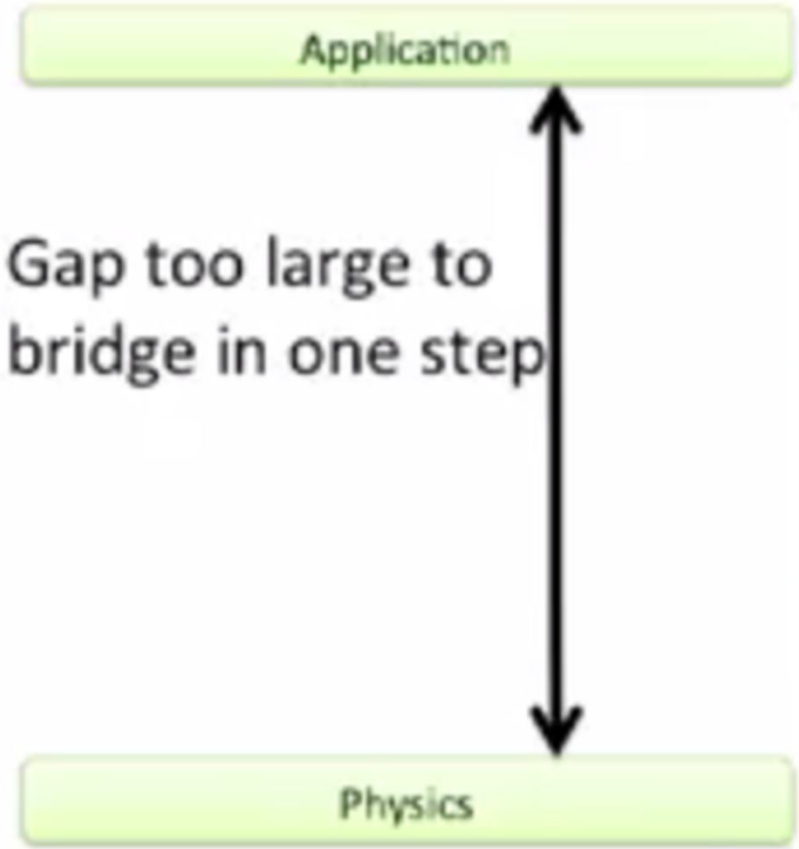
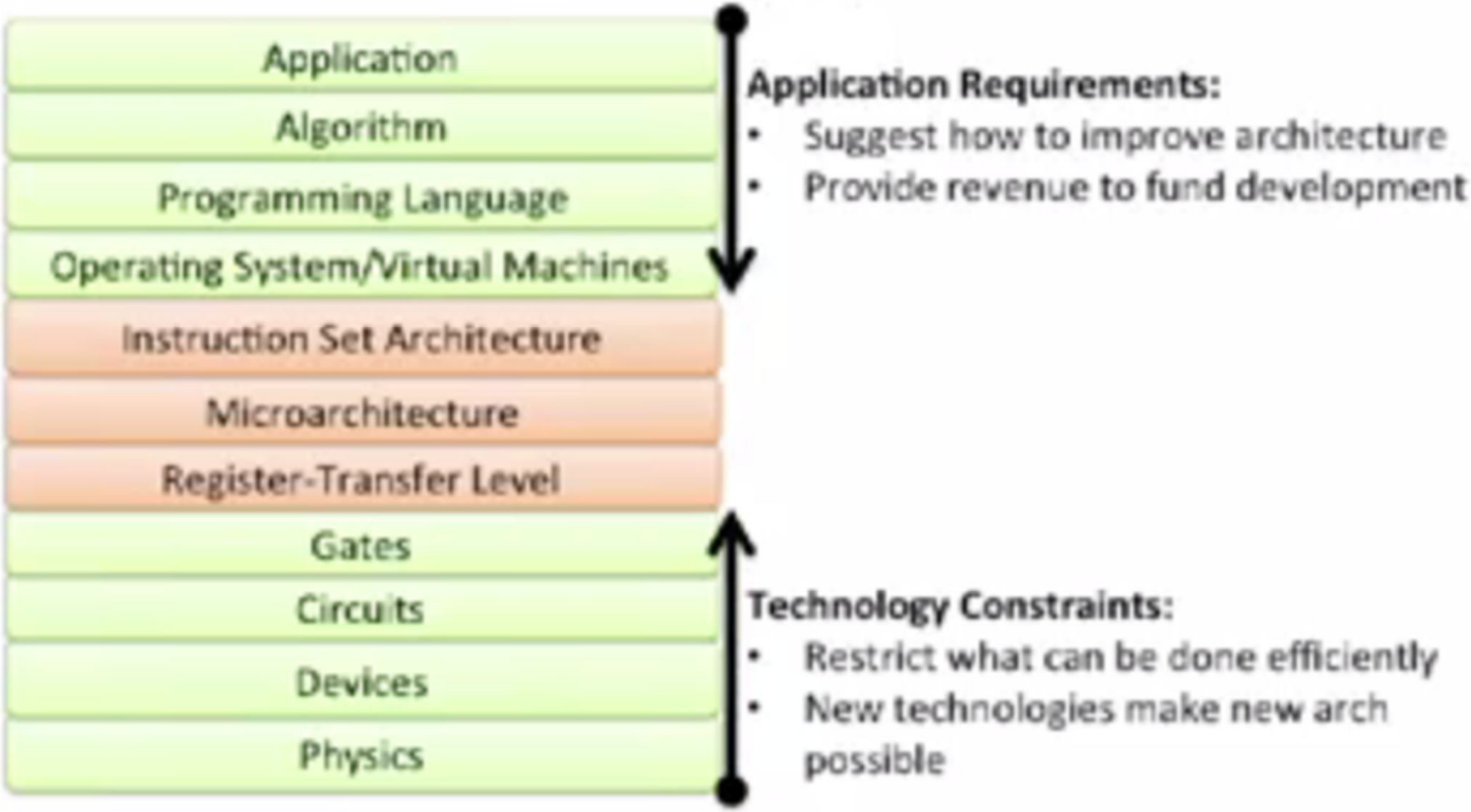
What is Computer Architecture?



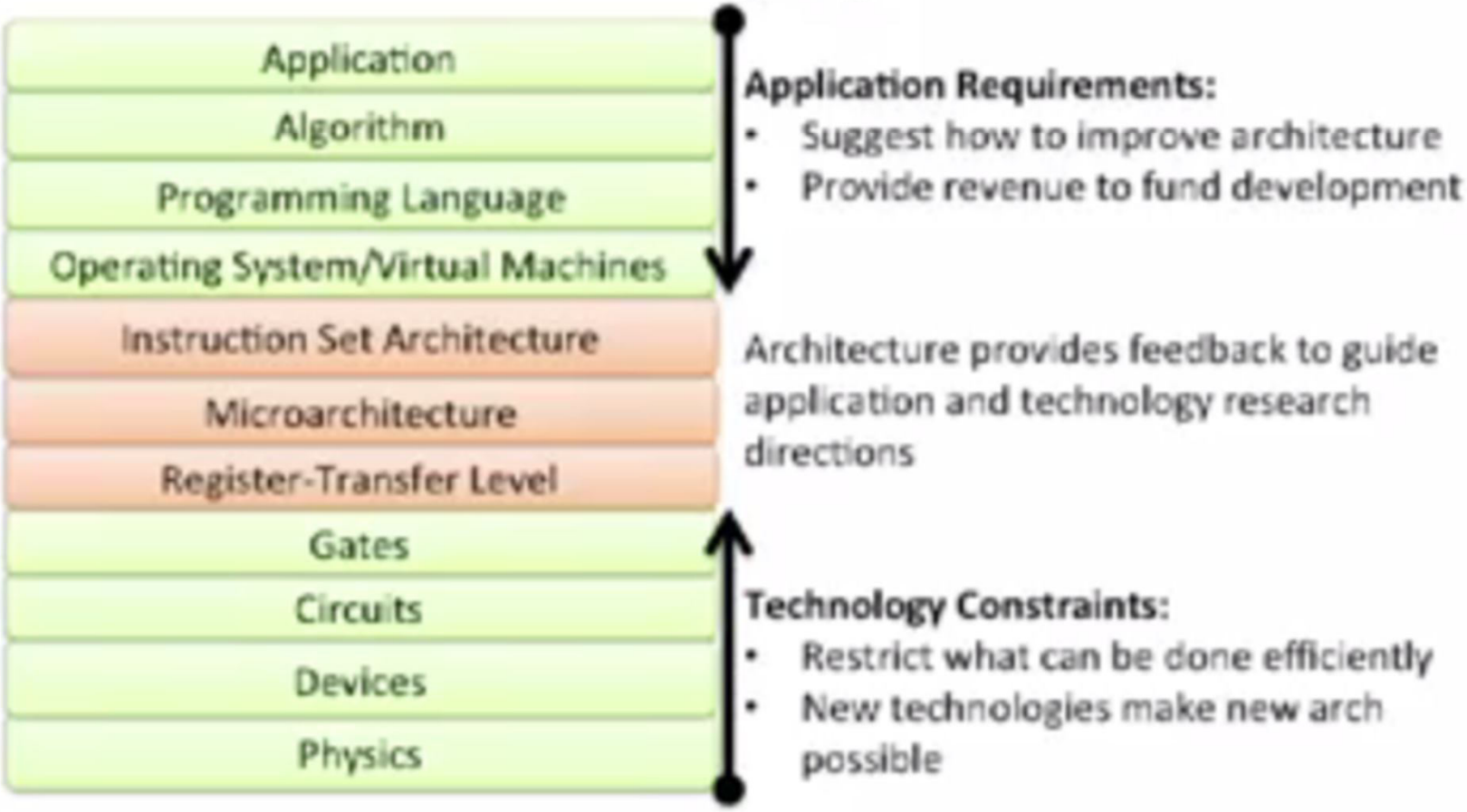
In its broadest definition, computer architecture is the design of the abstraction/ implementation layers that allow us to execute information processing applications efficiently using manufacturing technologies

Abstractions in Modern Computing Systems

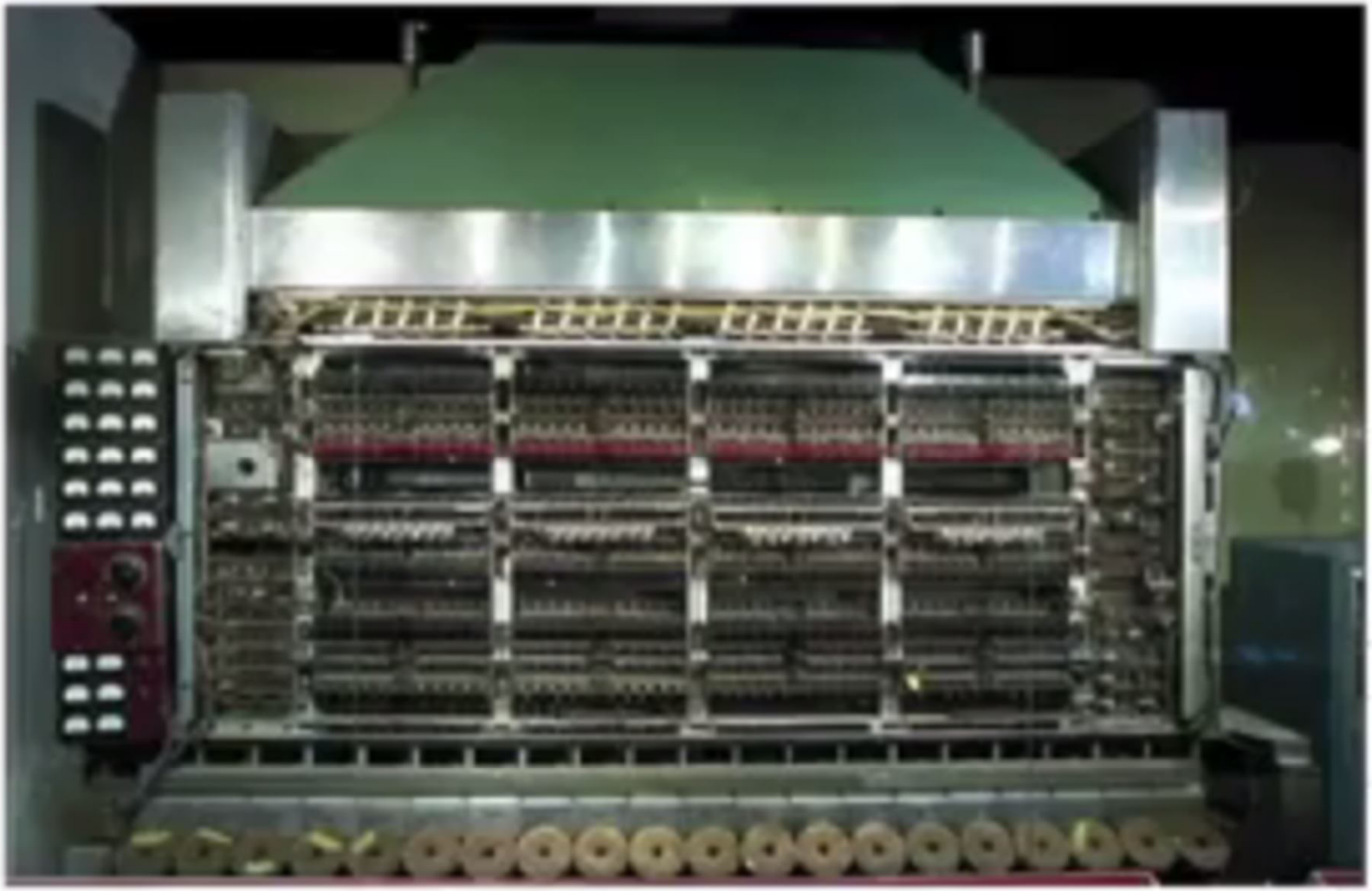
Computer Architecture is Constantly Changing



Computer Architecture is Constantly Changing



Computers Then…



IAS Machine. Design directed by John von Neumann.
First booted in Princeton NJ in 1952
Smithsonian Institution Archives (Smithsonian Image 95-06151)

Computers Now

- Sensor Networks
- Cameras
- Smartphones
- Mobile Audio Players
- Laptops
- Autonomous Cars
- Servers
- Game Players
- Routers
- Flying UAVs
- GPS
- eBooks
- Tablets
- Set-top Boxes

Architecture vs. Microarchitecture

"Architecture"/Instruction Set Architecture:

- Programmer visible state (Memory & Register)
- Operations (Instructions and how they work)
- Execution Semantics (interrupts)
- Input/Output
- Data Types/Sizes

Microarchitecture/Organization:

- Tradeoffs on how to implement ISA for some metric (Speed, Energy, Cost)
- Examples: Pipeline depth, number of pipelines, cache size, silicon area, peak power, execution ordering, bus widths, ALU widths

Computer Architecture

Logical aspects of system implementation as seen by the programmer; such as, instruction sets (ISA) and formats, opcode, data types, addressing modes and I/O.

Instruction set architecture (ISA) is different from "microarchitecture", which consist of various processor design techniques used to implement the instruction set.

Computers with different microarchitectures can share a common instruction set.

For example, the Intel Pentium and the AMD Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs.

Computer architecture is the conceptual design and fundamental operational structure of a computer system. It is a **functional description** of requirements and design implementations for the various parts of a computer.

It is the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals.

It deals with the **architectural attributes** like physical address memory, CPU and how they should be designed and **made to coordinate with each other** keeping the goals in mind.

Analogy: "building the design and architecture of house" – architecture may take more time due to planning and then organization is building house by bricks or by latest technology keeping the basic layout and architecture of house in mind.

Computer architecture comes before computer organization.

Computer organization (CO) is how operational attributes are linked together and contribute to realise the architectural specifications.

CO encompasses all physical aspects of computer systems

e.g. Circuit design, control signals, memory types.

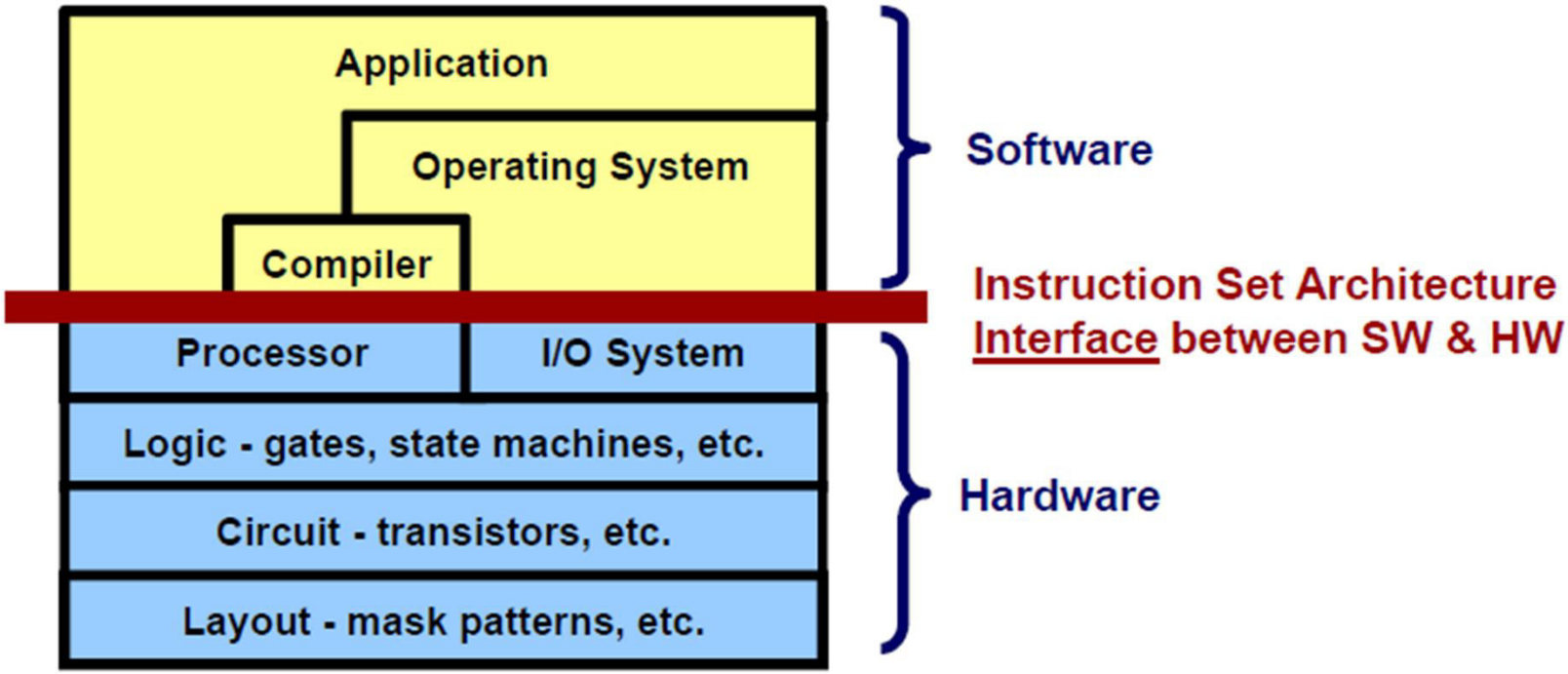
Microarchitecture, also known as **Computer organization** is a lower level, more concrete and detailed, description of the system that involves how the **constituent parts of the system are interconnected and how they interoperate** in order to implement the ISA.

The size of a computer's cache, for example, is an organizational issue that generally has nothing to do with the ISA.

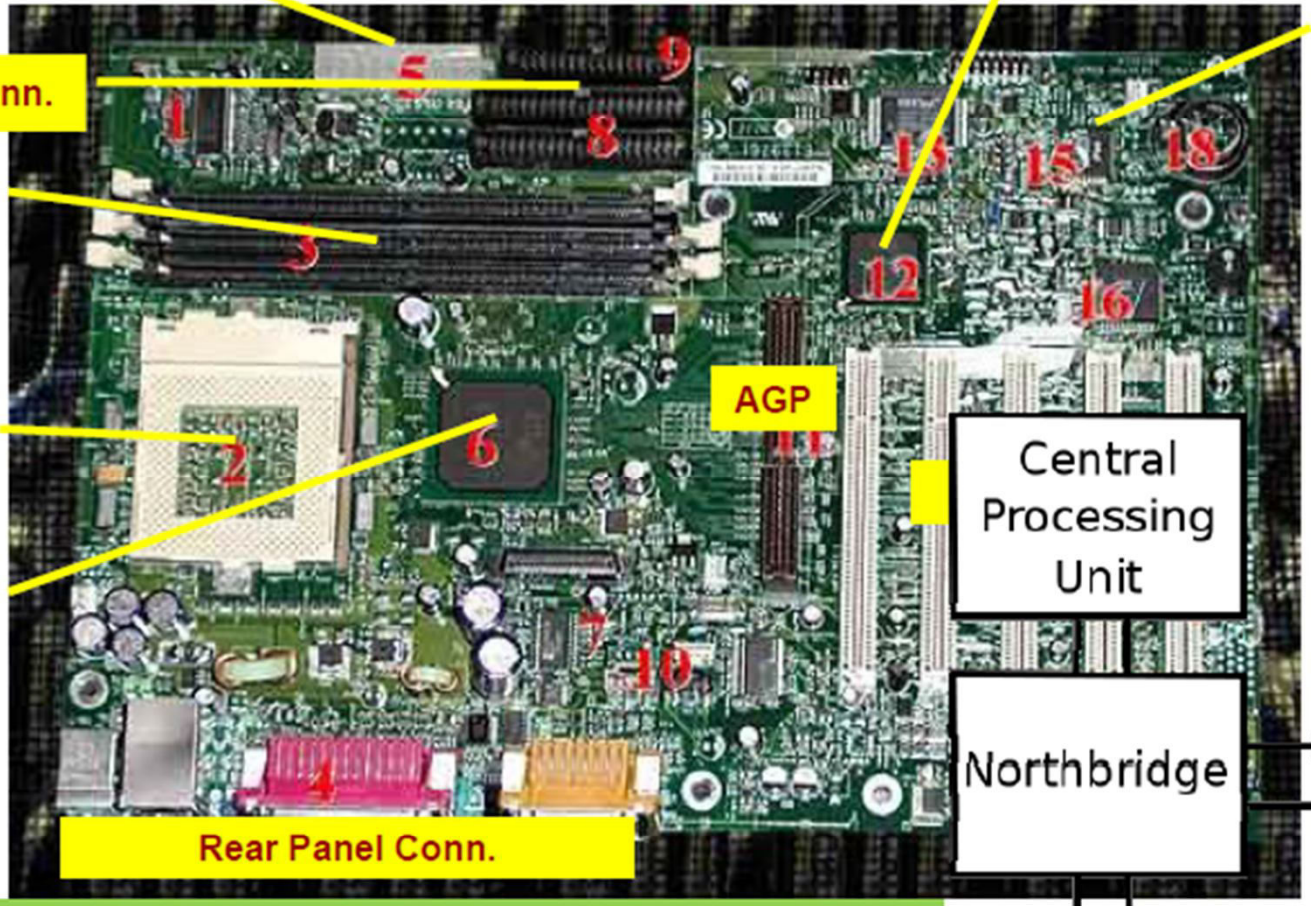
Another example: it is an architectural design issue whether a computer will have a **multiply** instruction. It is an organizational issue whether that instruction will be implemented by a special **multiply unit** or by a mechanism that makes repeated use of the **add unit** of the system.

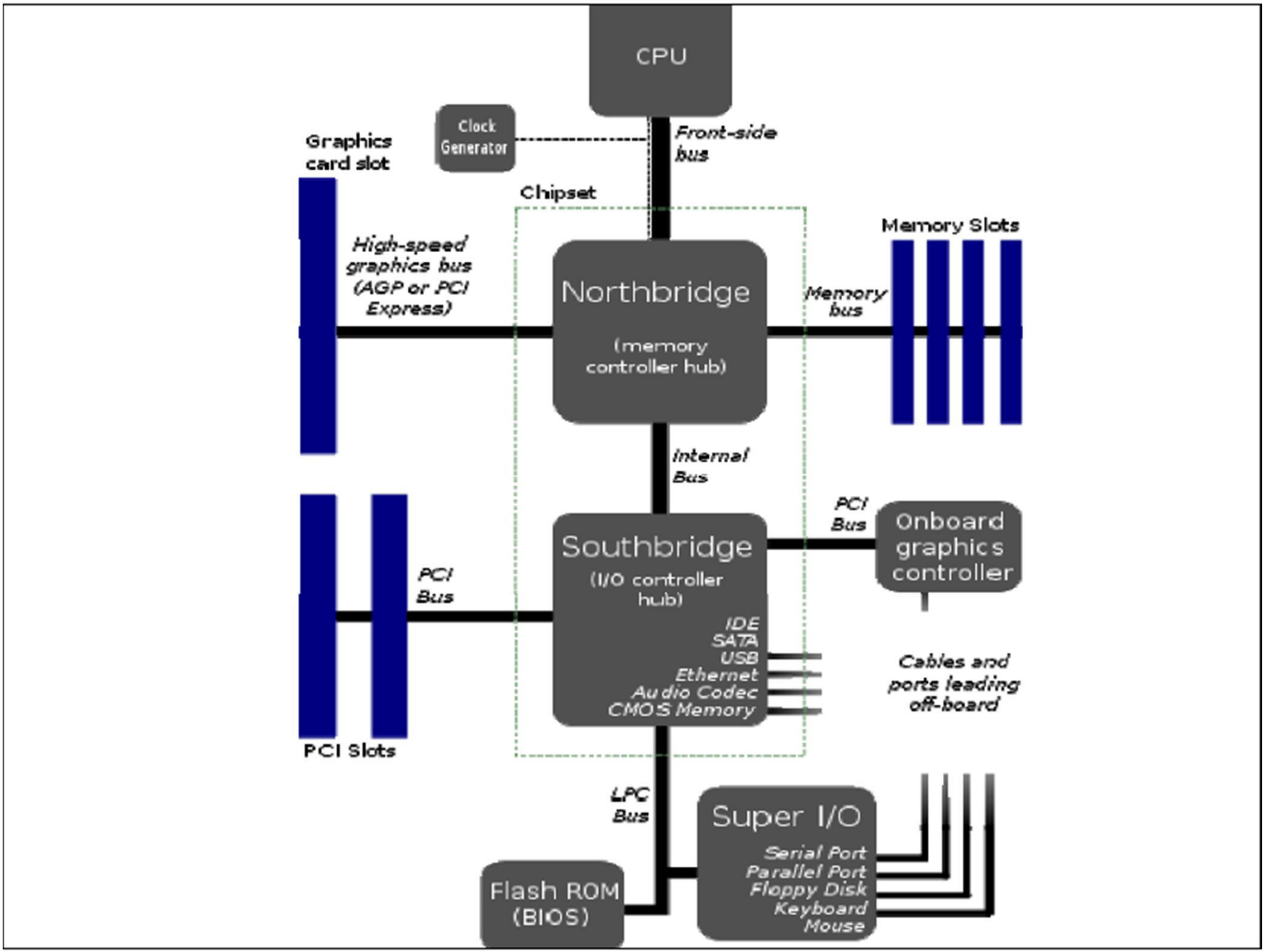
Instruction Set Architecture (ISA) - The Hardware-Software Interface

- The most important abstraction of computer design



Typical Motherboard (Pentium III)




CPU
Clock Generator
Front-side bus
Graphics card slot
Chipset
Memory Slots
High-speed graphics bus (AGP or PCI Express)
Northbridge
(memory controller hub)
Memory bus
Internal Bus
PCI Bus
Onboard graphics controller
Southbridge
(I/O controller hub)
PCI Bus
IDE
SATA
USB
Ethernet
Audio Codec
CMOS Memory
Cables and ports leading off-board
PCI Slots
LPC Bus
Super I/O
Serial Port
Parallel Port
Floppy Disk
Keyboard
Mouse
Flash ROM (BIOS)

What is Computer Architecture?

Figure 1: Courtesy: www.psychology-today.com

Computer Architecture

- The CPU is the brain of a computer system.
- It works both consciously and subconsciously.
- Consciously : Executes a program
- Sub-consciously : Runs the operating system, co-ordinates with I/O devices

Computer Architecture : Study of the CPU and the peripherals

Where does it fit in?

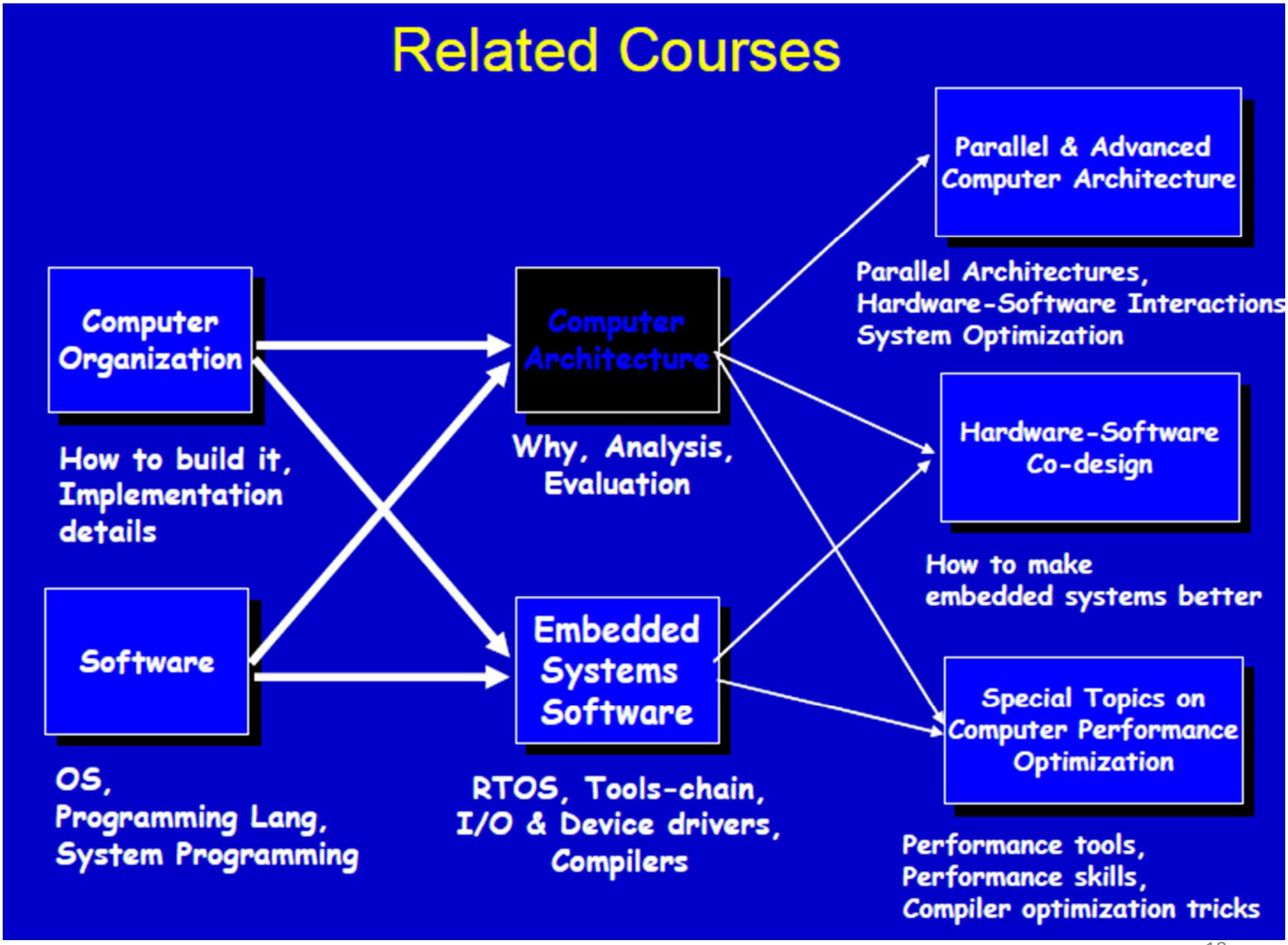
Figure 2: courtesy: www.coolnerds.com

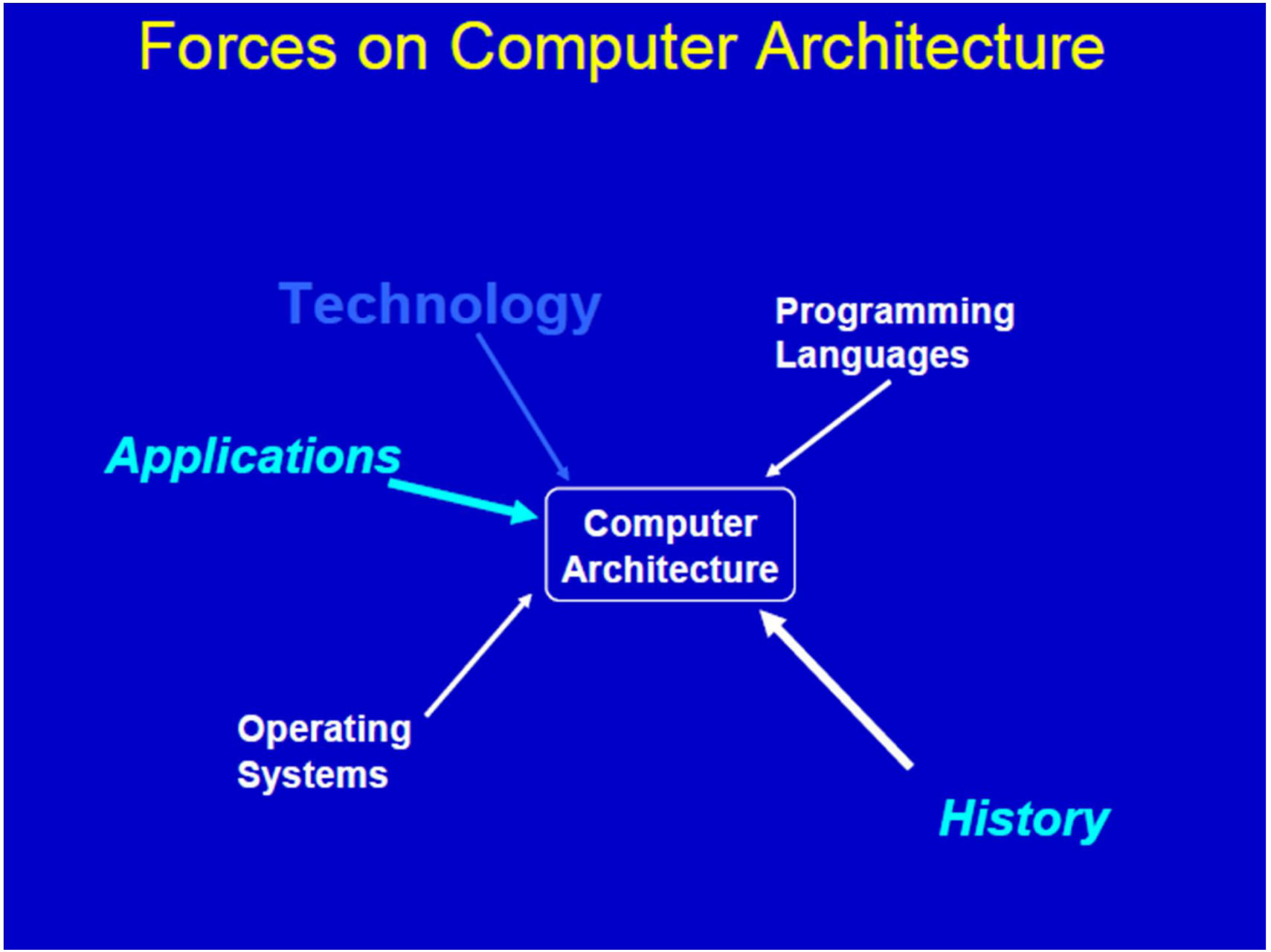
Example

- Computer Architecture –> Brain
- Networking –> Nervous and Circulatory System
- Computer Vision –> Eyes
- Operating System–> Endocrine and Immune System
- Databases –> Memory
- Algorithms –> Intelligence
- Prog. Languages –> Linguistic Center
- ...

Why Study Computer Architecture?

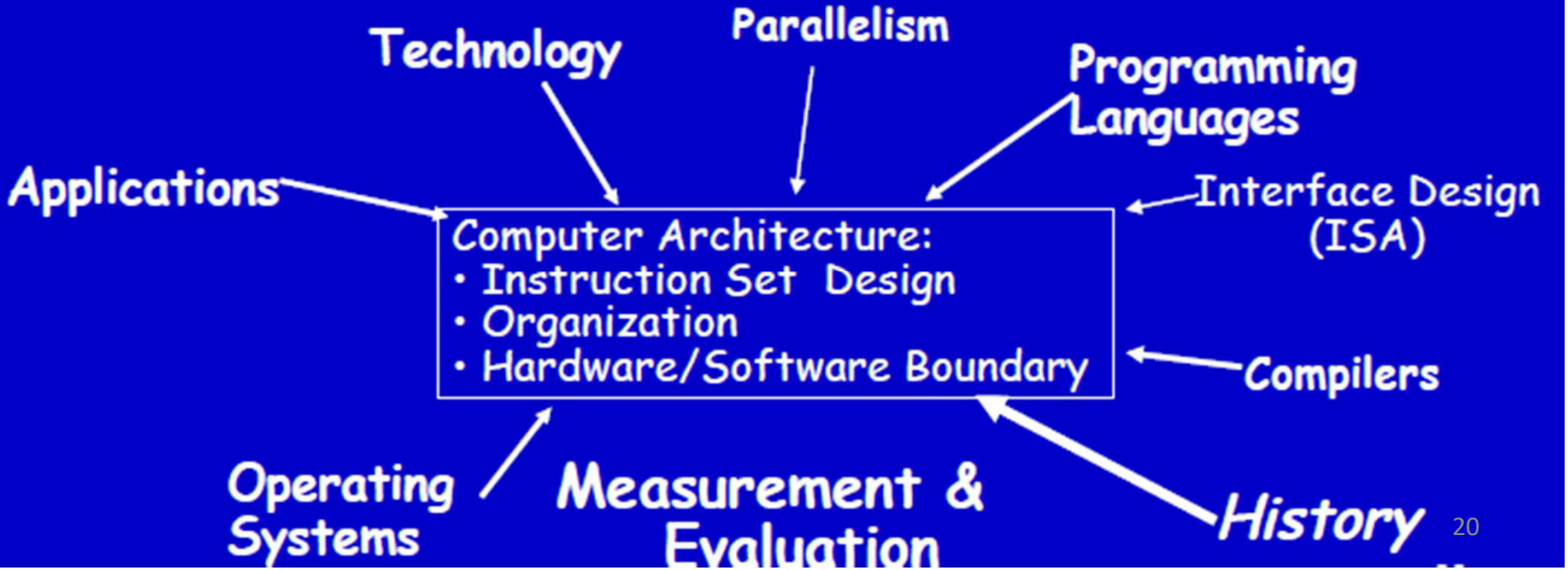
- Understanding
  - Learn the inner workings of processors
  - Understand hardware/software interaction
  - Design better operating systems and compilers
- Career Prospects
  - Companies directly working in architecture
    - Intel, AMD, Sun/Oracle, Arm, IBM
  - Systems Software
    - Google, Samsung, VMWare, Wind River, McAfee
- Higher Studies ...


Related Courses
Computer Organization
How to build it, Implementation details
Software
OS, Programming Lang, System Programming
Computer Architecture
Why, Analysis, Evaluation
Embedded Systems Software
RTOS, Tools-chain, I/O & Device drivers, Compilers
Parallel & Advanced Computer Architecture
Parallel Architectures, Hardware-Software Interactions System Optimization
Hardware-Software Co-design
How to make embedded systems better
Special Topics on Computer Performance Optimization
Performance tools, Performance skills, Compiler optimization tricks


Forces on Computer Architecture
Technology
Programming Languages
Applications
Computer Architecture
Operating Systems
History

Course Focus

Understanding the design techniques, machine structures, technology factors, evaluation methods that will determine the form of computers in 21st Century



Input/ Output Unit Overview

- Input units
  - Keyboard, mouse, microphone, CDROM, etc.
- Output units
  - Graphical display, printer, etc.
- The collective term input/ output (I/O) units
  - Input units, output units, disk drives, etc.

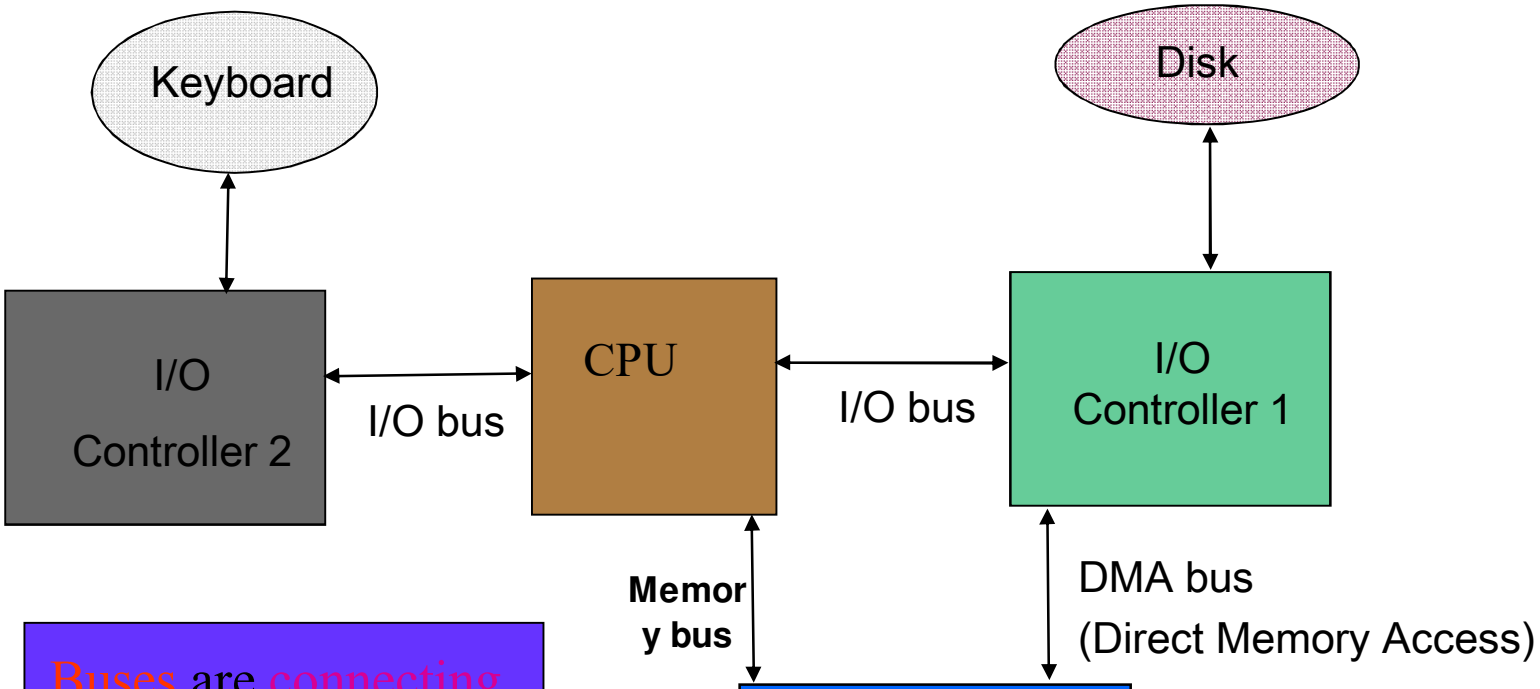
Memory Unit Overview

- Memory is used to store **programs** and **data**
- [Low-level] Unit of access is an **n**-bit word
  - Unique location is its address
  - Retrieval is in units of words
  - Commonly 32-bit today, moving to 64-bits
  - Typically 16-bit – 64-bit machines nowadays
- Primary storage: random-access memory (RAM)
- Secondary storage: hard disk, CDROM, etc.

Processor Overview

- Registers
  - Small but fast storage of intermediate values in a computation
- Arithmetic logic unit (ALU): performs computations
  - e.g. arithmetic operations: add, subtract, multiply, divide, etc.
  - e.g. logical operations: and, or, not, xor, etc.
  - c.f. calculator
  - Operands taken from registers
- Control
  - Orchestrates the transfer of data and sequencing of operations between memory, registers, ALU, I/O devices

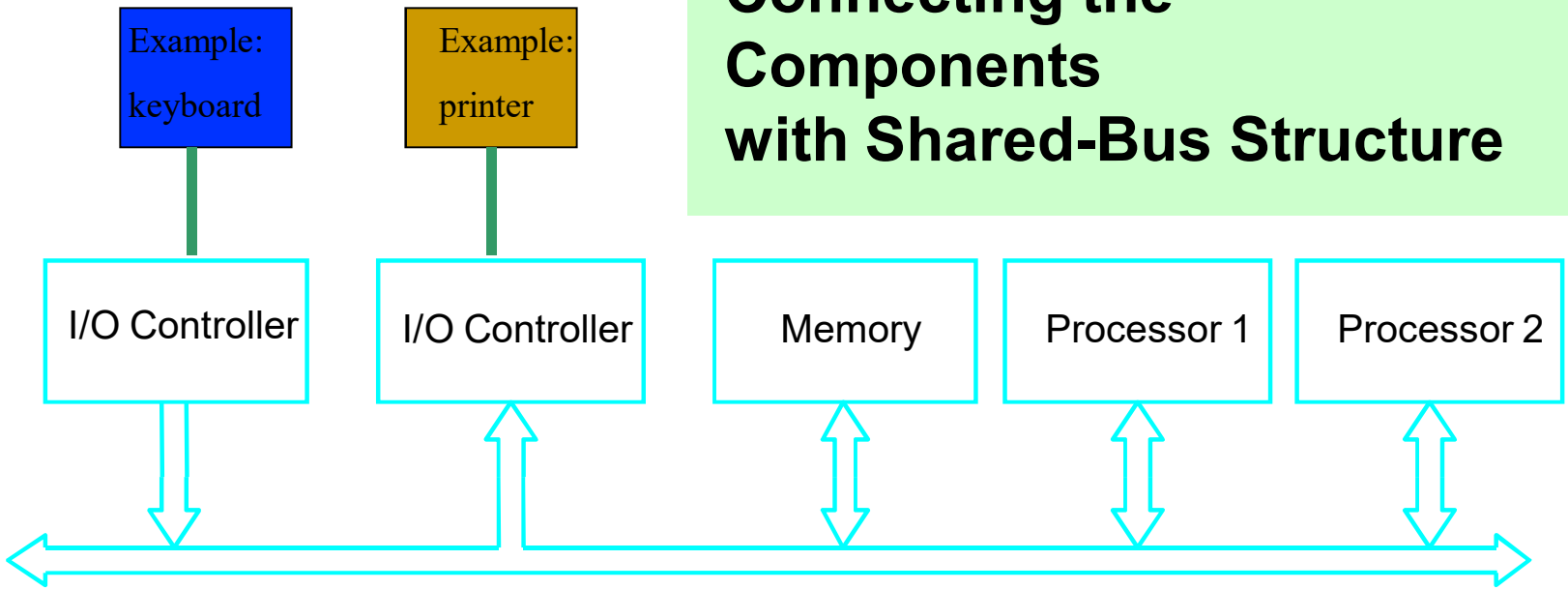
Connecting the Components with Dedicated/ Multiple Buses



Buses are connecting paths between **components in a computer**, and **even between computers**.

http://en.wikipedia.org/wiki/Computer_bus

Connecting the Components with Shared-Bus Structure



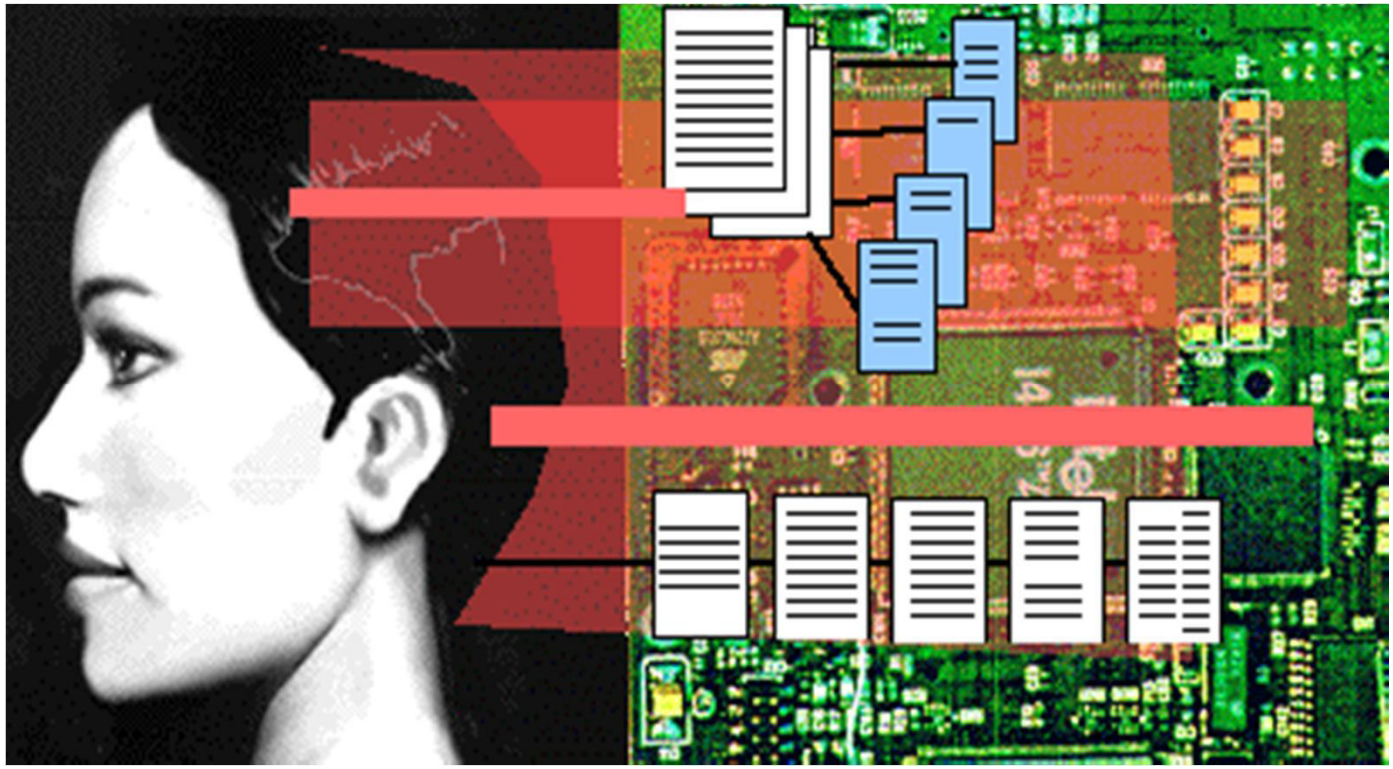
Information carried along a bus: address, data , control

- **All devices have same address structure**
- **All devices can be controlled by common machine instructions**
- **Only two devices can do data communication simultaneously**

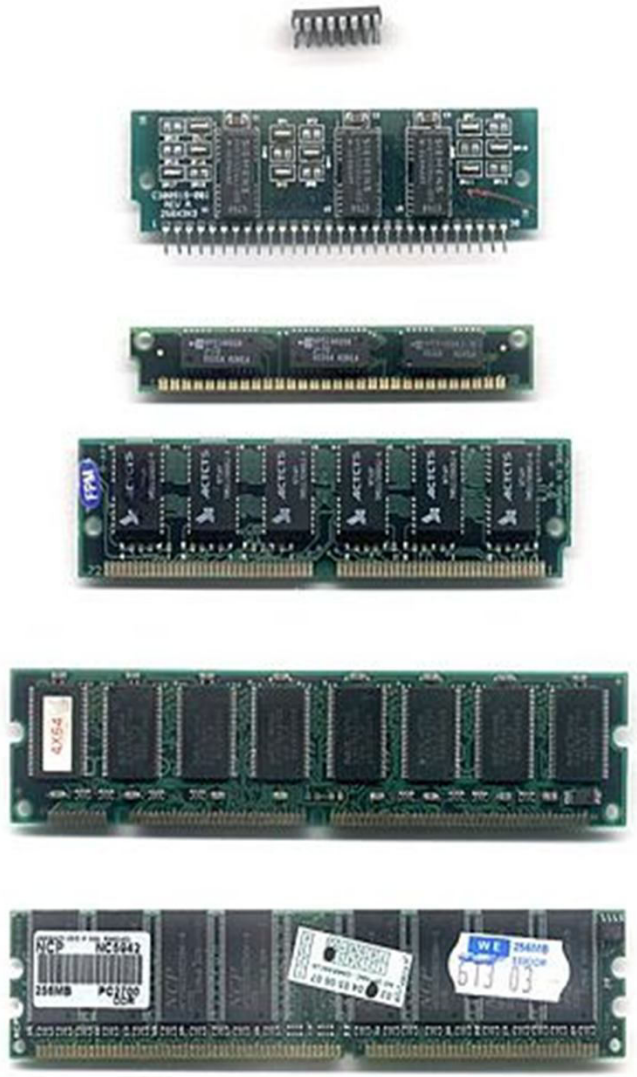
Bus

- Allows computer to be customized for different applications e.g. different peripherals
- Design criteria – speed, cost, etc.
- Word length can be different depending on application
  - E.g. USB is serial (1-bit), PCI bus is 32-bit

Computer Memory



Memory Hardware: Chips and Modules



Digital (Binary) Memory

- **Primary storage, main storage, main memory**
- **Fast Access (when compared to I/O)**

Units

bit (1 binary digit, a value of 0 or 1)

Byte (1 byte = 8 bits)

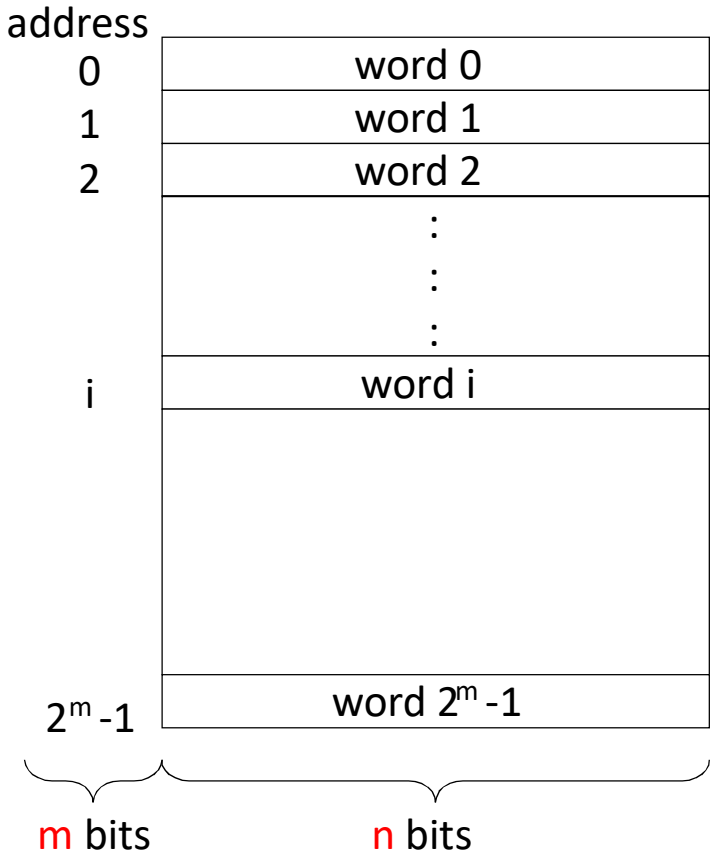
Word

Manipulation of data by CPU is in words.
A single access results in one word of data being transferred.

Word length is specified in number of bits/ bytes:
for example, 8-bit, 16-bit, 32-bit, 64-bit, 4-byte, etc.

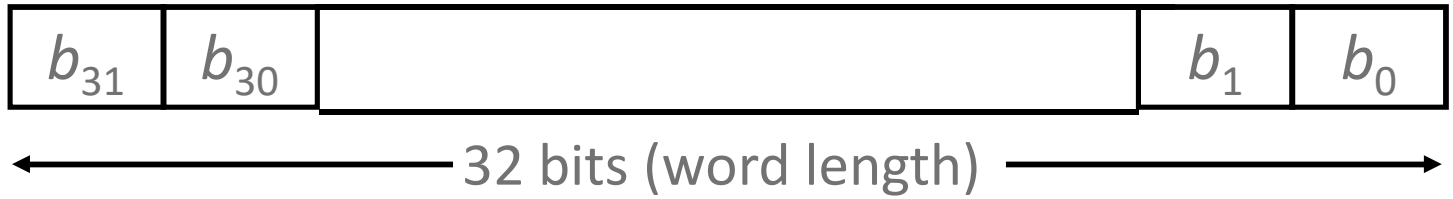
Main Memory (MM) Organization

- In a main memory with m-bit addresses, 0 to 2^m-1 words are available.
- Each word stores n bits
- m, n are independent
- m specifies the number of units; n specifies the unit size
- What is the total number of bits?



Memory: Contents of a Word (I)

Here is an example of a 32-bit word.



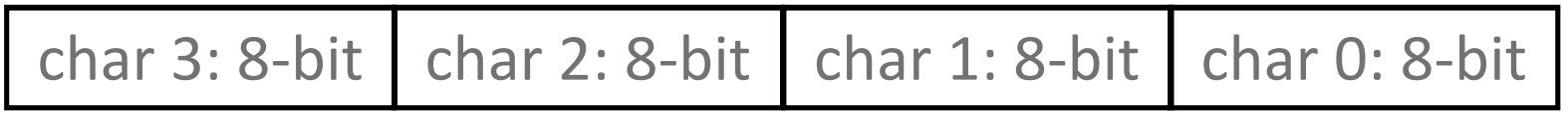
A word can store information. For example,

1. Four English characters, each encoded in a common 8-bit code

ASCII: American Standard Code for Information Interchange

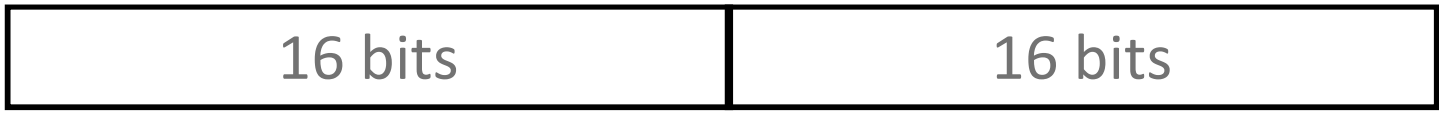
or

EBCDIC: Extended Binary Coded Decimal Interchange Code



Memory: Contents of a Word (II)

2. Two Chinese characters



Big5/GB code

3. A 32-bit Signed integer

sign bit

b_{31}=0 means positive integer, b_{31}=1 means negative integer

magnitude = $b_{30} \cdot 2^{30} + \ldots + b_0 \cdot 2^0$

4. A 32-bit machine instruction

operation code

address field

address field

Addresses

- Use addresses to store or retrieve a single item of information
- For some k, memory consists of 2^k unique addresses which range from 0 – (2^k-1)
  - The ***possible*** addresses are the address space of the computer
  - e.g. 28-bit address → 2^{28} = 268435456 locations
    - Talk about word address: we use words
    - Talk about byte address: we use bytes
    - Talk about memory size: we use bytes/ sometimes bits

Byte addresses

- Information quantities: bit, byte, word
- 1 Byte = 8 bits
- Word typically varies 16-64 bits (the IA-32 architecture has 32-bit words which we will assume from now on)

  Intel Architecture, 32-bit
- Most machines address memory in units of bytes (byte-addressable)
  - Implies for a 32-bit machine, successive words are at addresses 0, 4, 8, 12 …

More/ Less Significant Bytes

- Consider the hexadecimal (base 16) 32-bit number

  $12342A3F_{16}$

  $=1x16^7+2x16^6+3x16^5+4x16^4+2x16^3+10x16^2+3x16^1+15x16^0$

  $=305408575_{10}$

- This 32-bit number has four bytes 12, 34, 2A, 4F
  (4 bytes x 8 bits/byte = 32 bits)

- Bytes/bits with higher weighting are “ more significant”
  - e.g. the byte 34 is more significant than 2A

- Bytes/bits with lower weighting are “ less significant”

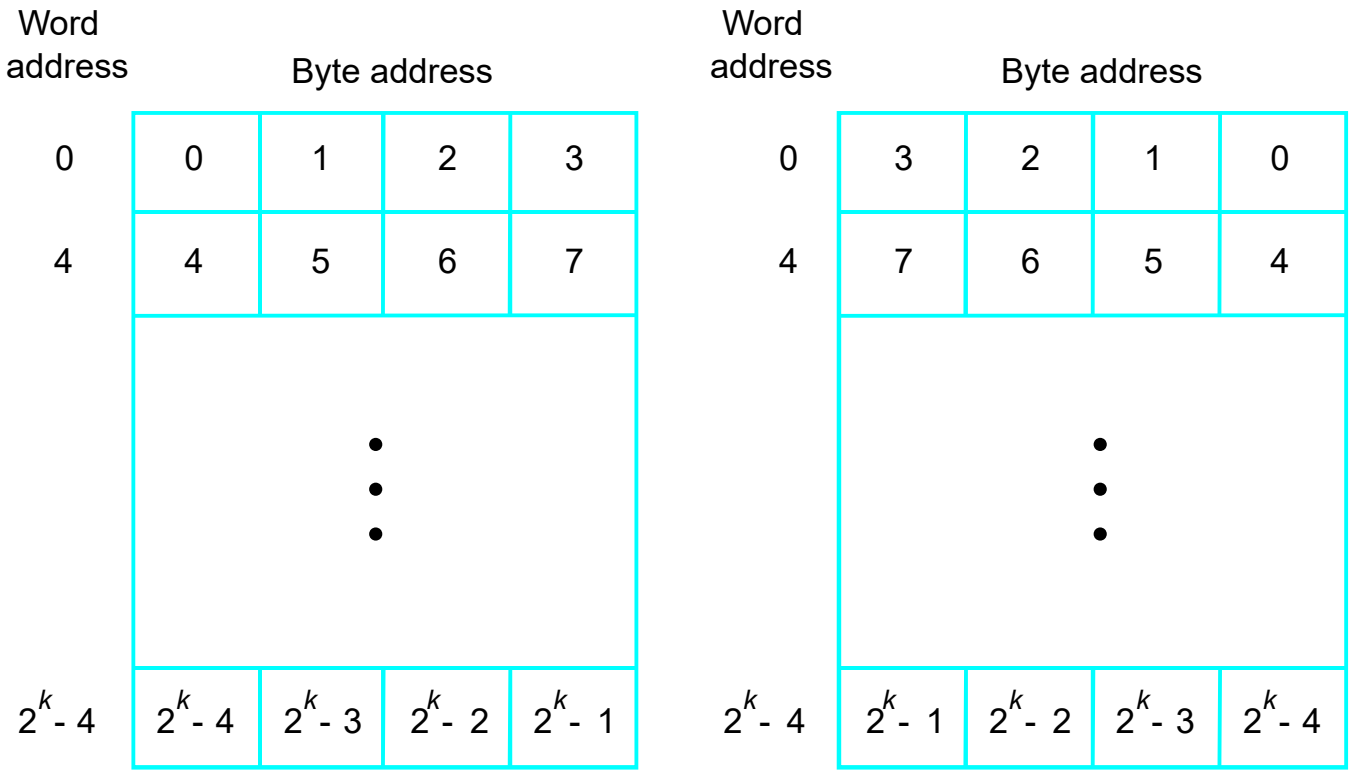
- We also use terms “ most significant byte/ bit” and “ least significant byte/ bit”

Big/ Little Endian

- Two ways byte addresses can be assigned/ arranged across words
  - more significant bytes first (big endian)
    - e.g. Motorola
  - less significant bytes first (little endian)
    - e.g. Intel

Big/ Little Endian

What would $12342A3F_{16}$ look like in these two endianship assignments?



(a) Big-endian assignment
e.g. Motorola

(b) Little-endian assignment
e.g. Intel

Assume k-bit address, 4-byte (32-bit) word

Word alignment

- 32-bit words ***align*** naturally at addresses 0, 4, 8 …
  - These are aligned addresses

- Unaligned accesses are either not allowed or slower, why?
  - e.g. read a 32-bit word from address 00000001

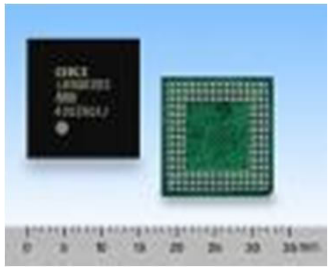
- What about for 16- and 64-bit words?

Central Processing Unit (CPU)

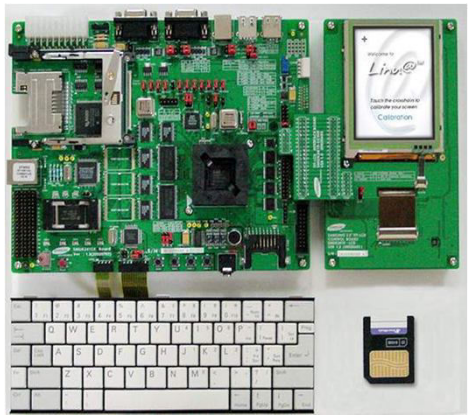
IBM Power PC

Intel Pentium

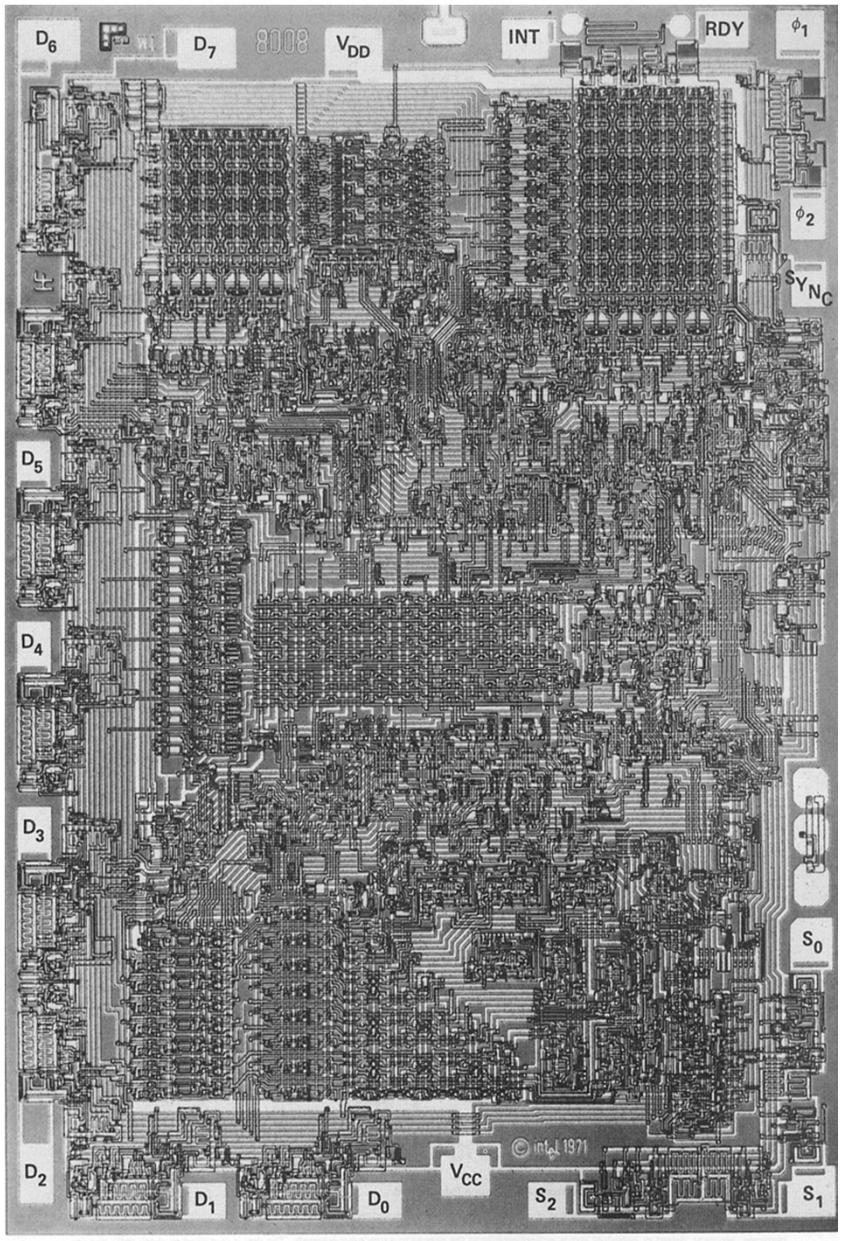
Digital signal processor IC



ARM 9

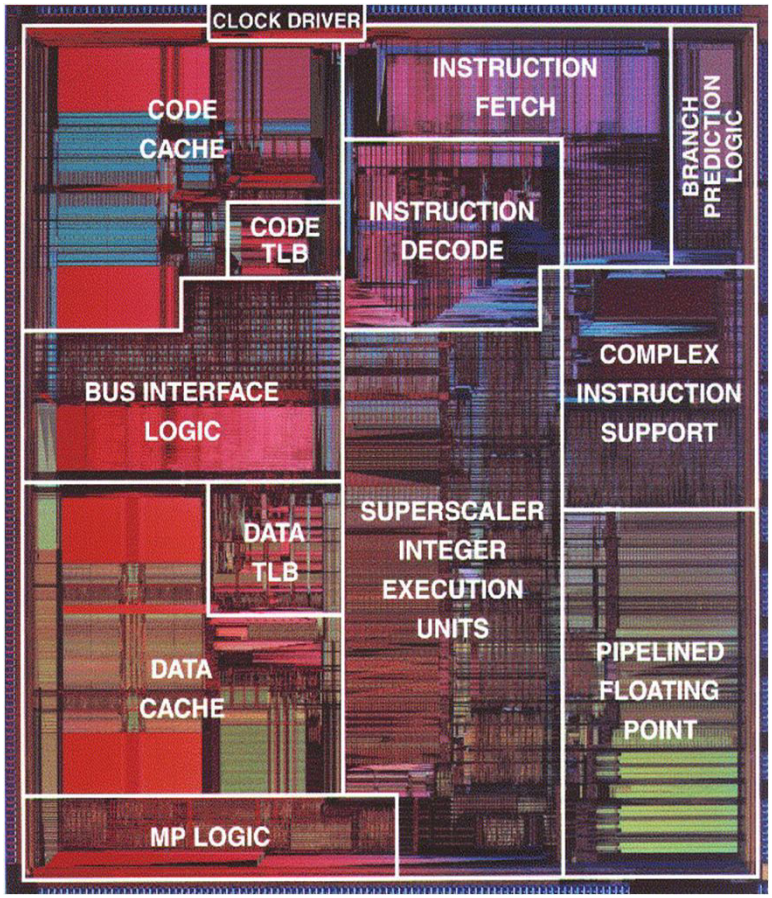


Some Intel CPUs
4004
8008
8080
8088/ 8086 [x86]
80186
80286 [286]
80386 [386]
80486 [486]
Pentium [586]
Pentium MMX
Pentium PRO [686]
Pentium II
Pentium III
Pentium 4
Core2 Duo



8008 Photomicrograph With Pin Designations

CPU on a Chip → Microprocessor



http://micro.magnet.fsu.edu/chipshots/index.html

CPU

- What does a CPU do?

  It executes programs.

- What is a program?

  Program = Instruction + Data

- Where are the instructions and data?

  In Memory – when they are not being processed

  In CPU – when they are being processed

Reduced Instruction Set Computer (RISC)

- Dave Patterson
  - RISC Project, 1982
  - UC Berkeley
  - RISC-I: ½ transistors & 3x faster
  - Influences: Sun SPARC, namesake of industry

- John L. Hennessy
  - MIPS, 1981
  - Stanford
  - Simple pipelining, keep full
  - Influences: MIPS computer system, PlayStation, Nintendo

Reduced Instruction Set Computer (RISC)

- MIPS Design Principles
- Simplicity favors regularity
  - 32 bit instructions
- Smaller is faster
  - Small register file
- Make the common case fast
  - Include support for constants
- Good design demands good compromises
  - Support for different type of interpretations/classes

Reduced Instruction Set Computer

- MIPS = Reduced Instruction Set Computer (RISC)
  - ≈ 200 instructions, 32 bits each, 3 formats
  - all operands in registers
    - almost all are 32 bits each
  - ≈ 1 addressing mode: Mem[reg + imm]

- x86 = Complex Instruction Set Computer (CISC)
  - > 1000 instructions, 1 to 15 bytes each
  - operands in dedicated registers, general purpose registers, memory, on stack, …
    - can be 1, 2, 4, 8 bytes, signed or unsigned
  - 10s of addressing modes
    - e.g. Mem[segment + reg + reg*scale + offset]

RISC vs CISC

- RISC Philosophy
- Regularity & simplicity
- Leaner means faster
- Optimize the common case

CISC Rebuttal

Compilers can be smart

Transistors are plentiful

Legacy is important

Code size counts

Micro-code!

- Energy efficiency
- Embedded Systems
- Phones/Tablets

Desktops/Servers

Role of Performance

Introduction

- Performance dictates the effectiveness of an entire system, including hardware and software
- Performance measurement is one of the most important and difficult problems in computers
  - Consider the code to initialize a million integers using a loop vs using a system call
- Different aspects of performance may require different performance metrics
- Our goal for understanding performance
  - Effect of software on performance (see the above example)
  - Effect of instruction set architecture
  - Hardware features
- Defining performance
  - Needs and desires, buying a car
  - Response time
  - Execution time
  - Clock time is dependent on computer load, I/O wait, and OS overhead
  - Throughput
  - For our purpose,
    $$\text{Performance} = \frac{c}{\text{Execution time}}$$
    where c is a constant
  - For two machines, performance (p_i) and execution time (e_i) obey the relation
    $$\frac{p_i}{p_j} = \frac{e_j}{e_i} = n$$
    and we say that machine i is n times faster than machine j

Measuring performance

- Amount of work and amount of time
- Simplest time definition is the real clock time
  - System time, user time, I/O time, overhead
- *System performance* – Elapsed time on unloaded system
- CPU *performance* – CPU time
- *Clock cycles*
  - Constant time interval for the clock within the system
  - Dictates how fast a CPU can execute each instruction
- Clock rate
  - Inverse of clock cycle

  - 500 MHz
  - Clock cycle for 500 MHz is 2ns

Performance metrics

- CPU execution time is given by the product of CPU clock cycles for program and clock cycle time
- It can also be measured by
$$\frac{\text{CPU clock cycles for program}}{\text{Clock rate}}$$
- Improving performance
  - Current system
    * Execution time – 10 sec
    * Clock speed – 400 MHz
  - New system
    * Execution time – 6 sec
    * Clock speed – ?
    * Number of clock cycles – 1.2 times current system
  - Compute the number of clock cycles for current system
    *
$$\begin{aligned} \text{CPU time} &= \frac{\text{CPU clock cycles for program}}{\text{Clock rate}} \\ 10sec &= \frac{\text{CPU clock cycles for program}}{400 \times 10^6 \text{cps}} \end{aligned}$$
    * CPU clock cycles for program $= 4000 \times 10^6$
  - Compute the clock speed for new system
    *
$$\begin{aligned} \text{CPU time} &= \frac{\text{CPU clock cycles for program}}{\text{Clock rate}} \\ 6sec &= \frac{1.2 \times 4000 \times 10^6}{\text{Clock rate}} \end{aligned}$$
    *
$$\begin{aligned} \text{Clock rate} &= \frac{1.2 \times 4000 \times 10^6}{6} \\ &= 800 \times 10^6 \\ &= 800\text{MHz} \end{aligned}$$
- *Clock cycles per instruction*, or CPI
  - Average number of cycles for all instructions for the program being executed
  - CPU clock cycles is given by the product of number of instrcutions and CPI
- Using performance equation
  - Two implementations of the same ISA – machines M_a and M_b
  - M_a clock cycle time 1ns and CPI 2.0 for some code p
  - M_b clock cycle time 2ns and CPI 1.2 for p

  - Identify faster machine
    * Let total clock cycles for the program on respective machines be c_a and c_b, and number of instructions be I
    *
$$\begin{aligned} c_a &= T \times 2.0 \\ c_b &= T \times 1.2 \end{aligned}$$
    * CPU time t = CPU clock cycles × Clock cycle time
    * $t_a = I \times 2.0 \times 1 = 2I$ ns
    * $t_b = I \times 1.2 \times 2 = 2.4I$ ns
    * Machine m_a is faster; since performance is inversely proportional to time, the performance gain is given by
$$\frac{t_b}{t_a} = \frac{2.4}{2} = 1.2$$

- Basic performance equation

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

  or

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

- Measuring the performance factors
  - Measure CPU time by actually running the program
  - Clock cycle time is usually available as part of documentation
  - Instruction count and CPI are more difficult to obtain
  - Instruction count can be measured by using profiling tools, for example, `gprof(1)` in Unix

```
$ gcc -pg -o foobar foobar.c
$ foobar
$ gprof > foobar.profile
```

  - CPI can be obtained by detailed simulation of an implementation or by combining hardware counters and simulation
  - You may be able to compute CPU clock cycles by looking at different types of instructions and using their individual clock cycle counts

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times C_i)$$

    * C_i is the number of instructions of class i
    * CPI_i is the average number of cycles per instruction for class i
    * n is the number of instruction classes

- Comparing code segments – deciding ow to write efficient code for a given machine by selecting a set of instructions
  - Instruction classes

Instruction class	CPI
A	1
B	2
C	3

  - Instruction count for different code sequences

Code sequence	Number of instructions		
	A	B	C
c_1	2	1	2
c_2	4	1	1

  - Find out the number of instructions for each code sequence, the faster code sequence, and CPI for each code sequence
    * Number of instructions in sequence $c_1 = 2 + 1 + 2 = 5$
    * Number of instructions in sequence $c_2 = 4 + 1 + 1 = 6$
    * Obviously, sequence c_1 executes fewer instructions
    * CPU clock cycles$_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10$
    * CPU clock cycles$_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9$
    * Code sequence c_2 is faster
    * CPI $= \frac{\text{CPU clock cycles}}{\text{Instruction count}}$
    * CPI$_1 = \frac{10}{5} = 2$
    * CPI$_2 = \frac{9}{6} = 1.5$

Benchmarks for performance evaluation

- *Workload*
  - Typical set of programs run in day-to-day work
  - Compare the execution time of workload on two computers to evaluate their relative performance
  - Not always feasible for real world
    * Too expensive (taking machines to prospective buyers' sites)
    * Proprietory issues (sending code and data to vendor sites)
- Benchmarks
  - Programs specifically chosen to simulate the actual workload performance
  - Selection of programs based on expected usage environment
  - Compiler optimization to beat benchmarks
    * Compiler may beat the benchmark but not guaranteed to produce correct working code at similar performance level
    * Code optimization to beat benchmark, especially if the benchmark is skewed towards some code
  - Benchmarks are used for
    * Easy coding and simulation
    * Simplicity
    * More easily standardized than large code
- Reproducibility
  - Most important component of a benchmark
  - Contains everything required to simulate a benchmark

Comparing and summarizing performance

- Summarizing implies loss of information but ease of understanding

  - Should not cause confusion with contradictory but true statements
    * *Machine A is 10 times faster than machine B for program 1*
    * *Machine B is 10 times faster than machine A for program 2*

- Total execution time
  - Compare total execution time of a set of programs taken together
  - If P_i is performance of machine i and E_i is execution time of machine i, then,

$$\frac{P_a}{P_b} = \frac{E_b}{E_a} = \frac{1001}{110} = 9.1$$

- Average execution time
  - Computed over a number of small benchmarks
  - Arithmetic mean $AM = \frac{1}{n}\sum_{i=1}^{n} E_i$
  - Smaller mean implies smaller execution time
- Weighted average execution time
  - Applies a weight to each task such that sum of all weights w_i is 1
  - Weighted arithmetic mean $WAM = \frac{1}{n}\sum_{i=1}^{n} w_i \times E_i$, with the condition that $\sum_{i=1}^{n} w_i = 1$

SPEC95 Benchmark

- SPEC – *System Performance Evaluation Cooperative*
- Most comprehensive and popular set of CPU benchmarks
- 8 integer programs written in C and 10 floating point programs written in Fortran 77
- Separate time measurement for each set
  - Measurement normalized by dividing the execution time of a Sun SPARCstation 10/40 by the execution time on measured machine, yielding SPEC ratio
  - SPECint95 or SPECfp95 – Summary measurement by taking the geometric mean of the SPEC ratios
- For a given ISA, performance improvement comes from
  1. Increase in clock rate
  2. Improvements in processor organization to lower the CPI
  3. Compiler enhancements to lower the instruction count, or generate instructions with a lower average CPI
- In Figure 2.7, we see that Pentium Pro is 1.4 to 1.5 times faster on SPECint95 and 1.6 to 1.7 times faster on SPECfp95, *at the same clock rate*
- Increasing clock speed (Figure 2.8) does not increase the SPEC performance by the same level because of memory speed bottleneck

Fallacies and pitfalls

Pitfall 1 *Expecting the improvement of one aspect of a machine to increase performance by an amount proportional to the size of the improvement.*

- A program runs in 100 sec on a machine, with multiply operations taking up 80 seconds of this time. How much does the speed of multiplication need to improve to get a five-fold increase in code execution?

$$\begin{aligned}\text{Execution time after improvement} &= \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}\\ \frac{100}{5} &= \frac{80}{n} + (100 - 80)\\ 20 &= \frac{80}{n} + 20\\ 0 &= \frac{80}{n}\end{aligned}$$

  There is no amount by which we can improve the performance of multiply to realize a five-fold increase in overall performance
  - This is *Amdahl's Law* in computing, or the law of diminishing returns in everyday life
  - Opportunity of improvement is affected by how many time the event occurs
- Common theme (Corollary of Amdahl's law) – make the common case fast

Fallacy 1 *Hardware-independent metrics predict performance.*

- Code size as a measure of speed
- ISA with smallest instruction set is the fastest

Pitfall 2 *Using MIPS as a performance metric.*

- MIPS $= \frac{\text{Instruction count}}{\text{Execution time}\times 10^6}$
- Intuitive, as more MIPS implies faster execution
- Problems
  1. MIPS does not account for capabilities of instructions
  2. A machine cannot have same MIPS rating for all programs
  3. MIPS can vary inversely with performance
- Consider the machine with three instruction classes and CPI measurements as follows:
  - Instruction classes

Instruction class	CPI
A	1
B	2
C	3

  - Instruction count (in billions of instructions for each class) for same program from two different compilers

Code from	Instruction count		
	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

  - Machine clock rate – 500 MHz
  - Which code sequence executes faster according to MIPS? According to execution time?

- Solution
  - Find the execution time on each compiler using the equation

$$\text{Execution time} = \frac{\text{CPU clock cycles}}{\text{Clock rate}}$$

  - If C_i is the number of instructions of class i executed

$$\text{CPU clock cycles} = \sum_{i=1}^{n}(\text{CPI}_i \times C_i)$$

$$\begin{aligned}\text{CPU clock cycles}_1 &= (5 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 10 \times 10^9\\ \text{CPU clock cycles}_2 &= (10 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 15 \times 10^9\end{aligned}$$

  - Execution time for two compilers

$$\begin{aligned}\text{Execution time}_1 &= \frac{10 \times 10^9}{500 \times 10^6} = 20s\\ \text{Execution time}_2 &= \frac{15 \times 10^9}{500 \times 10^6} = 30s\end{aligned}$$

  - MIPS rate

$$\begin{aligned}\text{MIPS} &= \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}\\ \text{MIPS}_1 &= \frac{(5+1+1) \times 10^9}{20 \times 10^6} = 350\\ \text{MIPS}_2 &= \frac{(10+1+1) \times 10^9}{30 \times 10^6} = 400\end{aligned}$$

- Conclusion – Code from compiler 1 runs faster but code from compiler 2 has higher MIPS

Fallacy 2 *Synthetic benchmarks predict performance.*

- Goal to create a benchmark where execution frequency of a synthetic benchmark matches the characteristics of a large set of programs
- Most popular synthetic benchmarks – Whetstone and Dhrystone
- Whetstone – Measurement of Algol programs in a scientific/engineering environment (converted to Fortran)
- Dhrystone – Systems programming environments, originally in Ada and later converted to C

Pitfall 3 *Using arithmetic mean of normalized execution times to predict performance.*

- Normalized arithmetic mean is dependent on the machine used for normalization
- Better way is to use geometric mean given by

$$\sqrt[n]{\prod_{i=1}^{n}\text{Execution time ratio}_i}$$

where Execution time ratio$_i$ is the execution time, normalized to the reference machine, for the ith program of a total of n in the total workload

- Geometric mean is independent of the data series used for normalization because of the property

$$\frac{\text{Geometric mean}(X_i)}{\text{Geometric mean}(Y_i)} = \text{Geometric mean}\left(\frac{X_i}{Y_i}\right)$$

  implying that mean of ratios, or ratio of means, is equal

Fallacy 3 *The geometric mean of execution time ratios is proportional to total execution time.*

- Geometric mean does not predict execution time

Performance

- Introduction
- Defining Performance
- The Iron Law of Processor Performance
- Processor performance enhancement
- Performance Evaluation Approaches
- Performance Reporting
- Amdahl's Law

Introduction

- Performance measurement is important:
  - Helps us to determine if one processir or computer works faster than other
  - Helps us to know how much performance improvement has taken place after incorproating some performance enhancement feature
  - Help to see through the marketing hype!
- Provides answer to the following questions:
  - Why is some hardware better than others for different programs?
  - What factors affect system performance?
    - Hardware, OS or compiler?
  - How does the machine's instruction set affect performance?

Defining Performance in terms of time

- Time is the final measure of computer performance
- A computer exhibits higher performance if it executes program faster
- Response Time (elapsed time, Latency):
  - How long does it take for my job to run?
  - How long does it take to execute (start to finish) my job?
  - How long must/wait for the database query?
- Throughput:
  - How many jobs can the machine run at once?
  - What is the average execution rate?
  - How much work is getting done?

Individual user concern

System Manager concern

Execution Time

- Elapsed time
  - Count everything (disk and memory access, waiting for IO, running other programs, etc) from start to finish
  - A useful number, but often not good for comparison purpose

  Elapsed time = CPU time +wait time (IO, other program, etc.)

- CPU time
  - Doesn't count waiting for IO or time spent running other programs
  - Can be divided into user CPU time and system CPU time(OS calls)

    CPU time = user CPU time + System CPU Time

  Elapsed time = user CPU time + System CPU Time + wait time

- Our focus: User CPU time
  - CPU execution time or simply execution time: time spent executing the lines of code that are in our program

Measuring performance

- for some program running on machine X:

$$\text{Performance} = \frac{1}{execution\ time_x}$$

- X is n times faster than Y means:

$$\frac{Performance_x}{Performance_y} = n$$

The IRON law of processor performance

$$Processor\ performance = \frac{Time}{Program}$$

$$= \frac{Instruction}{Program} \text{ X } \frac{Cycles}{Instruction} \text{X} \frac{Time}{Cycle}$$

(code Size) (CPI) (Cycle time)

Architecture → Implementation → Realization

Compiler Designer Processor designer Chip designer

- **Instructions/Program (Instruction count)**
  - Instructions executed, not static code size
  - Determined by algorithm, compiler, ISA
- **Cycles/Instruction (CPI)**
  - Determined by ISA and CPU organization
  - Overlap among instructions reduces this term
- **Time/cycle (Cycle time)**
  - Determined by technology, organization, clever circuit design

MIPS and MFLOPS

- Used extensively 30 years back.
- MIPS: millions of instructions processed per second.
- MFLOPS: Millions of Floating-point Operations completed per Second

$$\text{MIPS} = \frac{\text{Instruction Count}}{\text{Exec. Time} \times 10^6} = \frac{\text{Clock Rate}}{\text{CPI} \times 10^6}$$

Problems with MIPS

- Three significant problems with using MIPS:
- So severe, made some one term:
- "Meaningless Information about Processing Speed"
- Problem 1:
- MIPS is instruction set dependent.
- Problem 2:
- MIPS varies between programs on the same computer.
- Problem 3:
  - MIPS can vary inversely to performance!

❑ Consider the following computer:

The machine runs at 100MHz.

Instruction counts (in millions) for each instruction class

Code type-	A (1 cycle)	B (2 cycle)	C (3 cycle)
Compiler 1	5	1	1
Compiler 2	10	1	1

Instruction A requires 1 clock cycle, Instruction B requires 2 clock cycles, Instruction C requires 3 clock cycles.

$$CPI = \frac{\text{CPU Clock Cycles}}{\text{Instruction Count}} = \frac{\sum_{i=1}^{n} CPI_i \times N_i}{\text{Instruction Count}}$$

count cycles

$$CPI_1 = \frac{[(5x1) + (1x2) + (1x3)] \times 10^6}{(5 + 1 + 1) \times 10^6} = 10/7 = 1.43$$

$$MIPS_1 = \frac{100\ MHz}{1.43} = 69.9$$

$$CPI_2 = \frac{[(10x1) + (1x2) + (1x3)] \times 10^6}{(10 + 1 + 1) \times 10^6} = 15/12 = 1.25$$

$$MIPS_2 = \frac{100\ MHz}{1.25} = 80.0$$

So, compiler 2 has a higher MIPS rating and should be faster?

- Now let's compare CPU time:

$$\text{CPU Time} = \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

$$\text{CPU Time}_1 = \frac{7 \times 10^6 \times 1.43}{100 \times 10^6} = 0.10 \text{ seconds}$$

$$\text{CPU Time}_2 = \frac{12 \times 10^6 \times 1.25}{100 \times 10^6} = 0.15 \text{ seconds}$$

Therefore program 1 is faster despite a lower MIPS!

Computer Performance

"X is N% faster than Y."

$$\frac{\text{Execution Time of Y}}{\text{Execution Time of X}} = 1 + \frac{N}{100}$$

Amdahl's law for overall speedup

$$\text{Overall Speedup} = \frac{1}{(1-F) + \frac{F}{S}}$$

F = The fraction enhanced

S = The speedup of the enhanced fraction

Using Amdahl's law

Overall speedup if we make 90% of a program run 10 times faster.

$F = 0.9 \quad S = 10$

$$\text{Overall Speedup} = \frac{1}{(1-0.9)+\frac{0.9}{10}} = \frac{1}{0.1+0.09} = 5.26$$

Overall speedup if we make 80% of a program run 20% faster.

$F = 0.8 \quad S = 1.2$

$$\text{Overall Speedup} = \frac{1}{(1-0.8)+\frac{0.8}{1.2}} = \frac{1}{0.2+0.66} = 1.153$$

Amdahl's Law

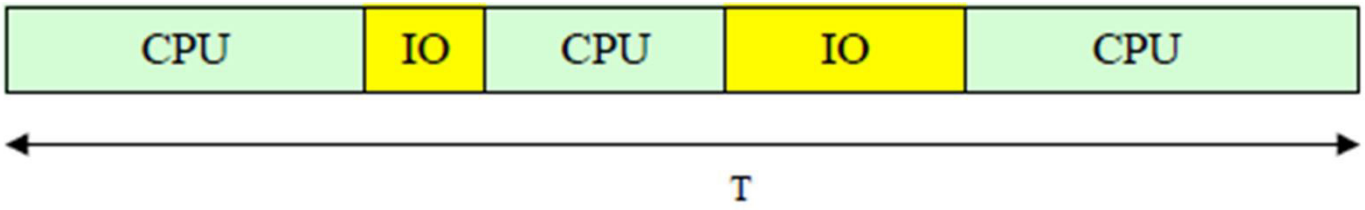
How is system performance altered when some component is changed?

Example 1:

Program execution time is made up of 75% CPU time and 25% I/O time. Which is the better enhancement:

(a) Increasing the CPU speed by 50% or (b) reducing I/O time by half?

Execution model: No overlap between CPU and I/O operations



Program execution time $T = T_{cpu} + T_{io}$

$T_{cpu} / T = 0.75$ and $T_{io} / T = 0.25$

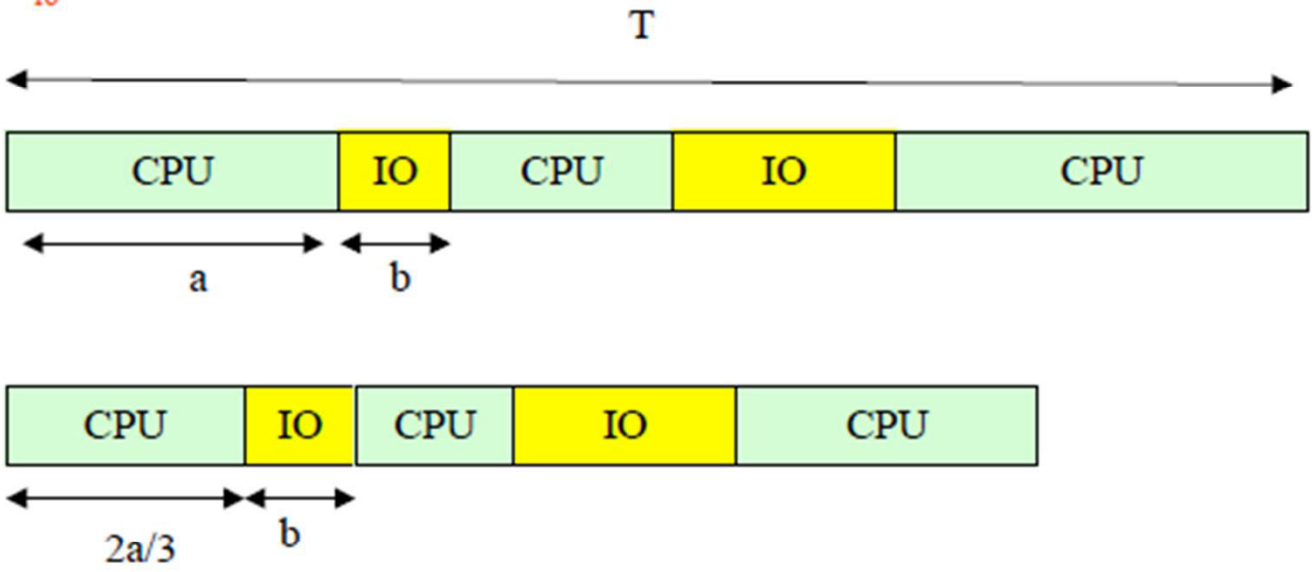
Amdahl's Law

(a) Increasing the CPU speed by 50%

Program execution time $T = T_{cpu} + T_{io}$ $T_{old} = T$

$T_{cpu} / T = 0.75$

$T_{io} / T = 0.25$



Program execution time $T_{new} = T_{cpu} / 1.5 + T_{io}$

$T_{new} = T_{cpu} / 1.5 + T_{io} = 0.75\ T / 1.5 + 0.25T = 0.75T$

For a 50% improvement in CPU speed: Execution time decreases by 25%

Speedup = $T_{old} / T_{new} = T / 0.75T = 1.33$

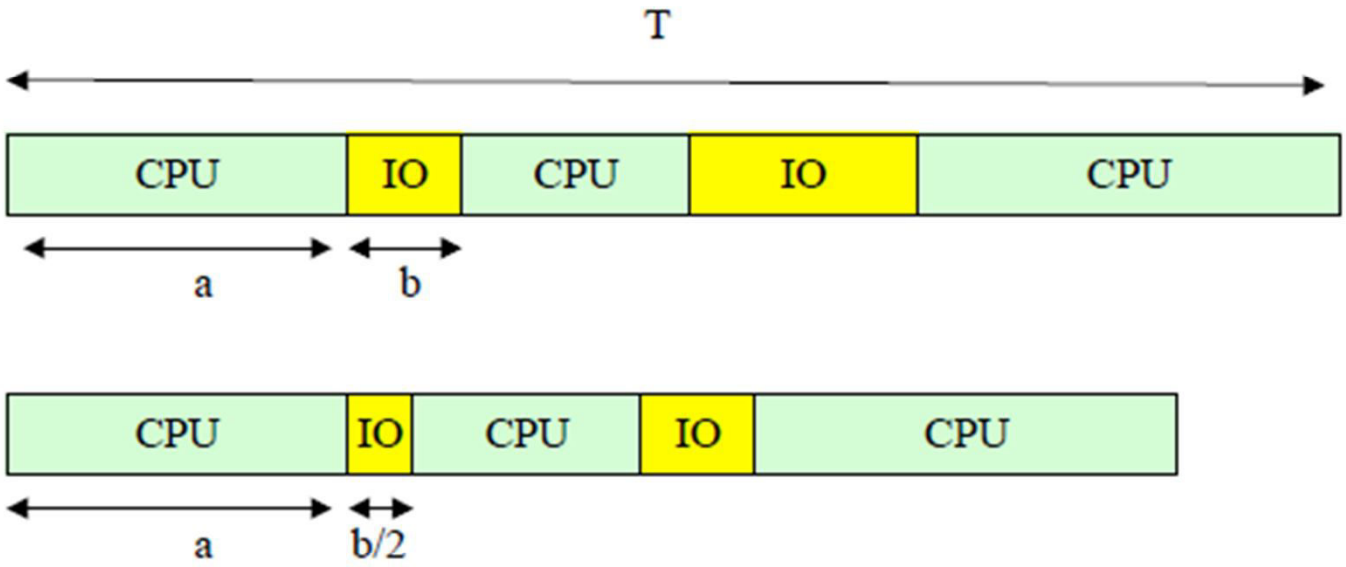
Amdahl's Law

(b) Halve the IO Time

Program execution time $T = T_{cpu} + T_{io}$ $T_{old} = T$

$T_{cpu} / T = 0.75$

$T_{io} / T = 0.25$



Program execution time $T_{new} = T_{cpu} + T_{io} / 2$

$T_{new} = 0.75\ T + 0.25T / 2 = 0.875T$

For a 100% improvement in IO speed: Execution time decreases by 12.5%

Speedup = $T_{old} / T_{new} = T / 0.875T = 1.14$

Amdahl's Law

Limiting Cases

- CPU speed improved infinitely so T_{CPU} tends to zero
  $T_{new} = T_{IO} = 0.25T$ Speedup limited to 4

- IO speed improved infinitely so T_{IO} tends to zero
  $T_{new} = T_{CPU} = 0.75T$ Speedup limited to 1.33

Problem

- Improving performance
  - Current system
    - Execution time – 10 sec
    - Clock speed – 400 MHz
  - New system
    - Execution time – 6 sec
    - Clock speed – ?
    - Number of clock cycles – 1.2 times current system

- Compute the number of clock cycles for current system
  - $$\text{CPU time} = \frac{\text{CPU clock cycles for program}}{\text{Clock rate}}$$
    $$10sec = \frac{\text{CPU clock cycles for program}}{400 \times 10^6 \text{cps}}$$
  - CPU clock cycles for program $= 4000 \times 10^6$

- Compute the clock speed for new system
  - $$\text{CPU time} = \frac{\text{CPU clock cycles for program}}{\text{Clock rate}}$$
    $$6sec = \frac{1.2 \times 4000 \times 10^6}{\text{Clock rate}}$$
  - $$\text{Clock rate} = \frac{1.2 \times 4000 \times 10^6}{6} = 800 \times 10^6 = 800\text{MHz}$$

- Basic performance equation

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

or

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

$$\text{CPU clock cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times C_i)$$

  - C_i is the number of instructions of class i
  - CPI_i is the average number of cycles per instruction for class i
  - n is the number of instruction classes

Code sequence	Number of instructions		
	A	B	C
c_1	2	1	2
c_2	4	1	1

Find out the number of instructions for each code sequence, the faster code sequence, and CPI for each code sequence

Number of instructions in sequence $c_1 = 2 + 1 + 2 = 5$

Number of instructions in sequence $c_2 = 4 + 1 + 1 = 6$

Obviously, sequence c_1 executes fewer instructions

$\text{CPU clock cycles}_1 = (2 \times 1) + (1 \times 2) + (2 \times 3) = 2 + 2 + 6 = 10$

$\text{CPU clock cycles}_2 = (4 \times 1) + (1 \times 2) + (1 \times 3) = 4 + 2 + 3 = 9$

Code sequence c_2 is faster

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

$\text{CPI}_1 = \frac{10}{5} = 2$

$\text{CPI}_2 = \frac{9}{6} = 1.5$

Bench mark sample with problem

- A program runs in 100 sec on a machine, with multiply operations taking up 80 seconds of this time. How much does the speed of multiplication need to improve to get a five-fold increase in code execution?

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

$$\frac{100}{5} = \frac{80}{n} + (100 - 80)$$

$$20 = \frac{80}{n} + 20$$

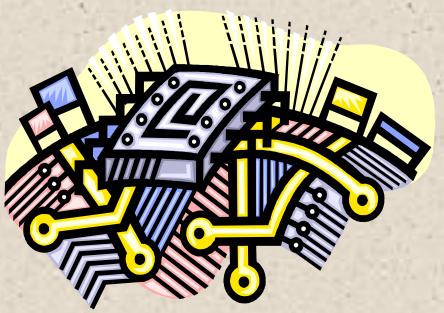
$$0 = \frac{80}{n}$$

There is no amount by which we can improve the performance of multiply to realize a five-fold increase in overall performance

  - This is *Amdahl's Law* in computing, or the law of diminishing returns in everyday life
  - Opportunity of improvement is affected by how many time the event occurs

Arithmetic & Logic Unit (ALU)

- Part of the computer that actually performs arithmetic and logical operations on data
- All of the other elements of the computer system are there mainly to bring data into the ALU for it to process and then to take the results back out
- Based on the use of simple digital logic devices that can store binary digits and perform simple Boolean logic operations



ALU Inputs and Outputs

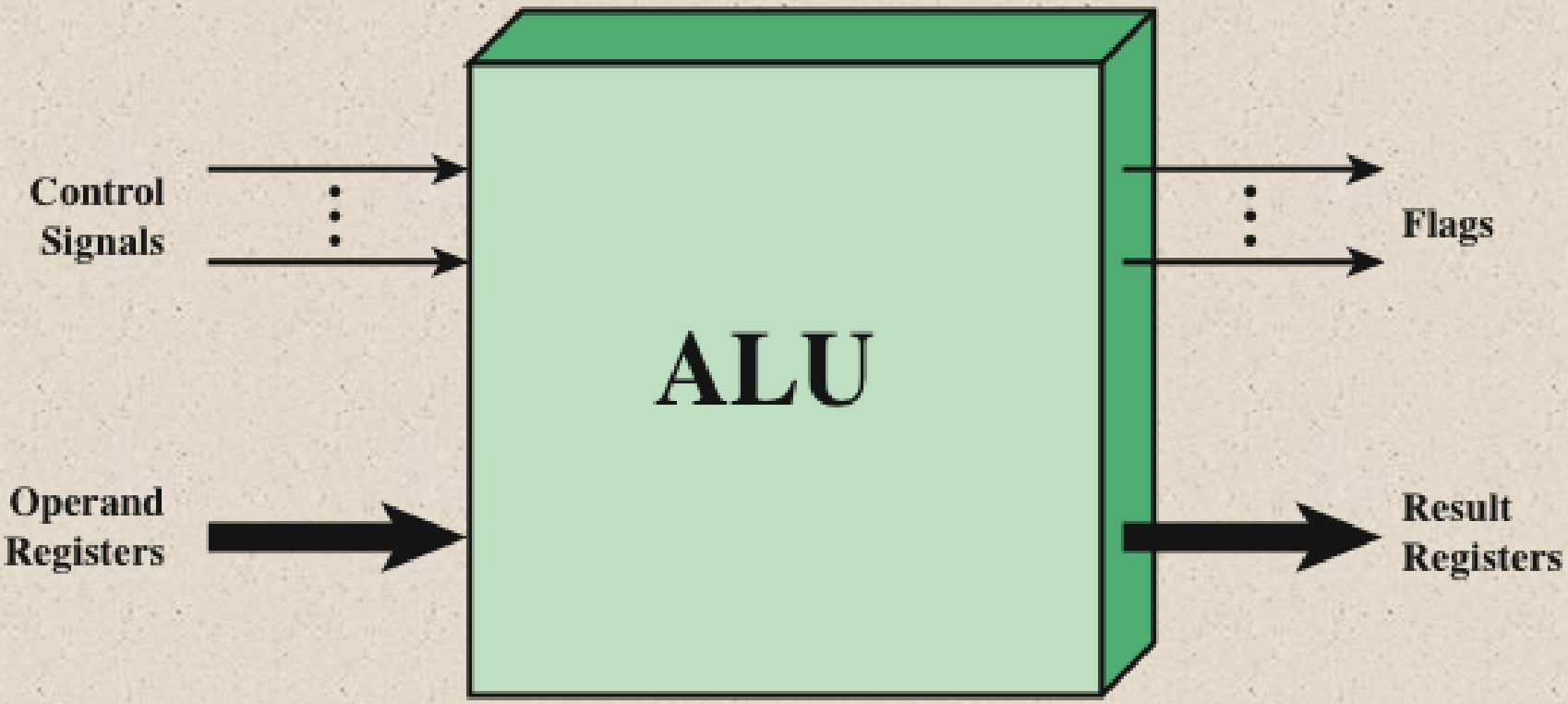


Figure 10.1 ALU Inputs and Outputs

Integer Representation

- In the binary number system arbitrary numbers can be represented with:
  - The digits zero and one
  - The minus sign (for negative numbers)
  - The period, or ***radix point*** (for numbers with a fractional component)
- For purposes of computer storage and processing we do not have the benefit of special symbols for the minus sign and radix point
- Only binary digits (0,1) may be used to represent numbers

Sign-Magnitude Representation

There are several alternative conventions used to represent negative as well as positive integers	•All of these alternatives 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
Sign-magnitude representation is the simplest form that employs a sign bit	
Drawbacks:	•Addition and subtraction require a consideration of both the signs of the numbers and their relative magnitudes to carry out the required operation •There are two representations of 0
Because of these drawbacks, sign-magnitude representation is rarely used in implementing the integer portion of the ALU	

Twos Complement Representation

- Uses the most significant bit as a sign bit
- Differs from sign-magnitude representation in the way that the other bits are interpreted

Range	-2_{n-1} through $2_{n-1} - 1$
Number of Representations of Zero	One
Negation	Take the Boolean complement of each bit of the corresponding positive number, then add 1 to the resulting bit pattern viewed as an unsigned integer.
Expansion of Bit Length	Add additional bit positions to the left and fill in with the value of the original sign bit.
Overflow Rule	If two numbers with the same sign (both positive or both negative) are added, then overflow occurs if and only if the result has the opposite sign.
Subtraction Rule	To subtract *B* from *A*, take the twos complement of *B* and add it to *A*.

Table 10.1 Characteristics of Twos Complement Representation and Arithmetic

Alternative Representations for 4-Bit Integers

Decimal Representation	Sign-Magnitude Representation	Twos Complement Representation	Biased Representation
+8	—	—	1111
+7	0111	0111	1110
+6	0110	0110	1101
+5	0101	0101	1100
+4	0100	0100	1011
+3	0011	0011	1010
+2	0010	0010	1001
+1	0001	0001	1000
+0	0000	0000	0111
–0	1000	—	—
–1	1001	1111	0110
–2	1010	1110	0101
–3	1011	1101	0100
–4	1100	1100	0011
–5	1101	1011	0010
–6	1110	1010	0001
–7	1111	1001	0000
–8	—	1000	—

Range Extension

- Range of numbers that can be expressed is extended by increasing the bit length
- In sign-magnitude notation this is accomplished by moving the sign bit to the new leftmost position and fill in with zeros
- This procedure will not work for twos complement negative integers
  - Rule is to move the sign bit to the new leftmost position and fill in with copies of the sign bit
  - For positive numbers, fill in with zeros, and for negative numbers, fill in with ones
  - This is called *sign extension*

Fixed-Point Representation

The radix point (binary point) is fixed and assumed to be to the right of the rightmost digit

Programmer can use the same representation for binary fractions by scaling the numbers so that the binary point is implicitly positioned at some other location

Negation

- Twos complement operation
  - Take the Boolean complement of each bit of the integer (including the sign bit)
  - Treating the result as an unsigned binary integer, add 1

```
                +18 = 00010010 (twos complement)
bitwise complement = 11101101
                   +        1
                    11101110 = -18
```

- The negative of the negative of that number is itself:

```
                -18 =  11101110 (twos complement)
bitwise complement =  00010001
                   +        1
                     00010010 = +18
```

Negation Special Case 1

0	=	00000000	(twos complement)
Bitwise complement	=	11111111	
Add 1 to LSB		+ 1	
Result		100000000	

Overflow is ignored, so:

- 0 = 0

Negation Special Case 2

-128 = 10000000 (twos complement)

Bitwise complement = 01111111

Add 1 to LSB + 1

Result 10000000

So:

-(-128) = -128 X

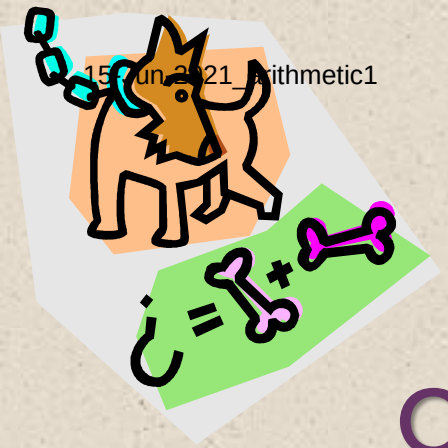
Monitor MSB (sign bit)

It should change during negation

Addition

1001 = −7 +0101 = 5 1110 = −2 (a) (−7) + (+5)	1100 = −4 +0100 = 4 10000 = 0 (b) (−4) + (+4)
0011 = 3 +0100 = 4 0111 = 7 (c) (+3) + (+4)	1100 = −4 +1111 = −1 11011 = −5 (d) (−4) + (−1)
0101 = 5 +0100 = 4 1001 = Overflow (e) (+5) + (+4)	1001 = −7 +1010 = −6 10011 = Overflow (f) (−7) + (−6)

Figure 10.3 Addition of Numbers in Twos Complement Representation



Overflow

Rule

OVERFLOW RULE:

If two numbers are added, and they are both positive or both negative, then overflow occurs if and only if the result has the opposite sign.

Subtraction

Rule

SUBTRACTION RULE:

To subtract one number (subtrahend) from another (minuend), take the twos complement (negation) of the subtrahend and add it to the minuend.

Subtraction

0010 = 2 +1001 = −7 1011 = −5	0101 = 5 +1110 = −2 10011 = 3
(a) M = 2 = 0010 S = 7 = 0111 −S = 1001	(b) M = 5 = 0101 S = 2 = 0010 −S = 1110
1011 = −5 +1110 = −2 11001 = −7	0101 = 5 +0010 = 2 0111 = 7
(c) M = −5 = 1011 S = 2 = 0010 −S = 1110	(d) M = 5 = 0101 S = −2 = 1110 −S = 0010
0111 = 7 +0111 = 7 1110 = Overflow	1010 = −6 +1100 = −4 10110 = Overflow
(e) M = 7 = 0111 S = −7 = 1001 −S = 0111	(f) M = −6 = 1010 S = 4 = 0100 −S = 1100

Figure 10.4 Subtraction of Numbers in Twos Complement Representation (M – S)

Geometric Depiction of Twos Complement Integers

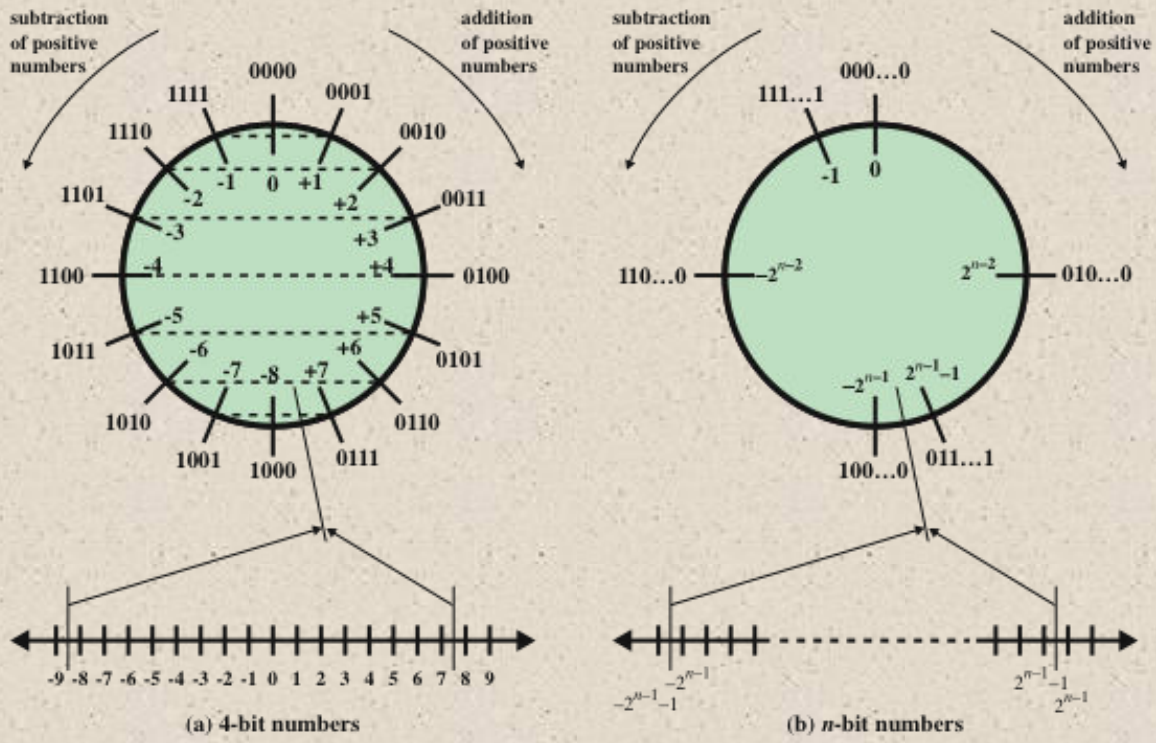
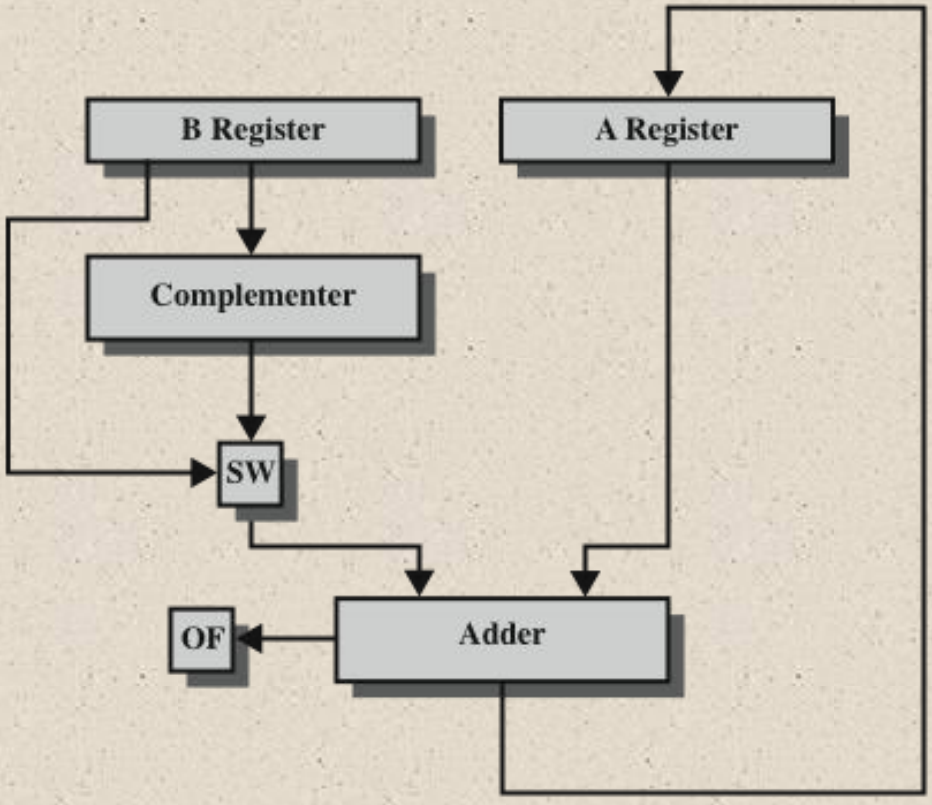


Figure 10.5 Geometric Depiction of Twos Complement Integers

Hardware for Addition and Subtraction



OF = overflow bit
SW = Switch (select addition or subtraction)

Figure 10.6 Block Diagram of Hardware for Addition and Subtraction

Subtraction using r's complement:

To find M-N in base r, we add M + r's complement of N

Result is M + (rn – N)

1) If M > N then result is M – N + rn (rn is an end carry and can be neglected.

2) If M < N then result is rn –(N-M) which is the r's complement of the result.

Signed Number Representation

- ***Signed Magnitude Method***
  - $N = \pm (a_{n-1} \ldots a_0.a_{-1} \ldots a_{-m})_r$ is represented as

    $N = (sa_{n-1} \ldots a_0.a_{-1} \ldots a_{-m})_{rsm}$, (1.6)

    where $s = 0$ if N is positive and $s = r - 1$ otherwise.
  - $N = -(15)_{10}$
  - In binary: $N = -(15)_{10} = -(1111)_2 = (1, 1111)_{2sm}$
  - In decimal: $N = -(15)_{10} = (9, 15)_{10sm}$

- ***Complementary Number Systems***
  - ***Radix complements*** (r's complements)

    $[N]_r = r^n - (N)_r$ (1.7)

    where n is the number of digits in $(N)_r$.
  - *Positive full scale*: $r^{n-1} - 1$
  - *Negative full scale*: $-r^n - 1$
  - ***Diminished radix complements*** ($r-1$'s complements)

$$[N]_{r-1} = r_n - (N)_r - 1$$

Radix Complement Number Systems (1)

- Two's complement of $(N)_2 = (101001)_2$

  $[N]_2 = 2^6 - (101001)_2 = (1000000)_2 - (101001)_2 = (010111)_2$

- $(N)_2 + [N]_2 = (101001)_2 + (010111)_2 = (1000000)_2$

  If we discard the carry, $(N)_2 + [N]_2 = 0$.

  Hence, $[N]_2$ can be used to represent $-(N)_2$.

- $[\,[N]_2\,]_2 = [(010111)_2]_2 = (1000000)_2 - (010111)_2 = (101001)_2 = (N)_2$.

- Two's complement of $(N)_2 = (1010)_2$ for $n = 6$

  $[N]_2 = (1000000)_2 - (1010)_2 = (110110)_2$.

Radix Complement Number Systems (2)

- ***Algorithm 1.4*** **Find $[N]_r$ given $(N)_r$.**
  - Copy the digits of N, beginning with the LSD and proceeding toward the MSD until the first nonzero digit, a_i, has been reached
  - Replace a_i with $r - a_i$.
  - Replace each remaining digit a_j , of N by $(r - 1) - a_j$ until the MSD has been replaced.
- ***Example***: 10's complement of $(56700)_{10}$ is $(43300)_{10}$
- ***Example***: 2's complement of $(10100)_2$ is $(01100)_2$.
- ***Example***: 2's complement of $N = (10110)_2$ for $n = 8$.
  - Put three zeros in the MSB position and apply algorithm 1.4
  - $N = 00010110$
  - $[N]_2 = (11101010)_2$
- The same rule applies to the case when N contains a radix point.

Radix Complement Number Systems (3)

- ***Algorithm 1.5*** **Find $[N]_r$ given $(N)_r$.**
  - First replace each digit, a_k, of $(N)_r$ by $(r - 1) - a_k$ and then add 1 to the resultant.
- For binary numbers ($r = 2$), complement each digit and add 1 to the result.
- ***Example***: Find 2's complement of $N = (01100101)_2$.

  $N = 01100101$

  10011010 Complement the bits

  $+1$ Add 1

  $[N]_2 = (10011011)_{10}$
- ***Example***: Find 10's complement of $N = (40960)_{10}$

  $N = 40960$

  59039 Complement the bits

  $+1$ Add 1

  $[N]_2 = (59040)_{10}$

Radix Complement Number Systems (4)

- ***Two's complement number system*** :
  - Positive number : $0 \le N \le 2^{n-1} - 1$
    - $N = +(a_{n-2}, \ldots, a_0)_2 = (0, a_{n-2}, \ldots, a_0)_{2cns}$,

      where .
  - Negative number: $-1 \ge N \ge -2^{n-1}$
    - $N = (a_{n-1}, a_{n-2}, \ldots, a_0)_2$
    - $-N = [a_{n-1}, a_{n-2}, \ldots, a_0]_2$ (two's complement of N),

      where .
  - ***Example***: Two's complement number system representation of $\pm (N)_2$ when $(N)_2 = (1011001)_2$ for $n = 8$:
    - $+(N)_2 = (0, 1011001)_{2cns}$
    - $-(N)_2 = [+(N)_2]_2 = [0, 1011001]_2 = (1, 0100111)_{2cns}$

Radix Complement Number Systems (5)

- ***Example***: Two's complement number system representation of $-(18)_{10}$, $n = 8$:
  - $+(18)_{10} = (0, 0010010)_{2cns}$
  - $-(18)_{10} = [0, 0010010]_2 = (1, 1101110)_{2cns}$
- ***Example***: Decimal representation of $N = (1, 1101000)_{2cns}$
  - $N = (1, 1101000)_{2cns} = -[1, 1101000]_2 = -(0, 0011000)_{2cns} = -(24)_2$.

Radix Complement Arithmetic (1)

- Radix complement number systems are used to convert subtraction to addition, which reduces hardware requirements (only adders are needed).
- $A - B = A + (-B)$ (add r's complement of B to A)
- Range of numbers in two's complement number system:

$$-2^{n-1} \le N \le 2^{n-1} - 1$$

, where n is the number of bits.

- $2^{n-1} - 1 = (0, 11 \ldots 1)_{2cns}$ and $-2^{n-1} = (1, 00 \ldots 0)_{2cns}$
- If the result of an operation falls outside the range, an ***overflow condition*** is said to occur and the result is not valid.
- Consider three cases:
  - $A = B + C$,
  - $A = B - C$,
  - $A = -B - C$,

  (where $B \ge 0$ and $C \ge 0$.)

Radix Complement Arithmetic (2)

- ***Case 1***: $A = B + C$
  - $(A)_2 = (B)_2 + (C)_2$
  - If $A > 2^{n-1} - 1$ (***overflow***), it is detected by the nth bit, which is set to 1.

  - ***Example***: $(7)_{10} + (4)_{10} = ?$ using 5-bit two's complement arithmetic.
    - $+(7)_{10} = +(0111)_2 = (0,0111)_{2cns}$
    - $+(4)_{10} = +(0100)_2 = (0,0100)_{2cns}$
    - $(0,0111)_{2cns} + (0,0100)_{2cns} = (0,1011)_{2cns} = +(1011)_2 = +(11)_{10}$
    - No overflow.
  - ***Example***: $(9)_{10} + (8)_{10} = ?$
    - $+(9)_{10} = +(1001)_2 = (0,1001)_{2cns}$
    - $+(8)_{10} = +(1000)_2 = (0,1000)_{2cns}$
    - $(0,1001)_{2cns} + (0,1000)_{2cns} = (\mathbf{1},0001)_{2cns}$ (***overflow***)

Radix Complement Arithmetic (3)

- ***Case 2***: $A = B - C$
  - $A = (B)_2 + (-(C)_2) = (B)_2 + [C]_2 = (B)_2 + 2^n - (C)_2 = 2^n + (B - C)_2$
  - If $B \geq C$, then $A \geq 2^n$ and the carry is discarded.
  - So, $(A)_2 = (B)_2 + [C]\,|_{\text{carry discarded}}$
  - If $B < C$, then $A = 2^n - (C - B)_2 = [C - B]_2$ or $A = -(C - B)_2$ (no carry in this case).
  - No overflow for Case 2.

  - ***Example***: $(14)_{10} - (9)_{10} = ?$
    - Perform $(14)_{10} + (-(9)_{10})$
    - $(14)_{10} = +(1110)_2 = (0, 1110)_{2cns}$
    - $-(9)_{10} = -(1001)_2 = (1, 0111)_{2cns}$
    - $(14)_{10} - (9)_{10} = (0, 1110)_{2cns} + (1, 0111)_{2cns} = (0, 0101)_{2cns} + carry$
      $= +(0101)_2 = +(5)_{10}$

Radix Complement Arithmetic (4)

- ***Example***: $(9)_{10} - (14)_{10} = ?$
  - Perform $(9)_{10} + (-(14)_{10})$
  - $(9)_{10} = +(1001)_2 = (0, 1001)_{2cns}$
  - $-(14)_{10} = -(1110)_2 = (1, 0010)_{2cns}$
  - $(9)_{10} - (14)_{10} = (0, 1001)_{2cns} + (1, 0010)_{2cns} = (1, 1011)_{2cns}$
    $= -(0101)_2 = -(5)_{10}$

- ***Example***: $(0, 0100)_{2cns} - (1, 0110)_{2cns} = ?$
  - Perform $(0, 0100)_{2cns} + (- (1, 0110)_{2cns})$
  - $- (1, 0110)_{2cns}$ = two's complement of $(1,0110)_{2cns}$
    $= (0, 1010)_{2cns}$
  - $(0, 0100)_{2cns} - (1, 0110)_{2cns} = (0, 0100)_{2cns} + (0, 1010)_{2cns}$
    $= (0, 1110)_{2cns} = +(1110)_2 = +(14)_{10}$
  - $+(4)_{10} - (-(10)_{10}) = +(14)_{10}$

Radix Complement Arithmetic (5)

- ***Case 3***: $A = -B - C$
  - $A = [B]_2 + [C]_2 = 2^n - (B)_2 + 2^n - (C)_2 = 2^n + 2^n - (B + C)_2 = 2^n + [B + C]_2$
  - The carry bit (2^n) is discarded.
  - An overflow can occur, in which case the sign bit is 0.

  - ***Example***: $-(7)_{10} - (8)_{10} = ?$
    - Perform $(-(7)_{10}) + (-(8)_{10})$
    - $-(7)_{10} = -(0111)_2 = (1, 1001)_{2cns}$, $-(8)_{10} = -(1000)_2 = (1, 1000)_{2cns}$
    - $-(7)_{10} - (8)_{10} = (1, 1001)_{2cns} + (1, 1000)_{2cns} = (1, 0001)_{2cns} + carry$
      $= -(1111)_2 = -(15)_{10}$
  - ***Example***: $-(12)_{10} - (5)_{10} = ?$
    - Perform $(-(12)_{10}) + (-(5)_{10})$
    - $-(12)_{10} = -(1100)_2 = (1, 0100)_{2cns}$, $-(5)_{10} = -(0101)_2 = (1, 1011)_{2cns}$
    - $-(7)_{10} - (8)_{10} = (1, 0100)_{2cns} + (1, 1011)_{2cns} = (0, 1111)_{2cns} + carry$
    - ***Overflow***, because the sign bit is 0.

Radix Complement Arithmetic (6)

- ***Example***: $A = (25)_{10}$ and $B = -(46)_{10}$
  - $A = +(25)_{10} = (0,0011001)_{2cns}$, $-A = (1,1100111)_{2cns}$
  - $B = -(46)_{10} = -(0,0101110)_2 = (1,1010010)_{2cns}$, $-B = (0,0101110)_{2cns}$
  - $A + B = (0,0011001)_{2cns} + (1,1010010)_{2cns} = (1,1101011)_{2cns} = -(21)_{10}$
  - $A - B = A + (-B) = (0,0011001)_{2cns} + (0,0101110)_{2cns}$
    $= (0,1000111)_{2cns} = +(71)_{10}$
  - $B - A = B + (-A) = (1,1010010)_{2cns} + (1,1100111)_{2cns}$
    $= (1,0111001)_{2cns} + carry = -(0,1000111)_{2cns} = -(71)_{10}$
  - $-A - B = (-A) + (-B) = (1,1100111)_{2cns} + (0,0101110)_{2cns}$
    $= (0,0010101)_{2cns} + carry = +(21)_{10}$
  - Note: Carry bit is discarded.

Radix Complement Arithmetic (7)

- Summary

Case	Carry	Sign Bit	Condition	Overflow ?
B + C	0	0	$B + C \leq 2^{n-1} - 1$	No
	0	1	$B + C > 2^{n-1} - 1$	Yes
B - C	1	0	$B \leq C$	No
	0	1	$B > C$	No
-B - C	1	1	$-(B + C) \geq -2^{n-1}$	No
	1	0	$-(B + C) < -2^{n-1}$	Yes

- When numbers are represented using two's complement number system:
  - Addition: Add two numbers.
  - Subtraction: Add two's complement of the subtrahend to the minuend.
  - Carry bit is discarded, and overflow is detected as shown above.

  - Radix complement arithmetic can be used for any radix.

Diminished Radix Complement Number systems (1)

- ***Diminished radix complement*** $[N]_{r-1}$ of a number $(N)_r$ is:

  $[N]_{r-1} = r^n - (N)_r - 1$ (1.10)

- ***One's complement*** $(r = 2)$:

  $[N]_{2-1} = 2^n - (N)_2 - 1$ (1.11)

- ***Example***: One's complement of $(01100101)_2$

  $[N]_{2-1} = 2^8 - (01100101)_2 - 1$

  $= (100000000)_2 - (01100101)_2 - (00000001)_2$

  $= (10011011)_2 - (00000001)_2$

  $= (10011010)_2$

Diminished Radix Complement Number systems (2)

- ***Example***: Nine's complement of (40960)

  $[N]_{2-1} = 10^5 - (40960)_{10} - 1$

  $= (100000)_{10} - (40960)_{10} - (00001)_{10}$

  $= (59040)_{10} - (00001)_{10}$

  $= (59039)_{10}$

- ***Algorithm 1.6*** **Find $[N]_{r-1}$ given $(N)_r$.**

  Replace each digit a_i of $(N)_r$ by $r - 1 - a$. Note that when $r = 2$, this simplifies to complementing each individual bit of $(N)_r$.

- Radix complement and diminished radix complement of a number (N):

  $[N]_r = [N]_{r-1} + 1$ (1.12)

Diminished Radix Complement Arithmetic (1)

- Operands are represented using diminished radix complement number system.
- The carry, if any, is added to the result (***end-around carry***).
- ***Example***: Add $+(1001)_2$ and $-(0100)_2$.

  One's complement of +(1001) = 01001

  One's complement of -(0100) = 11011

  01001 + 11011 = 100100 (carry)

  Add the carry to the result: correct result is 00101.

- ***Example***: Add $+(1001)_2$ and $-(1111)_2$.

  One's complement of +(1001) = 01001

  One's complement of -(1111) = 10000

  01001 + 10000 = 11001 (no carry, so this is the correct result).

Diminished Radix Complement Arithmetic (2)

- ***Example***: Add $-(1001)_2$ and $-(0011)_2$.
  One's complement of the operands are: 10110 and 11100
  10110 + 11100 = 110010 (carry)
  Correct result is 10010 + 1 = 10011.

- ***Example***: Add $+(75)_{10}$ and $-(21)_{10}$.
  Nine's complements of the operands are: 075 and 978
  075 + 978 = 1053 (carry)
  Correct result is 053 + 1 = 054

- ***Example***: Add $+(21)_{10}$ and $-(75)_{10}$.
  Nine's complements of the operands are: 021 and 924
  021 + 924 = 945 (no carry, so this is the correct result).

Example (3):

Using 2's complement, subtract 1010100 – 1000011

$X - Y$

X	=	1010100
2's complement of Y	= +	0111101
Sum	=	10010001
Discard end carry 2^7	= -	10000000
Answer: X - Y	=	0010001

Example (4):

Using 2's complement, subtract 1000011 – 1010100

$Y - X$

Y	=	1000011
2's complement of X	= +	0101100
Sum	=	1101111
No end carry.		
Answer: $Y - X$ - (2's complement of 1101111)	=	-0010001

Example (5): Using 1's complement, subtract X – Y = 1010100 – 1000011

X	=	1010100
1's complement of Y	= +	0111100 (+1 End-around carry)
Sum	=	10010000
		+ 1
Answer: X - Y	=	0010001

Example (6): Using 1's complement, subtract $Y - X$ 1000011 – 1010100

Y	=	1000011
1's complement of X	= +	0101011
Sum	=	1101110
No end carry.		
Answer: $Y - X$ - (1's complement of 1101110)	=	-0010001

Multiplication

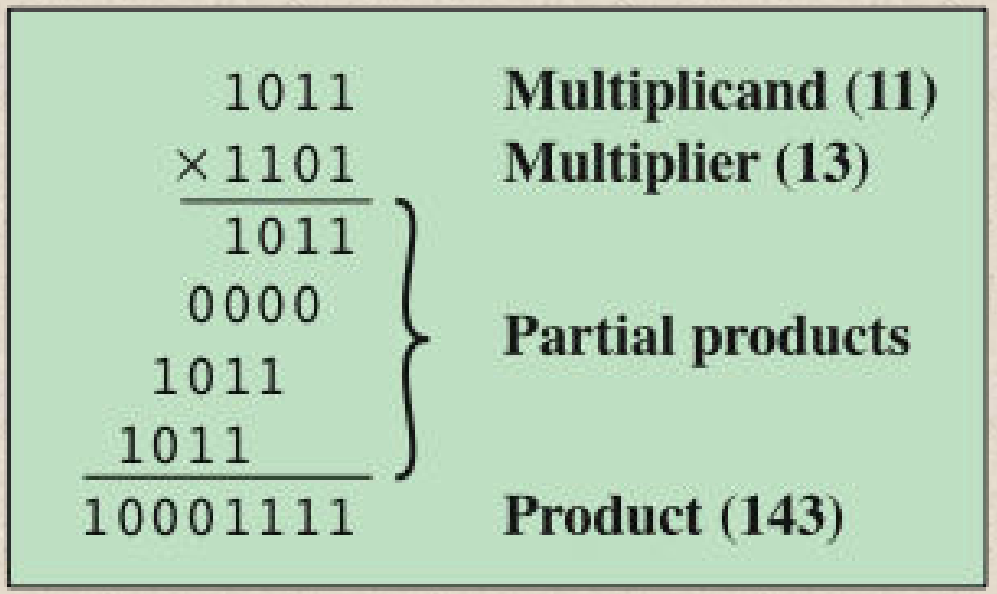
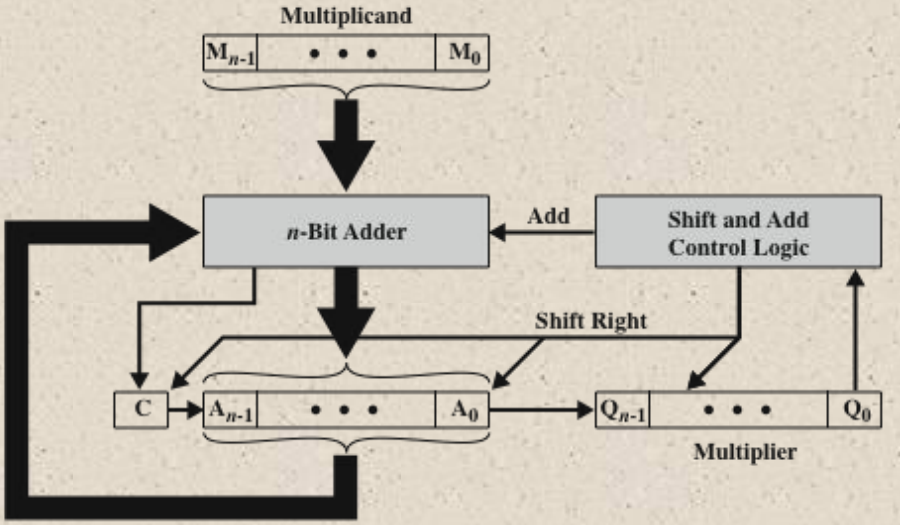


Figure 10.7 Multiplication of Unsigned Binary Integers

Hardware Implementation of Unsigned Binary Multiplication



(a) Block Diagram

C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

(b) Example from Figure 9.7 (product in A, Q)

Figure 10.8 Hardware Implementation of Unsigned Binary Multiplication

Flowchart for Unsigned Binary Multiplication

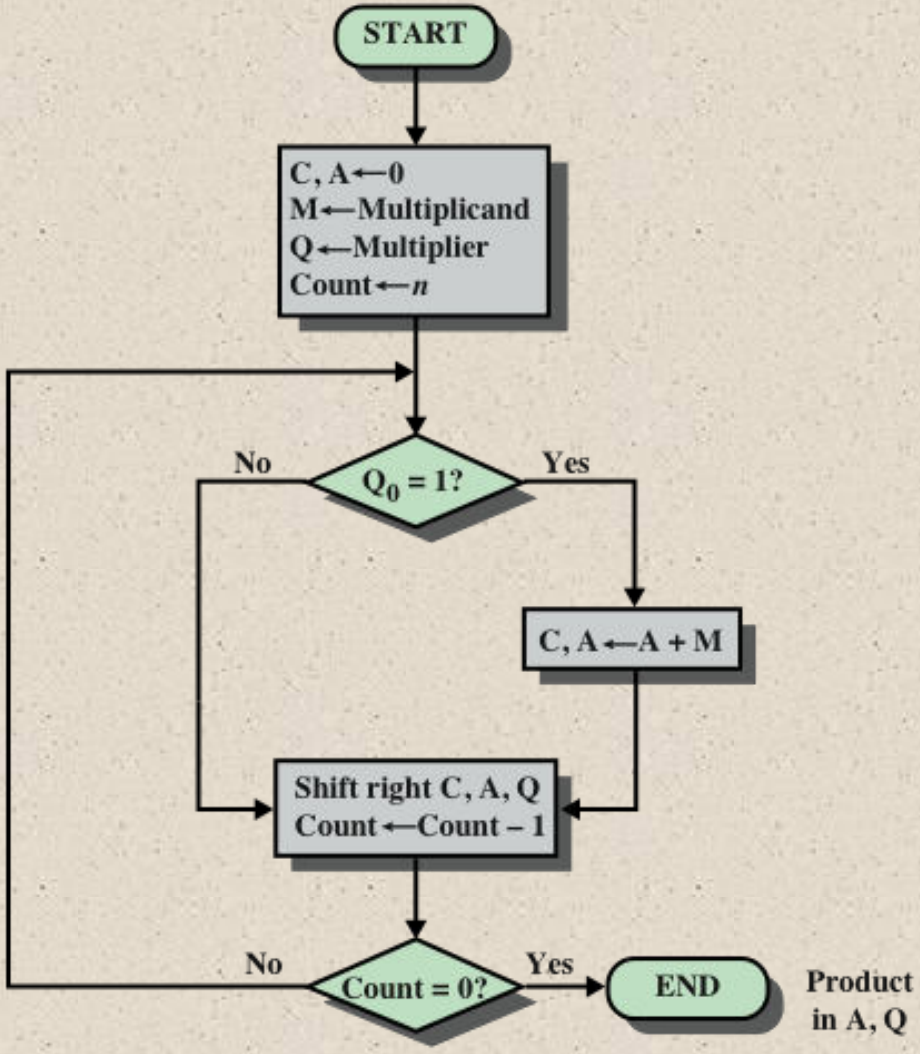


Figure 10.9 Flowchart for Unsigned Binary Multiplication

Twos Complement Multiplication

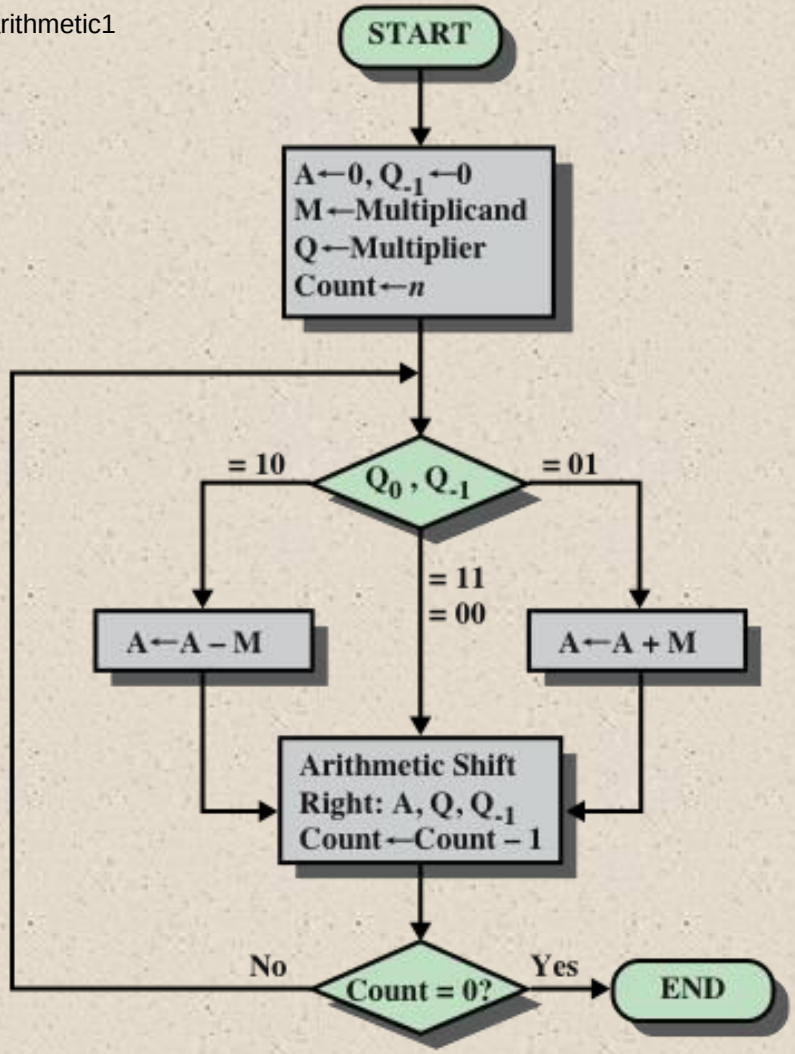
1011	
×1101	
00001011	$1011 \times 1 \times 2^0$
00000000	$1011 \times 0 \times 2^1$
00101100	$1011 \times 1 \times 2^2$
01011000	$1011 \times 1 \times 2^3$
10001111	

Figure 10.10 Multiplication of Two Unsigned 4-Bit Integers Yielding an 8-Bit Result

Comparison

1001 (9) ×0011 (3) 00001001 1001 × 2^0 00010010 1001 × 2^1 00011011 (27)	1001 (−7) ×0011 (3) 11111001 (−7) × 2^0 = (−7) 11110010 (−7) × 2^1 = (−14) 11101011 (−21)
(a) Unsigned integers	(b) Twos complement integers

Figure 10.11 Comparison of Multiplication of Unsigned and Twos Complement Integers



Booth's

Algorithm

Figure 10.12 Booth's Algorithm for Twos Complement Multiplication

Example of Booth's Algorithm

A	Q	Q_{-1}	M		
0000	0011	0	0111	Initial Values	
1001	0011	0	0111	$A \leftarrow A - M$	First Cycle
1100	1001	1	0111	Shift	
1110	0100	1	0111	Shift	Second Cycle
0101	0100	1	0111	$A \leftarrow A + M$	Third Cycle
0010	1010	0	0111	Shift	
0001	0101	0	0111	Shift	Fourth Cycle

Figure 10.13 Example of Booth's Algorithm (7× 3)

Examples Using Booth's Algorithm

```
    0111
   ×0011     (0)
11111001     1–0
0000000      1–1
000111       0–1
00010101     (21)
```

(a) (7) × (3) = (21)

```
    0111
   ×1101     (0)
11111001     1–0
0000111      0–1
111001       1–0
11101011     (–21)
```

(b) (7) × (–3) = (–21)

```
    1001
   ×0011     (0)
00000111     1–0
0000000      1–1
111001       0–1
11101011     (–21)
```

(c) (–7) × (3) = (–21)

```
    1001
   ×1101     (0)
00000111     1–0
1111001      0–1
000111       1–0
00010101     (21)
```

(d) (–7) × (–3) = (21)

Figure 10.14 Examples Using Booth's Algorithm

Division

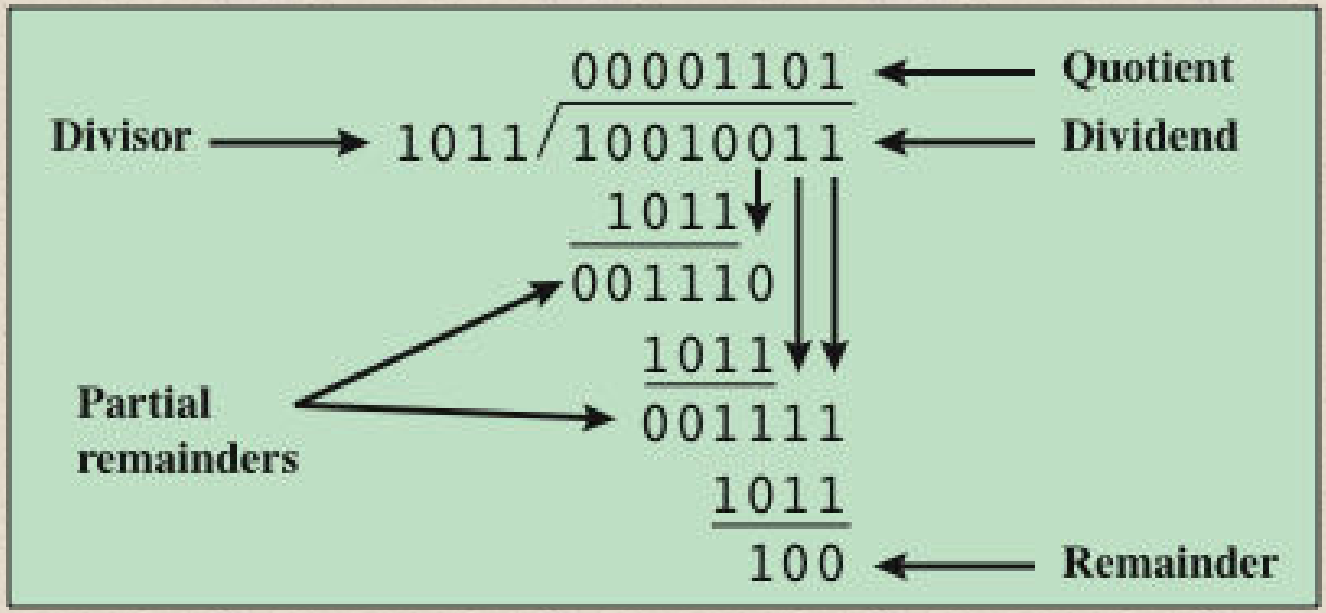


Figure 10.15 Example of Division of Unsigned Binary Integers

Flowchart for Unsigned Binary Division

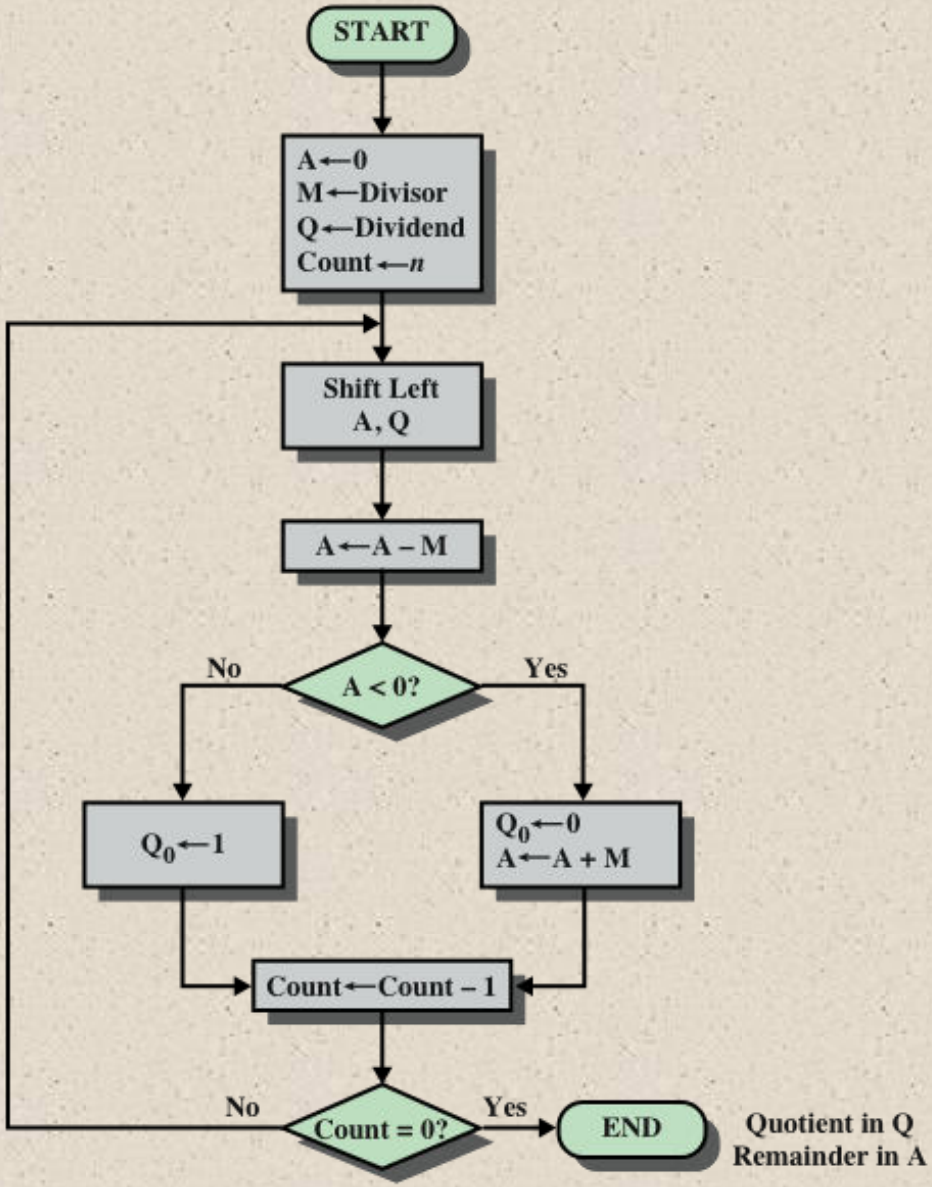


Figure 10.16 Flowchart for Unsigned Binary Division

Example of Restoring Twos Complement Division

A	Q	
0000	0111	Initial value
0000	1110	Shift
1101		Use twos complement of 0011 for subtraction
1101		Subtract
0000	1110	Restore, set $Q_0 = 0$
0001	1100	Shift
1101		
1110		Subtract
0001	1100	Restore, set $Q_0 = 0$
0011	1000	Shift
1101		
0000	1001	Subtract, set $Q_0 = 1$
0001	0010	Shift
1101		
1110		Subtract
0001	0010	Restore, set $Q_0 = 0$

Figure 10.17 Example of Restoring Twos Complement Division (7/3)

Floating-Point Representation

Principles

- With a fixed-point notation it is possible to represent a range of positive and negative integers centered on or near 0
- By assuming a fixed binary or radix point, this format allows the representation of numbers with a fractional component as well
- Limitations:
  - Very large numbers cannot be represented nor can very small fractions
  - The fractional part of the quotient in a division of two large numbers could be lost

Typical 32-Bit Floating-Point Format

(a) Format

$1.1010001 \times 2^{10100} = 0\ 10010011\ 10100010000000000000000 = 1.6328125 \times 2^{20}$

$-1.1010001 \times 2^{10100} = 1\ 10010011\ 10100010000000000000000 = -1.6328125 \times 2^{20}$

$1.1010001 \times 2^{-10100} = 0\ 01101011\ 10100010000000000000000 = 1.6328125 \times 2^{-20}$

$-1.1010001 \times 2^{-10100} = 1\ 01101011\ 10100010000000000000000 = -1.6328125 \times 2^{-20}$

(b) Examples

Figure 10.18 Typical 32-Bit Floating-Point Format

The closest binary number to Y that can be stored by computer in 32 bits is:

$$(-1)^S \frac{J \text{ (in binary)}}{2^{23}} \times 2^P$$

where $S=0$ if $y \le 0$ and $S=1$ if $y>1$

$\rightarrow z = |y|$

$\rightarrow P = \text{Floor}(\log_2 z) = \text{Floor}\left(\frac{\log z}{\log 2}\right)$

$J = \text{Round}(z \times 2^{23-P})$

a) Convert the given IEEE 754 formatted 32 –bit floating point number in to decimal

1 10111011 10110000000000000000

b) Define Normalization. Give two examples.

Give the flow chart for addition and subtraction of two floating-point numbers.

Show the IEEE 754 binary representation of the number $(-0.4375)_{ten}$ in single precision.

Floating-Point

Significand

- The final portion of the word
- Any floating-point number can be expressed in many ways

The following are equivalent, where the significand is expressed in binary form:

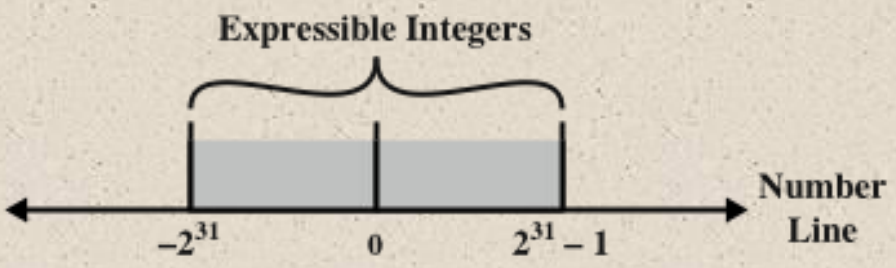
$0.110 * 2^5$

$110 * 2^2$

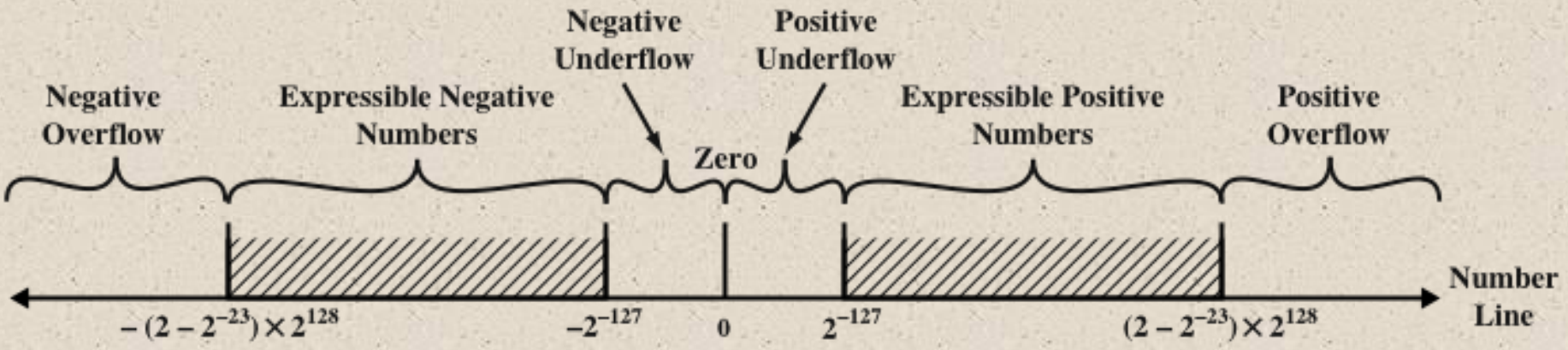
$0.0110 * 2^6$

- *Normal number*
  - The most significant digit of the significand is nonzero

Expressible Numbers



(a) Twos Complement Integers



(b) Floating-Point Numbers

Figure 10.19 Expressible Numbers in Typical 32-Bit Formats

Density of Floating-Point Numbers

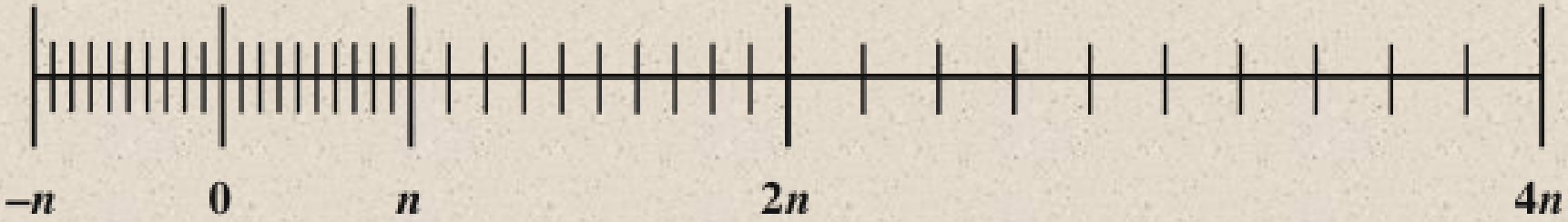


Figure 10.20 Density of Floating-Point Numbers

IEEE Standard 754

Most important floating-point representation is defined

Standard was developed to facilitate the portability of programs from one processor to another and to encourage the development of sophisticated, numerically oriented programs

Standard has been widely adopted and is used on virtually all contemporary processors and arithmetic coprocessors

IEEE 754-2008 covers both binary and decimal floating-point representations

IEEE 754-2008

- Defines the following different types of floating-point formats:
- Arithmetic format
  - All the mandatory operations defined by the standard are supported by the format. The format may be used to represent floating-point operands or results for the operations described in the standard.
- Basic format
  - This format covers five floating-point representations, three binary and two decimal, whose encodings are specified by the standard, and which can be used for arithmetic. At least one of the basic formats is implemented in any conforming implementation.
- Interchange format
  - A fully specified, fixed-length binary encoding that allows data interchange between different platforms and that can be used for storage.

IEEE 754 Formats

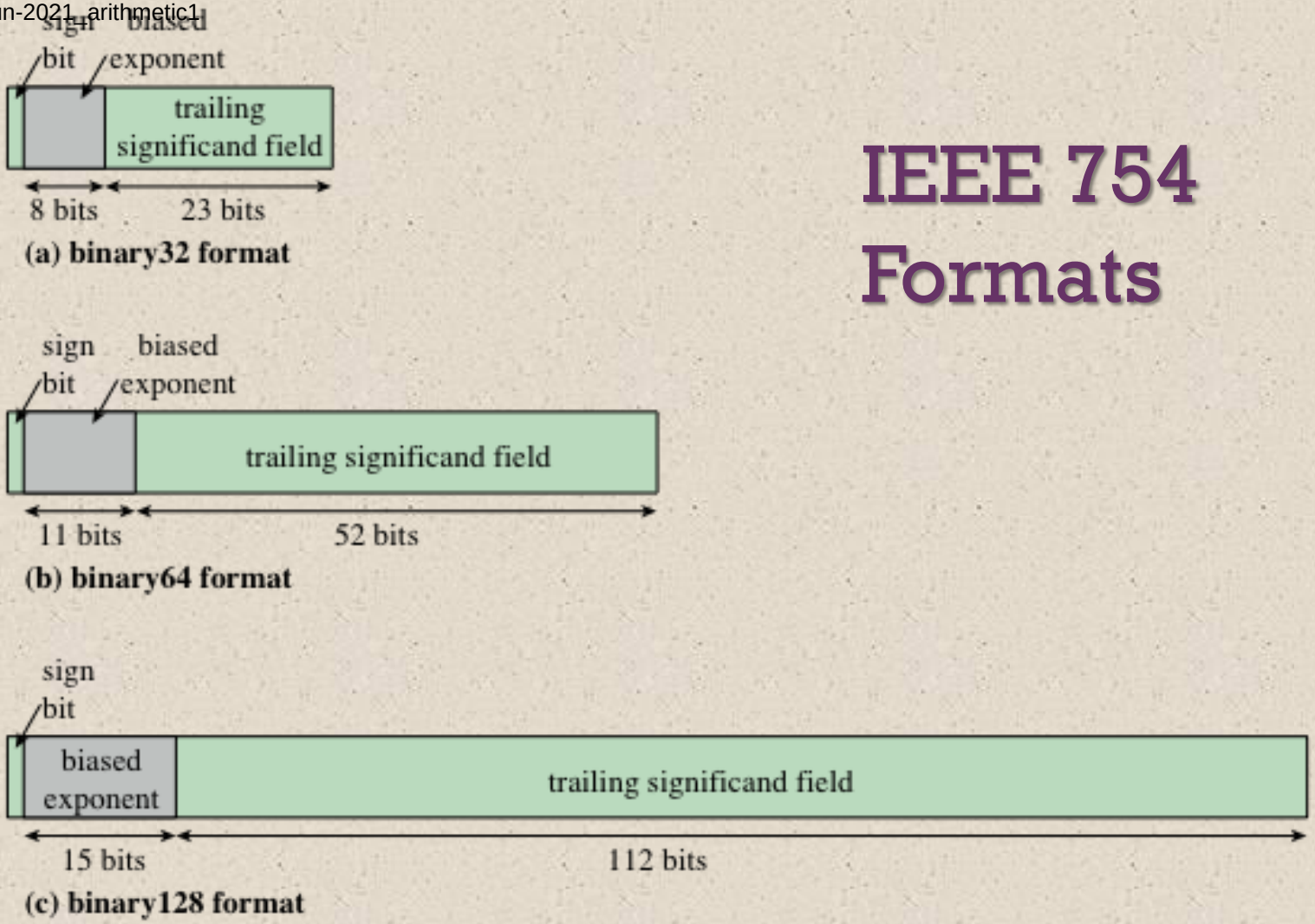


Figure 10.21 IEEE 754 Formats

Parameter	Format		
	binary32	binary64	binary128
Storage width (bits)	32	64	128
Exponent width (bits)	8	11	15
Exponent bias	127	1023	16383
Maximum exponent	127	1023	16383
Minimum exponent	−126	−1022	−16382
Approx normal number range (base 10)	$10_{-38}, 10_{+38}$	$10_{-308}, 10_{+308}$	$10_{-4932}, 10_{+4932}$
Trailing significand width (bits)*	23	52	112
Number of exponents	254	2046	32766
Number of fractions	2_{23}	2_{52}	2_{112}
Number of values	$1.98 \times 2_{31}$	$1.99 \times 2_{63}$	$1.99 \times 2_{128}$
Smallest positive normal number	2_{-126}	2_{-1022}	2_{-16362}
Largest positive normal number	$2_{128} - 2_{104}$	$2_{1024} - 2_{971}$	$2_{16384} - 2_{16271}$
Smallest subnormal magnitude	2_{-149}	2_{-1074}	2_{-16494}

Table 10.3

IEEE 754

Format Parameters

* not including implied bit and not including sign bit

Additional Formats

Extended Precision Formats

- Provide additional bits in the exponent (extended range) and in the significand (extended precision)
- Lessens the chance of a final result that has been contaminated by excessive roundoff error
- Lessens the chance of an intermediate overflow aborting a computation whose final result would have been representable in a basic format
- Affords some of the benefits of a larger basic format without incurring the time penalty usually associated with higher precision

Extendable Precision Format

- Precision and range are defined under user control
- May be used for intermediate calculations but the standard places no constraint or format or length

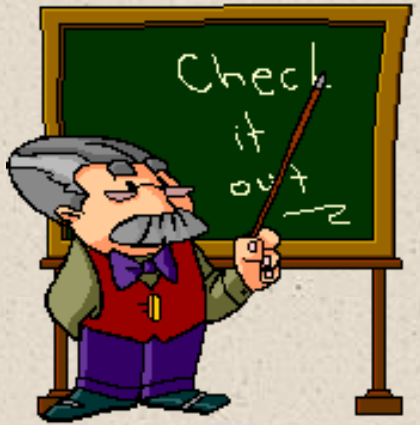


Table 10.4
IEEE Formats

Format	Format Type		
	Arithmetic Format	Basic Format	Interchange Format
binary16			X
binary32	X	X	X
binary64	X	X	X
binary128	X	X	X
binary{k} ($k = n \times 32$ for $n > 4$)	X		X
decimal64	X	X	X
decimal128	X	X	X
decimal{k} ($k = n \times 32$ for $n > 4$)	X		X
extended precision	X		
extendable precision	X		

Table 10.4 IEEE Formats

Interpretation of IEEE 754 Floating-Point Numbers

(a) binary 32 format

	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	–0
plus infinity	0	all 1s	0	∞
Minus infinity	1	all 1s	0	–∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	0 < e < 255	f	$2_{e-127}(1.f)$
negative normal nonzero	1	0 < e < 255	f	$-2_{e-127}(1.f)$
positive subnormal	0	0	f ≠ 0	$2_{e-126}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2_{e-126}(0.f)$

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers (page 1 of 3)

Interpretation of IEEE 754 Floating-Point Numbers

(b) binary 64 format

	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	–0
plus infinity	0	all 1s	0	∞
Minus infinity	1	all 1s	0	–∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	0 < e < 2047	f	$2_{e-1023}(1.f)$
negative normal nonzero	1	0 < e < 2047	f	$-2_{e-1023}(1.f)$
positive subnormal	0	0	f ≠ 0	$2_{e-1022}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2_{e-1022}(0.f)$

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers (page 2 of 3)

Interpretation of IEEE 754 Floating-Point Numbers

(c) binary 128 format

	Sign	Biased exponent	Fraction	Value
positive zero	0	0	0	0
negative zero	1	0	0	–0
plus infinity	0	all 1s	0	∞
minus infinity	1	all 1s	0	–∞
quiet NaN	0 or 1	all 1s	≠ 0; first bit = 1	qNaN
signaling NaN	0 or 1	all 1s	≠ 0; first bit = 0	sNaN
positive normal nonzero	0	all 1s	f	$2_{e-16383}(1.f)$
negative normal nonzero	1	all 1s	f	$-2_{e-16383}(1.f)$
positive subnormal	0	0	f ≠ 0	$2_{e-16383}(0.f)$
negative subnormal	1	0	f ≠ 0	$-2_{e-16383}(0.f)$

Table 10.5 Interpretation of IEEE 754 Floating-Point Numbers (page 3 of 3)

Table 10.6 Floating-Point Numbers and Arithmetic Operations

Floating Point Numbers	Arithmetic Operations
$X = X_S \times B^{X_E}$ $Y = Y_S \times B^{Y_E}$	$X + Y = \left(X_s \times B^{X_E - Y_E} + Y_s\right) \times B^{Y_E}$ $X - Y = \left(X_s \times B^{X_E - Y_E} - Y_s\right) \times B^{Y_E}$ $X_E \le Y_E$ $X \times Y = \left(X_s \times Y_s\right) \times B^{X_E + Y_E}$ $\frac{X}{Y} = \left(\frac{X_s}{Y_s}\right) \times B^{X_E - Y_E}$

Examples:

$X = 0.3 \times 10^2 = 30$
$Y = 0.2 \times 10^3 = 200$

$X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = 0.23 \times 10^3 = 230$
$X - Y = (0.3 \times 10^{2-3} - 0.2) \times 10^3 = (-0.17) \times 10^3 = -170$
$X \times Y = (0.3 \times 0.2) \times 10^{2+3} = 0.06 \times 10^5 = 6000$
$X \div Y = (0.3 \div 0.2) \times 10^{2-3} = 1.5 \times 10^{-1} = 0.15$

Floating-Point Addition and Subtraction

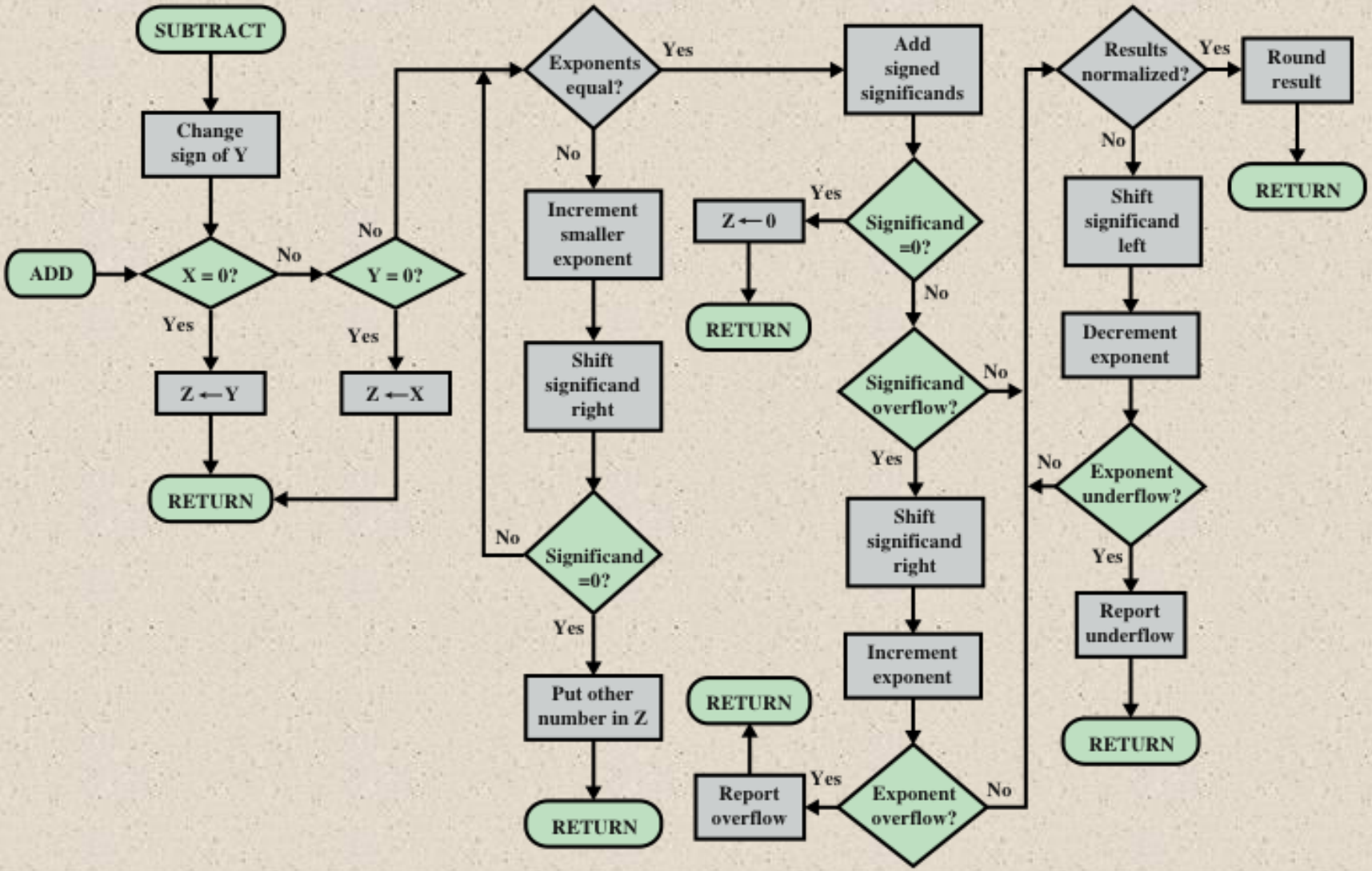


Figure 10.22 Floating-Point Addition and Subtraction (Z← X ± Y)

Floating-Point Multiplication

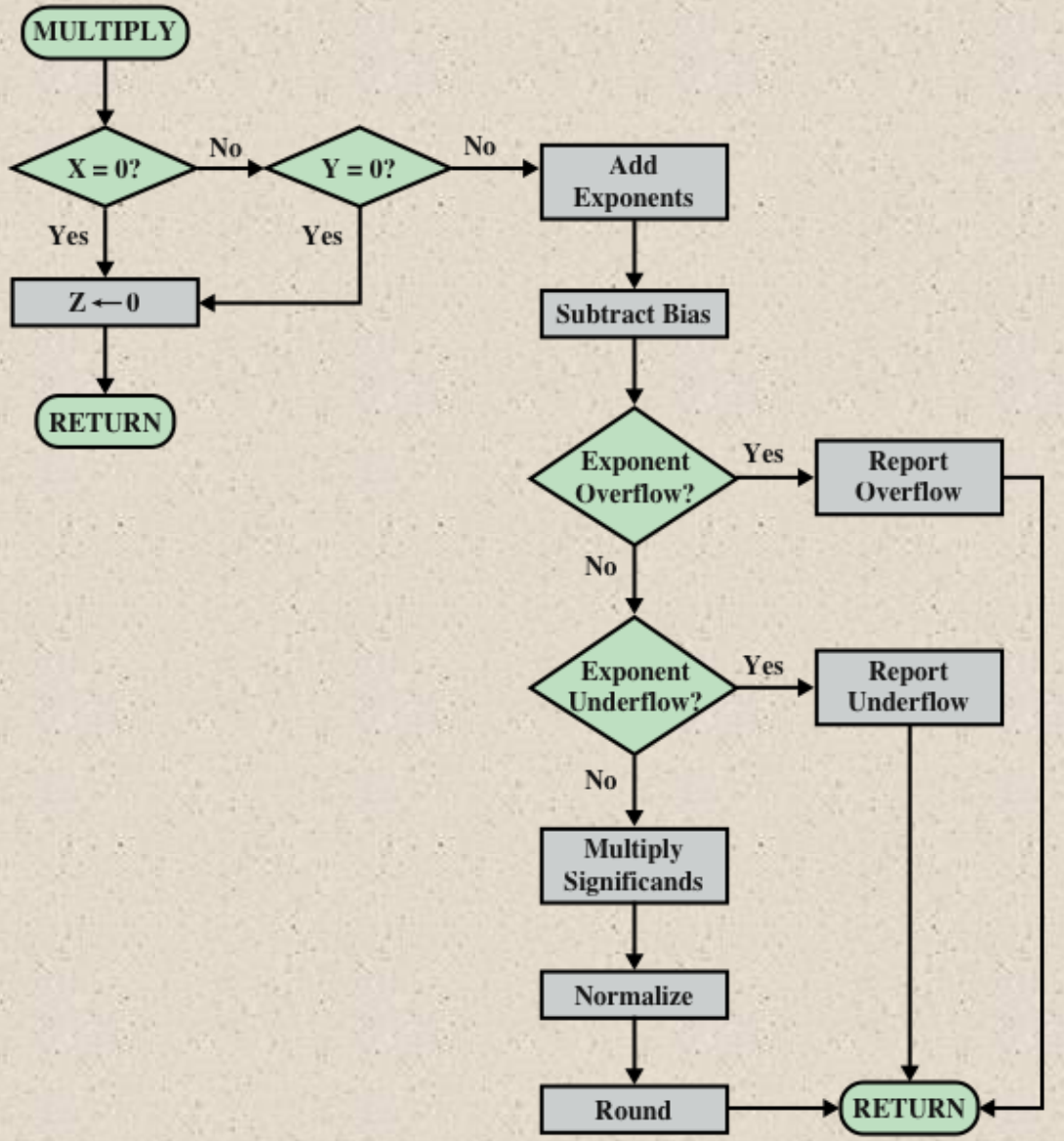


Figure 10.23 Floating-Point Multiplication (Z← X× Y)

Floating-Point Division

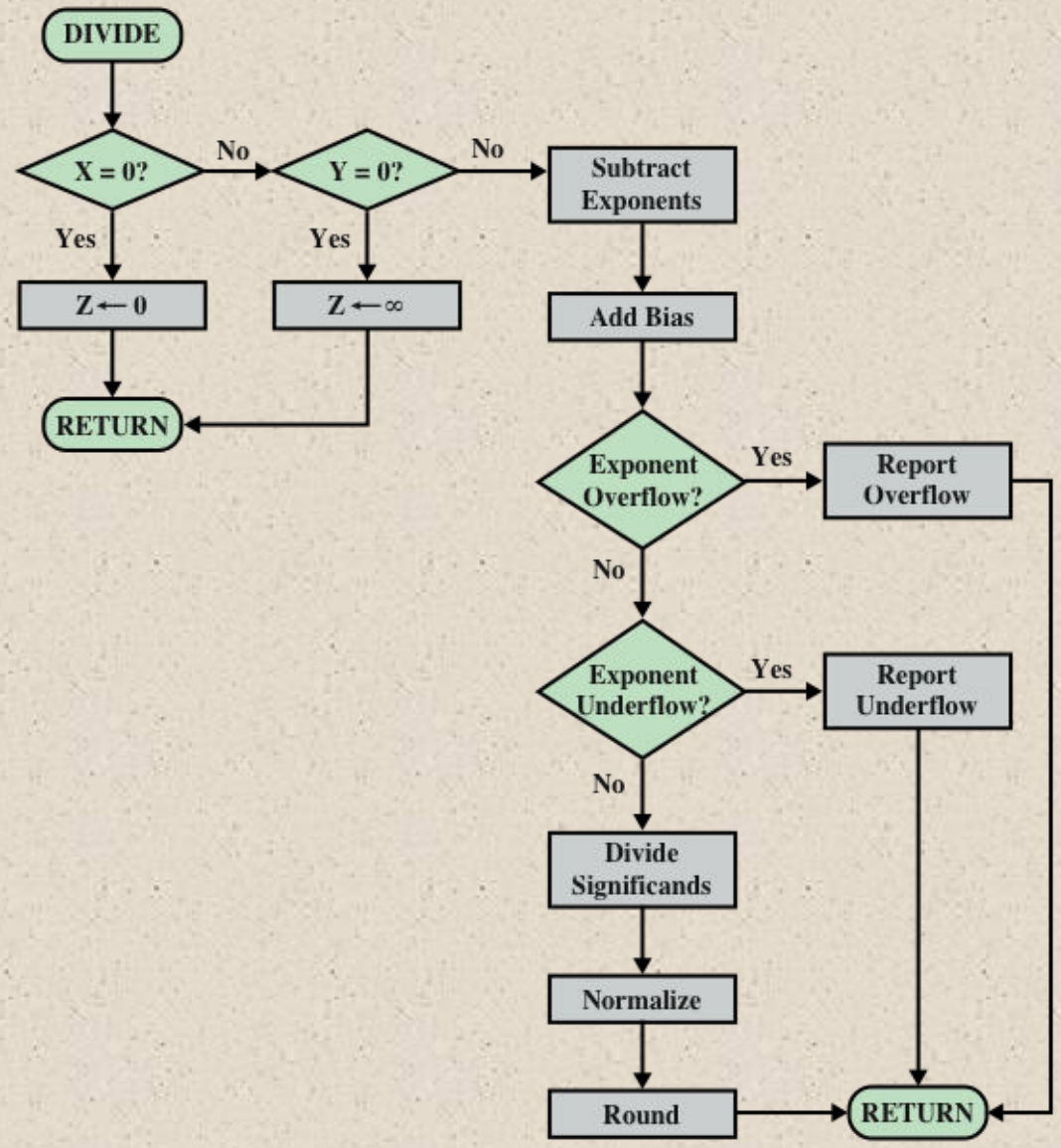


Figure 10.24 Floating-Point Division (Z← X/Y)

Precision Considerations

Guard Bits

$$
\begin{aligned}
x &= 1.000.....00 \times 2^{1} \\
-y &= \underline{0.111.....11} \times 2^{1} \\
z &= 0.000.....01 \times 2^{1} \\
&= 1.000.....00 \times 2^{-22}
\end{aligned}
$$

(a) Binary example, without guard bits

$$
\begin{aligned}
x &= .100000 \times 16^{1} \\
-y &= \underline{.0FFFFF} \times 16^{1} \\
z &= .000001 \times 16^{1} \\
&= .100000 \times 16^{-4}
\end{aligned}
$$

(c) Hexadecimal example, without guard bits

$$
\begin{aligned}
x &= 1.000.....00\ 0000 \times 2^{1} \\
-y &= \underline{0.111.....11\ 1000} \times 2^{1} \\
z &= 0.000.....00\ 1000 \times 2^{1} \\
&= 1.000.....00\ 0000 \times 2^{-23}
\end{aligned}
$$

(b) Binary example, with guard bits

$$
\begin{aligned}
x &= .100000\ 00 \times 16^{1} \\
-y &= \underline{.0FFFFF\ F0} \times 16^{1} \\
z &= .000000\ 10 \times 16^{1} \\
&= .100000\ 00 \times 16^{-5}
\end{aligned}
$$

(d) Hexadecimal example, with guard bits

Figure 10.25 The Use of Guard Bits

Precision Considerations

Rounding

- IEEE standard approaches:
  - Round to nearest:
    - The result is rounded to the nearest representable number.
  - Round toward +∞ :
    - The result is rounded up toward plus infinity.
  - Round toward -∞:
    - The result is rounded down toward negative infinity.
  - Round toward 0:
    - The result is rounded toward zero.

Interval Arithmetic

- Provides an efficient method for monitoring and controlling errors in floating-point computations by producing two values for each result
- The two values correspond to the lower and upper endpoints of an interval that contains the true result
- The width of the interval indicates the accuracy of the result
- If the endpoints are not representable then the interval endpoints are rounded down and up respectively
- If the range between the upper and lower bounds is sufficiently narrow then a sufficiently accurate result has been obtained
- *Minus infinity* and *rounding to plus* are useful in implementing interval arithmetic

Truncation

- *Round toward zero*
- Extra bits are ignored
- Simplest technique
- A consistent bias toward zero in the operation
  - Serious bias because it affects every operation for which there are nonzero extra bits

IEEE Standard for Binary Floating-Point Arithmetic

Infinity

Is treated as the limiting case of real arithmetic, with the infinity values given the following interpretation:

$$-\infty < (\text{every finite number}) < +\infty$$

For example:

$5 + (+\infty) = +\infty$

$5 - (+\infty) = -\infty$

$5 + (-\infty) = -\infty$

$5 - (-\infty) = +\infty$

$5 * (+\infty) = +\infty$

$5 \div (+\infty) = +0$

$(+\infty) + (+\infty) = +\infty$

$(-\infty) + (-\infty) = -\infty$

$(-\infty) - (+\infty) = -\infty$

$(+\infty) - (-\infty) = +\infty$

IEEE Standard for Binary Floating-Point Arithmetic

Quiet and Signaling NaNs

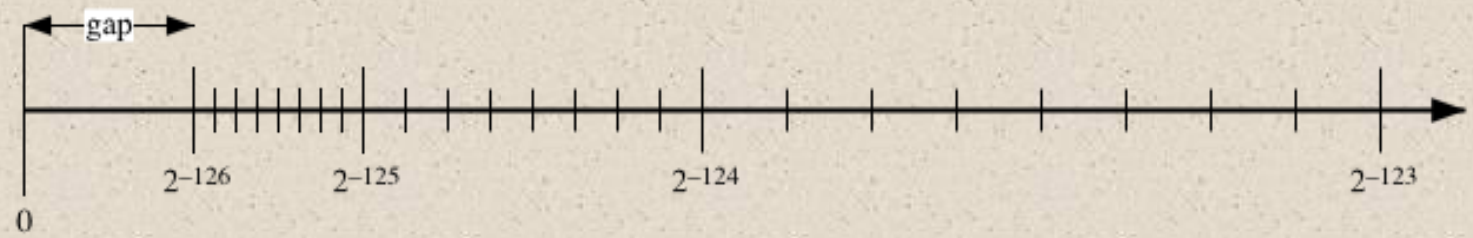
- Signaling NaN signals an invalid operation exception whenever it appears as an operand
- Quiet NaN propagates through almost every arithmetic operation without signaling an exception

Operation	Quiet NaN Produced by
Any	Any operation on a signaling NaN
Add or subtract	Magnitude subtraction of infinities: $(+\infty) + (-\infty)$ $(-\infty) + (+\infty)$ $(+\infty) - (+\infty)$ $(-\infty) - (-\infty)$
Multiply	$0 \times \infty$
Division	$\frac{0}{0}$ or $\frac{\infty}{\infty}$
Remainder	x REM 0 or ∞ REM y
Square root	$\sqrt{x}$ where $x < 0$

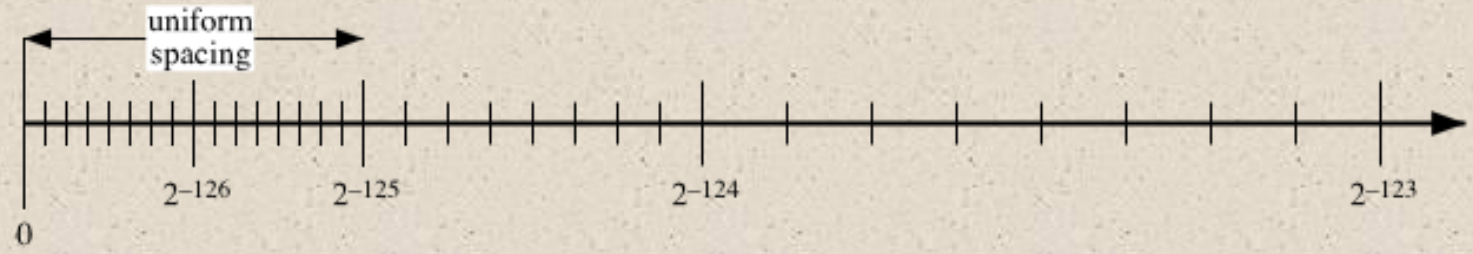
Operations that Produce a Quiet NaN

IEEE Standard for Binary Floating-Point Arithmetic

Subnormal Numbers



(a) 32-bit format without subnormal numbers



(b) 32-bit format with subnormal numbers

Figure 10.26 The Effect of IEEE 754 Subnormal Numbers

FUNDAMENTALS OF COMPUTER ARCHITECTURE

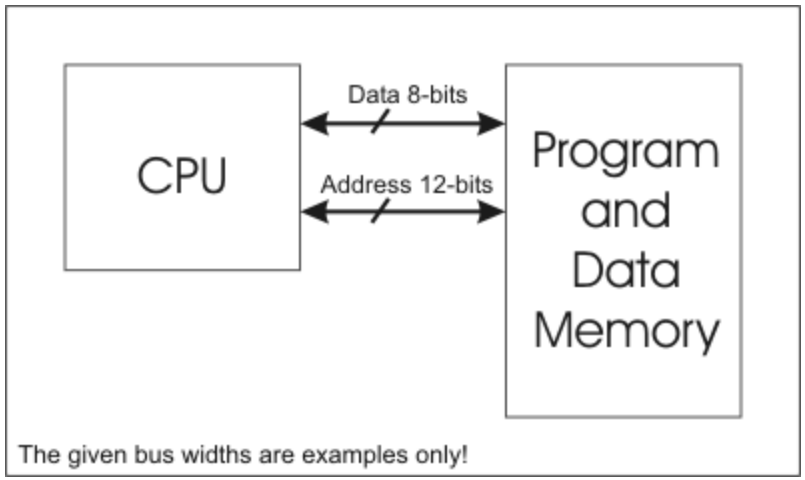
Unit - I

Topics of Unit - I

- Organization of the von Neumann machine
- Instruction formats
- The fetch/execute cycle, instruction decoding and execution;
- Registers and register files;
- Instruction types and addressing modes;
- Subroutine call and return mechanisms
- Programming in assembly language
- I/O techniques and interrupts
- Other design issues.

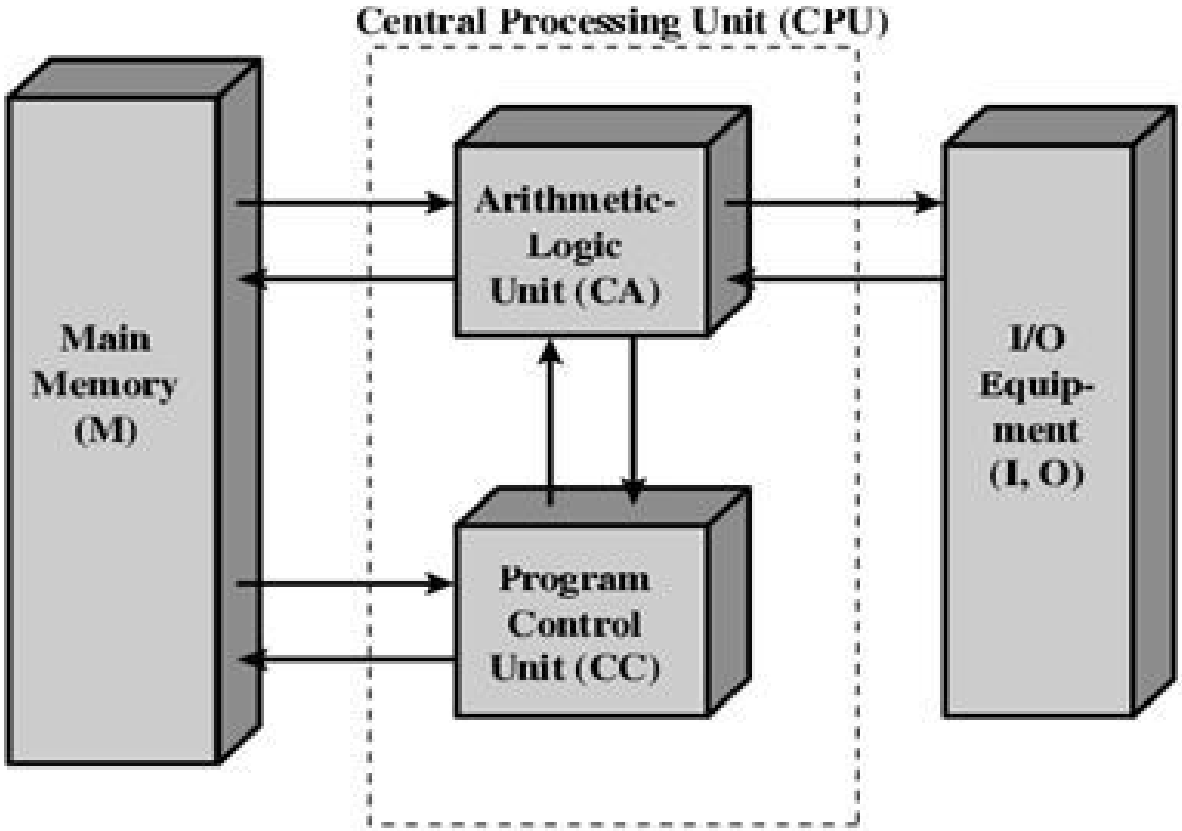
VON NEUMANN ARCHITECTURE:

- Computer has ***single*** storage system(memory) for storing data as well as program to be executed.
- ***Processor needs two clock cycles*** to complete an instruction (Query and Reply)



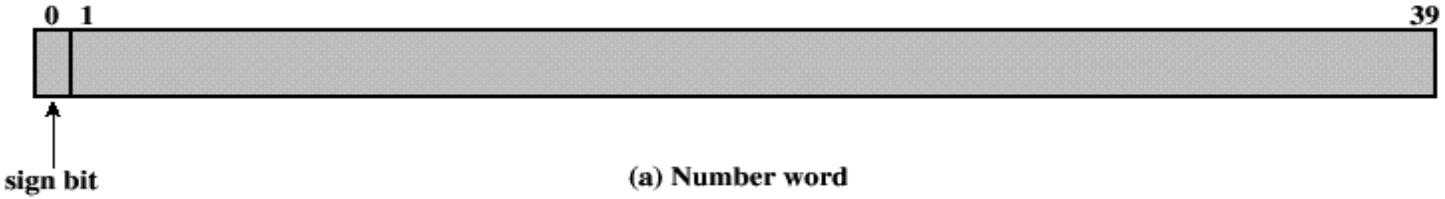
- Pipelining is not possible
- This is a relatively older architecture and was replaced by Harvard architecture.

Structure of Von Neumann Machine

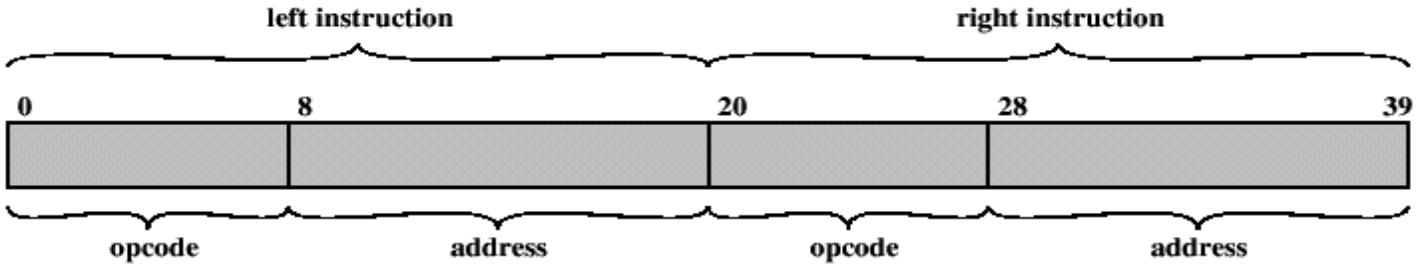


Memory of the IAS

- 1000 storage locations called words. (Institute for Advanced Studies (IAS)).
- Each word 40 bits.
- A word may contain:
  - A numbers stored as 40 binary digits (bits) – sign bit + 39 bit value
  - An instruction-pair. Each instruction:
    - An opcode (8 bits)
    - An address (12 bits) – designating one of the 1000 words in memory.



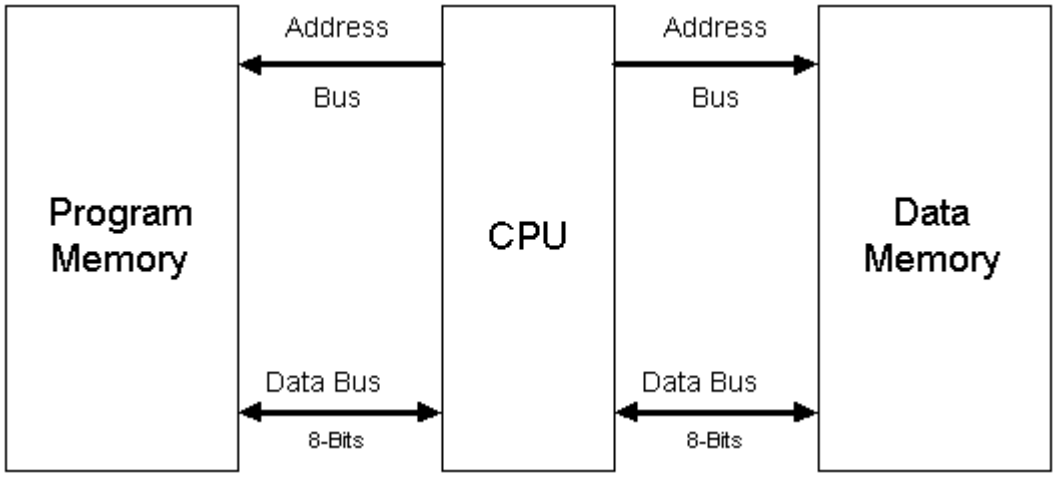
(a) Number word



(b) Instruction word

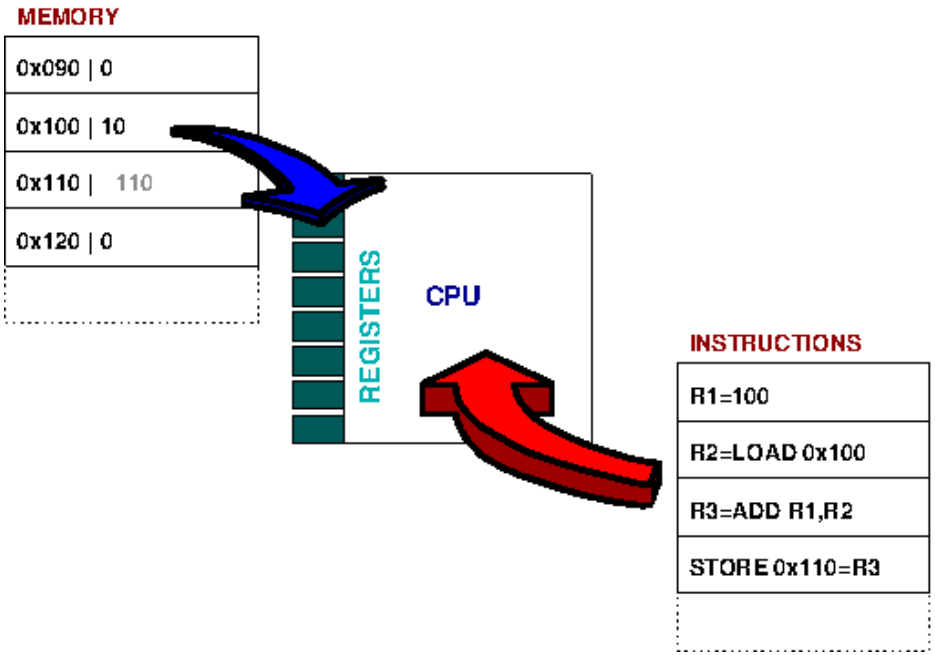
HARVARD ARCHITECTURE

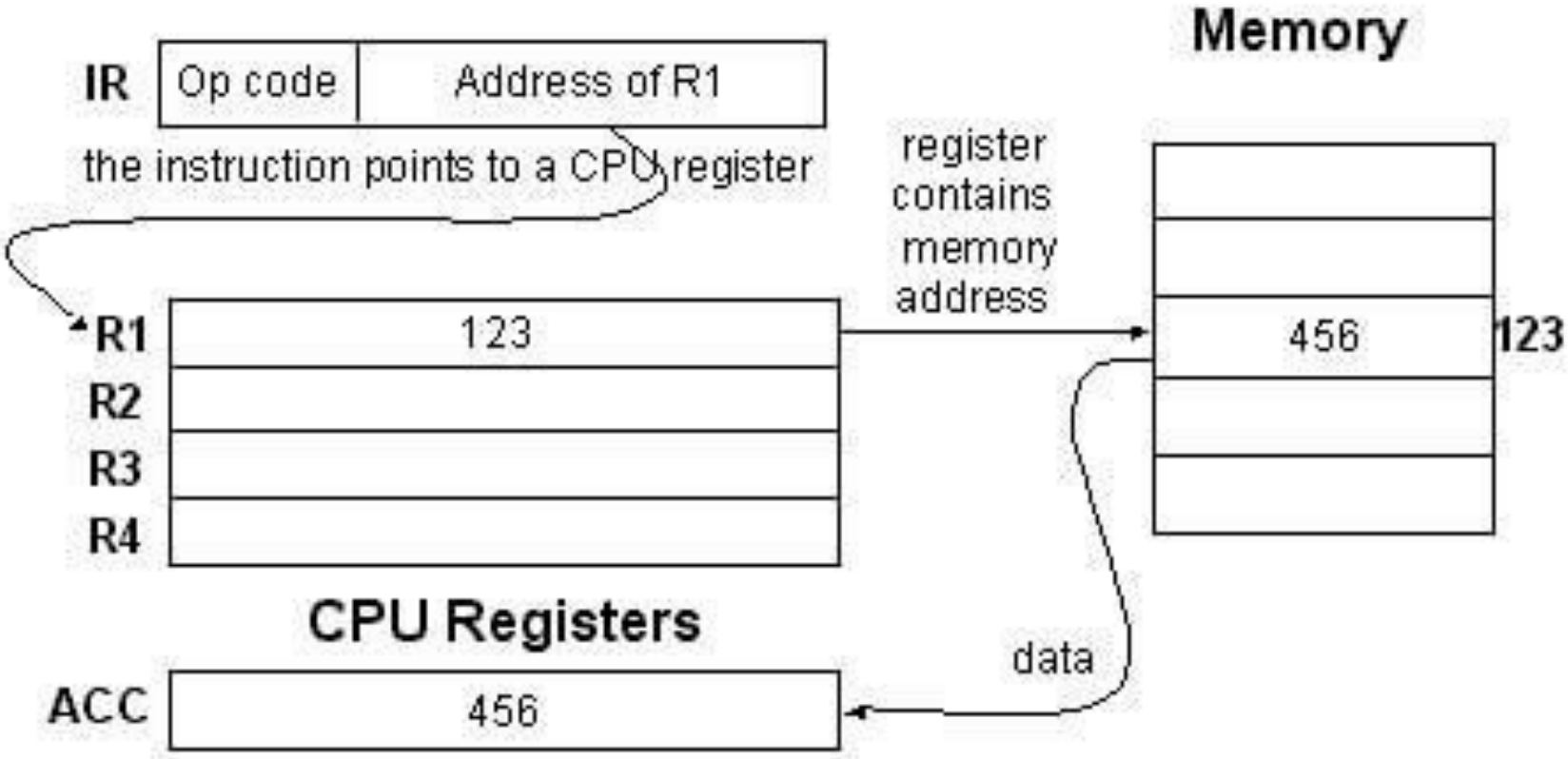
- Computer has two separate memories for storing data and program
- Processor can complete an instruction in one cycle
- Pipelining is possible



Instruction Format

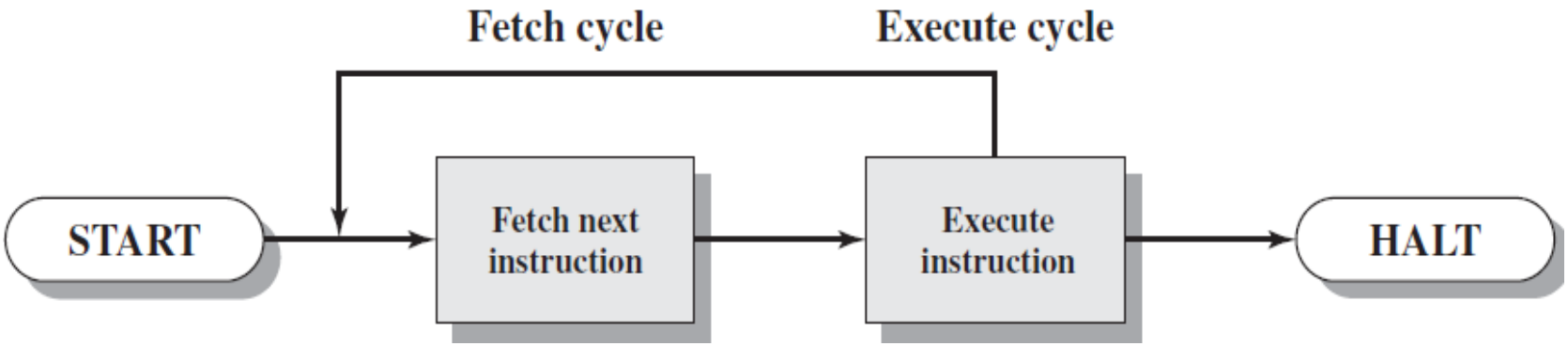
- An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming.



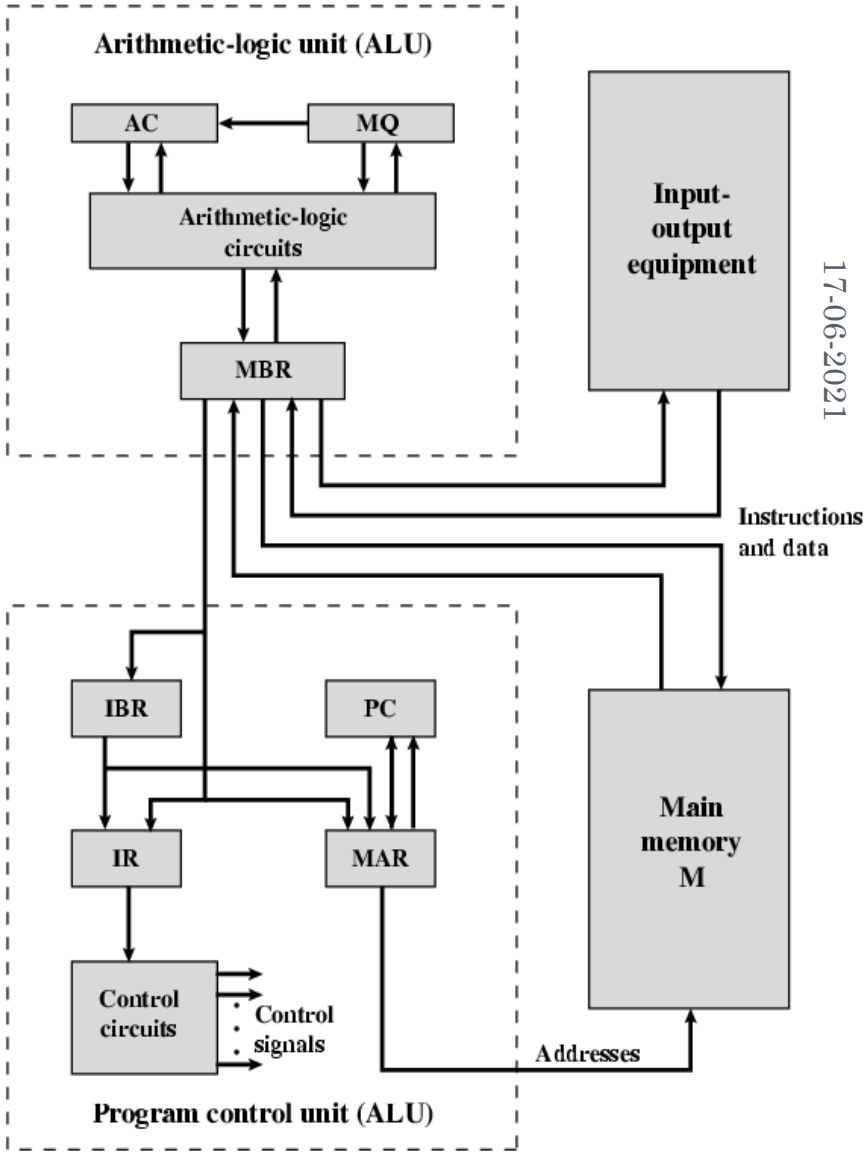

Memory
IR
Op code
Address of R1
the instruction points to a CPU register
register contains memory address
R1
123
R2
R3
R4
456
123
CPU Registers
ACC
456
data

COMPUTER FUNCTION

- The basic function performed by a computer is execution of a program, which consists of a set of instructions stored in memory.
- Instruction fetch-decode-execute



- **MBR: Memory Buffer Register** - contains the word to be stored in memory or just received from memory.
- **MAR: Memory Address Register** - specifies the address in memory of the word to be stored or retrieved.
- **IR: Instruction Register -** contains the 8-bit opcode currently being executed.
- **IBR: Instruction Buffer Register** - temporary store for RHS instruction from word in memory.
- **PC: Program Counter -** address of next instruction-pair to fetch from memory.
- **AC: Accumulator & MQ: Multiplier quotient** - holds operands and results of ALU ops.

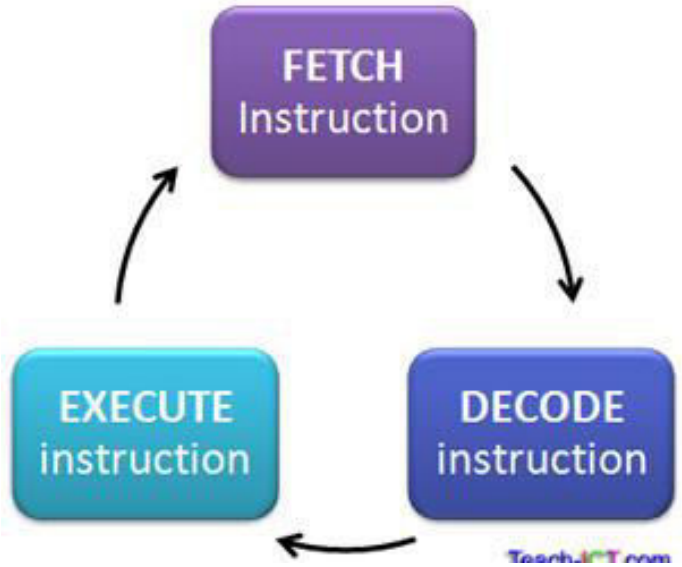


THE FETCH/EXECUTE CYCLE

- Standard process.

Also called as

- fetch-and-execute cycle,
- fetch-decode-execute cycle
- FDX



QUESTIONS:

- MBR –
- MAR –
- AC –
- IBR –
- IR –
- PC –
- MQ –
- IAS –
- What is Computer Architecture?
- What is Computer Organization?
- Number of words in IAS machine?
- Number of bits per word in IAS machine?
- Data is represented in _____ form in IAS machine
- Explain Stored program concept.

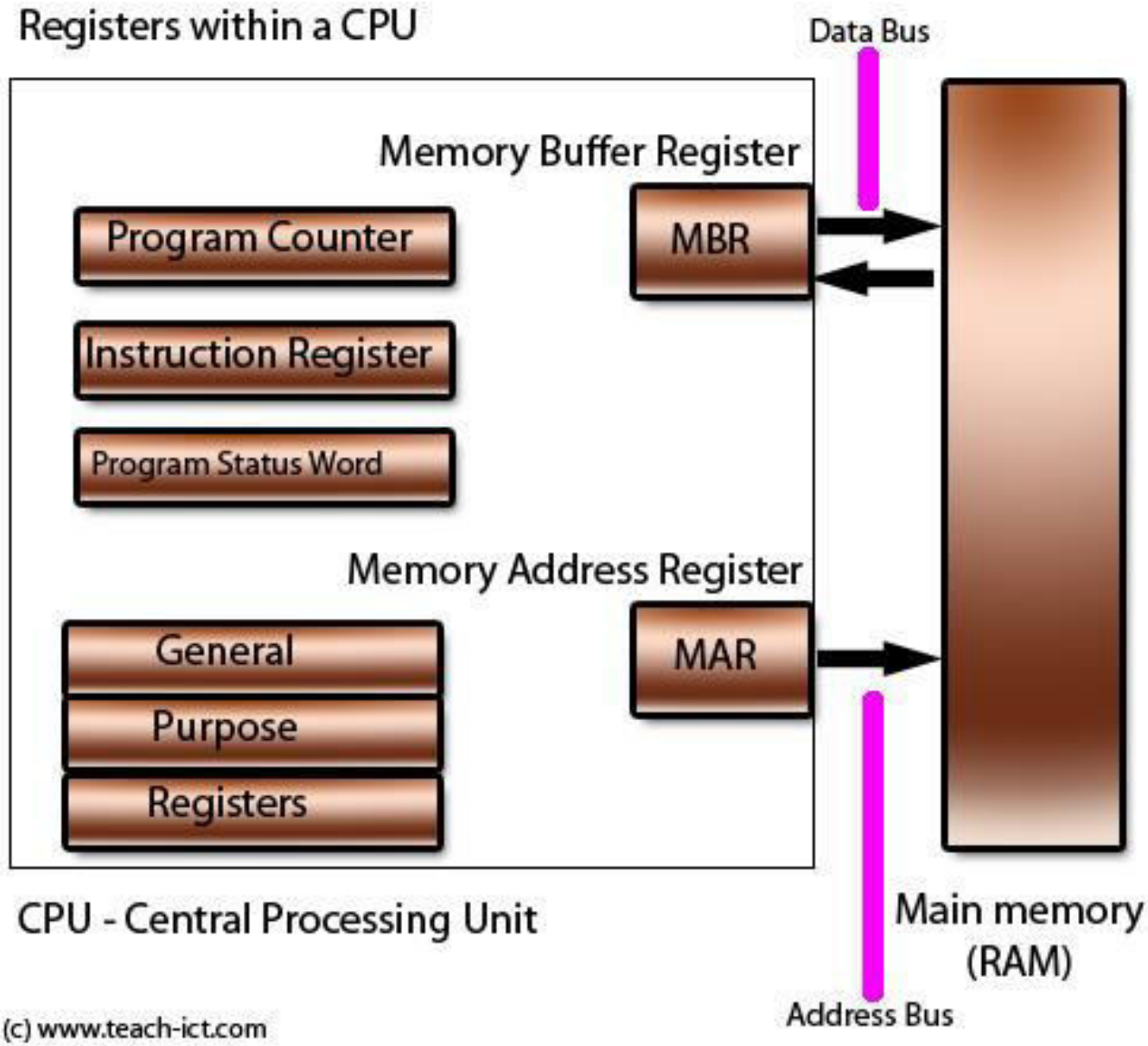
REGISTERS AND REGISTER FILES

- Registers?
  - Group of Flip-flops capable of storing one bit of information.
  - N bit registers consists of a group of N flip-flops capable of storing N bits.
  - Provides storage internal to the CPU.

As the instructions are interpreted and executed by the CPU, there is a movement of information between the various units of the computer system. In order to handle this process satisfactorily, and to speed up the rate of information transfer, the computer uses a number of special memory units, called registers. These registers are used to hold information on temporary basis, and are part of the CPU (not main memory).

CONT..

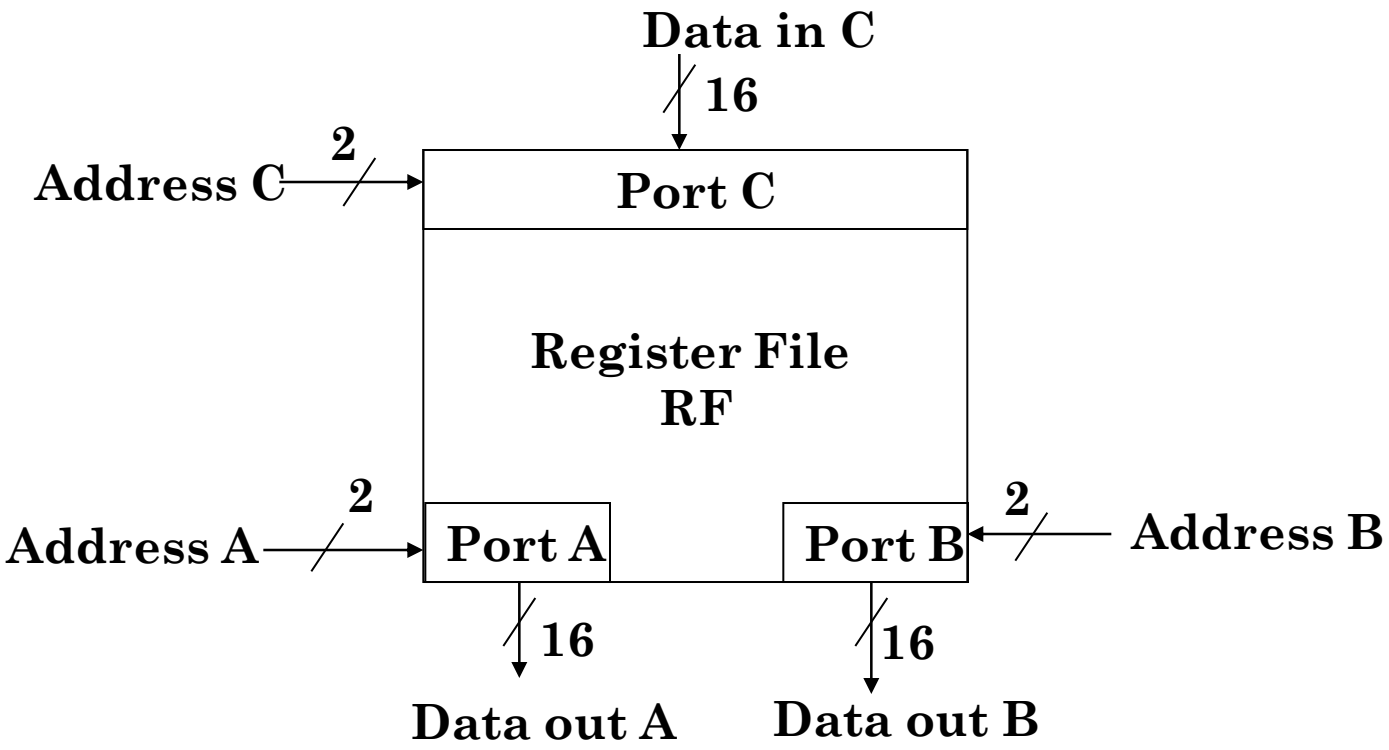
- The length of a register equals the number of bits it can store.
- Hence, a register that can store 8 bits is normally referred to as 8-bit register.
- Most CPU sold today, have 32-bit or 64-bit registers.
- The size of the registers is sometimes called the world size.
- The bigger the world size, the faster the computer can process a set of data.
- With all other parameters being same, a CPU with 32-bit registers, can process data twice as fast as one with 16-bit registers.


Registers within a CPU
Data Bus
Memory Buffer Register
Program Counter
MBR
Instruction Register
Program Status Word
Memory Address Register
General
Purpose
Registers
MAR
CPU - Central Processing Unit
Main memory
(RAM)
Address Bus
(c) www.teach-ict.com

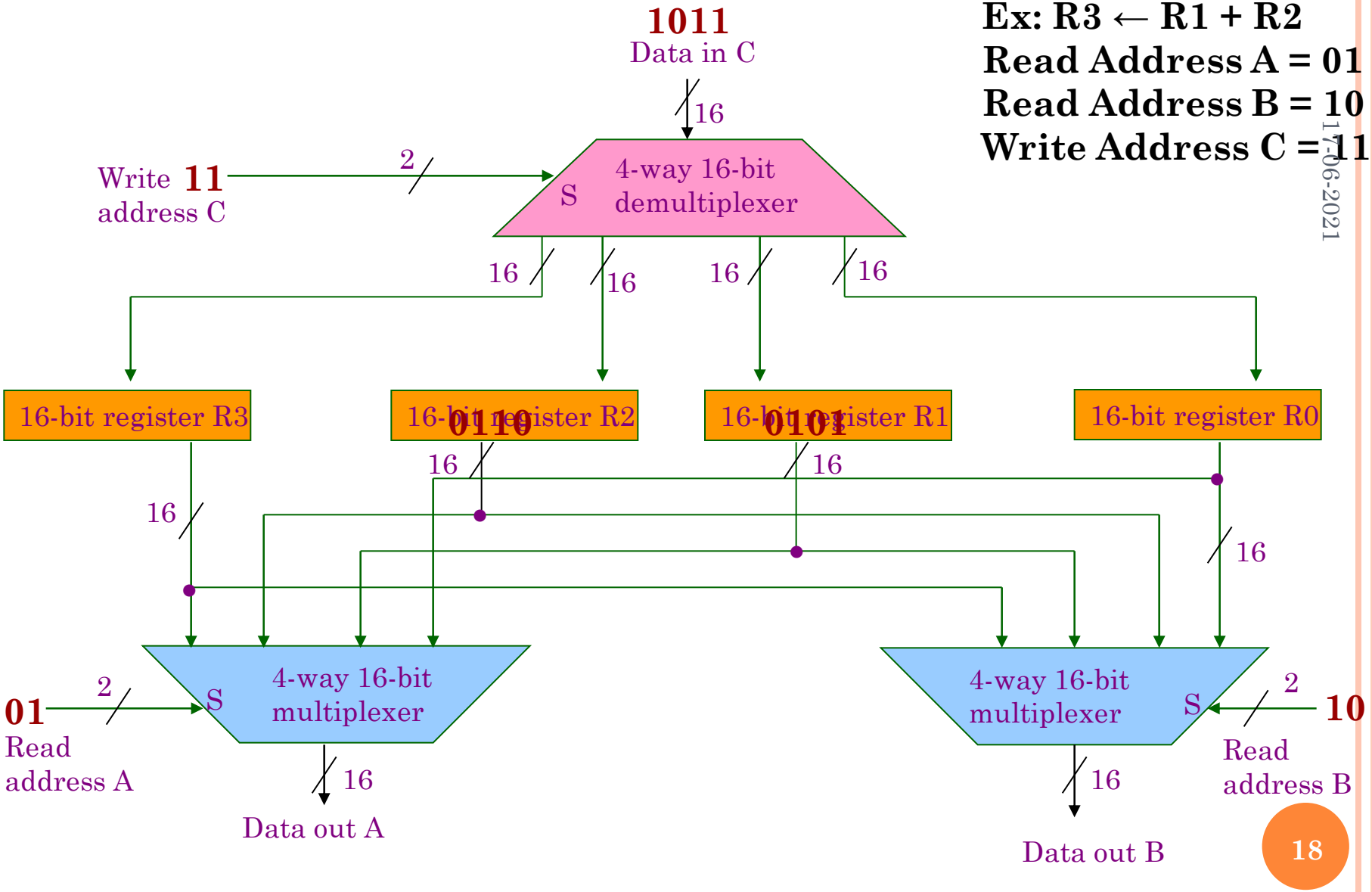
Register Files (RF)

- Set of general purpose registers.
- It functions as small RAM and implemented using fast RAM technology.
- RF needs several access ports for simultaneously reading from or writing to several different registers. Hence RF is realized as multiport RAM.
- A standard RAM has just one access port with an associated address bus and data bus.

A REGISTER FILE WITH THREE ACCESS PORTS - SYMBOL



A REGISTER FILE WITH THREE ACCESS PORTS – LOGIC DIAGRAM



Ex: R3 ← R1 + R2
Read Address A = 01
Read Address B = 10
Write Address C = 11

17-06-2021

INSTRUCTION TYPES

- Data transfer instructions
- Data manipulation instructions
  - Arithmetic instructions
  - Logical and bit manipulation instructions
  - Shift instructions
- Program control instructions

DATA TRANSFER INSTRUCTIONS

- Move data from one place to another without changing the data content in the computer.

- Different data transfers:
  - Memory ↔ processor registers
  - Processor registers ↔ input or output
  - Processor register ↔ processor register

SET OF DATA TRANSFER INSTRUCTIONS

- Load – transfer from memory to a processor register
- Store – transfer from processor register into memory
- Move – transfer from one register to another, transfer between register and memory or between two memory words.
- Exchange – swaps information between two registers or a register and a memory word
- Input – transfer data among registers/memory and input terminal
- Output – transfer data among register/memory and output terminal
- Push – transfer data from register/memory to memory stack
- Pop – transfer data from stack to register/memory

Data Manipulation Instructions

- Perform operations on data and provide the computational capabilities for the computer.
- Arithmetic instructions
  - Increment
  - Decrement
  - Add
  - Subtract
  - Multiply
  - Divide
  - Add with carry
  - Subtract with borrow
  - Negate (2's complement) – change the sign of the operand
  - Absolute – replace operand by its absolute value
  - Arithmetic shift left
  - Arithmetic shift right

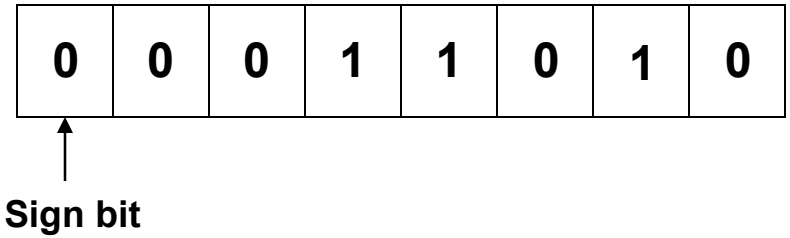
Arithmetic Operations

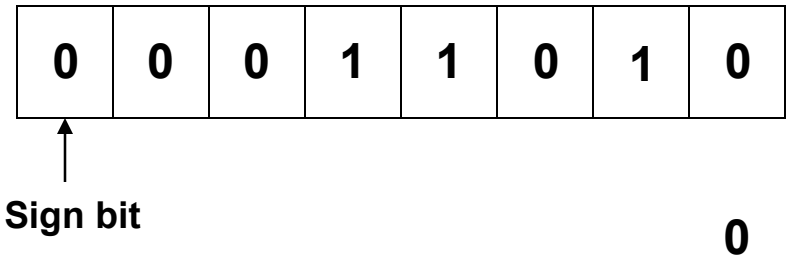
Mnemonic		Description	Byte	Cyc
ADD	A,Rn	Add register to Accumulator	1	1
ADD	A,direct	Add direct byte to Accumulator	2	1
ADD	A,@Ri	Add indirect RAM to Accumulator	1	1
ADD	A,#data	Add immediate data to Accumulator	2	1
ADDC	A,Rn	Add register to Accumulator with Carry	1	1
ADDC	A,direct	Add direct byte to A with carry flag	2	1
ADDC	A,@Ri	Add indirect RAM to A with Carry flag	1	1
ADDC	A,#data	Add immediate data to A with Carry flag	2	1
SUBB	A,Rn	Subtract register from A with Borrow	1	1
SUBB	A,direct	Subtract direct byte from A with Borrow	2	1
SUBB	A,@Ri	Subtract indirect RAM from A with borrow	1	1
SUBB	A,#data	Subtract immed data from A with Borrow	2	1
INC	A	Increment Accumulator	1	1
INC	Rn	Increment register	1	1
INC	direct	Increment direct byte	2	1
INC	@Ri	Increment indirect RAM	1	1
DEC	A	Decrement Accumulator	1	1
DEC	Rn	Decrement register	1	1
DEC	direct	Decrement direct byte	2	1
DEC	@Ri	Decrement indirect RAM	1	1
INC	DPTR	Increment data Pointer	1	2
MUL	AB	Multiply A and B	1	4
DIV	AB	Divide A by B	1	4
DA	A	Decimal Adjust Accumulator	1	1

Logic Operations

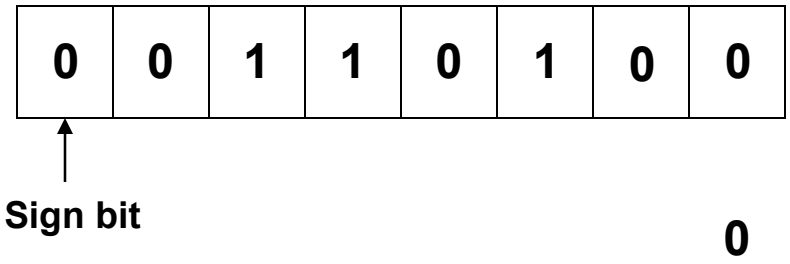
ANL	A,Rn	AND register to accumulator	1	1
ANL	A,direct	AND direct byte to accumulator	2	1
ANL	A,@Ri	AND indirect RAM to accumulator	1	1
ANL	A,#data	AND immediate data to accumulator	2	1
ANL	direct,A	AND accumulator to direct byte	2	1
ANL	direct,#data	AND immediate data to direct byte	3	2
ORL	A,Rn	OR register to accumulator	1	1
ORL	A,direct	OR direct byte to accumulator	2	1
ORL	A,@Ri	OR indirect RAM to accumulator	1	1
ORL	A,#data	OR immediate data to accumulator	2	1
ORL	direct,A	OR accumulator to direct byte	2	1
ORL	direct,#data	OR immediate data to direct byte	3	2
XRL	A,Rn	Exclusive OR register to accumulator	1	1
XRL	A direct	Exclusive OR direct byte to accumulator	2	1
XRL	A,@Ri	Exclusive OR indirect RAM to accumulator	1	1
XRL	A,#data	Exclusive OR immediate data to accumulator	2	1
XRL	direct,A	Exclusive OR accumulator to direct byte	2	1
XRL	direct,#data	Exclusive OR immediate data to direct byte	3	2
CLR	A	Clear accumulator	1	1
CPL	A	Complement accumulator	1	1
RL	A	Rotate accumulator left	1	1
RLC	A	Rotate accumulator left through carry	1	1
RR	A	Rotate accumulator right	1	1
RRC	A	Rotate accumulator right through carry	1	1
SWAP	A	Swap nibbles within the accumulator	1	1

Arithmetic shift left

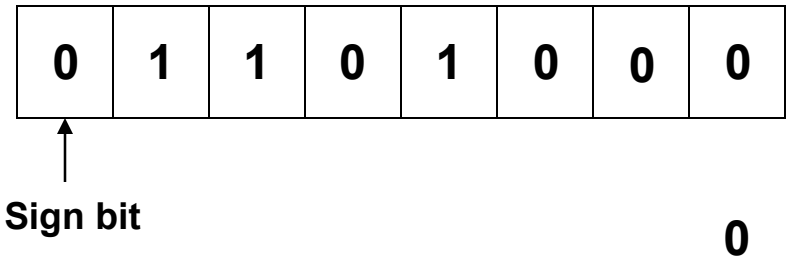




Shift by 1 bit towards left



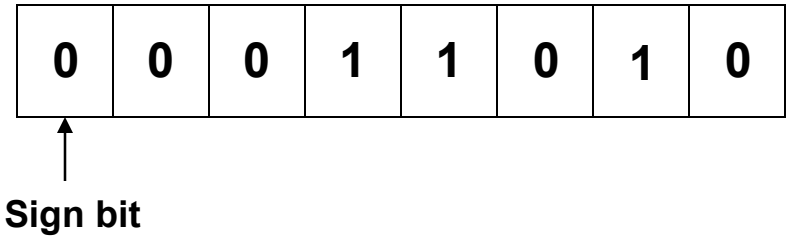
After shifting two times

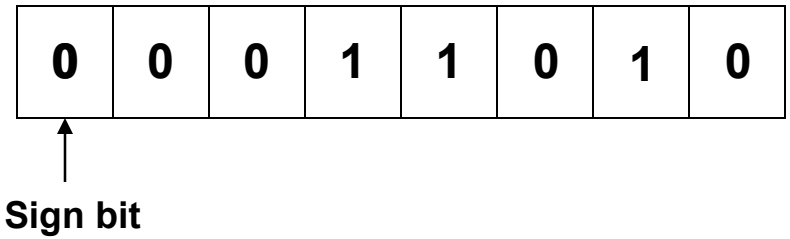


After shifting three times

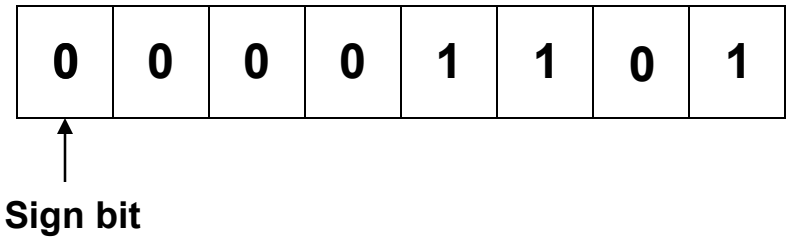
Overflow occurs as sign bit changes

Arithmetic shift Right

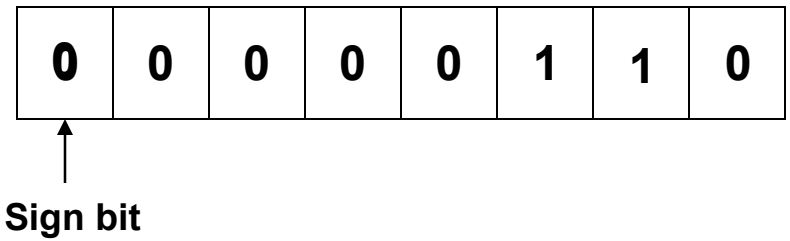




Shift by 1 bit towards right



After shifting two times



After shifting three times

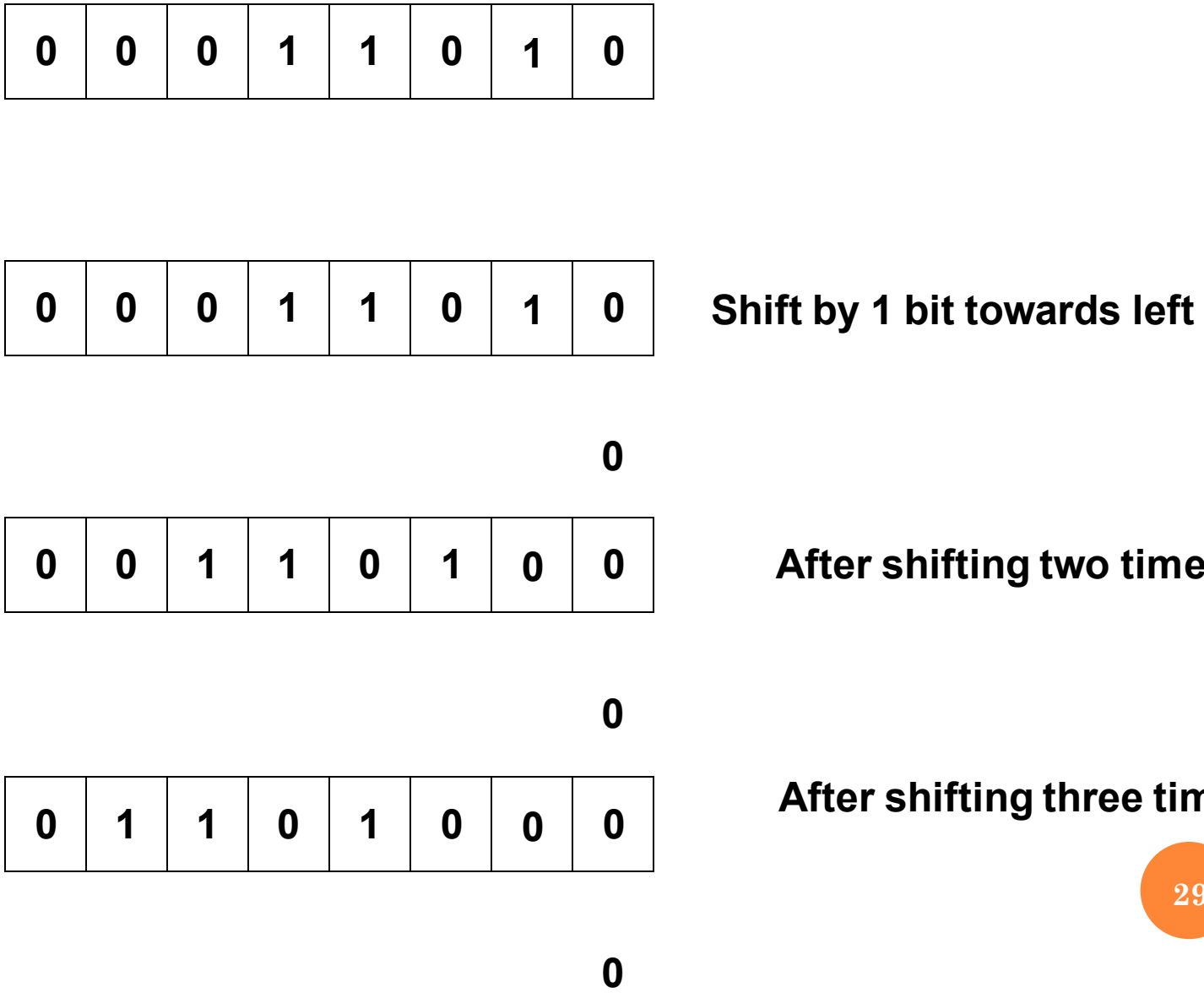
DATA MANIPULATION INSTRUCTIONS

- Logical and Bit manipulation instructions
  - Clear (can also be included in data transfer instruction based on the way the operation is performed – 0's transferred to the destination)
  - Complement
  - AND- to clear a bit
  - OR – set a bit
  - Ex-Or –to complement a bit
  - Clear carry
  - Set carry
  - Complement carry
  - Enable interrupt – flip-flop that controls the interrupt facility is enabled
  - Disable interrupt – flip-flop that controls the interrupt facility is disabled

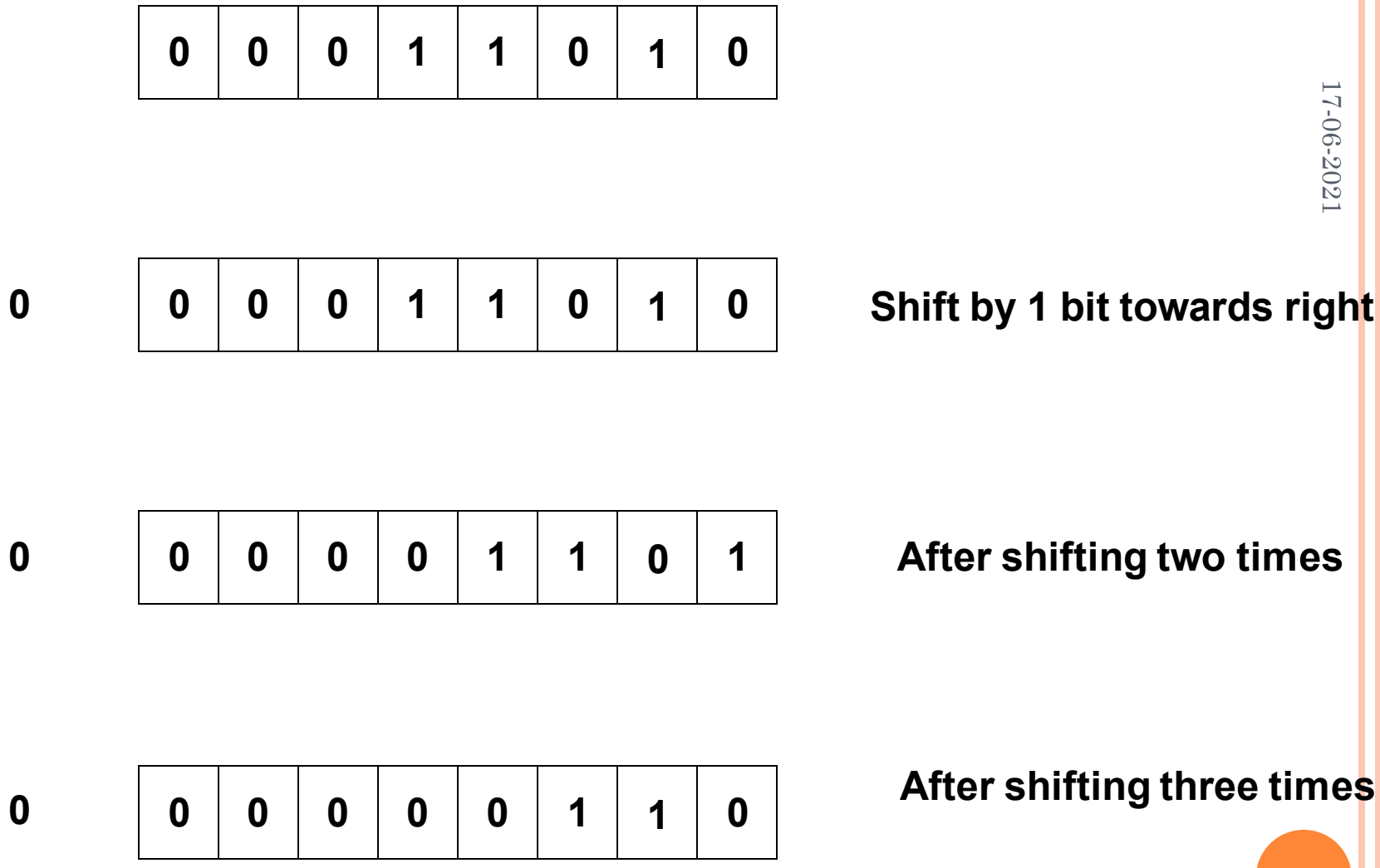
Data manipulation instructions

- Shift Instructions
  - Logical left shift
  - Logical right shift
  - Arithmetic shift left
  - Arithmetic shift right
  - Rotate right
  - Rotate left
  - Rotate right through carry
  - Rotate left through carry

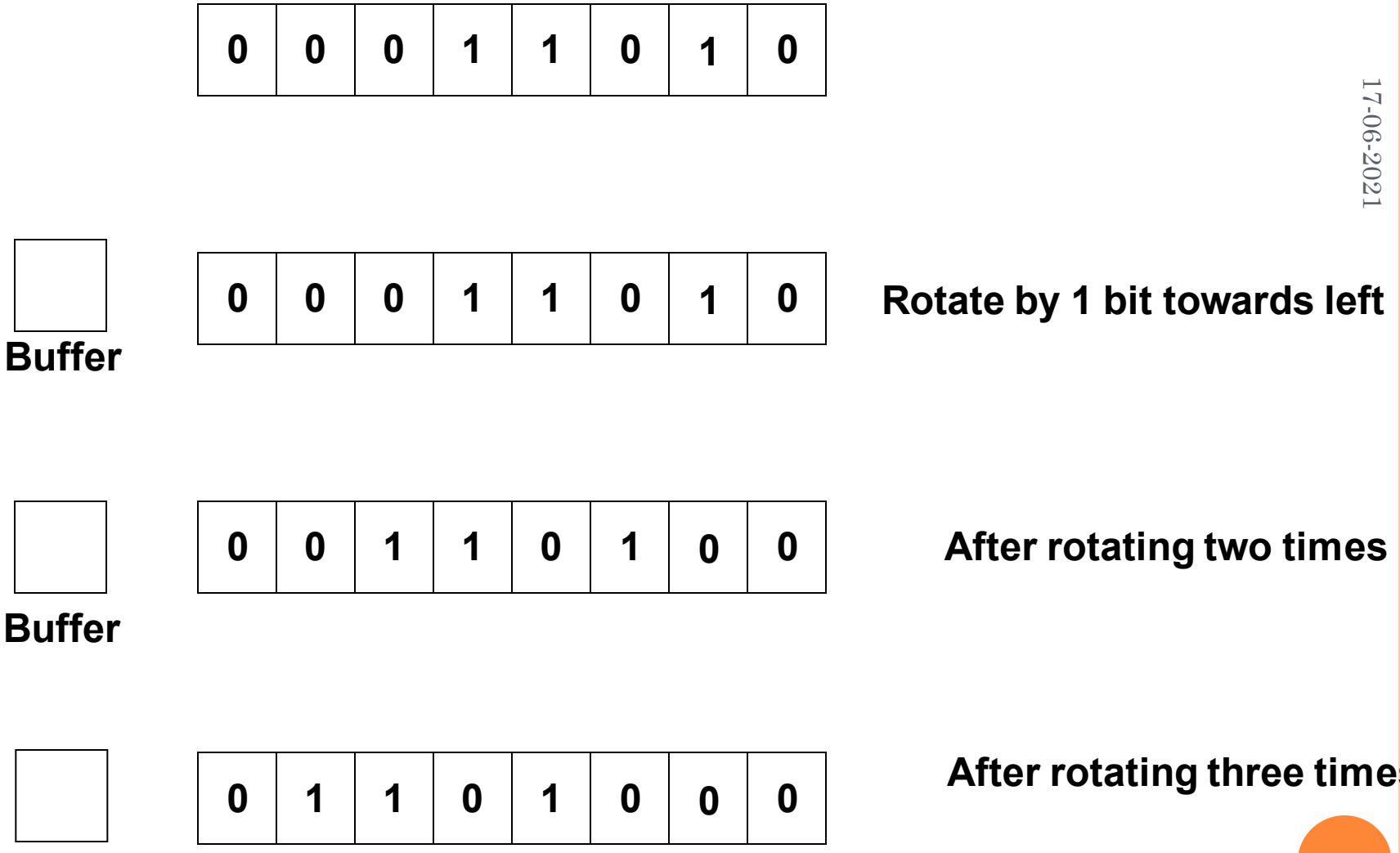
Logical shift left



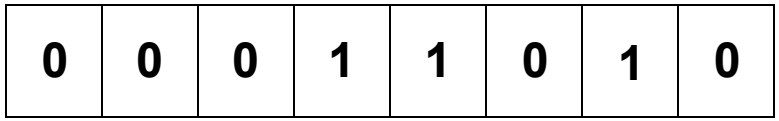
Logical shift Right



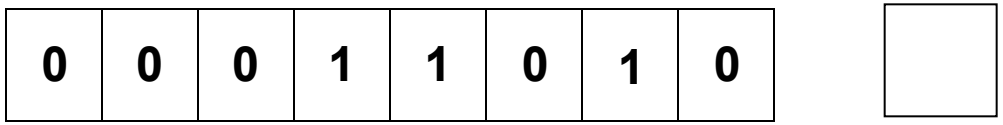
Rotate left


0 0 0 1 1 0 1 0
Buffer
0 0 0 1 1 0 1 0
Rotate by 1 bit towards left
Buffer
0 0 1 1 0 1 0 0
After rotating two times
Buffer
0 1 1 0 1 0 0 0
After rotating three times

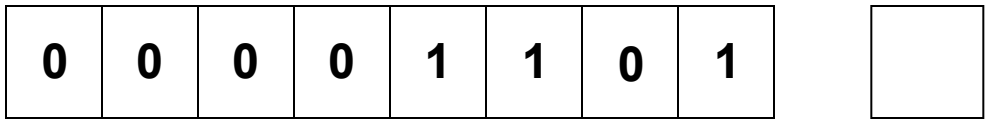
Rotate right



rotate by 1 bit towards right

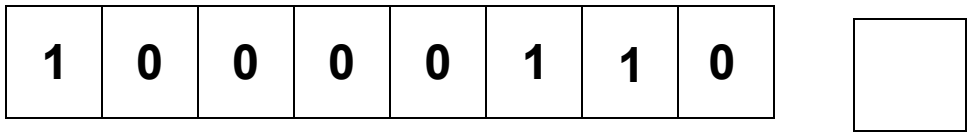


Buffer



After rotating two times

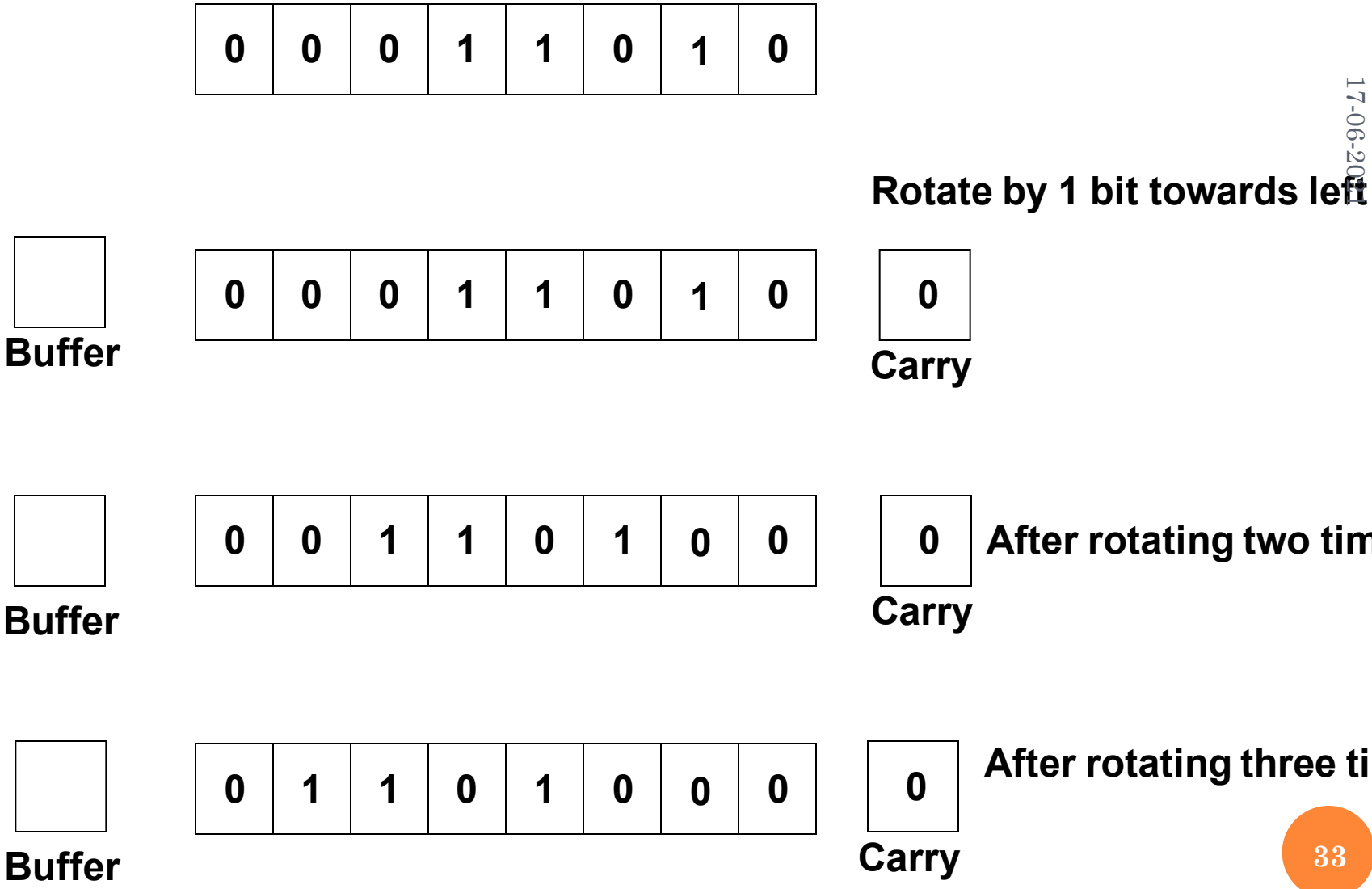
Buffer



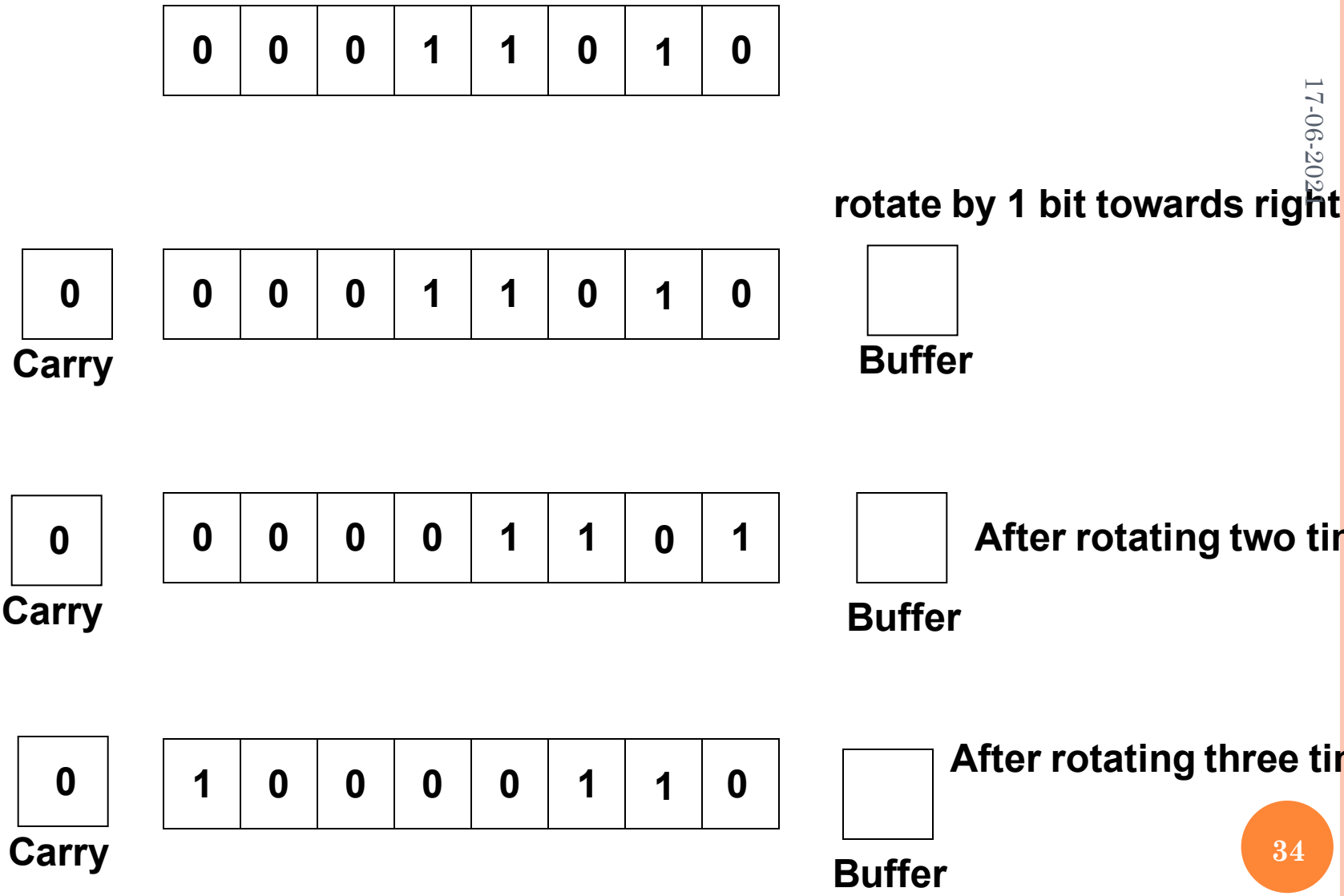
After rotating three times

Buffer

Rotate left through carry



Rotate right through carry



Program control Instructions

- Branch
- Jump
- Skip
- Call
- Return
- Compare (by subtraction)
- Test (by ANDing)

Addressing Modes

The way the operands are chosen during the program execution is dependent on the addressing mode of the instruction.

Different Types

- Implied Addressing Mode
- Immediate Addressing Mode
- Direct Addressing Mode
- Indirect Addressing Mode
- Register Direct Addressing Mode
- Register Indirect Addressing Mode
- Displacement Addressing Mode (combines the direct addressing and register addressing modes)
  - Relative Addressing Mode
  - Indexed Addressing Mode
  - Base Addressing Mode
- Auto Increment and Auto Decrement Addressing Mode

Implied Addressing Mode

- No address field is required
- Operand is implied / implicit
- Ex:
  - Complementing Accumulator
  - Set or Clearing the flag bits (CLC, STC etc.)
- 0 – address instructions in a stack organized computer are implied mode instructions.
- Effective Address (EA) = AC or Stack[SP]
- Ex: Tomorrow, I am on leave (implies that there is no CAO class)
- Come to my cabin (implies to come to 313A-07 SJT)

Immediate Addressing Mode

- Operand is specified in the instruction itself
- Useful for initializing the registers with constant value
- Operand = address field
- Ex: Mov Dx, #0034H
- Advantage: No memory Reference, fast
- Disadvantage: Limited operand magnitude
- Ex: Come to my cabin: 313A-07 SJT

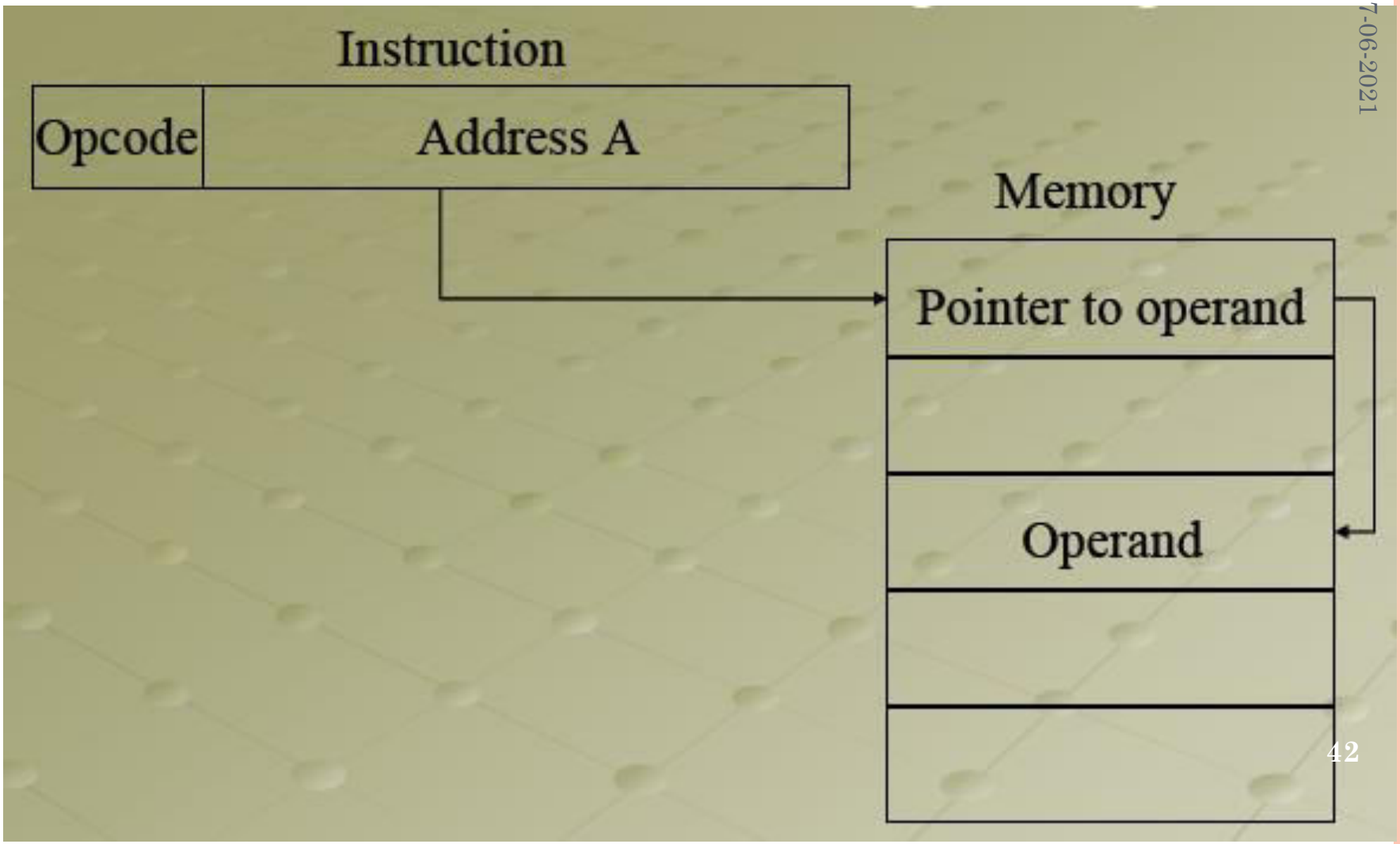
Direct Addressing Mode

- Effective address is the address part of the instruction
- EA (effective address) = A
- Ex:
- Mov CX, [4200]H
- Advantage: Simple memory reference to access data, no additional calculations to work out effective address
- Disadvantage: Limited address space
- Ex: Aashiq, please bring my laptop from my cabin (cabin is known to Aashiq)

Indirect Addressing Mode

- The address field of the instruction gives the address of the effective address of the operand stored in the memory.
- EA = (A)
- Ex: Mov CX, [BX]
- Advantage: Large address space, may be nested, multilevel or cascaded
- Disadvantage: Multiple memory accesses to find the operand, hence slower

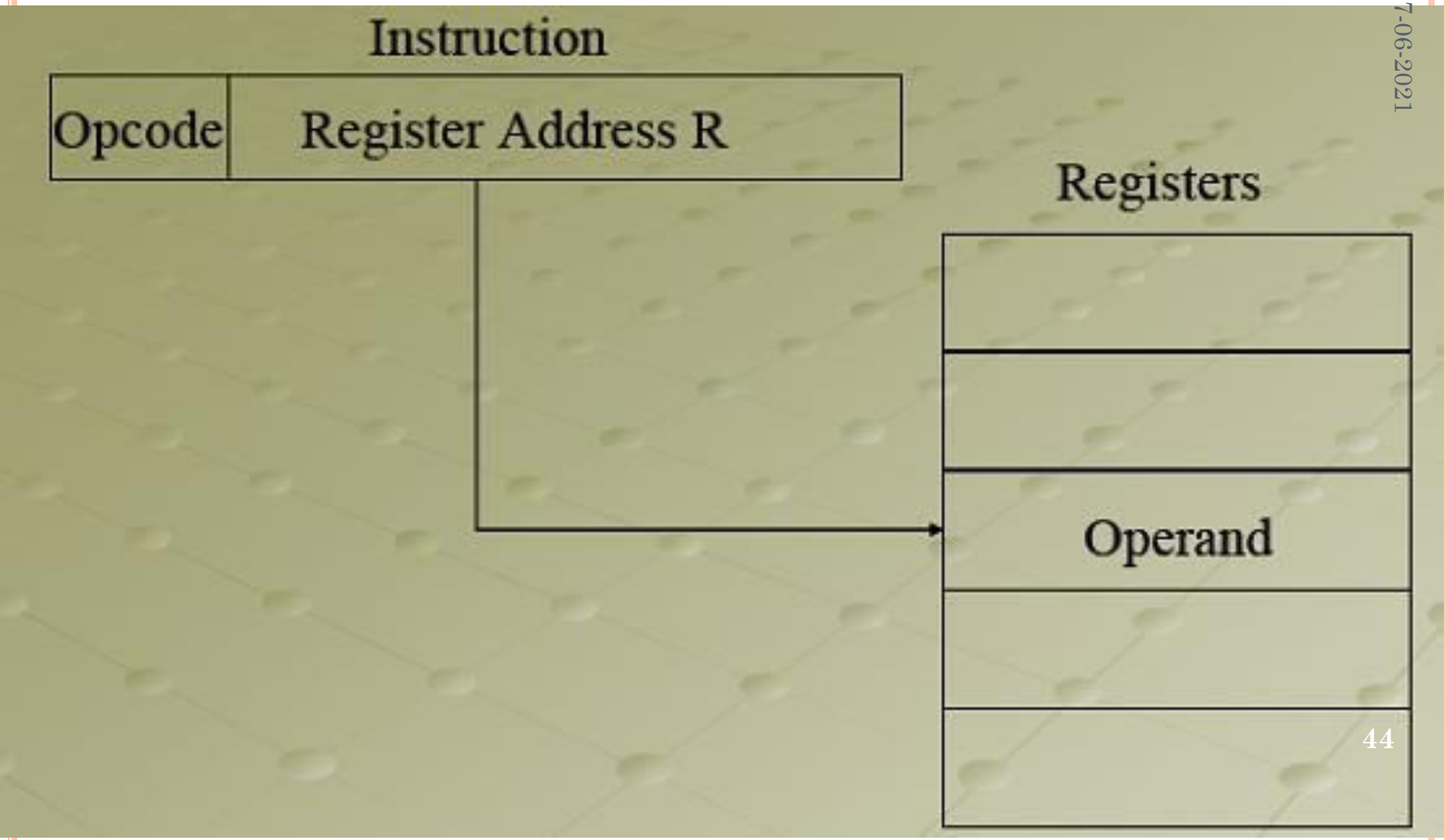
Indirect Addressing Mode Diagram



Register Direct Addressing Mode

- Operand is in the register specified in the address part of the instruction
- EA = R
- Ex: Mov AX, [BX]
- Special case of direct addressing
- Advantage: No memory reference, shorter instructions, faster instruction fetch, very fast execution
- Disadvantage: Limited address space as limited number of registers

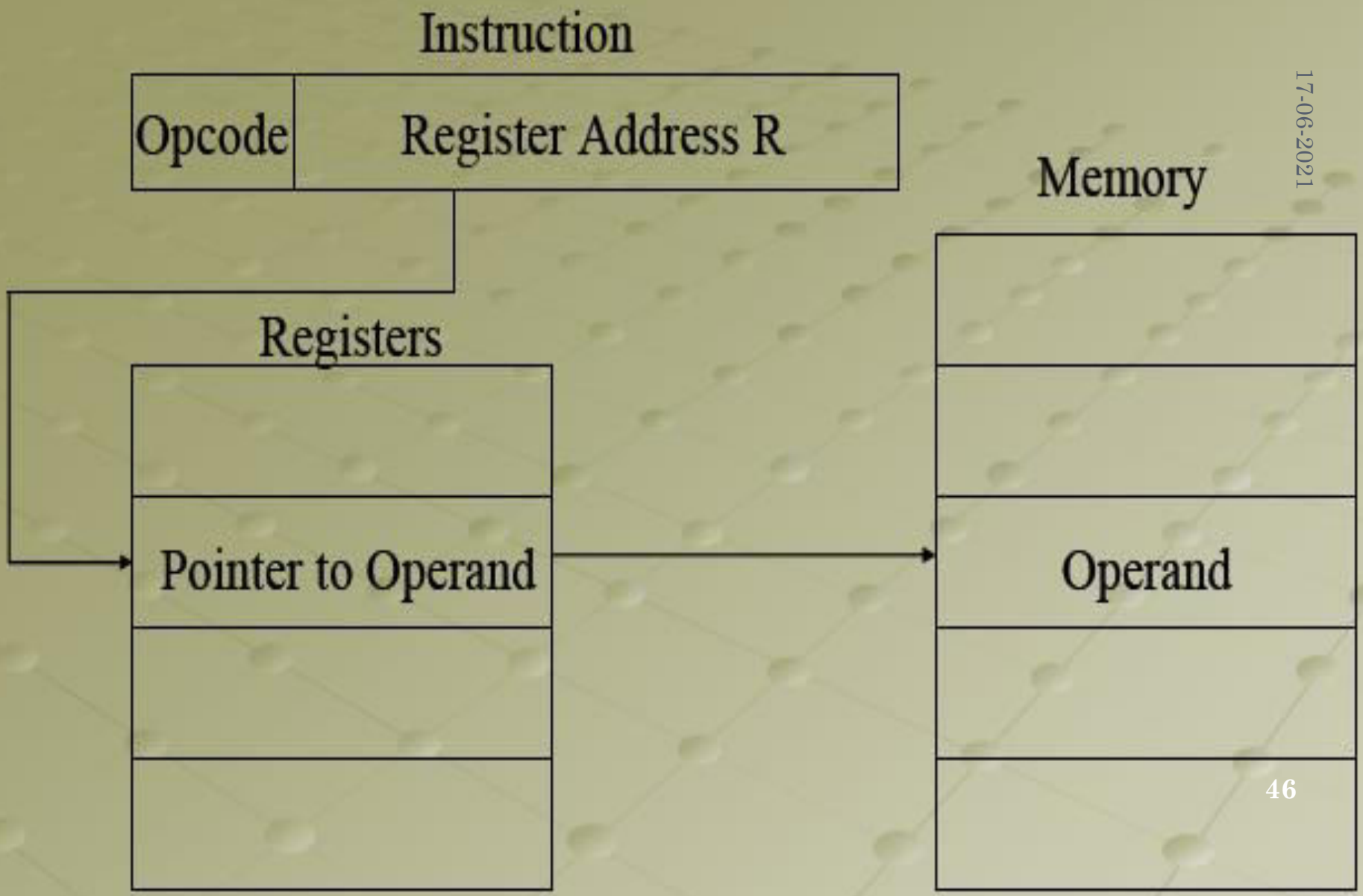
Register Addressing Diagram



Register Indirect Addressing Mode

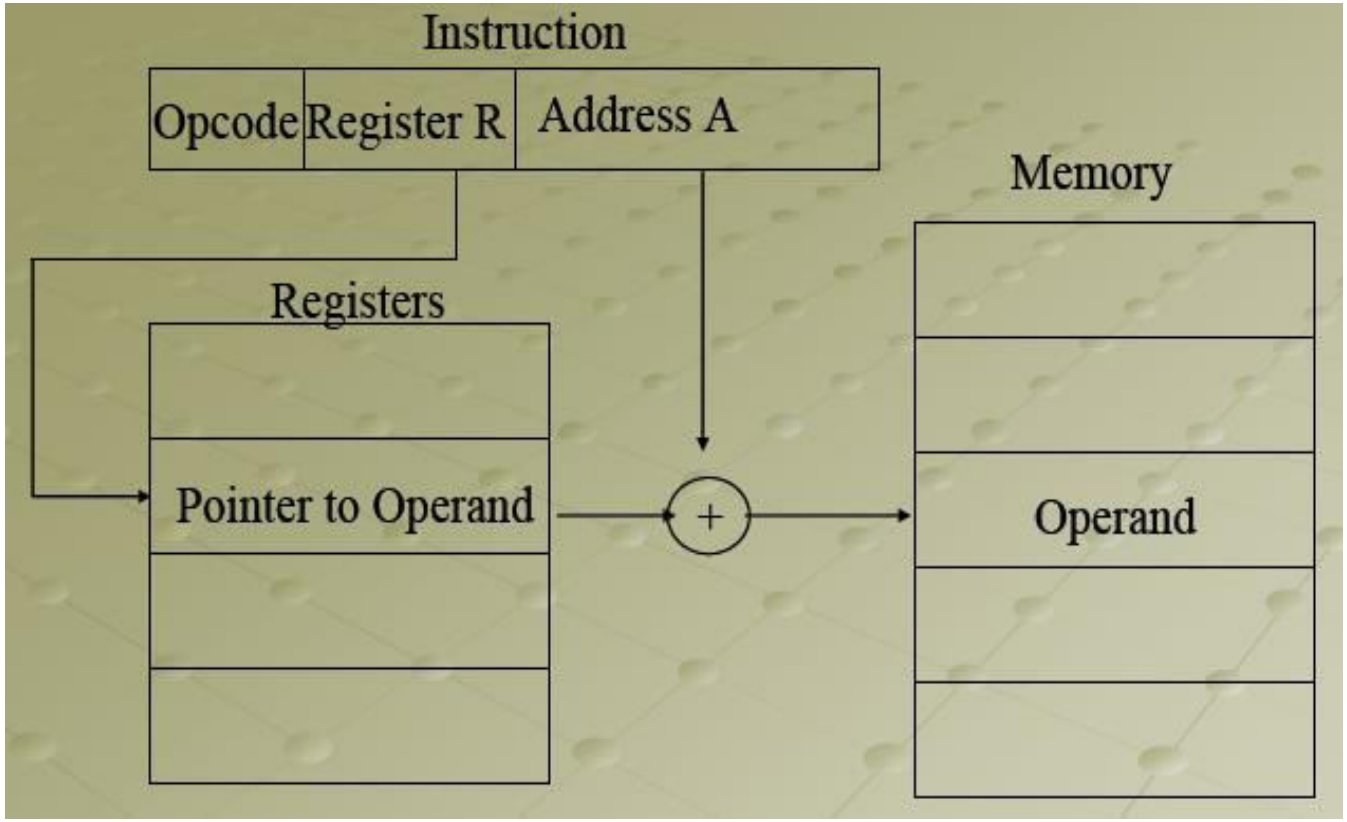
- Address part of the instruction specifies the register which gives the address of the operand in memory
- Special case of indirect addressing
- EA = (R)
- Ex: Mov BX, [DX]
- Advantage: Large address space
- Disadvantage: Extra memory reference

Register Indirect Addressing Diagram



Displacement Addressing Mode

- EA = A + (R)
- Address field holds two values
  - A = Base value
  - R = register that holds displacement
  - Or vice-versa



Relative Addressing Mode

- Version of the displacement addressing
- R = program counter, PC
- Content of PC is added to address part of the instruction to obtain the effective address of the operand
- EA = A + (PC)
- It is often used in branch (conditional and unconditional) instructions, locality of reference and cache usage
- Advantage: Flexibility
- Disadvantage: Complexity

Indexed Addressing Mode

- A holds base address
- R holds displacement, may be explicit or implicit (segment registers in 8086)
- Content of the index register is added to the address part of the instruction to obtain effective address of the operand.
- Used in performing iterative operations
- EA = A + (SI)
- Ex: Mov CX, [SI] 2400H
- Advantage: Flexibility, good for accessing arrays
- Disadvantage: Complexity

Base Register Addressing Mode

- The content of the base register is added to the address part of the instruction to obtain the effective address of the operand.
- Used to facilitate the relocation of programs in memory.
- EA = A + (BX)
- Ex: Mov 2345H [BX], 0AC24H
- Advantage: Flexibility
- Disadvantage: Complexity

Auto Increment and Auto Decrement Addressing Modes

- This addressing mode is used when the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table.
- Ex: Mov AX, (BX)+, Mov AX, -(BX)
- Used mostly in Motorola 680X0 series of computers

Basic Addressing Modes differences :

	Mode	Algorithm	Advantage	Disadvantage
1	Immediate	Operand=1	No memory Reference	Limited operand magnitude
2.	Direct	EA = A	Simple	Limited address space
3	Indirect	EA =(A)	Large Address space	Multiple Memory References
4.	Register	EA = R	No memory Reference	Limited address space
5.	Register Indirect	EA = (R)	Large address space	Extra memory space
6	Displacement	EA = A+(R)	Flexibility	Complexity
7.	Stack	EA= Top of Stack	No memory Reference	Limited Applicability

16 BIT ADDITION

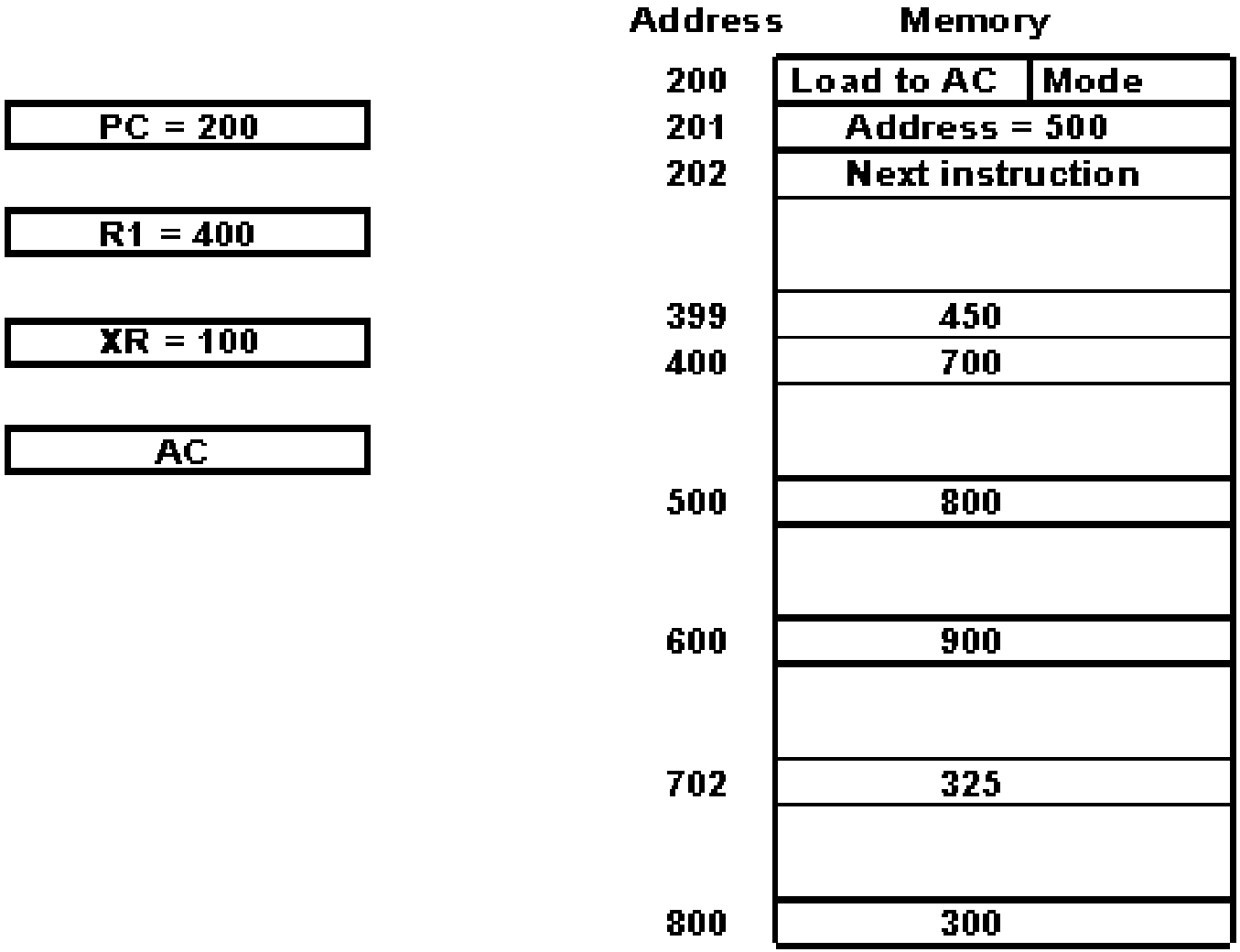
ADDRESS	LABEL	OPCODE	MNEMONICS	OPERAND	COMMENTS
4100		C3	CLR	C	Clear carry
4101		74,04	MOV	A, #DATA1	Move data1 to acc
4103		24,02	ADD	A, #DATA2	Add data2 with acc
4105		90,41,50	MOV	DPTR, #4150h	Move content in 4500 to DPTR.
4108		FO	MOVX	@DPTR, A	Move data to DPTR location
4109		A3	INC	DPTR	Increment DPTR
410A		74,12	MOV	A, #DATA1	Move data1 to acc
410C		34,56	ADDC	A, #DATA2	Add with carry
410E		FO	MOVX	@DPTR, A	Move data to dp location
410F	HERE	80, FE	SJMP	HERE	End of program

ADDRESS	LABEL	OPCODE	MNEMONICS	OPERAND	COMMENTS
4100		16,00	MVI	D, 00	Clear d register
4102		1E, 00	MVI	E, 00	Clear e register
4104		3A, 53,42	LDA	4253	Load data to acc
4107	HUND	FE, 64	CPI	64H	Compare data with acc
4109		DA, 12,41	JC	TEN	Jump on carry to adder
410C		D6, 64	SUI	64	Subtract data from acc
410E		1C	INR	E	Increment e register
410F		C3, 07,41	JMP	HUND	Jump to address
4112	TEN	FE, 0A	CPI	0AH	Compare data with acc
4114		DA, 1D, 41	JC	UNIT	Jump on carry to adder
4117		D6, 0A	SUI	0AH	Subtract data with acc
4119		14	INR	D	Increment d register
411A		C3, 12,41	JMP	TEN	Jump to address
411D	UNIT	4F	MOV	C, A	Move acc to c register
411E		7A	MOV	A, D	Move data to acc
411F		07	RLC		Rotate left without cy
4120		07	RLC		Rotate left without cy
4121		07	RLC		Rotate left without cy
4122		07	RLC		Rotate left without cy
4123		81	ADD	C	Add data to acc
4124		32,50,42	STA	4250	Store the result
4127		7B	MOV	A, E	Move data to acc
4128		**32, 51,42**	**STA**	**4251**	Store the result

Find the Output?

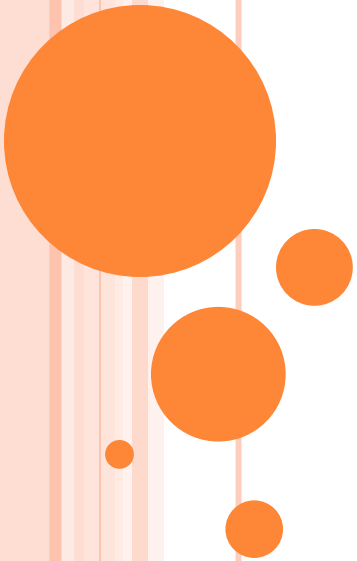
PROBLEMS

1. Find the effective address and the content of AC for the given data.



Addressing Mode	Effective Address		Content of AC
Direct Address	500	AC ← (500)	800
Immediate operand	201	AC ← 500	500
Indirect address	800	AC ← ((500))	300
Relative address	702	AC ← (PC + 500)	325
Indexed address	600	AC ← (XR + 500)	900
Register	-	AC ← R1	400
Register Indirect	400	AC ← (R1)	700
Autoincrement	400	AC ← (R1)+	700
Autodecrement	399	AC ← -(R1)	450

- 2. An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register R1 contains the number 200. Evaluate the effective address if the addressing mode of the instruction is (a) direct; (b) immediate (c) relative (d) register indirect; (e) index with R1 as the index register.



Subroutine Call and Return Mechanisms

SUBROUTINE

- Subroutine is a self-contained sequence of instructions that performs a given computational task.
- It may be called many times at various points in the main program
- When called, branches to 1st line of subroutine and at the end, returned to main program.
- Different names to the instruction that transfers program control to a subroutine
  - Call subroutine
  - Jump to subroutine
  - Branch to subroutine
  - Branch and save address

CONTROL TRANSFER FROM CALLED TO CALLER

- Subroutine instruction – Opcode + starting address of the subroutine
- Execution:
  - PC content (return address) is stored in a temporary location
  - Control is transferred to the subroutine
- when return
  - Transfers the return address from the temporary location to the PC.
  - Control is transferred back to the called routine

LOCATIONS TO STORE THE RETURN ADDRESS

- First memory location of the subroutine
- Fixed location in memory
- Processor registers
- Memory stack – best option
  - Adv: In the case of sequential calls to subroutines. So, the top of the stack always has the return address of the subroutine which to be returned first.

Micro-operations

Call:

SP ← SP – 1 // decrement stack pointer

M[SP] ← PC // push content of PC onto the stack

PC ← effective address /* transfer control to the subroutine */

Return:

PC ← M[SP] // pop stack and transfer to PC

SP ← SP + 1 // increment stack pointer

Recursive subroutines

- Subroutine that calls itself
- If only one register or memory location is used to hold the return address, when subroutine is called recursively, it destroys the previous return address.
- So, stack is the good solution for this problem

ASSIGNMENT

1. Write an assembly language program using IAS instruction set for performing all arithmetic operations (+, -, *, /)
2. Show the register transfer operations using IAS machine registers for division operation.
3. Given the memory contents of the IAS computer shown below. Show the assembly language code for the program, starting at address 08A. Explain what this program does. Given the memory contents of the IAS computer shown below. Show the assembly language code for the program, starting at address 08A. Explain what this program does.

Address	**Contents**
08A	010FA210FB
08B	010FA0F08D
08C	020FA210FB

4. Write an Assembly language programming for the following expressions using IAS computer Instruction set and interpret to the flow of IAS computer [Any one]
   1. A=(B-C)*D
   2. A=B*(C+D)
   3. A=(B-C)/D
   4. A=B/(C+D)
   5. A=-(B+C-D)
   6. A=(B*2)/2

   Make necessary assumptions.
5. On the IAS, describe in English the process that the CPU must undertake to read a value from memory and to write a value to memory in terms of what is put into the MAR, MBR, address bus, data bus, and control bus.
6. Find out the difference between Multicomputer, Multiprocessor, Distributed computer, Multicores

7. A two-word instruction is stored in memory at an address designated by the symbol W. The address field of the instruction (stored at W + 1) is designated by the symbol Y. The operand used during the execution of the instruction is stored at an address symbolized by Z. An index register contains the value X. State how Z is calculated from the other addresses if the addressing mode of the instruction is
   - Direct
   - Indirect
   - Relative
   - Indexed
8. A relative mode branch type of instruction is stored in memory at an address equivalent to decimal 750. The branch is made to an address equivalent to decimal 500. What should be the value of the relative address field of the instruction (in decimal)?

9. How many times does the control unit refer to memory when it fetches and executes an indirect addressing mode instruction if the instruction is (a) a computational type requiring an operand from memory; (b) a branch type.
10. What must the address field of an indexed addressing mode instruction be to make it the same as a register indirect mode instruction?
11. An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register R1 contains the number 200. Evaluate the effective address if the addressing mode of the instruction is (a) direct; (b) immediate (c) relative (d) register indirect; (e) index with R1 as the index register.

12. Assume that in a certain byte-addressed machine all instructions are 32 bits long. Assume the following state of affairs for the machine: Fill in the following table:

Address	Value
PC	100
R0	200
R1	300
100	200
104	300
108	400
200	500
300	600
500	700

Instruction	Addressing mode	Value in R0
Load r0, #200	**Immediate**	
Load r0, 200	**Direct**	
Load r0, (200)	**Indirect**	
Load r0,r1	**Register**	
Load r0, [r1]	**Register Indirect**	
Load r0, -100[r1]	**Based**	
Load r0, 200[PC]	**Relative**	

13. Given the following memory values and a one-address machine with an accumulator, what values do the following instructions load into the accumulator?
    - Word 20 contains 40
    - Word 30 contains 50
    - Word 40 contains 60
    - Word 50 contains 70
        - Load immediate 20
        - Load direct 20
        - Load indirect 20
        - Load immediate 30
        - Load direct 30
        - Load indirect 30

14. Let the address stored in the program counter be designated by the symbol X1. The instruction stored in X1 has the address part (operand reference) X2. The operand needed to execute the instruction is stored in the memory word with address X3. An index register contains the value X4. What is the relationship between these various quantities if the addressing mode of the instruction is (a) direct (b) indirect (c) PC relative (d) indexed?

15. An address field in an instruction contains decimal value 14. where is the corresponding operand located for:
    - Immediate addressing?
    - Direct addressing?
    - Indirect addressing?
    - Register addressing?
    - Register indirect addressing?
16. A PC-relative mode branch instruction is stored in memory at address 620_{10}. The branch is made to location 530_{10}. The address field in the instruction is 10 bits long. What is the binary value in the instruction?

REFERENCES

- William Stallings “Computer Organization and architecture” 8th edition:
  - History of computing :pg 35-56
  - IAS organization : pg 36 -42
  - Instruction fetch & execute : pg 87 – 91
  - Addressing Modes : pg 419 -426

M. M. Mano, Computer System Architecture, Prentice-Hall

Instruction format : pg 255-260

subroutine call & return statement : pg 278 -279

Vincent .P. Heuring, Harry F. Jordan “ Computer System design and Architecture” Pearson, 2nd Edition, 2003

Instruction format calculation