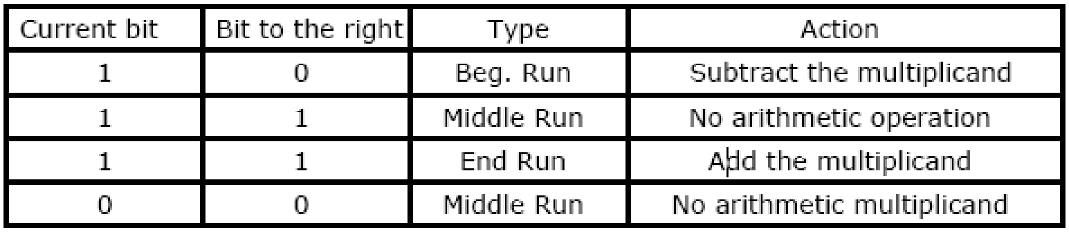


Using Booth's encoding for multiplication

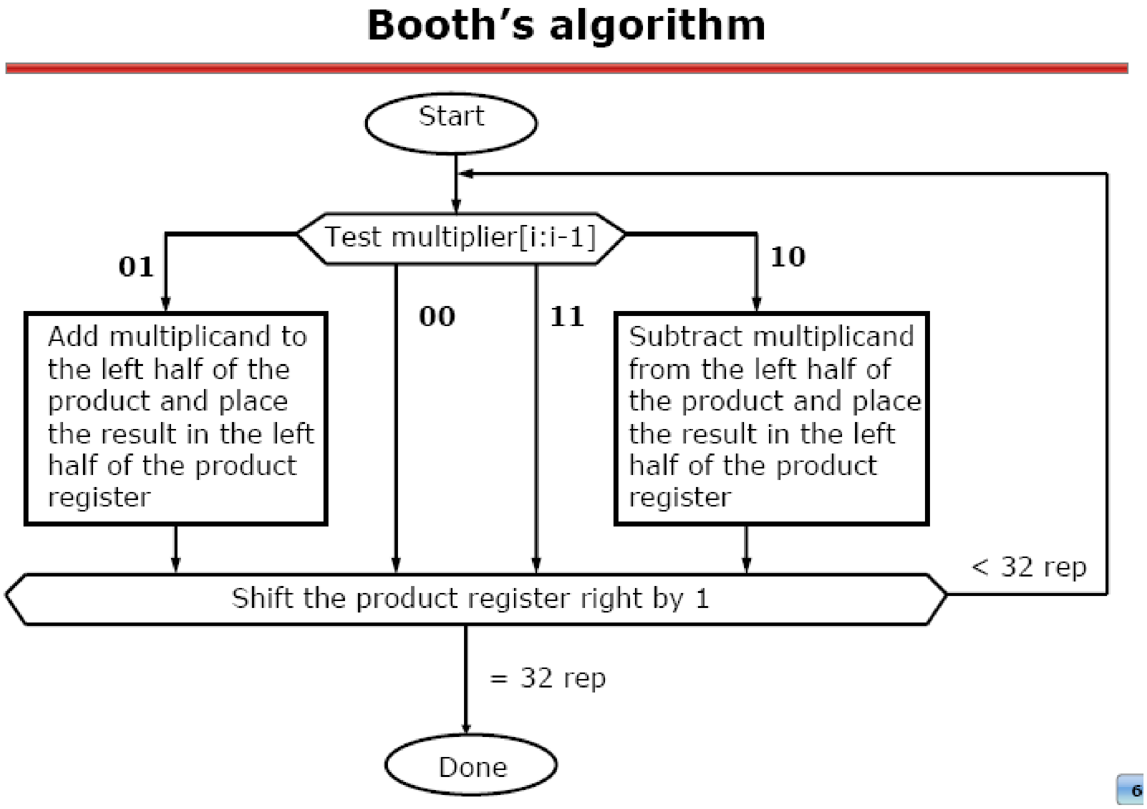
If the initial content of A is an-1…a0 then i-th multiply step, the low-order bit of register A is ai and step (i) in the multiplication algorithm becomes:

1. If ai=0 and ai-1=0, then add 0 to P
2. If ai=0 and ai-1=1, then add B to P
3. If ai=1 and ai-1=0, then subtract B from P
4. If ai=1 and ai-1=1, then add 0 to P

(For the first step when i=0, then add 0 to P)



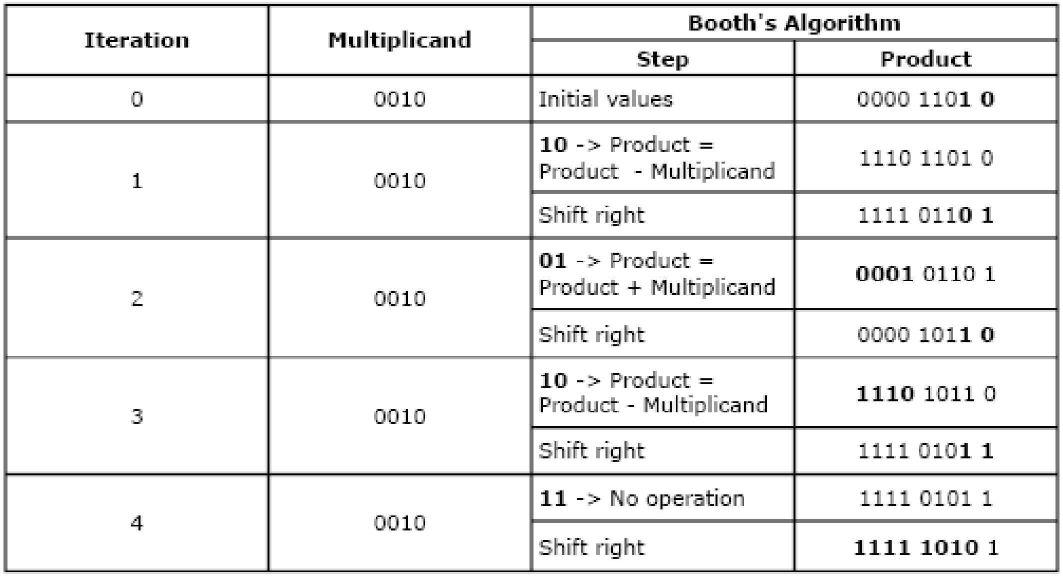
Current bit	Bit to the right	Type	Action
1	0	Beg. Run	Subtract the multiplicand
1	1	Middle Run	No arithmetic operation
1	1	End Run	Add the multiplicand
0	0	Middle Run	No arithmetic multiplicand



Booth's algorithm

Start
Test multiplier[i:i-1]
01
00
11
10
Add multiplicand to the left half of the product and place the result in the left half of the product register
Subtract multiplicand from the left half of the product and place the result in the left half of the product register
Shift the product register right by 1
< 32 rep
= 32 rep
Done
6

Booth's algorithm example



Iteration	Multiplicand	Booth's Algorithm	
		Step	Product
0	0010	Initial values	0000 110**1 0**
1	0010	**10** -> Product = Product - Multiplicand	1110 1101 0
		Shift right	1111 011**0 1**
2	0010	**01** -> Product = Product + Multiplicand	**0001** 0110 1
		Shift right	0000 101**1 0**
3	0010	**10** -> Product = Product - Multiplicand	**1110** 1011 0
		Shift right	1111 010**1 1**
4	0010	**11** -> No operation	1111 0101 1
		Shift right	**1111 1010** 1

Division

Even more complicated can be accomplished via shifting and addition/subtraction
More time and more area we will look at 3 versions based on grade school algorithm

0011 | 0010 0010 (Dividend)

(Divisor) 0011 | 0010 0010 (Dividend)

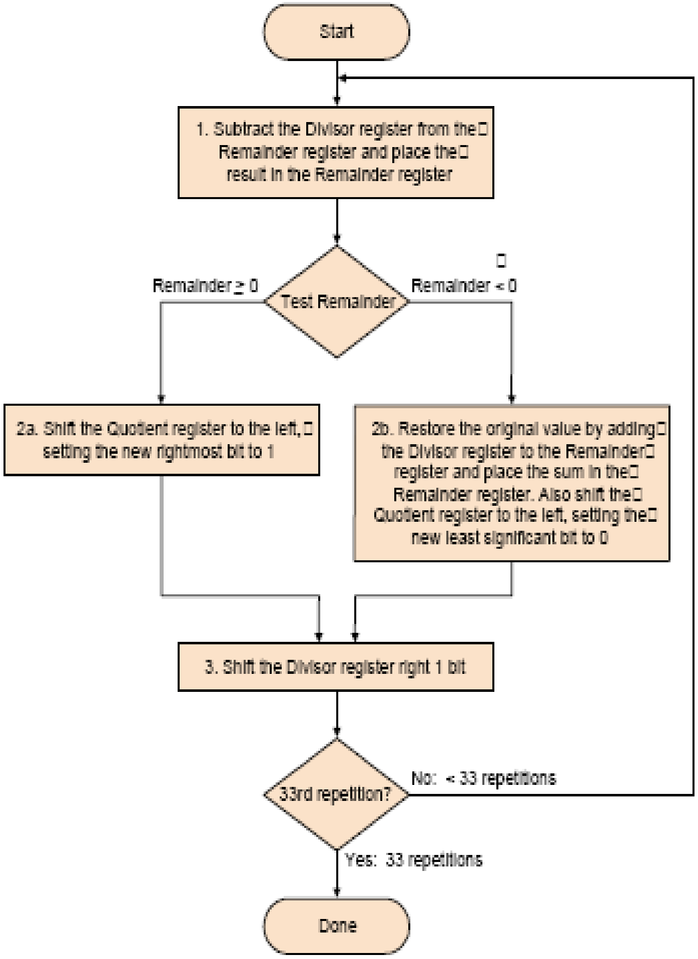
Negative numbers: Even more difficult There are better techniques, we won’t look at them

Division

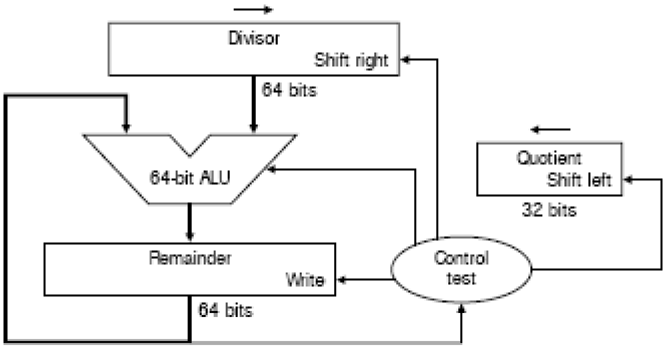
```
                    1001 (Quotient)
              ___________
Divisor 1000  |  1001010 (Dividend)
                -1000
               _________
                    10
                    101
                    1010
                   -1000
               _________
                      10 (Remainder)
```

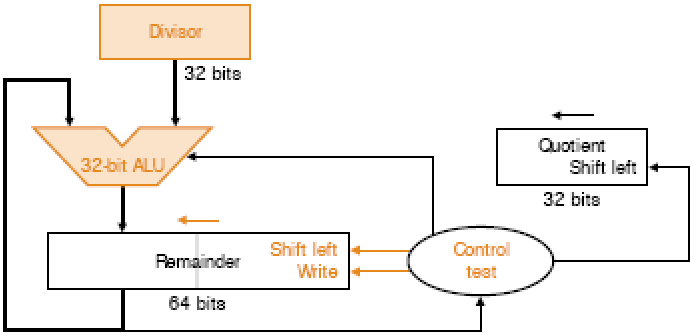
Dividend = Quotient x Divider + Remainder

Division: First Algorithm

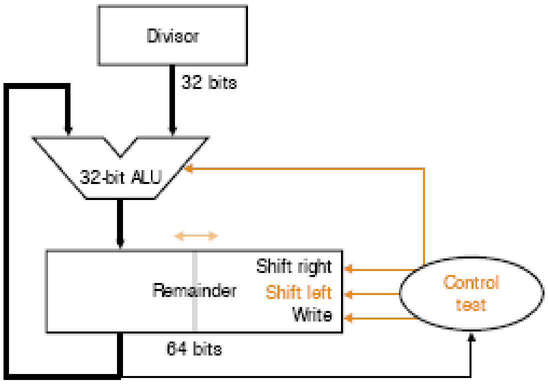

Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

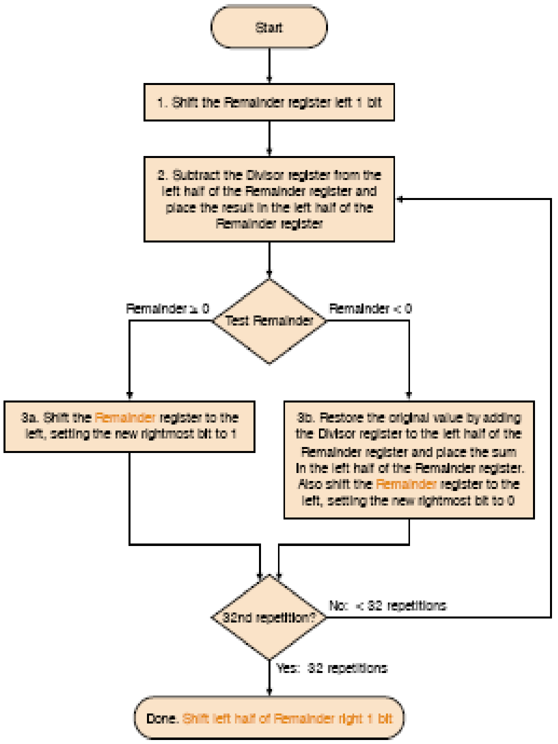
Division - Implementation





Division





Start

1. Shift the Remainder register left 1 bit

2. Subtract the Divisor register from the left half of the Remainder register and place the result in the left half of the Remainder register

Test Remainder

Remainder ≥ 0

Remainder < 0

3a. Shift the Remainder register to the left, setting the new rightmost bit to 1

3b. Restore the original value by adding the Divisor register to the left half of the Remainder register and place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new rightmost bit to 0

32nd repetition?

No: < 32 repetitions

Yes: 32 repetitions

Done. Shift left half of Remainder right 1 bit

Restoring division example

Iteration	Divisor	Divide Algorithm	
		Step	Product
0	0010	Initial values	0000 0111
		Shift reminder left by 1	0000 1110
1	0010	2. Reminder = Reminder - Divisor	**1110** 1110
		3b. (Reminder < 0); +Div; Shift left, R0 = 0	0001 110**0**
2	0010	2. Reminder = Reminder - Divisor	**1111** 1100
		3b. (Reminder < 0); +Div; Shift left, R0 = 0	0011 100**0**
3	0010	2. Reminder = Reminder - Divisor	**0001** 1000
		3a. (Reminder > 0); Shift left, R0 = 1	0011 000**1**
4	0010	2. Reminder = Reminder - Divisor	**0001** 0001
		3a. (Reminder > 0); Shift left, R0 = 1	0010 001**1**
Done	0010	Shift left half of reminder right by 1	**0001** 0011

Non-restoring division

How can we avoid adding the divisor back to the reminder?

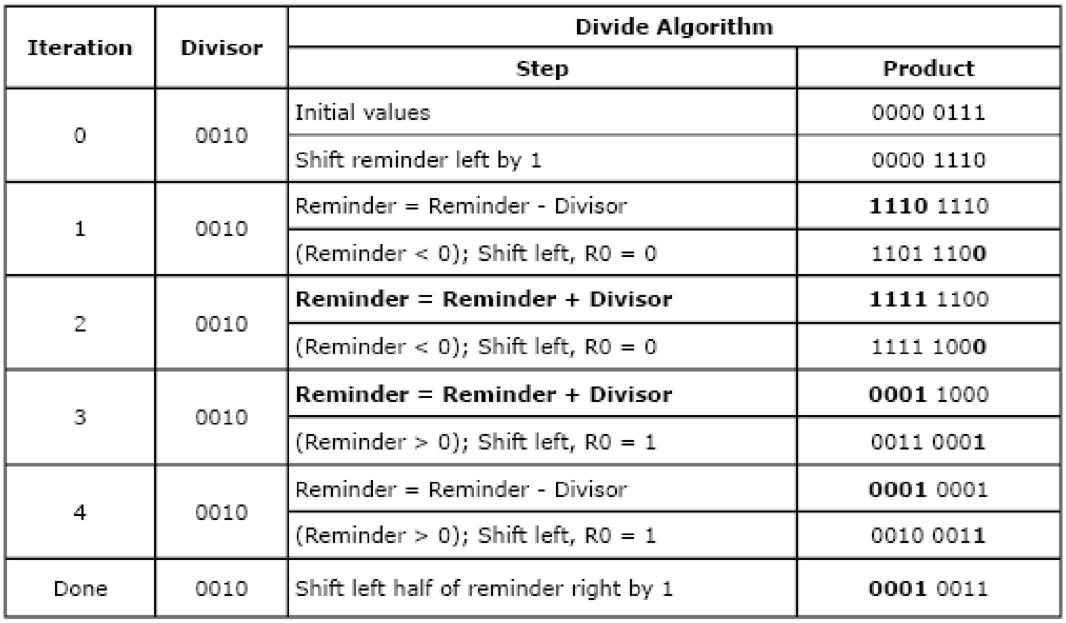
- Note that this addition is performed whenever the reminder is negative!

So, what exactly are we doing when the reminder is negative?

- We have a certain reminder: R (R < 0)
- We add the divisor back to it: R + D
- We shift the result left by 1:   2*(R + D) = 2*R + 2*D
- We subtract the divisor again in the next step: 2*R + 2*D - D = 2*R + D

- Equivalent of left shifting the reminder R by 1 bit
- Add the divisor in the next step, instead of subtracting

Non-restoring division example



Iteration	Divisor	Divide Algorithm	
		Step	**Product**
0	0010	Initial values	0000 0111
		Shift reminder left by 1	0000 1110
1	0010	Reminder = Reminder - Divisor	**1110** 1110
		(Reminder < 0); Shift left, R0 = 0	1101 110**0**
2	0010	**Reminder = Reminder + Divisor**	**1111** 1100
		(Reminder < 0); Shift left, R0 = 0	1111 100**0**
3	0010	**Reminder = Reminder + Divisor**	**0001** 1000
		(Reminder > 0); Shift left, R0 = 1	0011 000**1**
4	0010	Reminder = Reminder - Divisor	**0001** 0001
		(Reminder > 0); Shift left, R0 = 1	0010 001**1**
Done	0010	Shift left half of reminder right by 1	**0001** 0011

UNIT III
CONTROL DESIGN

HARDWIRED CONTROL:

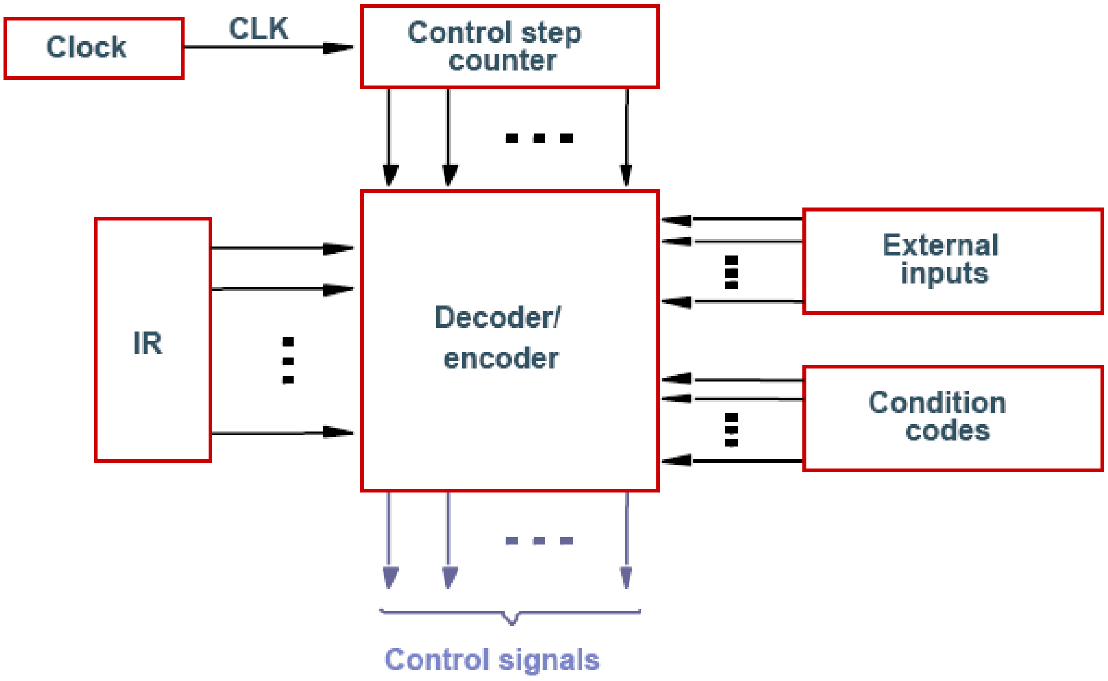


Figure 7.10. Control unit organization.

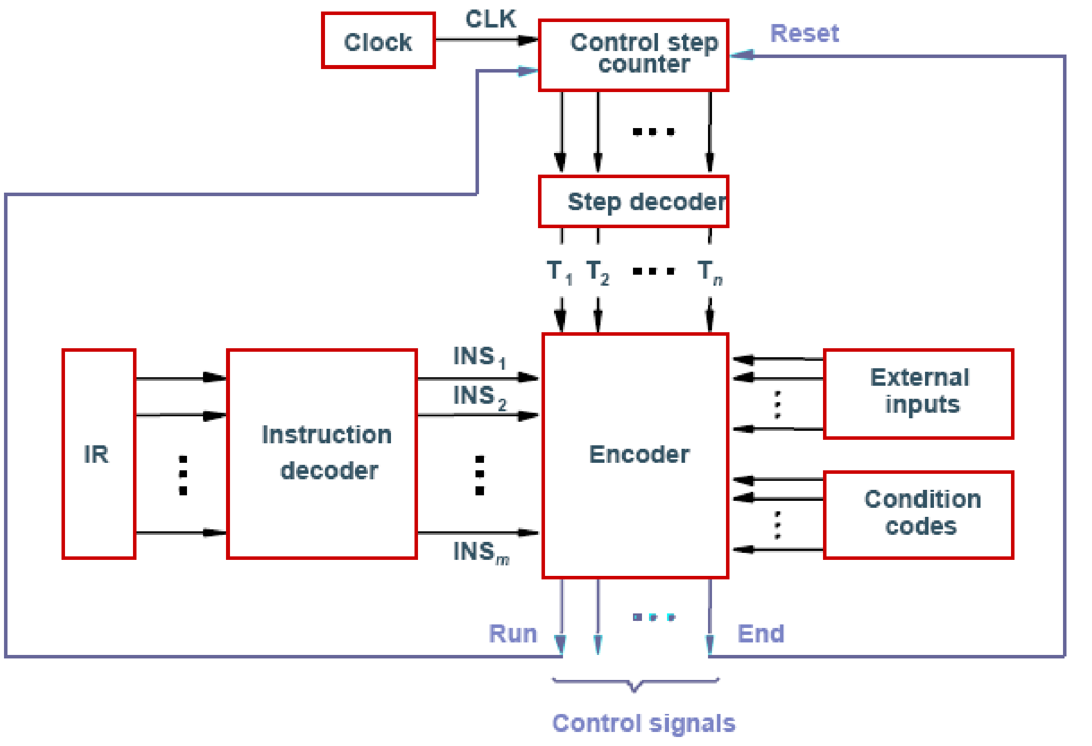
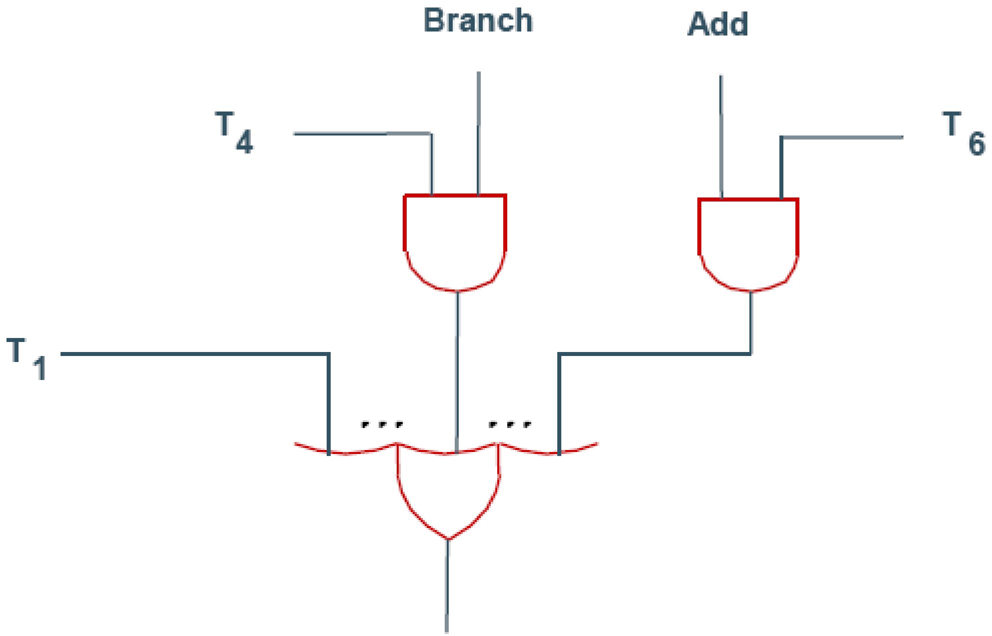
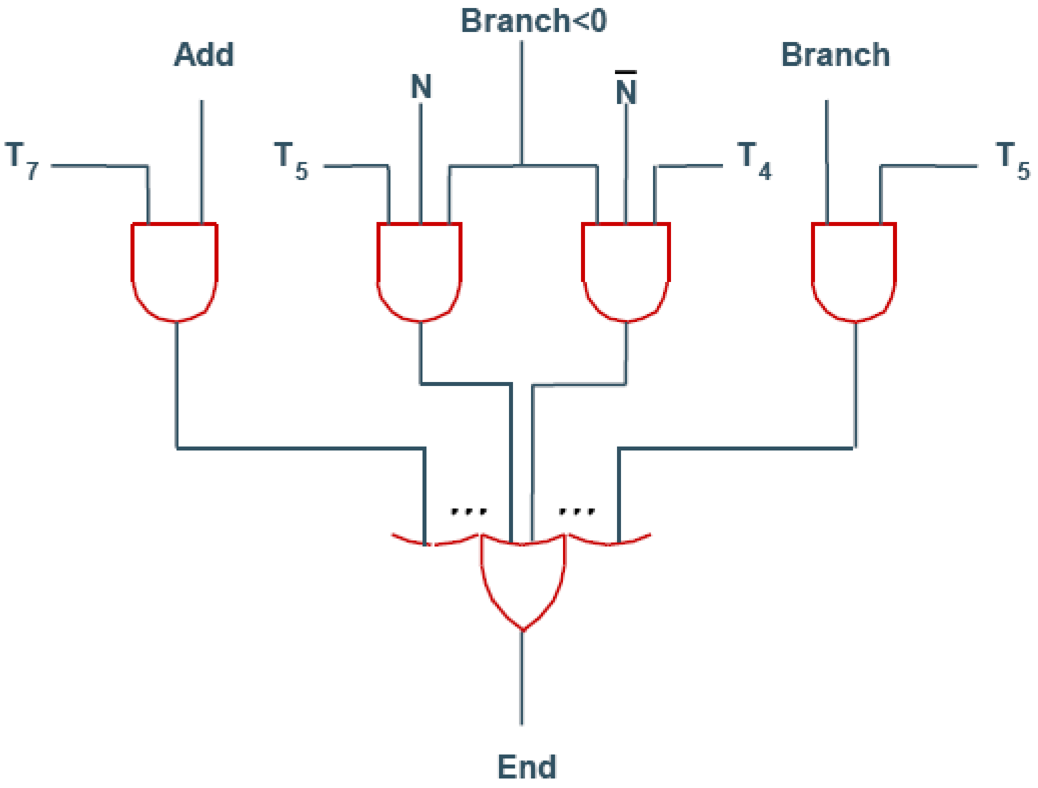


Figure 7.11.Separation of the decoding and encoding functions.



Generation of the Z_{in} control signal for the processor



Generation of the End control signal

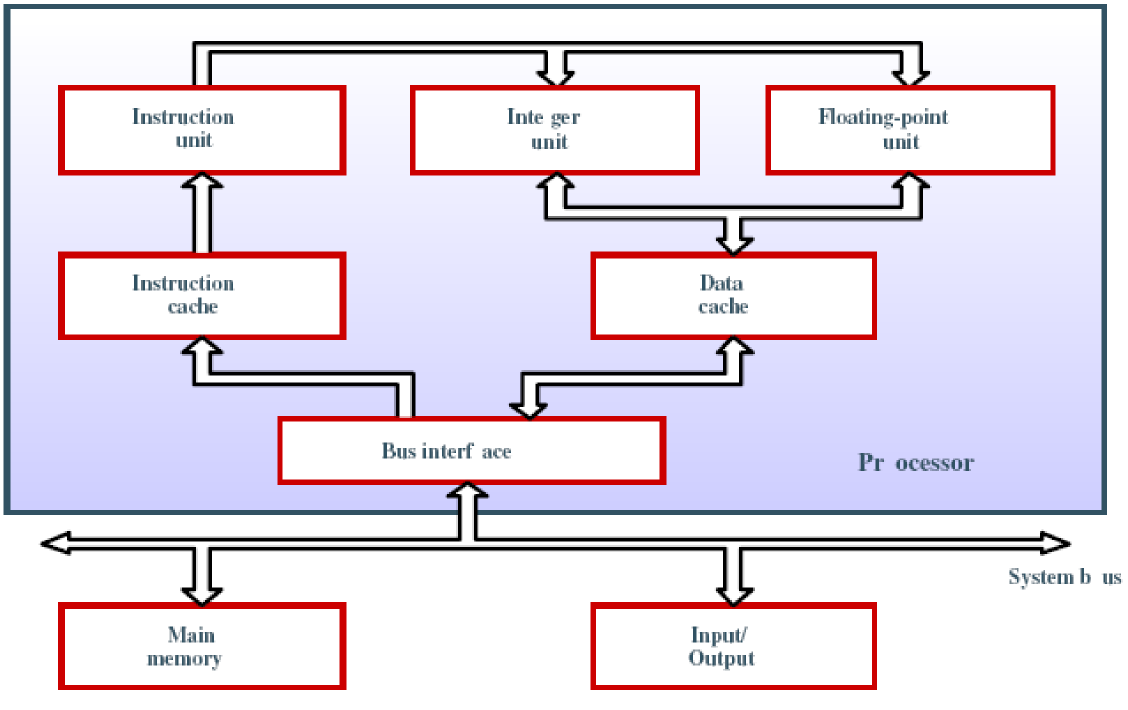


Figure 7.14. Block diagram of a complete processor

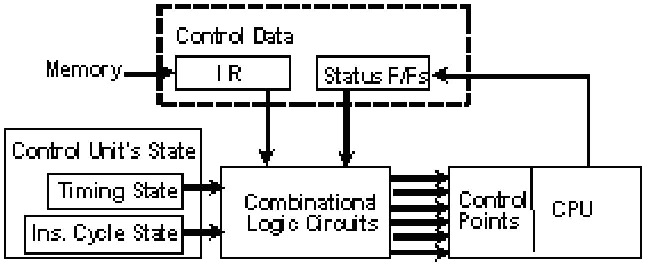
MICROPROGRAMMED CONTROL

- Control Memory
- Sequencing Microinstructions
- Microprogram Example
- Design of Control Unit
- Microinstruction Format
- Nanostorage and Nanoprogram

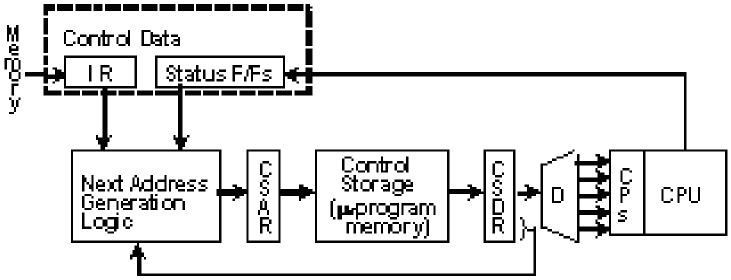
COMPARISON OF CONTROL UNIT IMPLEMENTATIONS

Control Unit Implementation

Combinational Logic Circuits (Hard-wired)



Microprogram


Memory
Control Data
I R
Status F/Fs
Next Address Generation Logic
CSAR
Control Storage (μ-program memory)
CSDR
D
CPs
CPU

TERMINOLOGY

Microprogram
- Program stored in memory that generates all the control signals required to execute the instruction set correctly
- Consists of microinstructions

Microinstruction
- Contains a control word and a sequencing word
  Control Word - All the control information required for one clock cycle
  Sequencing Word - Information needed to decide
  the next microinstruction address
- Vocabulary to write a microprogram

Control Memory(Control Storage: CS)
- Storage in the microprogrammed control unit to store the microprogram

Writeable Control Memory(Writeable Control Storage:WCS)
- CS whose contents can be modified
  -> Allows the microprogram can be changed
  -> Instruction set can be changed or modified

Dynamic Microprogramming
- Computer system whose control unit is implemented with a microprogram in WCS
- Microprogram can be changed by a systems programmer or a user

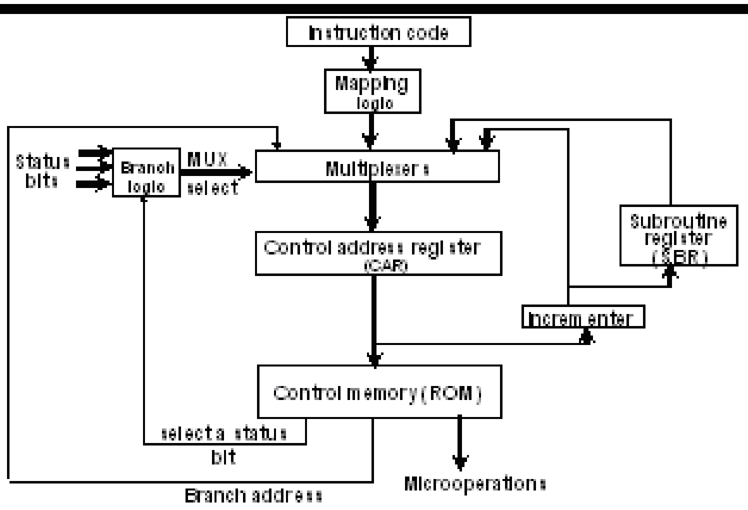
TERMINOLOGY

Sequencer (Microprogram Sequencer)

A Microprogram Control Unit that determines the Microinstruction Address to be executed in the next clock cycle

- In-line Sequencing
- Branch
- Conditional Branch
- Subroutine
- Loop
- Instruction OP-code mapping

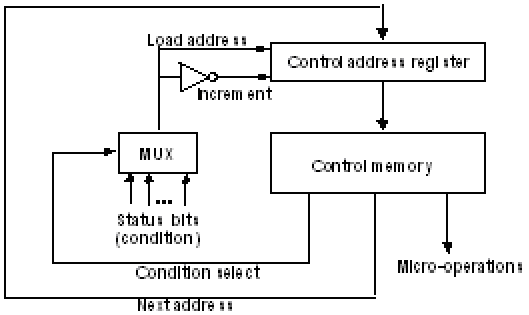
MICROINSTRUCTION SEQUENCING



Sequencing Capabilities Required in a Control Storage

- Incrementing of the control address register
- Unconditional and conditional branches
- A mapping process from the bits of the machine instruction to an address for control memory
- A facility for subroutine call and return

CONDITIONAL BRANCH



Conditional Branch

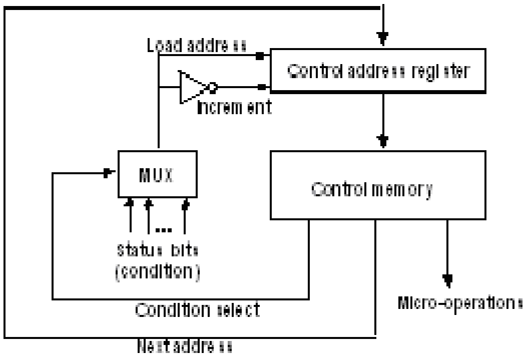
If *Condition* is true, then *Branch* (address from
the next address field of the current microinstruction)
else *Fall Through*

Conditions to Test: O(overflow), N(negative),
Z(zero), C(carry), etc.

Unconditional Branch

Fixing the value of one status bit at the input of the multiplexer to 1

CONDITIONAL BRANCH



Conditional Branch

If *Condition* is true, then *Branch* (address from
the next address field of the current microinstruction)
else *Fall Through*

Conditions to Test: O(overflow), N(negative),
Z(zero), C(carry), etc.

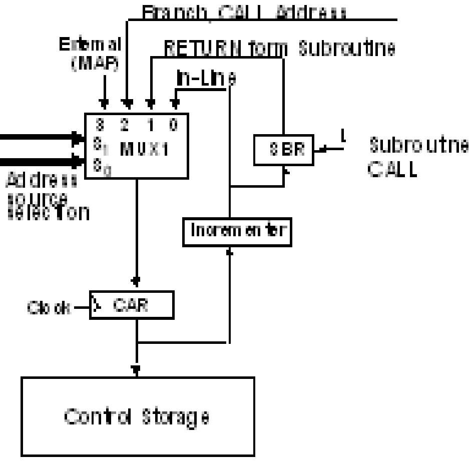
Unconditional Branch

Fixing the value of one status bit at the input of the multiplexer to 1

MICROPROGRAM SEQUENCER

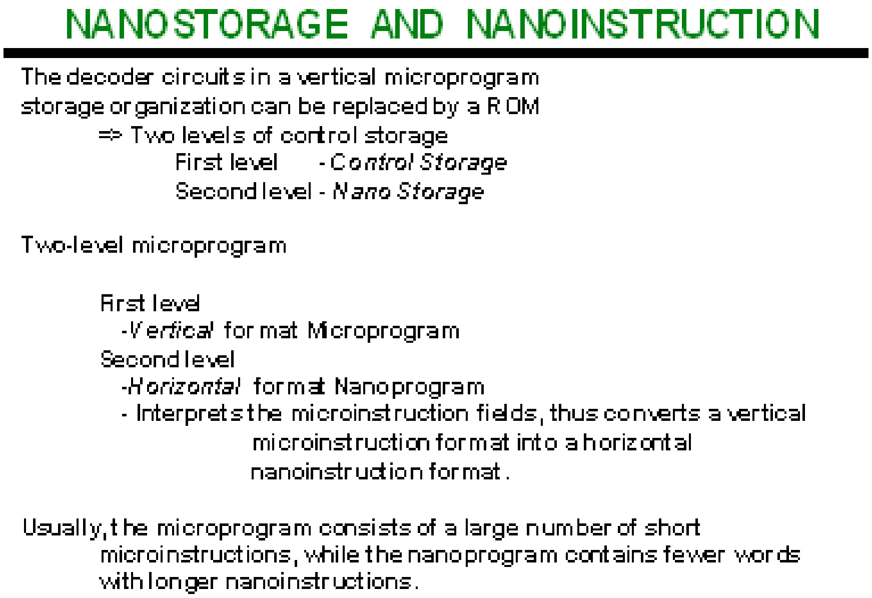
- NEXT MICROINSTRUCTION ADDRESS LOGIC -

S_1S_0	Address Source
00	CAR + 1, In-Line
01	SBR RETURN
10	CS(AD), Branch or CALL
11	MAP



MUX-1 selects an address from one of four sources and routes it into a CAR

- In-Line Sequencing → CAR + 1
- Branch, Subroutine Call → CS(AD)
- Return from Subroutine → Output of SBR
- New Machine instruction → MAP



NANOSTORAGE AND NANOINSTRUCTION

The decoder circuits in a vertical microprogram storage organization can be replaced by a ROM

=> Two levels of control storage

First level - *Control Storage*

Second level - *Nano Storage*

Two-level microprogram

First level

-*Vertical* format Microprogram

Second level

-*Horizontal* format Nanoprogram

- Interprets the microinstruction fields, thus converts a vertical microinstruction format into a horizontal nanoinstruction format.

Usually, the microprogram consists of a large number of short microinstructions, while the nanoprogram contains fewer words with longer nanoinstructions.

Pipelining

What is Pipelining?

The Pipeline Defined

John Hayes provides a definition of a pipeline as it applies to a computer processor.

"A pipeline processor consists of a sequence of processing circuits, called segments or stages, through which a stream of operands can be passed.

"Partial processing of the operands takes place in each segment.

"... a fully processed result is obtained only after an operand set has passed through the entire pipeline."

In everyday life, people do many tasks in stages. For instance, when we do the laundry, we place a load in the washing machine. When it is done, it is transferred to the dryer and another load is placed in the washing machine. When the first load is dry, we pull it out for folding or ironing, moving the second load to the dryer and start a third load in the washing machine. We proceed with folding or ironing of the first load while the second and third loads are being dried and washed, respectively. We may have never thought of it this way but we do laundry by **pipeline processing**.

A Pipeline

is a series of stages, where some work is done at each stage. The work is not finished until it has passed through all stages.

Let us review Hayes' definition as it pertains to our laundry example. The washing machine is one "sequence of processing circuits" or a **stage**. The second is the dryer. The third is the folding or ironing stage.

Partial processing takes place in each stage. We certainly aren't done when the clothes leave the washer. Nor when they leave the dryer, although we're getting close. We must take the third step and fold (if we're lucky) or iron the cloths. The "fully processed result" is obtained only after the operand (the load of clothes) has passed through the entire pipeline.

We are often taught to take a large task and to divide it into smaller pieces. This may make a unmanageable complex task into a series of more tractable smaller steps. In the case of manageable tasks such as the laundry example, it allows us to speed up the task by doing it in overlapping steps.

This is the key to pipelining: Division of a larger task into smaller **overlapping** tasks.

"A significant aspect of our civilization is the division of labor. Major engineering achievements are based on subdividing the total work into individual tasks which can be handled despite their inter-dependencies.

"Overlap and pipelining are essentially operation management techniques based on job sub-divisions under a precedence constraint."

Types of Pipelines

Instructional pipeline

where different stages of an instruction fetch and execution are handled in a pipeline.

Arithmetic pipeline

where different stages of an arithmetic operation are handled along the stages of a pipeline.

The above definitions are correct but are based on a narrow perspective, consider only the central processor. There are other type of computing pipelines. Pipelines are used to compress and transfer video data. Another is the use of specialized hardware to perform graphics display tasks. Discussing graphics displays, Ware Myers wrote:

"...the pipeline concept ... transforms a model of some object into representations that successively become more machine-dependent and finally results in an image upon a particular screen.

This example of pipelining fits the definitions from Hayes and Chen but not the categories offered by Tabaz. These broader categories are beyond the scope of this paper and are mentioned only to alert the reader that different authors mean different things when referring to pipelining.

Disadvantages

There are two disadvantages of pipeline architecture. The first is complexity. The second is the inability to continuously run the pipeline at full speed, i.e. the pipeline **stalls**.

Let us examine why the pipeline cannot run at full speed. There are phenomena called pipeline hazards which disrupt the smooth execution of the pipeline. The resulting delays in the pipeline flow are called bubbles. These pipeline hazards include

- structural hazards from hardware conflicts
- data hazards arising from data dependencies
- control hazards that come about from branch, jump, and other control flow changes

These issues can and are successfully dealt with. But detecting and avoiding the hazards leads to a considerable increase in hardware complexity. The control paths controlling the gating between stages can contain more circuit levels than the data paths being controlled. In 1970, this complexity is one reason that led Foster to call pipelining **"still-controversial"** .

The one major idea that is still controversial is "instruction look-ahead" [pipelining]...

Why then the controversy? First, there is a considerable increase in hardware complexity [...]

The second problem [...] when a branch instruction comes along, it is impossible to know in advance of execution which path the program is going to take and, if the machine guesses wrong, **all the partially processed instructions in the pipeline are useless and must be replaced [...]**

In the second edition of Foster's book, published 1976, this passage was gone. Apparently, Foster felt that pipelining was no longer controversial.

Doran also alludes to the nature of the problem. The model of pipelining is "amazingly simple" while the implementation is "very complex" and has many complications.

Because of the multiple instructions that can be in various stages of execution at any given moment in time, handling an interrupt is one of the more complex tasks. In the IBM 360, this can lead to several instructions executing after the interrupt is signaled, resulting in an **imprecise interrupt**. An imprecise interrupt can result from an instruction exception and precise address of the instruction causing the exception may not be known! This led Myers to criticize pipelining, referring to the imprecise interrupt as an "architectural nuisance". He stated that it was not an advance in computer architecture but an improvement in implementation that could be viewed as **a step backward**.

In retrospect, most of Myers' book *Advances in Computer Architecture* dealt with his concepts for improvements in computer architecture that would be termed CISC today. With the benefits of hindsight, we can see that pipelining is here today and that most of the new CPUs are in the RISC class. In fact, Myers is one of the co-architects of Intel's series of 32-bit RISC microprocessors. This processor is fully pipelined. I suspect that Myers no longer considers pipelining a step backwards.

The difficulty arising from imprecise interrupts should be viewed as a complexity to be overcome, not as an inherent flaw in pipelining. Doran explains how the B7700 carries the address of the instruction through the pipeline, so that any exception that the instruction may raise can be precisely located and not generate an imprecise interrupt

An **instruction pipeline** is a technique used in the design of computers and other digital electronic devices to increase their instruction throughput (the number of instructions that can be executed in a unit of time).

The fundamental idea is to split the processing of a computer instruction into a series of independent steps, with storage at the end of each step. This allows the computer's control circuitry to issue instructions at the processing rate of the slowest step, which is much faster than the time needed to perform all steps at once. The term pipeline refers to the fact that each step is carrying data at once (like water), and each step is connected to the next (like the links of a pipe.)

The origin of pipelining is thought to be either the project or the project. The IBM Stretch Project proposed the terms, "Fetch, Decode, and Execute" that became common usage.

Most modern CPUs are driven by a clock. The CPU consists internally of logic and memory (flip flops). When the clock signal arrives, the flip flops take their new value and the logic then requires a period of time to decode the new values. Then the next clock pulse arrives and the flip flops again take their new values, and so on. By breaking the logic into smaller pieces and inserting flip flops between the pieces of logic, the delay before the logic gives valid outputs is reduced. In this way the clock period can be reduced. For example, the RISC pipeline is broken into five stages with a set of flip flops between each stage.

1. Instruction fetch
2. Instruction decode and register fetch
3. Execute
4. Memory access
5. Register write back

Hazards: When a programmer (or compiler) writes assembly code, they make the assumption that each instruction is executed before execution of the subsequent instruction is begun. This assumption is invalidated by pipelining. When this causes a program to behave incorrectly, the situation is known as a hazard. Various techniques for resolving hazards such as forwarding and stalling exist.

A non-pipeline architecture is inefficient because some CPU components (modules) are idle while another module is active during the instruction cycle. Pipelining does not completely cancel out idle time in a CPU but making those modules work in parallel improves program execution significantly.

Processors with pipelining are organized inside into stages which can semi-independently work on separate jobs. Each stage is organized and linked into a 'chain' so each stage's output is fed to another stage until the job is done. This organization of the processor allows overall processing time to be significantly reduced.

A deeper pipeline means that there are more stages in the pipeline, and therefore, fewer logic gates in each pipeline. This generally means that the processor's frequency can be increased as the cycle time is lowered. This happens because there are fewer components in each stage of the pipeline, so the propagation delay is decreased for the overall stage.

Unfortunately, not all instructions are independent. In a simple pipeline, completing an instruction may require 5 stages. To operate at full performance, this pipeline will need to run 4 subsequent independent instructions while the first is completing. If 4 instructions that do not depend on the output of the first instruction are not available, the pipeline control logic must insert a stall or wasted clock cycle into the pipeline until the dependency is resolved. Fortunately, techniques such as forwarding can significantly reduce the cases where stalling is required. While pipelining can in theory increase performance over an unpipelined core by a factor of the number of stages (assuming the clock frequency also scales with the number of stages), in reality, most code does not allow for ideal execution.

Hazard (computer architecture)

In computer architecture, a **hazard** is a potential problem that can happen in a pipelined processor. It refers to the possibility of erroneous computation when a CPU tries to simultaneously execute multiple instructions which exhibit data dependence. There are typically three types of hazards: data hazards, structural hazards, and branching hazards (control hazards).

Instructions in a pipelined processor are performed in several stages, so that at any given time several instructions are being executed, and instructions may not be completed in the desired order.

A hazard occurs when two or more of these simultaneous (possibly out of order) instructions conflict.

- 1 Data hazards
  - 1.1 RAW - Read After Write
  - 1.2 WAR - Write After Read
  - 1.3 WAW - Write After Write
- 2 Structural hazards
- 3 Branch (control) hazards
- 4 Eliminating hazards
  - 4.1 Eliminating data hazards
  - 5.1 Eliminating branch hazards

Data hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data and control hazards. **Data hazards** occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on the unpipelined machine.

Consider the pipelined execution of these instructions:

		1	2	3	4	5	6	7	8	9
ADD	**R1**, R2, R3	IF	ID	EX	MEM	**WB**				
SUB	R4, R5, **R1**		IF	**ID**$_{\mathbf{sub}}$	EX	MEM	WB			
AND	R6, **R1,** R7			IF	**ID**$_{\mathbf{and}}$	EX	MEM	WB		
OR	R8, **R1**, R9				IF	**ID**$_{\mathbf{or}}$	EX	MEM	WB	
XOR	R10,**R1**,R11					IF	**ID**$_{\mathbf{xor}}$	EX	MEM	WB

All the instructions after the ADD use the result of the ADD instruction (in R1). The ADD instruction writes the value of R1 in the WB stage (shown black), and the **SUB** instruction reads the value during ID stage (**ID**$_{\mathbf{sub}}$). This problem is called ***a data hazard***. Unless precautions are taken to prevent it, the SUB instruction will read the wrong value and try to use it.

The **AND** instruction is also affected by this data hazard. The write of R1 does not complete until the end of cycle 5 (shown black). Thus, the AND instruction that reads the registers during cycle 4 (**ID$_{\textbf{and}}$**) will receive the wrong result.

The **OR** instruction can be made to operate without incurring a hazard by a simple implementation technique. ***The technique*** is to perform register file reads in the second half of the cycle, and writes in the first half. Because both **WB** for ADD and **ID$_{\textbf{or}}$** for OR are performed in one cycle 5, the write to register file by ADD will perform in the first half of the cycle, and the read of registers by OR will perform in the second half of the cycle.

The **XOR** instruction operates properly, because its register read occur in cycle 6 after the register write by ADD.

The next page discusses forwarding, a technique to eliminate the stalls for the hazard involving the SUB and AND instructions.

We will also classify the data hazards and consider the cases when stalls can not be eliminated. We will see what compiler can do to schedule the pipeline to avoid stalls.

A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap caused by pipelining would change the order of access to an operand. Our example hazards have all been with register operands, but it is also possible to create a dependence by writing and reading the same memory location. In DLX pipeline, however, memory references are always kept in order, preventing this type of hazard from arising.

All the data hazards discussed here involve registers within the CPU. By convention, ***the hazards are named by the ordering in the program that must be preserved by the pipeline.***

RAW (read after write)
WAW (write after write)
WAR (write after read)

Consider two instructions ***i*** and ***j***, with ***i*** occurring before ***j***. The possible data hazards are:

RAW (read after write) - ***j*** *tries to read a source before* ***i*** *writes it, so* ***j*** *incorrectly gets the old value.*

This is the most common type of hazard and the kind that we use forwarding to overcome.

WAW (write after write) - ***j*** *tries to write an operand before it is written by* ***i****. The writes end up being performed in the wrong order, leaving the value written by* ***i*** *rather than the value written by* ***j*** *in the destination.*

This hazard is present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled. The DLX integer pipeline writes a register only in WB and avoids this class of hazards.

WAW hazards would be possible if we made the following two changes to the DLX pipeline:

move write back for an ALU operation into the MEM stage, since the data value is available by then. suppose that the data memory access took two pipe stages.

Here is a sequence of two instructions showing the execution in this revised pipeline, highlighting the pipe stage that writes the result:

LW R1, 0(R2)	IF	ID	EX	MEM1	MEM2	**WB**
ADD R1, R2, R3		IF	ID	EX	**WB**	

Unless this hazard is avoided, execution of this sequence on this revised pipeline will leave the result of the first write (the LW) in R1, rather than the result of the ADD.

Allowing writes in different pipe stages introduces other problems, since two instructions can try to write during the same clock cycle. The DLX FP pipeline , which has both writes in different stages and different pipeline lengths, will deal with both write conflicts and WAW hazards in detail.

WAR (write after read) *-* ***j*** *tries to write a destination before it is read by* ***i*** *, so* ***i*** *incorrectly gets the new value.*

This can not happen in our example pipeline because all reads are early (in ID) and all writes are late (in WB). This hazard occurs when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline.

Because of the natural structure of a pipeline, which typically reads values before it writes results, such hazards are rare. Pipelines for complex instruction sets that support autoincrement addressing and require operands to be read late in the pipeline could create a WAR hazards.

If we modified the DLX pipeline as in the above example and also read some operands late, such as the source value for a store instruction, a WAR hazard could occur. Here is the pipeline timing for such a potential hazard, highlighting the stage where the conflict occurs:

SW R1, 0(R2)	IF	ID	EX	MEM1	**MEM2**	WB
ADD R2, R3, R4		IF	ID	EX	**WB**	

If the SW reads R2 during the second half of its MEM2 stage and the Add writes R2 during the first half of its WB stage, the SW will incorrectly read and store the value produced by the ADD.

RAR (read after read) *- this case is not a hazard :).*

Structural hazards

A structural hazard occurs when a part of the processor's hardware is needed by two or more instructions at the same time. A structural hazard might occur, for instance, if a program were to execute a branch instruction followed by a computation instruction. Because they are executed in parallel, and because branching is typically slow (requiring a comparison, program counter-related computation, and writing to registers), it is quite possible (depending on architecture) that the computation instruction and the branch instruction will both require the ALU (arithmetic logic unit) at the same time.

When a machine is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.

If some combination of instructions cannot be accommodated because of a resource conflict, the machine is said to have a structural hazard.

Common instances of structural hazards arise when

- Some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle
- Some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute.

Example1:
a machine may have only one register-file write port, but in some cases the pipeline might want to perform two writes in a clock cycle.

Example2:
a machine has shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference(load), it will conflict with the instruction reference for a later instruction (instr 3):

Clock cycle number								
Instr	1	2	3	4	5	6	7	8
Load	IF	ID	EX	MEM	WB			
Instr 1		IF	ID	EX	MEM	WB		
Instr 2			IF	ID	EX	MEM	WB	
Instr 3				IF	ID	EX	MEM	WB

To resolve this, we stall the pipeline for one clock cycle when a data-memory access occurs. The effect of the stall is actually to occupy the resources for that instruction slot. The following table shows how the stalls are actually implemented.

Clock cycle number									
Instr	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	**MEM**	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		

Stall				bubble	bubble	bubble	bubble	bubble	
Instr 3					**IF**	ID	EX	MEM	WB

Instruction 1 assumed not to be data-memory reference (load or store), otherwise Instruction 3 cannot start execution for the same reason as above.

To simplify the picture it is also commonly shown like this:

Clock cycle number									
Instr	1	2	3	4	5	6	7	8	9
Load	IF	ID	EX	MEM	WB				
Instr 1		IF	ID	EX	MEM	WB			
Instr 2			IF	ID	EX	MEM	WB		
Instr 3				stall	IF	ID	EX	MEM	WB

Introducing stalls degrades performance as we saw before. Why, then, would the designer allow structural hazards? There are two reasons:

To reduce cost. For example, machines that support both an instruction and a cache access every cycle (to prevent the structural hazard of the above example) require at least twice as much total memory.
To reduce the latency of the unit. The shorter latency comes from the lack of pipeline registers that introduce overhead.

Branch (control) hazards

Branching hazards (also known as control hazards) occur when the processor is told to branch - i.e., if a certain condition is true, then jump from one part of the instruction stream to another - not necessarily to the next instruction sequentially. In such a case, the processor cannot tell in advance whether it should process the next instruction (when it may instead have to move to a distant instruction).

This can result in the processor doing unwanted actions.

A ***cache miss***. A cache miss stalls all the instructions on pipeline both before and after the instruction causing the miss.

A ***hazard in pipeline.*** Eliminating a hazard often requires that some instructions in the pipeline to be allowed to proceed while others are delayed. When the instruction is stalled, all the instructions issued *later* than the stalled instruction are also stalled. Instructions issued *earlier* than the stalled instruction must continue, since otherwise the hazard will never clear.

A hazard causes pipeline bubbles to be inserted.The following table shows how the stalls are actually implemented. As a result, no new instructions are fetched during clock cycle 4, no instruction will finish during clock cycle 8.
In case of structural hazards:

Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				
Instr i+2			IF	ID	EX	MEM	WB			
Stall				bubble	bubble	bubble	bubble	bubble		
Instr i+3					IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

To simplify the picture it is also commonly shown like this:

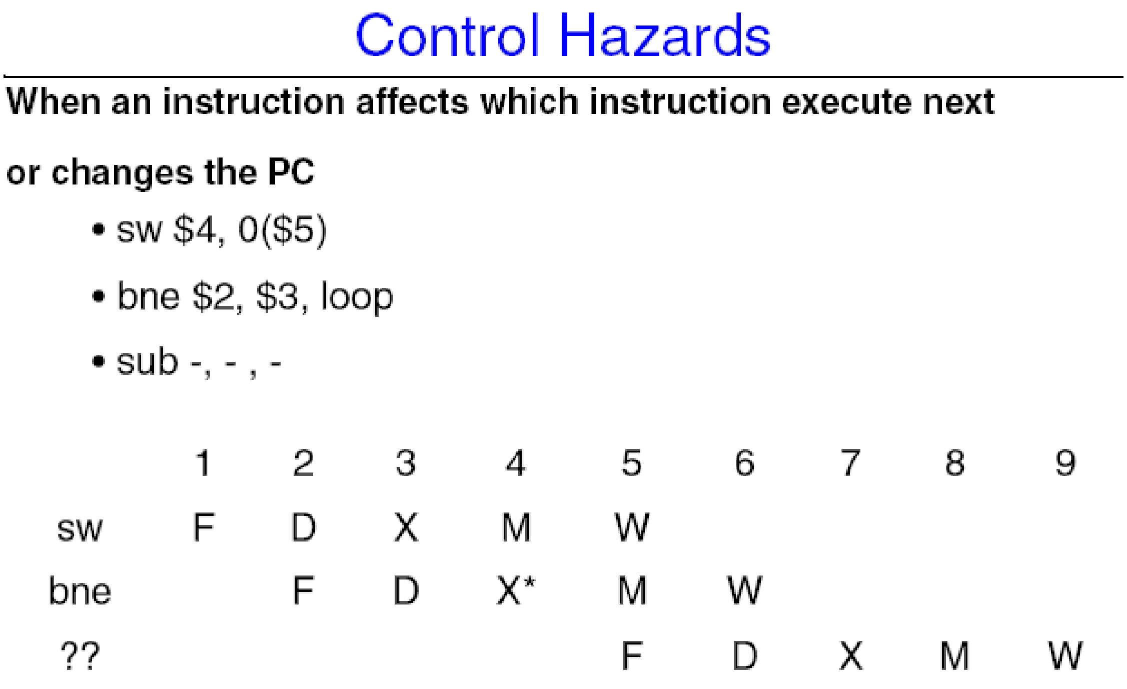
Clock cycle number										
Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	EX	MEM	WB				
Instr i+2			IF	ID	EX	MEM	WB			
Instr i+3				stall	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

In case of data hazards:

Clock cycle number										
Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	bubble	EX	MEM	WB			
Instr i+2			IF	bubble	ID	EX	MEM	WB		
Instr i+3				bubble	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB

which appears the same with stalls:

Clock cycle number										
Instr	1	2	3	4	5	6	7	8	9	10
Instr i	IF	ID	EX	MEM	WB					
Instr i+1		IF	ID	stall	EX	MEM	WB			
Instr i+2			IF	stall	ID	EX	MEM	WB		
Instr i+3				stall	IF	ID	EX	MEM	WB	
Instr i+4						IF	ID	EX	MEM	WB



Control Hazards

When an instruction affects which instruction execute next

or changes the PC

- sw $4, 0($5)
- bne $2, $3, loop
- sub -, - , -

	1	2	3	4	5	6	7	8	9
sw	F	D	X	M	W				
bne		F	D	X*	M	W			
??					F	D	X	M	W

Handling control hazards is very important
VAX e.g.,
• Emer and Clark report 39% of instr. change the PC
• Naive solution adds approx. 5 cycles every time
• *Or, adds 2 to CPI or ~20% increase*
DLX e.g.,
• H&P report 13% branches
• Naive solution adds 3 cycles per branch
• *Or, 0.39 added to CPI or ~30% increase*

Eliminating hazards

We can delegate the task of removing data dependencies to the compiler, which can fill in an appropriate number of `NOP` instructions between dependent instructions to ensure correct operation, or re-order instructions where possible.

Other methods include on-chip solutions such as:

- Scoreboarding method
- Tomasulo's method

There are several established techniques for either preventing hazards from occurring, or working around them if they do.

Bubbling the Pipeline

Bubbling the pipeline (a technique also known as a **pipeline break** or **pipeline stall**) is a method for preventing data, structural, and branch hazards from occurring. As instructions are fetched, control logic determines whether a hazard could/will occur. If this is true, then the control logic inserts NOPs into the pipeline. Thus, before the next instruction (which would cause the hazard) is executed, the previous one will have had sufficient time to complete and prevent the hazard. If the number of NOPs is equal to the number of stages in the pipeline, the processor has been cleared of all instructions and can proceed free from hazards. This is called **flushing the pipeline**. All forms of stalling introduce a delay before the processor can resume execution.

Eliminating data hazards

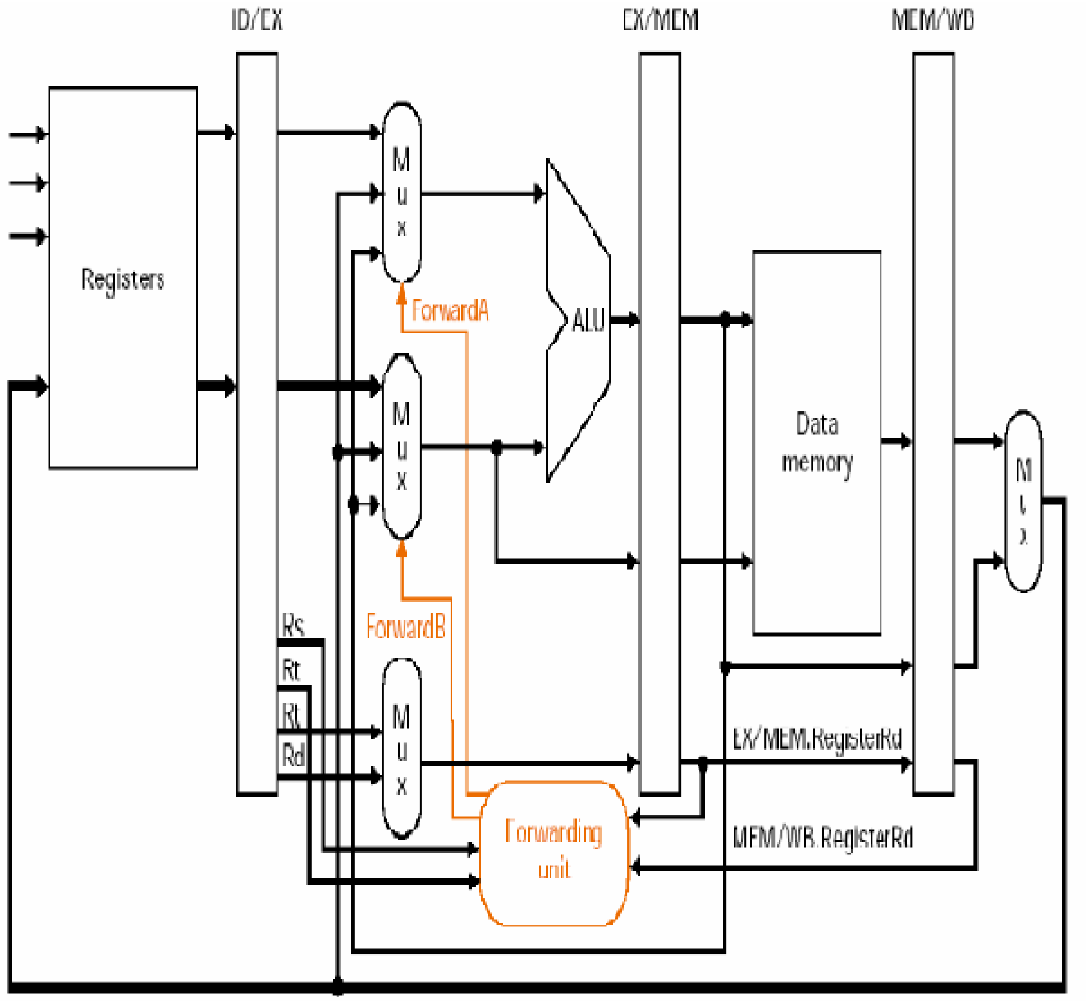
Forwarding

NOTE: *In the following examples, computed values are in* ***bold****, while Register numbers are not.*
Forwarding involves feeding output data into a previous stage of the pipeline. For instance, let's say we want to write the value 3 to register 1, (which already contains a 6), and then add 7 to register 1 and store the result in register 2, i.e.:
Instruction 0: Register 1 = ***6***
Instruction 1: Register 1 = ***3***
Instruction 2: Register 2 = Register 1 + ***7*** *=* ***10***
Following execution, register 2 should contain the value **10**. However, if Instruction 1 (write **3** to register 1) does not completely exit the pipeline before Instruction 2 starts execution, it means that Register 1 does not contain the value **3** when Instruction 2 performs its addition. In such an event, Instruction 2 adds **7** to the old value of register 1 (**6**), and so register 2 would contain **13** instead, i.e:
Instruction 0: Register 1 = ***6***
Instruction 1: Register 1 = ***3***
Instruction 2: Register 2 = Register 1 + ***7*** *=* ***13***
This error occurs because Instruction 2 reads Register 1 before Instruction 1 has committed/stored the result of its write operation to Register 1. So when Instruction 2 is reading the contents of Register 1, register 1 still contains **6**, *not* **3**.
Forwarding (described below) helps correct such errors by depending on the fact that the output of Instruction 1 (which is **3**) can be used by subsequent instructions *before* the value **3** is committed to/stored in Register 1.

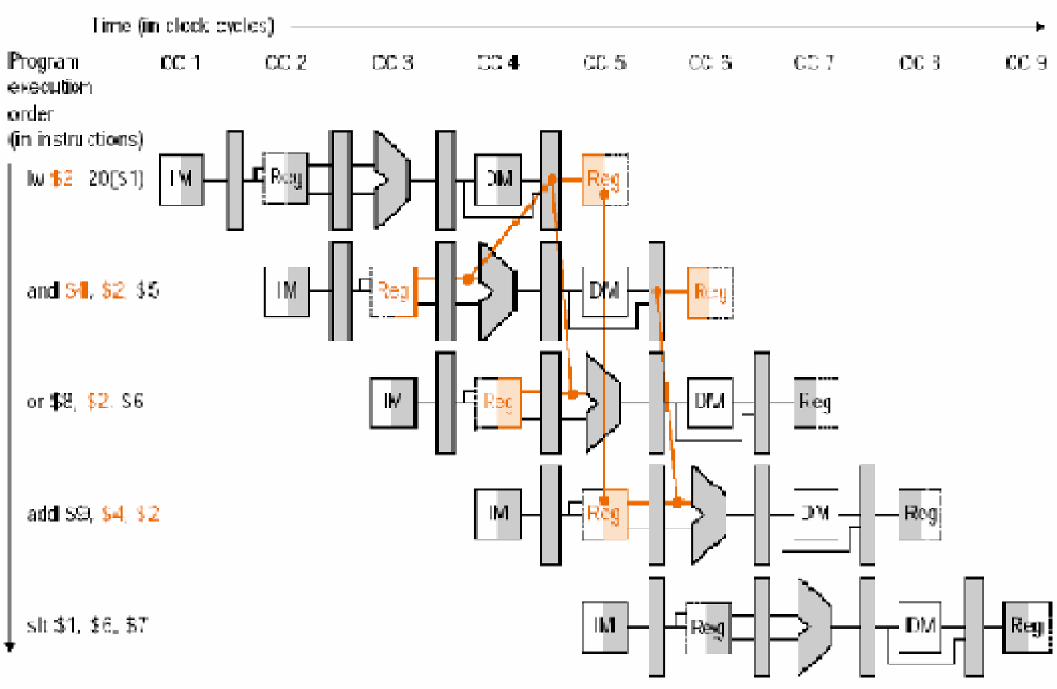
Forwarding is implemented by feeding back the output of an instruction into the previous stage(s) of the pipeline **as soon as the output of that instruction is available**. Forwarding applied to our example means that *we do not wait to commit/store the output of Instruction 1 in Register 1 (in this example, the output is* ***3****) before making that output available to the subsequent instruction (in this case, Instruction 2).* The effect is that Instruction 2 uses the correct (the more recent) value of Register 1: the commit/store was made immediately and not pipelined.

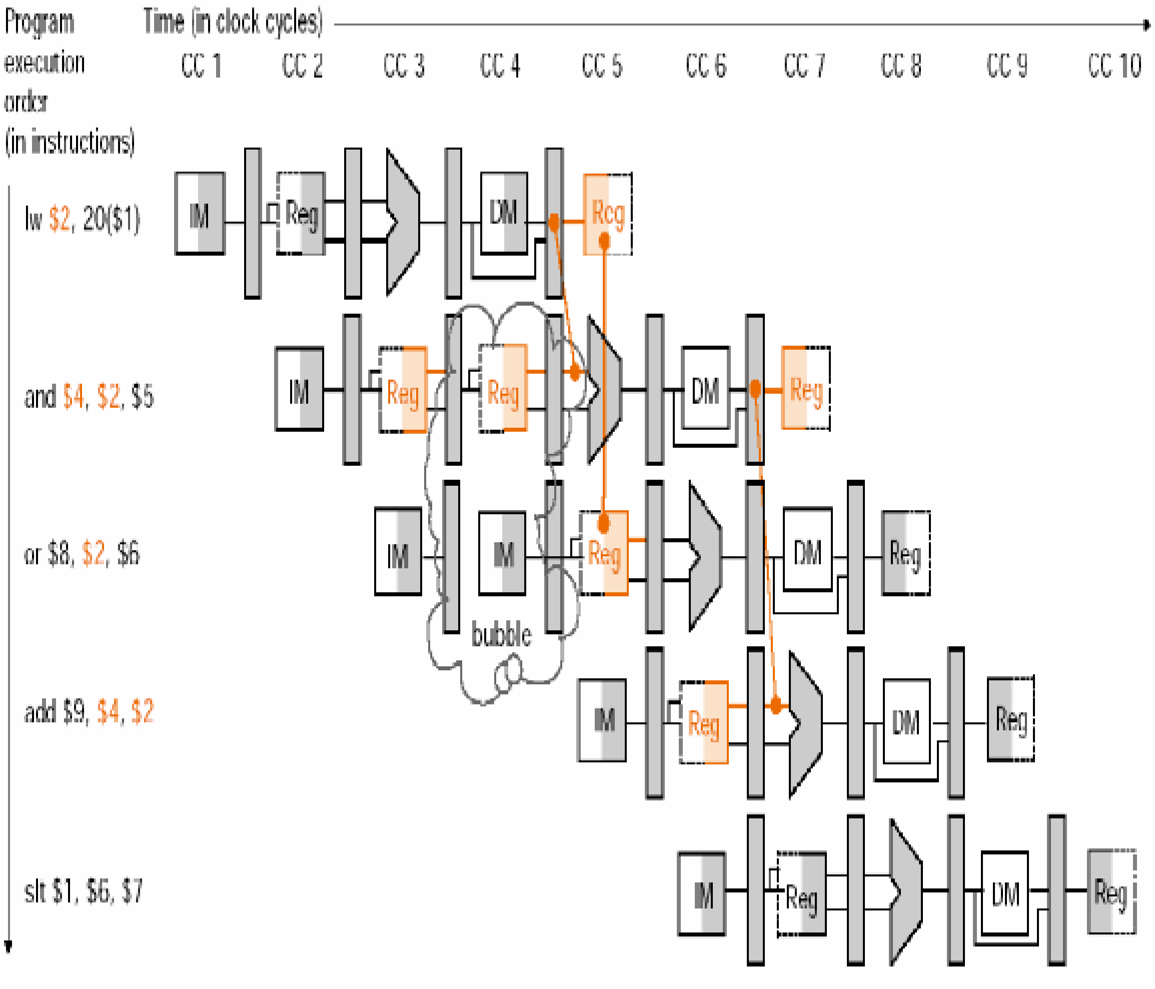
With forwarding enabled, the ID/EX[*clarification needed*] stage of the pipeline now has two inputs: the value read from the register specified (in this example, the value **6** from Register 1), and the new value of Register 1 (in this example, this value is **3**) which is sent from the next stage (EX/MEM)[*clarification needed*]. Additional control logic is used to determine which input to use.

Forwarding Unit



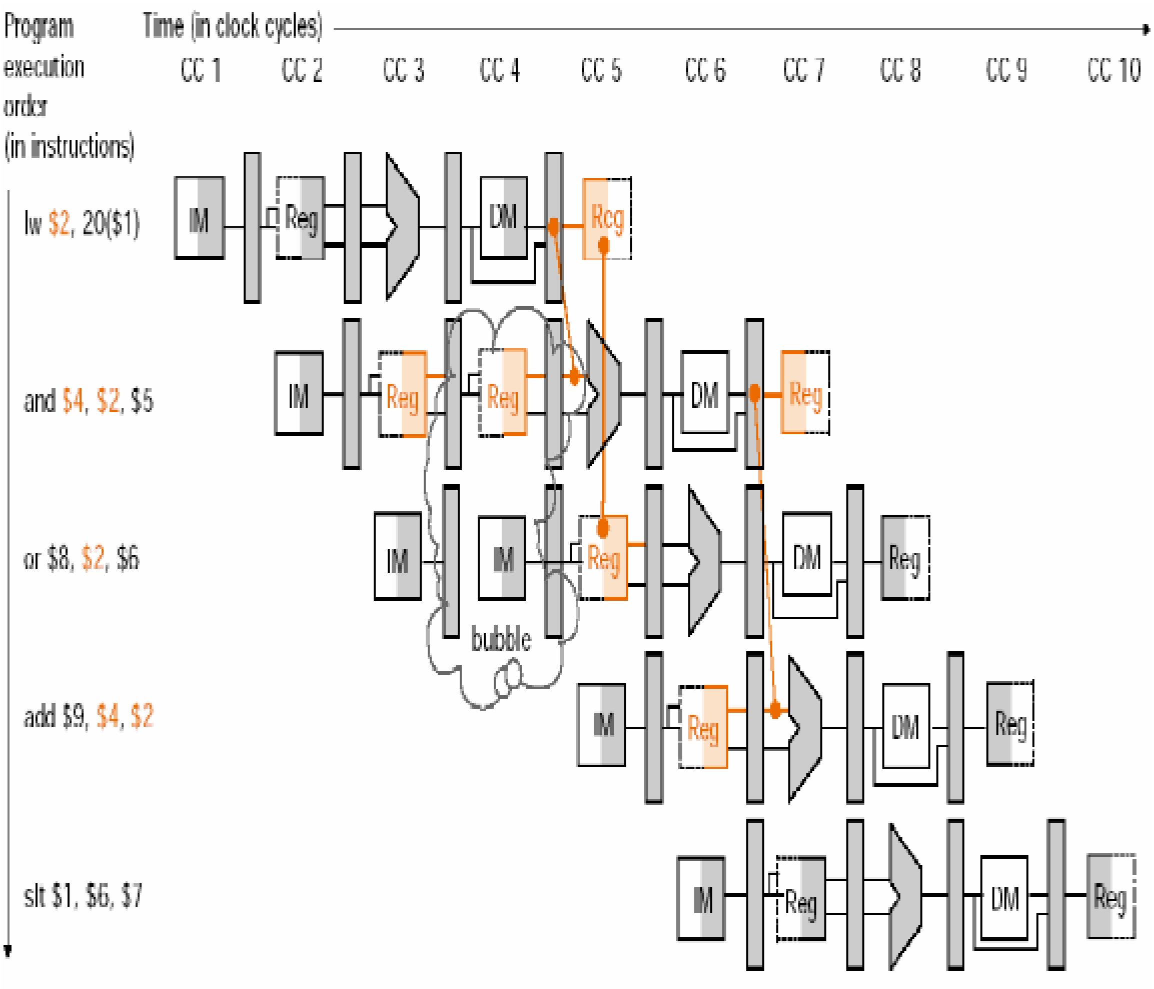
What About Load-Use Stall?




Program
execution
order
(in instructions)
Time (in clock cycles)
CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
CC 10
lw $2, 20($1)
and $4, $2, $5
or $8, $2, $6
add $9, $4, $2
slt $1, $6, $7
IM
Reg
DM
Rcg
bubble

What About Control Hazards?

(Predict-Not taken)



Reduce Branch Delay

Performance of Pipelines with Stalls

A stall causes the pipeline performance to degrade the ideal performance.

$$\textbf{Speedup from pipelining} = \frac{\textbf{Average instruction time unpipelined}}{\textbf{Average instruction time pipelined}}$$

$$= \frac{\textbf{CPI}_{\textbf{unpipelined}} * \textbf{Clock Cycle Time}_{\textbf{unpipelined}}}{\textbf{CPI}_{\textbf{pipelined}} * \textbf{Clock Cycle Time}_{\textbf{pipelined}}}$$

The ideal CPI on a pipelined machine is almost always 1. Hence, the pipelined CPI is

$$\begin{aligned}\textbf{CPI}_{\textbf{pipelined}} &= \textbf{Ideal CPI + Pipeline stall clock cycles per instruction}\\ &= \textbf{1 + Pipeline stall clock cycles per instruction}\end{aligned}$$

If we ignore the cycle time overhead of pipelining and assume the stages are all perfectly balanced, then the cycle time of the two machines are equal and

$$\textbf{Speedup} = \frac{\textbf{CPI}_{\textbf{unpipelined}}}{\textbf{1+ Pipeline stall cycles per instruction}}$$

If all instructions take the same number of cycles, which must also equal the number of pipeline stages (the depth of the pipeline) then unpipelined CPI is equal to the depth of the pipeline, leading to

$$\textbf{Speedup} = \frac{\textbf{Pipeline depth}}{\textbf{1 + Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of pipeline.

Nanoprogramming

- Second compromise: nanoprogramming

– Use a 2-level control storage organization

– Top level is a vertical format memory

» Output of the top level memory drives the address register of the bottom (nano-level) memory

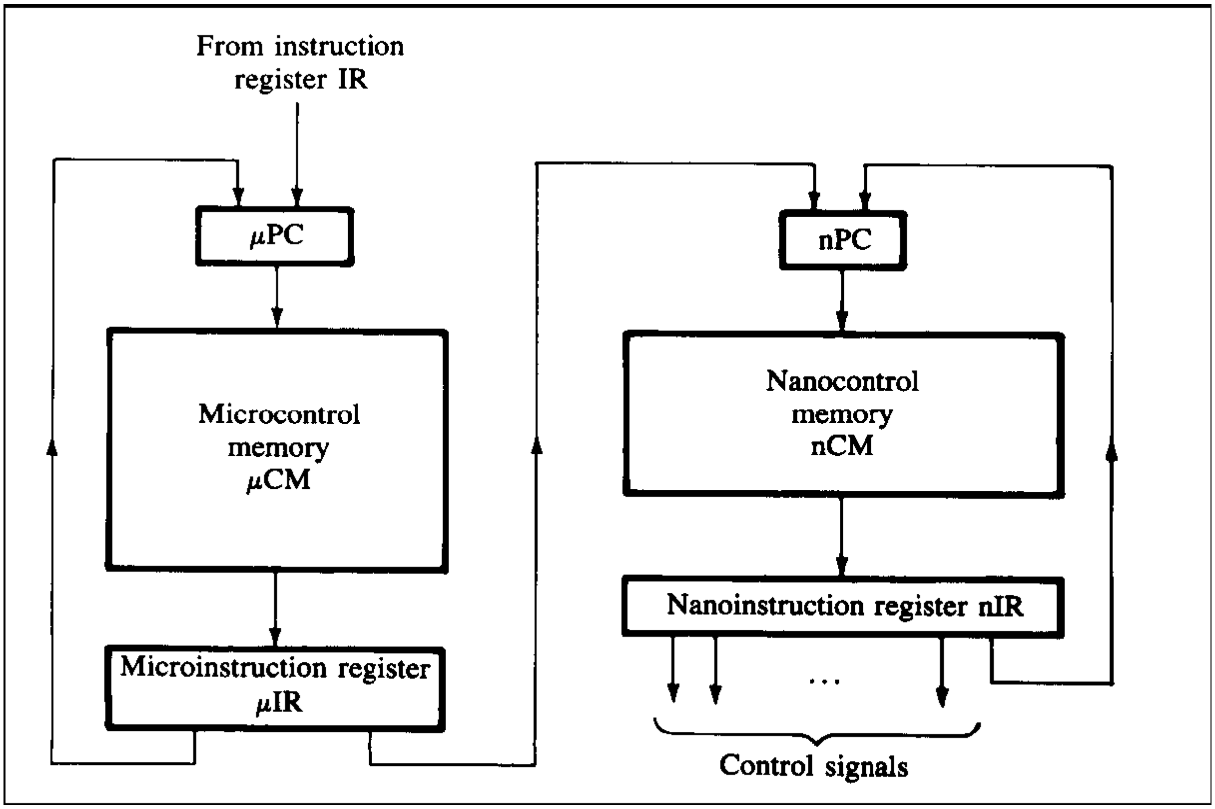
– Nanomemory uses the horizontal format

» Produces the actual control signal outputs

– The advantage to this approach is significant saving in control memory *size* (bits)

– Disadvantage is more complexity and slower operation (doing 2 memory accesses fro each microinstruction)

Nano programmed machine


From instruction register IR
μPC
Microcontrol memory μCM
Microinstruction register μIR
nPC
Nanocontrol memory nCM
Nanoinstruction register nIR
...
Control signals

- Example: Supppose that a system is being designed with 200 control points and 2048 microinstructions

– Assume that only 256 different combinations of control points are ever used

– A single-level control memory would require 2048x200=409,600 storage bits

- A nano programmed system would use

» Microstore of size 2048x8=16k

» Nanostore of size 256x200=51200

» Total size = 67,584 storage bits

- Nano programming has been used in many CISC microprocessors

Applications of Microprogramming

- Microprogramming application: emulation

– The use of a microprogram on one machine to execute programs originally written to run on another (different!) machine

– By changing the microcode of a machine, you can make it execute software from another machine

– Commonly used in the past to permit new machines to continue to run old software

» VAX11-780 had 2 “modes”

- Normal 11-780 mode
- Emulation mode for a PDP-11

– The Nanodata QM-1 machine was marketed with no native instruction set!

» Universal emulation engine

UNIT 4

Memory Organisaton

BASIC CONCEPTS:

Address space
– 16-bit : 216 = 64K mem. locations
– 32-bit : 232 = 4G mem. locations

- 40-bit : 240 = 1 T locations

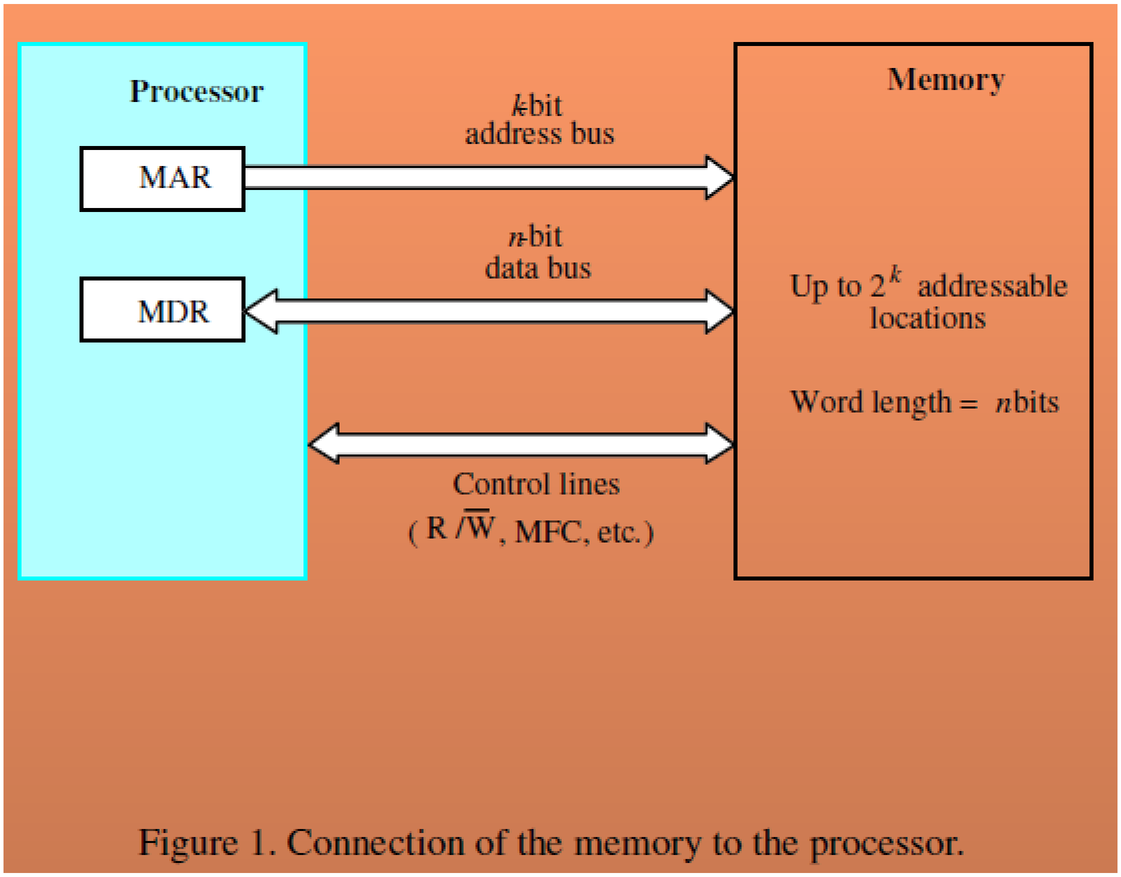


Figure 1. Connection of the memory to the processor.

Terminology:

Memory access time – time between Read and MFC signals
• *Memory cycle time* – min. time delay between initiation of two successive memory operations

Internal Organization of memory chips

- Form of an array
- Word line & bit lines
- 16x8 organization : 16 words of 8 bits each

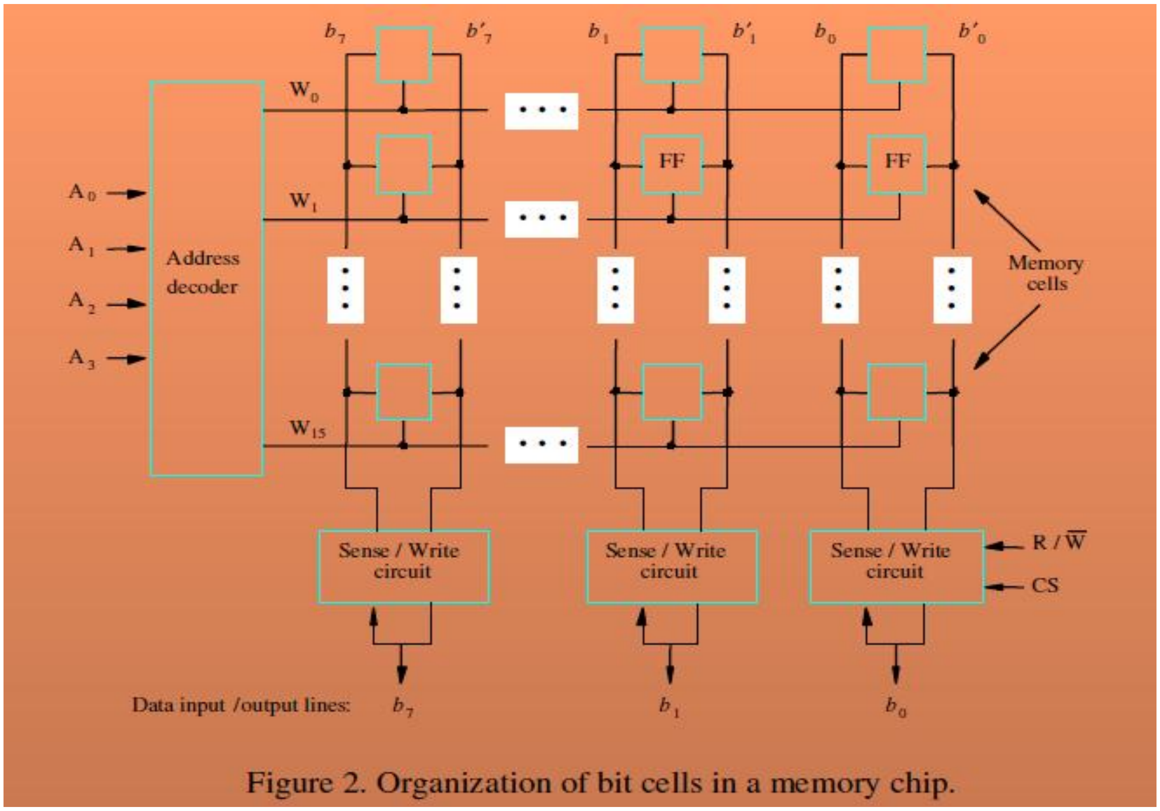


Figure 2. Organization of bit cells in a memory chip.

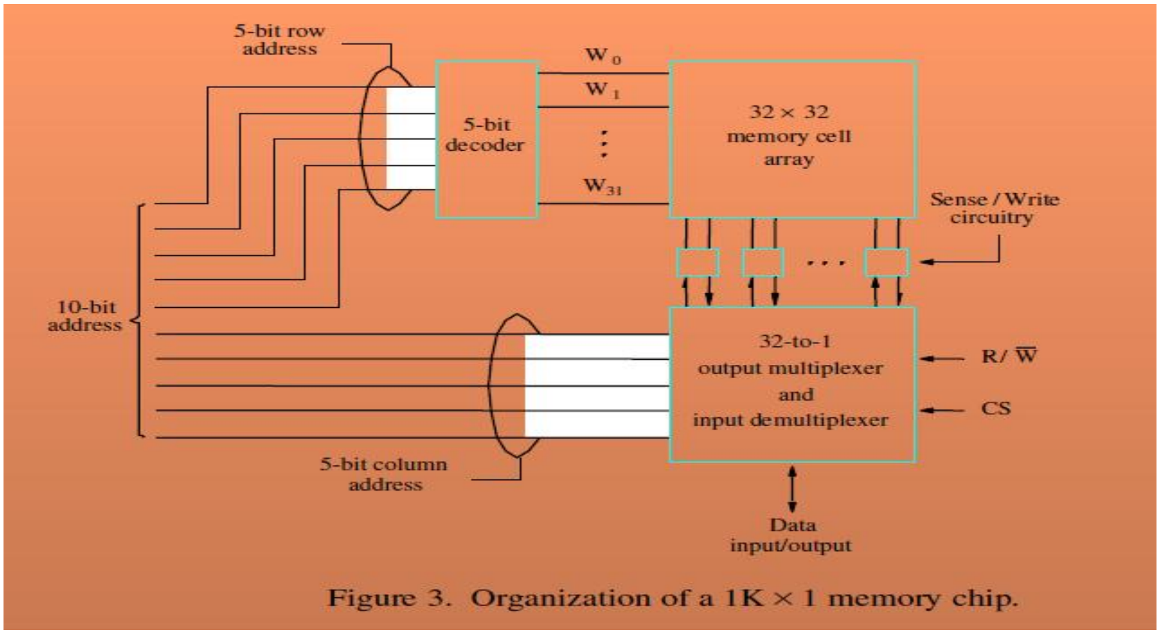


Figure 3. Organization of a 1K × 1 memory chip.

Static memories

– Circuits capable of retaining their state as long as power is applied

– *Static* RAM(SRAM)

– *volatile*

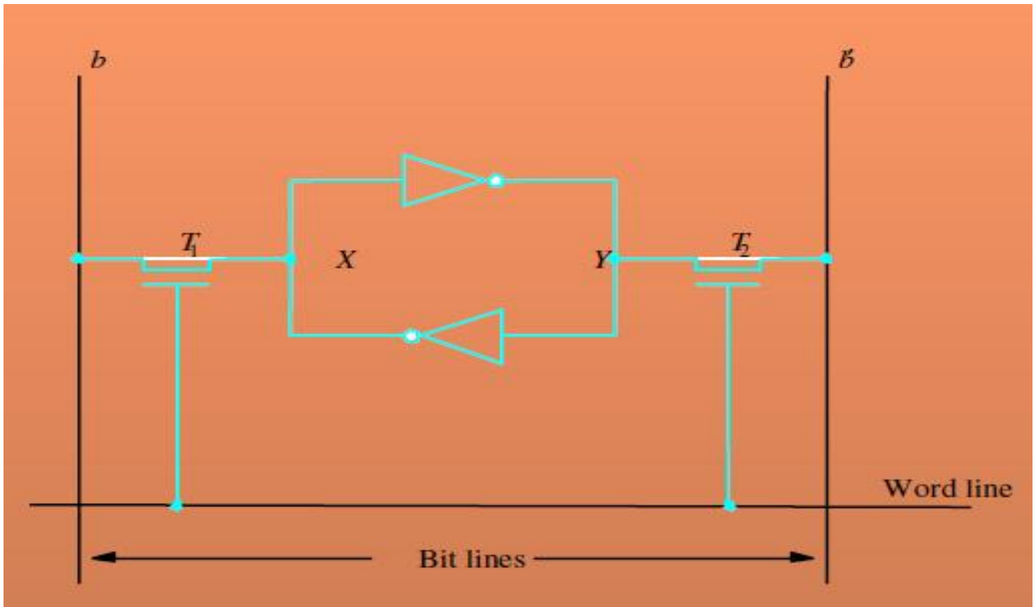


Figure 4. A static RAM cell.

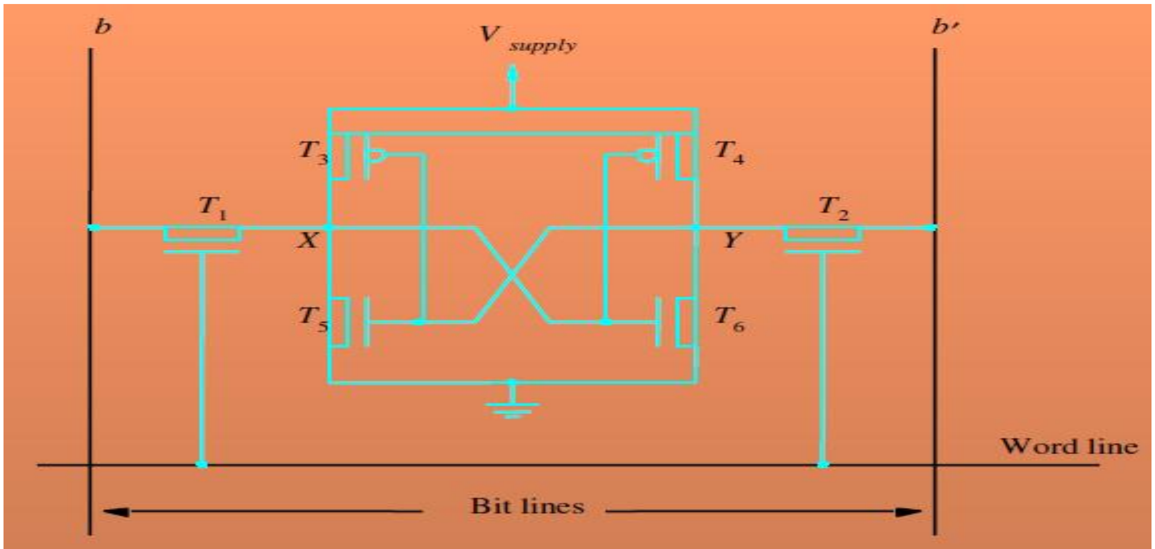
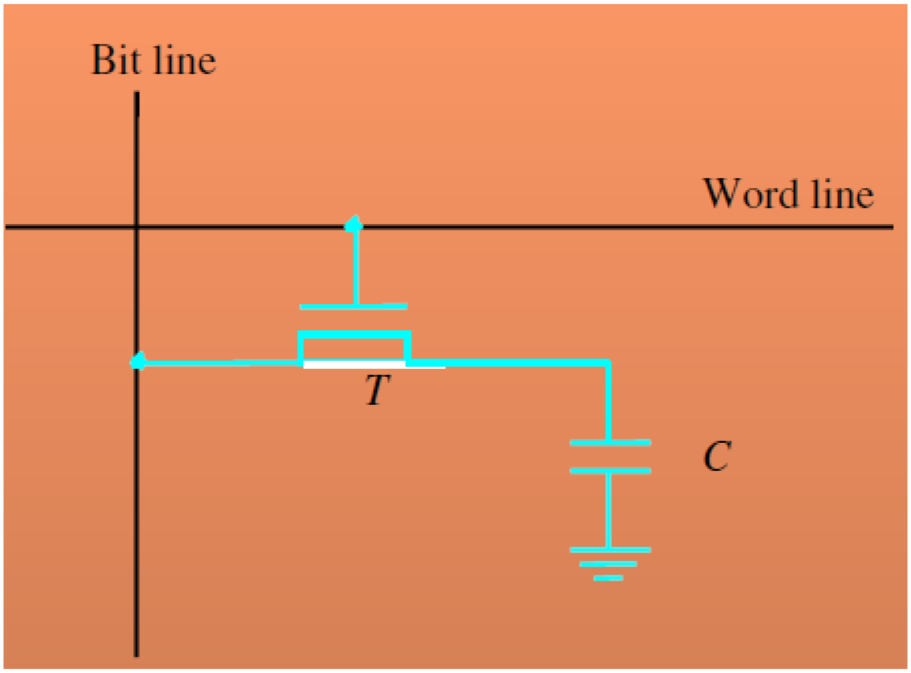


Figure 5. An example of a CMOS memory cell.

DRAMS:

- Charge on a capacitor
- Needs “Refreshing



A singletransistor dynamic memory cell

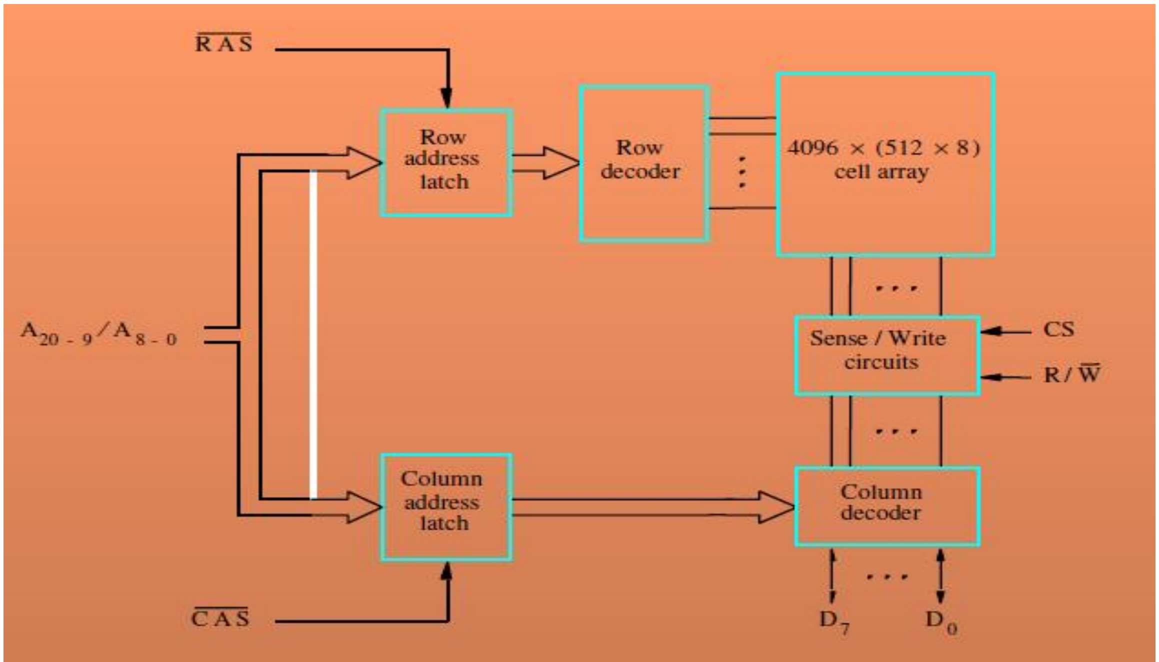


Figure 7. Internal organization of a 2M × 8 dynamic memory chip.

Synchronous DRAMs

Synchronized with a clock signal

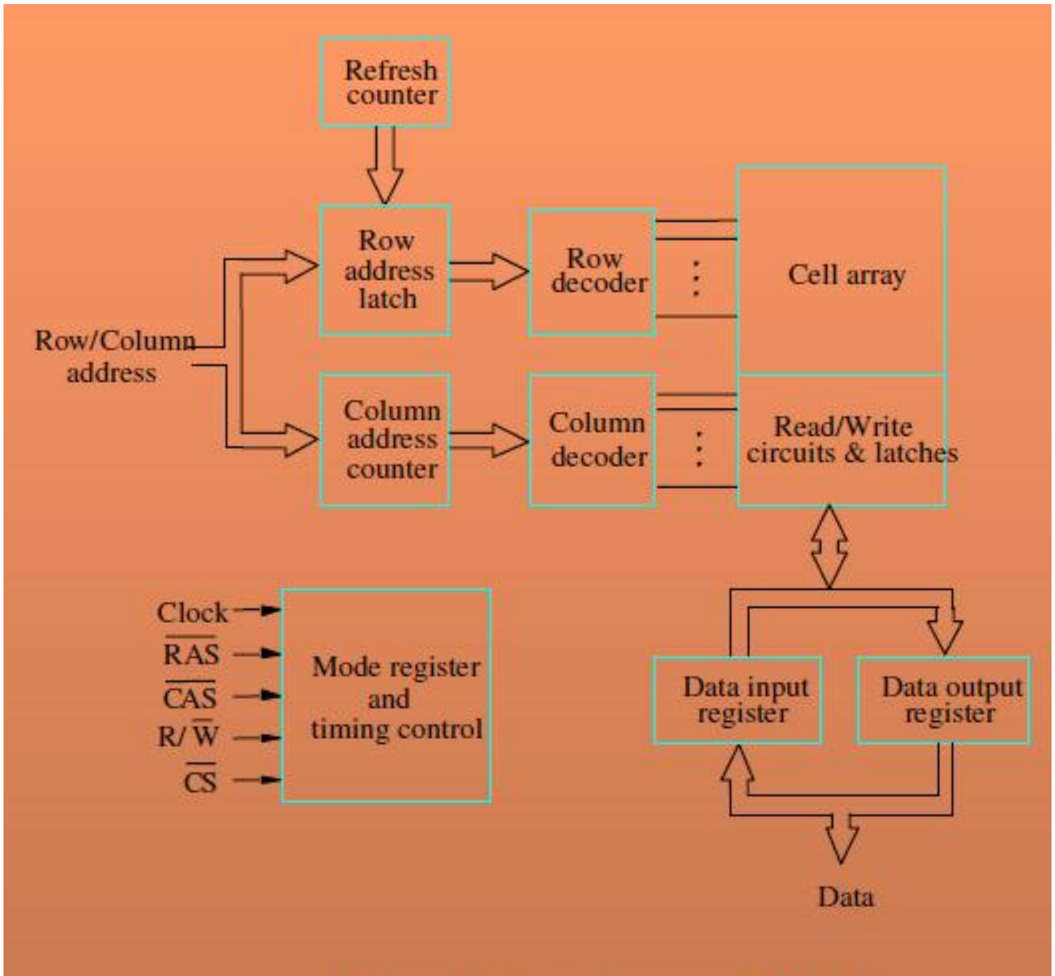


Figure 8. Synchronous DRAM.

Memory system considerations

– Cost
– Speed
– Power dissipation
– Size of chip

Memory controller

- Used Between processor and memory
- Refresh Overhead

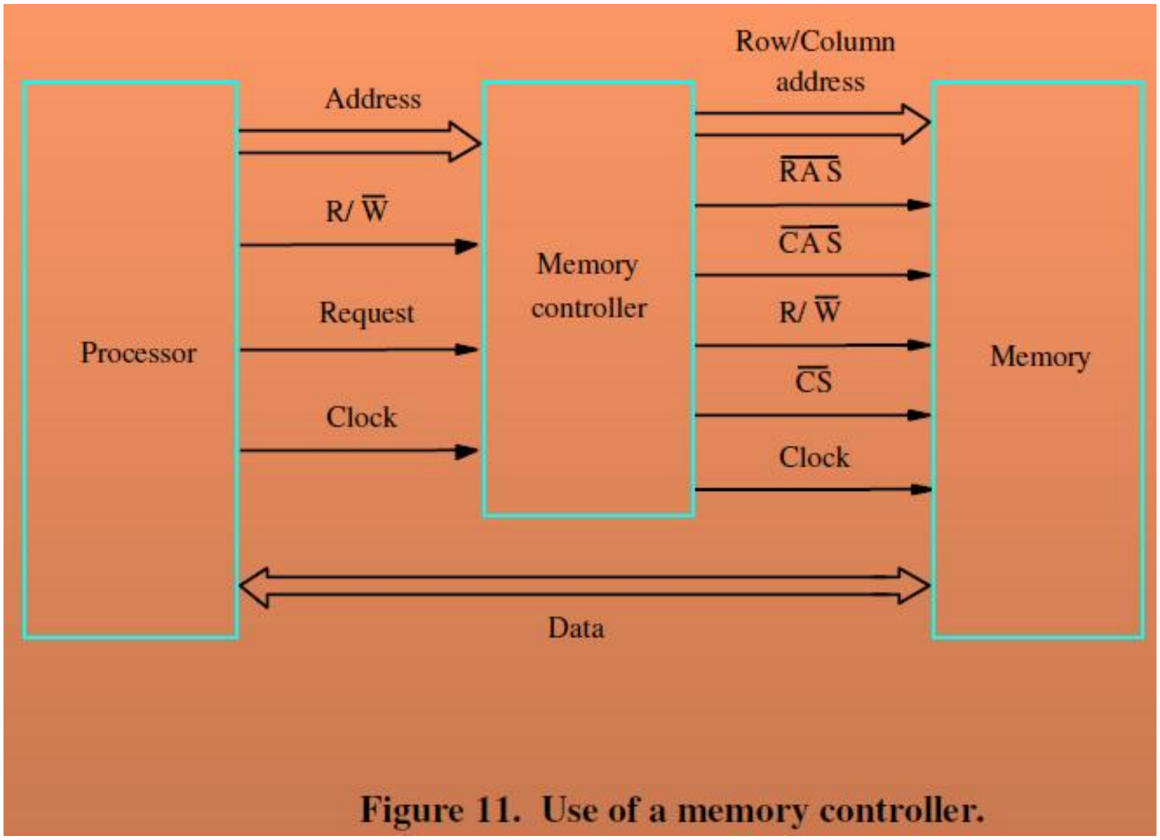
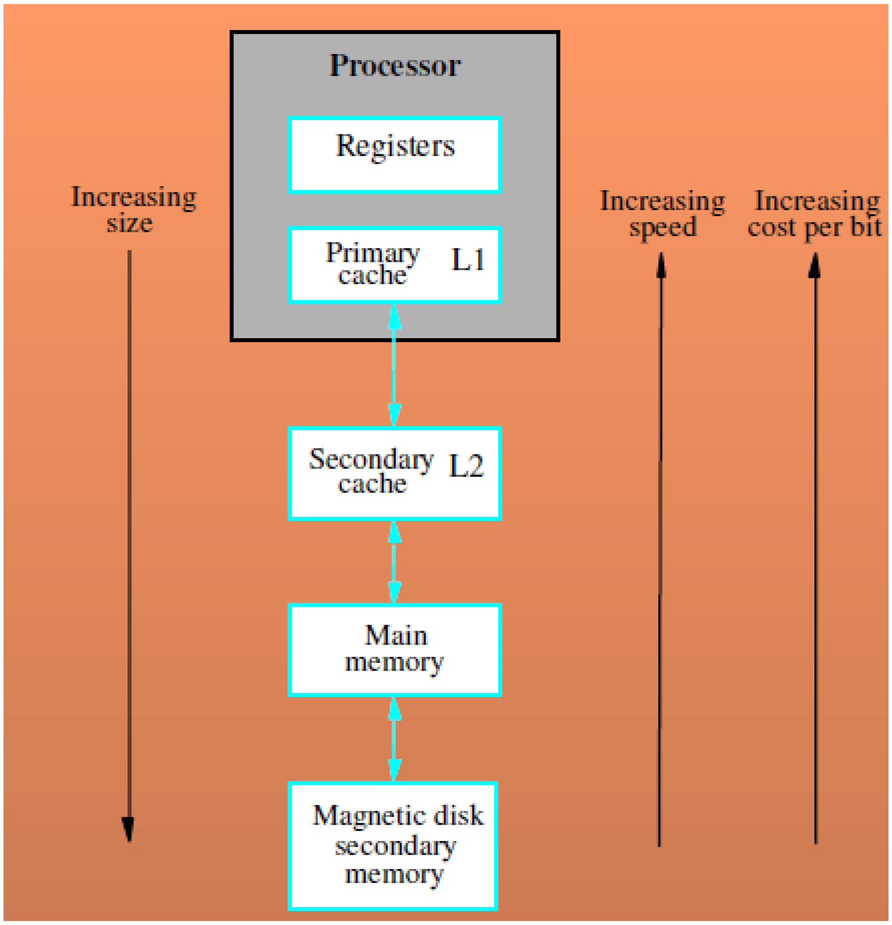


Figure 11. Use of a memory controller.

–

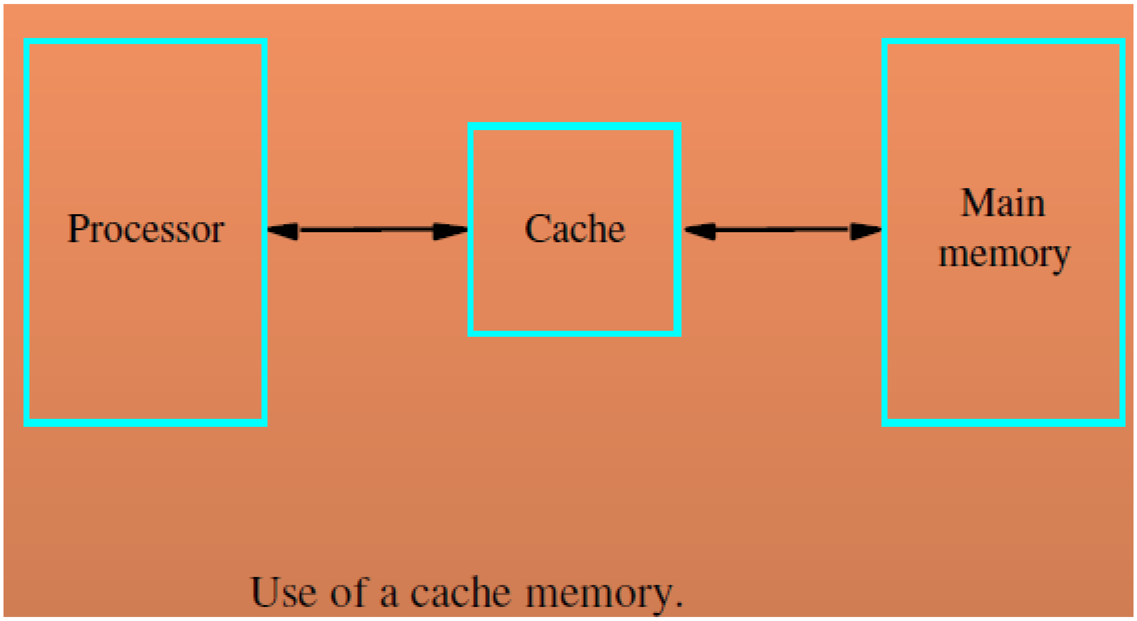
MEMORY HIERARCHY



<u>Principle of *locality*</u>:
□ □**Temporal locality** (locality in time): If an item is referenced, it will tend to be referenced again soon.
□ □**Spatial locality** (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon.
□ □**Sequentiality** (subset of spatial locality).
The principle of locality can be exploited implementing the memory of computer as a *memory hierarchy*, taking advantage of all types of memories.
Method: The level closer to processor (the fastest) is a subset of any level further away, and all the data is stored at the lowest level (the slowest).

Cache Memories

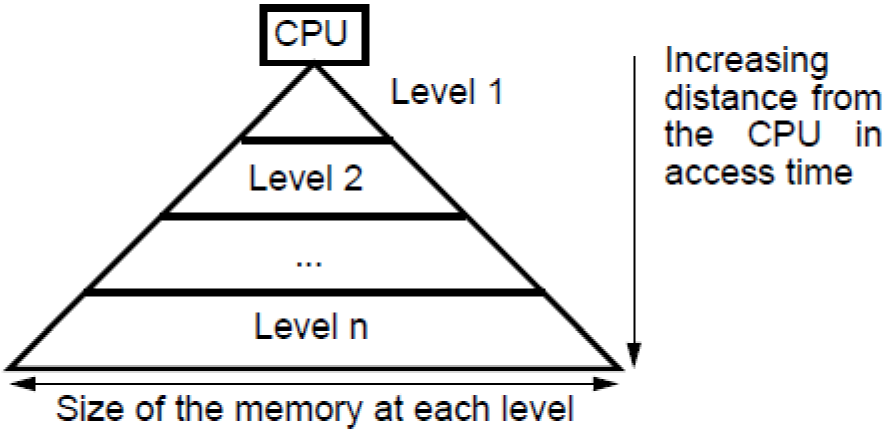
- Speed of the main memory is very low in comparison with the speed of processor
- For good performance, the processor cannot spend much time of its time waiting to access instructions and data in main memory.
- Important to device a scheme that reduces the time to access the information
- An efficient solution is to use fast cache memory
- When a cache is full and a memory word that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contain the referenced word.



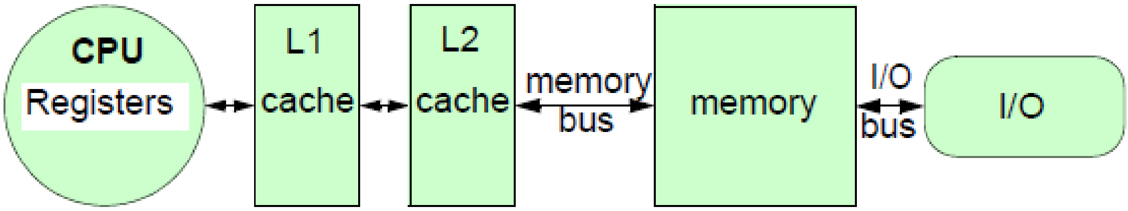
Use of a cache memory.

The basics of Caches

" The caches are organized on basis of *blocks*, the smallest amount of data which can be copied between two adjacent levels at a time.
" If data requested by the processor is present in some block in the upper level,
it is called a *hit*.
" If data is not found in the upper level, the request is called a *miss* and the data is retrieved from the lower level in the hierarchy.
" The fraction of memory accesses found in the upper level is called a *hit ratio*.
" The storage, which takes advantage of locality of accesses is called a *cache*


CPU
Level 1
Level 2
...
Level n
Increasing distance from the CPU in access time
Size of the memory at each level

Typical memory hierarchy:



Level	1	2	3	4
Named as	Registers	Cache	memory	disk storage
Typical size	<1 KB	< 4 MB	<2 GB	>2GB
Access time (ns)	2 - 5	3 - 10	80 - 400	5’000’000
Bandwidth(MB/sec)	4000 - 32’000	800 - 5000	400 - 2000	4 - 32
Managed by	Compiler	Hardware	Operating system	Operating system / user

Performance of caches

Amdahl's Law about overall speedup:

$$\text{Speedup} = \frac{1}{(1 - \text{fraction of time cache can be used}) + \dfrac{\text{fraction of time cache can be used}}{\text{Speedup using cache}}}$$

Alternatively, CPU stalls can be considered[1]:

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle}$$

The number of memory stall cycles depends:

$$\text{Memory stall cycles} = \text{IC} \times \text{Memory references per instruction} \times \text{Miss rate} \times \text{Miss penalty}$$

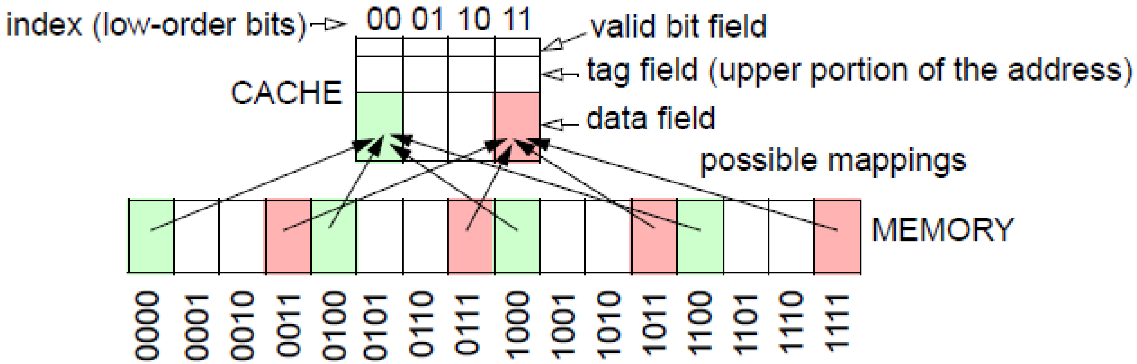
Size	Instruction cache	Data cache	
1 KB	3.06%	24.61%	13.34%
4 KB	1.76%	15.94%	7.24%
16 KB	0.64%	6.47%	2.87%
64 KB	0.15%	3.77%	1.35%

Table: Direct mapped cache, 32-byte blocks, SPEC92, DECstation 5000.

[1] Here is assumed that CPU clock cycles include the time to handle a cache hit, and the CPU is stalled during a cache miss.

Accessing a Cache

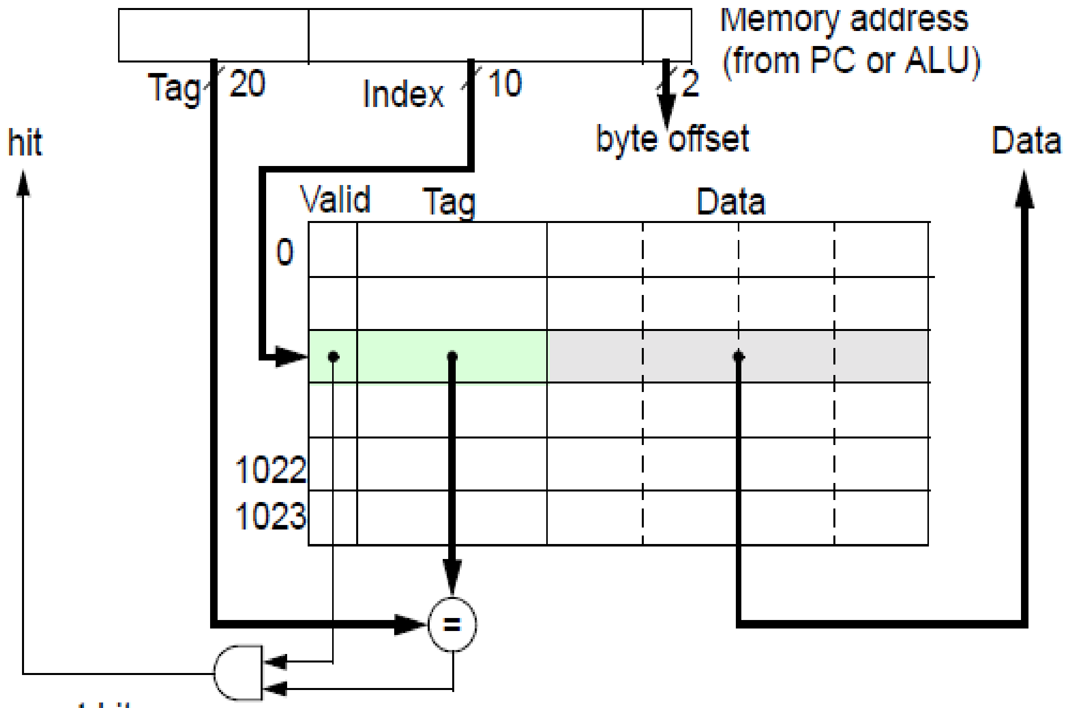
- *Direct mapping*: (Block address) modulo (Number of cache blocks in the cache)



The *valid bit* indicates weather an entry contains a valid address. Initially, all valid bits are reset (“0” - not valid).

- Small fast memory + big slow memory
- Looks like a big fast memory

Address Mapping in Cache:



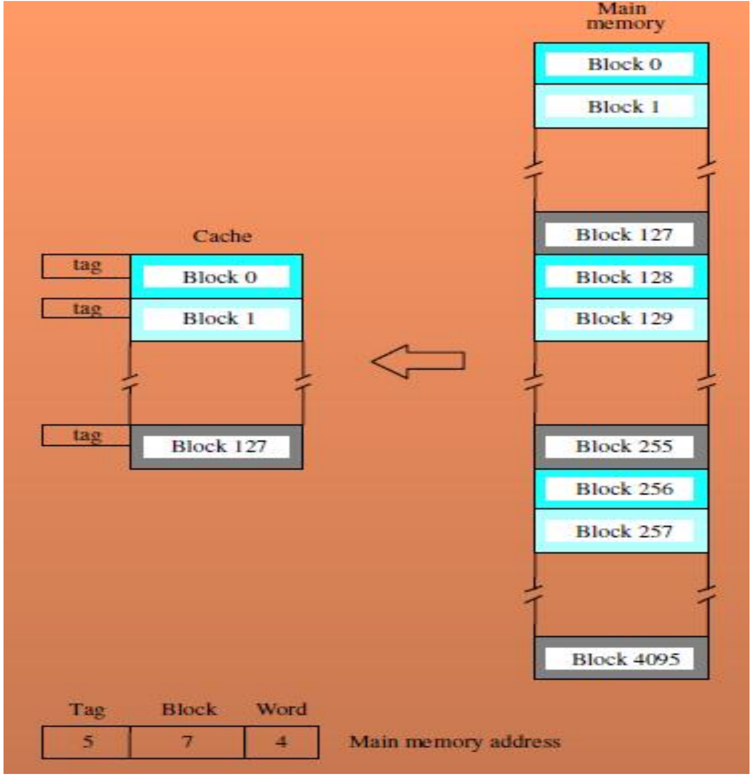
miss = not hit,
meaning the data has to be retrieved form lower level of memory system.

Direct Mapping

In this technique, block j of the main memory maps onto block j modulo 128 of the cache.

- Main memory blocks 0,128,256,…is loaded in the cache, it is stored in cache block 0.
- Blocks 1,129,257,…are stored in cache block 1.

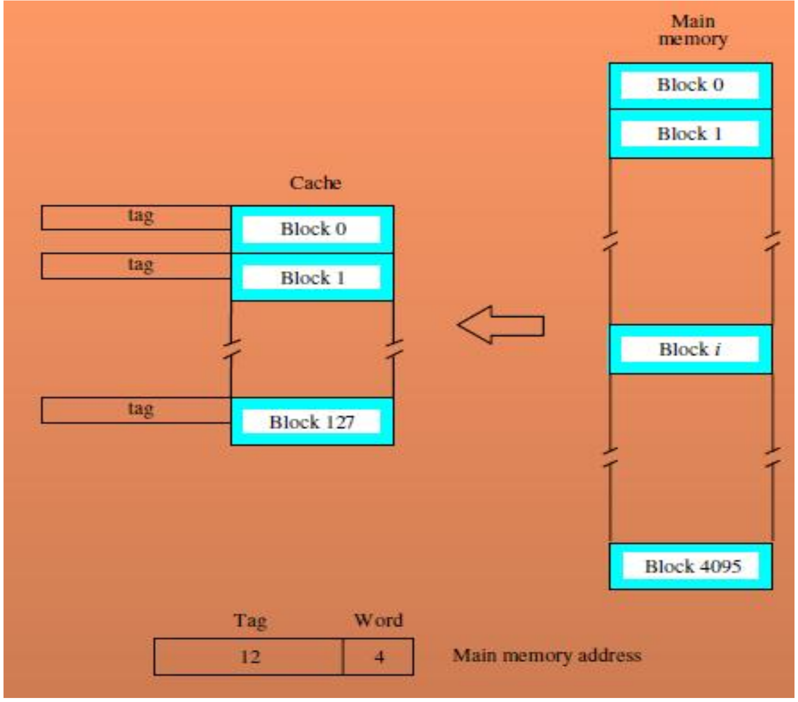
Direct Mapped Cache:



Associative Mapping

More flexible mapping technique

- A main memory block can be placed inot any cache block position.
- Space in the cache can be used more efficiently, but need to search all 128 tag patterns.

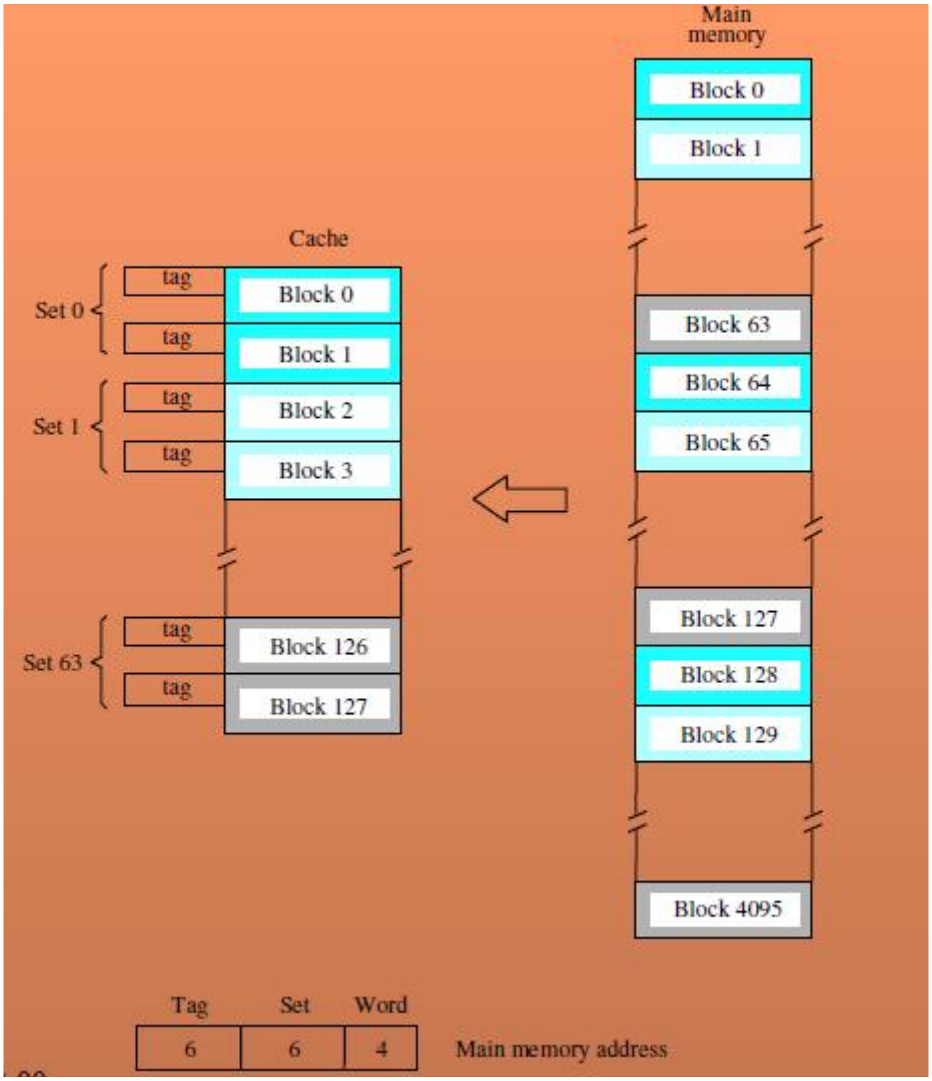


Set-Associate Mapping

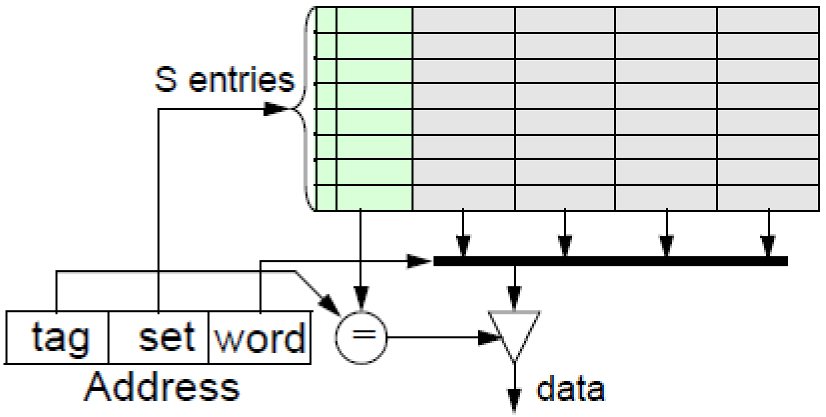
Combination of the direct- and associative mapping technique

- Blocks of the cache are grouped into sets, and the mapping allows a block of the main

memory to reside in any block of a specific set.
Note: Memory blocks 0,64,128,…,4032 maps into cache set 0.



Calculating Block Size:

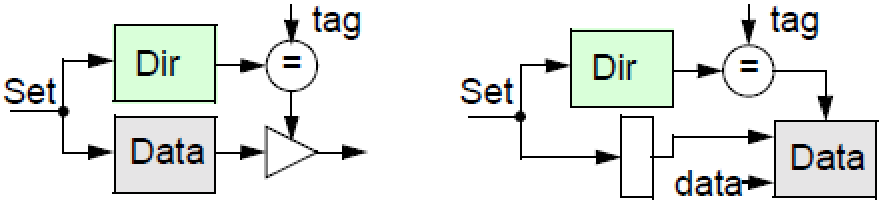


B_A Unit of associativity - one tag for B_A words.
B_T Unit of transfer- B_T words to/from main memory as a unit.
One valid bit for each B_T words.
Need not be the same, but $B_T \leq B_A$

- 👍 **Bigger blocks give lower miss rate**
- 👎 **Big blocks increase miss penalty**

Write Hit Policies:

- write through
    - update next level on every write
    - cache is always clean
    - lots of traffic to next level (mostly writes)
- write back
    - write to cache and mark block dirty
    - update main memory on eviction
    - less traffic to next level, but more complex eviction and coherence
- Reservation problem
    - reads use directory and data array at the same time
    - writes use directory first, then data array
    - how to we pipeline to allow one read or write per cycle?



REPLACEMENT POLICY:

On a cache miss we need to evict a line to make room for the new line
" In an A-way set associative cache, we have A choices of which block to evict
" Which block gets booted out?
! random
! *least-recently used* (true LRU is too costly)
! pseudo LRU (Approximated LRU - in case of four-way set associativity one bit keeps track of which pair of blocks is LRU, and then tracking which block in each pair is LRU (one bit per pair))
! fixed (processing audio stream)
For a two-way set associative cache, random replacement has a miss rate about
1.1 time higher than LRU replacement. As the caches become larger, the miss rate for both replacement strategies fall, and the difference becomes small.
Random replacement is sometimes better than simple LRU approximations that can be easily implemented in hardware.

WRITE MISS POLICY:

- Write allocate
- allocate a new block on each write
- fetch on write
- **fetch entire block, then write word into block**
- no-fetch
- **allocate block, but don't fetch**
- **requires valid bits per word**
- **more complex eviction**
- Write no-allocate
- don't allocate a block if it is not already in the cache
- write around the cache
- typically used by write through since we need update main memory anyway
- Write invalidate
- instead of update for write-through

Measuring and Improving Cache Performance

1. Reduce the probability that two different memory blocks will contend for the same cache location
2. Additional cache levels.

$$\text{Memory-stall clock cycles} = \text{Read-stall cycles} + \text{Write-stall cycles}$$

$$\text{Read-stall cycles} = \frac{\text{Reads}}{\text{Program}} \times \text{Read miss rate} \times \text{Read miss penalty}$$

$$\text{Write-stall cycles} = \left(\frac{\text{Writes}}{\text{Program}} \times \text{Write miss rate} \times \text{Write miss penalty}\right) + \text{Write buffer stalls}$$

Simplifying, ignoring write buffer stalls whish are not remarkable in case of enough large write buffers:

$$\text{Memory-stall clock cycles} = \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

or:

$$\text{Memory-stall clock cycles} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Rule of dumb: Increase cache size by 8 to halve miss rate

Virtual memory

Virtual memory is a computer system technique which gives an application program the impression that it has contiguous working memory (an address space), while in fact it may be physically fragmented and may even overflow on to disk storage.

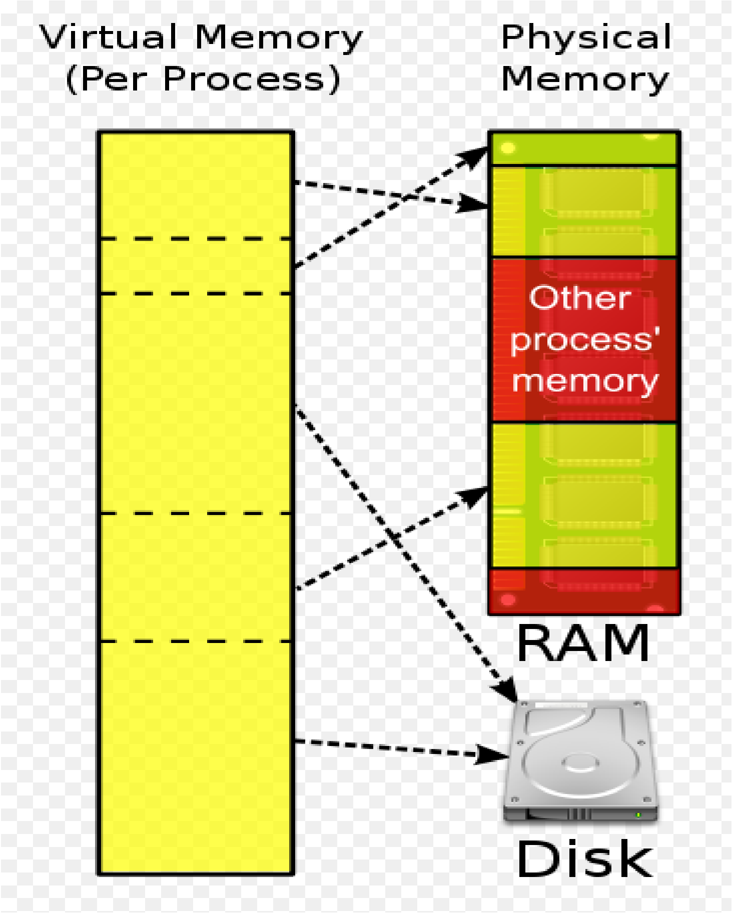
Virtual memory provides two primary functions:

1. Each process has its own address space, thereby not required to be relocated nor required to use relative addressing mode.
2. Each process sees one contiguous block of free memory upon launch. Fragmentation is hidden.

All implementations (Excluding emulators) require hardware support. This is typically in the form of a Memory Management Unit built into the CPU.

Systems that use this technique make programming of large applications easier and use real physical memory (e.g. RAM) more efficiently than those without virtual memory. Virtual memory differs significantly from memory virtualization in that virtual memory allows resources to be virtualized as memory for a specific system, as opposed to a large pool of memory being virtualized as smaller pools for many different systems.

Note that "virtual memory" is more than just "using disk space to extend physical memory size" - that is merely the extension of the memory hierarchy to include hard disk drives. Extending memory to disk is a normal consequence of using virtual memory techniques, but could be done by other means such as overlays or swapping programs and their data completely out to disk while they are inactive. The definition of "virtual memory" is based on redefining the address space with a *contiguous virtual memory addresses* to "trick" programs into thinking they are using large blocks of *contiguous* addresses.



Paged virtual memory

Almost all implementations of virtual memory divide the virtual address space of an application program into pages; a page is a block of contiguous virtual memory addresses. Pages are usually at least 4K bytes in size, and systems with large virtual address ranges or large amounts of real memory (e.g. RAM) generally use larger page sizes.

Page tables

Almost all implementations use page tables to translate the virtual addresses seen by the application program into physical addresses (also referred to as "real addresses") used by the hardware to process instructions. Each entry in the page table contains a mapping for a virtual page to either the real memory address at which the page is stored, or an indicator that the page is currently held in a disk file. (Although most do, some systems may not support use of a disk file for virtual memory.)

Systems can have one page table for the whole system or a separate page table for each application. If there is only one, different applications which are running at the same time share a single virtual address space, i.e. they use different parts of a single range of virtual addresses. Systems which use multiple page tables provide multiple virtual address spaces - concurrent applications think they are using the same range of virtual addresses, but their separate page tables redirect to different real addresses.

Dynamic address translation

If, while executing an instruction, a CPU fetches an instruction located at a particular virtual address, or fetches data from a specific virtual address or stores data to a particular virtual address, the virtual address must be translated to the corresponding physical address. This is done by a hardware component, sometimes called a memory management unit, which looks up the real address (from the page table) corresponding to a virtual address and passes the real address to the parts of the CPU which execute instructions.

Paging supervisor

This part of the operating system creates and manages the page tables. If the dynamic address translation hardware raises a page fault exception, the paging supervisor searches the page space on secondary storage for the page containing the required virtual address, reads it into real physical memory, updates the page tables to reflect the new location of the virtual address and finally tells the dynamic address translation mechanism to start the search again. Usually all of the real physical memory is already in use and the paging supervisor must first save an area of real physical memory to disk and update the page table to say that the associated virtual addresses are no longer in real physical memory but saved on disk. Paging supervisors generally save and overwrite areas of real physical memory which have been least recently used, because these are probably the areas which are used least often. So every time the dynamic address translation hardware matches a virtual address with a real physical memory address, it must put a time-stamp in the page table entry for that virtual address.

Permanently resident pages

All virtual memory systems have memory areas that are "pinned down", i.e. cannot be swapped out to secondary storage, for example:

- Interrupt mechanisms generally rely on an array of pointers to the handlers for various types of interrupt (I/O completion, timer event, program error, page fault, etc.). If the pages containing these pointers or the code that they invoke were pageable, interrupt-handling would become even more complex and time-consuming; and it would be especially difficult in the case of page fault interrupts.
- The page tables are usually not pageable.
- Data buffers that are accessed outside of the CPU, for example by peripheral devices that use direct memory access (DMA) or by I/O channels. Usually such devices and the buses (connection paths) to which they are attached use physical memory addresses rather than virtual memory addresses. Even on buses with an IOMMU, which is a special memory management unit that can translate virtual addresses used on an I/O bus to physical addresses, the transfer cannot be stopped if a page fault occurs and then restarted when the page fault has been processed. So pages containing locations to which or from which a peripheral device is transferring data are either permanently pinned down or pinned down while the transfer is in progress.
- Timing-dependent kernel/application areas cannot tolerate the varying response time caused by paging.

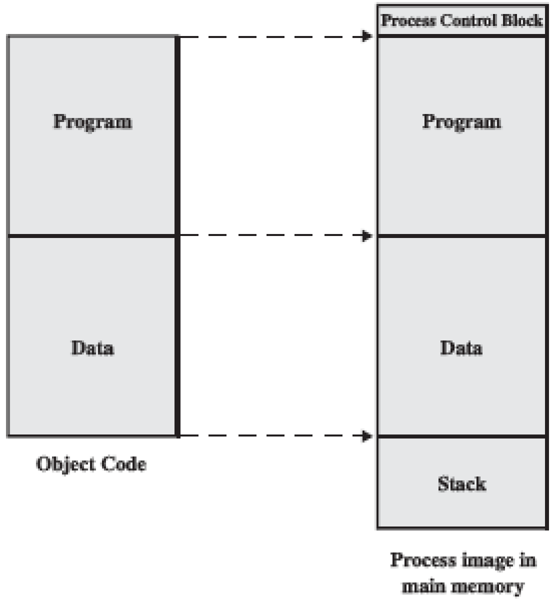
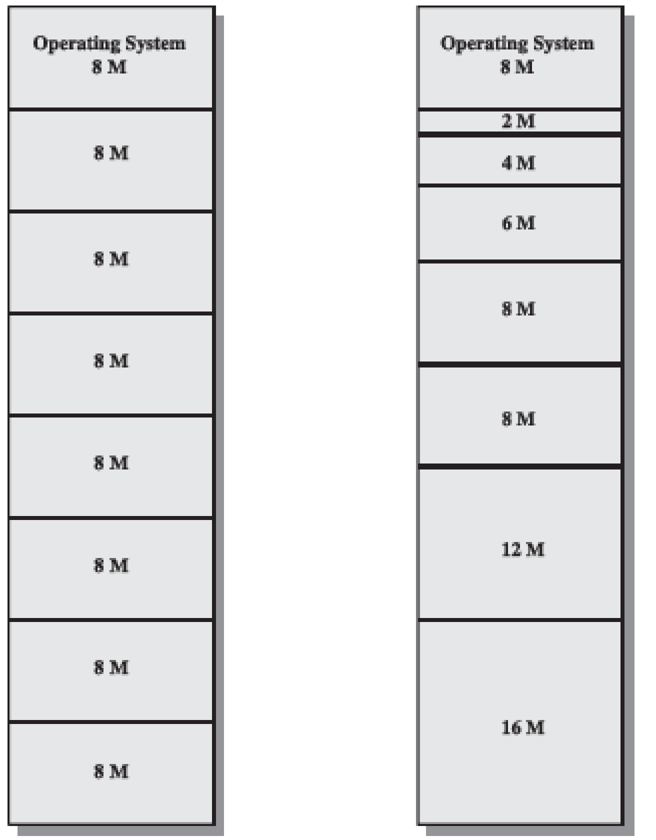


Figure 2: Address translation

Figure 2: Address translation

• **Compiler time**: If it is known in advance that a program will reside at a specific location of main memory, then the compiler may be told to build the object code with absolute addresses right away. For example, the boot sect in a bootable disk may be compiled with the starting point of code set to 007C:0000.

• **Load time**: It is pretty rare that we know the location a program will be assigned ahead of its execution. In most cases, the compiler must generate relocatable code with logical addresses. Thus the address translation may be performed on the code during load time. Figure 3 shows that a program is loaded at location x. If the whole program resides on a monolithic block, then every memory reference may be translated to be physical by added to x.



(a) Equal-size partitions

(b) Unequal-size partitions

Figure 4: Example of fixed partitioning of a 64-Megabyte memory

Figure 4: Example of fixed partitioning of a 64-Megabyte memory

However two disadvantages are

- A program that is too big to be held in a partition needs some special design, called overlay, which brings heavy burden on programmers. With overlay, a process consists of several portions with each being mapped to the same location of the partition, and at any time, only one portion may reside in the partition. When another portion is referenced, the current portion will be switched out.
- A program may be much smaller than a partition, thus space left in the partition will be wasted, which is referred to as internal fragmentation. As an improvement shown in Figure 4 (b), unequal-size partitions may be configured in main memory so that small programs will occupy small partitions and big programs are also likely to be able to fit into big partitions. Although this may solve the above problems with fixed 5 equal-size partitioning to some degree, the fundamental weakness still exists: The number of partitions are the maximum of the number of processes that could reside in main memory at the same time. When most processes are small, the system should be able to accommodate more of them but fails to do so due to the limitation. More flexibility is needed.

Dynamic partitioning

To overcome difficulties with fixed partitioning, partitioning may be done dynamically, called dynamic partitioning. With it, the main memory portion for user applications is initially a single contiguous block. When a new process is created, the exact amount of memory space is allocated to the process. Similarly when no enough space is available, a process may be swapped out temporarily to release space for a new process. The way how the dynamic partitioning works is illustrated in Figure 5.

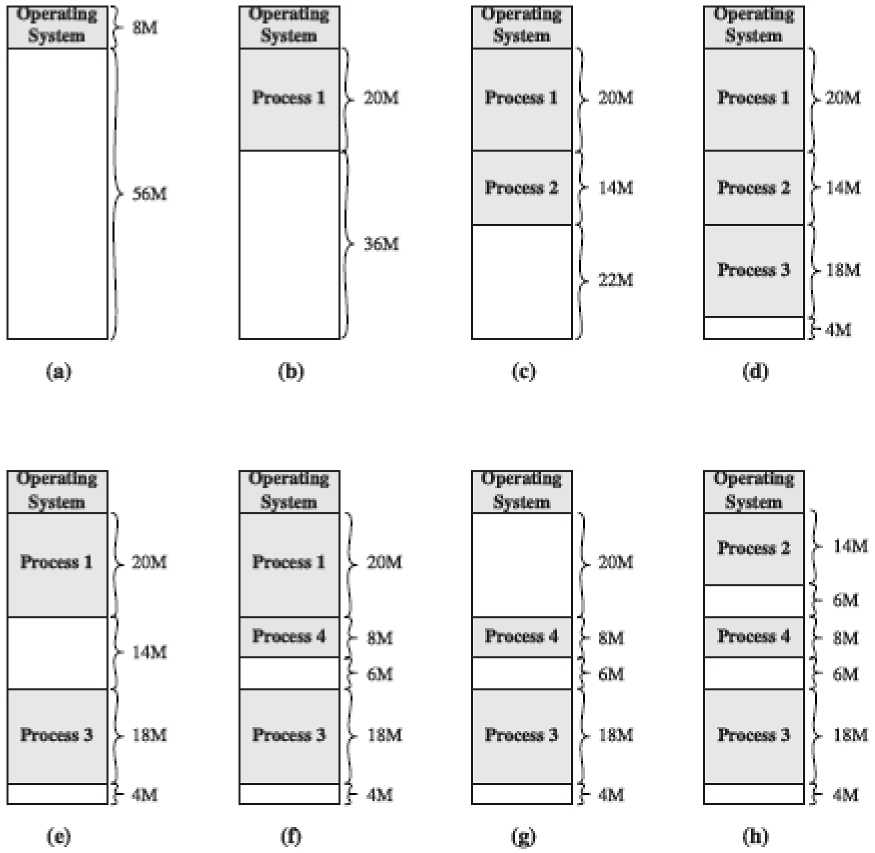


Figure 5: The effect of dynamic partitioning

Figure 5: The effect of dynamic partitioning

As time goes on, there will appear many small holes in the main memory, which is referred to 6 as external fragmentation. Thus although much space is still available, it cannot be allocated to new processes. A method for overcoming external fragmentation is compaction. From time to time, the operating system moves the processes so that they occupy contiguous sections and all of the small holes are brought together to make a big block of space. The disadvantage of compaction is: The procedure is time-consuming and requires relocation capability.

Address translation

Figure 6 shows the address translation procedure with dynamic partitioning, where the processor provides hardware support for address translation, protection, and relocation.

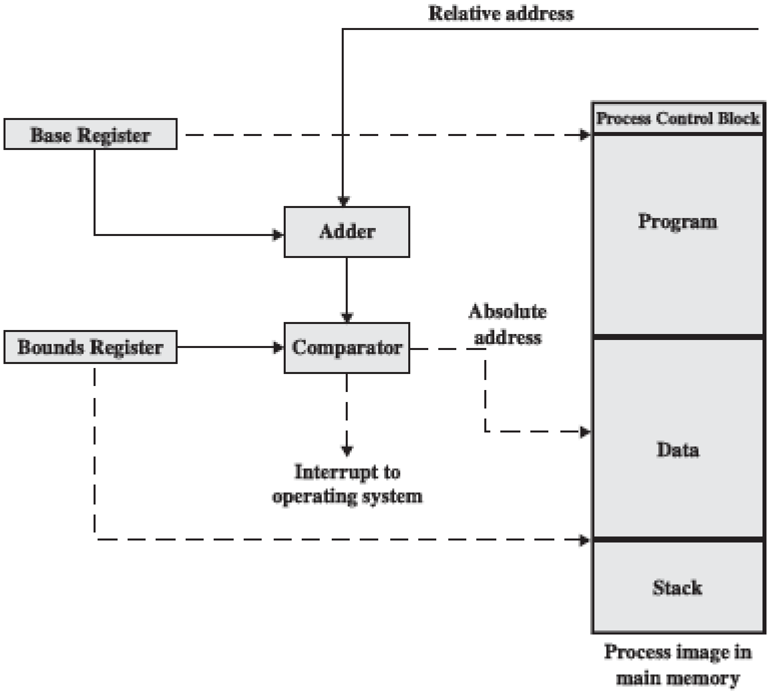


Figure 6: Address translation with dynamic partitioning

The base register holds the entry point of the program, and may be added to a relative address to generate an absolute address. The bounds register indicates the ending location of the program, which is used to compare with each physical address generated. If the later is within bounds, then the execution may proceed; otherwise, an interrupt is generated, indicating illegal access to memory.

The relocation can be easily supported with this mechanism with the new starting address and ending address assigned respectively to the base register and the bounds 7 register.

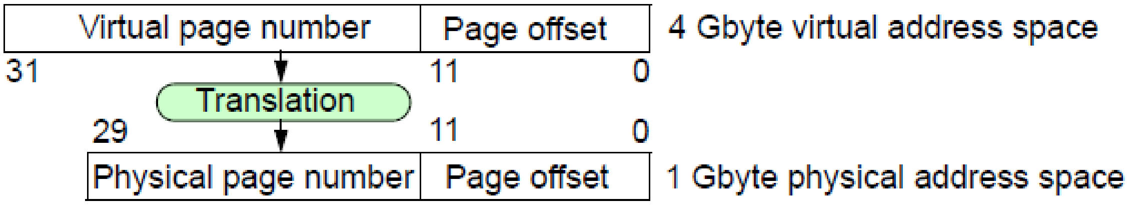
Placement algorithm

Different strategies may be taken as to how space is allocated to processes:

• **First fit**: Allocate the first hole that is big enough. Searching may start either at the beginning of the set of holes or where the previous first-fit search ended.

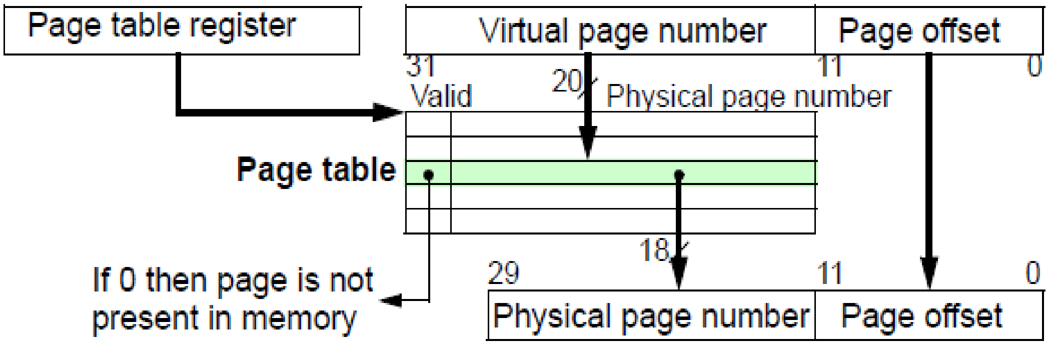
• **Best fit**: Allocate the smallest hole that is big enough. The entire list of holes must be searched unless it is sorted by size. This strategy produces the smallest leftover hole.

• **Worst fit**: Allocate the largest hole. In contrast, this strategy aims to produce the largest leftover hole, which may be big enough to hold another process. Experiments have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization.



Handling a Page:

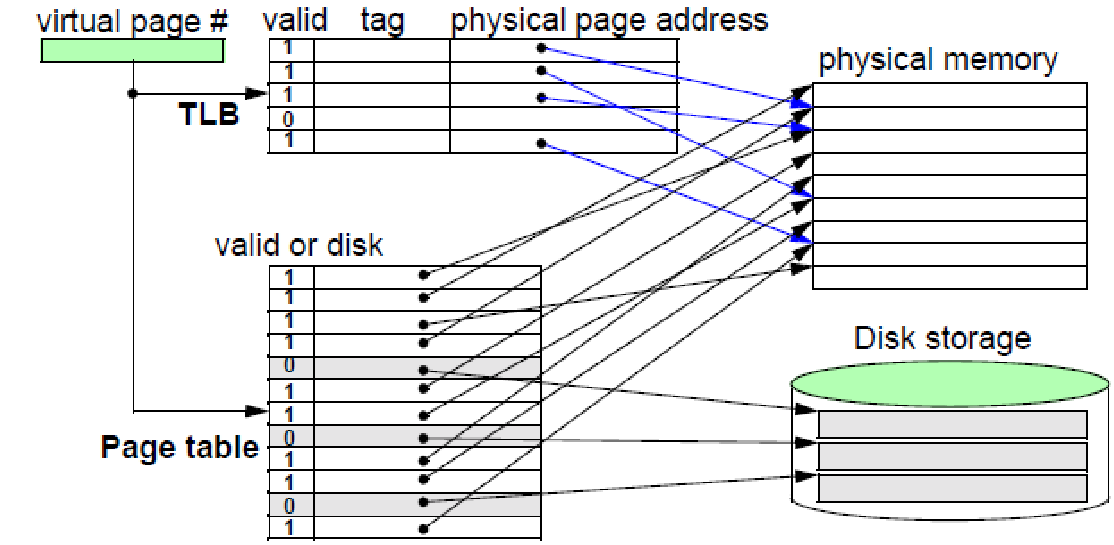
- Typical page sizes today are 4KB to 16KB, with tendency to use even larger page sizes[1].
- Organizations that reduce the page fault rate are attractive (fully associative placement of pages).
- Page faults can be handled in software because the overhead is small compared to the access time to disk. Also, better algorithms can be used.
- Write-back is used to manage writes (write-through takes too long...).



[1] There is a variable size-block scheme called *segmentation*. The segment register is mapped to physical address, and the *offset* is added to find the actual physical address. Also, the bounds check is needed.

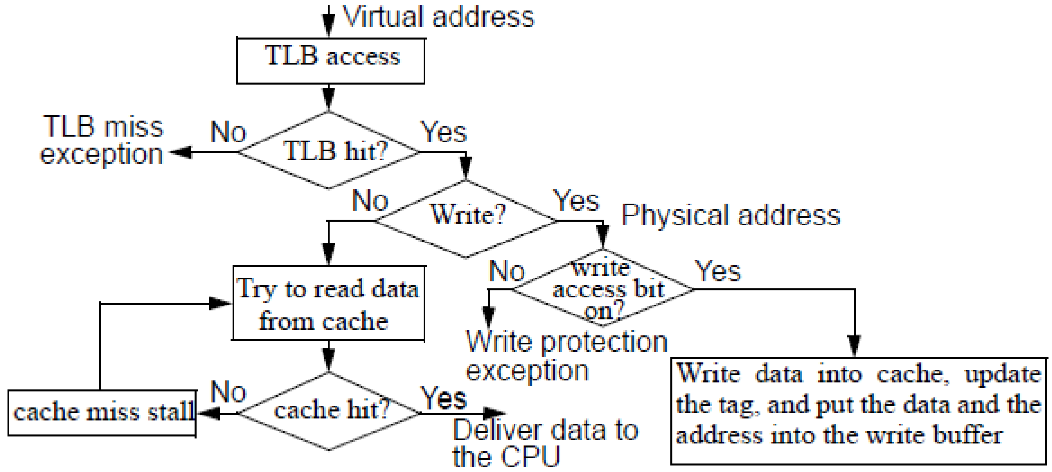
Translation Look aside Buffer:

A TLB is a cache which holds only page table mappings



- If there is no matching entry in the TLB for a page, the page table must be examined.
- If page table indicates that the page resides on disk, a page fault occurs.

Integrating Virtual Memory, TLBs, and Caches



Three types of misses:

1. TLB miss
2. a cache miss
3. a page fault.

Two techniques of writes: *write-through* (write buffer needed) or *write-back* (*dirty bit* in page table needed). The last one is also called the *copy-back* technique.

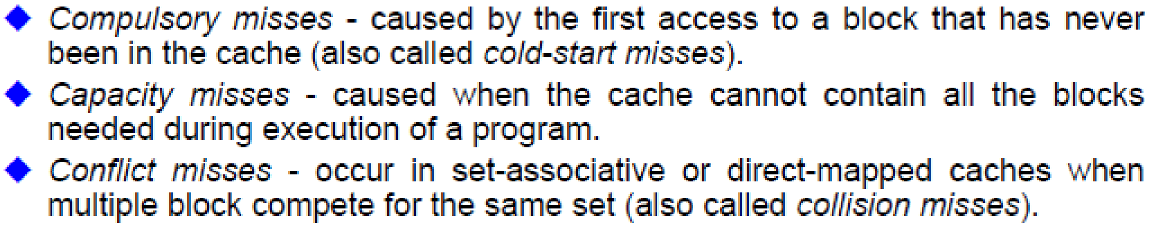
Implementing Protection with Virtual Memory

To enable the OS to implement protection in the VM system, the HW must:

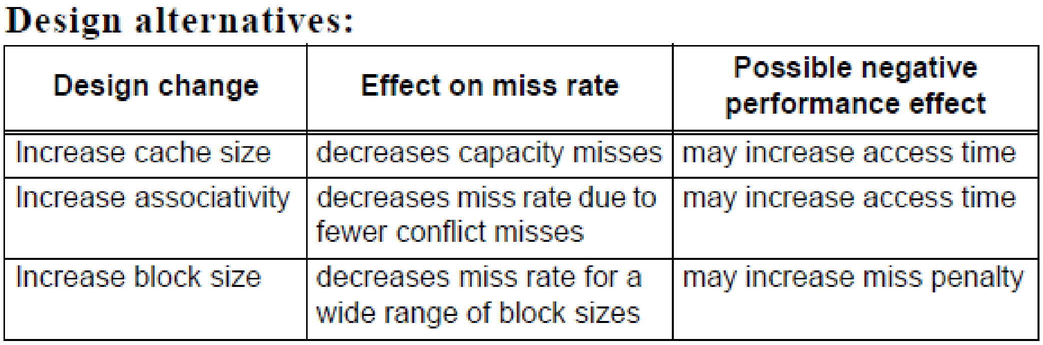
1. Support at least two modes that indicate weather the running process is a user process (*executive process*) or an OS process (*kernel/supervisor* process).
2. provide a portion of the CPU state that a user process can read but not write (includes supervisor mode bit).

3. provide mechanism whereby the CPU can go from user mode to supervisor mode (accomplished by a *system call* exception) and vice versa (*return from exception* instruction).
□ □ Only OS process can change page tables. Page tables are held in OSaddress space thereby preventing a user process from changing them. □ □ When processes want to share information in a limited way, the operating system must assist them.
□ □ The write accessbit (in both the TLB and the page table) can be used to restrict the sharing to just read sharing.

Cache Misses



- *Compulsory misses* - caused by the first access to a block that has never been in the cache (also called *cold-start misses*).
- *Capacity misses* - caused when the cache cannot contain all the blocks needed during execution of a program.
- *Conflict misses* - occur in set-associative or direct-mapped caches when multiple block compete for the same set (also called *collision misses*).



Design alternatives:

Design change	Effect on miss rate	Possible negative performance effect
Increase cache size	decreases capacity misses	may increase access time
Increase associativity	decreases miss rate due to fewer conflict misses	may increase access time
Increase block size	decreases miss rate for a wide range of block sizes	may increase miss penalty

Computer data storage

Computer data storage, often called **storage** or **memory**, refers to computer components, devices, and recording media that retain digital data used for computing for some interval of time. Computer data storage provides one of the core functions of the modern computer, that of information retention. It is one of the fundamental components of all modern computers, and coupled with a central processing unit (CPU, a processor), implements the basic computer model used since the 1940s.

In contemporary usage, *memory* usually refers to a form of semiconductor storage known as random-access memory (RAM) and sometimes other forms of fast but temporary storage. Similarly, *storage* today more commonly refers to mass storage — optical discs, forms of magnetic storage like hard disk drives, and other types slower than RAM, but of a more permanent nature. Historically, *memory* and *storage* were respectively called *main memory* and *secondary storage*. The terms *internal memory* and *external memory* are also used.

The contemporary distinctions are helpful, because they are also fundamental to the architecture of computers in general. The distinctions also reflect an important and significant technical difference between memory and mass storage devices, which has been blurred by the historical usage of the term *storage*. Nevertheless, this article uses the traditional nomenclature.

Purpose of storage

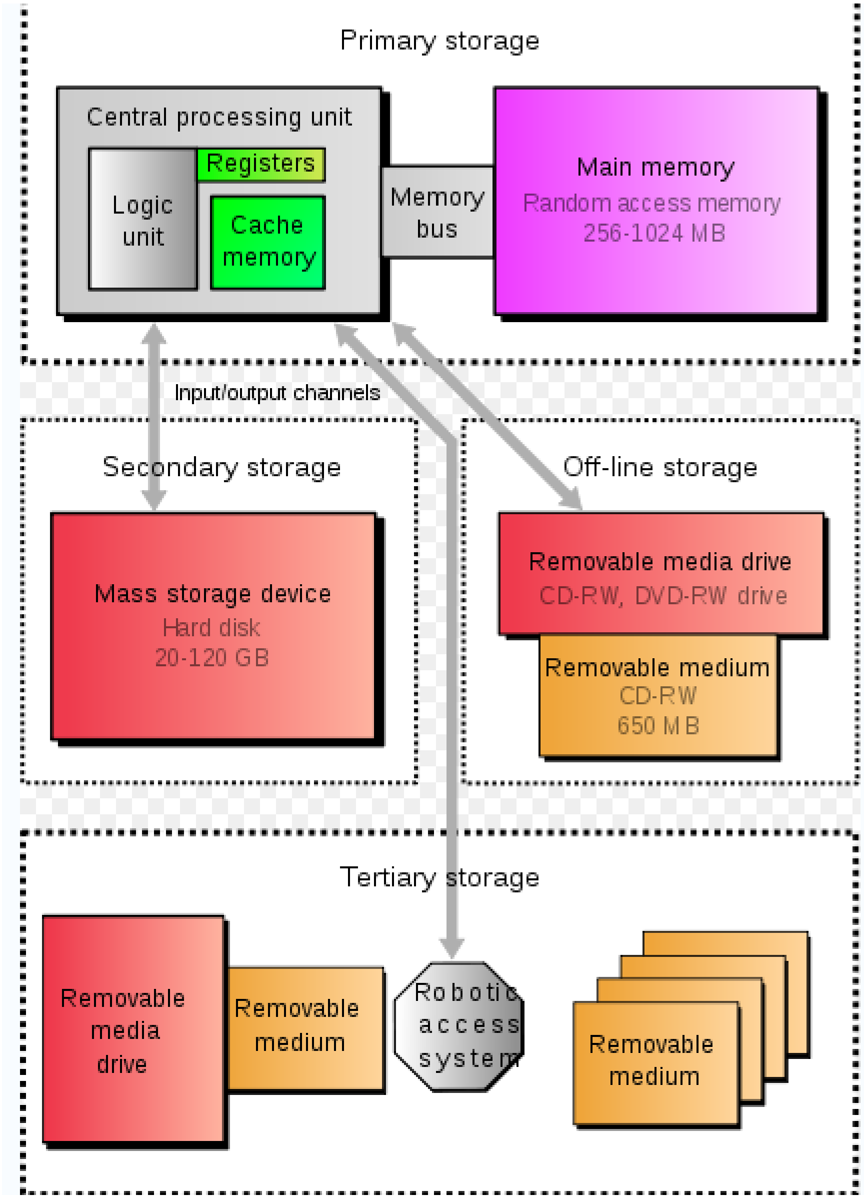
Many different forms of storage, based on various natural phenomena, have been invented. So far, no practical universal storage medium exists, and all forms of storage have some drawbacks. Therefore a computer system usually contains several kinds of storage, each with an individual purpose.

A digital computer represents data using the binary numeral system. Text, numbers, pictures, audio, and nearly any other form of information can be converted into a string of bits, or binary digits, each of which has a value of 1 or 0. The most common unit of storage is the byte, equal to 8 bits. A piece of information can be handled by any computer whose storage space is large enough to accommodate *the binary representation of the piece of information*, or simply data. For example, using eight million bits, or about one megabyte, a typical computer could store a short novel.

Traditionally the most important part of every computer is the central processing unit (CPU, or simply a processor), because it actually operates on data, performs any calculations, and controls all the other components.

In practice, almost all computers use a variety of memory types, organized in a storage hierarchy around the CPU, as a tradeoff between performance and cost. Generally, the lower a storage is in the hierarchy, the lesser its bandwidth and the greater its access latency is from the CPU. This traditional division of storage to primary, secondary, tertiary and off-line storage is also guided by cost per bit.

Hierarchy of storage



Secondary storage

Secondary storage (or **external memory**) differs from primary storage in that it is not directly accessible by the CPU. The computer usually uses its input/output channels to access secondary storage and transfers the desired data using intermediate area in primary storage. Secondary storage does not lose the data when the device is powered down—it is non-volatile. Per unit, it is typically also an order of magnitude less expensive than primary storage. Consequently, modern computer systems typically have an order of magnitude more secondary storage than primary storage and data is kept for a longer time there.

In modern computers, hard disk drives are usually used as secondary storage. The time taken to access a given byte of information stored on a hard disk is typically a few thousandths of a second, or milliseconds. By contrast, the time taken to access a given byte of information stored in random access memory is measured in billionths of a second, or nanoseconds. This illustrates the very significant access-time difference which distinguishes solid-state memory from rotating magnetic storage devices: hard disks are typically about a million times slower than memory. Rotating optical storage devices, such as CD and DVD drives, have even longer access times. With disk drives, once the disk read/write head reaches the proper placement and the data of interest rotates under it, subsequent data on the track are very fast to access. As a result, in order to hide the initial seek time and rotational latency, data are transferred to and from disks in large contiguous blocks.

When data reside on disk, block access to hide latency offers a ray of hope in designing efficient external memory algorithms. Sequential or block access on disks is orders of magnitude faster than random access, and many sophisticated paradigms have been developed to design efficient algorithms based upon sequential and block access . Another way to reduce the I/O bottleneck is to use multiple disks in parallel in order to increase the bandwidth between primary and secondary memory.

Some other examples of secondary storage technologies are: flash memory (e.g. USB flash drives or keys), floppy disks, magnetic tape, paper tape, punched cards, standalone RAM disks, and Iomega Zip drives.

Characteristics of storage

A 1GB DDR RAM memory module

Storage technologies at all levels of the storage hierarchy can be differentiated by evaluating certain core characteristics as well as measuring characteristics specific to a particular implementation. These core characteristics are volatility, mutability, accessibility, and addressibility. For any particular implementation of any storage technology, the characteristics worth measuring are capacity and performance.

Volatility

Non-volatile memory

> Will retain the stored information even if it is not constantly supplied with electric power. It is suitable for long-term storage of information. Nowadays used for most of secondary, tertiary, and off-line storage. In 1950s and 1960s, it was also used for primary storage, in the form of magnetic core memory.

Volatile memory

Requires constant power to maintain the stored information. The fastest memory technologies of today are volatile ones (not a universal rule). Since primary storage is required to be very fast, it predominantly uses volatile memory.

Differentiation

Dynamic random access memory

A form of volatile memory which also requires the stored information to be periodically re-read and re-written, or refreshed, otherwise it would vanish.

Static memory

A form of volatile memory similar to DRAM with the exception that it never needs to be refreshed.

Mutability

Read/write storage or mutable storage

Allows information to be overwritten at any time. A computer without some amount of read/write storage for primary storage purposes would be useless for many tasks. Modern computers typically use read/write storage also for secondary storage.

Read only storage

Retains the information stored at the time of manufacture, and **write once storage** (Write Once Read Many) allows the information to be written only once at some point after manufacture. These are called **immutable storage**. Immutable storage is used for tertiary and off-line storage. Examples include CD-ROM and CD-R.

Slow write, fast read storage

Read/write storage which allows information to be overwritten multiple times, but with the write operation being much slower than the read operation. Examples include CD-RW and flash memory.

Accessibility

Random access

Any location in storage can be accessed at any moment in approximately the same amount of time. Such characteristic is well suited for primary and secondary storage.

Sequential access

The accessing of pieces of information will be in a serial order, one after the other; therefore the time to access a particular piece of information depends upon which piece of information was last accessed. Such characteristic is typical of off-line storage.

Addressability

Location-addressable

Each individually accessible unit of information in storage is selected with its numerical memory address. In modern computers, location-addressable storage usually limits to primary storage, accessed internally by computer programs, since location-addressability is very efficient, but burdensome for humans.

File addressable

Information is divided into *files* of variable length, and a particular file is selected with human-readable directory and file names. The underlying device is still location-addressable, but the operating system of a computer provides the file system abstraction to make the operation more understandable. In modern computers, secondary, tertiary and off-line storage use file systems.

Content-addressable

Each individually accessible unit of information is selected based on the basis of (part of) the contents stored there. Content-addressable storage can be implemented using software (computer program) or hardware (computer device), with hardware being faster but more expensive option. Hardware content addressable memory is often used in a computer's CPU cache.

Capacity

Raw capacity

The total amount of stored information that a storage device or medium can hold. It is expressed as a quantity of bits or bytes (e.g. 10.4 megabytes).

Memory storage density

The compactness of stored information. It is the storage capacity of a medium divided with a unit of length, area or volume (e.g. 1.2 megabytes per square inch).

Performance

Latency

The time it takes to access a particular location in storage. The relevant unit of measurement is typically nanosecond for primary storage, millisecond for secondary storage, and second for tertiary storage. It may make sense to separate read latency and write latency, and in case of sequential access storage, minimum, maximum and average latency.

Throughput

The rate at which information can be read from or written to the storage. In computer data storage, throughput is usually expressed in terms of megabytes per second or MB/s, though bit rate may also be used. As with latency, read rate and write rate may need to be differentiated. Also accessing media sequentially, as opposed to randomly, typically yields maximum throughput.

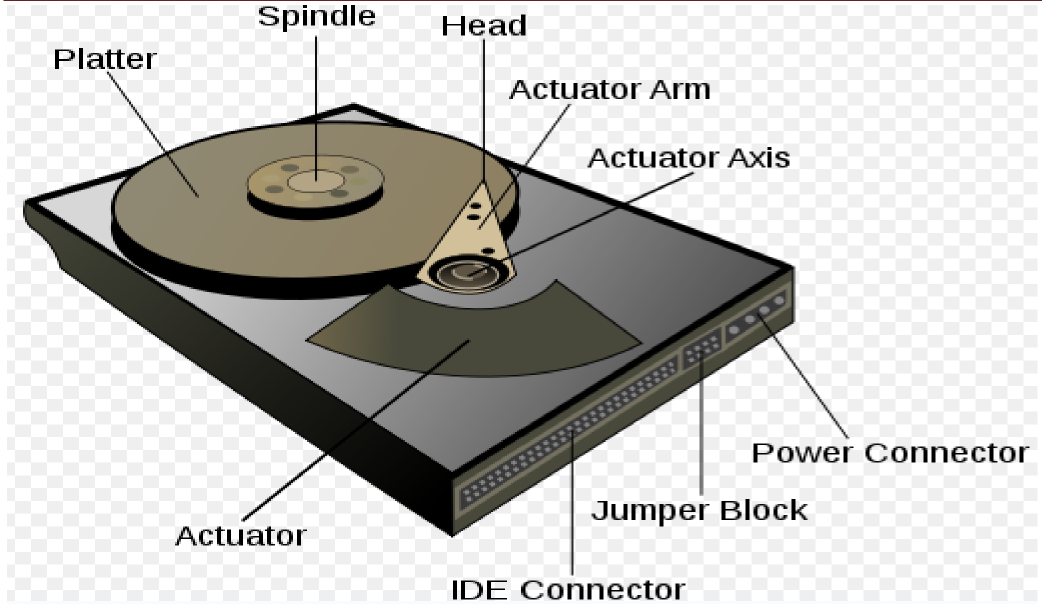
Magnetic

Magnetic storage uses different patterns of magnetization on a magnetically coated surface to store information. Magnetic storage is *non-volatile*. The information is accessed using one or more read/write heads which may contain one or more recording transducers. A read/write head only covers a part of the surface so that the head or medium or both must be moved relative to another in order to access data. In modern computers, magnetic storage will take these forms:

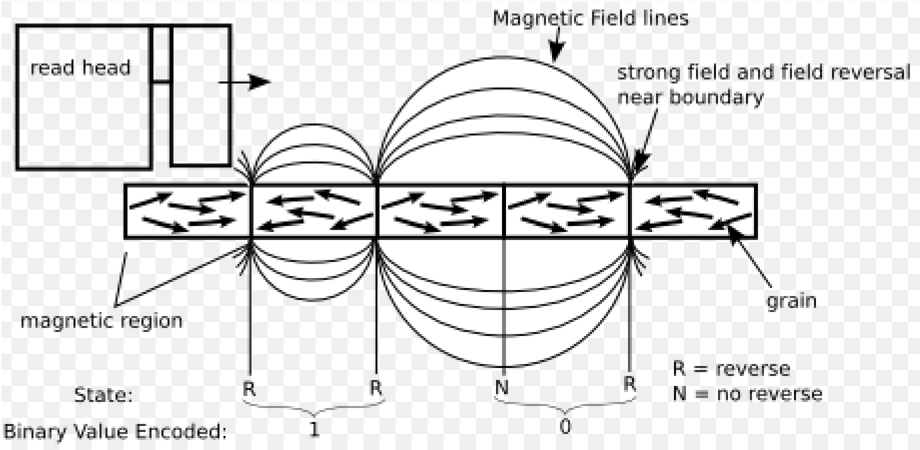
- Magnetic disk
  - Floppy disk, used for off-line storage
  - Hard disk drive, used for secondary storage
- Magnetic tape data storage, used for tertiary and off-line storage

Hard Disk Technology

Diagram of a computer hard disk drive



HDDs record data by magnetizing ferromagnetic material directionally, to represent either a 0 or a 1 binary digit. They read the data back by detecting the magnetization of the material. A typical HDD design consists of a spindle that holds one or more flat circular disks called platters, onto which the data is recorded. The platters are made from a non-magnetic material, usually aluminum alloy or glass, and are coated with a thin layer of magnetic material, typically 10-20 nm in thickness with an outer layer of carbon for protection.



The platters are spun at very high speeds. Information is written to a platter as it rotates past devices called read-and-write heads that operate very close (tens of nanometers in new drives) over the magnetic surface. The read-and-write head is used to detect and modify the magnetization of the material immediately under it. There is one head for each magnetic platter surface on the spindle, mounted on a common arm. An actuator arm (or access arm) moves the heads on an arc (roughly radially) across the platters as they spin, allowing each head to access almost the entire surface of the platter as it spins. The arm is moved using a voice coil actuator or in some older designs a stepper motor.

The magnetic surface of each platter is conceptually divided into many small sub-micrometre-sized magnetic regions, each of which is used to encode a single binary unit of information. Initially the regions were oriented horizontally, but beginning about 2005, the orientation was changed to perpendicular. Due to the polycrystalline nature of the magnetic material each of these magnetic regions is composed of a few hundred magnetic grains. Magnetic grains are typically 10 nm in size and each form a single magnetic domain. Each magnetic region in total forms a magnetic dipole which generates a highly localized magnetic field nearby. A write head magnetizes a region by generating a strong local magnetic field. Early HDDs used an electromagnet both to magnetize the region and to then read its magnetic field by using electromagnetic induction. Later versions of inductive heads included metal in Gap (MIG) heads and thin film heads. As data density increased, read heads using magnetoresistance (MR) came into use; the electrical resistance of the head changed according to the strength of the magnetism from the platter. Later development made use of spintronics; in these heads, the magnetoresistive effect was much greater than in earlier types, and was dubbed "giant" magnetoresistance (GMR). In today's heads, the read and write elements are separate, but in close proximity, on the head portion of an actuator arm. The read element is typically magneto-resistive while the write element is typically thin-film inductive.

HD heads are kept from contacting the platter surface by the air that is extremely close to the platter; that air moves at, or close to, the platter speed. The record and playback head are mounted on a block called a slider, and the surface next to the platter is shaped to keep it just barely out of contact. It's a type of air bearing.

In modern drives, the small size of the magnetic regions creates the danger that their magnetic state might be lost because of thermal effects. To counter this, the platters are coated with two parallel magnetic layers, separated by a 3-atom-thick layer of the non-magnetic element ruthenium, and the two layers are magnetized in opposite orientation, thus reinforcing each other. Another technology used to overcome thermal effects to allow greater recording densities is perpendicular recording, first shipped in 2005, as of 2007 the technology was used in many HDDs.

The grain boundaries turn out to be very important in HDD design. The reason is that, the grains are very small and close to each other, so the coupling between adjacent grains is very strong. When one grain is magnetized, the adjacent grains tend to be aligned parallel to it or demagnetized. Then both the stability of the data and signal-to-noise ratio will be sabotaged. A clear grain boundary can weaken the coupling of the grains and subsequently increase the signal-to-noise ratio. In longitudinal recording, the single-domain grains have uniaxial anisotropy with easy axes lying in the film plane. The consequence of this arrangement is that adjacent magnets repel each other. Therefore the magnetostatic energy is so large that it is difficult to increase areal density. Perpendicular recording media, on the other hand, has the easy axis of the grains oriented perpendicular to the disk plane. Adjacent magnets attract to each other and magnetostatic energy are much lower. So, much higher areal density can be achieved in perpendicular recording. Another unique feature in perpendicular recording is that a soft magnetic underlayer are incorporated into the recording disk.This underlayer is used to conduct writing magnetic flux so that the writing is more efficient. This will be discussed in writing process. Therefore, a higher anisotropy medium film, such as L10-FePt and rare-earth magnets, can be used.

Error handling

Modern drives also make extensive use of Error Correcting Codes (ECCs), particularly Reed–Solomon error correction. These techniques store extra bits for each block of data that are determined by mathematical formulas. The extra bits allow many errors to be fixed. While these extra bits take up space on the hard drive, they allow higher recording densities to be employed, resulting in much larger storage capacity for user data. In 2009, in the newest drives, low-density parity-check codes (LDPC) are supplanting Reed-Solomon. LDPC codes enable performance close to the Shannon Limit and thus allow for the highest storage density available.

Typical hard drives attempt to "remap" the data in a physical sector that is going bad to a spare physical sector—hopefully while the number of errors in that bad sector is still small enough that the ECC can completely recover the data without loss.

Architecture

A hard disk drive with the platters and motor hub removed showing the copper colored stator coils surrounding a bearing at the center of the spindle motor. The orange stripe along the side of the arm is a thin printed-circuit cable. The spindle bearing is in the center.

A typical hard drive has two electric motors, one to spin the disks and one to position the read/write head assembly. The disk motor has an external rotor attached to the platters; the stator windings are fixed in place. The actuator has a read-write head under the tip of its very end (near center); a thin printed-circuit cable connects the read-write head to the hub of the actuator. A flexible, somewhat 'U'-shaped, ribbon cable, seen edge-on below and to the left of the actuator arm in the first image and more clearly in the second, continues the connection from the head to the controller board on the opposite side.

Capacity and access speed

PC hard disk drive capacity (in GB). The vertical axis is logarithmic, so the fit line corresponds to exponential growth.

Using rigid disks and sealing the unit allows much tighter tolerances than in a floppy disk drive. Consequently, hard disk drives can store much more data than floppy disk drives and can access and transmit it faster.

- As of April 2009, the highest capacity consumer HDDs are 2 TB.
- A typical "desktop HDD" might store between 120 GB and 2 TB although rarely above 500GB of data (based on US market data rotate at 5,400 to 10,000 rpm, and have a media transfer rate of 1 Gbit/s or higher. (1 GB = 10^9 B; 1 Gbit/s = 10^9 bit/s)
- The fastest “enterprise” HDDs spin at 10,000 or 15,000 rpm, and can achieve sequential media transfer speeds above 1.6 Gbit/s. and a sustained transfer rate up to 1 Gbit/s. Drives running at 10,000 or 15,000 rpm use smaller platters to mitigate increased power requirements (as they have less air drag) and therefore generally have lower capacity than the highest capacity desktop drives.
- "Mobile HDDs", *i.e.*, laptop HDDs, which are physically smaller than their desktop and enterprise counterparts, tend to be slower and have lower capacity. A typical mobile HDD spins at 5,400 rpm, with 7,200 rpm models available for a slight price premium. Because of physically smaller platter(s), mobile HDDs generally have lower capacity than their physically larger counterparts.

The exponential increases in disk space and data access speeds of HDDs have enabled the commercial viability of consumer products that require large storage capacities, such as digital video recorders and digital audio players.

The main way to decrease access time is to increase rotational speed, thus reducing rotational delay, while the main way to increase throughput and storage capacity is to increase areal density. Based on historic trends, analysts predict a future growth in HDD bit density (and therefore capacity) of about 40% per year. Access times have not kept up with throughput increases, which themselves have not kept up with growth in storage capacity.

The first 3.5″ HDD marketed as able to store 1 TB was the Hitachi Deskstar 7K1000. It contains five platters at approximately 200 GB each, providing 1 TB (935.5 GiB) of usable space; note the difference between its capacity in decimal units (1 TB = 10^{12} bytes) and binary units (1 TiB = 1024 GiB = 2^{40} bytes). Hitachi has since been joined by Samsung (Samsung SpinPoint F1, which has 3 × 334 GB platters), Seagate and Western Digital in the 1 TB drive market.

In September 2009, Showa Denko announced capacity improvements in platters that they manufacture for HDD makers. A single 2.5" platter is able to hold 334 GB worth of data, and preliminary results for 3.5" indicate a 750 GB per platter capacity.

Optical

Optical storage, the typical Optical disc, stores information in deformities on the surface of a circular disc and reads this information by illuminating the surface with a laser diode and observing the reflection. Optical disc storage is *non-volatile*. The deformities may be permanent (read only media), formed once (write once media) or reversible (recordable or read/write media). The following forms are currently in common use:

- CD, CD-ROM, DVD, BD-ROM: Read only storage, used for mass distribution of digital information (music, video, computer programs)
- CD-R, DVD-R, DVD+R BD-R: Write once storage, used for tertiary and off-line storage
- CD-RW, DVD-RW, DVD+RW, DVD-RAM, BD-RE: Slow write, fast read storage, used for tertiary and off-line storage
- Ultra Density Optical or UDO is similar in capacity to BD-R or BD-RE and is slow write, fast read storage used for tertiary and off-line storage.

Magneto-optical disc storage is optical disc storage where the magnetic state on a ferromagnetic surface stores information. The information is read optically and written by combining magnetic and optical methods. Magneto-optical disc storage is *non-volatile*, *sequential access*, slow write, fast read storage used for tertiary and off-line storage.

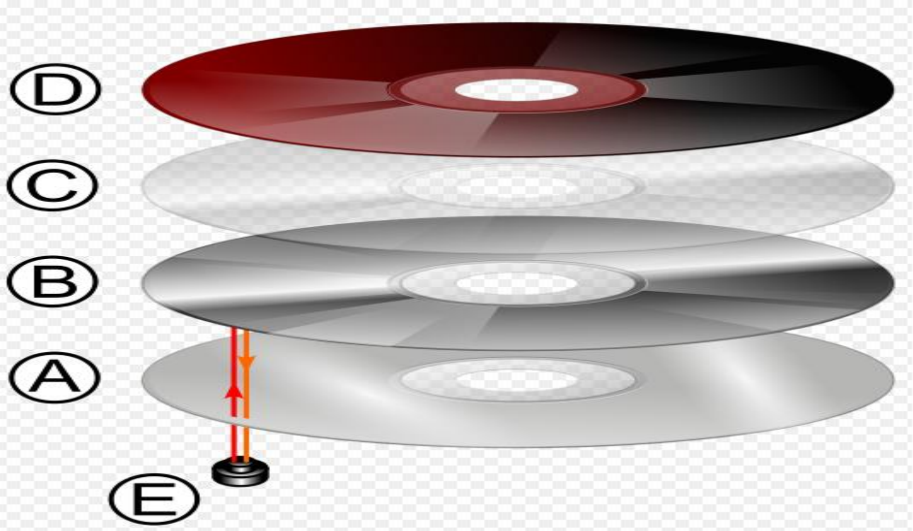
A **Compact Disc** (also known as a **CD**) is an optical disc used to store digital data. It was originally developed to store sound recordings exclusively, but later it also allowed the preservation of other types of data. Audio CDs have been commercially available since October 1982. In 2009, they remain the standard physical storage medium for audio.

Standard CDs have a diameter of 120 mm and can hold up to 80 minutes of uncompressed audio (700 MB of data). The Mini CD has various diameters ranging from 60 to 80 mm; they are sometimes used for CD singles or device drivers, storing up to 24 minutes of audio.

The technology was eventually adapted and expanded to encompass data storage CD-ROM, write-once audio and data storage CD-R, rewritable media CD-RW, Video Compact Discs (VCD), Super Video Compact Discs (SVCD), PhotoCD, PictureCD, CD-i, and Enhanced CD.

Physical details

Diagram of CD layers.



A. A polycarbonate disc layer has the data encoded by using bumps.
B. A shiny layer reflects the laser.
C. A layer of lacquer helps keep the shiny layer shiny.
D. Artwork is screen printed on the top of the disc.
E. A laser beam reads the CD and is reflected back to a sensor, which converts it into electronic data.

A CD is made from 1.2 mm thick, almost-pure polycarbonate plastic and weighs approximately 15–20 grams. From the center outward components are at the center (spindle) hole, the first-transition area (clamping ring), the clamping area (stacking ring), the second-transition area (mirror band), the information (data) area, and the rim.

A thin layer of aluminium or, more rarely, gold is applied to the surface to make it reflective, and is protected by a film of lacquer that is normally spin coated directly on top of the reflective layer, upon which the label print is applied. Common printing methods for CDs are screen-printing and offset printing.

CD data are stored as a series of tiny indentations known as "*pits*", encoded in a spiral track molded into the top of the polycarbonate layer. The areas between pits are known as "lands". Each pit is approximately 100 nm deep by 500 nm wide, and varies from 850 nm to 3.5 µm in length.

The distance between the tracks, the pitch, is 1.6 µm. A CD is read by focusing a 780 nm wavelength (near infrared) semiconductor laser through the bottom of the polycarbonate layer. The change in height between pits (actually ridges as seen by the laser) and lands results in a difference in intensity in the light reflected. By measuring the intensity change with a photodiode, the data can be read from the disc.

The pits and lands themselves do not directly represent the zeros and ones of binary data. Instead, Non-return-to-zero, inverted (NRZI) encoding is used: a change from pit to land or land to pit indicates a one, while no change indicates a series of zeros. There must be at least two and no more than ten zeros between each one, which is defined by the length of the pit. This in turn is decoded by reversing the Eight-to-Fourteen Modulation used in mastering the disc, and then reversing the Cross-Interleaved Reed-Solomon Coding, finally revealing the raw data stored on the disc.

CDs are susceptible to damage from both daily use and environmental exposure. Pits are much closer to the label side of a disc, so that defects and dirt on the clear side can be out of focus during playback. Consequently, CDs suffer more scratch damage on the label side whereas scratches on the clear side can be repaired by refilling them with similar refractive plastic, or by careful polishing. Initial music CDs were known to suffer from "CD rot", or "laser rot", in which the internal reflective layer degrades. When this occurs the CD may become unplayable.

Disc shapes and diameters

A Mini-CD is 8 centimetres in diameter.

The digital data on a CD begin at the center of the disc and proceeds toward the edge, which allows adaptation to the different size formats available. Standard CDs are available in two sizes. By far the most common is 120 mm in diameter, with a 74- or 80-minute audio capacity and a 650 or 700 MB data capacity. This diameter has also been adopted by later formats, including Super Audio CD, DVD, HD DVD, and Blu-ray Disc. 80 mm discs ("Mini CDs") were originally designed for CD singles and can hold up to 21 minutes of music or 184 MB of data but never really became popular. Today, nearly every single is released on a 120 mm CD, called a Maxi single.

UNIT 5

System Organization

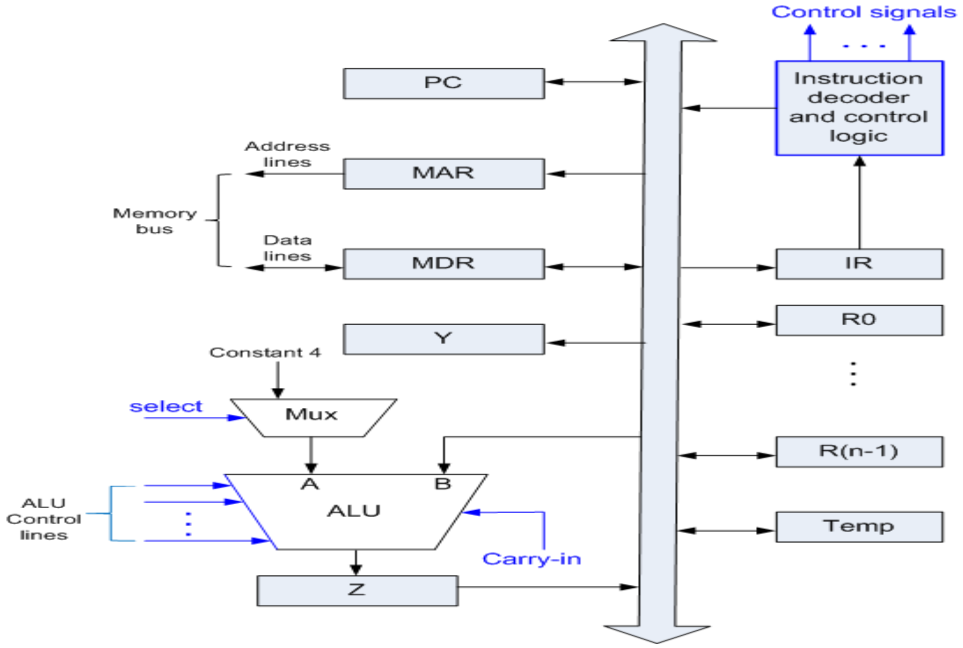
Buses

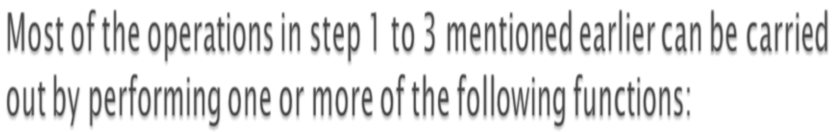
- **Execution of one instruction requires the following three steps to be performed by the CPU:**

1. **Fetch the contents of the memory location pointed at by the PC. The contents of this location are intepreted as an instruction to be executed. Hence, they are stored in the instruction register (IR). Simbolically, this can be written as:**
   IR ⇓[[PC]]
2. **Assuming that the memory is byte addressable, increment the contents of the PC by 4, that is**
   PC ⇓ [PC] + 4
3. **Carry out the actions specified by the instruction stored in the IR**

- **But, in cases where an instruction occupies more than one word, steps 1 and 2 must be repeated as many times as necessary to fetch the complete instruction.**
- **Two first steps are ussually referred to as the fetch phase.**
- **Step 3 constitutes the execution phase**

SINGLE BUS ORGANIZATION OF THE DATAPATH INSIDE A PROCESSOR




Most of the operations in step 1 to 3 mentioned earlier can be carried out by performing one or more of the following functions:

But, in cases where an instruction occupies more than one word, steps 1 and 2 must be repeated as many times as necessary to fetch the complete instruction.

- Two first steps are ussually referred to as the fetch phase.
- Step 3 constitutes the execution phase

Fetch the contents of a given memory location and load them into a CPU
Register

- Store a word of data from a CPU register into a given memory location.
- Transfer a word of data from one CPU register to another or to ALU.
- Perform an arithmetic or logic operation, and store the result in a CPU register.

REGISTER TRANSFER:

The input and output gates for register Ri are controlled by the signals Riin and Riout, respectively.

- Thus, when Riin is set to 1, the data available on the common bus is loaded into Ri.
- Similarly, when Riout is set to 1, the contents of register Ri are placed on the bus.
- While Riout is equal to 0, the bus can be used for transferring data from other registers.

Let us now consider data transfer between two registers. For example, to transfer the contents of register R1 to R4, the following actions are needed:

Enable the output gate of register R1 by setting R1out to 1. This places the contents of R1 on the CPU bus.

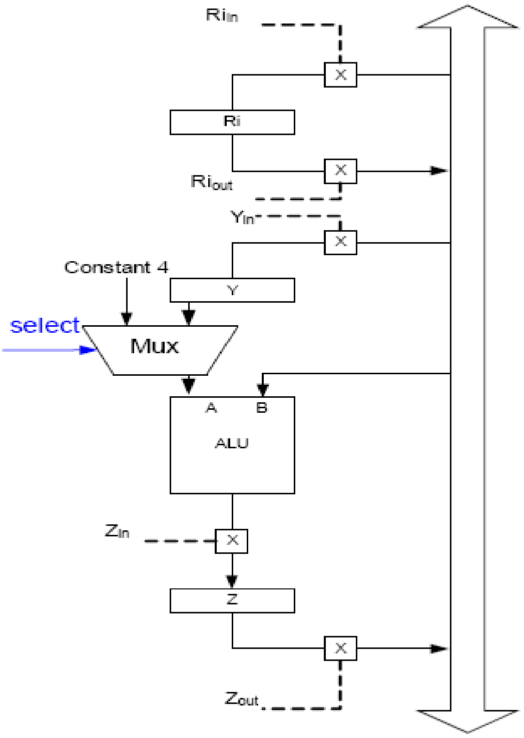
- Enable the input gate of register R4 by setting R4in to 1. This loads data from the CPU bus into register R4.
- *This data transfer can be represented symbolically as R1out, R4in*

PERFORMING AN ARITHMETIC OR LOGIC OPERATION

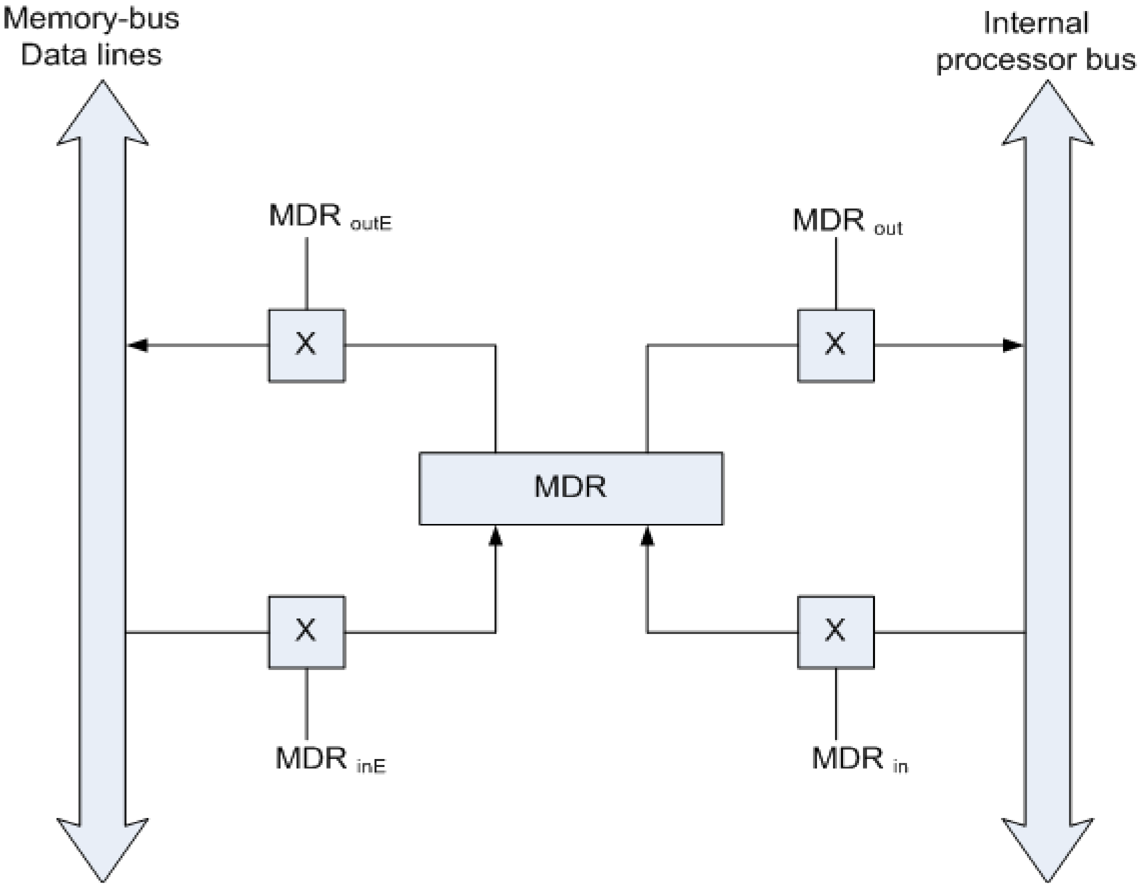
A SEQUENCE OF OPERATIONS TO ADD THE CONTENTS OF REGISTER R1 TO THOSE OF REGISTER R2 AND STORE THE RESULT IN REGISTER R3 IS:

- R1out, Yin
- R2out, Select Y, Add, Zin
- Zout, R3in

Input and output gating for the registers



FETCHING A WORD FROM MEMORY:



CPU transfers the address of the required information word to the memory address register (MAR). Address of the required word is transferred to the main memory.

□ Meanwhile, the CPU uses the control lines of the memory bus to indicate that a read operation is required.

□ After issuing this request, the CPU waits until it receives an answer from the memory, informing it that the requested function has been completed. This is accomplished through the use of another control signal on the memory bus, which will be referred to as Memory Function Completed (MFC).

□ The memory sets this signal to 1 to indicate that the contents of the specified location in the memory have been read and are available on the data lines of the memory bus.

□ We will assume that as soon as the MFC signal is set to 1, the information on the data lines is loaded into MDR and is thus available for use inside the CPU. This completes the memory fetch operation.

The actions needed for instruction Move (R1), R2 are:

MAR □ [R1]

□ Start Read operation on the memory bus
□ Wait for the MFC response from the memory

□ Load MDR from the memory bus
□ R2 □ [MDR]

Signals activated for that problem are:

R1out, MARin, Read
MDRinE, WMFC
MDRout, R2in

Storing a word in Memory

That is similar procedure with fetching a word from memory.

□ The desired address is loaded into MAR
□ Then data to be written are loaded into MDR, and a write command is issued.
□ If we assume that the data word to be stored in the memory is in R2 and that the memory address is in R1, the Write operation requires the following sequence :

□ MAR □ [R1]
□ MDR □ [R2]
□ Write
□ Wait for the MFC

Move R2, (R1) requires the following sequence (signal):

R1out, MARin
R2out, MDRin. Write
MDRoutE,WMFC

Execution of a complete Instruction

Consider the instruction :
Add (R3), R1
□ Executing this instruction requires the following actions :

1. Fetch the instruction
2. Fetch the first operand (the contents of the memory location pointed to by R3)
3. Perform the addition
4. Load the result into R1

Control Sequence for instruction Add (R3), R1:

PCout, MARin, Read, Select4, Add, Zin
□ Zout, PCin, Yin, Wait for the MFC
□ MDRout, IRin

□ R3out, MARin, Read
□ R1out, Yin, Wait for MFC
□ MDRout, Select Y, Add, Zin
□ Zout, R1in, End

Branch Instructions:

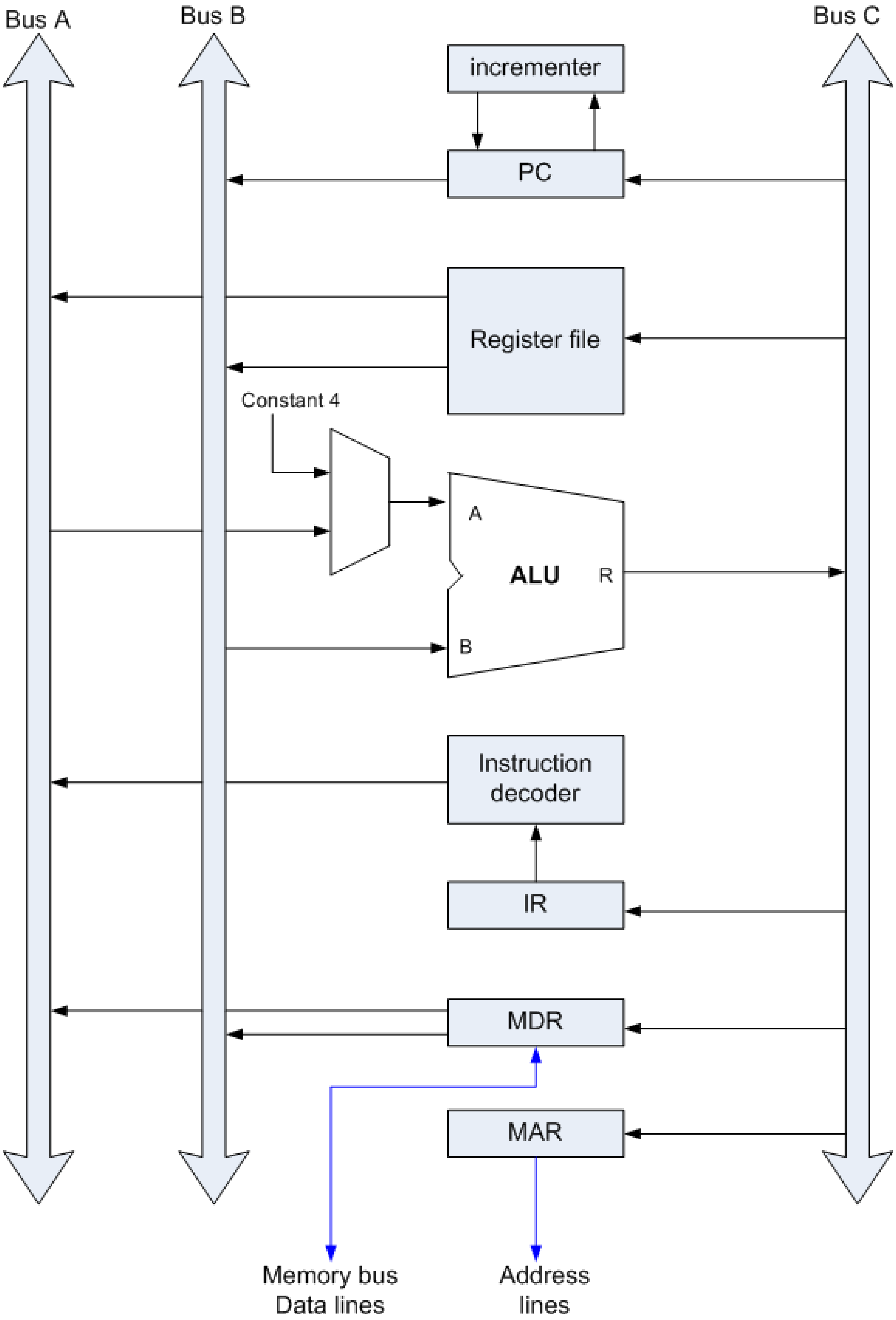
PCout, MARin, Read, Select4, Add, Zin
□ Zout, PCin, Yin, Wait for the MFC (WFMC)
□ MDRout, Irin
□ offset_field_of_IRout, Add, Zin
□ Zout, PCin, End

Multiple bus architecture

One solution to the bandwidth limitation of a single bus is to simply add additional buses. Consider the architecture shown in Figure 2.2 that contains *N* processors, P1 P2 P*N*, each having its own private cache, and all connected to a shared memory by *B* buses B1 B2 B*B*. The shared memory consists of *M* interleaved banks M1 M2 M*M* to allow simultaneous memory requests concurrent access to the shared memory. This avoids the loss in performance that occurs if those accesses must be serialized, which is the case when there is only one memory bank. Each processor is connected to every bus and so is each memory bank. When a processor needs to access a particular bank, it has *B* buses from which to choose. Thus each processor-memory pair is connected by several redundant paths, which implies that the failure of one or more paths can, in principle, be tolerated at the cost of some degradation in system performance.

In a multiple bus system several processors may attempt to access the shared memory simultaneously. To deal with this, a policy must be implemented that allocates the available buses to the processors making requests to memory. In particular, the policy must deal with the case when the number of processors exceeds *B*. For performance reasons this allocation must be carried out by hardware arbiters which, as we shall see,
add significantly to the complexity of the multiple bus interconnection network.

PCout, R=B, MARin, Read, IncPC
□ WFMC
□ MDRoutB, R=B, IRin
□ R4out, R5outB, SelectA, Add, R6in, End.


Bus A
Bus B
Bus C
incrementer
PC
Register file
Constant 4
A
ALU
R
B
Instruction decoder
IR
MDR
MAR
Memory bus Data lines
Address lines

Bus arbitration

In a single bus architecture when more than one device requests the bus, a controller called bus arbiter decides who gets the bus, this is called the bus arbitration.

Bus Master:

In computing, **bus mastering** is a feature supported by many bus architectures that enables a device connected to the bus to initiate transactions.

The procedure in bus communication that chooses between connected devices contending for control of the shared bus; the device currently in control of the bus is often termed the bus master. Devices may be allocated differing priority levels that will determine the choice of bus master in case of contention. A device not currently bus master must request control of the bus before attempting to initiate a data transfer via the bus. The normal protocol is that only one device may be bus master at any time and that all other devices act as slaves to this master. Only a bus master may initiate a normal data transfer on the bus; slave devices respond to commands issued by the current bus master by supplying data requested or accepting data sent.

- ➢ Centralized arbitration
- ➢ Distributed arbitration

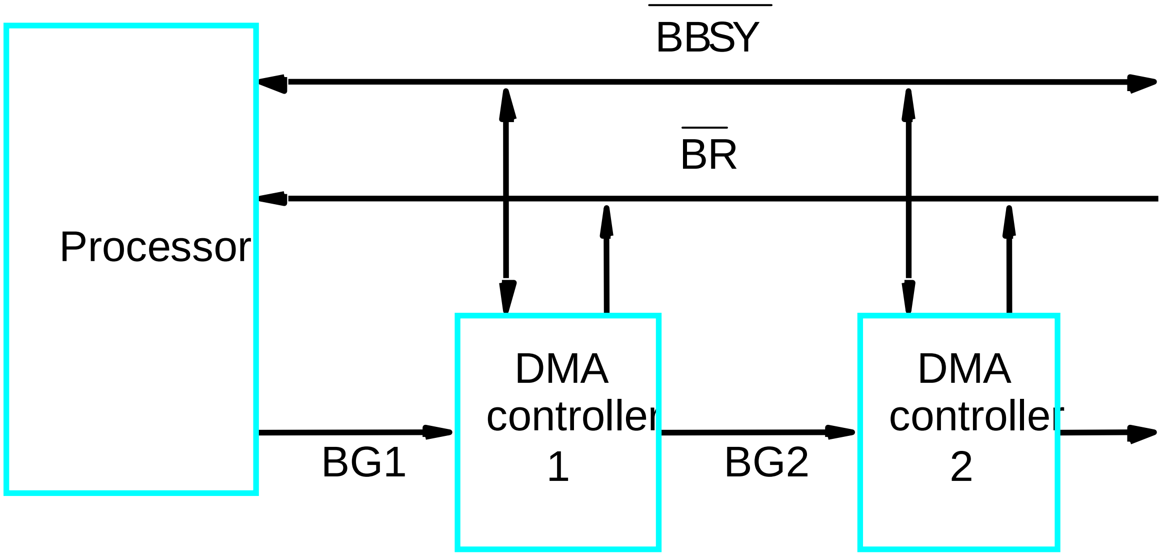


Figure. A simple arrangement for bus arbitration using a daisy chain.

- The bus arbiter may be the processor or a separate unit connected to the bus.
- One bus-request line and one bus-grant line form a daisy chain.
- This arrangement leads to considerable flexibility in determining the order.

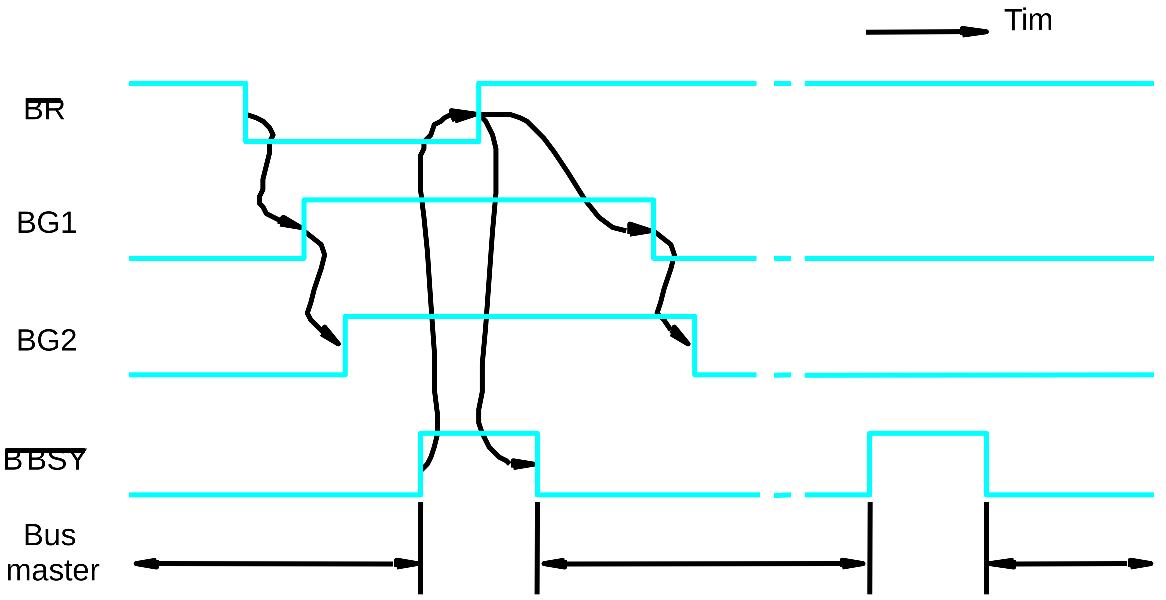
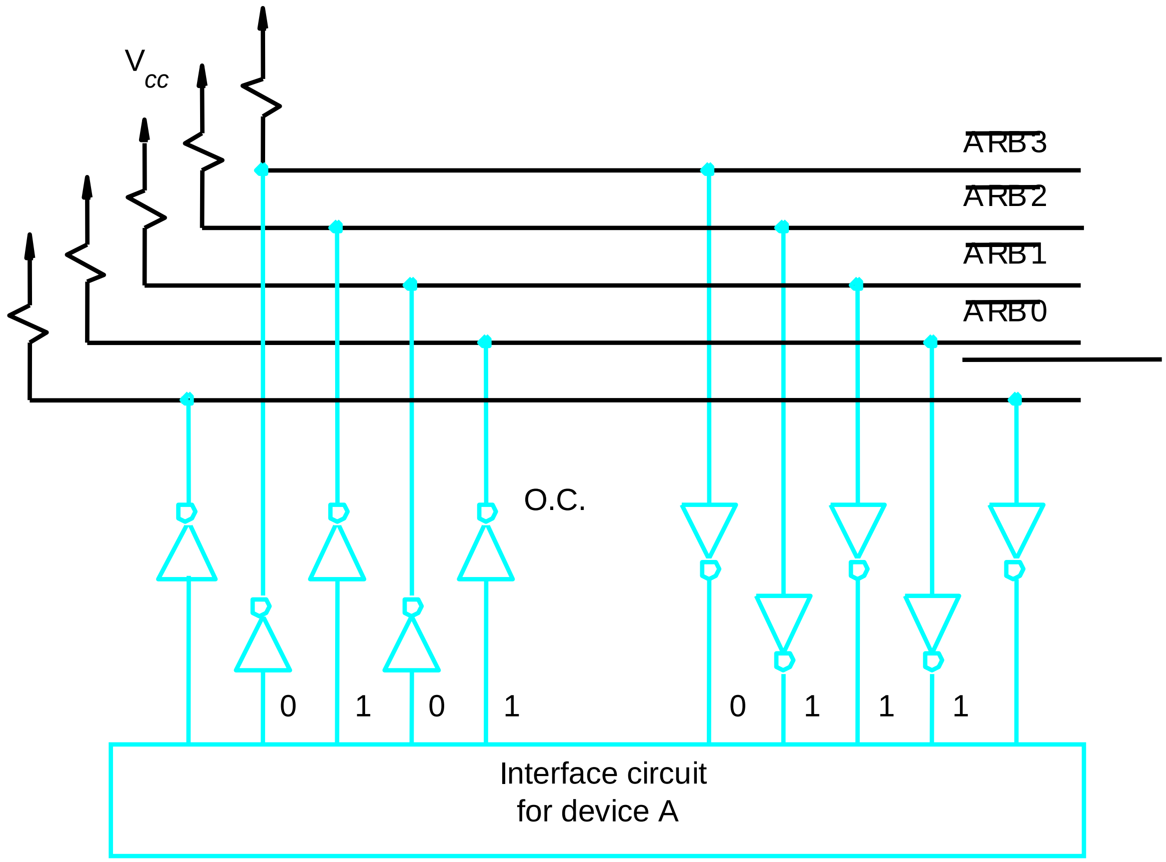


Fig. Sequence of Signals during transfer of mastership for the devices

Distributed Arbitration



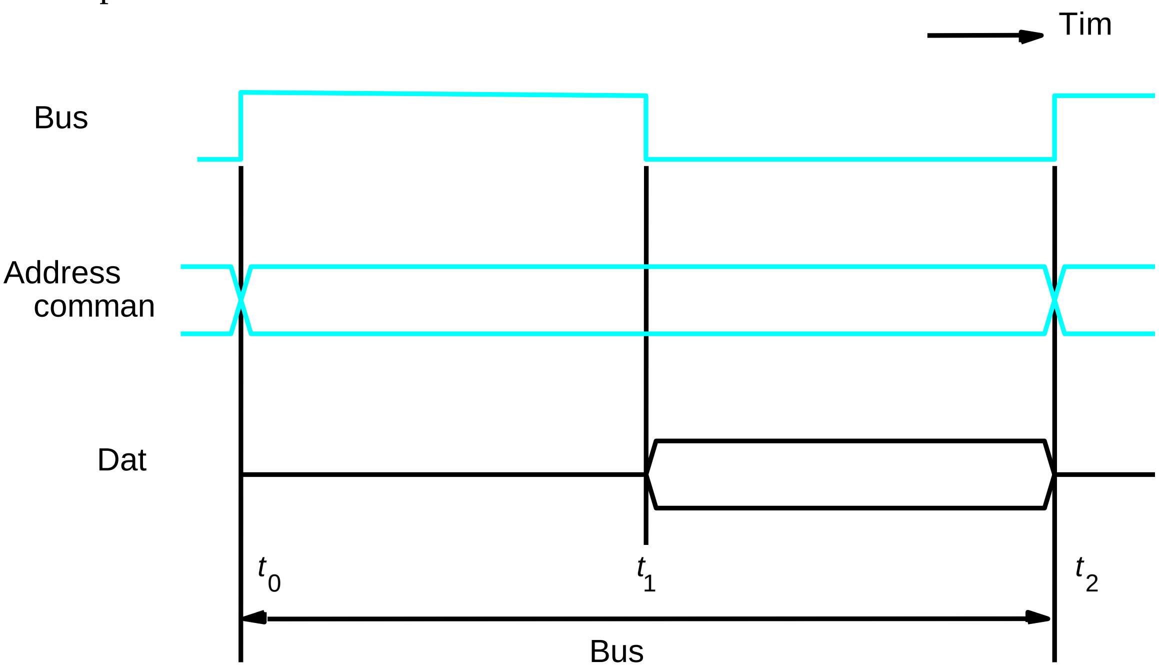
- All devices have equal responsibility in carrying out the arbitration process.
- Each device on the bus assigned an identification number.
- Place their ID numbers on four open-collector lines.

- A winner is selected as a result.

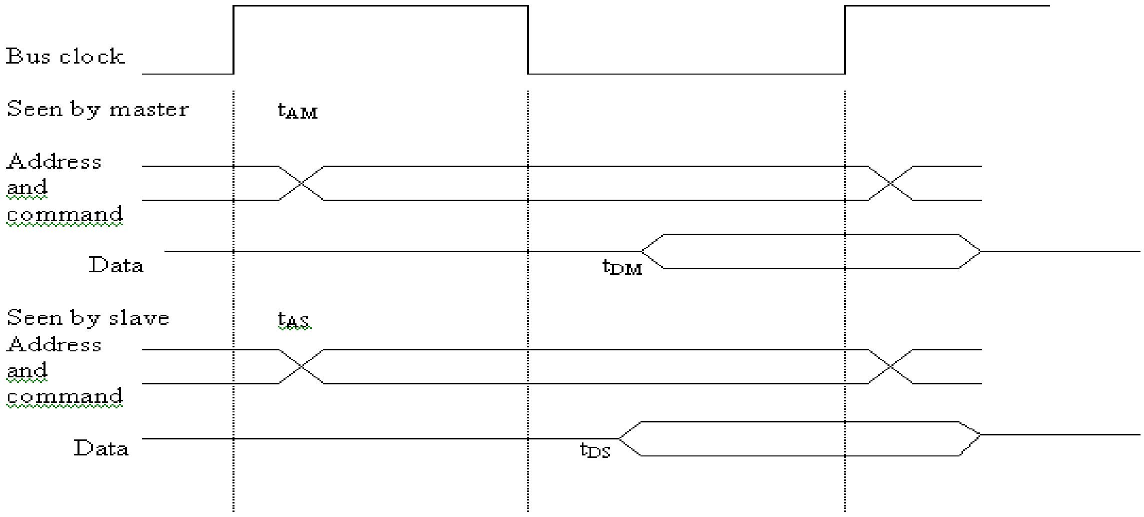
Types of Bus

Synchronous Bus

- All devices derive timing information from a common clock line.
- Each of these intervals constitutes a bus cycle during which one data transfer can take place.



Synchronous Bus Input Transfer



Asynchronous Bus

- Data transfers on the bus is based on the use of a handshake between the master and the salve.
- The common clock is replaced by two timing control lines, Master-ready and Slave-ready.

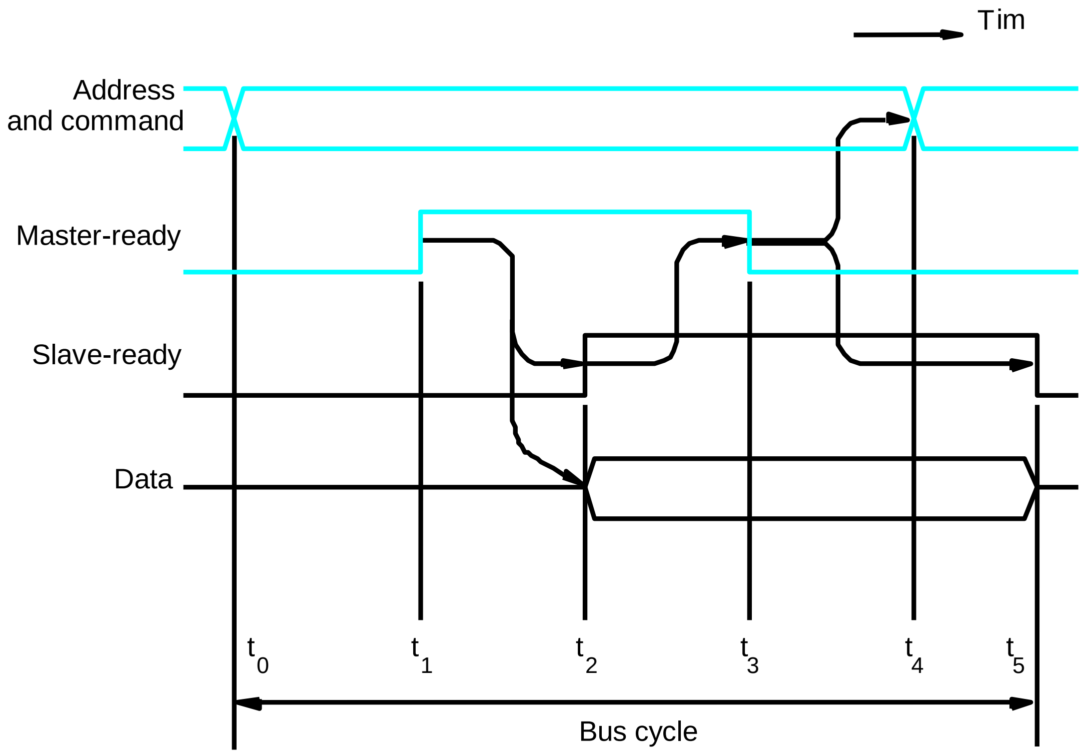


Figure. Handshake control of data transfer during an input operation

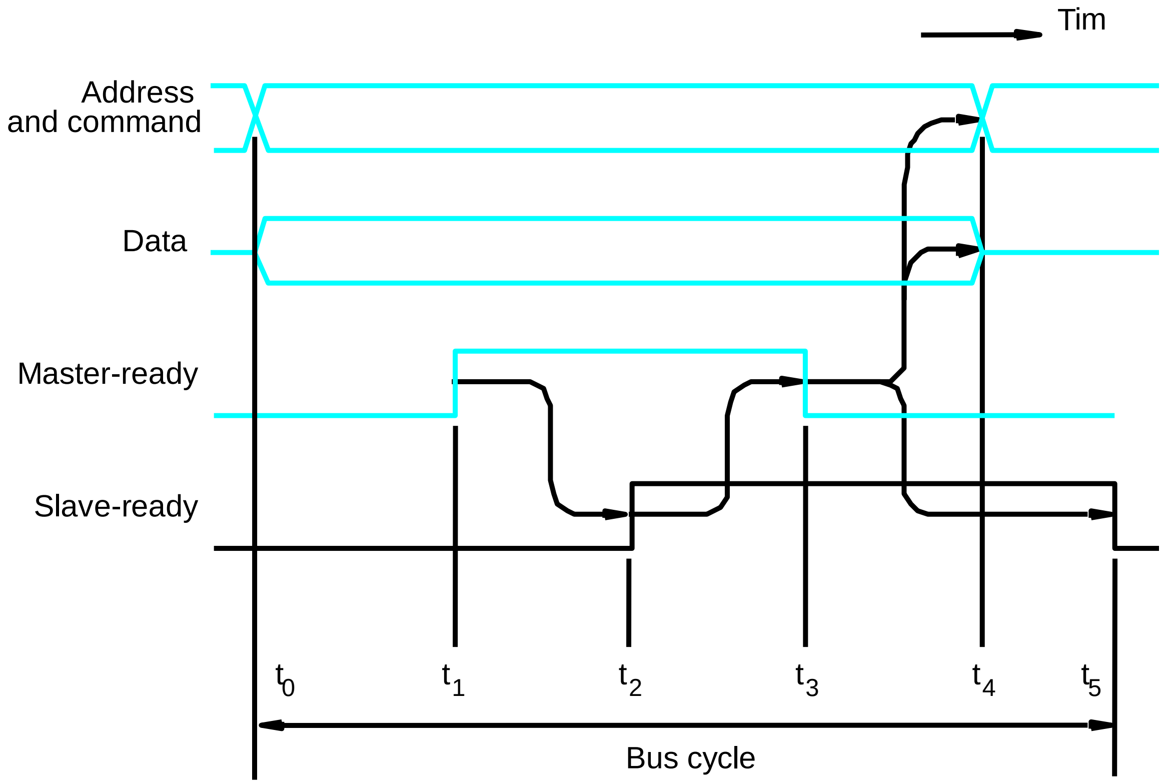


Figure. Handshake control of data transfer during an output operation

Input/Output Module

Interface to CPU and Memory

•Interface to one or more peripherals

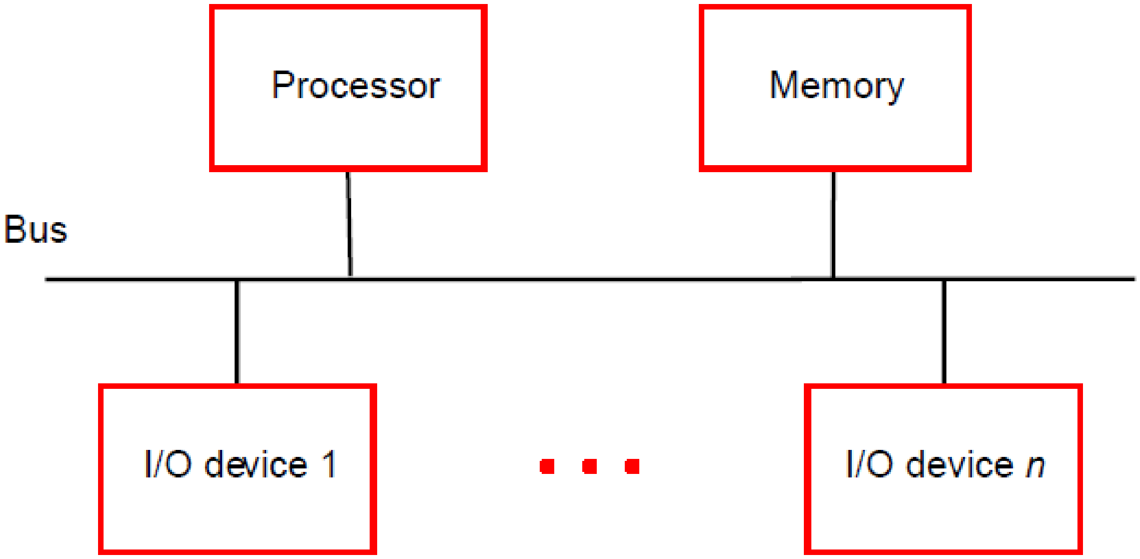
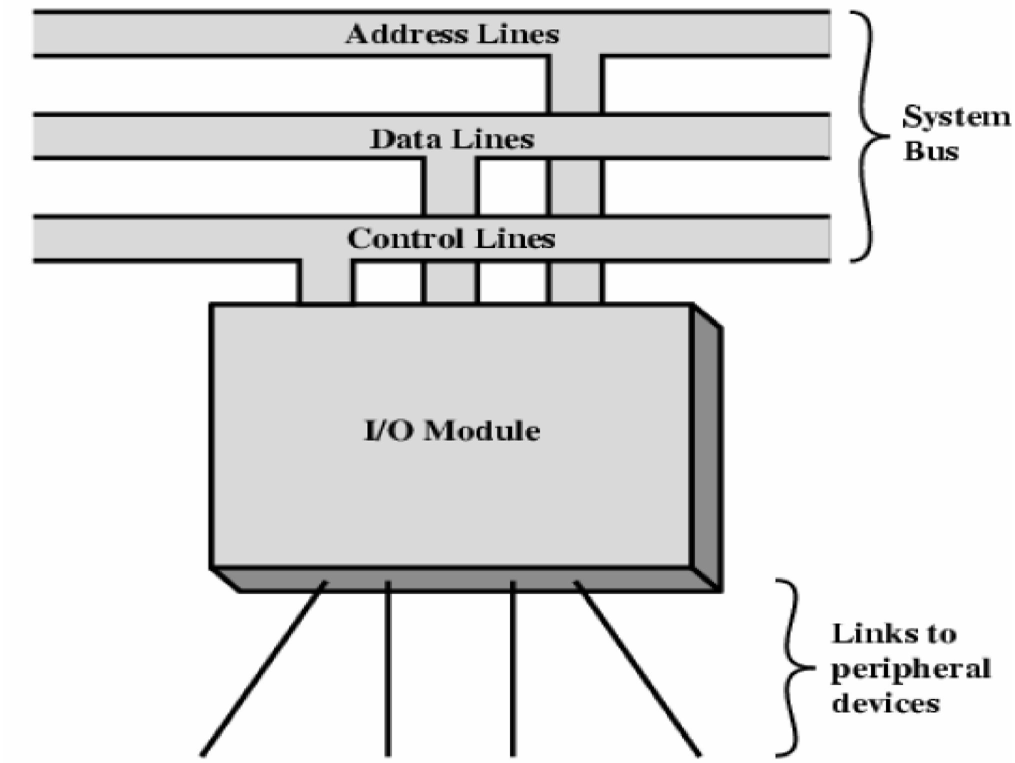


Figure: A single-bus structure.

Generic Model of IO Module



Interface for an IO Device:

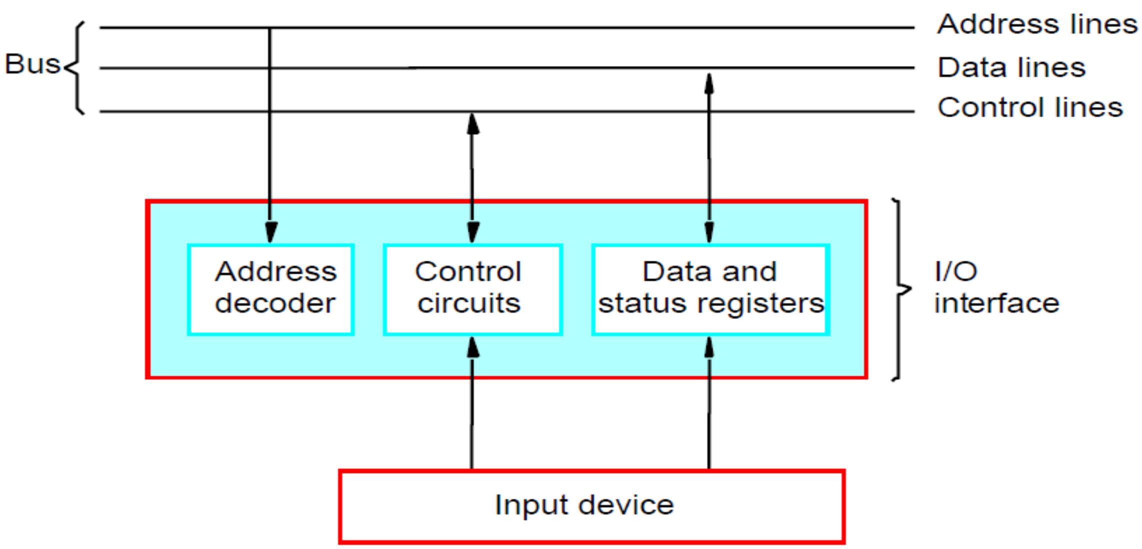


Figure I/O interface for an input device.

CPU checks I/O module device status •I/O module returns status
•If ready, CPU requests data transfer •I/O module gets data from device
•I/O module transfers data to CPU

Input Output Techniques

- Programmed
- Interrupt driven
- Direct Memory Access (DMA)

Programmed I/O

•CPU has direct control over I/O
–Sensing status
–Read/write commands
–Transferring data
•CPU waits for I/O module to complete operation
•Wastes CPU time

•CPU requests I/O operation
•I/O module performs operation
•I/O module sets status bits
•CPU checks status bits periodically
•I/O module does not inform CPU directly
•I/O module does not interrupt CPU
•CPU may wait or come back later

•Under programmed I/O data transfer is very like memory access (CPU viewpoint)
•Each device given unique identifier

•CPU commands contain identifier (address)

I/O Mapping

•Memory mapped I/O

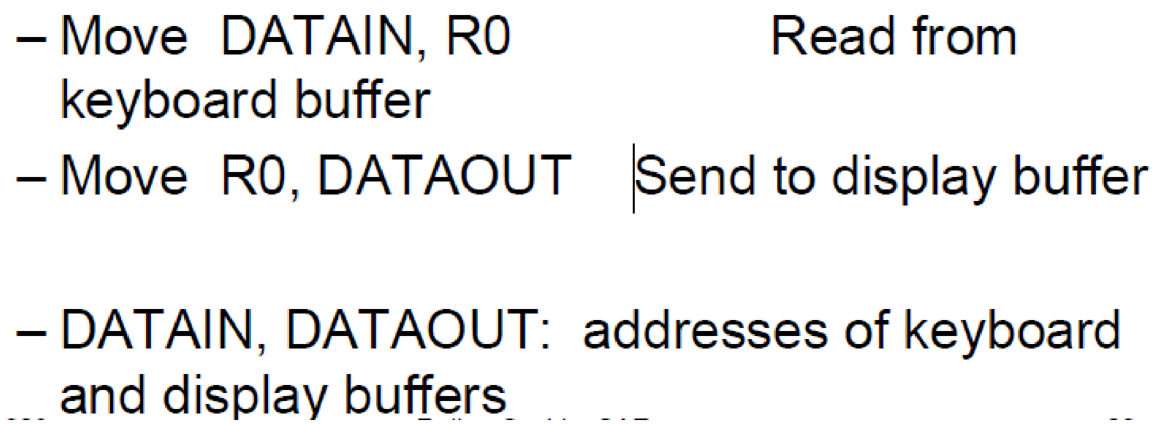
–Devices and memory share an address space
–I/O looks just like memory read/write
–No special commands for I/O
•Large selection of memory access commands available

•Isolated I/O

–Separate address spaces
–Need I/O or memory select lines
–Special commands for I/O
•Limited set

Memory Mapped IO:

•Input and output buffers use same address spaceas memory locations
•All instructions can access the buffer



– Move DATAIN, R0 Read from keyboard buffer

– Move R0, DATAOUT Send to display buffer

– DATAIN, DATAOUT: addresses of keyboard and display buffers

Interrupts

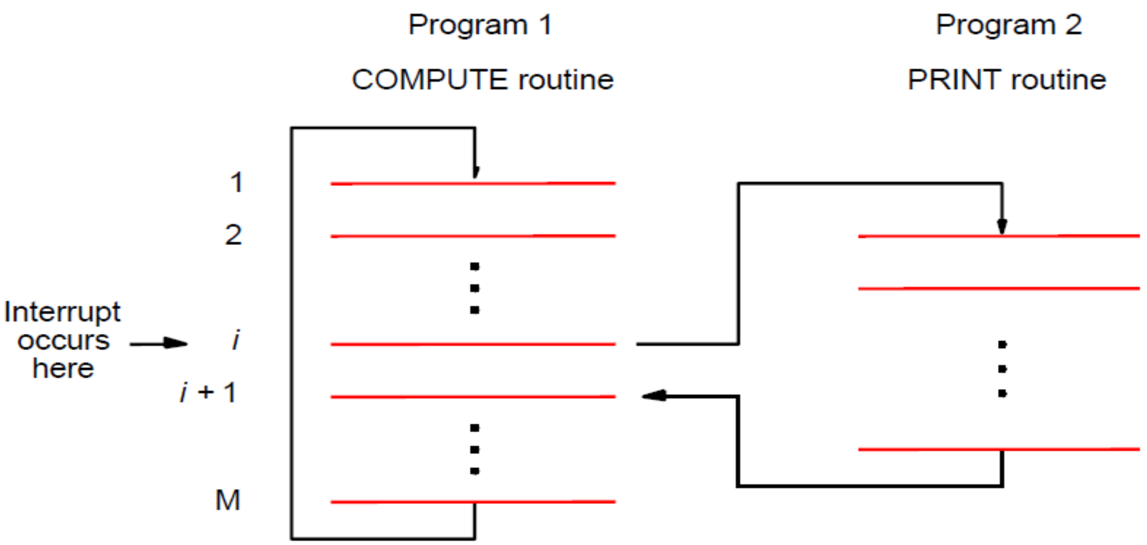
•Interrupt-request line

–Interrupt-request signal
–Interrupt-acknowledge signal

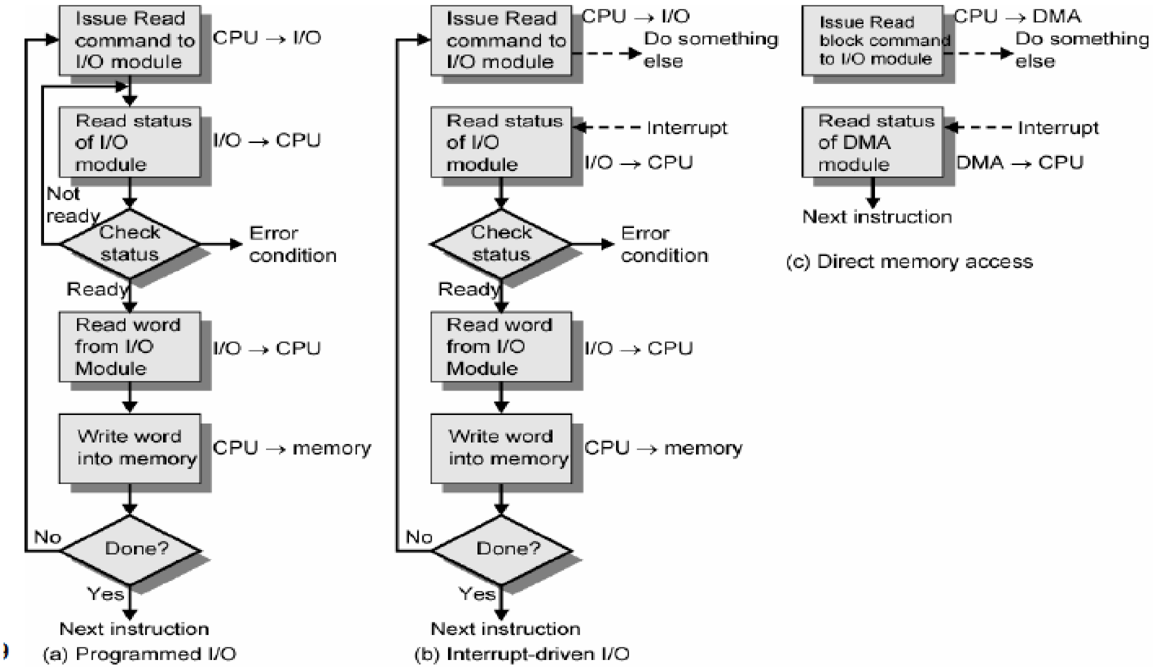
•Interrupt-service routine
–Similar to subroutine
–May have no relationship to program being executed at time of interrupt

•Program info must be saved
•Interrupt latency

Transfer of control through the use of interrupts



Three Techniques for Input of a Block of Data



Interrupts

Examples:

- power failing, arithmetic overflow
- I/O device request, OS call, page fault
- Invalid opcode, breakpoint, protection violation

Interrupts (aka faults, exceptions, traps) often require

- surprise jump (to vectored address)
- linking return address
- saving of PSW (including CCs)
- state change (e.g., to kernel mode)

Classifying Interrupts

1a. synchronous

- function of program state (e.g., overflow, page fault)

1b. asynchronous

- external device or hardware malfunction

2a. user request

- OS call

2b. coerced

- from OS or hardware (page fault, protection violation)

3a. User Maskable

User can disable processing

3b. Non-Maskable

User cannot disable processing

4a. Between Instructions

Usually asynchronous

4b. Within an instruction

Usually synchronous - Harder to deal with

5a. Resume

As if nothing happened? Program will continue execution

5b. Termination

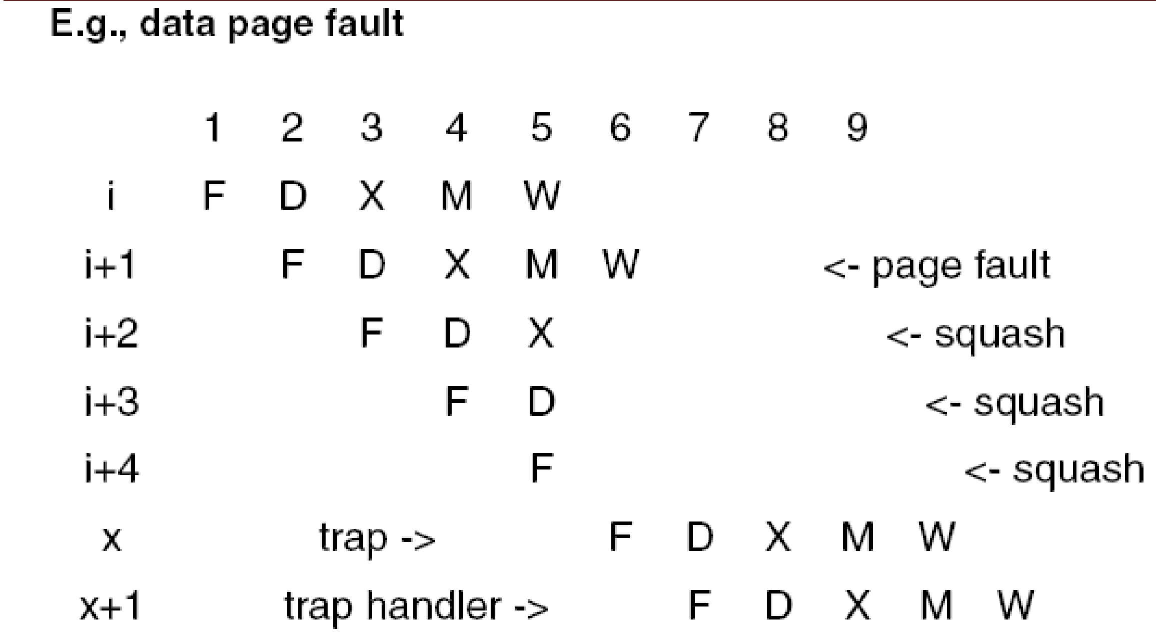
Handling Interrupts

Precise interrupts (sequential semantics)

- Complete instructions before the offending instr
- Squash (effects of) instructions after
- Save PC (& next PC with delayed branches)
- Force trap instruction into IF

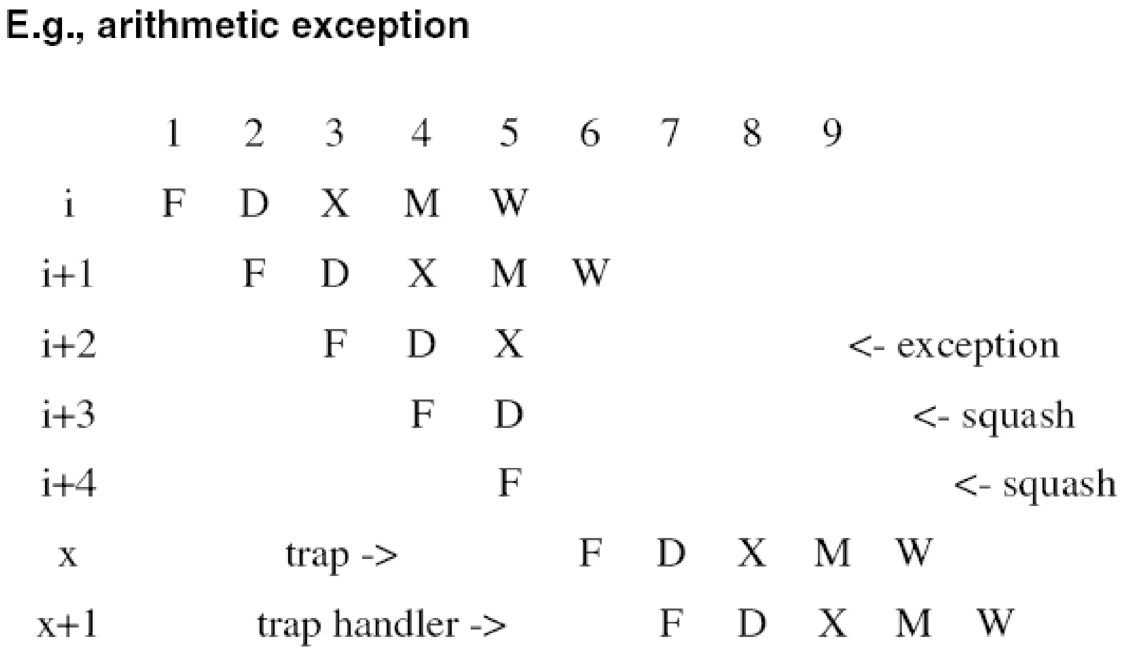
Must handle simultaneous interrupts

- IF, M - memory access (page fault, misaligned, protection)
- ID - illegal/privileged instruction
- EX - arithmetic exception



E.g., data page fault

	1	2	3	4	5	6	7	8	9	
i	F	D	X	M	W					
i+1		F	D	X	M	W				<- page fault
i+2			F	D	X					<- squash
i+3				F	D					<- squash
i+4					F					<- squash
x			trap ->			F	D	X	M	W
x+1			trap handler ->				F	D	X	M W



E.g., arithmetic exception

	1	2	3	4	5	6	7	8	9	
i	F	D	X	M	W					
i+1		F	D	X	M	W				
i+2			F	D	X					<- exception
i+3				F	D					<- squash
i+4					F					<- squash
x			trap ->			F	D	X	M	W
x+1			trap handler ->				F	D	X	M W

Out-of-Order Interrupts

Post interrupts

- check interrupt bit on entering WB
- precise interrupts
- longer latency

Handle immediately

- not fully precise
- interrupt may occur in order different from sequential CPU
- may cause implementation headaches!

Other complications

- odd bits of state (e.g., CC)
- early-writes (e.g., autoincrement)
- instruction buffers and prefetch logic
- dynamic scheduling
- out-of-order execution

Interrupts come at random times
Both Performance and Correctness

- frequent case not everything
- rare case MUST work correctly

Delayed Branches and Interrupts

What happens on interrupt while in delay slot

- next instruction is not sequential

Solution #1: save multiple PCs

- save current and next PC
- special return sequence, more complex hardware

Solution #2: single PC plus

- branch delay bit
- PC points to branch instruction
- SW Restrictions

Problems with interrupts

- DIVF f0, f2,f4
- ADDF f2,f8, f10
- SUBF f6, f4, f10

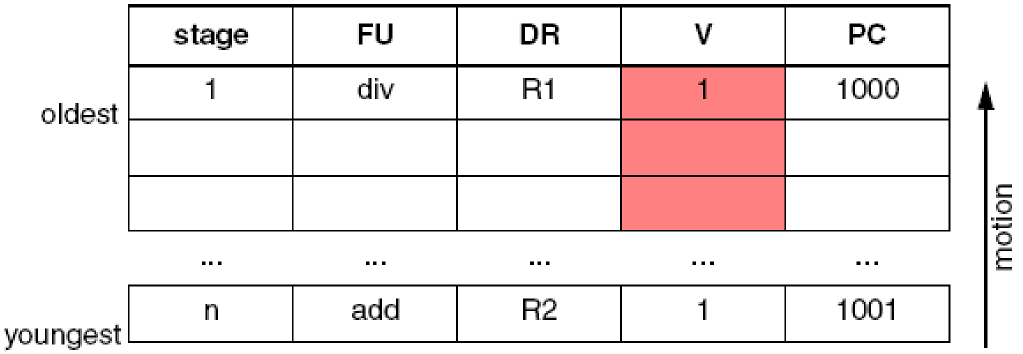
ADDF completes before DIVF

- Out-Of-Order completion
- Possible imprecise interrupts

Precice Interrupts

- Simple solution: Modify state only when all preceding insts. are *known* to be exception free.

Mechanism: *Result Shif Register*

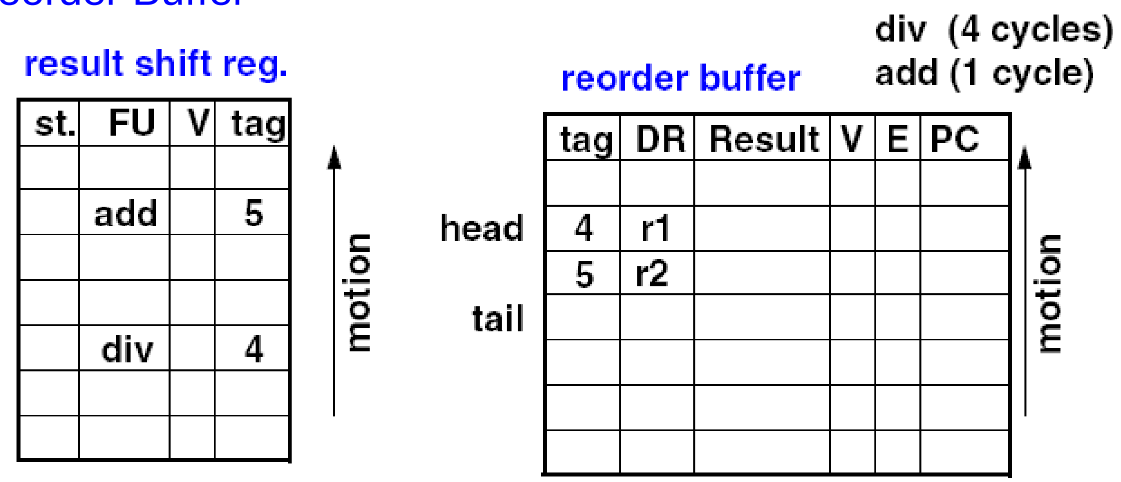


	stage	FU	DR	V	PC
oldest	1	div	R1	1	1000
	...	...	...	...	...
youngest	n	add	R2	1	1001

Reserve all stages for the duration of the instruction

Memory: Either stall stores at decode or use dummy store

Reorder Buffer



div (4 cycles)
add (1 cycle)

result shift reg.

st.	FU	V	tag
	add		5
	div		4

reorder buffer

	tag	DR	Result	V	E	PC
head	4	r1				
	5	r2				
tail						

- Out-of-order completition
- Commit: Write results to register file or memory
- Reorder buffer holds not yet committed state

INTERRUPT HANDLING

Handling Interrupts

• Many situations where the processor should ignore interrupt requests

–Interrupt-disable

–Interrupt-enable

•Typical scenario

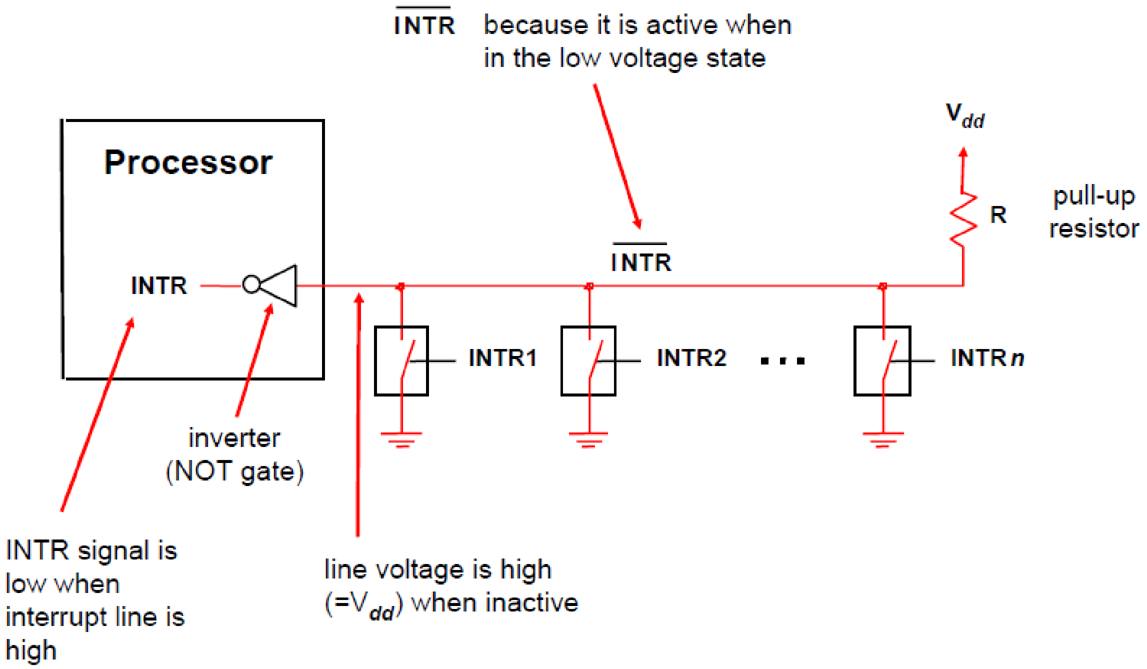
–Device raises interrupt request

–Processor interrupts program being executed

–Processor disables interrupts and acknowledges interrupt

–Interrupt-service routine executed

–Interrupts enabled and program execution resumed


$\overline{\text{INTR}}$ because it is active when in the low voltage state
V_{dd}
Processor
R
pull-up resistor
$\overline{\text{INTR}}$
INTR
INTR1
INTR2
. . .
INTRn
inverter (NOT gate)
INTR signal is low when interrupt line is high
line voltage is high (=V_{dd}) when inactive

An equivalent circuit for an open-drain bus used to implement a common interrupt-request line.

Handling Multiple Devices

Interrupt Priority

•During execution of interrupt-service routine

–Disable interrupts from devices at the same level priority or lower

–Continue to accept interrupt requests from higher priority devices

–Privileged instructions executed in supervisor mode

•Controlling device requests

–Interrupt-enable

•KEN, DEN

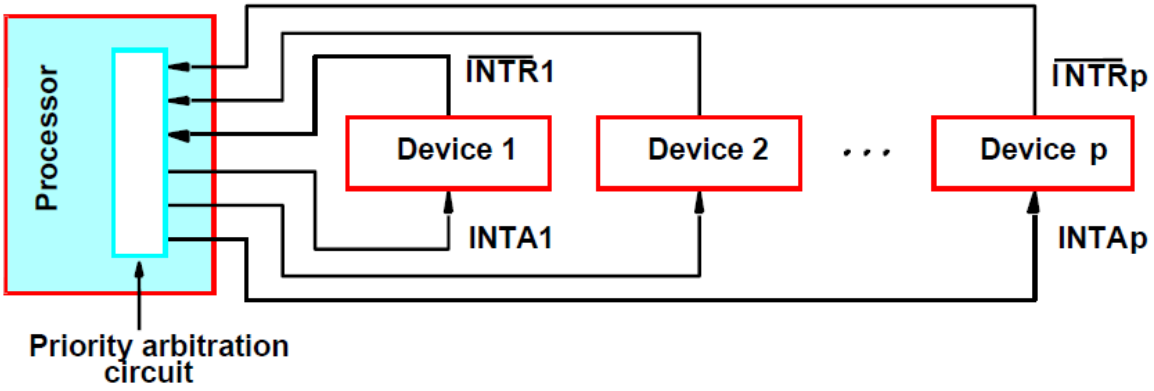
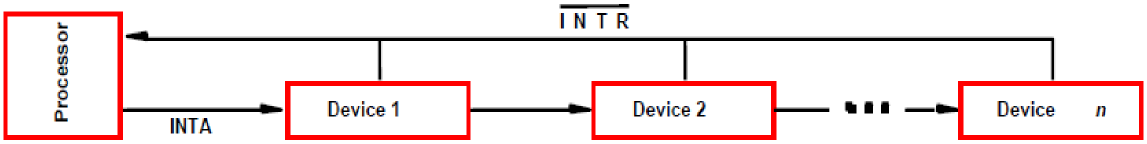


Figure Implementation of interrupt priority using individual interrupt-request and acknowledge lines.

Priority determined by the order in which processor accepts and acknowledges interrupts

Polled interrupts:Priority determined by the order in which processor polls the devices (polls their status registers)Vectored interrupts:Priority determined by the order in which processor tells deviceto put its code on the address lines (order of connection in the chain)



(a) Daisy chain

Daisy chaining of INTA:If device has not requested service, passes the INTA signal to next deviceIf needs service, does not pass the INTA, puts its code on the address lines Polled

Multiple Interrupts

•Priority in Processor Status Word

–Status Register --active program

–Status Word --inactive program

•Changed only by privileged instruction

•Mode changes --automatic or by privileged instruction

•Interrupt enable/disable, by device, system-wide

Main Program

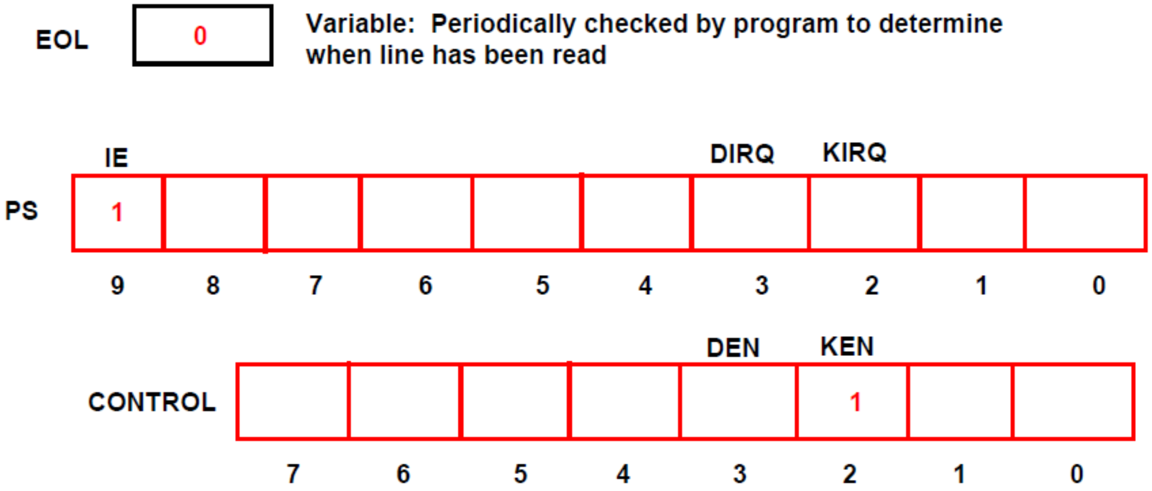
```
        Move          #LINE, PNTR     Initialize buffer pointer
        Clear         EOL             Clear end-of-line indicator
        BitSet        #2, CONTROL     Enable keyboard interrupts
        BitSet        #9, PS          Set interrupt-enable bit in the PS
         … continue to process
```

Interrupt Service Routine

```
READ    MoveMultiple  R0-R1,  -(SP)   Push registers R0,R1 onto stack
        Move          PNTR, R0        Load memory address pointer
        MoveByte      DATAIN, R1      Get input character
        MoveByte      R1, (R0)+       Store it in memory
        Move          R0, PNTR        Update memory pointer
        CompareByte   #$0D, R1        Check if Carriage Return
        Branch NZ     RTRN
        Move          #1, EOL         Indicate end-of-line
        BitClear      #2, CONTROL     Disable keyboard interrupts
RTRN    MoveMultiple  (SP)+, R0-R1    Pop and restore registers
        Return                        Return from interrupt
```

Main Program

```
        Move        #LINE, PNTR     Initialize buffer pointer
        Clear       EOL             Clear end-of-line indicator
        BitSet      #2, CONTROL     Enable keyboard interrupts
        BitSet      #9, PS          Set interrupt-enable bit in the PS
```



EOL 0 **Variable: Periodically checked by program to determine when line has been read**

	IE						DIRQ	KIRQ		
PS	1									
	9	8	7	6	5	4	3	2	1	0

					DEN	KEN		
CONTROL						1		
	7	6	5	4	3	2	1	0

Interrupt Service Routine

```
READ    MoveMultiple    R0-R1,  -(SP)   Push registers R0,R1 onto stack
        Move            PNTR, R0        Load memory address pointer
        MoveByte        DATAIN, R1      Get input character
        MoveByte        R1, (R0)+       Store it in memory
        Move            R0, PNTR        Update memory pointer
        CompareByte     #$0D, R1        Check if Carriage Return
        Branch NZ       RTRN
        Move            #1, EOL         Indicate end-of-line
        BitClear        #2, CONTROL     Disable keyboard interrupts
RTRN    MoveMultiple    (SP)+, R0-R1    Pop and restore registers
        Return                          Return from interrupt
```

Invoked each time the keyboard puts a character in DATAIN and causes an interrupt

Continues until CR character encountered, then keyboard interrupts are disabled until another line is requested

Common Functions of Interrupts

•Interrupt transfers control to the interrupt service routine, generally through the *interrupt vector table*, which contains the addresses of all the service routines.

•Interrupt architecture must save the address of the interrupted instruction and the contents of the processor status register.

•Incoming interrupts are *disabled*while another interrupt is being processed to prevent a *lost interrupt*.

•A software-generated interrupt may be caused either by an error or a user request (sometimes called a *trap*).

•An operating system is *interrupt*driven.

Hardware interrupts—from I/O devices, memory, processor,
Software interrupts—Generatedby a program.

Direct Memory Access (DMA)

•Polling or interrupt driven I/O incurs considerable overhead
–Multiple program instructions
–Saving program state
–Incrementing memory addresses
–Keeping track of word count

•Transfer large amounts of data at high speed without continuous intervention by the processor

•Special control circuit required in the I/O device interface, called a DMA controller

•DMA controller keeps track of memory locations, transfers directly to memory (via the bus) independent of the processor

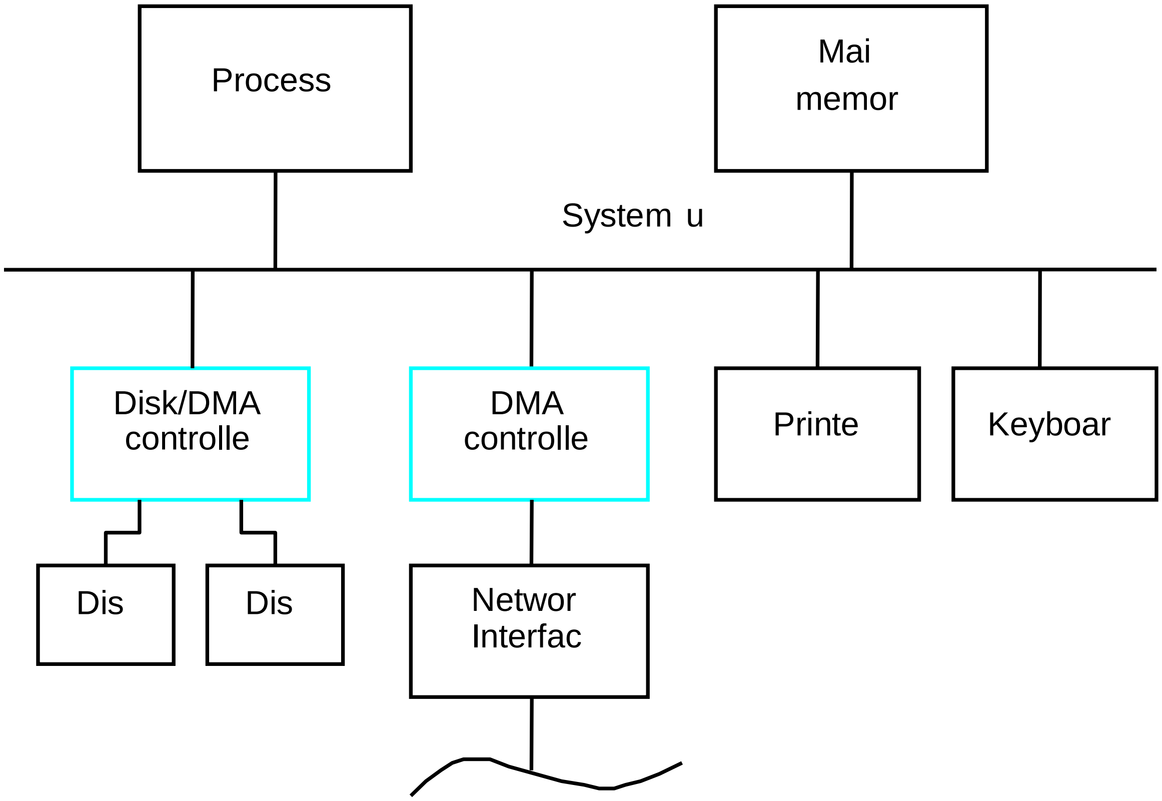


Figure. Use of DMA controllers in a computer system

DMA Controller

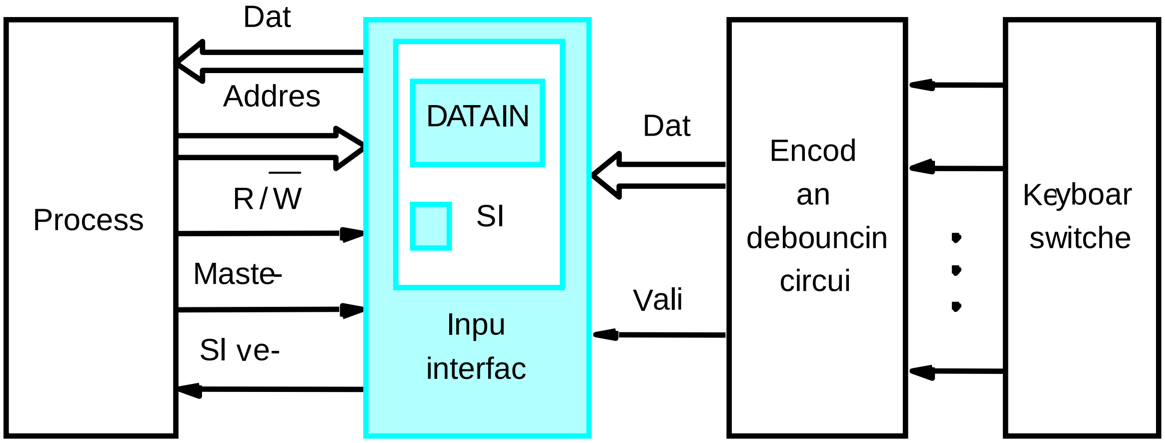
•Part of the I/O device interface
–DMA Channels
•Performs functions that would normally be carried out by the processor
–Provides memory address
–Bus signals that control transfer
–Keeps track of number of transfers
•Under control of the processor

INTERFACE CIRCUITS

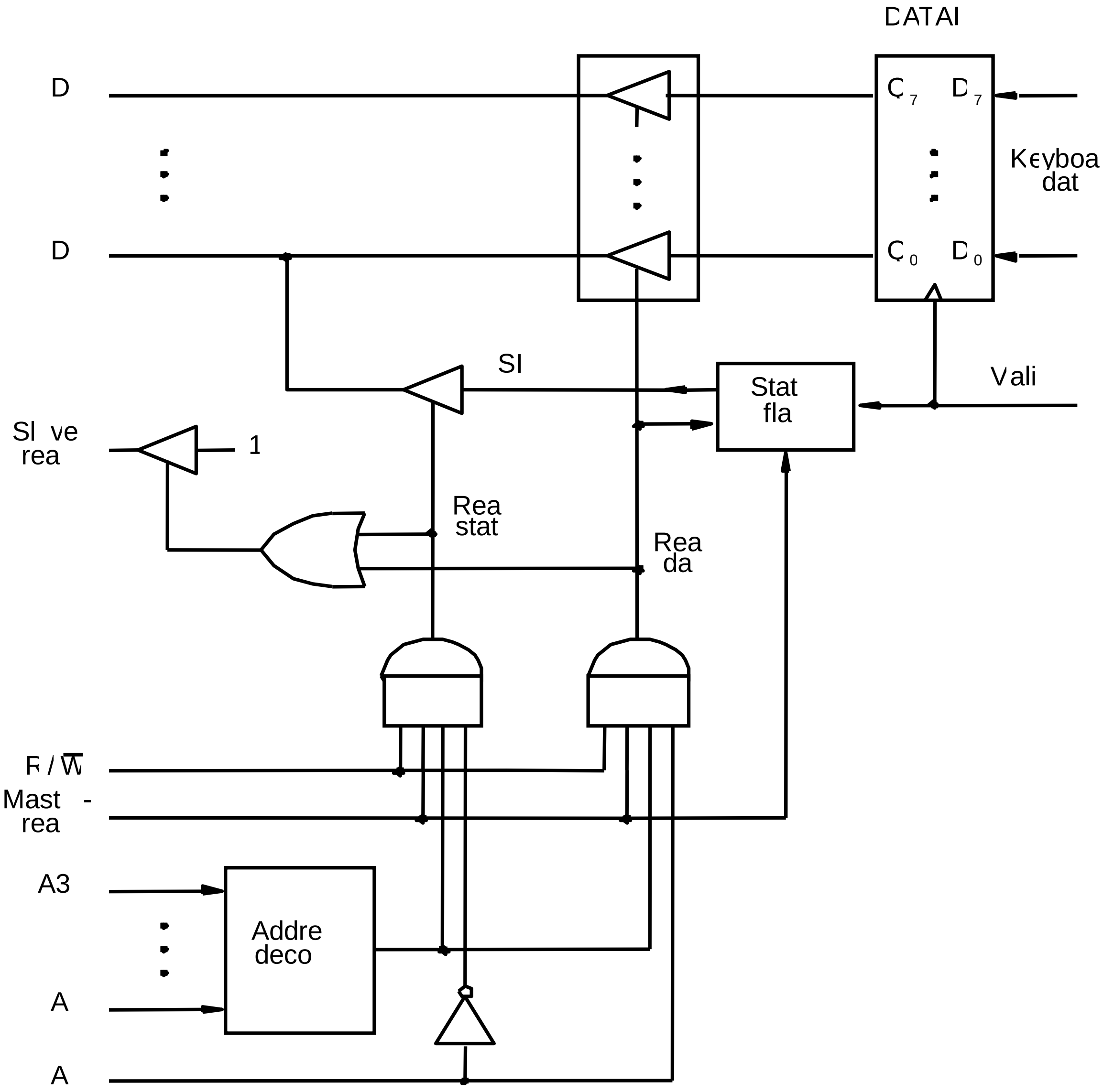
- Circuitry required to connect an I/O device to a computer bus
- Provides a storage buffer for at least one word of data.
- Contains status flag that can be accessed by the processor.
- Contains address-decoding circuitry
- Generates the appropriate timing signals required by the bus control scheme.
- Performs format conversions

- Ports
  - Serial port
  - Parallel port

Figure. Keyboard to processor connection



INPUT INTERFACE CIRCUIT



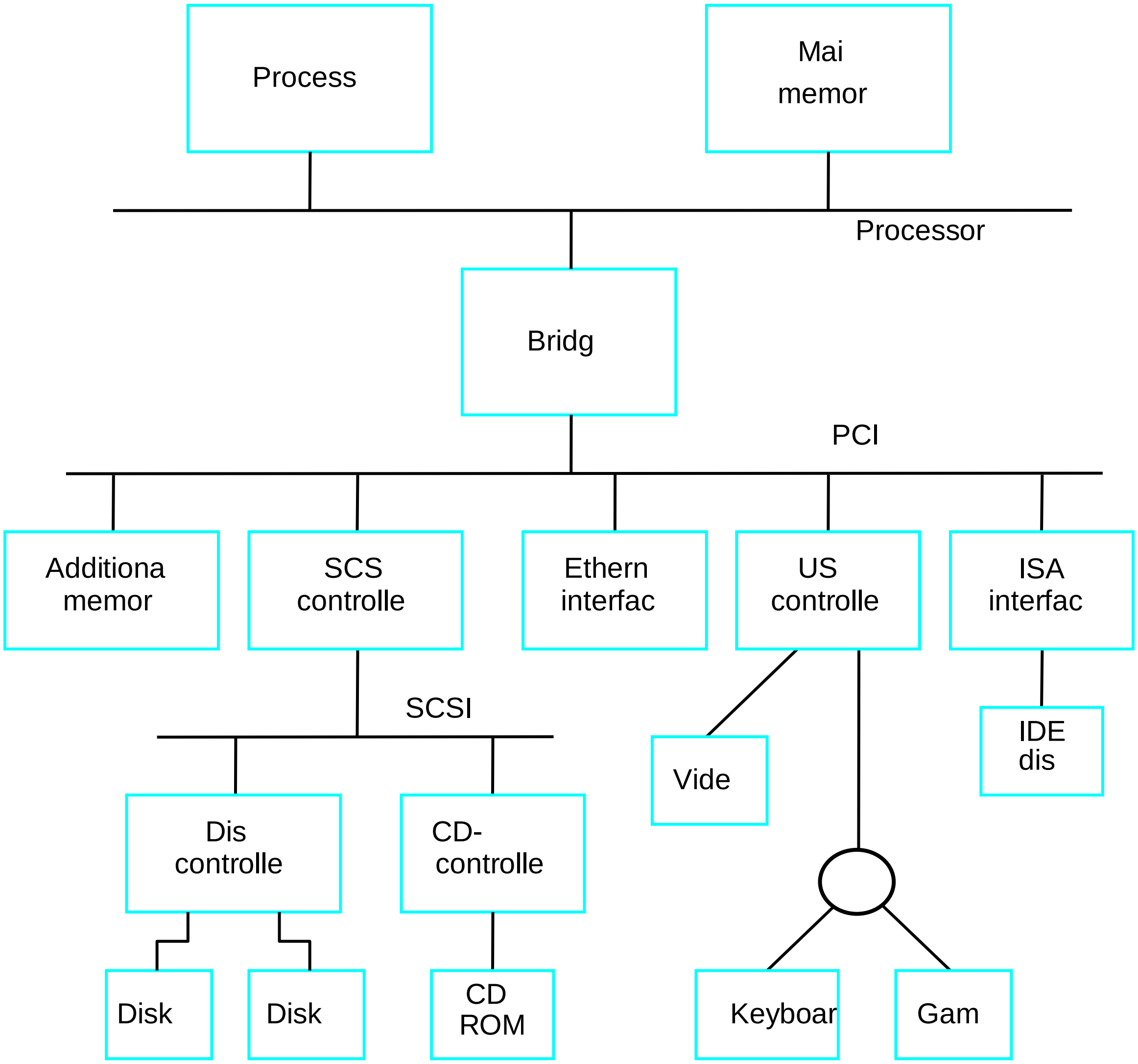


Figure . An example of a computer system using different interface standards.

PCI (Peripheral Component Interconnect)

- PCI stands for *Peripheral Component Interconnect*
- Introduced in 1992
- It is a Low-cost bus
- It is Processor independent
- It has Plug-and-play capability

PCI bus transactions

PCI bus traffic is made of a series of PCI bus transactions. Each transaction is made up of an *address phase* followed by one or more *data phases*. The direction of the data phases may be from initator to target (write transaction) or vice-versa (read transaction), but all of the data phases must be in the same direction. Either party may pause or halt the data phases at any point. (One common example is a low-performance PCI device that does not support burst transactions, and always halts a transaction after the first data phase.)

Any PCI device may initiate a transaction. First, it must request permission from a PCI bus arbiter on the motherboard. The arbiter grant permission to one of the requesting devices. The initiator begins the address phase by broadcasting a 32-bit address plus a 4-bit command code, then waits for a target to respond. All other devices examine this address and one of them responds a few cycles later.

64-bit addressing is done using a 2-stage address phase. The initiator broadcasts the low 32 address bits, accompanied by a special "dual address cycle" command code. Devices which do not support 64-bit addressing can simply not respond to that command code. The next cycle, the initiator transmits the high 32 address bits, plus the real command code. The transaction operates identically from that point on. To ensure compatibility with 32-bit PCI devices, it is forbidden to use a dual address cycle if not necessary, i.e. if the high-order address bits are all zero.

While the PCI bus transfers 32 bits per data phase, the initiator transmits a 4-bit byte mask indicating which 8-bit bytes are to be considered significant. In particular, a masked write must affect only the desired bytes in the target PCI device.

Arbitration

Any device on a PCI bus that is capable of acting as a bus master may initiate a transaction with any other device. To ensure that only one transaction is initiated at a time, each master must first wait for a bus grant signal, GNT#, from an arbiter located on the motherboard. Each device has a separate request line REQ# that requests the bus, but the arbiter may "park" the bus grant signal at any device if there are no current requests.

The arbiter may remove GNT# at any time. A device which loses GNT# may complete its current transaction, but may not start one (by asserting FRAME#) unless it observes GNT# asserted the cycle before it begins.

The arbiter may also provide GNT# at any time, including during another master's transaction. During a transaction, either FRAME# or IRDY# or both are asserted; when both are deasserted, the bus is idle. A device may initiate a transaction at any time that GNT# is asserted and the bus is idle.

Address phase

A PCI bus transaction begins with an *address phase*. The initiator, seeing that it has GNT# and the bus is idle, drives the target address onto the AD[31:0] lines, the associated command (e.g. memory read, or I/O write) on the C/BE[3:0]# lines, and pulls FRAME# low.

Each other device examines the address and command and decides whether to respond as the target by asserting DEVSEL#. A device must respond by asserting DEVSEL# within 3 cycles. Devices which promise to respond within 1 or 2 cycles are said to have "fast DEVSEL" or "medium DEVSEL", respectively. (Actually, the time to respond is 2.5 cycles, since PCI devices must transmit all signals half a cycle early so that they can be received three cycles later.)

Note that a device must latch the address on the first cycle; the initiator is required to remove the address and command from the bus on the following cycle, even before receiving a DEVSEL# response. The additional time is available only for interpreting the address and command after it is captured.

On the fifth cycle of the address phase (or earlier if all other devices have medium DEVSEL or faster), a catch-all "subtractive decoding" is allowed for some address ranges. This is commonly used by an ISA bus bridge for addresses within its range (24 bits for memory and 16 bits for I/O).

On the sixth cycle, if there has been no response, the initiator may abort the transaction by deasserting FRAME#. This is known as *master abort termination* and it is customary for PCI bus bridges to return all-ones data (0xFFFFFFFF) in this case. PCI devices therefore are generally designed to avoid using the all-ones value in important status registers, so that such an error can be easily detected by software.

Address phase timing

```
             _  0_  1_  2_  3_  4_  5_
       CLK _/ \_/ \_/ \_/ \_/ \_/ \_/
           ___
      GNT#    \___/XXXXXXXXXXXXXXXXXXX (GNT# Irrelevant after cycle has started)
           _______
    FRAME#        \___________________
                   ___
  AD[31:0] -------<___>--------------- (Address only valid for 1 cycle.)
                   ___ _______________
C/BE[3:0]# -------<___X_______________ (Command, then first data phase byte enables)
           _______________________
   DEVSEL#            \___\___\___\___
                    Fast Med Slow Subtractive
             _   _   _   _   _   _   _
       CLK _/ \_/ \_/ \_/ \_/ \_/ \_/
                0   1   2   3   4   5
```

On the rising edge of clock 0, the initiator observes FRAME# and IRDY# both high, and GNT# low, so it drives the address, command, and asserts FRAME# in time for the rising edge of clock 1. Targets latch the address and begin decoding it. They may respond with DEVSEL# in time for clock 2 (fast DEVSEL), 3 (medium) or 4 (slow). Subtractive decode devices, seeing no other response by clock 4, may respond on clock 5. If the master does not see a response by clock 5, it will terminate the transaction and remove FRAME# on clock 6.

TRDY# and STOP# are deasserted (high) during the address phase. The initiator may assert IRDY# as soon as it is ready to transfer data, which could theoretically be as soon as clock 2.

Data phases

After the address phase (specifically, beginning with the cycle that DEVSEL# goes low) comes a burst of one or more *data phases*. In all cases, the initiator drives active-low byte select signals on the C/BE[3:0]# lines, but the data on the AD[31:0] may be driven by the initiator (on case of writes) or target (in case of reads).

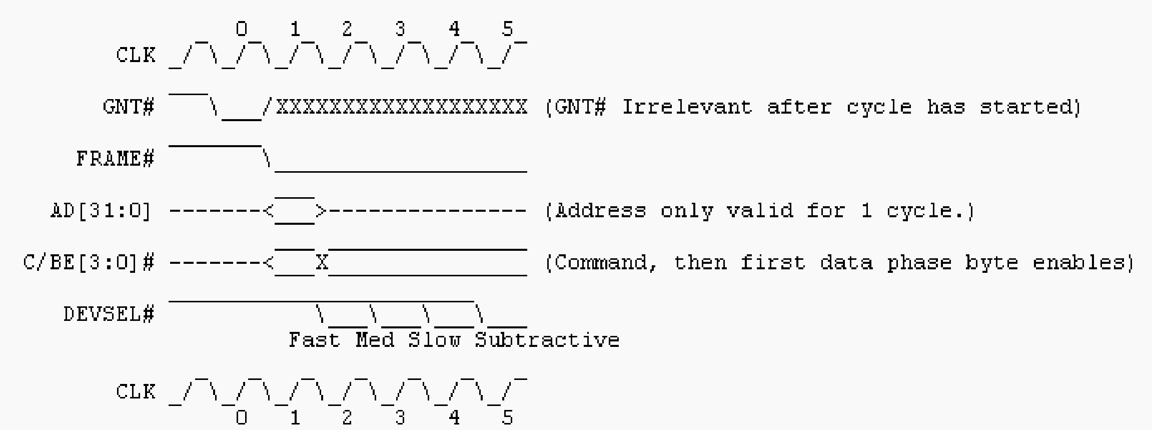
During data phases, the C/BE[3:0]# lines are interpreted as active-low *byte enables*. In case of a write, the asserted signals indicate which of the four bytes on the AD bus are to be written to the addressed location. In the case of a read, they indicate which bytes the initiator is interested in. For reads, it is always legal to ignore the byte enable signals and simply return all 32 bits; cacheable memory resources are required to always return 32 valid bits. The byte enables are mainly useful for I/O space accesses where reads have side effects.

A data phase with all four C/BE# lines deasserted is explicitly permitted by the PCI standard, and must have no effect on the target (other than to advance the address in the burst access in progress).

The data phase continues until both parties are ready to complete the transfer and continue to the next data phase. The initiator asserts IRDY# (*initiator ready*) when it no longer needs to wait, while the target asserts TRDY# (*target ready*). Whichever side is providing the data must drive it on the AD bus before asserting its ready signal.

Once one of the participants asserts its ready signal, it may not become un-ready or otherwise alter its control signals until the end of the data phase. The data recipient must latch the AD bus each cycle until it sees both IRDY# and TRDY# asserted, which marks the end of the current data phase and indicates that the just-latched data is the word to be transferred.

To maintain full burst speed, the data sender then has half a clock cycle after seeing both IRDY# and TRDY# asserted to drive the next word onto the AD bus.



```
             _  0_  1_  2_  3_  4_  5_
       CLK _/ \_/ \_/ \_/ \_/ \_/ \_/
           ___
      GNT#    \___/XXXXXXXXXXXXXXXXXXX (GNT# Irrelevant after cycle has started)
           _______
    FRAME#        \___________________
                   ___
  AD[31:0] -------<___>--------------- (Address only valid for 1 cycle.)
                   ___ _______________
C/BE[3:0]# -------<___X_______________ (Command, then first data phase byte enables)
           _______________________
   DEVSEL#            \___\___\___\___
                    Fast Med Slow Subtractive
             _   _   _   _   _   _   _
       CLK _/ \_/ \_/ \_/ \_/ \_/ \_/
                0   1   2   3   4   5
```

This continues the address cycle illustrated above, assuming a single address cycle with medium DEVSEL, so the target responds in time for clock 3. However, at that time, neither side is ready to transfer data. For clock 4, the initiator is ready, but the target is not. On clock 5, both are ready, and a data transfer takes place (as indicated by the vertical lines). For clock 6, the target is ready to transfer, but the initator is not. On clock 7, the initiator becomes ready, and data is transferred. For clocks 8 and 9, both sides remain ready to transfer data, and data is transferred at the maximum possible rate (32 bits per clock cycle).

In case of a read, clock 2 is reserved for turning around the AD bus, so the target is not permitted to drive data on the bus even if it is capable of fast DEVSEL.

Fast DEVSEL# on reads

A target that supports fast DEVSEL could in theory begin responding to a read the cycle after the address is presented. This cycle is, however, reserved for AD bus turnaround. Thus, a target may not drive the AD bus (and thus may not assert TRDY#) on the second cycle of a transaction. Note that most targets will not be this fast and will not need any special logic to enforce this condition.

Ending transactions

Either side may request that a burst end after the current data phase. Simple PCI devices that do not support multi-word bursts will always request this immediately. Even devices that do support bursts will have some limit on the maximum length they can support, such as the end of their addressable memory.

The initiator can mark any data phase as the final one in a transaction by deasserting FRAME# at the same time as it asserts IRDY#. The cycle after the target asserts TRDY#, the final data transfer is complete, both sides deassert their respective RDY# signals, and the bus is idle again. The master may not deassert FRAME# before asserting IRDY#, nor may it assert FRAME# while waiting, with IRDY# asserted, for the target to assert TRDY#.

The only minor exception is a *master abort termination*, when no target responds with DEVSEL#. Obviously, it is pointless to wait for TRDY# in such a case. However, even in this case, the master must assert IRDY# for at least one cycle after deasserting FRAME#. (Commonly, a master will assert IRDY# before receiving DEVSEL#, so it must simply hold IRDY# asserted for one cycle longer.) This is to ensure that bus turnaround timing rules are obeyed on the FRAME# line.

The target requests the initiator end a burst by asserting STOP#. The initiator will then end the transaction by deasserting FRAME# at the next legal opportunity. If it wishes to transfer more data, it will continue in a separate transaction. There are several ways to do this:

Disconnect with data

If the target asserts STOP# and TRDY# at the same time, this indicates that the target wishes this to be the last data phase. For example, a target that does not support burst transfers will always do this to force single-word PCI transactions. This is the most efficient way for a target to end a burst.

Disconnect without data

If the target asserts STOP# without asserting TRDY#, this indicates that the target wishes to stop without transferring data. STOP# is considered equivalent to TRDY# for the purpose of ending a data phase, but no data is transferred.

Retry

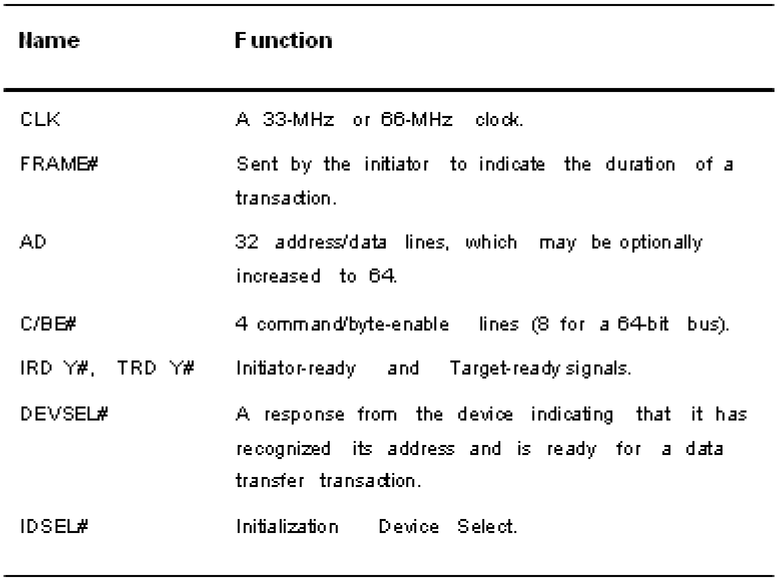
A Disconnect without data before transferring any data is a *retry*, and unlike other PCI transactions, PCI initiators are required to pause slightly before continuing the operation. See the PCI specification for details.

Target abort

Normally, a target holds DEVSEL# asserted through the last data phase. However, if a target deasserts DEVSEL# before disconnecting without data (asserting STOP#), this indiates a *target abort*, which is a fatal error condition. The initiator may not retry, and typically treats it as a bus error. Note that a target may not deassert DEVSEL# while waiting with TRDY# or STOP# low; it must do this at the beginning of a data phase.

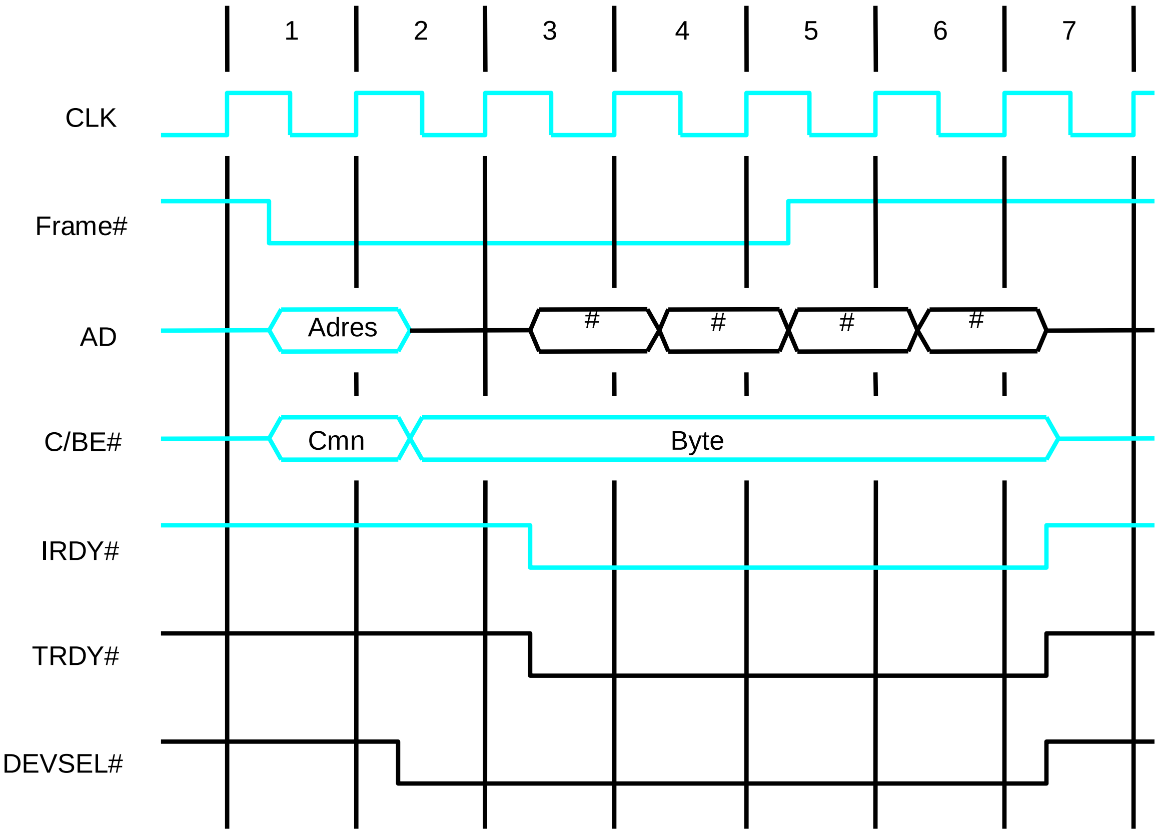
After seeing STOP#, the initiator will terminate the transaction at the next legal opportunity, but if it has already signaled its desire to continue a burst (by asserting IRDY# without deasserting FRAME#), it is not permitted to deassert FRAME# until the following data phase. A target that requests a burst end (by asserting STOP#) may have to wait through another data phase (holding STOP# asserted without TRDY#) before the transaction can end.

Table 4.3. Data transfer signals on the PCI bus.

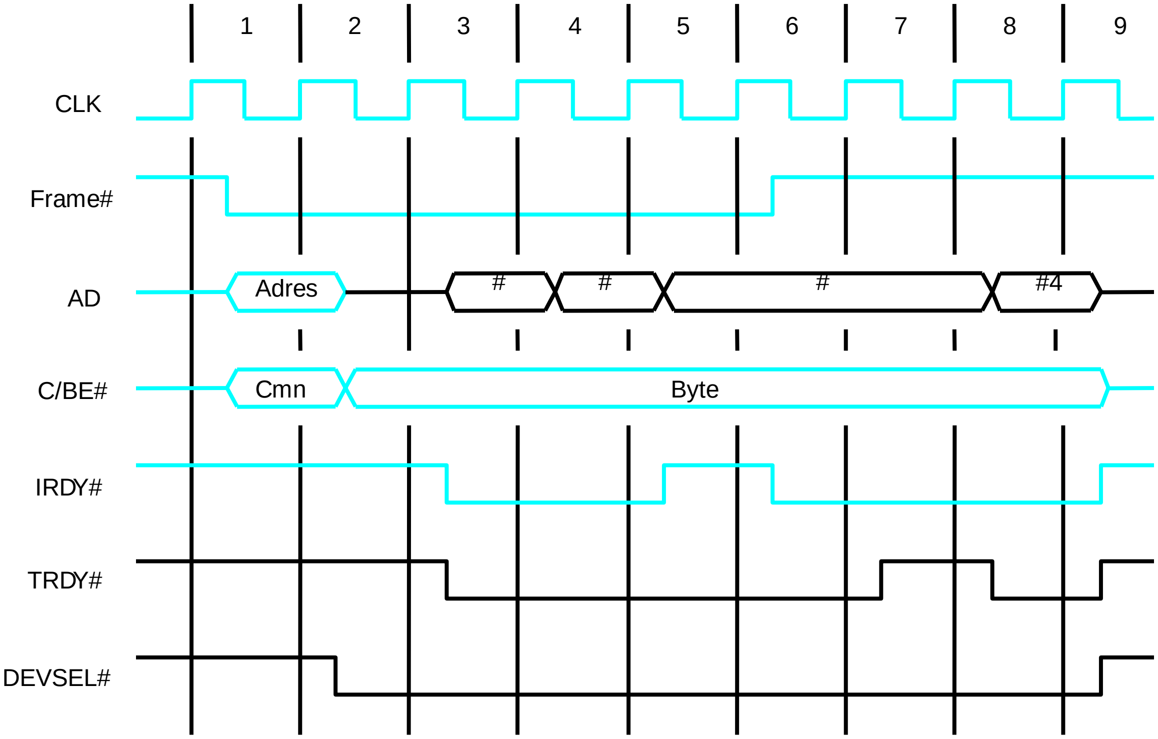


Name	Function
CLK	A 33-MHz or 66-MHz clock.
FRAME#	Sent by the initiator to indicate the duration of a transaction.
AD	32 address/data lines, which may be optionally increased to 64.
C/BE#	4 command/byte-enable lines (8 for a 64-bit bus).
IRDY#, TRDY#	Initiator-ready and Target-ready signals.
DEVSEL#	A response from the device indicating that it has recognized its address and is ready for a data transfer transaction.
IDSEL#	Initialization Device Select.

Read operation on the PCI Bus



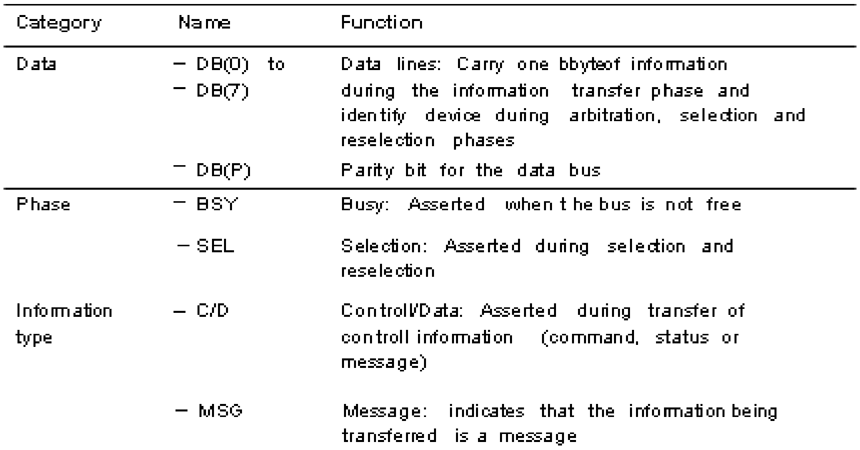
Read operation showing the role of the IRDY#, TRDY#



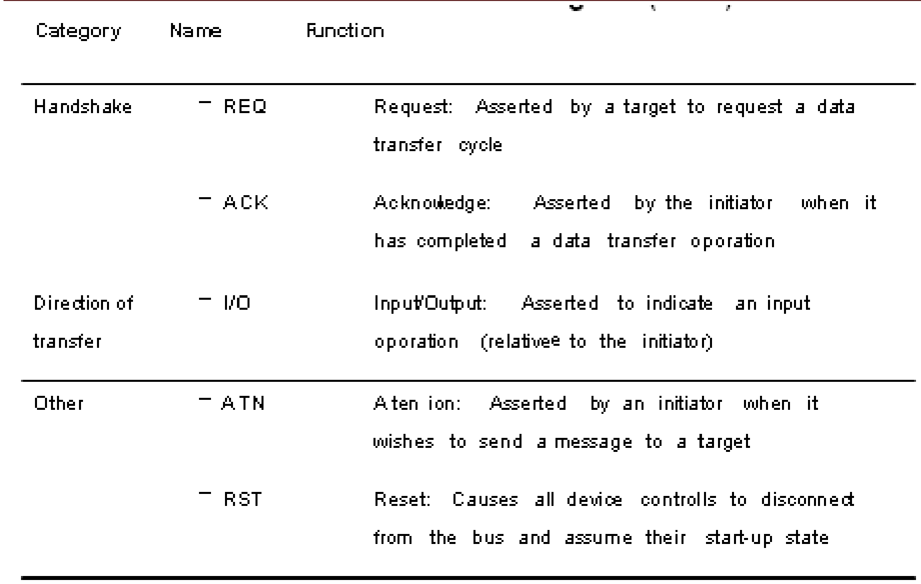
SCSI Bus

- Defined by ANSI – X3.131
- *Small Computer System Interface*
- 50, 68 or 80 pins
- Max. transfer rate – 160 MB/s, 320 MB/s.

SCSI Bus Signals



Category	Name	Function
Data	– DB(0) to – DB(7)	Data lines: Carry one bbyteof infomation during the infomation transfer phase and identify device during arbitration, selection and reselection phases
	– DB(P)	Parity bit for the data bus
Phase	– BSY	Busy: Asserted when t he bus is not free
	– SEL	Selection: Asserted during selection and reselection
Infomation type	– C/D	Controll/Data: Asserted during transfer of controll infomation (command, status or message)
	– MSG	Message: indicates that the infomation being transferred is a message

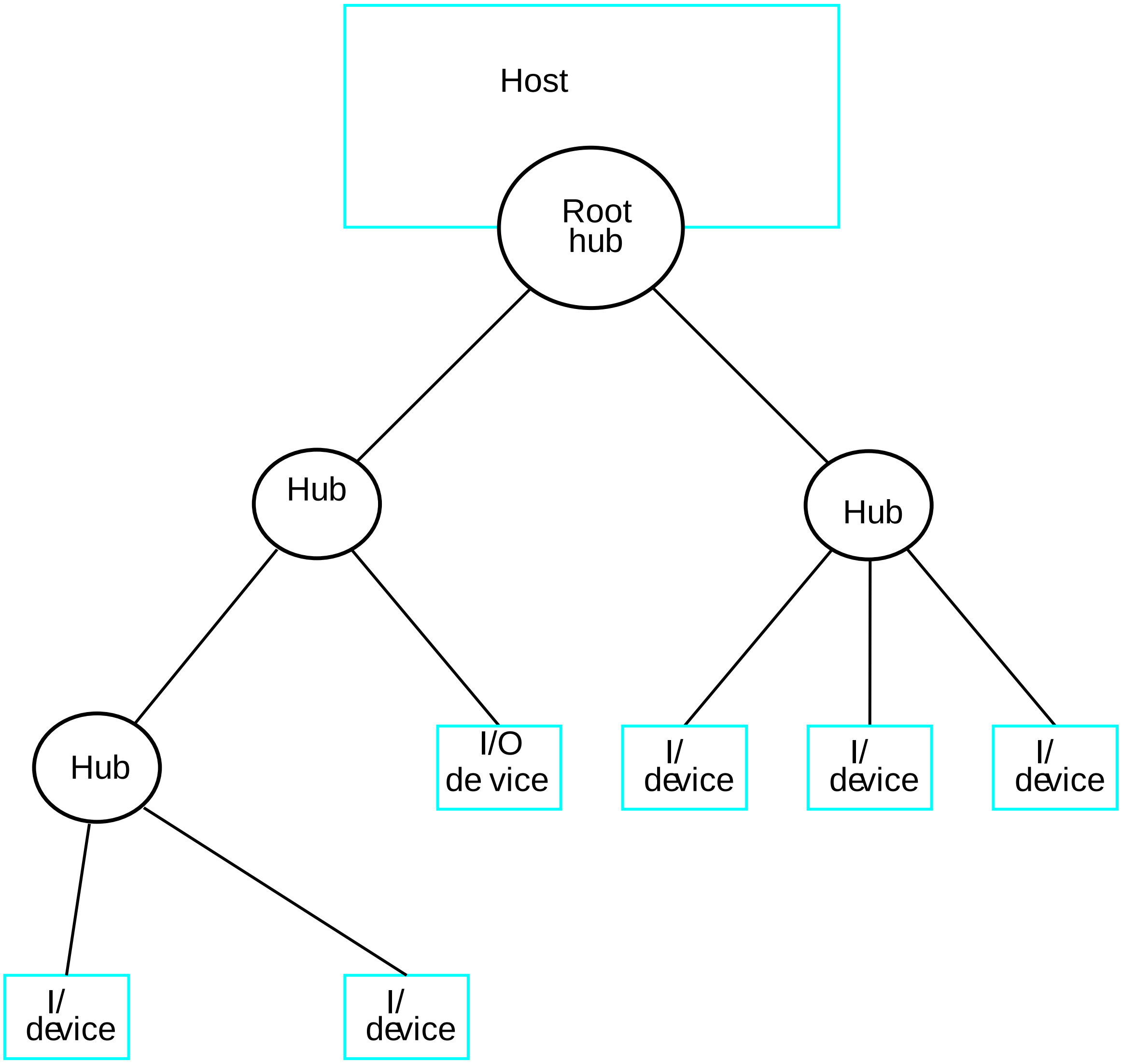


Category	Name	Function
Handshake	− REQ	Request: Asserted by a target to request a data transfer cycle
	− ACK	Acknowledge: Asserted by the initiator when it has completed a data transfer oporation
Direction of transfer	− I/O	Input/Output: Asserted to indicate an input oporation (relativee to the initiator)
Other	− ATN	Aten ion: Asserted by an initiator when it wishes to send a message to a target
	− RST	Reset: Causes all device controlls to disconnect from the bus and assume their start-up state

USB

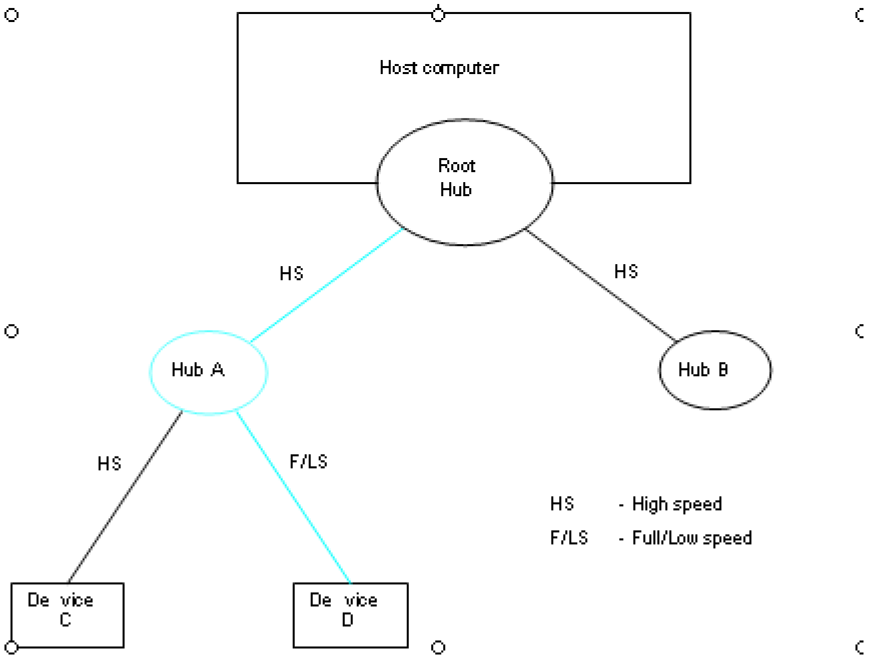
- *Universal Serial Bus*
- Speed
  - Low-speed(1.5 Mb/s)
  - Full-speed(12 Mb/s)
  - High-speed(480 Mb/s)
- Port Limitation
- Device Characteristics
- Plug-and-play

Universal Serial Bus Tree Structure



USB (**Universal Serial Bus**) is a specification to establish communication between devices and a host controller (usually personal computers). USB is intended to replace many varieties of serial and parallel ports. USB can connect computer peripherals such as mice, keyboards, digital cameras, printers, personal media players, flash drives, and external hard drives. For many of those devices, USB has become the standard connection method. USB was designed for personal computers[*citation needed*], but it has become commonplace on other devices such as smartphones, PDAs and video game consoles, and as a power cord between a device and an AC adapter plugged into a wall plug for charging. As of 2008, there are about 2 billion USB devices sold per year, and approximately 6 billion total sold to date.

Split Bus Operation



Signaling

USB supports following signaling rates:

- A **low speed** rate of 1.5 Mbit/s is defined by USB 1.0. It is very similar to "full speed" operation except each bit takes 8 times as long to transmit. It is intended primarily to save cost in low-bandwidth human interface devices (HID) such as keyboards, mice, and joysticks.
- The **full speed** rate of 12 Mbit/s is the basic USB data rate defined by USB 1.1. All USB hubs support full speed.
- A **hi-speed** (USB 2.0) rate of 480 Mbit/s was introduced in 2001. All hi-speed devices are capable of falling back to full-speed operation if necessary; they are backward compatible. Connectors are identical.
- A **SuperSpeed** (USB 3.0) rate of 5.0 Gbit/s. The USB 3.0 specification was released by Intel and partners in August 2008, according to early reports from CNET news. The first USB 3 controller chips were sampled by NEC May 2009 [11] and products using the 3.0 specification are expected to arrive beginning in Q3 2009 and 2010.[12] USB 3.0 connectors are generally backwards compatible, but include new wiring and full duplex operation. There is some incompatibility with older connectors.

USB signals are transmitted on a braided pair data cable with 90Ω ±15% Characteristic impedance,[13] labeled D+ and D−. Prior to USB 3.0, These collectively use half-duplex differential signaling to reduce the effects of electromagnetic noise on longer lines. Transmitted signal levels are 0.0–0.3 volts for low and 2.8–3.6 volts for high in full speed (FS) and low speed (LS) modes, and −10–10 mV for low and 360–440 mV for high in hi-speed (HS) mode. In FS mode the cable wires are not terminated, but the HS mode has termination of 45 Ω to ground, or 90 Ω differential to match the data cable impedance, reducing interference of particular kinds. USB 3.0 introduces two additional pairs of shielded twisted wire and new, mostly interoperable contacts in USB 3.0 cables, for them. They permit the higher data rate, and full duplex operation.

A USB connection is always between a host or hub at the "A" connector end, and a device or hub's "upstream" port at the other end. Originally, this was a "B' connector, preventing erroneous loop connections, but additional upstream connectors were specified, and some cable vendors designed and sold cables which permitted erroneous connections (and potential damage to the circuitry). USB interconnections are not as fool-proof or as simple as originally intended.

The host includes 15 kΩ pull-down resistors on each data line. When no device is connected, this pulls both data lines low into the so-called "single-ended zero" state (SE0 in the USB documentation), and indicates a reset or disconnected connection.

A USB device pulls one of the data lines high with a 1.5 kΩ resistor. This overpowers one of the pull-down resistors in the host and leaves the data lines in an idle state called "J". For USB 1.x, the choice of data line indicates a device's speed support; full-speed devices pull D+ high, while low-speed devices pull D− high.

USB data is transmitted by toggling the data lines between the J state and the opposite K state. USB encodes data using the NRZI convention; a 0 bit is transmitted by toggling the data lines from J to K or vice-versa, while a 1 bit is transmitted by leaving the data lines as-is. To ensure a minimum density of signal transitions, USB uses bit stuffing; an extra 0 bit is inserted into the data stream after any appearance of six consecutive 1 bits. Seven consecutive 1 bits is always an error. USB 3.00 has introduced additional data transmission encodings.

A USB packet begins with an 8-bit synchronization sequence 00000001. That is, after the initial idle state J, the data lines toggle KJKJKJKK. The final 1 bit (repeated K state) marks the end of the sync pattern and the beginning of the USB frame.

A USB packet's end, called EOP (end-of-packet), is indicated by the transmitter driving 2 bit times of SE0 (D+ and D− both below max) and 1 bit time of J state. After this, the transmitter ceases to drive the D+/D− lines and the aforementioned pull up resistors hold it in the J (idle) state. Sometimes skew due to hubs can add as much as one bit time before the SE0 of the end of packet. This extra bit can also result in a "bit stuff violation" if the six bits before it in the CRC are '1's. This bit should be ignored by receiver.

A USB bus is reset using a prolonged (10 to 20 milliseconds) SE0 signal.

USB 2.0 devices use a special protocol during reset, called "chirping", to negotiate the high speed mode with the host/hub. A device that is HS capable first connects as an FS device (D+ pulled high), but upon receiving a USB RESET (both D+ and D− driven LOW by host for 10 to 20 mS) it pulls the D− line high, known as chirp K. This indicates to the host that the device is high speed. If the host/hub is also HS capable, it chirps (returns alternating J and K states on D− and D+ lines) letting the device know that the hub will operate at high speed. The device has to receive at least 3 sets of KJ chirps before it changes to high speed terminations and begins high speed signaling. Because USB 3.0 use wiring separate and additional to that used by USB 2.0 and USB 1.x, such speed negotiation is not required.

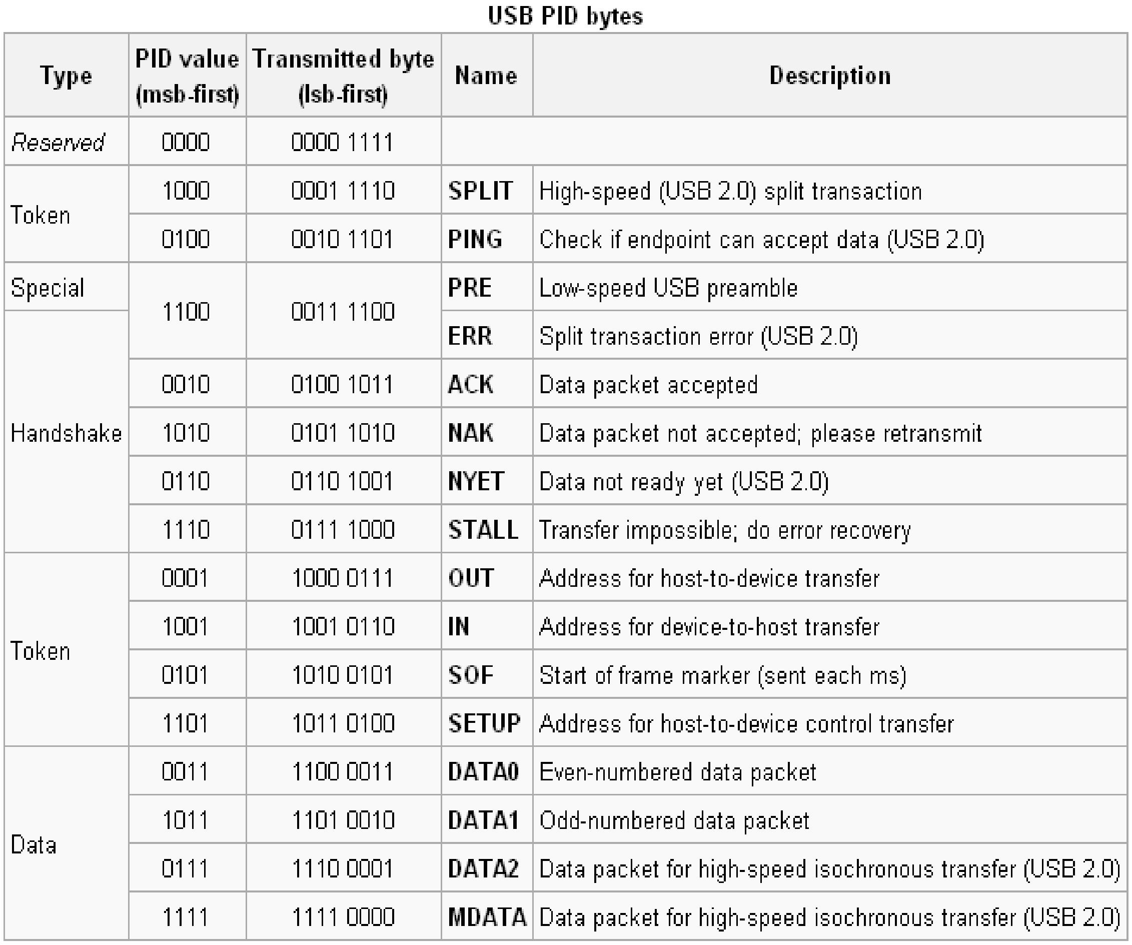
Clock tolerance is 480.00 Mbit/s ±500 ppm, 12.000 Mbit/s ±2500 ppm, 1.50 Mbit/s ±15000 ppm.

Though high speed devices are commonly referred to as "USB 2.0" and advertised as "up to 480 Mbit/s", not all USB 2.0 devices are high speed. The USB-IF certifies devices and provides licenses to use special marketing logos for either "basic speed" (low and full) or high speed after passing a compliance test and paying a licensing fee. All devices are tested according to the latest specification, so recently-compliant low speed devices are also 2.0 devices.

Data packets

USB communication takes the form of packets. Initially, all packets are sent from the host, via the root hub and possibly more hubs, to devices. Some of those packets direct a device to send some packets in reply.

After the sync field described above, all packets are made of 8-bit bytes, transmitted least-significant bit first. The first byte is a packet identifier (PID) byte. The PID is actually 4 bits; the byte consists of the 4-bit PID followed by its bitwise complement. This redundancy helps detect errors. (Note also that a PID byte contains at most four consecutive 1 bits, and thus will never need bit-stuffing, even when combined with the final 1 bit in the sync byte. However, trailing 1 bits in the PID may require bit-stuffing within the first few bits of the payload.)



USB PID bytes

Type	PID value (msb-first)	Transmitted byte (lsb-first)	Name	Description
Reserved	0000	0000 1111		
Token	1000	0001 1110	**SPLIT**	High-speed (USB 2.0) split transaction
	0100	0010 1101	**PING**	Check if endpoint can accept data (USB 2.0)
Special	1100	0011 1100	**PRE**	Low-speed USB preamble
Handshake			**ERR**	Split transaction error (USB 2.0)
	0010	0100 1011	**ACK**	Data packet accepted
	1010	0101 1010	**NAK**	Data packet not accepted; please retransmit
	0110	0110 1001	**NYET**	Data not ready yet (USB 2.0)
	1110	0111 1000	**STALL**	Transfer impossible; do error recovery
Token	0001	1000 0111	**OUT**	Address for host-to-device transfer
	1001	1001 0110	**IN**	Address for device-to-host transfer
	0101	1010 0101	**SOF**	Start of frame marker (sent each ms)
	1101	1011 0100	**SETUP**	Address for host-to-device control transfer
Data	0011	1100 0011	**DATA0**	Even-numbered data packet
	1011	1101 0010	**DATA1**	Odd-numbered data packet
	0111	1110 0001	**DATA2**	Data packet for high-speed isochronous transfer (USB 2.0)
	1111	1111 0000	**MDATA**	Data packet for high-speed isochronous transfer (USB 2.0)

Handshake packets

Handshake packets consist of nothing but a PID byte, and are generally sent in response to data packets. The three basic types are **ACK**, indicating that data was successfully received, **NAK**, indicating that the data cannot be received at this time and should be retried, and **STALL**, indicating that the device has an error and will never be able to successfully transfer data until some corrective action (such as device initialization) is performed.

USB 2.0 added two additional handshake packets, **NYET** which indicates that a split transaction is not yet complete. A NYET packet is also used to tell the host that the receiver has accepted a data packet, but cannot accept any more due to buffers being full. The host will then send PING packets and will continue with data packets once the device ACK's the PING. The other packet added was the **ERR** handshake to indicate that a split transaction failed.

The only handshake packet the USB host may generate is ACK; if it is not ready to receive data, it should not instruct a device to send any.

Token packets

Token packets consist of a PID byte followed by 2 payload bytes: 11 bits of address and a 5-bit CRC. Tokens are only sent by the host, never a device.

IN and **OUT** tokens contain a 7-bit device number and 4-bit function number (for multifunction devices) and command the device to transmit DATAx packets, or receive the following DATAx packets, respectively.

An IN token expects a response from a device. The response may be a NAK or STALL response, or a DATAx frame. In the latter case, the host issues an ACK handshake if appropriate.

An OUT token is followed immediately by a DATAx frame. The device responds with ACK, NAK, NYET, or STALL, as appropriate.

SETUP operates much like an OUT token, but is used for initial device setup. It is followed by an 8-byte DATA0 frame with a standardized format.

Every millisecond (12000 full-speed bit times), the USB host transmits a special **SOF** (start of frame) token, containing an 11-bit incrementing frame number in place of a device address. This is used to synchronize isochronous data flows. High-speed USB 2.0 devices receive 7 additional duplicate SOF tokens per frame, each introducing a 125 µs "microframe" (60000 high-speed bit times each).

USB 2.0 added a **PING** token, which asks a device if it is ready to receive an OUT/DATA packet pair. The device responds with ACK, NAK, or STALL, as appropriate. This avoids the need to send the DATA packet if the device knows that it will just respond with NAK.

USB 2.0 also added a larger 3-byte **SPLIT** token with a 7-bit hub number, 12 bits of control flags, and a 5-bit CRC. This is used to perform split transactions. Rather than tie up the high-speed USB bus sending data to a slower USB device, the nearest high-speed capable hub receives a SPLIT token followed by one or two USB packets at high speed, performs the data transfer at full or low speed, and provides the response at high speed when prompted by a second SPLIT token. The details are complex; see the USB specification.

Data packets

A data packet consists of the PID followed by 0–1023 bytes of data payload (up to 1024 in high speed, at most 8 at low speed), and a 16-bit CRC.

There are two basic data packets, **DATA0** and **DATA1**. They must always be preceded by an address token, and are usually followed by a handshake token from the receiver back to the transmitter. The two packet types provide the 1-bit sequence number required by Stop-and-wait ARQ. If a USB host does not receive a response (such as an ACK) for data it has transmitted, it does not know if the data was received or not; the data might have been lost in transit, or it might have been received but the handshake response was lost.

To solve this problem, the device keeps track of the type of DATAx packet it last accepted. If it receives another DATAx packet of the same type, it is acknowledged but ignored as a duplicate. Only a DATAx packet of the opposite type is actually received.

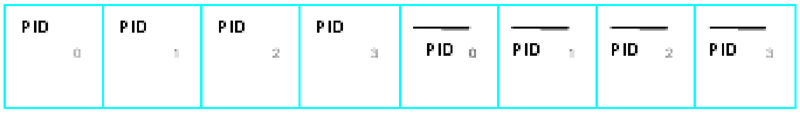
When a device is reset with a SETUP packet, it expects an 8-byte DATA0 packet next.

USB 2.0 added **DATA2** and **MDATA** packet types as well. They are used only by high-speed devices doing high-bandwidth isochronous transfers which need to transfer more than 1024 bytes per 125 µs "microframe" (8192 kB/s).

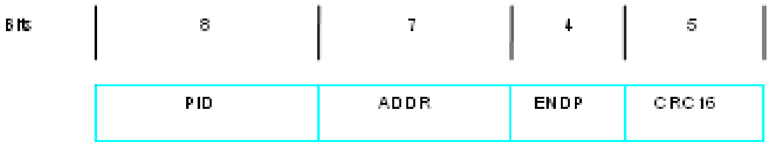
PRE "packet"

Low-speed devices are supported with a special PID value, **PRE**. This marks the beginning of a low-speed packet, and is used by hubs which normally do not send full-speed packets to low-speed devices. Since all PID bytes include four 0 bits, they leave the bus in the full-speed K state, which is the same as the low-speed J state. It is followed by a brief pause during which hubs enable their low-speed outputs, already idling in the J state, then a low-speed packet follows, beginning with a sync sequence and PID byte, and ending with a brief period of SE0. Full-speed devices other than hubs can simply ignore the PRE packet and its low-speed contents, until the final SE0 indicates that a new packet follows.

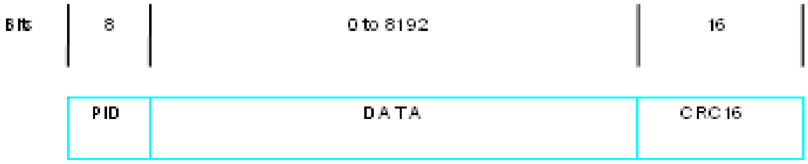
USB Packet Format



(a) Packet identifier field

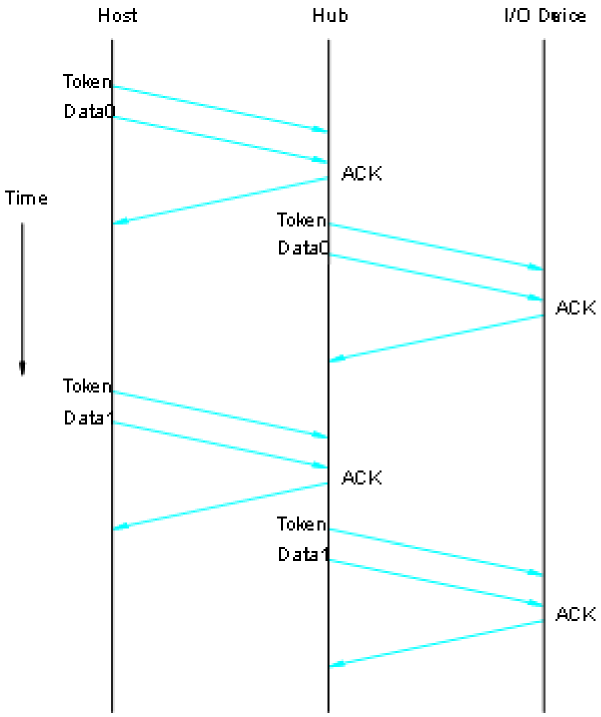


(b) Token packet, IN or OUT

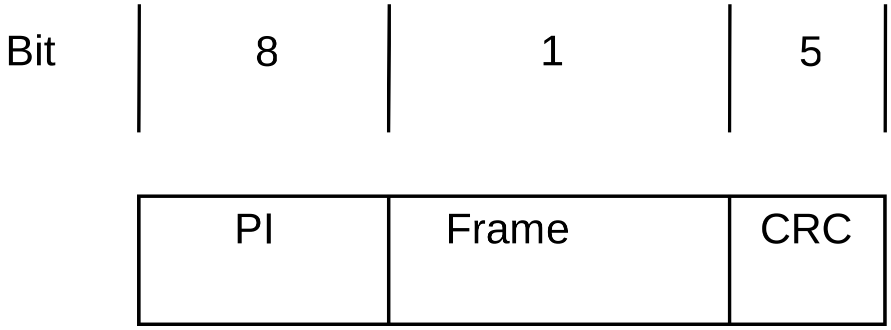


(c) Data packet

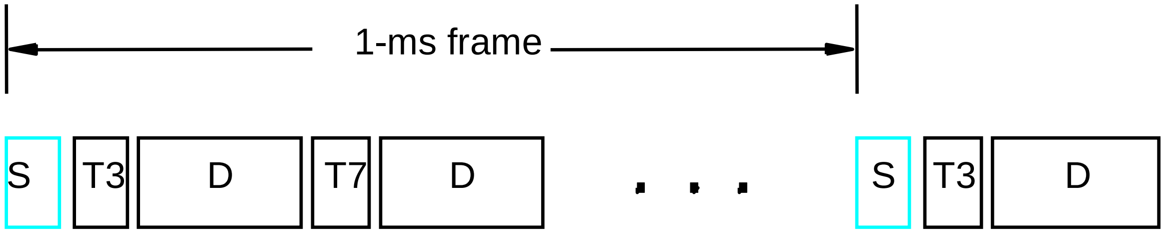
Output Transfer



USB FRAMES



(a) SOF Packet



S - Start-of-frame

T*n*- Token packet, address = *n*

D - Data packet

A - ACK packet

(b) Frame example

2 marks

1. What are the basic functional units of a computer?

Ans: A computer consists of five functionally independent main parts namely

- Input Unit
- Memory Unit
- Arithmetic and logic Unit
- Output Unit
- Control Unit

2. Define RAM.

Ans: Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called random access memory.

3. Define memory access time.

Ans: The time required to access one word is called memory access time.

4. What is instruction register (IR) and program counter (PC) used for ?

Ans: The instruction register (IR) holds the instruction that is currently being executed .Its output is available to the control circuits which generate the timing signals that control the various processing elements. The program counter PC is used to keep track of the execution of the program. It contains the memory address of the next instruction to be fetched and executed.

5. What do you mean by memory address register(MAR) and memory data register(MDR)?

ans: The MAR holds the address of the location to be accessed.

6. What is an interrupt?

Ans: An interrupt is a request from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine.

7. Explain about Bus.

Ans: Bus is a group of lines that serves as a connecting path for several devices. In addition to the lines that carry the data , the bus must have the lines for address and control purposes.

8. What do you mean by multiprogramming or multitasking?

Ans: The operating system manages the concurrent execution of several application programs to make best possible use of computer resources. This pattern of concurrent execution is called multiprogramming or multitasking.

9. Give the basic performance equation.

Ans: The basic performance equation is given as

$T = N * S/ R$

T = It is the processor time required to execute a program

N= It is the actual number of instruction executions.

S = It is the average number of basic steps neede to execute one machine instruction. R = It is the clock rate.

10. Explain the concept of pipelining.

Ans: Pipelining is the means of executing machine instructions concurrently. It is the effective way of organizing concurrent activity in a computer system. It is a process of substantial improvement in the performance by overlapping the execution of successive instructions.

11. What are the two techniques used to increase the clock rate R?

Ans: The two techniques used to increase the clock rate R are:

1. The integrated – circuit (IC) technology can be increased which reduces the time needed to complete a basic step.

2. We can reduce the amount of processing done in one basic step.

12. What is Big – Endian and Little- Endian representations.

Ans: The Big- endian is used when lower byte addresses are used for the more significant bytes (The leftmost bytes) of the word.

The little-endian is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word.

13. What are the different types of addressing modes available?

Ans: The different types of addressing modes available are:

- Immediate addressing mode
- Register addressing mode
- Direct or absolute addressing mode
- Indirect addressing mode
- Indexed addressing mode
- Relative addressing mode
- Autoincrement
- Autodecrement

14. What is indirect addressing mode?

Ans: The effective address of the operand is the contents of a register or memory location whose address appears in the instruction

15. What is indexed addressing mode?

Ans: The effective address of the operand is generated by adding a constant value to the contents of a register.

16. Define autoincrement mode of addressing?

Ans: The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next item in the list.

17. Define autodecrement mode of addressing?

Ans: The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

18. What are condition code flags? What are the commonly used flags?

Ans:The processor has to keep track of the information about the results of various operations for the subsequent conditional branch instructions. This is done by recording required information in individual bits called condition code flags.

Four commonly used flags are:

- N(Negative)
- Z(Zero)
- V(overflow)
- C(Carry)

19. What do you mean by assembler directives?

Ans: These are the instructions which direct the program to be executed. They have no binary equivalent so they are called pseudo-opcodes. These instructions are used to define symbols, allocate space for variable, generate fixed tables etc.

Examples : END, NAME

22. What do you man by relative addressing mode?

Ans: The effective address is determined by the index mode using the program counter in place of the general purpose register R_i.

23. What is Stack?

Ans: A stack is a list of data elements, usually words or bytes with the accessing restriction that elements can be added or removed at one end of the list only. It follows last in first out (LIFO) mechanism.

24. What is a queue?

Ans: Is a type of datastructure in which the data are stored in and retrieved on a First in first out(FIFO) basis. It grows in the direction of increasing addresses in the memory. New data are added at the back (High-address end) and retrieved from the front (low-address end) of the queue.

25. What are the difference between Stack and Queue?

STACK : One end of stack is fixed (the bottom) while the other end rises and falls as data are pushed and popped Single Pointer is needed to point to the top of the stack

QUEUE : Both ends of a queue move to higher addresses Two pointers are needed to keep track of the two ends of the queue.

UNIT II - ARITHMETIC UNIT

1. What is half adder?

Ans: A half adder is a logic circuit with two inputs and two outputs, which adds two bits at a time, producing a sum and a carry.

2. What is full adder?

Ans: A full adder is logic circuit with three inputs and two outputs, which adds three bits at a time giving a sum and a carry.

3. What is signed binary?

Ans: A system in which the leading bit represents the sign and the remaining bits the magnitude of the number is called signed binary. This is also known as sign magnitude.

4. What are the two approaches used to reduce delay in adders?

Ans:1) The first approach is to use the fastest possible electronic technology in

5. What is a carry look-ahead adder?

Ans: The input carry needed by a stage is directly computed from carry signals obtained from all the preceding stages i-1,i-2,…..0, rather than waiting for normal carries to supply slowly from stage to stage. An adder that uses this principle is a called carry look-ahead adder.

6. What are the main features of Booth's algorithm?

Ans: 1) It handles both positive and negative multipliers uniformly.

2) It achieves some efficiency in the number of addition required when the multiplier has a few large blocks of 1s.

7. How can we speed up the multiplication process?

Ans: There are two techniques to speed up the multiplication process:

1) The first technique guarantees that the maximum number of summands that must be added is n/2 for n-bit operands. 2) The second technique reduces the time needed to add the summands.

8. What is bit pair recoding? Give an example.

Ans: Bit pair recoding halves the maximum number of summands. Group the Booth-recoded multiplier bits in pairs and observe the following: The pair (+1 -1) is equivalent to to the pair (0 +1). That is instead of adding -1 times the multiplicand m at shift position i to +1 * M at position i+1, the same result is obtained by adding +1 * M at position i. Eg: 11010 – Bit Pair recoding value is 0 -1 -2

9. What are the two methods of achieving the 2's complement?

a. Take the 1's complement of the number and add 1.

b. Leave all least significant 0's and the first unchanged and then complement the remaining bits.

10. What is the advantage of using Booth algorithm?

Ans: 1) It handles both positive and negative multiplier uniformly.

2) It achieves efficiency in the number of additions required when the multiplier has a few large blocks of 1's.

3) The speed gained by skipping 1's depends on the data.

11. Write the algorithm for restoring division.

Ans: Do the following for n times:

1) Shift A and Q left one binary position.

2) Subtract M and A and place the answer back in A.

3) If the sign of A is 1, set q0 to 0 and add M back to A.

Where A- Accumulator, M- Divisor, Q- Dividend.

Step 1: Do the following for n times:

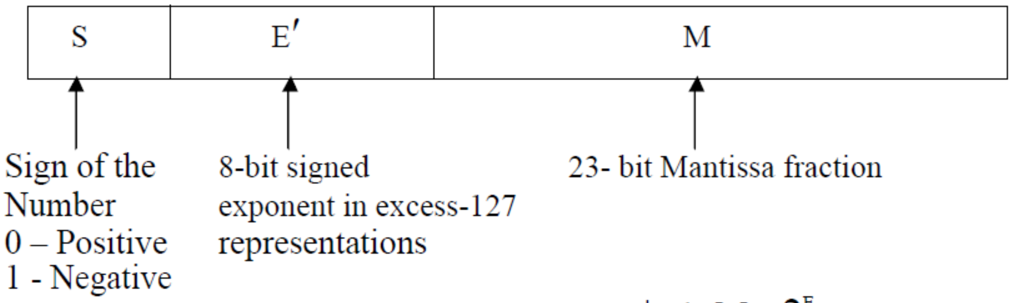
1) If the sign of A is 0 , shift A and Q left one bit position and subtract M from A; otherwise , shift A and Q left and add M to A.

2) Now, if the sign of A is 0,set q0 to 1;otherwise , set q0 to0.

Step 2: if the sign of A is 1, add M to A.

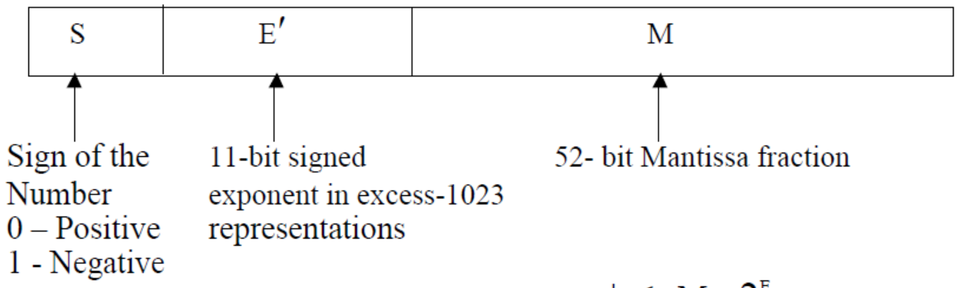
13. Give the IEEE standard for floating point numbers for single precision number.

Ans:


S
E′
M
Sign of the Number
0 – Positive
1 - Negative
8-bit signed exponent in excess-127 representations
23- bit Mantissa fraction

value represented= $\pm\ 1.M\times 2^{E}$-127

14. Give the IEEE standard for floating point numbers for double precision number.


S
E′
M
Sign of the Number
0 – Positive
1 - Negative
11-bit signed exponent in excess-1023 representations
52- bit Mantissa fraction

Ans:

value represented= $\pm\ 1.M\times 2^{E}$-1023

15. When can you say that a number is normalized?

Ans: When the decimal point is placed to the right of the first (nonzero) significant digit, the number is said to be normalized.

Ans: The end values 0 to 255 of the excess-127 exponent E are used to represent special values such as:

a) When E^1= 0 and the mantissa fraction M is zero the value exact 0 is represented.

b) When E^1= 255 and M=0, the value ∞ is represented.
c) When E^1= 0 and $M \neq 0$, denormal values are represented.
d) When E^1= 2555 and M≠0, the value represented is called Not a number.

17. Write the Add/subtract rule for floating point numbers.

Ans: 1) Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2) Set the exponent of the result equal to the larger exponent.
3) Perform addition/subtraction on the mantissa and determine the sign of the result
4) Normalize the resulting value, if necessary.

18. Write the multiply rule for floating point numbers.

Ans:1) Add the exponent and subtract 127.
2) Multiply the mantissa and determine the sign of the result .
3) Normalize the resulting value , if necessary.

19. What is guard bit?

Ans: Although the mantissa of initial operands are limited to 24 bits, it is important to retain extra bits, called as guard bits.

20. What are the ways to truncate the guard bits?

Ans: There are several ways to truncate the guard bits:
1) Chooping
2) Von Neumann rounding
3) Rounding

21. Define carry save addition(CSA) process.

Ans: Instead of letting the carries ripple along the rows, they can be saved and introduced into the next roe at the correct weighted position. Delay in CSA is less than delay through the ripple carry adder.

22. What are generate and propagate function?

Ans: The generate function is given by
$G_i=x_iy_i$ and
The propagate function is given as $P_i=x_i+y_{i.}$

23. What is excess-127 format?

Ans: Instead of the signed exponent E, the value actually stored in the exponent field is and unsigned integer E□

24. In floating point numbers when so you say that an underflow or overflow has occurred?

Ans: In single precision numbers when an exponent is less than -126 then we say that an underflow has occurred. In single precision numbers when an exponent is less than +127 then we say that an overflow has occurred.

UNIT III- BASIC PROCESSING UNIT

1. What are the basic steps required to execute an instruction by the processor?

Ans: The basic steps required to execute an instruction by the processor are:
1) Fetch the contents of the memory location pointed to by the PC. They are loaded into the IR.

IR←[[PC]]

2) Assuming that the memory is byte addressable, increment the contents of the PC by 4, that is

PC← [PC} + 4

3) Carry out the action specified by the instruction in the IR.

2. Define datapath in the processor unit.

Ans: The registers , The ALU and the interconnecting bus are collectively referred to as the datapath.

3. What is processor clock?

Ans: All operations and data transfers within the processor take place within time periods defined by the processor clock .

4. Write down the control sequence for Move (R1), R2.

Ans: The control sequence is :

$R1_{out}$, MAR_{in},Read

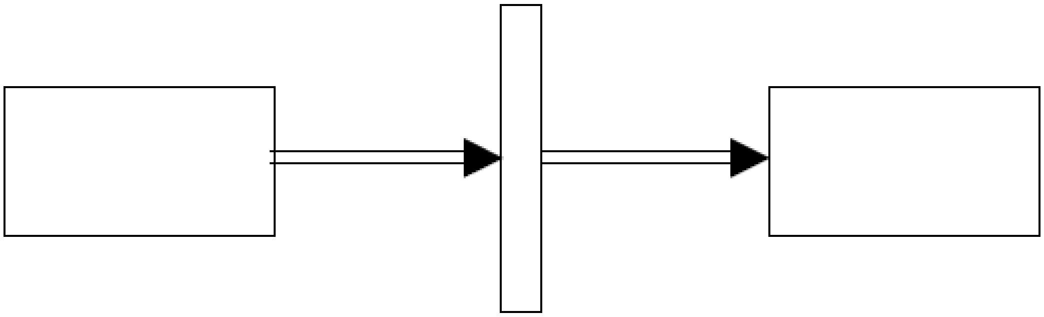
MDRoutE,WMFC

MDR_{out},$R2_{in}$

5.Define register file .

Ans: A set of general purpose registers are called as register file Each register from register file R_0 is individually addressable.

6. Draw the hardware organization of two-stage pipeline?



Inter stage buffer

7.What is the role of cache memory in pipeline?

Ans: The use of cache memory is to solve the memory access problem. When cache is included in the processor the access time to the cache is usually the same time needed to perform other basic operation inside the processor.

8.Name the methods for generating the control signals.

Ans: The methods for generating the control signals are:

1) Hardwired control

2) Microprogrammed control

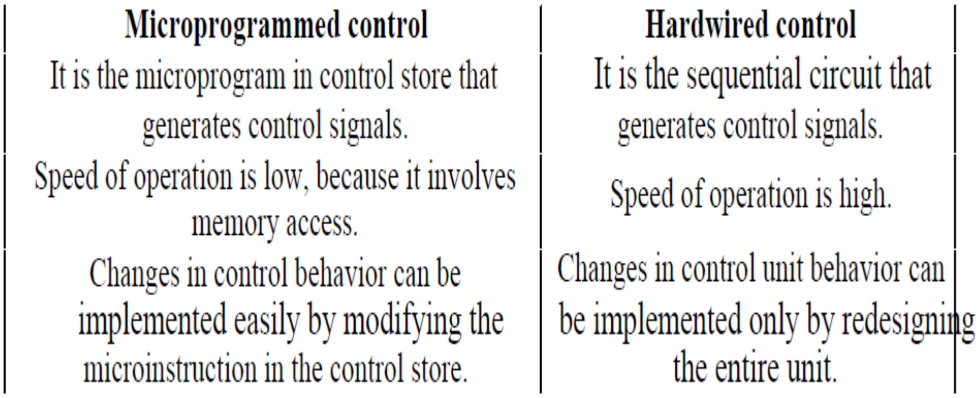
9. Define hardwired control.

Ans: Hard-wired control can be defined as sequential logic circuit that generates specific sequences of control signal in response to externally supplied instruction.

10. Define microprogrammed control.

Ans: A microprogrammed control unit is built around a storage unit is called a control store where all the control signals are stored in a program like format. The control store stores a set of microprograms designed to implement the behavior of the given instruction set.

11. Differentiate Microprogrammed control from hardwired control.



Microprogrammed control	Hardwired control
It is the microprogram in control store that generates control signals.	It is the sequential circuit that generates control signals.
Speed of operation is low, because it involves memory access.	Speed of operation is high.
Changes in control behavior can be implemented easily by modifying the microinstruction in the control store.	Changes in control unit behavior can be implemented only by redesigning the entire unit.

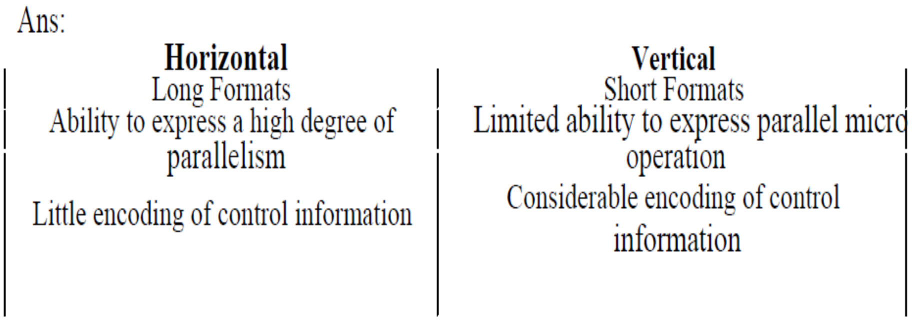
12. Define parallelism in microinstruction

Ans: The ability to represent maximum number of micro operations in a single microinstruction is called parallelism in microinstruction.

13. What are the types of microinstructions available?

Ans: 1) Horizontal microinstruction

2) Vertical microinstruction



Ans:

Horizontal	Vertical
Long Formats	Short Formats
Ability to express a high degree of parallelism	Limited ability to express parallel micro operation
Little encoding of control information	Considerable encoding of control information

15. What is MFC?

Ans: To accommodate the variability in response time, the processor waits until it receives an indication that the requested read operation has been completed. This is accomplished by a control signal called Memory – Function – Completed.

16. What are the major characteristics of a pipeline?

Ans: The major characteristics of a pipeline are:

a) Pipelining cannot be implemented on a single task, as it works by splitting multiple tasks into a number of subtasks and operating on them simultaneously.

b) The speedup or efficiency achieved by suing a pipeline depends on the number of pipe stages and the number of available tasks that can be subdivided.

c) If the task that can be subdivided has uneven length of execution times, then the speedup of the pipeline is reduced.

d) Though the pipeline architecture does not reduce the time of execution of a single task, it reduces the overall time taken for the entire job to get completed.

17. What is a pipeline hazard?

Ans: Any condition that causes the pipeline to stall is called hazard. They are also called as stalls or bubbles.

18. What are the types of pipeline hazards?

Ans: The various pipeline hazards are:

1. Data hazard
2. Structural Hazard
3. Control Hazard.

19. What is data hazard?

Ans: Any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline is called data hazard.

20. What is Instruction or control hazard?

Ans: The pipeline may be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called control hazards or instruction hazard.

21. What is side effect?

Ans: When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect.

22. What do you mean by branch penalty?

Ans: The time lost as a result of a branch instruction is often referred to as branch penalty.

23. What is branch folding?

Ans: When the instruction fetch unit executes the branch instruction concurrently with the execution of the other instruction, then this technique is called branch folding.

24. What do you mean by delayed branching?

Ans: Delayed branching is used to minimize the penalty incurred as a result of conditional branch instruction. The location following the branch instruction is called delay slot. The instructions in the delay slots are always fetched and they are arranged such that they are fully executed whether or not branch is taken. That is branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory hence the name delayed branching.

25. What are the two types of branch prediction techniques available?

Ans: The two types of branch prediction techniques are

1) Static branch prediction

2) Dynamic branch prediction

UNIT IV - MEMORY SYSTEM

1. Define Memory Access Time?

Ans: It is the time taken by the memory to supply the contents of a location, from the time, it receives “READ”.

2. Define memory cycle time.

Ans: It is defined as the minimum time delay required between the initaiation of two successive memory operations.

3. What is RAM?

This storage location can be accessed in any order and access time is independent of the location being accessed

4. Explain virtual memory.

Ans: The data is to be stored in physical memory locations that have addresses different from those specified by the program. The memory control circuitry translates the address specified by the program into an address that can be used to access the physical memory.

5. List the various semiconductors RAMs?

i] Static RAM.

ii] Dynamic RAM

6. What do you mean by static memories?

Ans: Memories that consist of circuits capable of retaining their state as long as power is applied are known as static memories.

7. Define DRAM’s.

Ans: Atatic Rams are fast but their cost is high so we use dynamic RAMs which do not retain their state indefinitely but here the information are stored in the form of charge on a capacitor

8. Define DDR SDRAM.

The double data rate SDRAMs are the faster version of SDRAM. It transfers data on both edges of the clock.

9. What is ROM?

ROM is by definition Non Volatile Preprogrammed with information permanently encoded in the chip.

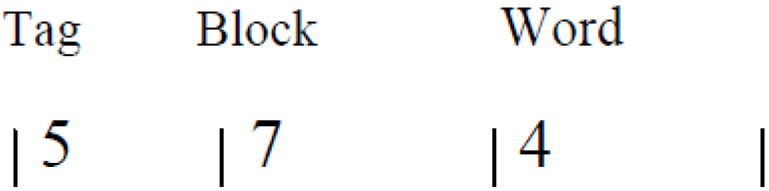
10. What is the mapping procedures adopted in the organization of a Cache Memory?

i) Associative mapping.

ii) Direct mapping.

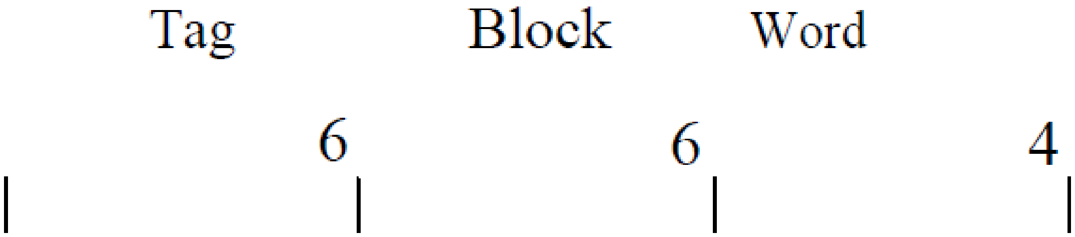
iii) Set-associative mapping

11. Give the format for main memory address using direct mapping function for 4096 blocks in main memory and 128 blocks in cache with 16 blocks per cache.



Tag	Block	Word
5	7	4

12. Give the format for main memory address using set associative mapping function for 4096 blocks in main memory and 128 blocks in cache with 16 blocks per cache.



Tag	Block	Word
6	6	4

13. Define Hit and Miss?

The performance of cache memory is frequently measured in terms of a quantity called hit ratio. When the CPU refers to memory and finds the word in cache, it is said to produce a hit. If the word is not found in cache, then it is in main memory and it counts as a miss.

14. Write the formula for the average access time experienced by the processor in a system with two levels of caches.

Ans: The formula for the average access time experienced by the processor in a system with two levels of caches is

t_{ave}= h1C1+(1-h1)h2C2+(1-h1)(1-h2)M

h1= hit rate in the L1 cache.

h2= hit rate in the L2 cache.

C1=time to access information in the L1 cache.

C2= time to access information in the L1 cache.

M= time to access information in the main memory.

15. What are the enhancements used in the memory management?

Ans: 1) Write Buffer

2) Pre fetching

3) Look- up Cache.

16. What do you mean by memory management unit?

Ans: The memory management unit is a hardware unit which translates virtual addresses into physical addresses in the virtual memory techniques.

17. Explain main (primary) memory.

Ans: This memory stores programs and data that are active in use. Storage locations in main memory are addressed directly by the CPU's load and store instructions.

18. What do you mean by seek time?

Ans: It is the time required to move the read/write head in the proper track.

19. What is RAID?

Ans: High performance devices tend to be expensive. So we can achieve very high performance at a reasonable cost by using a number of low-cost devices oerating in parallel. This is called RAID(Redundant array of Inexpensive Disks).

20. Define data stripping?

Ans: A single large file is stored in several separate disk units by breaking the file up into a number of smaller pieces and storing these pieces on different disks. This is called data stripping.

22. How the data is organized in the disk?

Ans: Each surface is divided into concentric tracks and each track is divided into sectors. The set of corresponding tracks on all surfaces of a stack of disks forms a logical cylinder. The data are accessed by using read/write head.

23. Define latency time.

Ans: This is the amount of time that elapses after the head is positioned over the correct track until the starting position of the addressed sector passes under the read/write head.

UNIT V- I/O ORGANIZATION

1. Why IO devices cannot be directly be connected to the system bus?

Ans: The IO devices cannot be directly connected to the system bus because

i. The data transfer rate of IO devices is slower that of CPU.

ii. The IO devices in computer system has different data formats and
work lengths that of CPU. So it is necessary to use a module between system bus and IO device called IO module or IO system

2. What are the major functions of IO system?

Ans: i. Interface to the CPU and memory through the system bus.

ii. Interface to one or more IO devices by tailored data link.

3. What is an I/O Interface?

Ans: Input-output interface provides a method for transferring binary information between internal storage, such as memory and CPU registers, and external I/O devices

4. Write the factors considered in designing an I/O subsystem?

Ans:

1. Data Location: Device selection, address of data with in device(track, sector etc)
2. Data transfer: Amount, rate to or from device.
3. Synchronization: Output only when device is ready, input only when
4. Memory or between an I/O device and CPU.

5. Explain Direct Memory Access.

Ans: A modest increase in hardware enables an IO device to transfer a block of information to or from memory without CPU intervention. This task requires the IO device to generate memory addresses and transfer data through the bus using interface controllers.

6. Define DMA controller.

Ans: The I/O device interface control circuit that is used for direct memory access is known as DMA controller.

7. What is polling?

Ans: Polling is a scheme or an algorithm to identify the devices interrupting the processor. Polling is employed when multiple devices interrupt the processor through one interrupt pin of the processor.

8. What is the need of interrupt controller?

Ans: The interrupt controller is employed to expand the interrupt inputs. It can handle the interrupt requests from various devices and allow one by one to the processor.

9. What is a Priority Interrupt?

Ans: A priority interrupt is an interrupt that establishes a priority over the various sources to determine which condition is to be serviced first when two or more requests arrive simultaneously.

10. Define bus.

Ans: When a word of data is transferred between units, all the bits are transferred in parallel over a set of lines called bus. In addition to the lines that carry the data, the bus must have lines for address and control purposes.

11. Define synchronous bus.

Ans: Synchronous buses are the ones in which each item is transferred during a time slot(clock cycle) known to both the source and destination units. Synchronization can be achieved by connecting both units to a common clock source.

12. Define asynchronous bus.

Ans: Asynchronous buses are the ones in which each item being transferred is accompanied by a control signal that indicates its presence to the destination unit. The destination can respond with another control signal to acknowledge receipt of the items.

13. What do you mean by memory mapped I/O?

Ans: In Memory mapped I/O, there are no specific input or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words.

14. What is program-controlled I/O?

Ans: In program controlled I/O the processor repeatedly checks a status flags to achieve the required synchronization between the processor and an input and output device.

15. Define interrupt.

Ans: An interrupt is any exceptional event that causes a CPUU to temporarily transfer control from its current program to another program , an interrupt handler that services the event in question.

16. Define exception.

Ans: The term exception is used to refer to any event that causes an interruption

17. What are the different methods used for handling the situation when multiple interrupts occurs?

Ans: 1) Vectores interrupts2) Interrupt nesting 3) Simultaneous Requests.

18. What is a privileged instruction?

Ans: To protect the operating system of a computer from being corrupted by user programs, certain instructions can be executed only while the processor is in the supervisor mode. These are called privileged instruction.

19. What is bus arbitration?

Ans: it is process by which the next device to become the bus master is selected and bus mastership is transferred to it. There are two ways for doing this:

1. Centralized arbitration 2. Distributed arbitration.

20. What is port? What are the types of port available?

Ans: An I/O interface consists of circuitry required to connect an I/O device to computer bus. One side consists of a data path with its associated controls to transfer data between the interface and I/O device. This is called port. It is classified into:

1) Parallel port

2) Serial port.

21. What is a parallel port?

Ans: A parallel port transfers data in the form a number of bits, typically 8 to 16, simultaneously to or from the device.

22. What is a serial port?

Ans: A serial port transfers and receives data one bit at a time.

23. What is PCI bus?

Ans: The Peripheral component interconnect(PCI) bus is a standard that supports the any particular processor.

24. What is SCSI?

Ans: It is the acronym for small computer system interface. It refers to a standard bus defined ANSI. Devices such as disks are connected to a computer via 50-wire cable, which can be upto 25 meters in length and can transfer data at rate up to 55 megabytes/s.

25. Define USB.

Ans: The Universal Serial Bus(USB) is an industry standard developed to provide two speed of operation called low-speed and full-speed. They provide simple, low cost and easy to use interconnection system.

16 MARKS QUESTIONS WITH HINTS:

1. What are the functional units of a computer? Explain briefly.

Hints: The different functional units are:

1) Input Unit.

2) Output Unit.
3) Memory Unit.
4) Arithmetic & logic Unit.
5) Control Unit.

2. Explain the basic operational concepts of a computer.

Hints: Explain about the MAR, MDR, and the connection between the processor and the memory with a neat diagram.

3. Describe the different classes of instruction format with example and different addressing modes.

Hints: The different instruction formats are :
1) Three address instruction
2) Two address instruction
3) Zero address instruction

The different addressing modes are:

- Immediate addressing mode
- Register addressing mode
- Direct or absolute addressing mode
- Indirect addressing mode
- Indexed addressing mode
- Relative addressing mode
- Autoincrement
- Autodecrement

4. Explain the basic input operations with suitable examples.

Hints: Expalin about program-controlled I/O and memory mapped I/O. Draw the diagram of bus connection for processor , keyboard and display.

5. Write short notes on
i) Software performance
ii) Memory locations and addresses

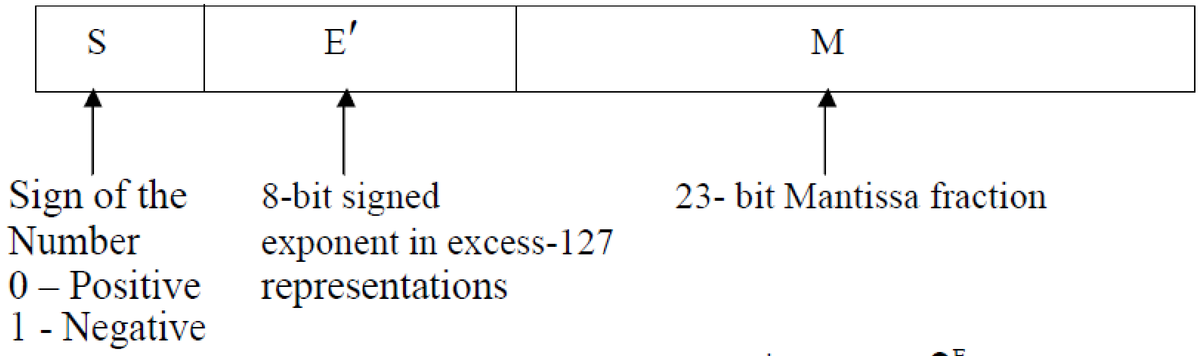
Hints:
Explain about byte addressability, big endian and little endian assignments, word alignment.

6. Describe the multiplication speed up technique with an example.

Hints: There are two techniques to speed up the multiplication process:
1) The first technique guarantees that the maximum number of summands that must be added is n/2 for n-bit operands ie bit pair recoding .
2) The second technique reduces the time needed to add the summands i.e Carry save addition.

7. Explain the floating point Addition – subtraction unit with neat diagram.

Hints: In some cases, the binary point is variable and is automatically adjusted as computation proceeds. In such case, the binary point is said to float and the numbers are
called floating point numbers.


S
E′
M
Sign of the Number
0 – Positive
1 - Negative
8-bit signed exponent in excess-127 representations
23- bit Mantissa fraction

value represented= $\pm\ 1.M\times 2^{E}$-127

1) Choose the number with the smaller exponent and shift its mantissa right a number of steps equal to the difference in exponents.
2) Set the exponent of the result equal to the larger exponent.
3) Perform addition/subtraction on the mantissa and determine the sign of the result
4) Normalize the resulting value, if necessary.
Draw the diagram.

8. Explain the organization of a sequential binary multiplier with example.

Hints:
procedure.Draw the diagram.

9. Explain the Booth algorithm. Multiply 11*9 using Booth algorithm

Hints: Explain about the Booth algorithm.The advantages are:
1) It handles both positive and negative multiplier uniformly.
2) It achieves efficiency in the number of additions required when the multiplier has a few large blocks of 1’s.
3) The speed gained by skipping 1’s depends on the data

10. Explain the Integer division techniques with suitable example.

Hints: The algorithm for restoring division:
Do the following for n times:
1) Shift A and Q left one binary position.
2) Subtract M and A and place the answer back in A.
3) If the sign of A is 1, set q0 to 0 and add M back to A.
Where A- Accumulator, M- Divisor, Q- Dividend
Give an example.
The algorithm for non restoring division:
Do the following for n times:***Step 1:*** Do the following for n times:
1) If the sign of A is 0 , shift A and Q left one bit position and subtract M from A; otherwise , shift A and Q left and add M to A.
2) Now, if the sign of A is 0,set q0 to 1;otherwise , set q0 to0.
Step 2: if the sign of A is 1, add M to A. Give an example.

11. Explain the multiple bus organization structure with neat diagram.

Hints: The multiple bus organization is using more buses instead of one bus to reduce the number of steps needed and to provide multiple paths that enable several transfers to take place in parallel.

12. Describe the Hardwired control method for generating the control signals

Hints: Hard-wired control can be defined as sequential logic circuit that generates specific sequences of control signal in response to externally supplied instruction

13.Describe the micro programmed control unit in detail.

Hints: A micro programmed control unit is built around a storage unit is called a control store where all the control signals are stored in a program like format. The control store stores a set of micro programs designed to implement the behavior of the given instruction set.

14. Give the organization of the internal data path of a processor that supports a 4-stage pipeline for instructions and uses a 3- bus structure and discuss the same.

Hints: The speed of execution of programs can be improved by arranging the hardware so that more than one operation can be performed at the same time.
Explain about the 4- stage pipeline.

15. What is pipelining? What are the various hazards encountered in pipelining? Explain in detail.

Hints: The major characteristics of a pipeline are:

a) Pipelining cannot be implemented on a single task, as it works by splitting multiple tasks into a number of subtasks and operating on them simultaneously.

b) The speedup or efficiency achieved by suing a pipeline depends on the number of pipe stages and the number of available tasks that can be subdivided.

c) If the task that can be subdivided has uneven length of execution times, then the speedup of the pipeline is reduced.

d) Though the pipeline architecture does not reduce the time of execution of a single task, it reduces the overall time taken for the entire job to get completed.

The various pipeline hazards are:

1. Data hazard
2. Structural Hazard
3. Control Hazard.

16. Describe the three mapping techniques used in cache memories with suitable Example.

Hints: The cache memory is a fast memory that is inserted between the larger slower main memory and the processor. It holds the currently active segments of a program and their data.

Associative mapping.
Direct mapping.
Set-associative mapping

17. Explain with neat diagram the internal organization of bit cells in a memory chip.

Hints: Memory cells are usually organized in the form of an array, in which each cell is capable of storing one bit of information. Each row consists a memory word, and all cells of a row are connected to a common line referred to as word line, which is driven by the address decoder on the chip.

18. Explain the various secondary storage devices in detail.

Hints: The various secondary storage devices are:

1. Magnetic hard disks
2. Optical disks
3. Magnetic tape systems Refer page no. 344-359

19. What is memory interleaving? Explain with neat diagram.

Hints: The main memory of a computer is structure as a collection of physically separate modules each with its own address buffer register and data buffer register, memory access operations may proceed in more than one module at the same time. Thus the aggregate rate of transmission of words to and from the main memory system can be increased.

20. Describe the data transfer method using DMA.

Hints: A modest increase in hardware enables an IO device to transfer a block of information to or from memory without CPU intervention. This task requires the IO device to generate memory addresses and transfer data through the bus using interface controllers.

21. Explain about the interrupts in detail

Hints: An interrupt is any exceptional event that causes a CPUU to temporarily transfer control from its current program to another program , an interrupt handler that services the event in question.

22. Explain the different types of buses with neat diagram.

Hints: When a word of data is transferred between units, all the bits are transferred in parallel over a set of lines called bus. In addition to the lines that carry the data, the bus must have lines for address and control purposes.The different types of buses are:

1. Synchronous Buses:

Synchronous buses are the ones in which each item is transferred during a time slot(clock cycle) known to both the source and destination units. Synchronization can be achieved by connecting both units to a common clock source.

2. Asynchronousbuses

by a control signal that indicates its presence to the destination unit. The destination can respond with another control signal to acknowledge receipt of the items.

23. Explain the various interface circuits.

Hints: An I/O interface consists of circuitry required to connect an I/O device to computer bus. One side consists of a data path with its associated controls to transfer data between the interface and I/O device. This is called port. It is classified into:

1) Parallel port

2) Serial port.

24. Explain in details the various standard I/O interfaces.

Hints: The various standard I/O interfaces are:

1. The Peripheral component interconnect(PCI) bus is a standard that supports the functions found on a processor bus but in a standardized format that is independent of any particular processor

2. It is the acronym for small computer system interface. It refers to a standard bus defined ANSI. Devices such as disks are connected to a computer via 50-wire cable, which can be upto 25 meters in length and can transfer data at rate up to 55 megabytes/s.

3. The Universal Serial Bus(USB) is an industry standard developed to provide two speed of operation called low-speed and full-speed. They provide simple, low cost and easy to use interconnection system.

25. Discuss in detail the various measures of performance of a computer

- Processor clock
- Pipelining and super scalar operation
- Clock rate
- Instruction set
- Compiler

26. Discuss the following:

- Execution steps
- Diagram
- Branching.
- Explanation
- Diagram

27. Explain in detail the data transfer between the memory & I/O unit.

- Program controlled I/O
- Flags(SIN, SOUT)
- Buffers(DATAIN, DATAOUT)
- Coding
- Diagram

29.Explain in detail the principle of Carry look ahead adder?

- Generate function
- Propagate function
- 4-bit carry look-ahead adder

30.Explain the sequential circuit binary multiplier and give an example for it.

- Explanation
- Diagram

31.Multiply the following pair of signed 2's complement number using Bit pair recoding

of the multipliers.

A = 010111

NonRestoring division:10101 / 00101

32.Expalin 4X4 array multiplier . What is the delay in this case?

- Explanation
- Diagram

33. Draw the organization of a single-bus processor and give the control sequences for fetching a word from memory, storing a word in memory , executing a complete instruction and unconditional and conditional branch.

- Diagram
- Control Sequences

34. Write notes on super scalar operation?

- Explanation
- Diagram

35. Write notes on semiconductor RAM memories.

- Internal organization of memory chips
- Static memories
- Asynchronous DRAMs
- Synchronous DRAMs

36.Write notes on various types of ROMs.

- ROM
- PROM
- EPROM
- EEPROM

37. Write notes on Virtual memories.

- Address Translation
- Diagram
- Explanation