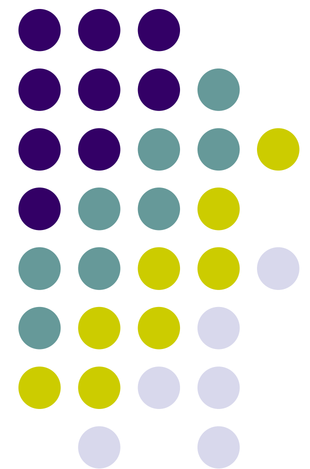
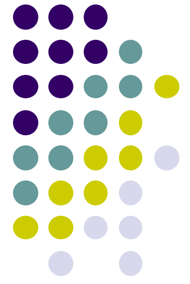
UNIT-1.
Basic Structure of Computer Hardware and Software



EduTechLearners

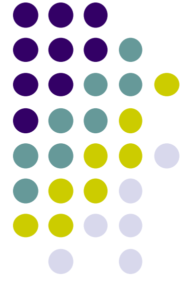
(http://www.edutechlearners.com)

COMPUTER ORGANISATION AND ARCHITECTURE



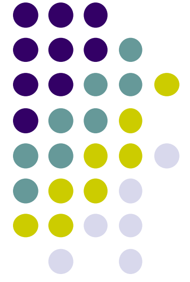
- The components from which computers are built, i.e., computer organization.
- In contrast, computer architecture is the science of integrating those components to achieve a level of functionality and performance.
- It is as if computer organization examines the lumber, bricks, nails, and other building material
- While computer architecture looks at the design of the house.

UNIT-1 CONTENTS



- Evolution of Computer Systems (Historical Prospective)
- Computer Types
- Functional units
- Bus structures
- Register Transfer and Micro-operations
- Information Representation
- Instruction Format and Types
- Addressing modes
- Machine and Assembly Language Programming
- Macros and Subroutines

HISTORICAL PROSPECTIVE

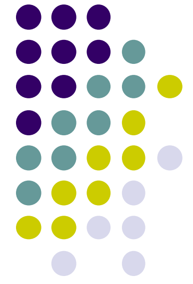


Brief History of Computer Evolution

Two phases:

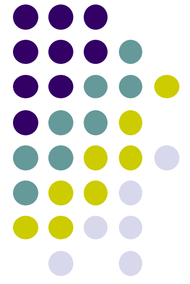
1. before VLSI 1945 – 1978
    - ENIAC
    - IAS
    - IBM
    - PDP-8
2. VLSI 1978 → present day
    - microprocessors AND microcontrollers…

Evolution of Computers
FIRST GENERATION (1945 – 1955)



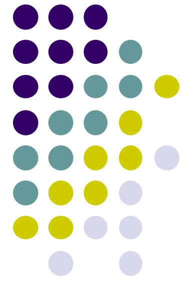
- Program and data reside in the same memory (stored program concepts – John von Neumann)
- ALP was made used to write programs
- Vacuum tubes were used to implement the functions (ALU & CU design)
- Magnetic core and magnetic tape storage devices are used
- Using electronic vacuum tubes, as the switching components

SECOND GENERATION (1955 – 1965)



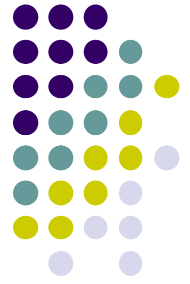
- Transistor were used to design ALU & CU
- HLL is used (FORTRAN)
- To convert HLL to MLL compiler were used
- Separate I/O processor were developed to operate in parallel with CPU, thus improving the performance
- Invention of the transistor which was faster, smaller and required considerably less power to operate

THIRD GENERATION (1965-1975)



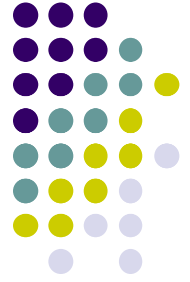
- IC technology improved
- Improved IC technology helped in designing low cost, high speed processor and memory modules
- Multiprogramming, pipelining concepts were incorporated
- DOS allowed efficient and coordinate operation of computer system with multiple users
- Cache and virtual memory concepts were developed
- More than one circuit on a single silicon chip became available

FOURTH GENERATION (1975-1985)



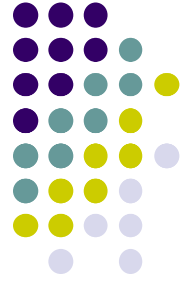
- CPU – Termed as microprocessor
- INTEL, MOTOROLA, TEXAS,NATIONAL semiconductors started developing microprocessor
- Workstations, microprocessor (PC) & Notebook computers were developed
- Interconnection of different computer for better communication LAN,MAN,WAN
- Computational speed increased by 1000 times
- Specialized processors like Digital Signal Processor were also developed

BEYOND THE FOURTH GENERATION (1985 – TILL DATE)



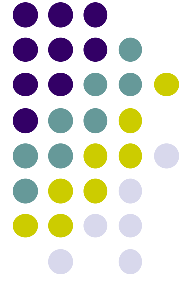
- E-Commerce, E- banking, home office
- ARM, AMD, INTEL, MOTOROLA
- High speed processor - GHz speed
- Because of submicron IC technology lot of added features in small size

COMPUTER TYPES



Computers are classified based on the parameters like

- Speed of operation
- Cost
- Computational power
- Type of application

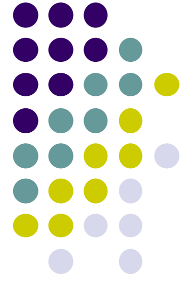


DESK TOP COMPUTER

- Processing & storage units, visual display &audio units, keyboards
- Storage media-Hard disks, CD-ROMs
- Eg: Personal computers which is used in homes and offices
- Advantage: Cost effective, easy to operate, suitable for general purpose educational or business application

NOTEBOOK COMPUTER

- Compact form of personal computer (laptop)
- Advantage is portability

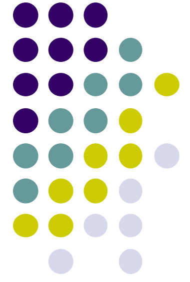


WORK STATIONS

- More computational power than PC
- Costlier
- Used to solve complex problems which arises in engineering application (graphics, CAD/CAM etc)

ENTERPRISE SYSTEM (MAINFRAME)

- More computational power
- Larger storage capacity
- Used for business data processing in large organization
- Commonly referred as servers or super computers



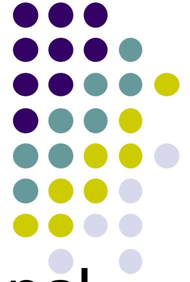
SERVER SYSTEM

• Supports large volumes of data which frequently need to be accessed or to be modified
•Supports request response operation

SUPER COMPUTERS

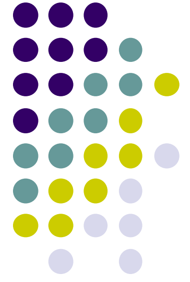
•Faster than mainframes
•Helps in calculating large scale numerical and algorithm calculation in short span of time
•Used for aircraft design and testing, military application and weather forecasting

HANDHELD



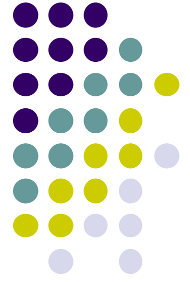
- Also called a PDA (Personal Digital Assistant).
- A computer that fits into a pocket, runs on batteries, and is used while holding the unit in your hand.
- Typically used as an appointment book, address book, calculator, and notepad.
- Can be synchronized with a personal microcomputer as a backup.

Basic Terminology

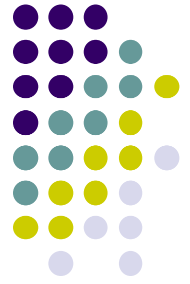


- **Computer**
  - A device that accepts input, processes data, stores data, and produces output, all according to a series of stored instructions.
- **Hardware**
  - Includes the electronic and mechanical devices that process the data; refers to the computer as well as peripheral devices.
- **Software**
  - A computer program that tells the computer how to perform particular tasks.
- **Network**
  - Two or more computers and other devices that are connected, for the purpose of sharing data and programs.
- **Peripheral devices**
  - Used to expand the computer's input, output and storage capabilities.

Basic Terminology



- **Input**
  - Whatever is put into a computer system.
- **Data**
  - Refers to the symbols that represent facts, objects, or ideas.
- **Information**
  - The results of the computer storing data as bits and bytes; the words, numbers, sounds, and graphics.
- **Output**
  - Consists of the processing results produced by a computer.
- **Processing**
  - Manipulation of the data in many ways.
- **Memory**
  - Area of the computer that temporarily holds data waiting to be processed, stored, or output.
- **Storage**
  - Area of the computer that holds data on a permanent basis when it is not immediately needed for processing.



Basic Terminology

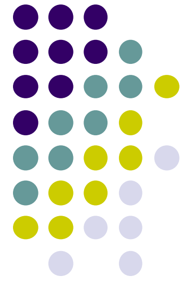
•**Assembly language program (ALP**) – Programs are written using mnemonics

•**Mnemonic** – Instruction will be in the form of English like form

•**Assembler** – is a software which converts ALP to MLL (Machine Level Language)

•**HLL (High Level Language)** – Programs are written using English like statements

•**Compile**r - Convert HLL to MLL, does this job by reading source program at once



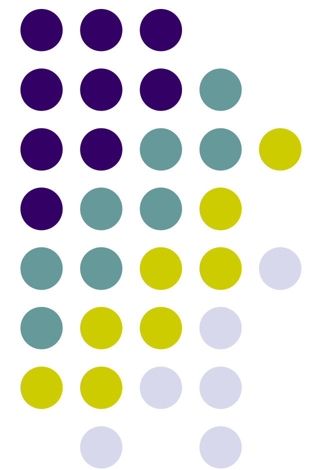
Basic Terminology

•**Interpreter** – Converts HLL to MLL, does this job statement by statement

•**System software** – Program routines which aid the user in the execution of programs eg: Assemblers, Compilers

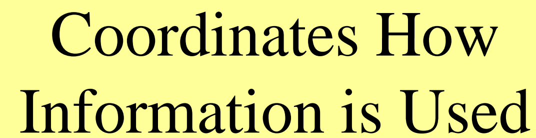
•**Operating system** – Collection of routines responsible for controlling and coordinating all the activities in a computer system

Functional Units

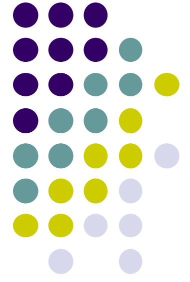


Function

- ALL computer functions are:
  - Data PROCESSING
  - Data STORAGE
  - Data MOVEMENT
  - CONTROL



- NOTHING ELSE!



Functional Units

Input

Output

I/O

Memory

Arithmetic
and
logic

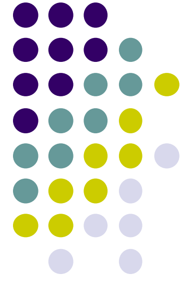
Control

Processor

System Interconnections

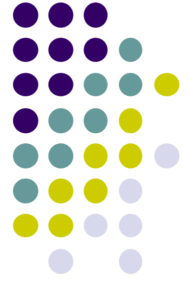
Figure 1.1. Basic functional units of a computer.

INPUT UNIT



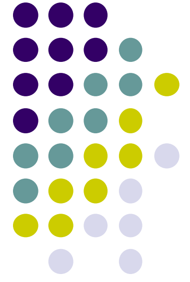
- Computer accepts the coded information through input unit.
- It has the capability of reading the instruction & data to be processed.
- Converts the external world data to a binary format, which can be understood by CPU.
- Eg: Keyboard, Mouse, Joystick etc

Functional Unit(I/O)



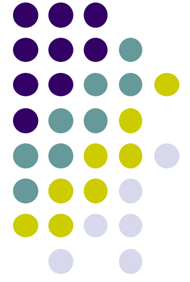
- A computer handles *two types of information* :
- **Instruction** :
  - An instruction *controls the transfer of information* between a computer and its I/O devices and also within the computer.
  - A *list of instructions* that performs a task is called a *program*, which is stored in the memory.
  - To execute a program, *computer fetches the instructions one by one* and specifies the arithmetic and logical operations to be performed which are needed for the desired program.
  - A computer is completely controlled by *the stored programs except any external interrupts* comes from any I/O device.

Functional Unit(I/O)



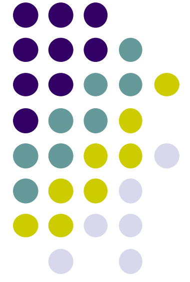
- **Data** :
  - Data is a kind of information which is *used as an operand* for a program.
  - So, data can be any *number or character*.
  - Even, a list of instructions, means *an entire program can be data* if it is processed by another high-level program.
  - In such case, that data is called *source program*.
- The most well-known input device is the keyboard, beside this, there are many other kinds of input devices are available, i.e., mouses, joysticks etc.

OUTPUT UNIT



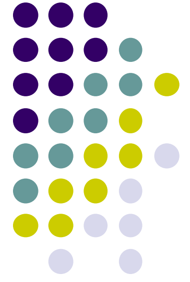
- Converts the binary format data to a format that a common man can understand
- Displays the processed results.
- Eg: Monitor, Printer, LCD, LED etc

MEMORY UNIT



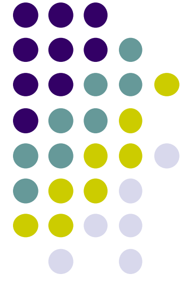
- Composed of large array of bytes.
- Store programs and data .
- **Parts of the memory subsystem**
  - Fetch/store controller
    - Fetch: Retrieve a value from memory
    - Store: Store a value into memory
  - Memory address register (MAR)
  - Memory data register (MDR)
  - Memory cells with decoder(s) to select individual cells

Types of Memory Unit



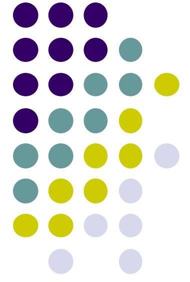
- **Primary storage**
  - Fast and Direct Access
  - Programs must be stored in memory while they are being executed.
  - Large number of semiconductor storage cells.
  - RAM and ROM
- **Secondary storage**
  - used for bulk storage or mass storage.
  - Indirect Access and slow.
  - Magnetic Harddisks,CDs. Etc.

CACHE MEMORY



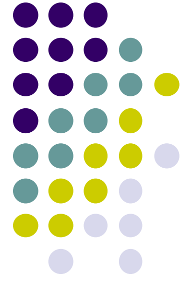
- Memory access is much slower than processing time.
- Faster memory is too expensive to use for all memory cells.
- Small size, fast memory just for values currently in use speeds computing time.
- System Performance improved using this buffer memory.

Arithmetic and Logic Unit (ALU)



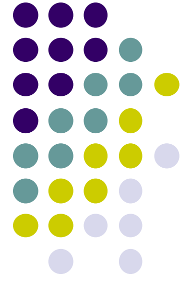
- Most computer operations are executed in ALU of the processor.
- Load the operands into memory – bring them to the processor – perform operation in ALU – store the result back to memory or retain in the processor.
- Registers
- Fast control of ALU

Arithmetic and Logic Unit (ALU)

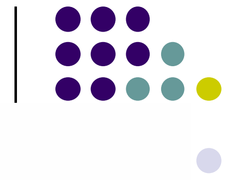


- Actual computations are performed
- Primitive operation circuits
  - Arithmetic (ADD)
  - Comparison (CE)
  - Logic (AND)
- Data inputs and results stored in registers
- Multiplexor selects desired output

Arithmetic and Logic Unit (continued)



- ALU process
  - Values for operations copied into ALU's input register locations
  - All circuits compute results for those inputs
  - Multiplexor selects the one desired result from all values
  - Result value copied to desired result register



a b

ALU

a + b

a – b

a = b

a · b

Line 0 (00)

Line 1 (01)

Line 2 (10)

Line 3 (11)

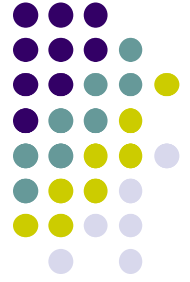
Multiplexor circuit

Output

Selector lines

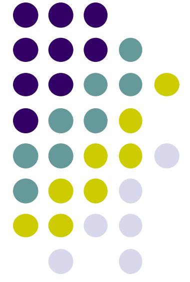
Using a Multiplexor Circuit to Select the Proper ALU Result

The Control Unit



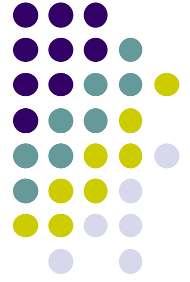
- Manages stored program execution.
- The timing signals that govern the I/O transfers are also generated by the control unit.
- Task
  - Fetch from memory the next instruction to be executed
  - Decode it: Determine what is to be done
  - Execute it: Issue appropriate command to ALU, memory, and I/O controllers

Overall operation of a computer



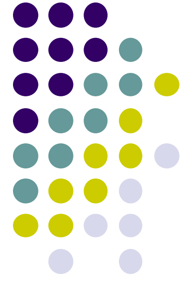
- The total operation of the computer is executed as
  1. Computer *accepts programs and data* through input unit.
  2. *Information is also fetched* in the processor from memory.
  3. Then *information is processed* and the *operation is executed.*

Overall operation of a computer



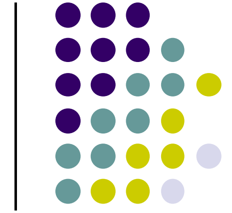
4. Processed information is passed from *output unit .*
5. All these activities described above are sequentially done under the *control signal from the control unit*.

COMPUTER ARCHITECTURE:Bus Structures

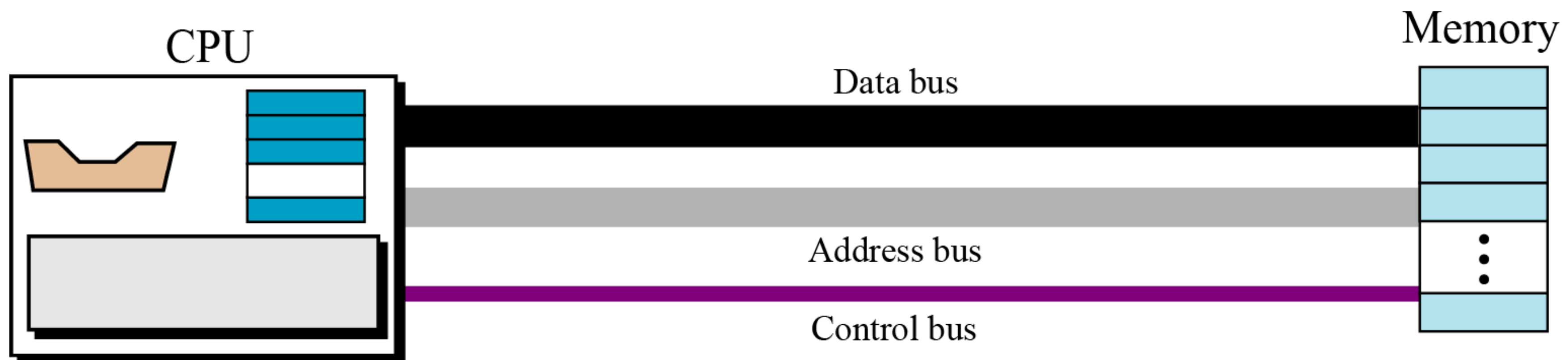


- There are many ways to connect different parts inside a computer together.
- A group of lines or wires that serves as a connecting path for several devices is called a *bus*.
- Address/data/control

BUS STRUCTURE
Connecting CPU and memory

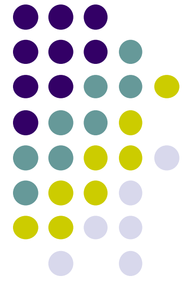


The CPU and memory are normally connected by three groups of connections, each called a bus: *data bus*, *address bus* and *control bus*



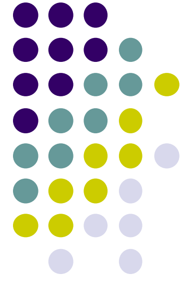
Connecting CPU and memory using three buses

INTERRUPT



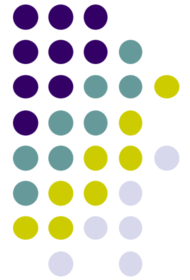
- An interrupt is a request from I/O device for service by processor
- Processor provides requested service by executing interrupt service routine (ISR)
- Contents of PC, general registers, and some control information are stored in memory .
- When ISR completed, processor restored, so that interrupted program may continue

REGISTER TRANSFER AND MICROOPERATIONS



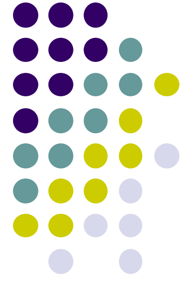
- Register Transfer Language
- Register Transfer
- Bus and Memory Transfers
- Arithmetic Micro-operations
- Logic Micro-operations
- Shift Micro-operations
- Arithmetic Logic Shift Unit

SIMPLE DIGITAL SYSTEMS



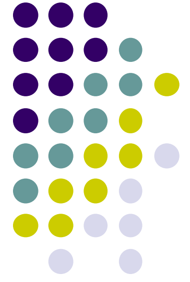
- Combinational and sequential circuits can be used to create simple digital systems.
- These are the low-level building blocks of a digital computer.
- Simple digital systems are frequently characterized in terms of
  - the registers they contain, and
  - the operations that they perform.

MICROOPERATIONS (1)

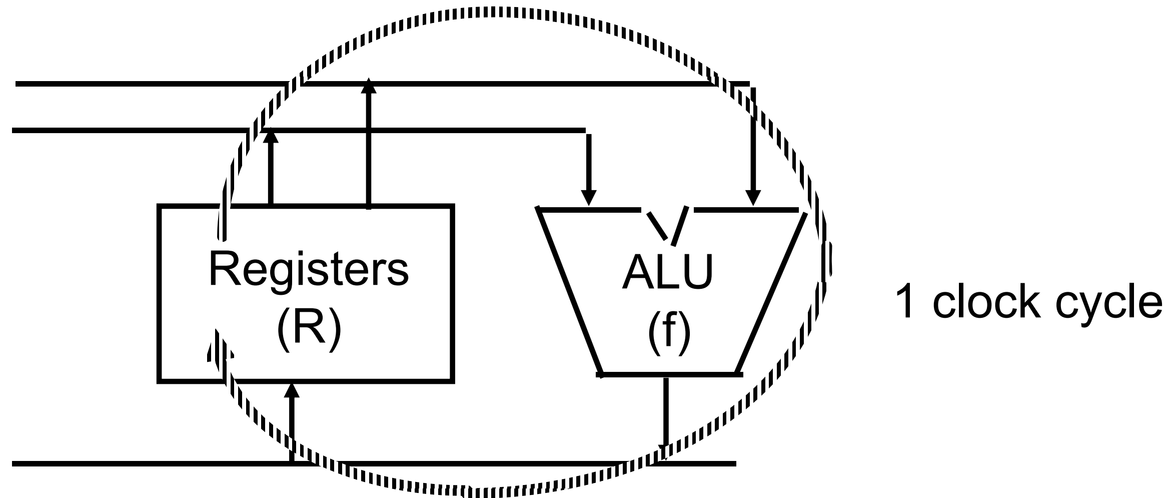


- The operations on the data in registers are called micro-operations.
- The functions built into registers are examples of micro-operations
  - Shift
  - Load
  - Clear
  - Increment

MICRO-OPERATION (2)



An elementary operation performed (during one clock pulse), on the information stored in one or more registers

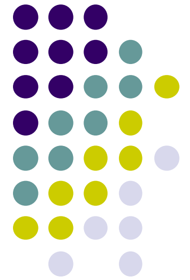


1 clock cycle

$R \leftarrow f(R, R)$

f: shift, load, clear, increment, add, subtract, complement, and, or, xor, …

ORGANIZATION OF A DIGITAL SYSTEM



- Definition of the (internal) organization of a computer

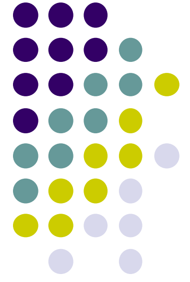
  - Set of registers and their functions

  - Microoperations set

    Set of allowable microoperations provided by the organization of the computer

  - Control signals that initiate the sequence of microoperations (to perform the functions)

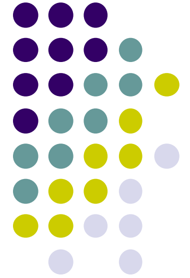
REGISTER TRANSFER LEVEL



- Viewing a computer, or any digital system, in this way is called the register transfer level

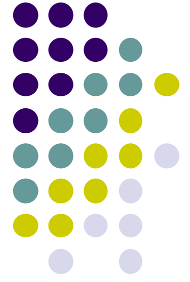
- This is because we're focusing on
  - The system's registers
  - The data transformations in them, and
  - The data transfers between them.

REGISTER TRANSFER LANGUAGE



- Rather than specifying a digital system in words, a specific notation is used, *register transfer language*
- For any function of the computer, the register transfer language can be used to describe the (sequence of) microoperations
- Register transfer language
  - A symbolic language
  - A convenient tool for describing the internal organization of digital computers
  - Can also be used to facilitate the design process of digital systems.

DESIGNATION OF REGISTERS

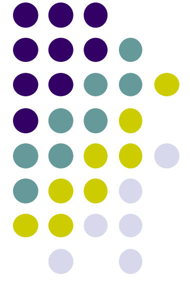


- Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR)
- Often the names indicate function:
  - MAR - memory address register
  - PC - program counter
  - IR - instruction register

- Registers and their contents can be viewed and represented in *various ways*
  - A register can be viewed as a single entity:

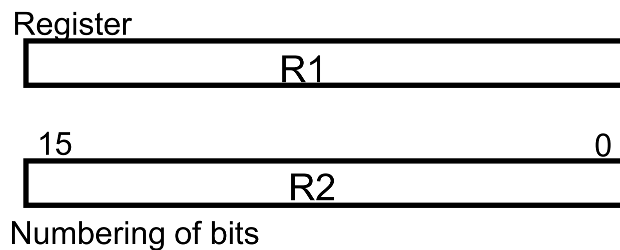
  - Registers may also be represented showing the bits of data they contain

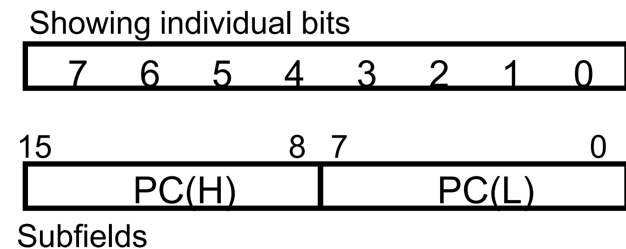
DESIGNATION OF REGISTERS



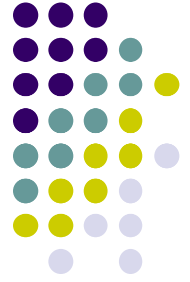
- Designation of a register
  - a register
  - portion of a register
  - a bit of a register

- Common ways of drawing the block diagram of a register





REGISTER TRANSFER

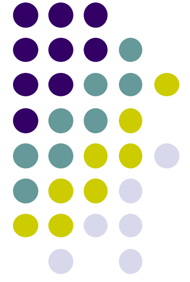


- Copying the contents of one register to another is a register transfer
- A register transfer is indicated as

  $R2 \leftarrow R1$

  - In this case the contents of register R1 are copied (loaded) into register R2
  - A simultaneous transfer of all bits from the source R1 to the destination register R2, during one clock pulse
  - Note that this is a non-destructive; i.e. the contents of R1 are not altered by copying (loading) them to R2

REGISTER TRANSFER



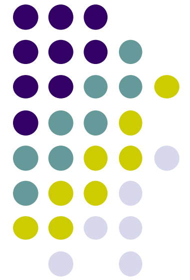
- A register transfer such as

R3 $\leftarrow$ R5

Implies that the digital system has

- the data lines from the source register (R5) to the destination register (R3)
- Parallel load in the destination register (R3)
- Control lines to perform the action

CONTROL FUNCTIONS

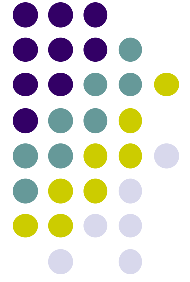


- Often actions need to only occur if a certain condition is true
- This is similar to an “if” statement in a programming language
- In digital systems, this is often done via a *control signal*, called a *control function*
  - If the signal is 1, the action takes place
- This is represented as:

$P: R2 \leftarrow R1$

Which means “if P = 1, then load the contents of register R1 into register R2”, i.e., if (P = 1) then ($R2 \leftarrow R1$)

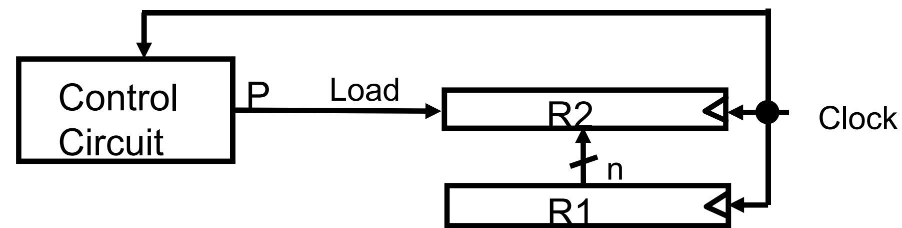
HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS



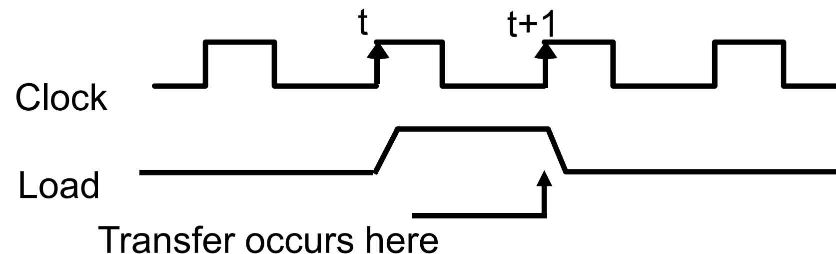
Implementation of controlled transfer

P: R2 ← R1

Block diagram

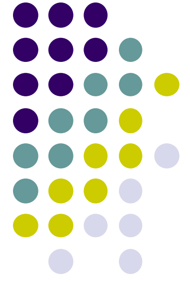


Timing diagram



- The same clock controls the circuits that generate the control function and the destination register
- Registers are assumed to use *positive-edge-triggered* flip-flops

SIMULTANEOUS OPERATIONS

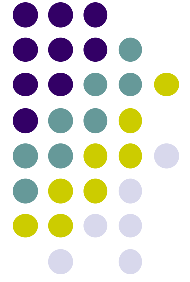


- If two or more operations are to occur simultaneously, they are separated with commas

P: $R3 \leftarrow R5, MAR \leftarrow IR$

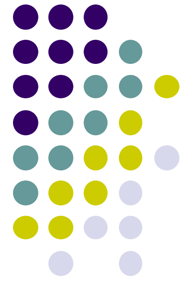
- Here, if the control function P = 1, load the contents of R5 into R3, and at the same time (clock), load the contents of register IR into register MAR

BASIC SYMBOLS FOR REGISTER TRANSFERS



Symbols	Description	Examples
Capital letters & numerals	Denotes a register	MAR, R2
Parentheses ()	Denotes a part of a register	R2(0-7), R2(L)
Arrow $\leftarrow$	Denotes transfer of information	R2 $\leftarrow$ R1
Colon :	Denotes termination of control function	P:
Comma ,	Separates two micro-operations	A $\leftarrow$ B, B $\leftarrow$ A

CONNECTING REGISTERS

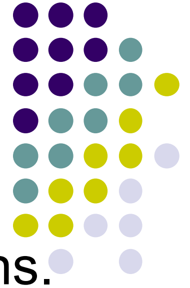


- In a digital system with many registers, it is impractical to have data and control lines to directly allow each register to be loaded with the contents of every possible other registers

- To completely connect n registers → n(n-1) lines
- O(n^2) cost
  - This is not a realistic approach to use in a large digital system

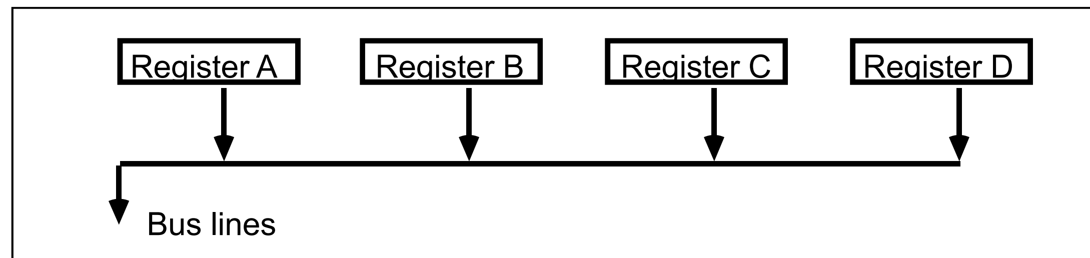
- Instead, take a different approach
- Have one centralized set of circuits for data transfer – the bus
- Have control circuits to select which register is the source, and which is the destination

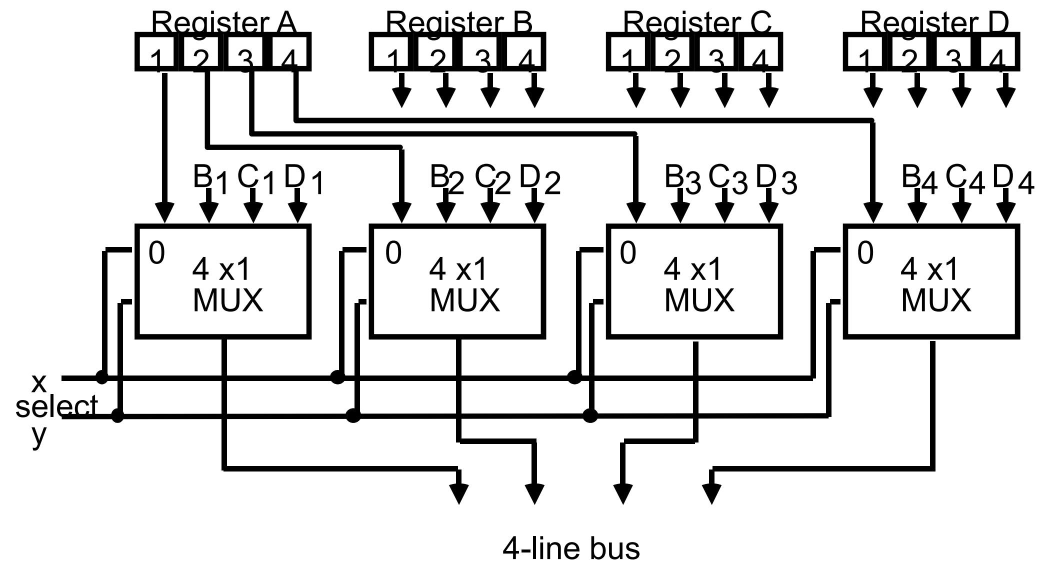
BUS AND BUS TRANSFER



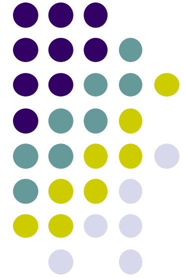
Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

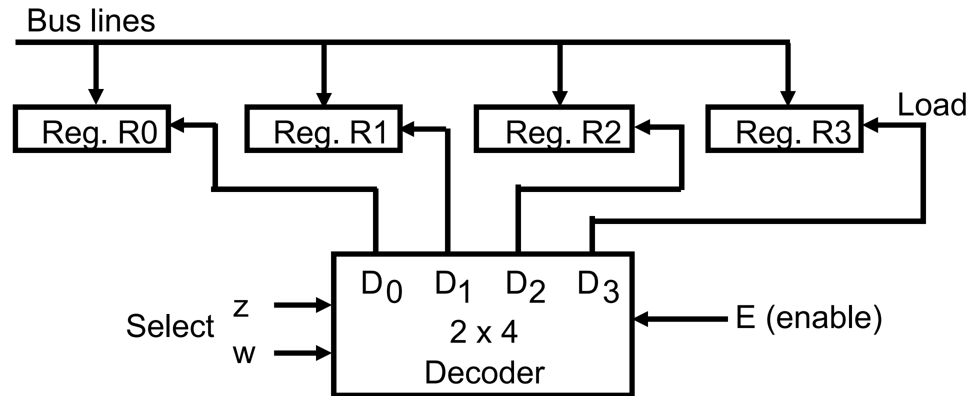
From a register to bus: BUS $\leftarrow$ R





TRANSFER FROM BUS TO A DESTINATION REGISTER

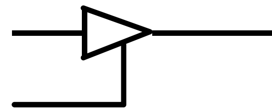




Three-State Bus Buffers

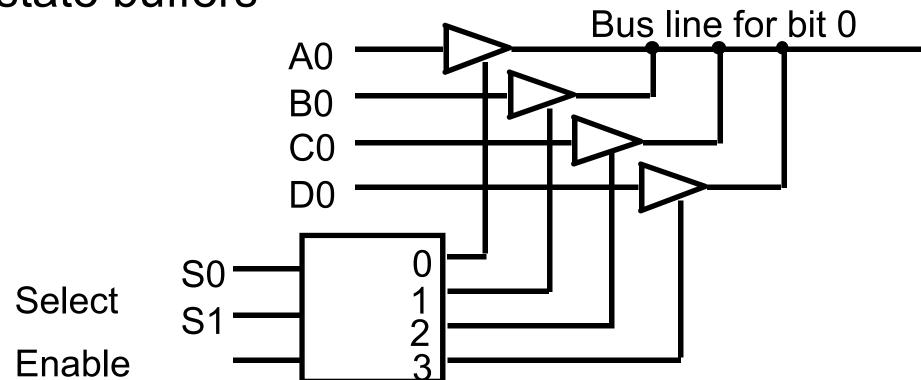
Normal input A

Control input C

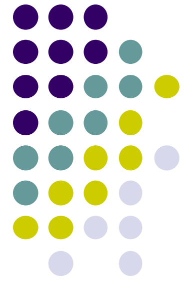


Output Y=A if C=1
High-impedence if C=0

Bus line with three-state buffers



BUS TRANSFER IN RTL



- Depending on whether the bus is to be mentioned explicitly or not, register transfer can be indicated as either

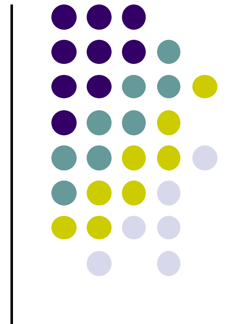
$$R2 \leftarrow R1$$

or

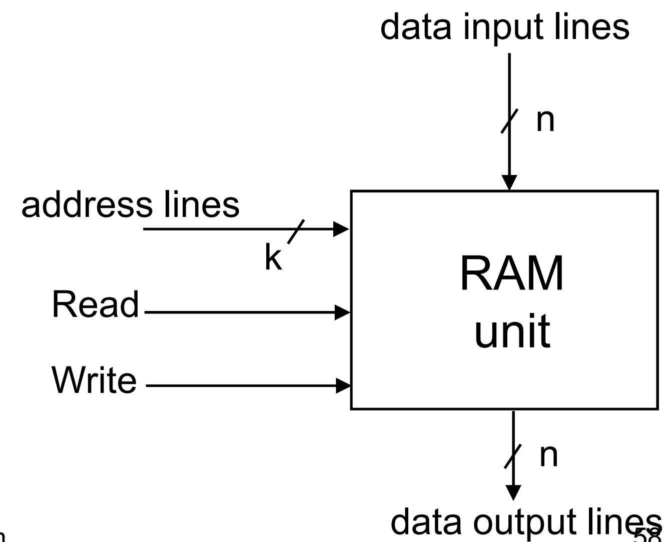
$$BUS \leftarrow R1, R2 \leftarrow BUS$$

- In the former case the bus is implicit, but in the latter, it is explicitly indicated

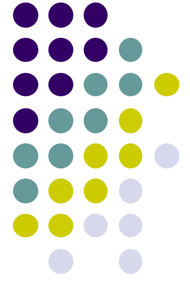
MEMORY (RAM)



- Memory (RAM) can be thought as a sequential circuits containing some number of registers
- These registers hold the *words* of memory
- Each of the r registers is indicated by an *address*
- These addresses range from 0 to r-1
- Each register (word) can hold n bits of data
- Assume the RAM contains r = 2k words. It needs the following
  - n data input lines
  - n data output lines
  - k address lines
  - A Read control line
  - A Write control line

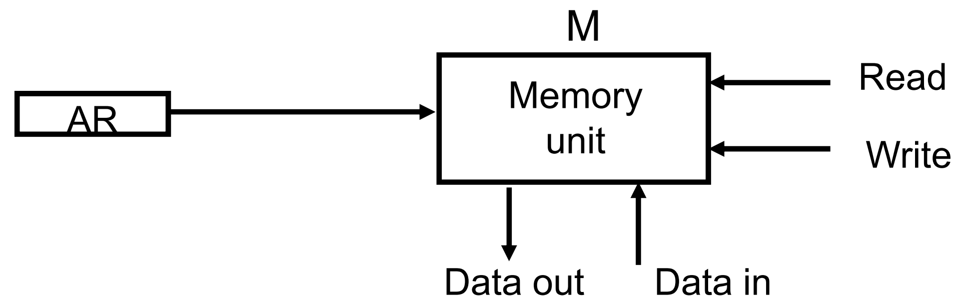


MEMORY TRANSFER

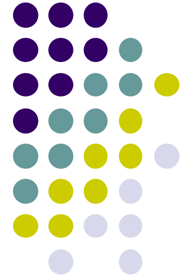


- Collectively, the memory is viewed at the register level as a device, M.
- Since it contains multiple locations, we must specify which address in memory we will be using
- This is done by indexing memory references

- Memory is usually accessed in computer systems by putting the desired address in a special register, the *Memory Address Register* (*MAR*, or *AR*)
- When memory is accessed, the contents of the MAR get sent to the memory unit's address lines



MEMORY READ

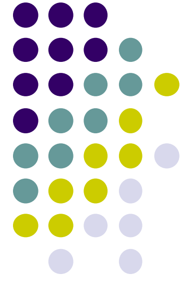


- To read a value from a location in memory and load it into a register, the register transfer language notation looks like this:

$$R1 \leftarrow M[MAR]$$

- This causes the following to occur
  - The contents of the MAR get sent to the memory address lines
  - A Read (= 1) gets sent to the memory unit
  - The contents of the specified address are put on the memory's output data lines
  - These get sent over the bus to be loaded into register R1

MEMORY WRITE

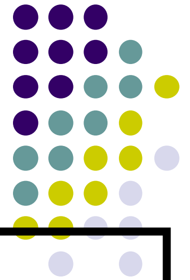


- To write a value from a register to a location in memory looks like this in register transfer language:

$$M[MAR] \leftarrow R1$$

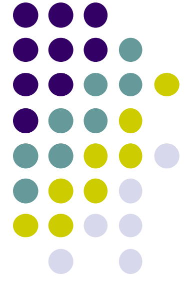
- This causes the following to occur
  - The contents of the MAR get sent to the memory address lines
  - A Write (= 1) gets sent to the memory unit
  - The values in register R1 get sent over the bus to the data input lines of the memory
  - The values get loaded into the specified address in the memory

SUMMARY OF R. TRANSFER MICROOPERATIONS



A ← B	Transfer content of reg. B into reg. A
AR ← DR(AD)	Transfer content of AD portion of reg. DR into reg. AR
A ← constant	Transfer a binary constant into reg. A
ABUS ← R1,	Transfer content of R1 into bus A and, at the same time,
R2 ← ABUS	transfer content of bus A into R2
AR	Address register
DR	Data register
M[R]	Memory word specified by reg. R
M	Equivalent to M[AR]
DR ← M	Memory *read* operation: transfers content of memory word specified by AR into DR
M ← DR	Memory *write* operation: transfers content of DR into memory word specified by AR

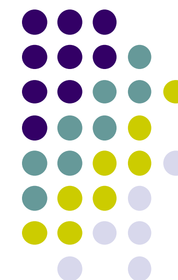
MICROOPERATIONS



- Computer system microoperations are of four types:

    - Register transfer microoperations
    - Arithmetic microoperations
    - Logic microoperations
    - Shift microoperations

ARITHMETIC MICROOPERATIONS



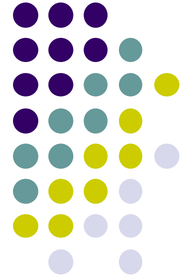
- The basic arithmetic micro-operations are
  - Addition
  - Subtraction
  - Increment
  - Decrement

- The additional arithmetic micro-operations are
  - Add with carry
  - Subtract with borrow
  - Transfer/Load
  - etc. …

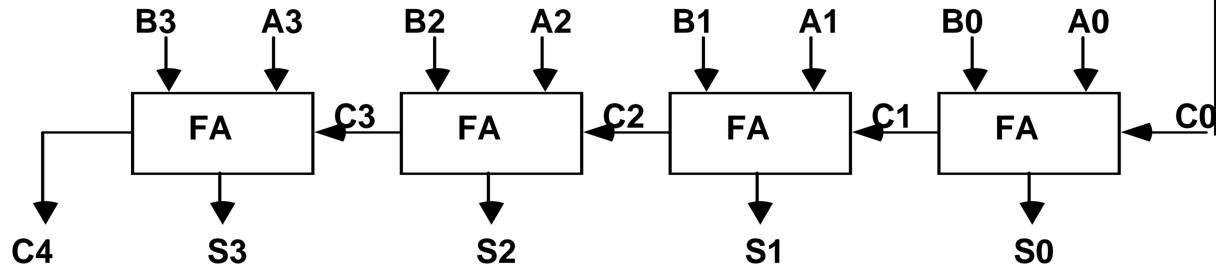
Summary of Typical Arithmetic Micro-Operations

R3 ← R1 + R2	Contents of R1 plus R2 transferred to R3
R3 ← R1 - R2	Contents of R1 minus R2 transferred to R3
R2 ← R2’	Complement the contents of R2
R2 ← R2’+ 1	2's complement the contents of R2 (negate)
R3 ← R1 + R2’+ 1	subtraction
R1 ← R1 + 1	Increment
R1 ← R1 - 1	Decrement

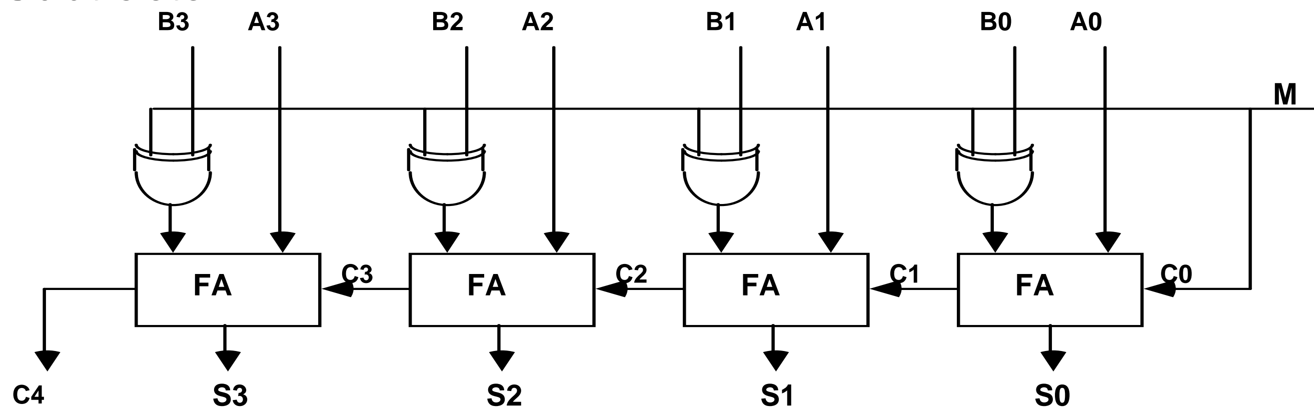
BINARY ADDER / SUBTRACTOR / INCREMENTER



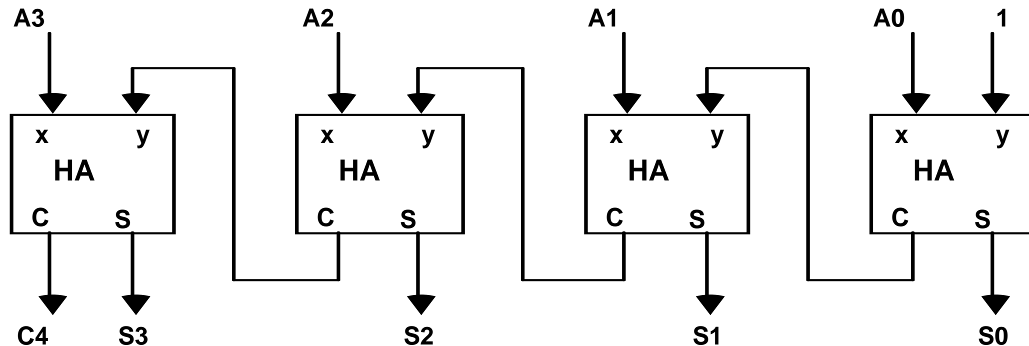
Binary Adder



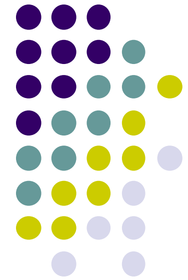
Binary Adder-Subtractor

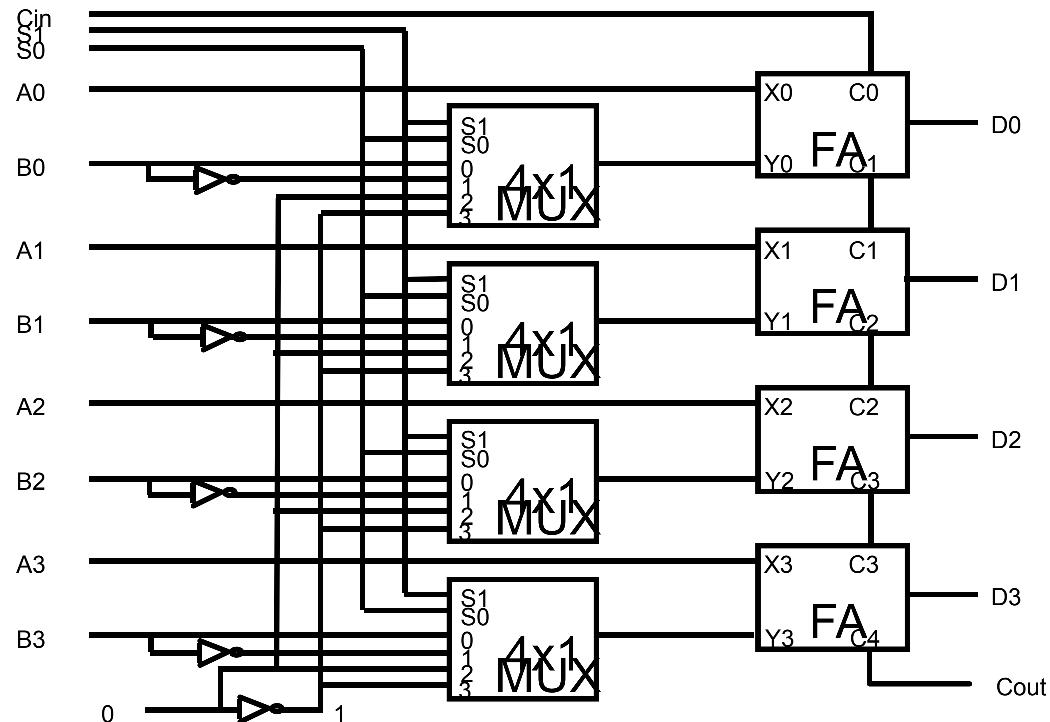


Binary Incrementer



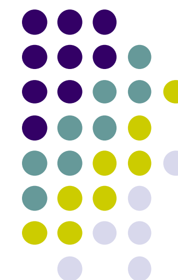
ARITHMETIC CIRCUIT





S1	S0	Cin	Y	Output	Microoperation
0	0	0	B	D = A + B	Add
0	0	1	B	D = A + B + 1	Add with carry
0	1	0	B’	D = A + B’	Subtract with borrow
0	1	1	B’	D = A + B’+ 1	Subtract
1	0	0	0	D = A	Transfer A
1	0	1	0	D = A + 1	Increment A
1	1	0	1	D = A - 1	Decrement A
1	1	1	1	D = A	Transfer A

LOGIC MICROOPERATIONS

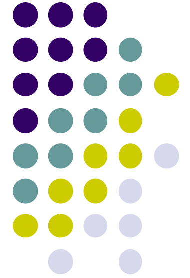


- Specify binary operations on the strings of bits in registers
  - Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data
  - useful for bit manipulations on binary data
  - useful for making logical decisions based on the bit value
- There are, in principle, 16 different logic functions that can be defined over two binary input variables

A	B	F_0	F_1	F_2	…	F_{13}	F_{14}	F_{15}
0	0	0	0	0	…	1	1	1
0	1	0	0	0	…	1	1	1
1	0	0	0	1	…	0	1	1
1	1	0	1	0	…	1	0	1

- However, most systems only implement four of these
  - AND ($\wedge$), OR ($\vee$), XOR ($\oplus$), Complement/NOT
- The others can be created from combination of these

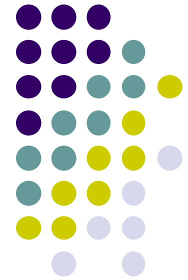
LIST OF LOGIC MICROOPERATIONS



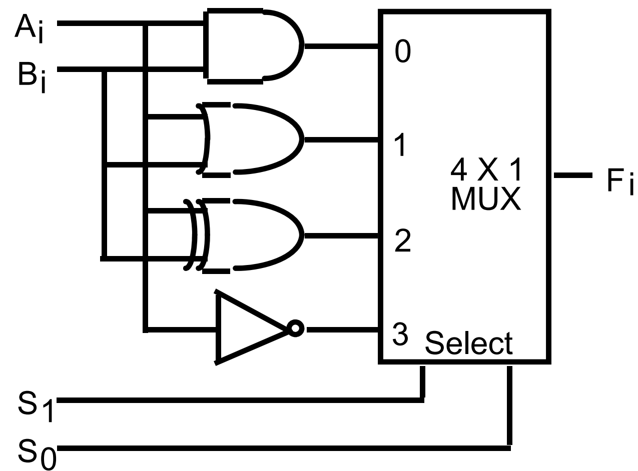
- List of Logic Microoperations
  - 16 different logic operations with 2 binary vars.
  - n binary vars $\rightarrow$ 2^{2^n} functions

- Truth tables for 16 functions of 2 variables and the corresponding 16 logic micro-operations

x 0 0 1 1 y 0 1 0 1	*Boolean Function*	*Micro-Operations*	*Name*
0 0 0 0	F0 = 0	$F \leftarrow 0$	Clear
0 0 0 1	F1 = xy	$F \leftarrow A \wedge B$	AND
0 0 1 0	F2 = xy'	$F \leftarrow A \wedge B'$	
0 0 1 1	F3 = x	$F \leftarrow A$	Transfer A
0 1 0 0	F4 = x'y	$F \leftarrow A' \wedge B$	
0 1 0 1	F5 = y	$F \leftarrow B$	Transfer B
0 1 1 0	F6 = $x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
0 1 1 1	F7 = x + y	$F \leftarrow A \vee B$	OR
1 0 0 0	F8 = (x + y)'	$F \leftarrow (A \vee B)'$	NOR
1 0 0 1	F9 = $(x \oplus y)'$	$F \leftarrow (A \oplus B)'$	Exclusive-NOR
1 0 1 0	F10 = y'	$F \leftarrow B'$	Complement B
1 0 1 1	F11 = x + y'	$F \leftarrow A \vee B$	
1 1 0 0	F12 = x'	$F \leftarrow A'$	Complement A
1 1 0 1	F13 = x' + y	$F \leftarrow A' \vee B$	
1 1 1 0	F14 = (xy)'	$F \leftarrow (A \wedge B)'$	NAND
1 1 1 1	F15 = 1	F ← all 1's	Set to all 1's



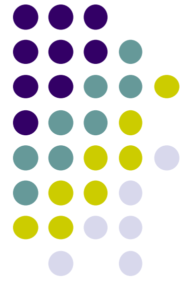
HARDWARE IMPLEMENTATION OF LOGIC MICRO-OPERATIONS



Function table

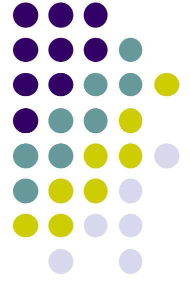
S_1 S_0	Output	μ-operation
0 0	$F = A \wedge B$	AND
0 1	$F = A \vee B$	OR
1 0	$F = A \oplus B$	XOR
1 1	$F = A'$	Complement

APPLICATIONS OF LOGIC MICROOPERATIONS



- Logic micro-operations can be used to manipulate individual bits or a portions of a word in a register
- Consider the data in a register A. In another register, B, is bit data that will be used to modify the contents of A
  - Selective-set $A \leftarrow A + B$
  - Selective-complement $A \leftarrow A \oplus B$
  - Selective-clear $A \leftarrow A \bullet B'$
  - Mask (Delete) $A \leftarrow A \bullet B$
  - Clear $A \leftarrow A \oplus B$
  - Insert $A \leftarrow (A \bullet B) + C$
  - Compare $A \leftarrow A \oplus B$
  - ….

SELECTIVE SET

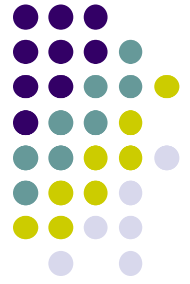


- In a selective set operation, the bit pattern in B is used to *set* certain bits in A

$$\begin{array}{ll} 1\,1\,0\,0 & A_t \\ 1\,0\,1\,0 & B \\ \hline 1\,1\,1\,0 & A_{t+1} \quad (A \leftarrow A + B) \end{array}$$

- If a bit in B is set to 1, that same position in A gets set to 1, otherwise that bit in A keeps its previous value

SELECTIVE COMPLEMENT

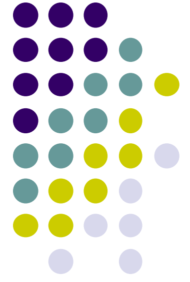


- In a selective complement operation, the bit pattern in B is used to *complement* certain bits in A

$$
\begin{array}{ll}
1\,1\,0\,0 & A_t \\
1\,0\,1\,0 & B \\
\hline
0\,1\,1\,0 & A_{t+1} \qquad (A \leftarrow A \oplus B)
\end{array}
$$

- If a bit in B is set to 1, that same position in A gets complemented from its original value, otherwise it is unchanged

SELECTIVE CLEAR

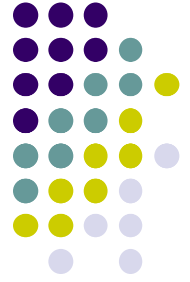


- In a selective clear operation, the bit pattern in B is used to *clear* certain bits in A

$$
\begin{array}{ll}
1\,1\,0\,0 & A_t \\
\underline{1\,0\,1\,0} & \underline{B} \\
0\,1\,0\,0 & A_{t+1} \qquad (A \leftarrow A \cdot B')
\end{array}
$$

- If a bit in B is set to 1, that same position in A gets set to 0, otherwise it is unchanged

MASK OPERATION

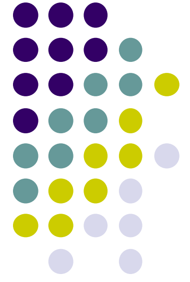


- In a mask operation, the bit pattern in B is used to *clear* certain bits in A

$$
\begin{array}{ll}
1\,1\,0\,0 & A_t \\
1\,0\,1\,0 & B \\
\hline
1\,0\,0\,0 & A_{t+1} \qquad (A \leftarrow A \cdot B)
\end{array}
$$

- If a bit in B is set to 0, that same position in A gets set to 0, otherwise it is unchanged

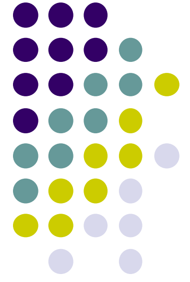
CLEAR OPERATION



- In a clear operation, if the bits in the same position in A and B are the same, they are cleared in A, otherwise they are set in A

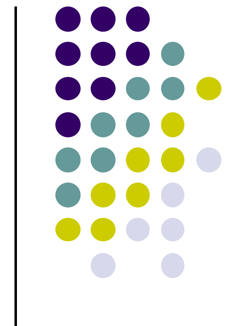
$$
\begin{array}{ll}
1\,1\,0\,0 & A_t \\
1\,0\,1\,0 & B \\
\hline
0\,1\,1\,0 & A_{t+1} \quad (A \leftarrow A \oplus B)
\end{array}
$$

INSERT OPERATION

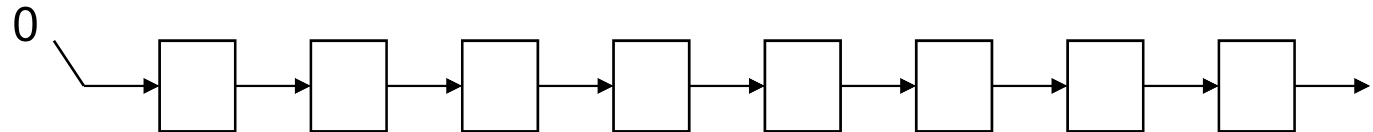


- An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged
- This is done as
  - A mask operation to clear the desired bit positions, followed by
  - An OR operation to introduce the new bits into the desired positions
  - Example
    - Suppose you wanted to introduce 1010 into the low order four bits of A: 1101 1000 1011 0001 A (Original)
      1101 1000 1011 1010 A (Desired)
    - ```
      1101 1000 1011 0001      A (Original)
      1111 1111 1111 0000      Mask
      1101 1000 1011 0000      A (Intermediate)
      0000 0000 0000 1010      Added bits
      1101 1000 1011 1010      A (Desired)
      ```
      ```

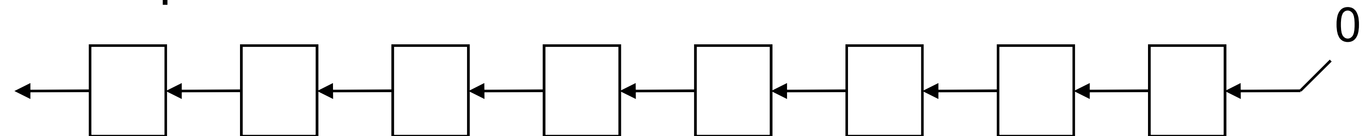
LOGICAL SHIFT



- In a logical shift the serial input to the shift is a 0.
- A right logical shift operation:

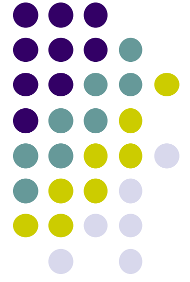


- A left logical shift operation:

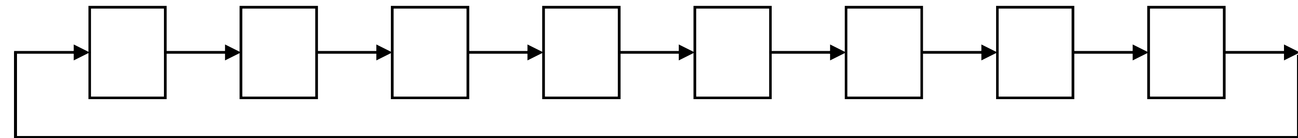


- In a Register Transfer Language, the following notation is used
  - *shl* for a logical shift left
  - *shr* for a logical shift right
  - Examples:
    - R2 $\leftarrow$ *shr* R2
    - R3 $\leftarrow$ *shl* R3

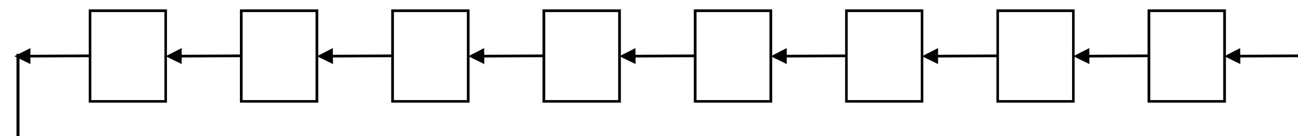
CIRCULAR SHIFT



- In a circular shift the serial input is the bit that is shifted out of the other end of the register.
- A right circular shift operation:

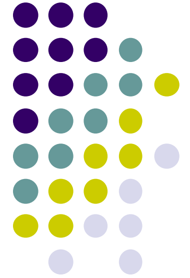


- A left circular shift operation:

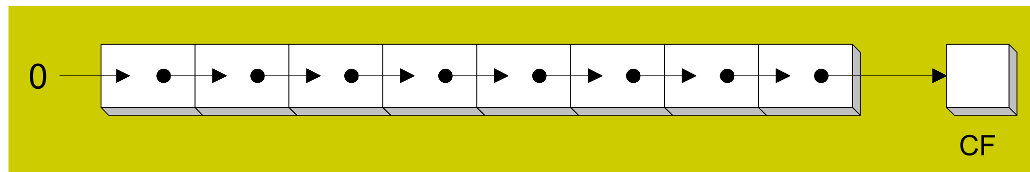


- In a RTL, the following notation is used
  - *cil* for a circular shift left
  - *cir* for a circular shift right
  - Examples:
    - R2 ← *cir* R2
    - R3 ← *cil* R3

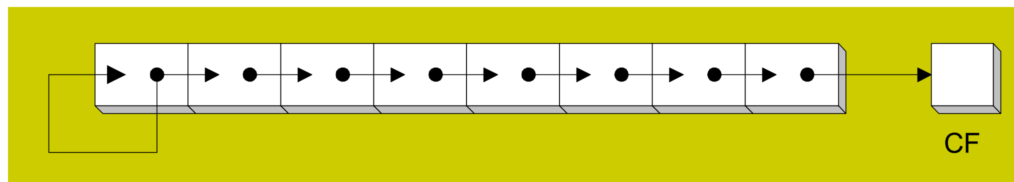
Logical versus Arithmetic Shift



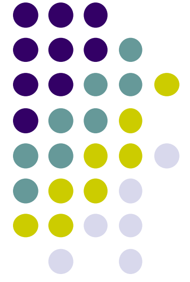
- A logical shift fills the newly created bit position with zero:



- An arithmetic shift fills the newly created bit position with a copy of the number's sign bit:

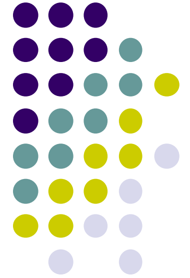


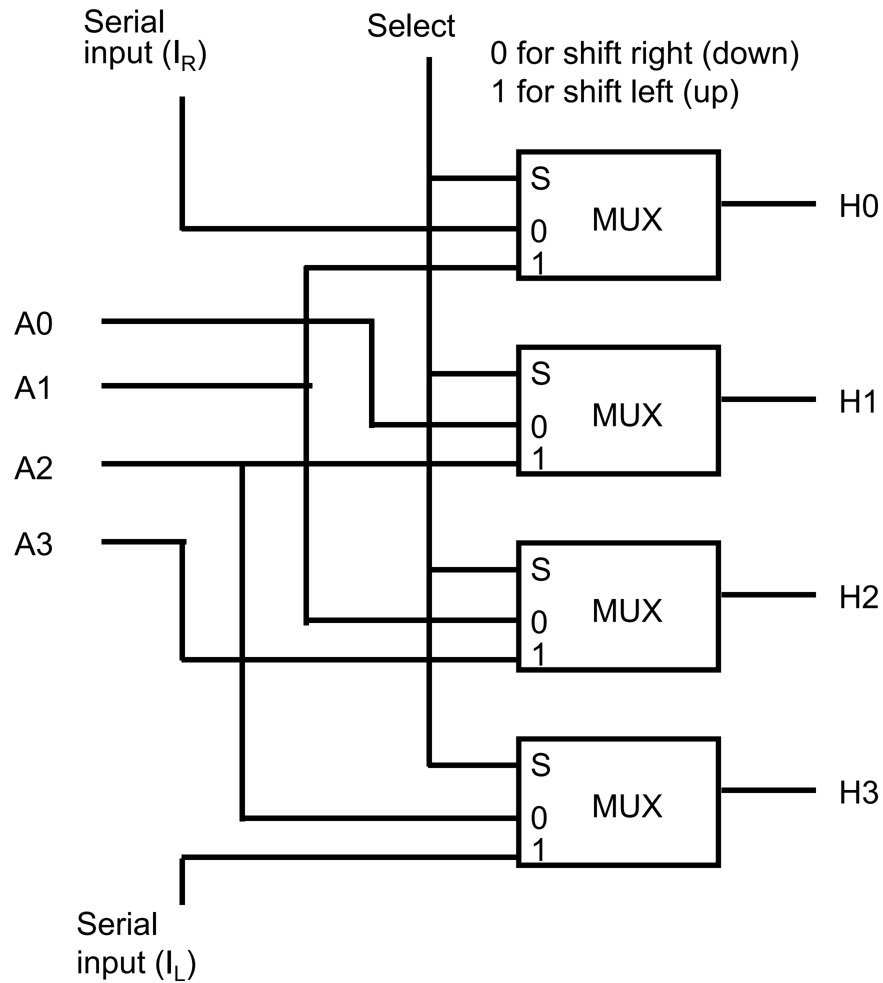
ARITHMETIC SHIFT



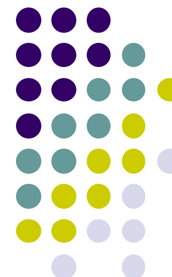
- In a RTL, the following notation is used
  - *ashl* for an arithmetic shift left
  - *ashr* for an arithmetic shift right
  - Examples:
    - » R2 $\leftarrow$ *ashr* R2
    - » R3 $\leftarrow$ *ashl* R3

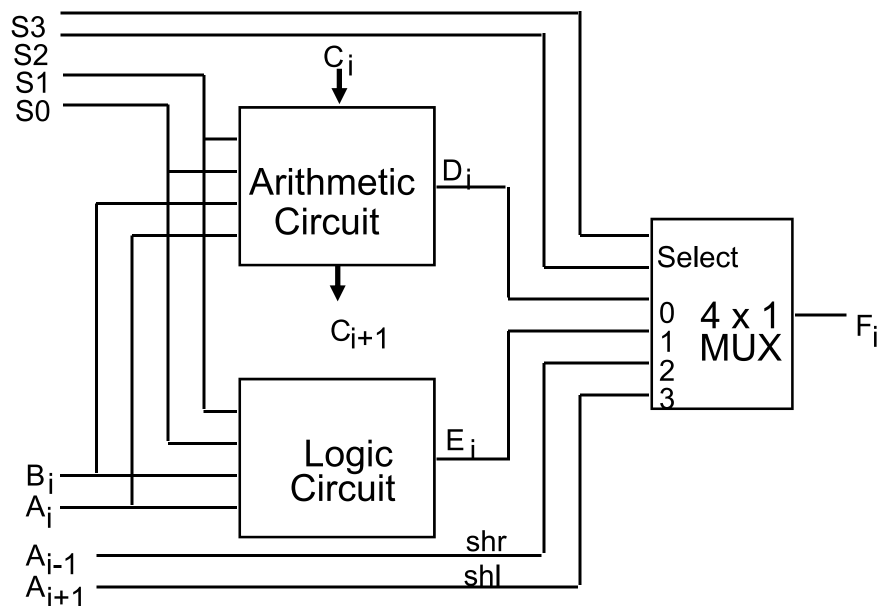
HARDWARE IMPLEMENTATION OF SHIFT MICROOPERATIONS





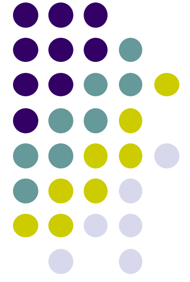
ARITHMETIC LOGIC SHIFT UNIT





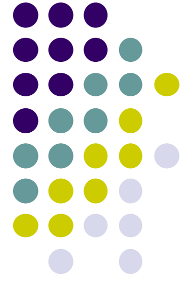
S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Addition
0	0	0	1	1	F = A + B + 1	Add with carry
0	0	1	0	0	F = A + B’	Subtract with borrow
0	0	1	0	1	F = A + B’+ 1	Subtraction
0	0	1	1	0	F = A - 1	Decrement A
0	0	1	1	1	F = A	TransferA
0	1	0	0	X	$F = A \wedge B$	AND
0	1	0	1	X	$F = A \vee B$	OR
0	1	1	0	X	$F = A \oplus B$	XOR
0	1	1	1	X	F = A’	Complement A
1	0	X	X	X	F = shr A	Shift right A into F
1	1	X	X	X	F = shl A	Shift left A into F

Information Representation



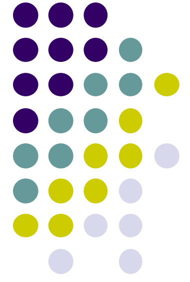
- How is “information” represented in a computer system ?
- What are the different types of information
  - Text
  - Numbers
  - Images
    - Video
    - Photographic
  - Audio

Everything is in Binary ! (1's and 0's)



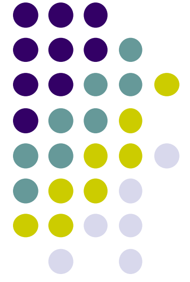
- Computers are **digital** devices - they can only manipulate information in digital (binary) form.
- Easy to represent 1 and 0 in electronic, magnetic and optical devices
- Only need two states
  - High/low
  - On/off
  - Up/down
  - etc
- All information in a computer system is
  - Processed in binary form
  - Stored in binary form
  - Transmitted in binary form

How is information converted to Binary form



- I/O and Storage Devices are digital
- I/O devices convert information to/from binary
  - A keyboard converts the character “A” you type into a **binary code** to represent “A”
  - E.g. “A” is represented by the binary code 01000001
  - Monitor converts 01000001 to the “A” that you read

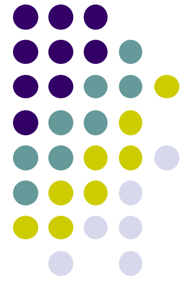
Bits and Bytes



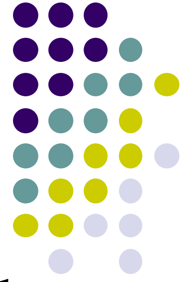
- One **b**inary dig**it** i.e. 1 or 0 is called a **bit**
- A group of 8 bits is one **byte**
- Byte is the unit of storage measurement

Number of Bytes	Unit
1024 bytes (2^{10} bytes)	1 **Kilobyte** (Kb)
1024 Kb (2^{20} bytes)	1 **Megabyte** (Mb)
1024 Mb (2^{30} bytes)	1 **Gigabyte** (Gb)
1024 Gb (2^{40} bytes)	1 **Terabyte** (Tb)
1024 Tb (2^{50} bytes)	1 **Petabyte** (Pb)

Representing Text- ASCII Code

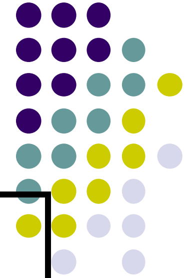


- Textual information is made up of individual **characters** e.g.
- Letters:
  - Lowercase: a,b,c,..z
  - Uppercase: A,B,C..Z
- Digits: 0,1,2,..9
- Punctuation characters: ., :, ; ,, “, ’
- Other symbols: -, +, &, %, #, /,\,£, etc.).



Representing Text- ASCII Code

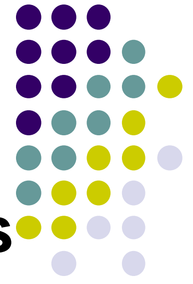
- Each character is represented by a unique binary code.
- ASCII is one international standard that specifies the binary code for each character.
- **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
- It is a 7-bit code - every character is represented by 7 bits
- There are other standards such as EBCDIC but these are not widely used.
- ASCII is being superceded by Unicode of which ASCII is a subset. Unicode is a 16-bit code.



Sample ASCII Codes

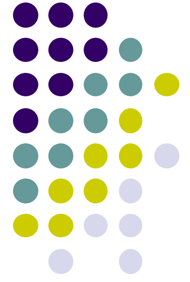
Char	ASCII	Decimal	Char	ASCII	Decimal
NUL	000 0000	00	BEL	000 0111	07
LF	000 1010	10	CR	000 1011	13
0	011 0000	48	SP	010 0000	20
1	011 0001	49	!	010 0001	21
2	011 0010	50	“	010 0010	22
9	011 1001	57			
A	100 0001	65	a	110 0001	97
B	100 0010	66	b	110 0010	98
C	100 0011	67	c	110 0011	99
Y	101 1001	89	y	111 1001	121
Z	101 1010	90	z	111 1010	122

Comments on ASCII Codes



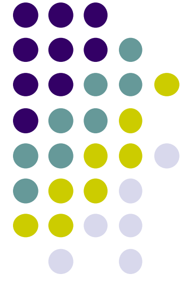
- Codes for A to Z and a to z form **collating sequences**
  - A is 65, B is 66, C is 67 and so on
  - A is 97, b is 98, c is 99 and so on
- Lowercase code is 32 greater than Uppercase equivalent
- Note that digit '0' is not the same as number 0
  - ASCII is used for characters
  - Not used to represent numbers (See later)
- Codes 0 to 30 are typically for **Control Characters**
  - Bel - causes speaker to beep !
  - Carriage Return (CR); LineFeed (LF)
  - Others used to control communication between devices
    - SYN, ACK, NAK, DLE etc

Review



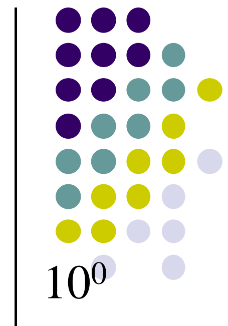
- All information stored/transmitted in binary
- Devices convert to/from binary to other forms that humans understand
- Bits and Bytes
- KB, Mb, GB, TB and PB are storage metrics
- ASCII code is a 7-bit code to represent text characters
- Text “numbers” not the same as “math's” numbers
    - Do not add phone numbers or get average of PPS numbers

Representing Numbers: Integers



- Humans use **Decimal** Number System
- Computers use Binary Number System
- Important to understand Decimal system before looking at binary system
- **Decimal Numbers - Base 10**
  - 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - **Positional** number system: the position of a digit in a number determines its value
  - Take the number 1649
    - The 1 is worth 1000
    - The 9 is worth 9 units
  - Formally, the digits in a decimal number are weighted by increasing powers of 10 i.e. they use the base 10. We can write 1649 in the following form:
    - $1*10^3 + 6*10^2 + 4*10^1 + 9*10^0$

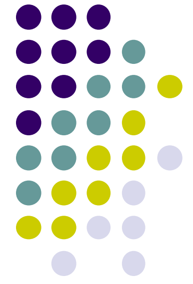
Representing Numbers: Integers



weighting:	10^3	10^2	10^1	10^0
Digits	1	6	4	9
1649 =	$1*10^3$ +	$6*10^2$ +	$4*10^1$ +	$9*10^0$

- **Least Significant Digit**: rightmost one - *9* above
  - Lowest power of 10 weighting
  - Digits on the right hand side are called the **low-order digits** (lower powers of 10).
- **Most Significant Digit**: leftmost one - *1* above
  - Highest power of 10 weighting
  - The digits on the left hand side are called the **high-order digits** (higher powers of 10)

Representing Numbers: Decimal Numbers



- Largest n-digit number ?
  - Made up of *n* consecutive *9*'s (= 10^n -1)
  - Largest 4-digit number if 9999
  - 9999 is 10^4 -1

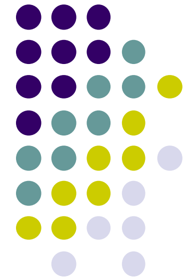
- Distinguishing Decimal from other number systems such as Binary, Hexadecimal (base 16) and Octal (base 8)

  - How do we know whether the number 111 is decimal or binary

  - One convention is to **use subscripts**
  - Decimal: 111_{10} Binary: 111_2 Hex: 111_{16} Octal: 111_8
    - Difficult to write use keyboard

  - Another convention is to append a letter (D, B, H, O)
    - Decimal: 111**D** Binary:111**B** Hex: 111**H** Octal: 111**O**

Representing Numbers: Binary Numbers

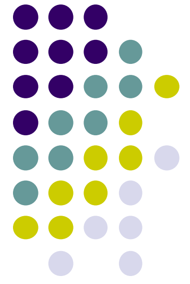


- **Binary numbers** are **Base 2** numbers
  - Only 2 digits: 0 and 1
  - Formally, the digits in a binary number are weighted by increasing powers of 2
  - They operate as decimal numbers do in all other respects
  - Consider the binary number 0101 1100

Weight	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
bits	0	1	0	1	1	1	0	0

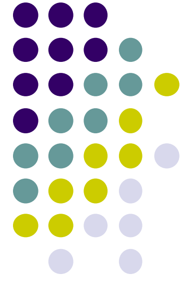
- $01011100 = 0*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0$
  $= 0 + 64_{10} + 0 + 16_{10} + 8_{10} + 4_{10} + 0 + 0$
  $= 92_{10}$

Representing Numbers: Binary Numbers



- Leftmost bit is the **most significant bit** (**MSB**).
  - The leftmost bits in a binary number are referred to as the **high-order** bits.
- Rightmost bit is the **least significant bit** (**LSB**).
  - The rightmost bits in a binary number are referred to as the **low-order** bits.
  - Largest n-bit binary number ?
    - Made up of n consecutive 1's ($-2^n - 1$)
    - e.g. largest 4-bit number: $1111 = 2^4 - 1 = 15$

Representing Numbers: Binary Numbers

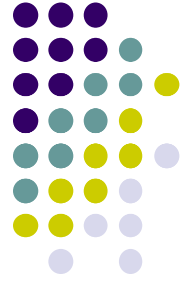


- **Exercises**

- Convert the following binary numbers to decimal:
- (i) 1000 1000 (ii) 1000 1001 (iii) 1000 0111
- (iv) 0100 0001 (v) 0111 1111 (vi) 0110 0001

- **Joe Carty Formatting Convention**
- In these notes we insert a **space** after every 4 bits to make the numbers easier to read

Representing Numbers: Converting Decimal to Binary



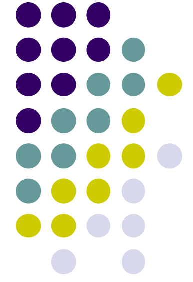
- To convert from one number base to another:
  - you repeatedly divide the number to be converted by the new base
  - the remainder of the division at each stage becomes a digit in the new base
  - until the result of the division is 0.

- Example: To convert decimal 35 to binary we do the following:

	Remainder
35 / 2	1
17 / 2	1
8 / 2	0
4 / 2	0
2 / 2	0
1 / 2	1
0	

- The result is read **upwards** giving $35_{10} = 100011_2$.

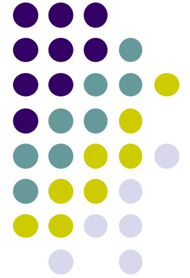
Representing Numbers: Converting Decimal to Binary



- Exercise: Convert the following decimal numbers to binary
- (1) 64 (2) 65 (3) 32 (4) 16 (5) 48

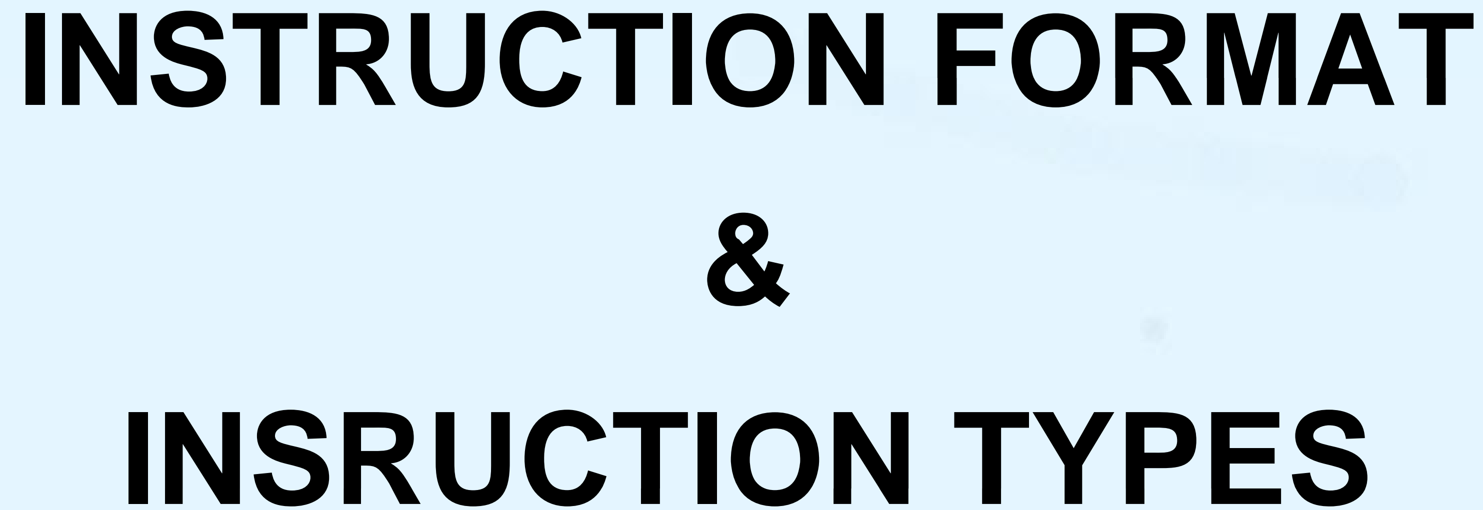
- **Shortcuts**
- To convert any decimal number which is a power of 2, to binary, simply write 1 followed by the number of zeros given by the power of 2.
- For example, 32 is 2^5, so we write it as 1 followed by 5 zeros, i.e. 10000; 128 is 2^7 so we write it as 1 followed by 7 zeros, i.e. 100 0000.
- Remember that the largest binary number that can be stored in a given number of bits is made up of n 1's.
- An easy way to convert this to decimal, is to note that this is 2n - 1.
- For example, if we are using 4-bit numbers, the largest value we can represent is 1111 which is 2^4-1, i.e. 15

Representing Numbers: Converting Decimal to Binary



- Binary Numbers that you should remember because they occur so frequently

Binary	Decimal
111	7
1111	15
0111 1111	127
1111 1111	255


INSTRUCTION FORMAT
&
INSRUCTION TYPES

Instruction Formats

- Bits of the instruction are divided in to fields..
- Most common fields are:

➢ **OPERATION CODE FIELD** that specifies the operation to be performed.

➢ **ADDRESS FIELD** that designates a memory address or a processor register.

➢ **MODE FIELD** that specifies the way the operand or effective address is determined. Or we can say it tells about addressing mode to be adopted.

REGISTER ADDRESS

- Operands are generally stored in Memory or Processor registers.
- Operands residing in memory are specified by their memory address.
- Operands residing in processor registers are specified by their register address.
- A register address is a binary number of k bits which defines one of the 2^k registers in the CPU.
- For eg. CPU having 16 processor registers R0 to R15 will have a address field of 4 bits.
- As binary no. 0101 will designate R5.

Instruction Formats

- Most systems today are GPR systems.
- There are three types:
  - Memory-memory where two or three operands may be in memory.
  - Register-memory where at least one operand must be in a register.
  - Load-store where no operands may be in memory.
- The number of operands and the number of available registers has a direct affect on instruction length.

- The next consideration for architecture design concerns how the CPU will store data .
- As the no. of address fields in the instruction format depends on internal organization or architecture of its registers.
- We have three choices of CPU organizations
  1. A stack architecture
  2. An accumulator architecture
  3. A general purpose register architecture.
- In choosing one over the other, the tradeoffs are simplicity (and cost) of hardware design with execution speed and ease of use.

Instruction Formats

- In a **stack architecture**, instructions and operands are implicitly taken from the stack.
  - A stack cannot be accessed randomly.
- In an **accumulator architecture**, one operand of a binary operation is implicitly in the accumulator.
  - Both operand is in memory, creating lots of bus traffic.
- In a **general purpose register (GPR) architecture**, registers can be used instead of memory.
  - Faster than accumulator architecture.
  - Efficient implementation for compilers.
  - Results in longer instructions.

STACK ARCHITECTURE

- Stack machines use one- and zero-operand instructions.
- **`LOAD`** and **`STORE`** instructions require a single memory address operand.
- Other instructions use operands from the stack implicitly.
- **`PUSH`** and **`POP`** operations involve only the stack's top element.
- Binary instructions (e.g., **`ADD`**, **`MULT`**) use the top two items on the stack.

STACK ARCHITECTURE

- Stack architectures require us to think about arithmetic expressions a little differently.
- We are accustomed to writing expressions using *infix* notation, such as: Z = X + Y.
- Stack arithmetic requires that we use *postfix* notation: Z = XY+.
- Eg. PUSH X

  PUSH Y

  ADD

ACCUMULATOR ARCHITECTURE

- All operations will be performed using an accumulator register.
- It only requires one address field.
- Eg. ADD X

where X is the address of operand and adding will be held like AC+M[X] goes to AC where AC is accumulator register.

GENERAL REGISTER ARCHITECTURE

- General processor registers and memory word will be used here for storage of operands.
- It needs two to three address fields
- Eg: ADD R1, R2, R3

  ADD R1, R2

  MOV R1, R2

  ADD R1, X

TYPES OF ADDRESS INSTRUCTIONS

- THREE ADDRESS INSTRUCTIONS
- TWO ADDRESS INSTRUCTIONS
- ONE ADDRESS INSTRUCTIONS
- ZERO ADDRESS INSTRUCTIONS
- RISC INSTRUCTIONS

Example to explain different address instructions

- For example, the infix expression,

$$Z = (X \times Y) + (W \times U)$$

Where X,Y,W and U are the memory addresses where the operands are stored.

THREE ADDRESS INSTRUCTION

- With a three-address ISA, (e.g.,mainframes), the infix expression,

  $$Z = X \times Y + W \times U$$

  might look like this:

```
MUL R1,X,Y
MUL R2,W,U
ADD  Z,R1,R2
```

TWO ADDRESS INSTRUCTION

- In a two-address ISA, (e.g.,Intel, Motorola), the infix expression,

  $$Z = X \times Y + W \times U$$

  might look like this:

```
MOV R1,X
MUL R1,Y
MOV R2,W
MUL R2,U
ADD  R1,R2
MOV Z,R1
```

ONE ADDRESS INSTRUCTION

- In a one-address ISA, the infix expression,

  $$Z = X \times Y + W \times U$$

  looks like this:

```
LOAD X
MUL Y
STORE TEMP
LOAD W
MUL U
ADD TEMP
STORE Z
```

Accumulator will be used here.

ZERO ADDRESS INSTRUCTION

- In a stack ISA, the postfix expression,

  Z = X Y × W U × +

  might look like this:

```
PUSH X
PUSH Y
MUL
PUSH W
PUSH U
MUL
ADD
PUSH Z
```

Note: The result of a binary operation is implicitly stored on the top of the stack!

RISC INSTRUCTION

- With a RISC (Reduced Instruction set computer) the infix expression,

  $$Z = X \times Y + W \times U$$

  might look like this:

```
LOAD R1,X
LOAD R2,Y
LOAD R3,W
LOAD R4,U
MUL R1,R1,R2
MUL R3,R3,R4
ADD  R1,R1,R3
STORE Z,R1
```

Contains less no. of instructions.

- We have seen how instruction length is affected by the number of operands supported by the ISA.
- In any instruction set, not all instructions require the same number of operands.
- Operations that require no operands, such as `HALT`, necessarily waste some space when fixed-length instructions are used.

- **This is all about INSTRUCTION FORMAT.**

Instruction types

Instructions fall into several broad categories that you should be familiar with:

- Data Transfer Instructions
- Data Manipulation Instructions
  - ❖ Arithmetic
  - ❖ Logical
  - ❖ Shift
- Program Control Instructions

Data Transfer Instructions

Name	Mnemonic
Load	LD
Store	ST
Move	MOV
Exchange	XCH
Input	IN
Output	OUT
Push	PUSH
Pop	POP

Data Manipulation Instructions

Arithmetic Instructions:

Name	Mnemonics
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with Carry	ADDC
Subtract with borrow	SUBB
Negate(2's compliment)	NEG

Data Manipulation Instructions

Logical and Bit manipulation Instructions:

Name	Mnemonics
Clear	CLR
Compliment	COM
AND	AND
OR	OR
EXOR	XOR
Clear carry	CLRC
Set carry	SETC
Compliment carry	COMPC
Enable and Disable Interrupt	EI & DI

Data Manipulation Instructions

Shift Instructions:

Name	Mnemonics
Logical shift Right	SHR
Logical shift Left	SHL
Arithmetic Shift Right	SHRA
Arithmetic Shift Left	SHLA
Rotate Right	ROR
Rotate Left	ROL
Rotate Right with carry	RORC
Rotate Left with carry	ROLC

Program Control Instructions

Name	Mnemonics
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare(by subtraction)	CMP
Test(by ANDing)	TST

Program Control Instructions (Conditional Branch Instructions)

Name	Mnemonics	Tested Conditions
Branch if zero	BZ	Z=1
Branch if not zero	BNZ	Z−0
Branch if carry	BC	C=1
Branch if not carry	BNC	C=0
Branch if overflow OR not overflow	BV or BNV	V=1 OR 0
Branch if greater than	BGT	A>B
Branch if less than	BLT	A<B
Branch if equal	BE	A=B
Branch if not equal	BNE	A not equal to B
Branch if higher	BHI	A>B
Branch if lower	BLO	A<B

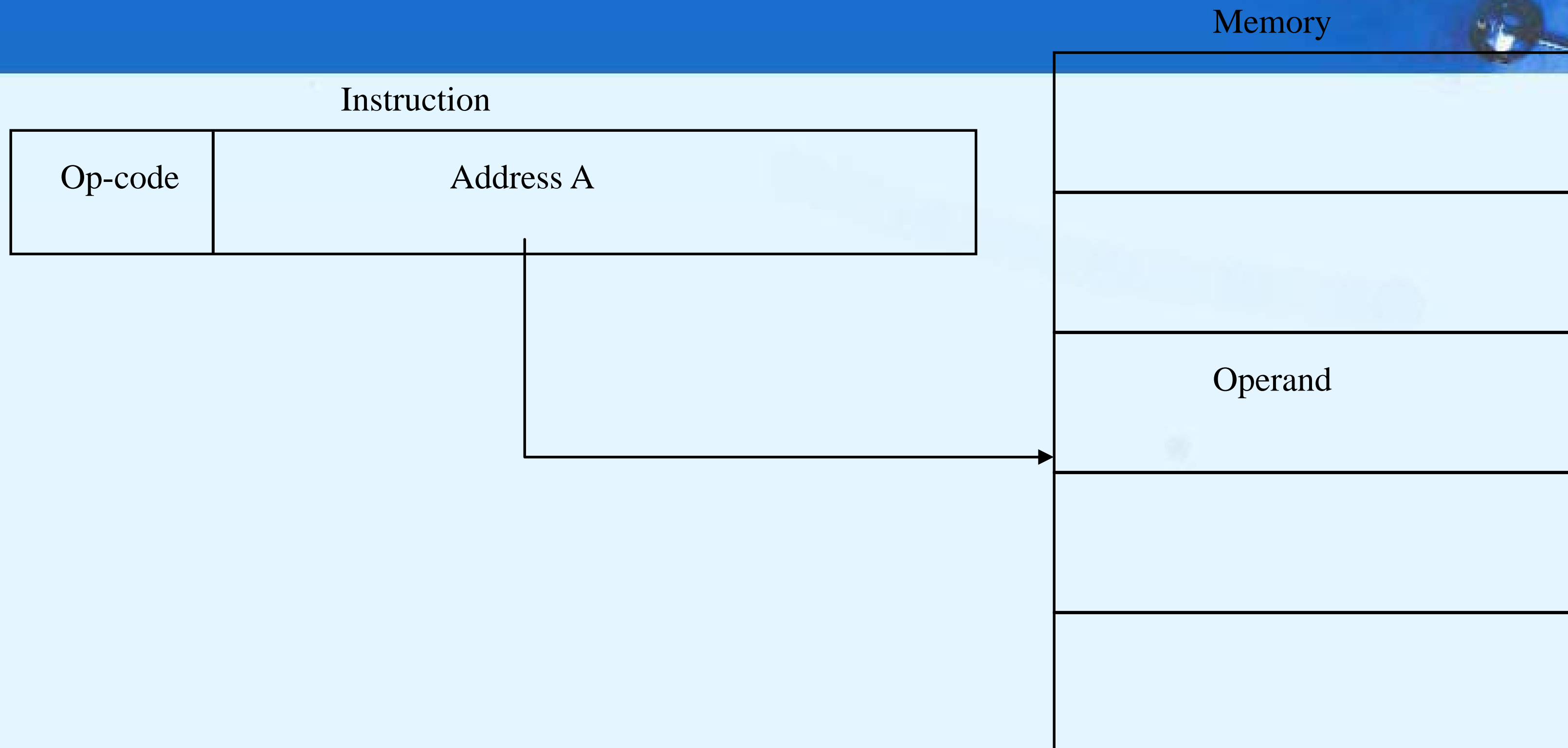
Addressing Modes

- An architecture addressing mode is the set of syntaxes and methods that instructions use to specify a memory address
  - For operands or results
  - As a target address for a branch instruction
- When a microprocessor accesses memory, to either read or write data, it must specify the memory address it needs to access
- Several addressing modes to generate this address are known, a microprocessor instruction set architecture may contain some or all of those modes, depending on its design
- In the following examples we will use the LDAC instruction (loads data from a memory location into the AC - accumulator - microprocessor register)

Types of Addressing Modes

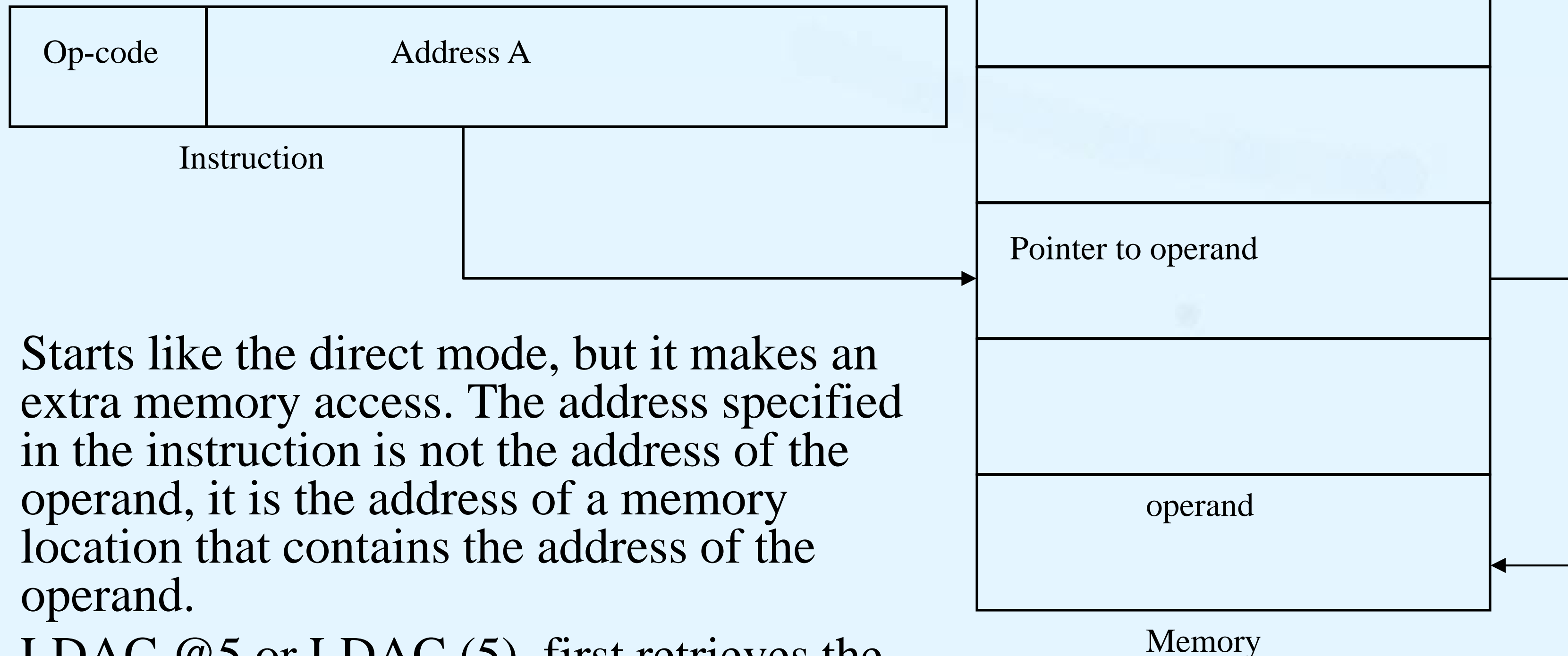
- Direct Mode
- Indirect Mode
- Register Direct Mode
- Register Indirect Mode
- Immediate Mode
- Implicit Addressing Mode
- Displacement Addressing Mode
- Relative Addressing Mode
- Indexed Addressing Mode
- Base Addressing Mode
- Auto Increment and Auto Decrement Mode

Direct mode



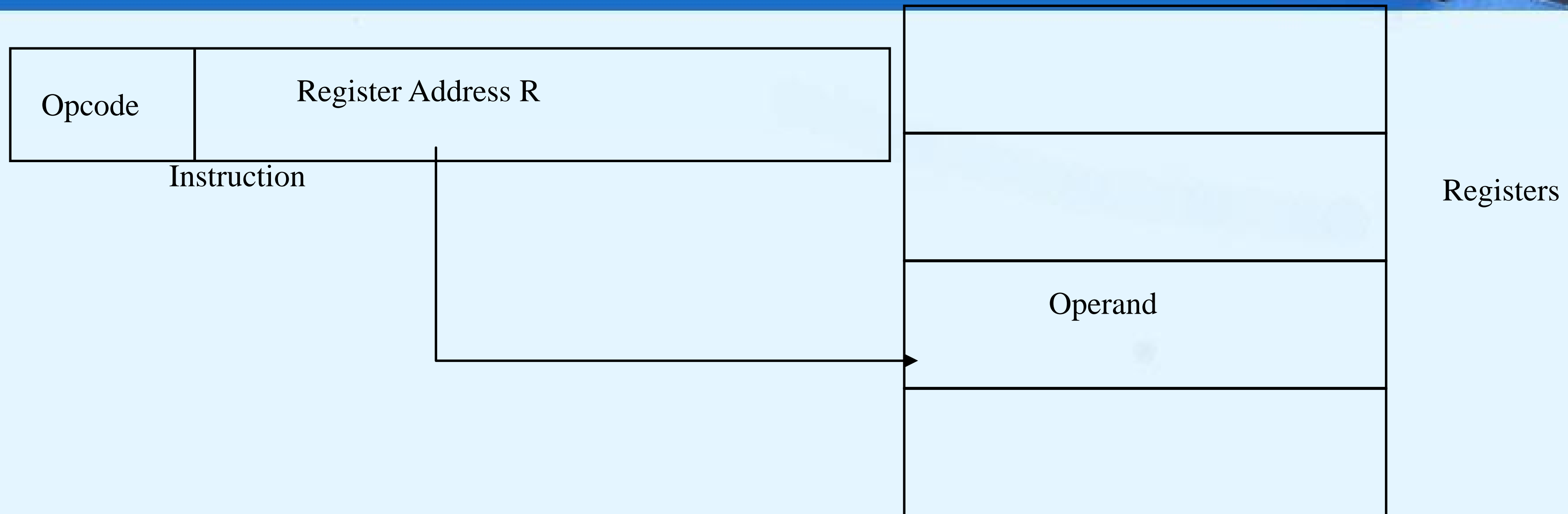
- Instruction includes the A memory address
- LDAC 5 – accesses memory location 5, reads the data (10) and stores the data in the microprocessor's accumulator
- This mode is usually used to load variables and operands into the CPU

Indirect mode



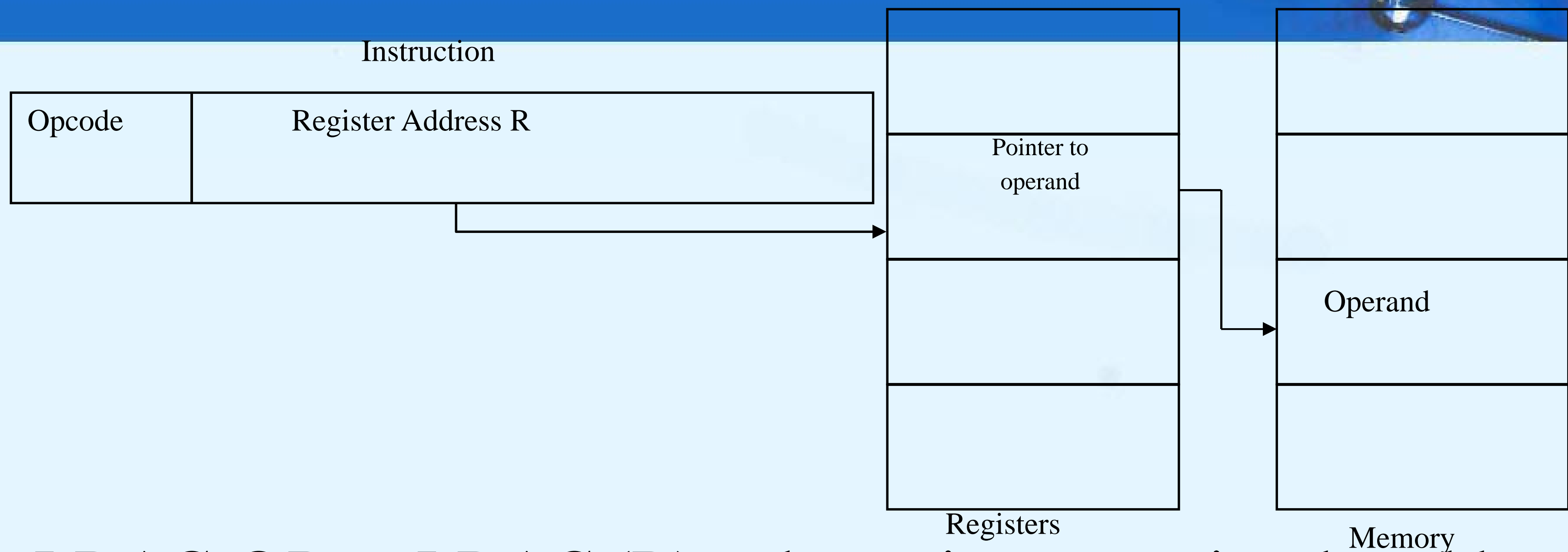
- Starts like the direct mode, but it makes an extra memory access. The address specified in the instruction is not the address of the operand, it is the address of a memory location that contains the address of the operand.
- LDAC @5 or LDAC (5), first retrieves the content of memory location 5, say 10, and then CPU goes to location 10, reads the content (20) of that location and loads the data into the CPU

Register direct mode



- It specifies a register instead a memory address
- LDAC R – if register R contains an value 5, then the value 5 is copied into the CPU's accumulator
- No memory access
- Very fast execution
- Very limited address space

Register indirect mode



- LDAC @R or LDAC (R) – the register contains the address of the operand in the memory
- Register R (selected by the operand), contains value 5 which represents the address of the operand in the memory (10)
- One fewer memory access than indirect addressing

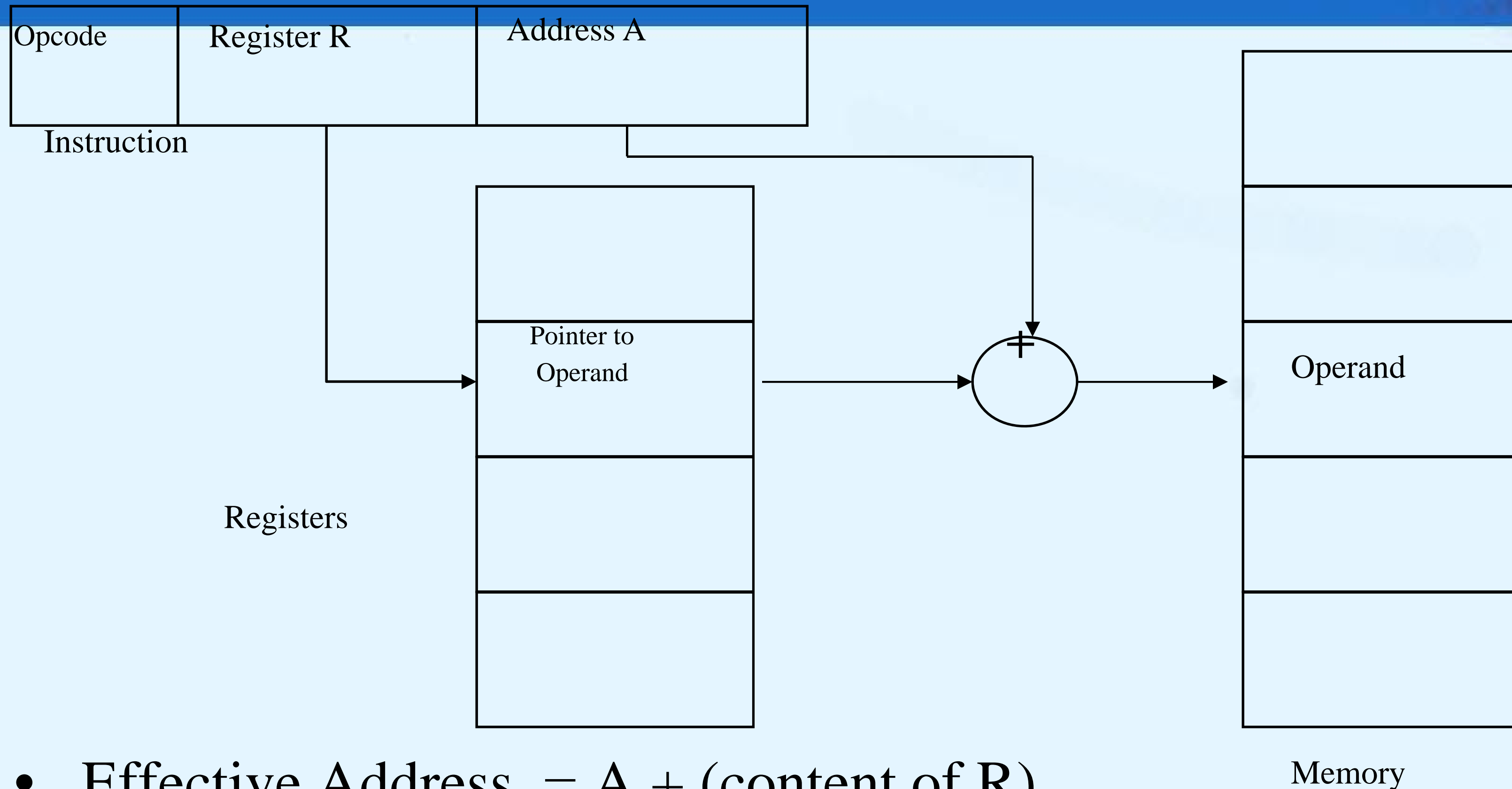
Immediate mode

- The operand is not specifying an address, it is the actual data to be used
- LDAC #5 loads actually value 5 into the CPU's accumulator
- No memory reference to fetch data
- Fast, no memory access to bring the operand

Implicit addressing mode

- Doesn’t explicitly specify an operand
- The instruction implicitly specifies the operand because always applies to a specific register
- This is not used for load instructions
- As an example, consider an instruction CLAC, that is clearing the content of the accumulator in a processor and it is always referring to the accumulator
- This mode is used also in CPUs that do use a stack to store data; they don’t specify an operand because it is implicit that the operand must come from the stack

Displacement addressing mode



- Effective Address = A + (content of R)
- Address field hold two values
  - A = base value
  - R = register that holds displacement
  - or vice versa

Relative addressing mode

- It is a particular case of the displacement addressing, where the register is the program counter; the supplied operand is an offset; Effective Address = A + (PC)
- The offset is added to the content of the CPU's program counter register to generate the required address
- The program counter contains the address of next instruction to be executed, so the same relative instruction will produce different addresses at different locations in the program
- Consider that the relative instruction LDAC $5 is located at memory address 10 and it takes two memory locations; the next instruction is at location 12, so the operand is actually located at (12 +5) 17; the instruction loads the operand at address 17 and stores it in the CPU's accumulator
- This mode is useful for short jumps and reloadable code

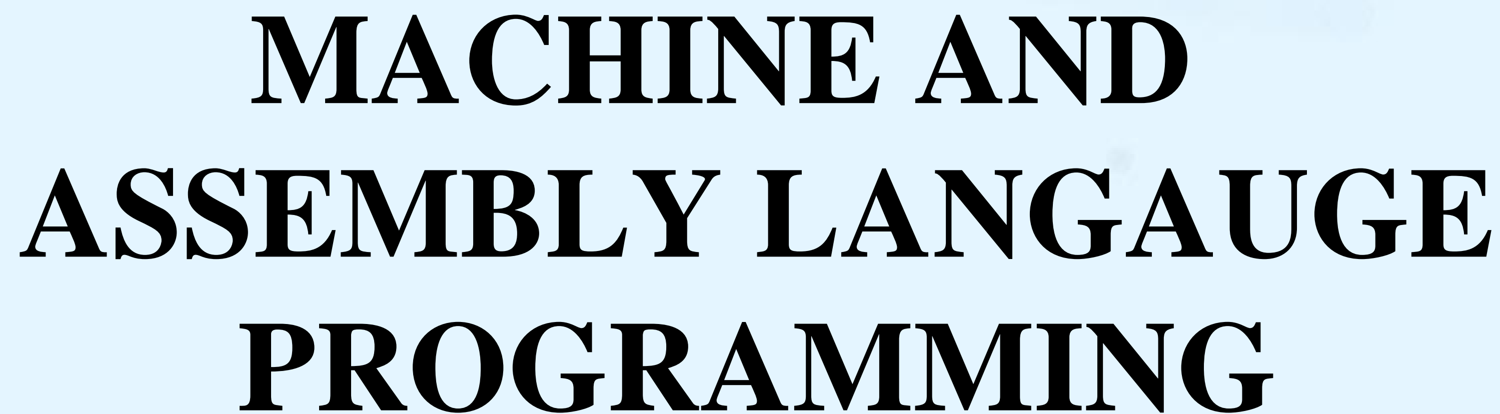
Indexed addressing mode

- Works like relative addressing mode; instead adding the A to the content of program counter (PC), the A is added to the content of an index register
- If the index register contains value 10, then the instruction LDAC 5(X) reads data from memory at location (5+10) 15 and stores it in the accumulator
- Good for accessing arrays
  – Effective Address = A + IndexReg
  – R++

- Works the same with indexed addressing mode, except that the index register is replaced by a base address register
- A holds displacement
- R holds pointer to base address
- R may be explicit or implicit
- e.g. segment registers in 80x86

Auto increment & Decrement addressing mode

- Similar to the register indirect mode except that the register is incremented or decremented after its value is used to access memory.
- Increment and Decrement Instructions are used here.


MACHINE AND
ASSEMBLY LANGAUGE
PROGRAMMING

Computer Operations

- A computer is a programmable electronic device that can store, retrieve, and process data
- Data and instructions to manipulate the data are logically the same and can be stored in the same place
- Store, retrieve, and process are actions that the computer can perform on data

Machine Language

- **Machine language** The instructions built into the hardware of a particular computer
- Initially, humans had no choice but to write programs in machine language because other programming languages had not yet been invented

Machine Language

- Every processor type has its own set of specific machine instructions
- The relationship between the processor and the instructions it can carry out is completely integrated
- Each machine-language instruction does only one very low-level task

Assembly Language

- **Assembly languages** A language that uses mnemonic codes to represent machine-language instructions
  - The programmer uses these alphanumeric codes in place of binary digits
  - A program called an assembler reads each of the instructions in mnemonic form and translates it into the machine-language equivalent

Assembly Language Format

Mnemonic	Operand, Mode Specifier	Meaning of Instruction
STOP		Stop execution
LOADA LOADA	h#008B,i h#008B,d	Load 008B into register A Load the contents of location 8B into register A
STOREA	h#008B,d	Store the contents of register A into location 8B
ADDA ADDA	h#008B,i h#008B,d	Add 008B to register A Add the contents of location 8B to register A
SUBA SUBA	h#008B,i h#008B,d	Subtract 008B from register A Subtract the contents of location 8B from register A
CHARI	h#008B,d	Read a character and store it into byte 8B
CHARO	c#/B/,i h#008B,d	Write the character B Write the character stored in byte 8B
DECI DECO DECO	h#008B,d h#008B,i h#008B,d	Read a decimal number and store it into location 8B Write the decimal number 139 (8B in hex) Write the decimal number stored in 8B

Assembly Process