18CSC203J – COMPUTER ORGANIZATION AND ARCHITECTURE

Course Outcome

CLR-1:*Utilize the functional units of a computer*

CLO-1 :*Identify the computer hardware and how software interacts with computer hardware*

Topics Covered

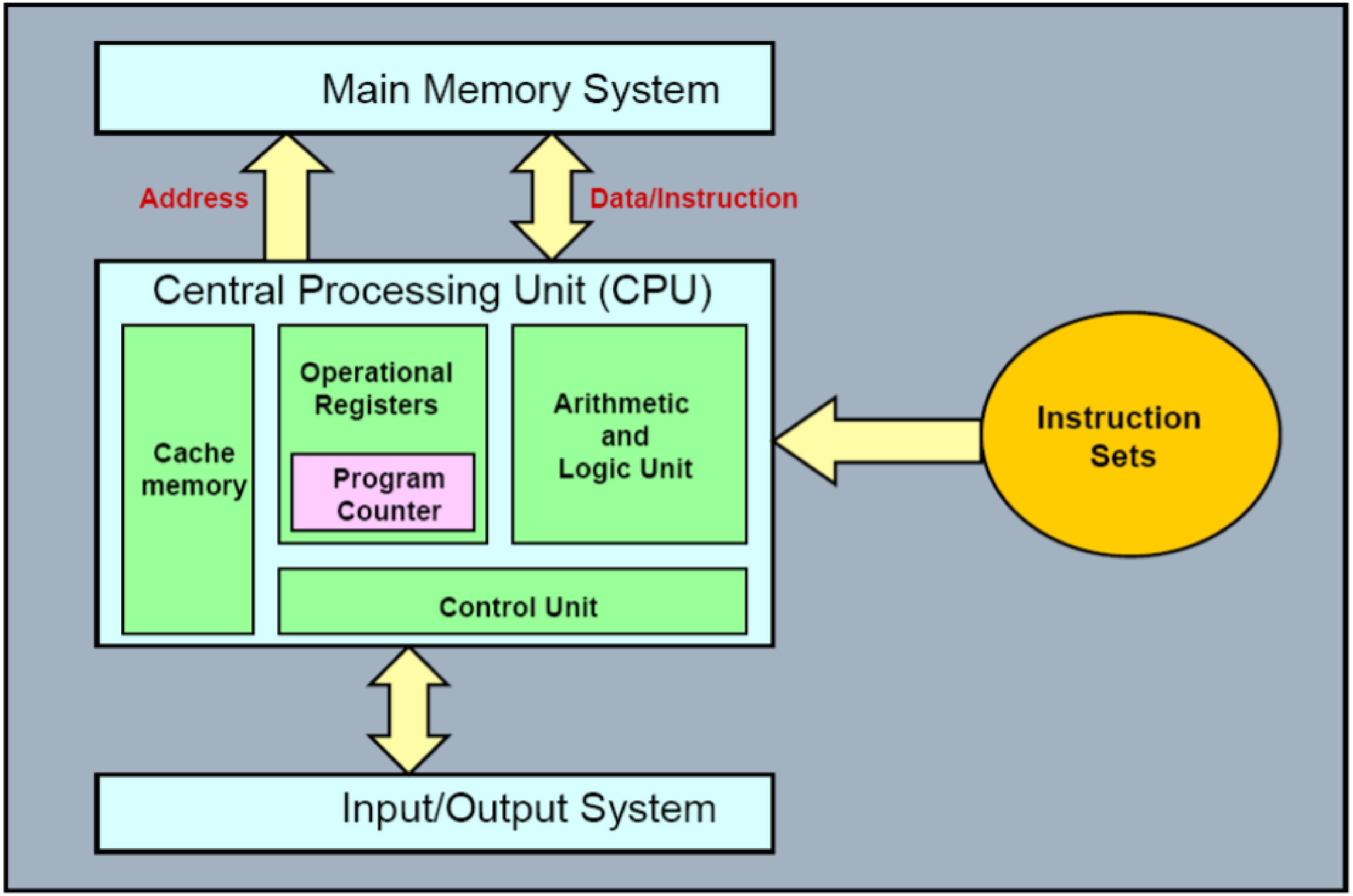
- Functional Units of a computer
- Operational Concepts
- Bus Structures
- Memory Location and Addresses
- Memory Operations
- Instructions and Instruction Sequencing
- Addressing modes
- Problem Solving
- Introduction to Microprocessor
- Introduction to Assembly Language
- Writing of Assembly Language Programming
- ARM Processor: The Thumb instruction set
- Processor and CPU CORES
- Instruction Encoding Format
- Memory load and store instruction in ARM
- Basics of IO Operations

Functional Units of a Computer

FUNCTIONAL UNITS OF COMPUTER

- **Input Unit**
- **Output Unit**
- **Central processing Unit (ALU and Control Units)**
- **Memory**
- **Bus Structure**

Basic Functional Unit of a Computer



Main Memory System

Address

Data/Instruction

Central Processing Unit (CPU)

Cache memory

Operational Registers

Program Counter

Arithmetic and Logic Unit

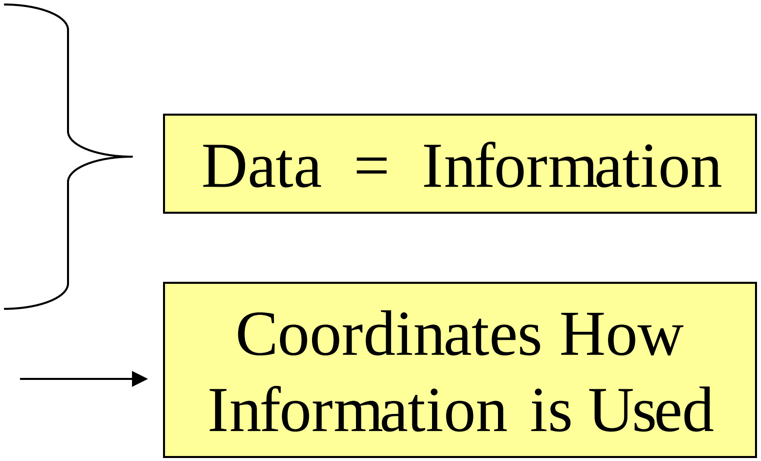
Control Unit

Instruction Sets

Input/Output System

Functions

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



Data = Information

Coordinates How Information is Used

Functions of a computer

The operations performed by a computer using the functional units can be summarized as follows:

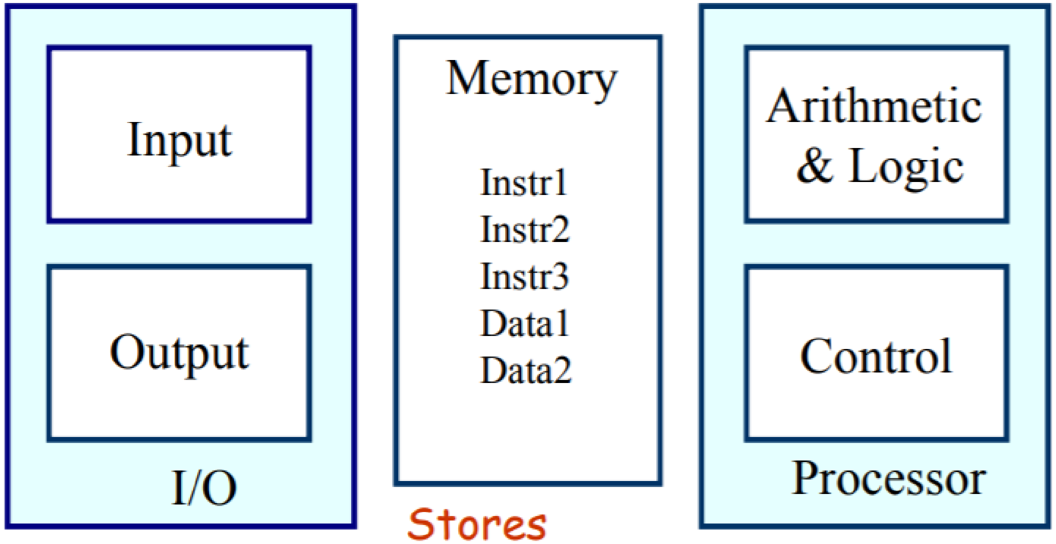
- It accepts information (program and data) through input unit and transfers it to the memory.
- Information stored in the memory is fetched, under program control, into an arithmetic and logic unit for processing.
- Processed information leaves the computer through an output unit.
- The control unit controls all activities taking place inside a computer.

Input unit accepts information:

- Human operators,
- Electromechanical devices (keyboard)
- Other computers

Arithmetic and logic unit(ALU):

- Performs the desired operations on the input information as determined by instructions in the memory



Output unit sends results of processing:

- To a monitor display,
- To a printer

Stores information:

- Instructions,
- Data

Control unit coordinates various actions

- Input,
- Output
- Processing

INPUT UNIT:

- Converts the external world data to a binary format, which can be understood by CPU

Eg: Keyboard, Mouse, Joystick etc

OUTPUT UNIT:

- Converts the binary format data to a format that a common man can understand

Eg: Monitor, Printer, LCD, LED etc

CPU (Central processing Unit)

- The “brain” of the machine

- Responsible for carrying out computational task

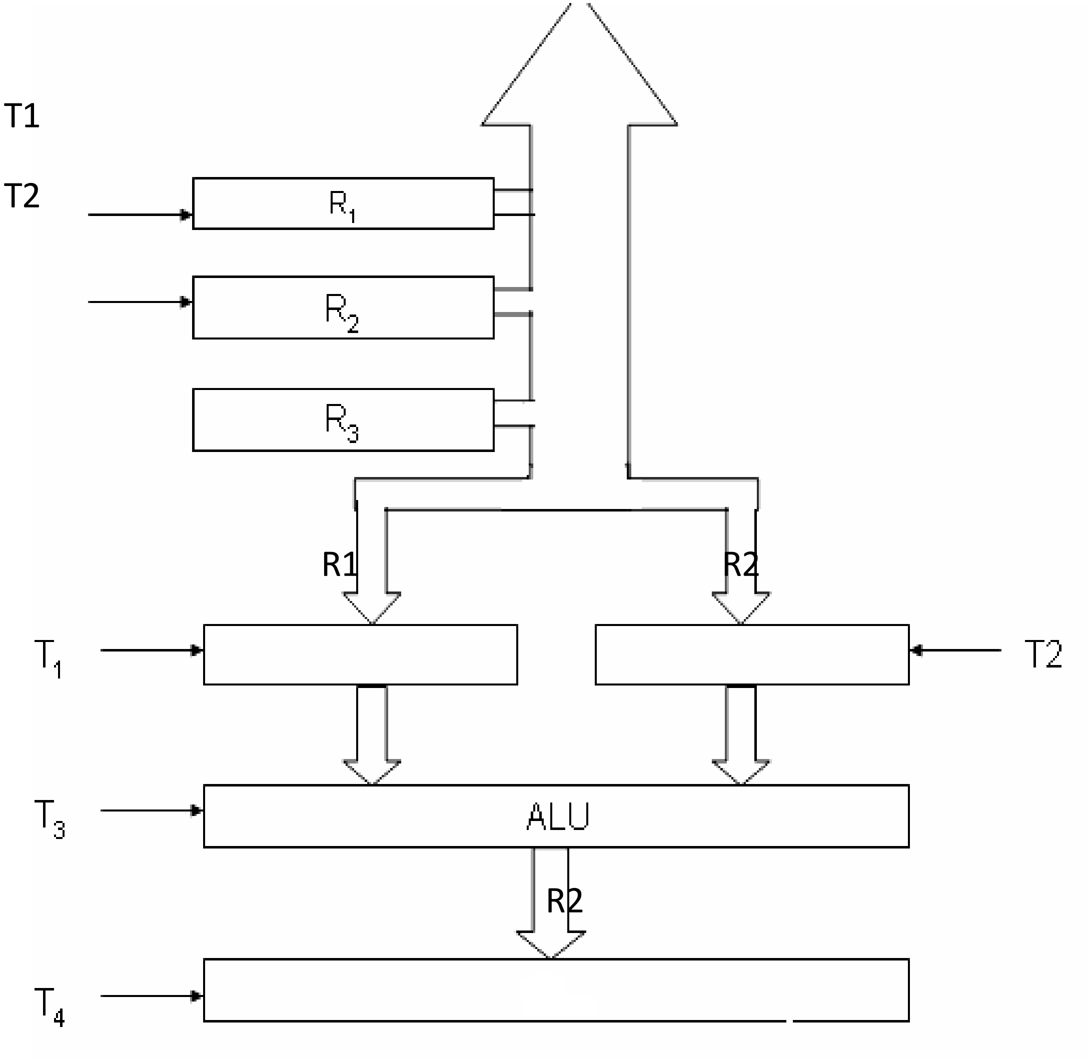
- Contains ALU, CU, Registers

- ALU Performs Arithmetic and logical operations

- CU Provides control signals in accordance with some timings which in turn controls the execution process

- Register Stores data and result and speeds up the operation

CONTROL UNIT



- Control unit works with a reference signal called Processor clock

- Processor divides the operations into basic steps

- Each basic step is executed in one clock cycle

Example

Add R1, R2

T1 → **Enable R1**

T2 → **Enable R2**

T3 → **Enable ALU for addition operation**

T4 → **Enable out put of ALU to store result of the operation**

MEMORY UNIT

- Stores data, results, programs

- Two class of storage (i) Primary (ii) Secondary

- Two types are RAM or R/W memory and ROM read only memory

- ROM is used to store data and program which is not going to change.

- Secondary storage is used for bulk storage or mass storage

Basic Operational Concepts

Basic Function of Computer

- To Execute a given task as per the appropriate program
- Program consists of list of instructions stored in memory

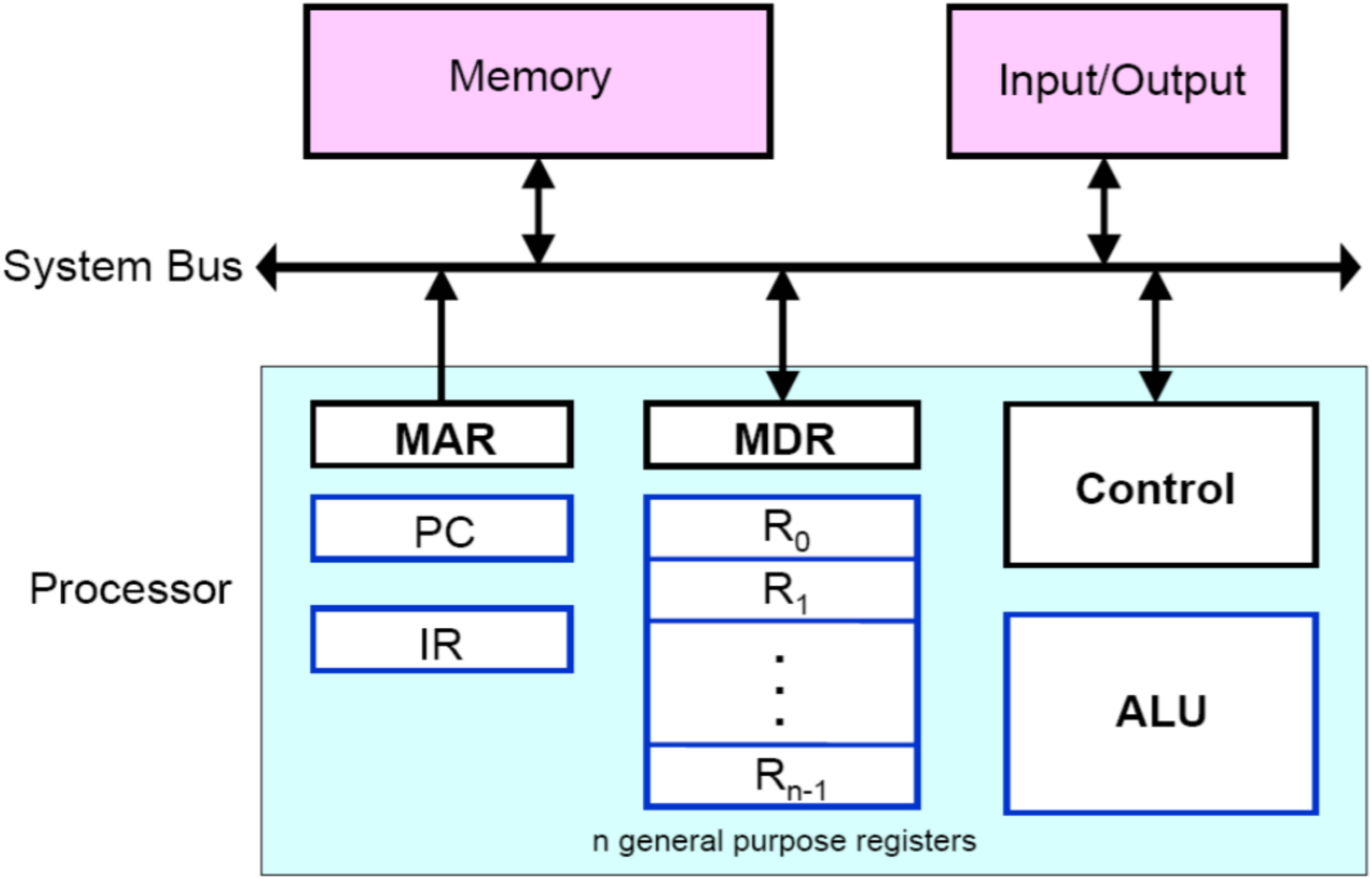
Review

- Activity in a computer is governed by instructions.
- To perform a task, an appropriate program consisting of a list of instructions is stored in the memory.
- Individual instructions are brought from the memory into the processor, which executes the specified operations.
- Data to be used as operands are also stored in the memory.

A Typical Instruction

Add R0, LOCA

- Add the operand at memory location LOCA to the operand in a register R0 in the processor.
- Place the sum into register R0.
- The original contents of LOCA are preserved.
- The original contents of R0 is overwritten.
- Instruction is fetched from the memory into the processor – the operand at LOCA is fetched and added to the contents of R0 – the resulting sum is stored in register R0.



Connections between Processor and memory

Registers

Registers are fast stand-alone storage locations that hold data temporarily. Multiple registers are needed to facilitate the operation of the CPU. Some of these registers are

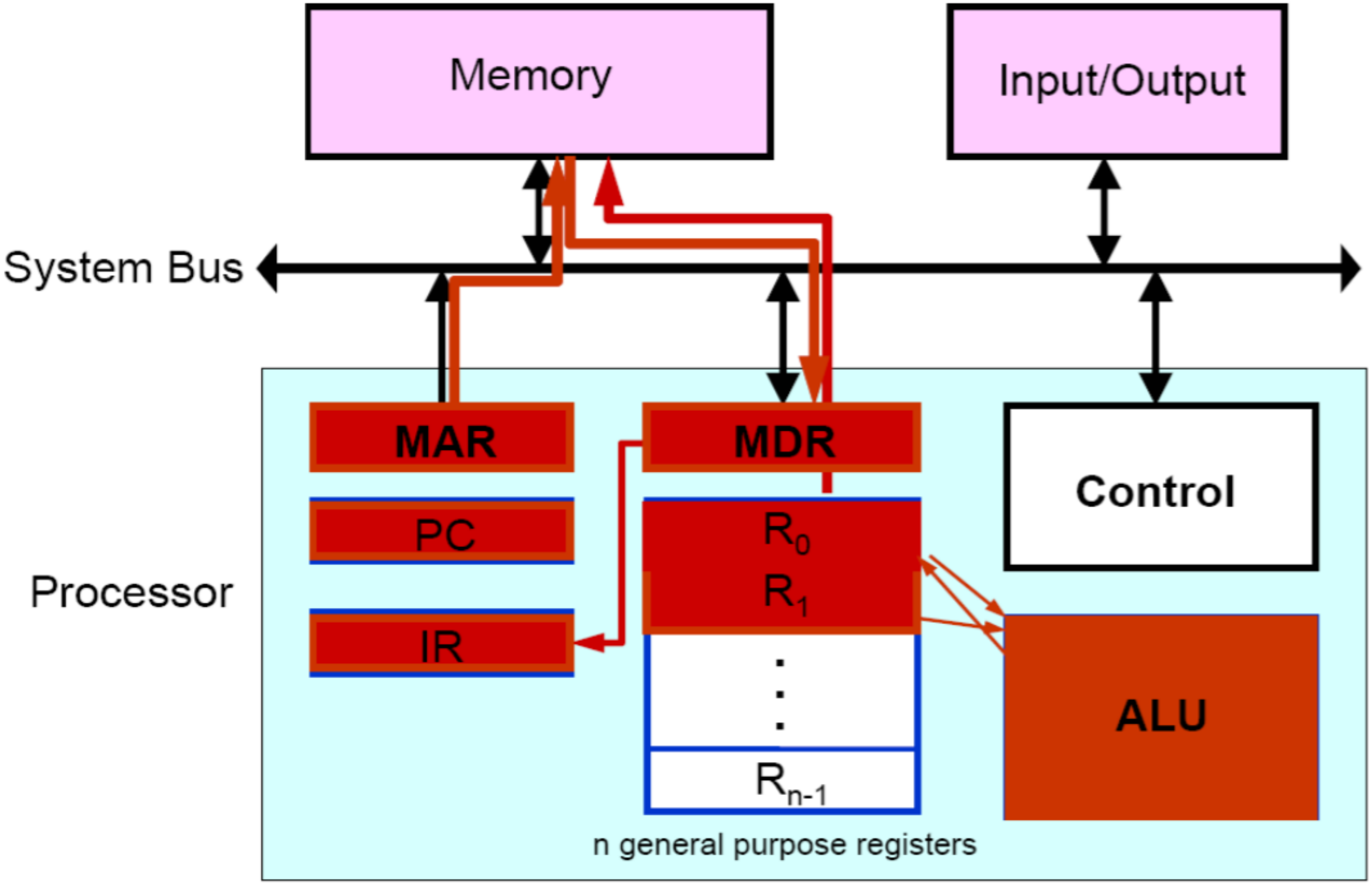
❑ **Two registers-MAR (Memory Address Register) and MDR (Memory Data Register) : To handle the data transfer between main memory and processor. MAR-Holds addresses, MDR-Holds data**

❑ **Instruction register (IR) : Hold the Instructions that is currently being executed**

❑ **Program counter (PC) : Points to the next instructions that is to be fetched from memory**

❑**General-purpose Registers: are used for holding data, intermediate results of operations. They are also known as scratch-pad registers.**

Basic Operational Concepts



INSTRUCTION FETCH – STEPS INVOLVED

- Program gets into the memory through an input device.
- Execution of a program starts by setting the PC to point to the first instruction of the program.
- The contents of PC are transferred to the MAR and a Read control signal is sent to the memory.
- The addressed word (here it is the first instruction of the program) is read out of memory and loaded into the MDR.
- The contents of MDR are transferred to the IR for instruction decoding

INSTRUCTION EXECUTION – STEPS IVOLVED

- The operation field of the instruction in IR is examined to determine the type of operation to be performed by the ALU.
- The specified operation is performed by obtaining the operand(s) from the memory locations or from GP registers.

  1) Fetching the operands from the memory requires sending the memory location address to the MAR and initiating a Read cycle.

  2) The operand is read from the memory into the MDR and then from MDR to the ALU.

INSTRUCTION EXECUTION – STEPS IVOLVED (Contd..)

3) The ALU performs the desired operation on one or more operands 13 fetched in this manner and sends the result either to memory location or to a GP register.

4) The result is sent to MDR and the address of the location where the result is to be stored is sent to MAR and Write cycle is initiated.

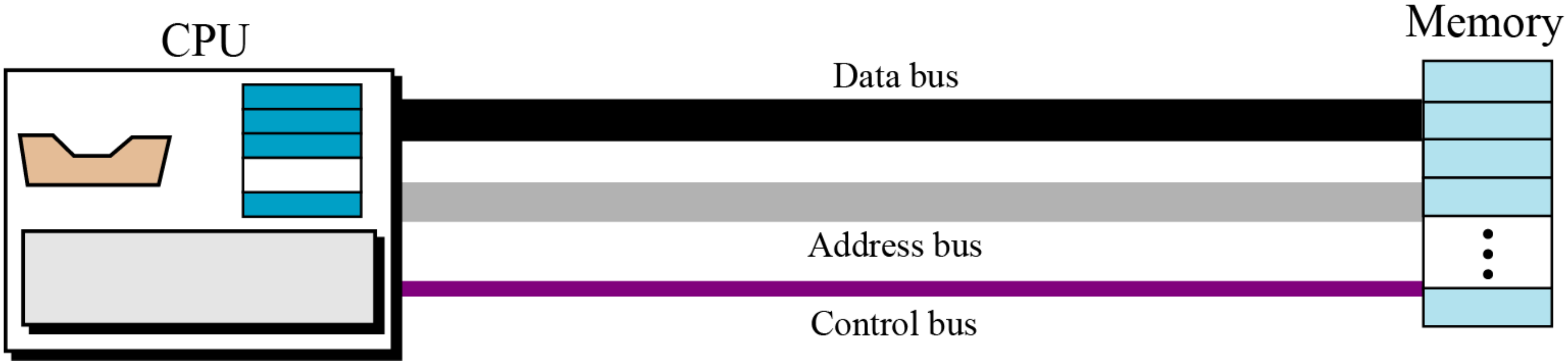
Thus, the execute cycle ends for the current instruction and the PC is incremented to point to the next instruction for a new fetch cycle.

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

BUS STRUCTURE
Connecting CPU and memory

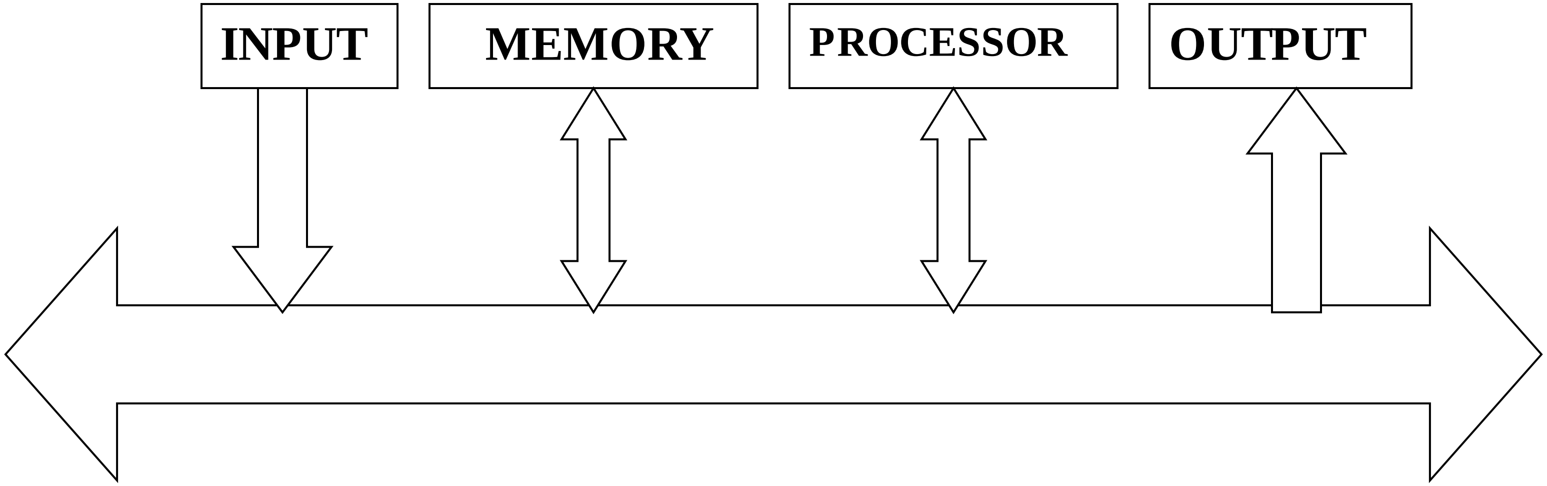
The CPU and memory are normally connected by three groups of connections, each called a **bus**: *data bus*, *address bus* and *control bus*



Connecting CPU and memory using three buses

BUS STRUCTURE

- Group of wires which carries information form CPU to peripherals or vice – versa
- **Single bus structure**: Common bus used to communicate between peripherals and microprocessor



SINGLE BUS STRUCTURE

Drawbacks of the Single Bus Structure

- The devices connected to a bus vary widely in their speed of operation.
  - Some devices are relatively slow, such as printer and keyboard.
  - Some devices are considerably fast, such as optical disks.
  - Memory and processor units operate are the fastest parts of a computer.
- Efficient transfer mechanism thus is needed to cope with this problem.
  - A common approach is to include buffer registers with the devices to hold the information during transfers .
  - An another approach is to use two-bus structure and an additional transfer mechanism

TWO BUS STRUCTURE:

- **In two – bus structure** : One bus can be used to fetch instruction other can be used to fetch data, required for execution. The bus is said to perform two distinct functions The main advantage of this structure is good operating speed but on account of more cost.

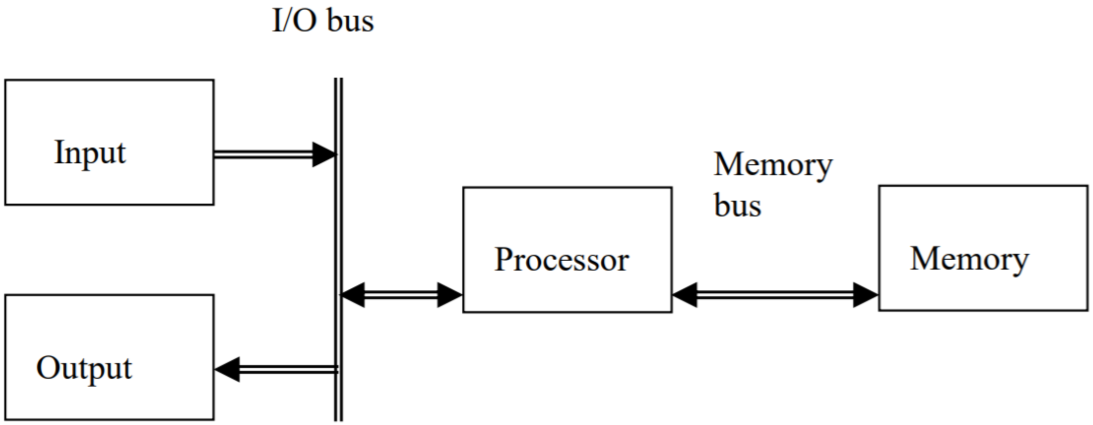
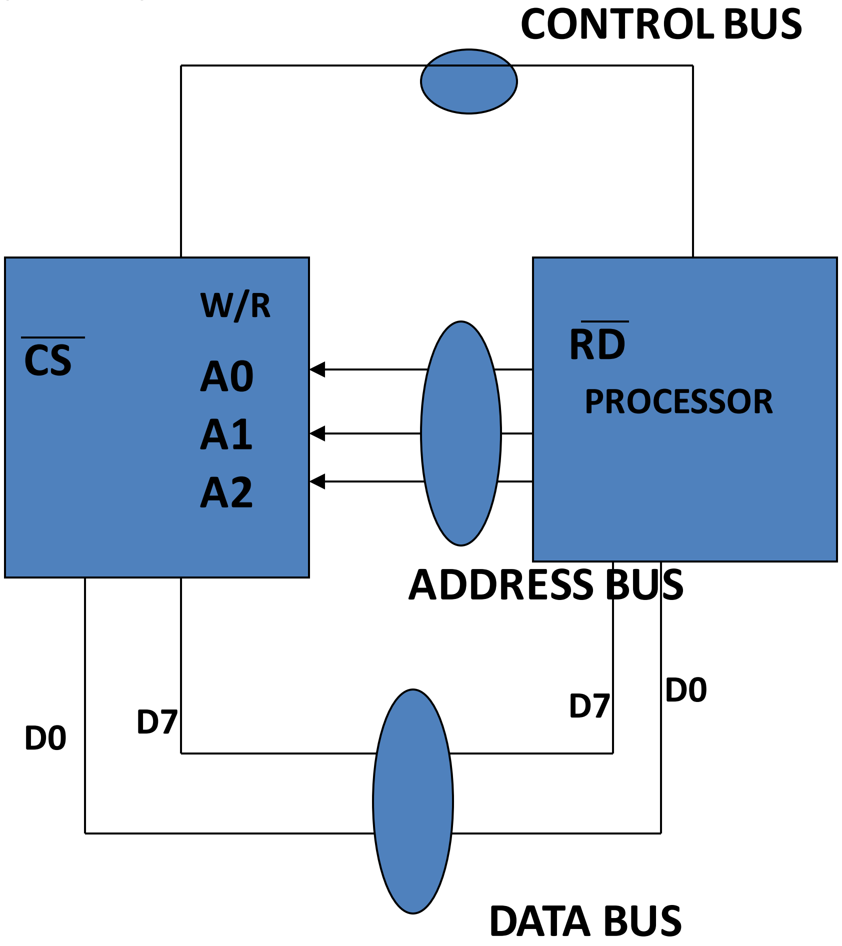


Figure 2.2 Two-bus Structure

MULTI BUS STRUCTURE

To improve performance **multi bus** structure can be used.



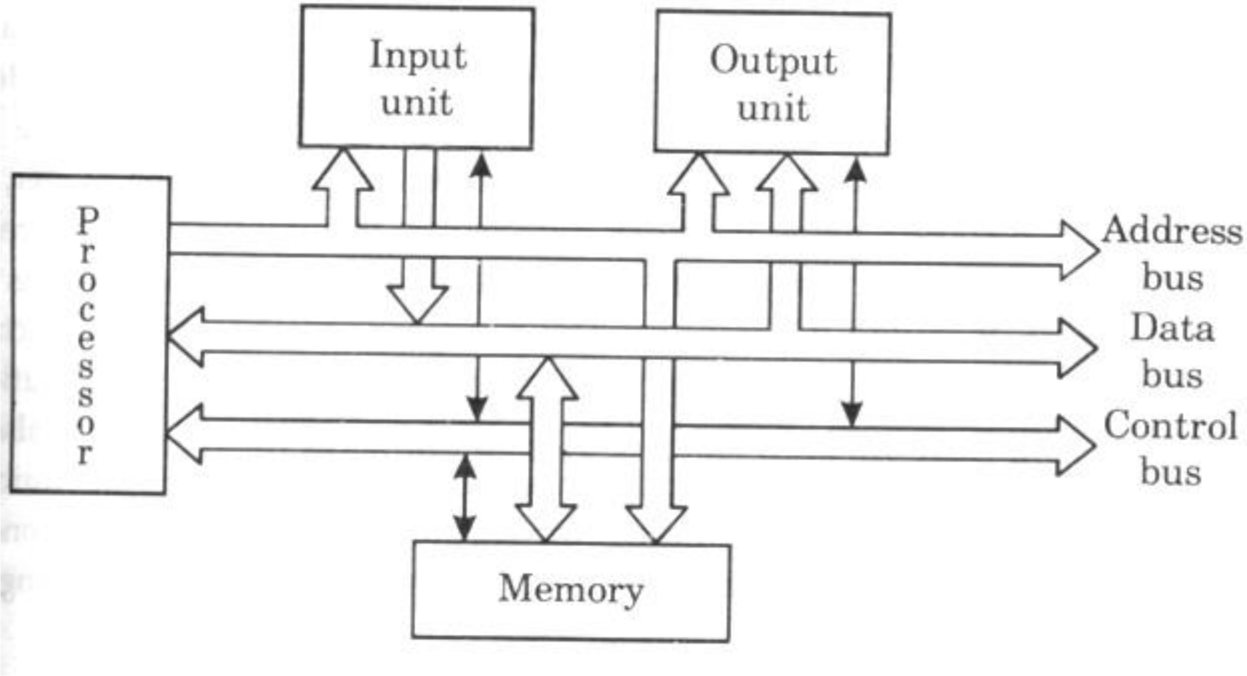
A_2	A_1	A_0	Selected location
0	0	0	0th Location
0	0	1	1st Location
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

- $2^3 = 8$ i.e. 3 address line is required to select 8 location

- In general $2^x = n$ where x number of address lines (address bit) and n is number of location

- **Address bus** : unidirectional : group of wires which carries address information bits form processor to peripherals (16,20,24 or more parallel signal lines)
- **Data bus**: bidirectional : group of wires which carries data information bit form processor to peripherals and vice – versa
- **Control bus**: bidirectional: group of wires which carries control signals form processor to peripherals and vice – versa

Figure below shows address, data and control bus and their connection with peripheral and microprocessor



Single bus structure showing the details of connection

Memory Locations and Addresses

Memory Location and Addresses

- Memory consists of many millions of storage cells, each of which can store 1 bit.
- Data is usually accessed in *n*-bit groups. *n* is called word length.
- The memory of a computer can be schematically represented as a collection of words as shown in Figure 1.

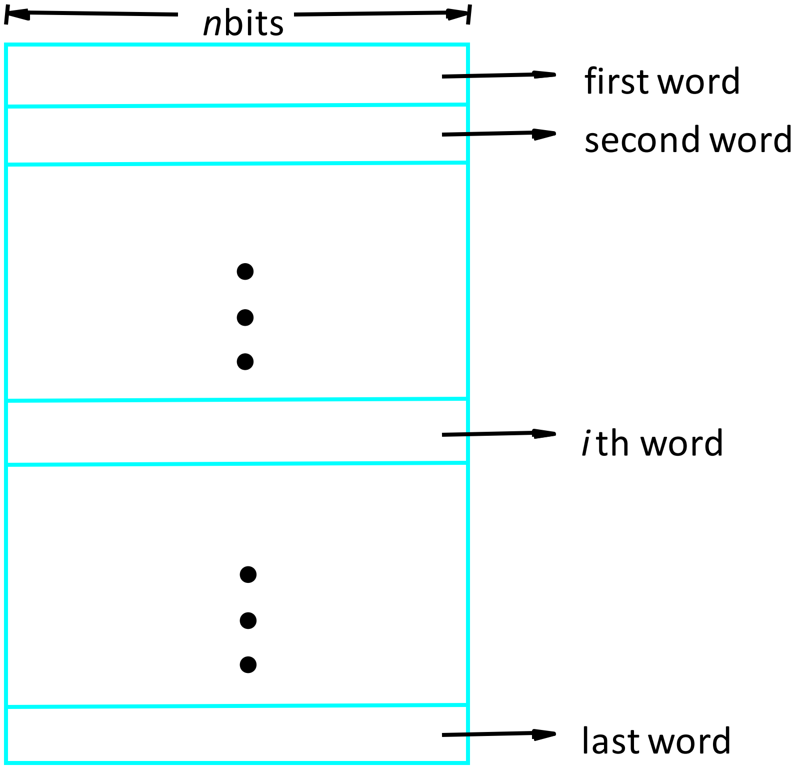


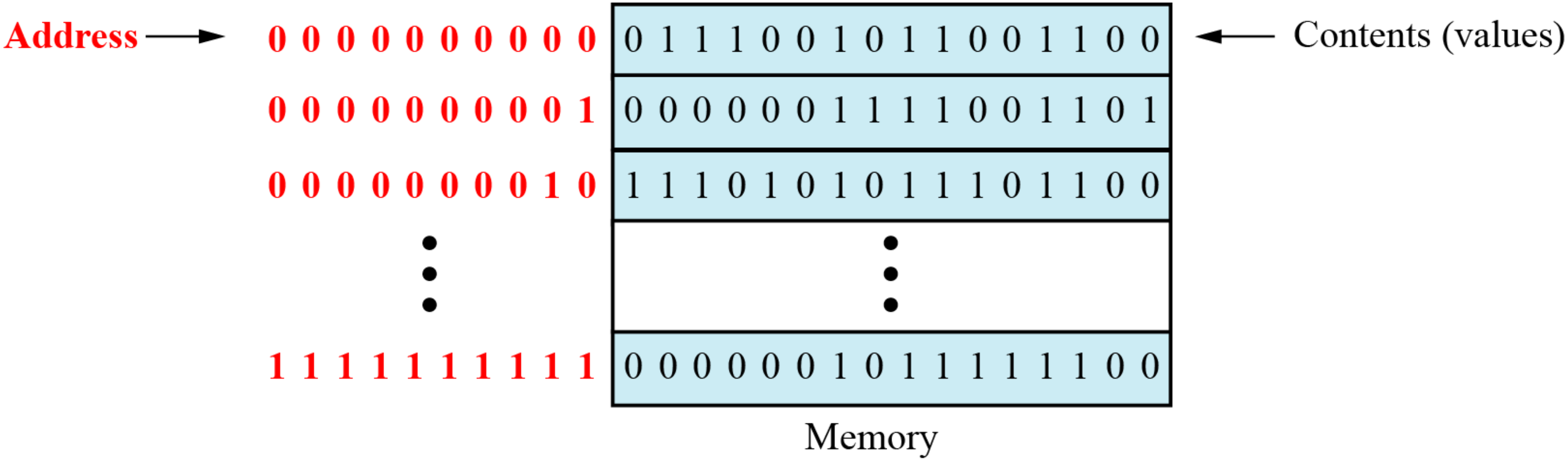
Figure 1 Main Memory words.

MEMORY LOCATIONS AND ADDRESSES

- **Main memory** is the second major subsystem in a computer. It consists of a collection of storage locations, each with a unique identifier, called an **address**.

- Data is transferred to and from memory in groups of bits called **words**. A word can be a group of 8 bits, 16 bits, 32 bits or 64 bits (and growing).

- If the word is 8 bits, it is referred to as a **byte**. The term “byte” is so common in computer science that sometimes a 16-bit word is referred to as a 2-byte word, or a 32-bit word is referred to as a 4-byte word.



Address	Contents (values)
0 0 0 0 0 0 0 0 0 0	0 1 1 1 0 0 1 0 1 1 0 0 1 1 0 0
0 0 0 0 0 0 0 0 0 1	0 0 0 0 0 0 1 1 1 1 0 0 1 1 0 1
0 0 0 0 0 0 0 0 1 0	1 1 1 0 1 0 1 0 1 1 1 0 1 1 0 0
⋮	⋮
1 1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 1 0 1 1 1 1 1 1 0 0

Main memory

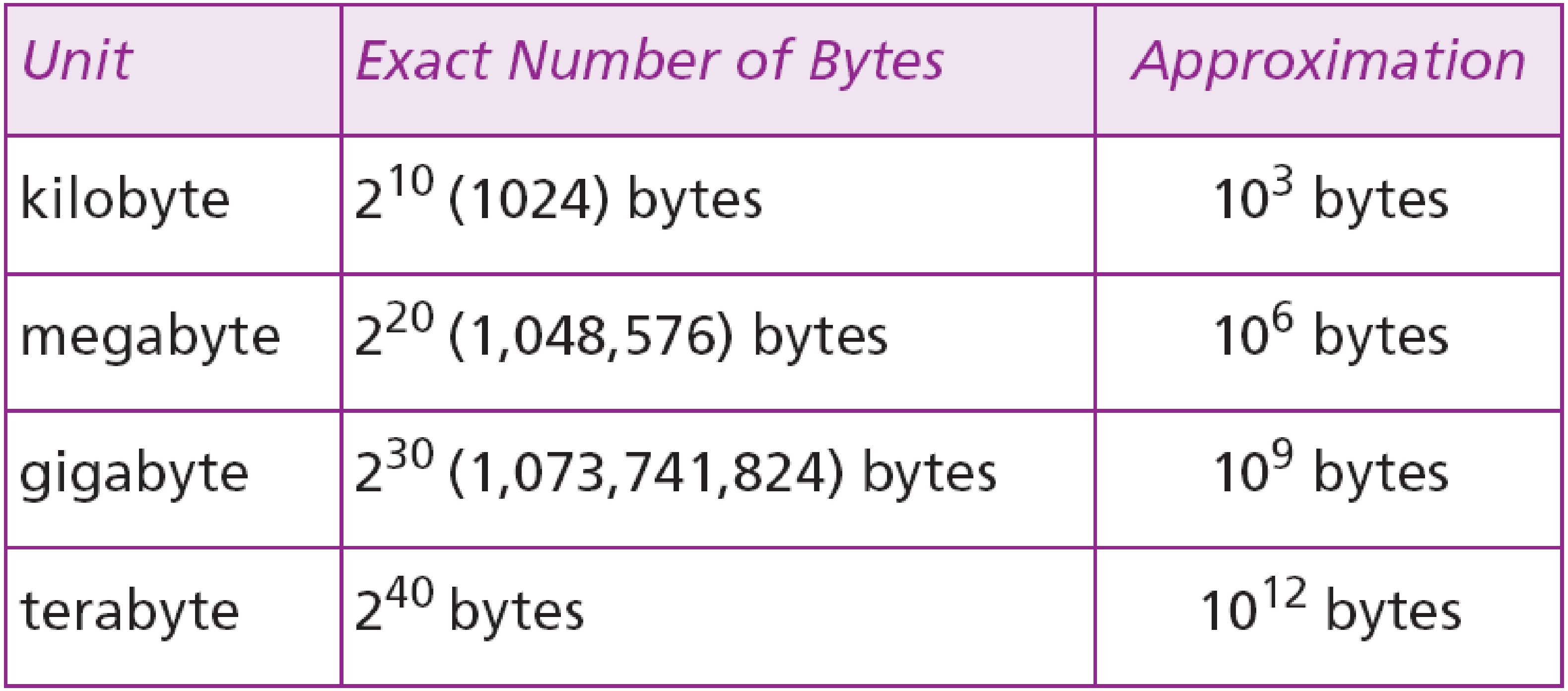
Address space

- To access a word in memory requires an identifier. Although programmers use a name to identify a word (or a collection of words), at the hardware level each word is identified by an address.

- The total number of uniquely identifiable locations in memory is called the **address space**.

- For example, a memory with 64 kilobytes (16 address line required) and a word size of 1 byte has an address space that ranges from 0 to 65,535.

Table 5.1 Memory units



Unit	*Exact Number of Bytes*	*Approximation*
kilobyte	2^{10} (1024) bytes	10^{3} bytes
megabyte	2^{20} (1,048,576) bytes	10^{6} bytes
gigabyte	2^{30} (1,073,741,824) bytes	10^{9} bytes
terabyte	2^{40} bytes	10^{12} bytes

Memory addresses are defined using unsigned binary integers.

Example 1

A computer has 32 MB (megabytes) of memory. How many bits are needed to address any single byte in memory?

Solution

The memory address space is 32 MB, or 2^{25} ($2^5 \times 2^{20}$). This means that we need $\log_2 2^{25}$, or **25 bits**, to address each byte.

Example 2

A computer has 128 MB of memory. Each word in this computer is eight bytes. How many bits are needed to address any single word in memory?

Solution

The memory address space is 128 MB, which means 2^{27}. However, each word is eight (2^3) bytes, which means that we have 2^{24} words. This means that we need $\log_2 2^{24}$, or **24 bits**, to address each word.

MEMORY OPERATIONS

- Today, **general-purpose computers** use a set of instructions called a **program** to process data.

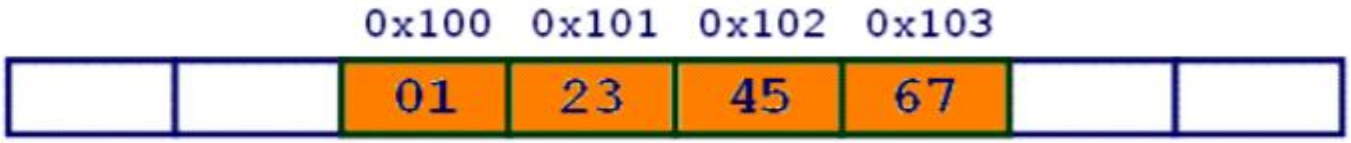
- A computer executes the program to create output data from input data

- Both program instructions and data operands are stored in memory

- Two basic operations requires in memory access
  - **Load operation (Read or Fetch)**-Contents of specified memory location are read by processor
  - **Store operation (Write)-** Data from the processor is stored in specified memory location

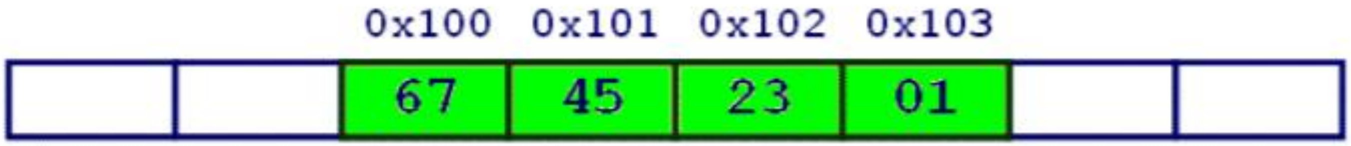
Assignment of Byte Address

- **Big-endian** and **little-endian** are terms that describe the order in which a sequence of bytes are stored in computer **memory**.
- **Big-endian** is an order in which the "**big**end" (most significant value in the sequence) is stored first (at the lowest storage address).
- **Little-endian** is an order in which the “**Little** end" (least significant value in the sequence) is stored first (at the lowest storage address).



		0x100	0x101	0x102	0x103		
		01	23	45	67		

Big Endian



		0x100	0x101	0x102	0x103		
		67	45	23	01		

Little Endian

Assignment of byte addresses

- Little Endian (e.g., in DEC, Intel)
  » low order byte stored at lowest address
  » byte0 byte1 byte2 byte3

- Eg: 46,78,96,54 (32 bit data)
- H BYTE ← L BYTE

8000	54
8001	96
8002	78
8003	46
8004	

Big Endian

- Big Endian (e.g., in IBM, Motorolla, Sun, HP)

  » high order byte stored at lowest address

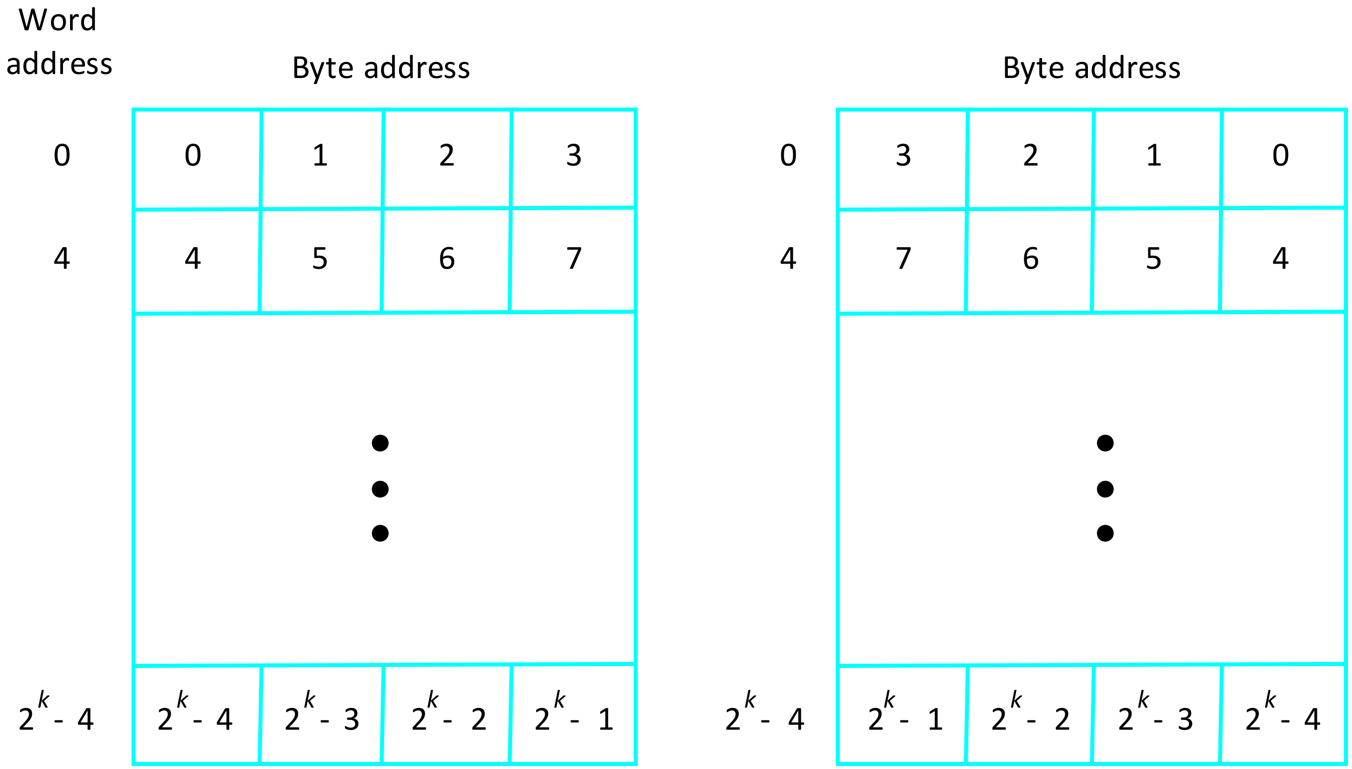
  » byte3 byte2 byte1 byte0

- Programmers/protocols should be careful when transferring binary data between Big Endian and Little Endian machines

Big-Endian and Little-Endian Assignments

Big-Endian: lower byte addresses are used for the most significant bytes of the word

Little-Endian: opposite ordering. lower byte addresses are used for the less significant bytes of the word



Word address	Byte address			
0	0	1	2	3
4	4	5	6	7
	•	•	•	
$2^k - 4$	$2^k - 4$	$2^k - 3$	$2^k - 2$	$2^k - 1$

(a) Big-endian assignment

	Byte address			
0	3	2	1	0
4	7	6	5	4
	•	•	•	
$2^k - 4$	$2^k - 1$	$2^k - 2$	$2^k - 3$	$2^k - 4$

(b) Little-endian assignment

Byte and word addressing.

- In case of 16 bit data, aligned words begin at byte addresses of 0,2,4,…………………………
- In case of 32 bit data, aligned words begin at byte address of 0,4,8,…………………………
- In case of 64 bit data, aligned words begin at byte addresses of 0,8,16,………………………..
- In some cases words can start at an arbitrary byte address also then, we say that word locations are unaligned

INSTRUCTION AND INSTRUCTION SEQUENCING

Introduction

- **Instruction**: is command to the microprocessor to perform a given task on specified data.
- **Instruction Set**: The entire group of these instructions are called instruction set.
- **instruction sequencing :** The order in which the instructions in a program are carried out.

Types of Instructions

- Most computer instructions are classified into 3 categories
  - Data transfer Instructions

    To Transfer data from one location to another
  - Data Manipulation Instructions

    To perform the operations by the ALU
  - Program control Instructions

    To control the system

Data transfer Instructions

They are also called copy instructions.

Some instructions in 8086:

- MOV -Copy from the source to the destination
- LDA - Load the accumulator
- STA - Store the accumulator
- PUSH - Push the register pair onto the stack
- POP - Pop off stack to the register pair

Data Manipulation Instructions

- To perform the operations by the ALU

- Three categories:
  - Arithmetic Instructions
  - Logical and bit manipulation instructions
  - Shift instructions

Arithmetic Instructions

Used to perform arithmetic operations

Some instruction in 8086

INC → Increment the data by 1

DEC → Decreases data by 1

ADD → perform sum of data

ADC → Add with carry bit.

MUL → perform multiplication

Logical and bit manipulation instructions

Used to perform logical operations

Some instructions are:

AND ➔ bitwise AND operation

OR ➔ bitwise AND operation

NOT ➔ invert each bit of a byte or word

XOR ➔ Exclusive-OR operation over each bit

Shift instructions

used for shifting or rotating the contents of the register

Some instructions are:

SHR → shift bits towards the right and put zero(S) in MSBs

ROL → rotate bits towards the left, i.e. MSB to LSB and to Carry Flag [CF]

RCL → rotate bits towards the left, i.e. MSB to CF and CF to LSB.

Instruction Formats

(Types of instruction based on the address field)

- A instruction in computer comprises of groups called fields.
- These field contains different information
- The most common fields are:

  Operation field : specifies the operation to be performed like addition.

  Address field : contain the location of operand

  Mode field : specifies how to find the operand

- A instruction is of various length depending upon the number of addresses it contain.
- On the basis of number of address, instruction are classified as:
    - **Zero Address Instructions**
    - **One Address Instructions**
    - **Two Address Instructions**
    - **Three Address Instructions**

Zero Address Instructions

Used in stack based computers which do not use address field in instruction

- Location of all operands are defined implicitly
- Operands are stored in a structure called pushdown stack

Example: Evaluate (A+B) ∗ (C+D)

- Using Zero-Address instruction

```
1.  PUSH   A                  ; TOS ← A
2.  PUSH   B                  ; TOS ← B
3.  ADD                       ; TOS ← (A + B)
4.  PUSH   C                  ; TOS ← C
5.  PUSH   D                  ; TOS ← D
6.  ADD                       ; TOS ← (C + D)
7.  MUL                       ; TOS ← (C+D)∗(A+B)
8.  POP    X                  ; M[X] ← TOS
```

One address Instruction

- This use a implied ACCUMULATOR register for data manipulation.
- One operand is in accumulator and other is in register or memory location.

Example: Evaluate (A+B) ∗ (C+D)

- Using One-Address Instruction

```
1.  LOAD    A        ; AC ← M[A]
2.  ADD     B        ; AC ← AC + M[B]
3.  STORE   T        ; M[T] ← AC
4.  LOAD    C        ; AC ← M[C]
5.  ADD     D        ; AC ← AC + M[D]
6.  MUL     T        ; AC ← AC ∗ M[T]
7.  STORE   X        ; M[X] ← AC
```

Two Address Instruction

This is common in commercial computers.

Here two address can be specified in the instruction.

Example: Evaluate (A+B) ∗ (C+D)

Using Two address Instruction:

1. MOV R1,A ; R1=M[A]
2. ADD R1,B ;R1=R1+M[B]
3. MOV R2,C ;R2=M[C]
4. ADD R2,D ;R2=R2+M[D]
5. MUL R1,R2 ; R1=R1*R2
6. MOV X,R1 ; M[X]=R1

Three Address Instruction

This has three address field to specify a register or a memory location.

Program created are much short in size

creation of program much easier

does not mean that program will run much faster

Example: Evaluate (A+B) ∗ (C+D)

Using Three address Instruction

1. ADD R1,A,B ;R1=M[A]+M[B]
2. ADD R2,C,D ;R2=M[C]+M[D]
3. MUL X,R1,R2 ;M[X]=R1*R2

Instruction Cycle

- the basic operational process of a computer.
- also known as **fetch-decode-execute cycle**
- This process is repeated continuously by CPU from boot up to shut down of computer.

steps that occur during an instruction cycle:

1. **Fetch the Instruction**
2. **Decode the Instruction**
3. **Read the Effective Address**
4. **Execute the Instruction**

1. **Fetch the Instruction**

The instruction is fetched from memory address that is stored in PC(Program Counter) and stored in the (instruction register) IR.

At the end of the fetch operation, PC is incremented by 1 and it then points to the next instruction to be executed.

2. **Decode the Instruction**

The instruction in the IR is decoded(Interpreted).

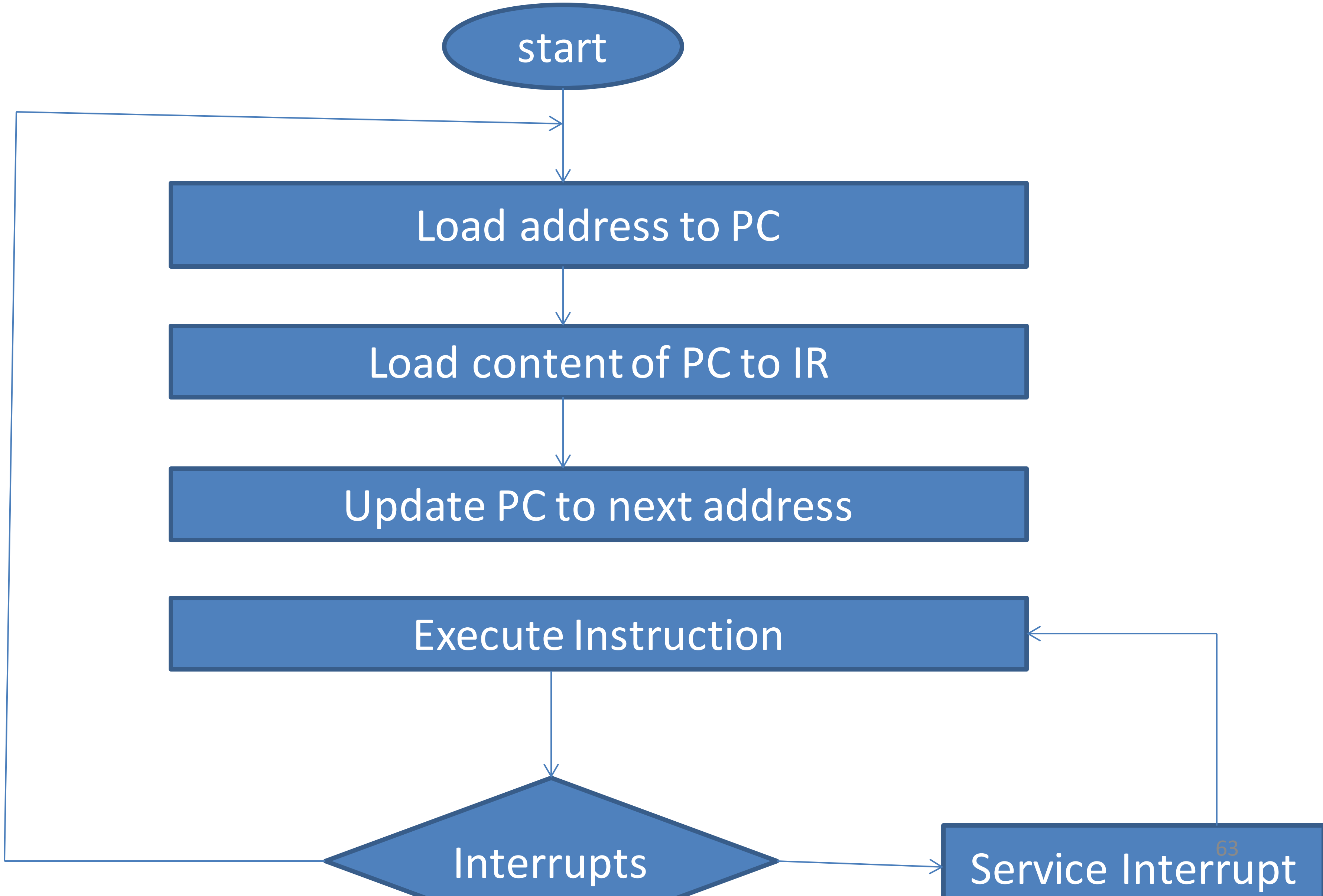
3. Read the Effective Address

If the instruction has an indirect address, the effective address is read from the memory. Otherwise operands are directly read in case of immediate operand instruction.

4. Execute the Instruction

The Control Unit passes the information in the form of control signals to the functional unit of CPU. The result generated is stored in main memory or sent to an output device.

The cycle is then repeated by fetching the next instruction. Thus in this way the instruction cycle is repeated continuously.


start
Load address to PC
Load content of PC to IR
Update PC to next address
Execute Instruction
Interrupts
Service Interrupt

Addressing Modes

Different ways in which the location of the operand is specified in an instruction is referred as addressing modes

The purpose of using addressing mode is:

To give the programming versatility to the user.

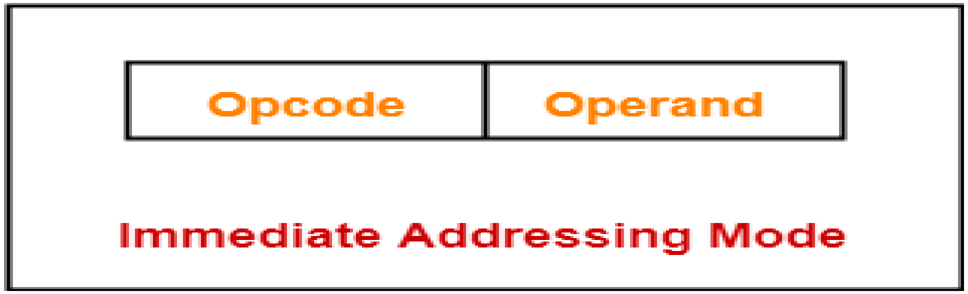
To reduce the number of bits in addressing field of instruction.

Types of Addressing Modes

- Immediate Addressing
- Direct Addressing
- Indirect Addressing
- Register Addressing
- Register Indirect Addressing
- Relative Addressing
- Indexed Addressing
- Auto Increment
- Auto Decrement

Immediate Addressing

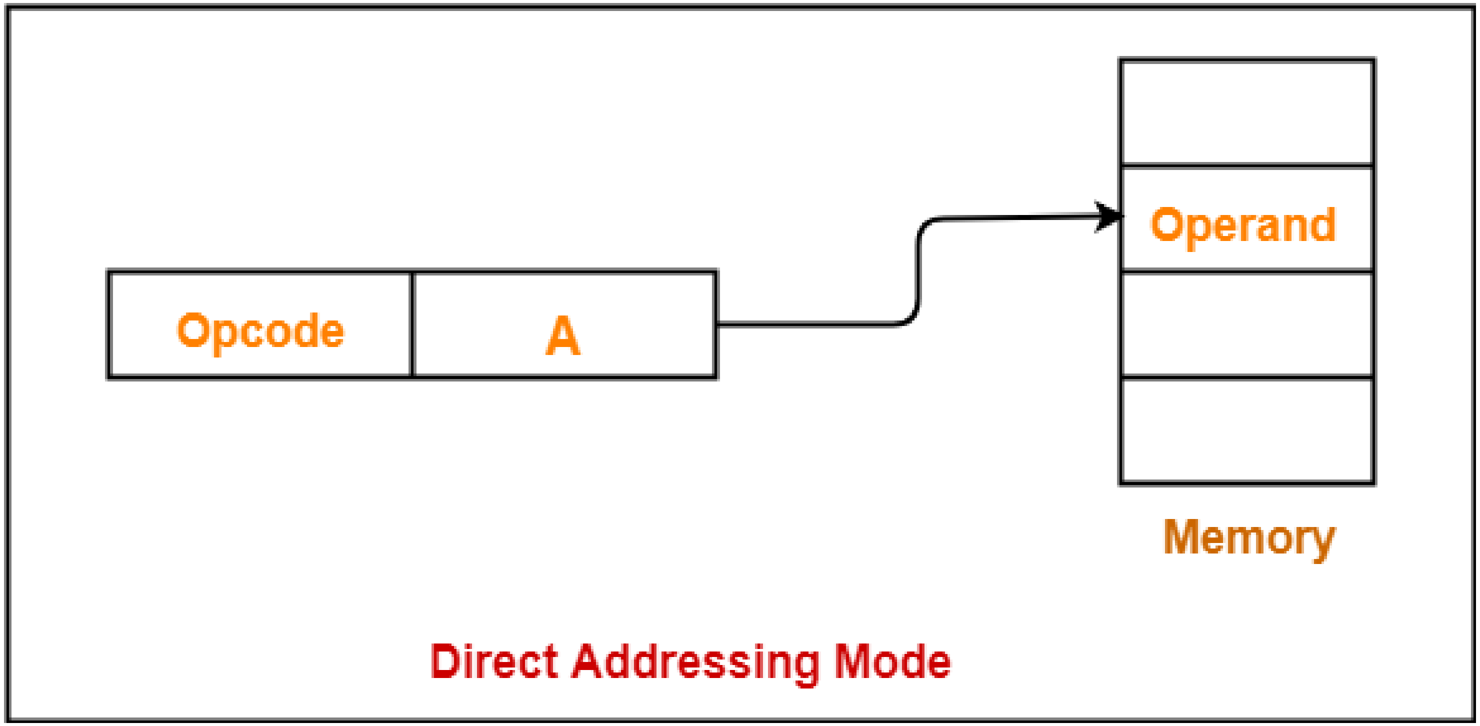
- Operand is given explicitly in the instruction
- e.g. ADD 5
  - Add 5 to contents of accumulator
  - 5 is operand
- No memory reference to fetch data
- Fast
- Limited range



Immediate Addressing Mode

Direct Addressing

- Address field contains address of operand
- Effective address (EA) = address field (A)
- e.g. ADD A
  - Add contents of cell A to accumulator
  - Look in memory at address A for operand
- Single memory reference to access data
- No additional calculations to work out effective address
- Limited address space

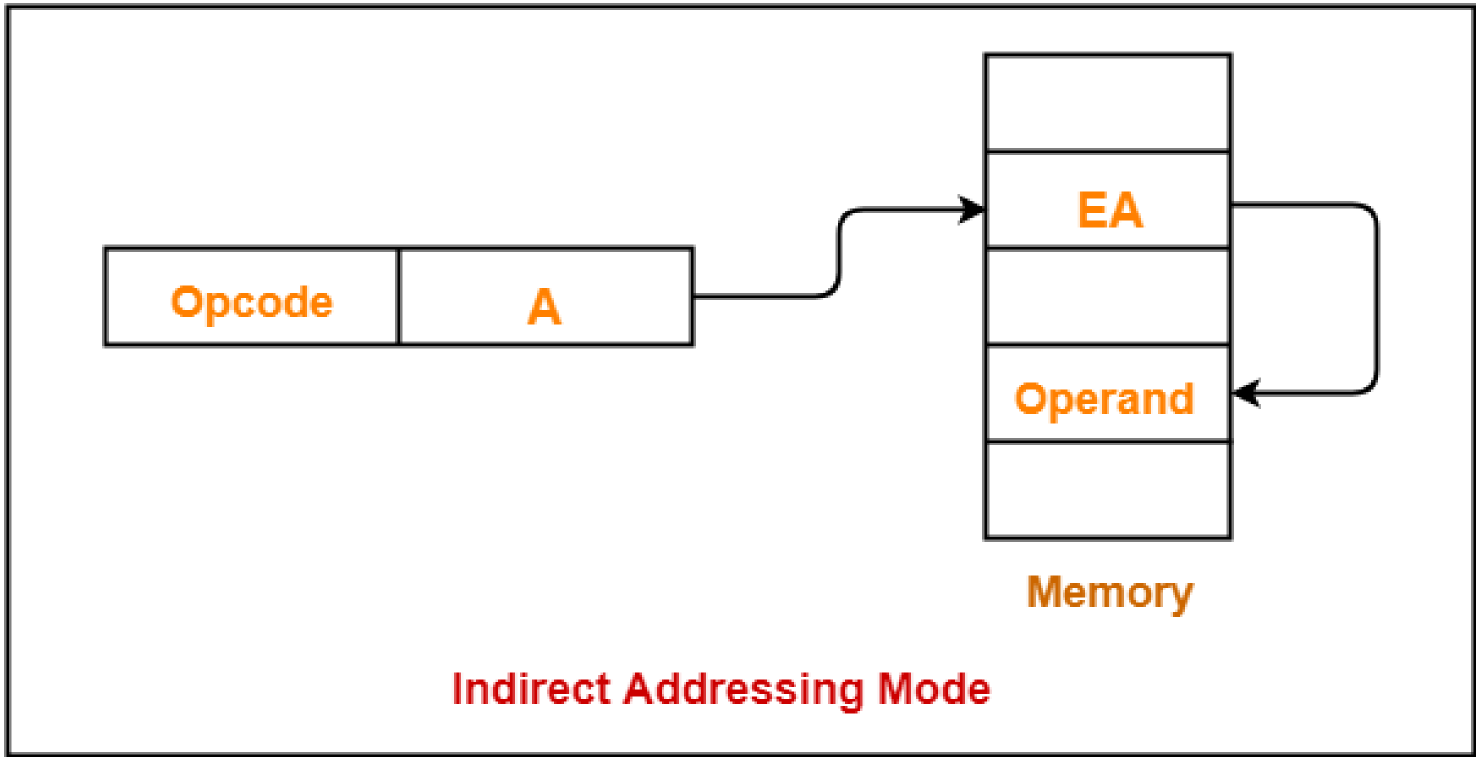


Direct Addressing Mode

Indirect Addressing (1)

- Memory cell pointed to by address field contains the address of (pointer to) the operand

  Two references to memory are required to fetch the operand.
- Effective Address = [A]
  - Look in A, find address (A) and look there for operand
- e.g. ADD (A)
  - Add contents of cell pointed to by contents of A to the accumulator


Opcode
A
EA
Operand
Memory
Indirect Addressing Mode

Register Direct Addressing

In this addressing mode,

- The operand is contained in a register set.
- The address field of the instruction refers to a CPU register that contains the operand.

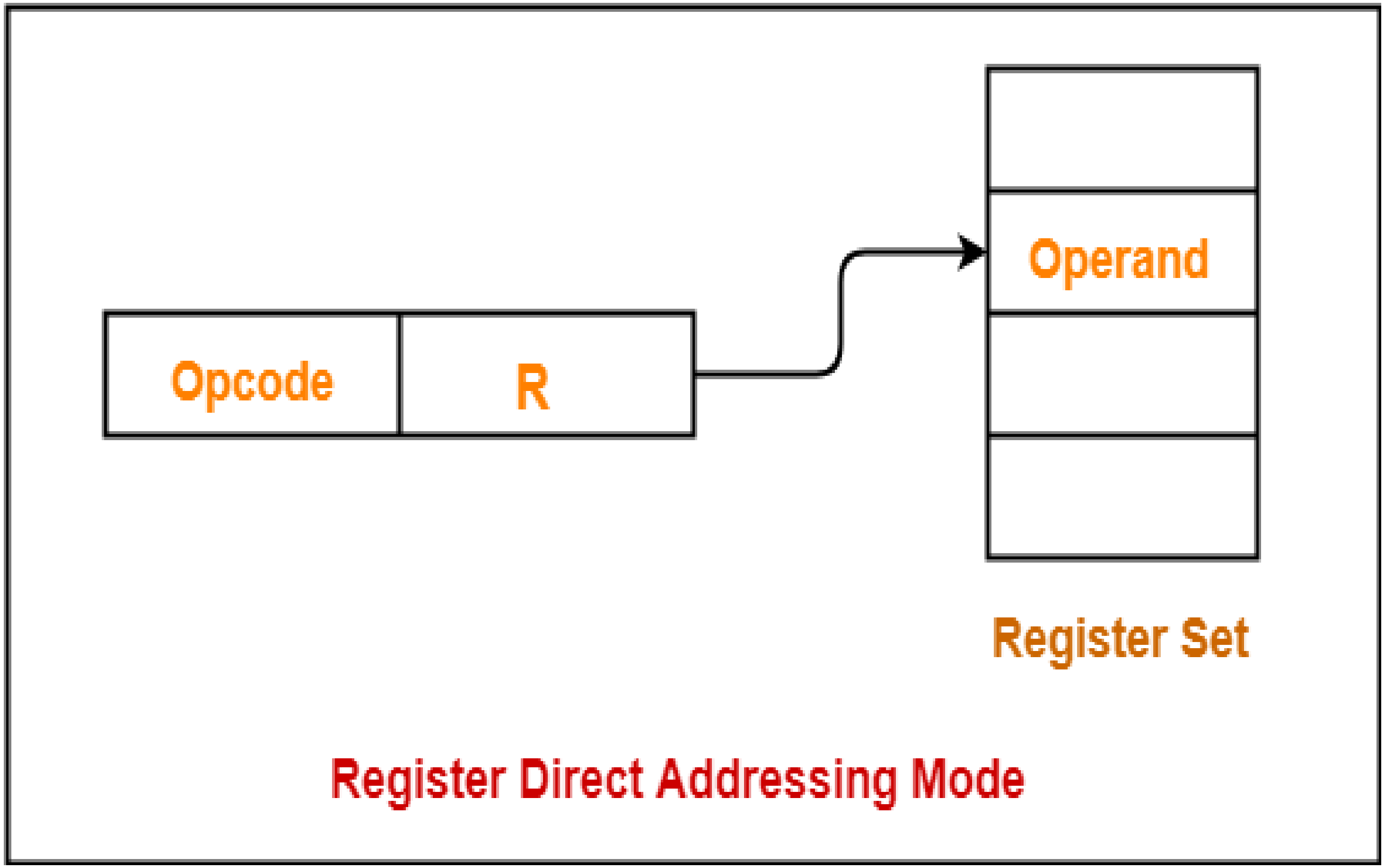
- No memory access
- Very fast execution
- Very limited address space
- Limited number of registers
- Very small address field needed
  - Shorter instructions
  - Faster instruction fetch

Eg:

ADD R will increment the value stored in the accumulator by the content of register R.

$$AC \leftarrow AC + [R]$$

- This addressing mode is similar to direct addressing mode.
- The only difference is address field of the instruction refers to a CPU register instead of main memory.



Register Direct Addressing Mode

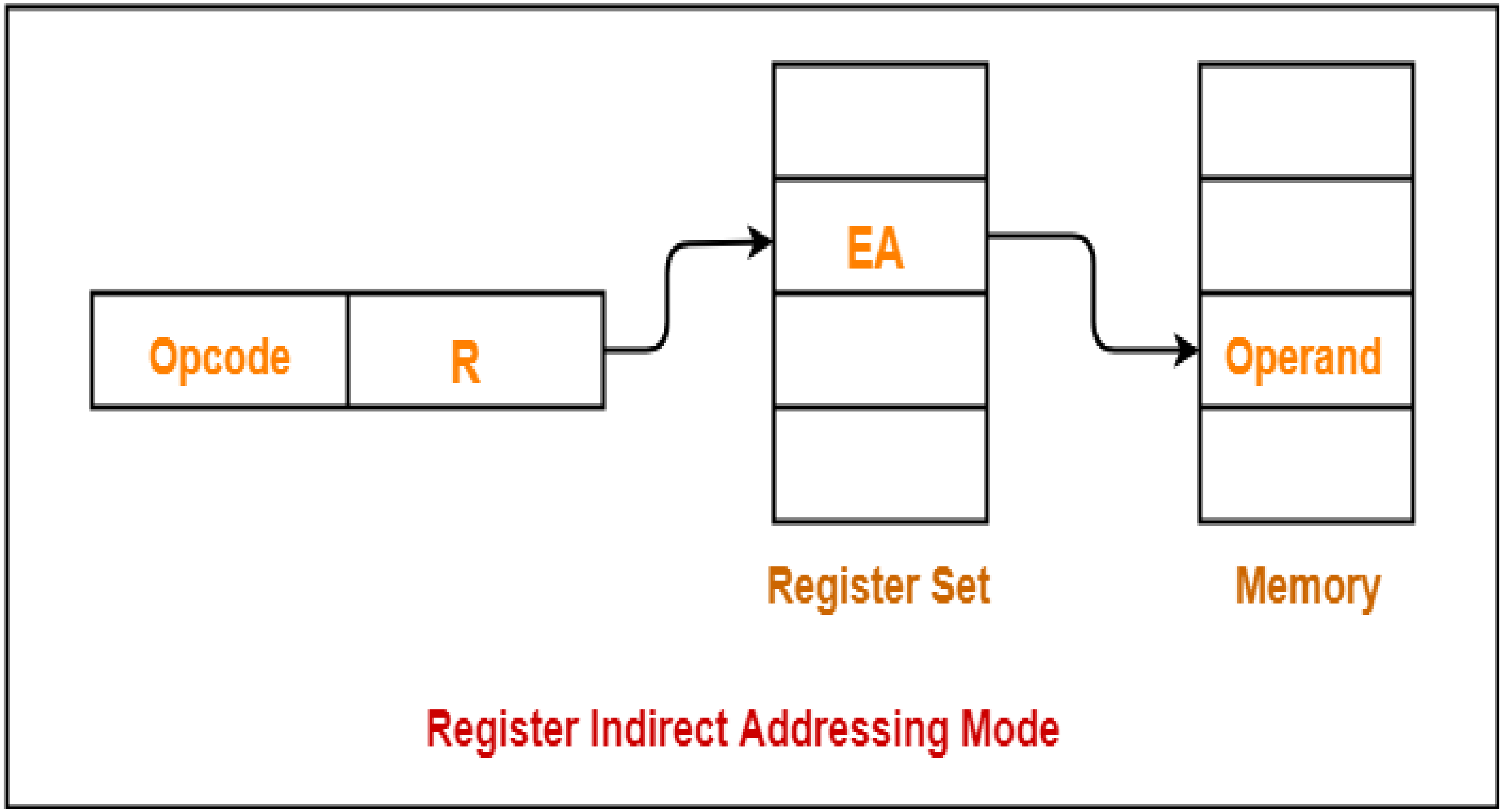
Register Indirect Addressing

- The address field of the instruction refers to a CPU register that contains the effective address of the operand.
- Only one reference to memory is required to fetch the operand

Eg:

ADD R will increment the value stored in the accumulator by the content of memory location specified in register R.

$$AC \leftarrow AC + [[R]]$$



Register Indirect Addressing Mode

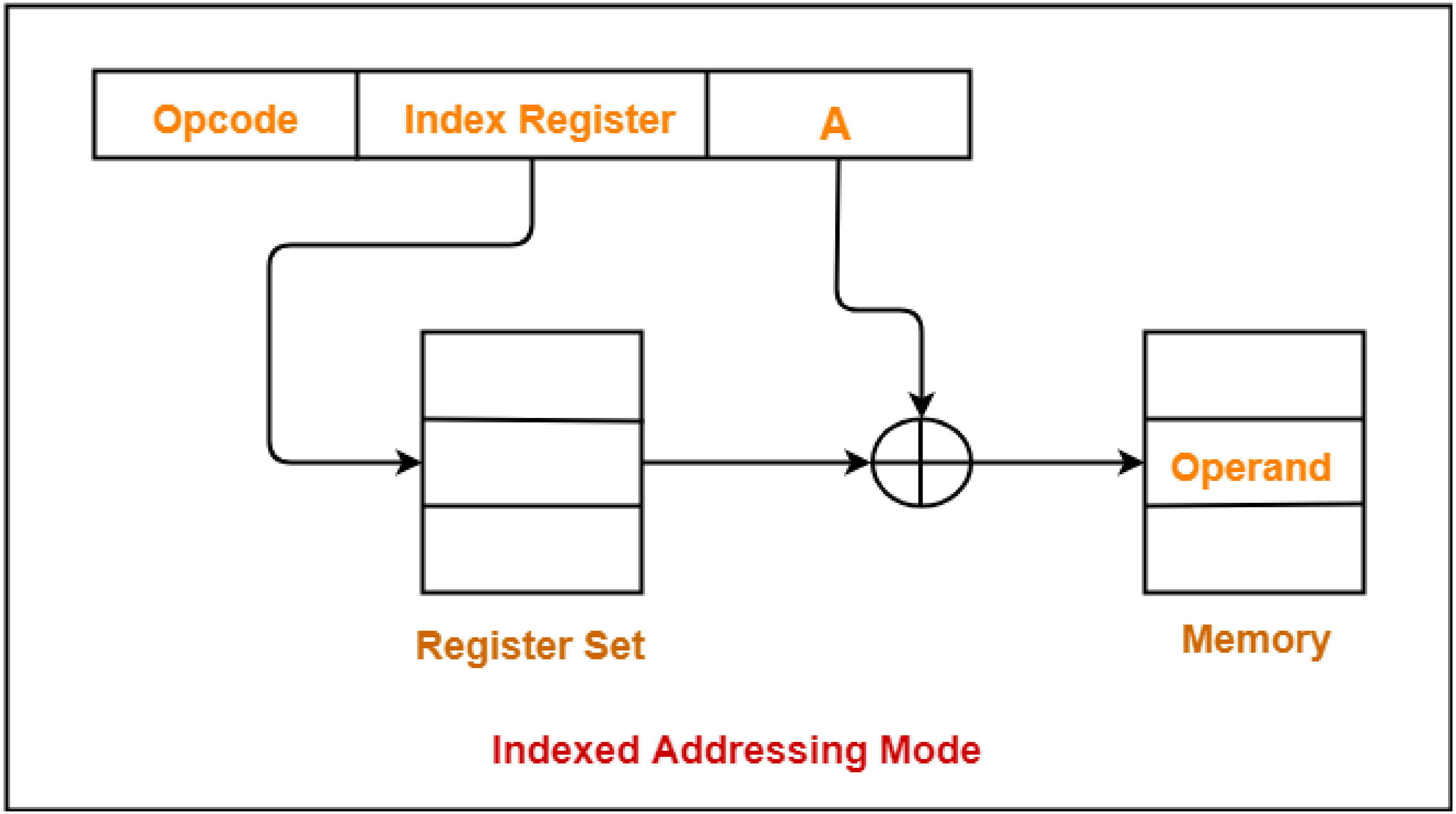
Indexed Addressing

In this addressing mode,

- Effective address of the operand is obtained by adding the content of index register with the address part of the instruction.

Effective Address

= Content of Index Register +
Address part of the instruction



Indexed Addressing Mode

Relative Addressing

A version of displacement addressing

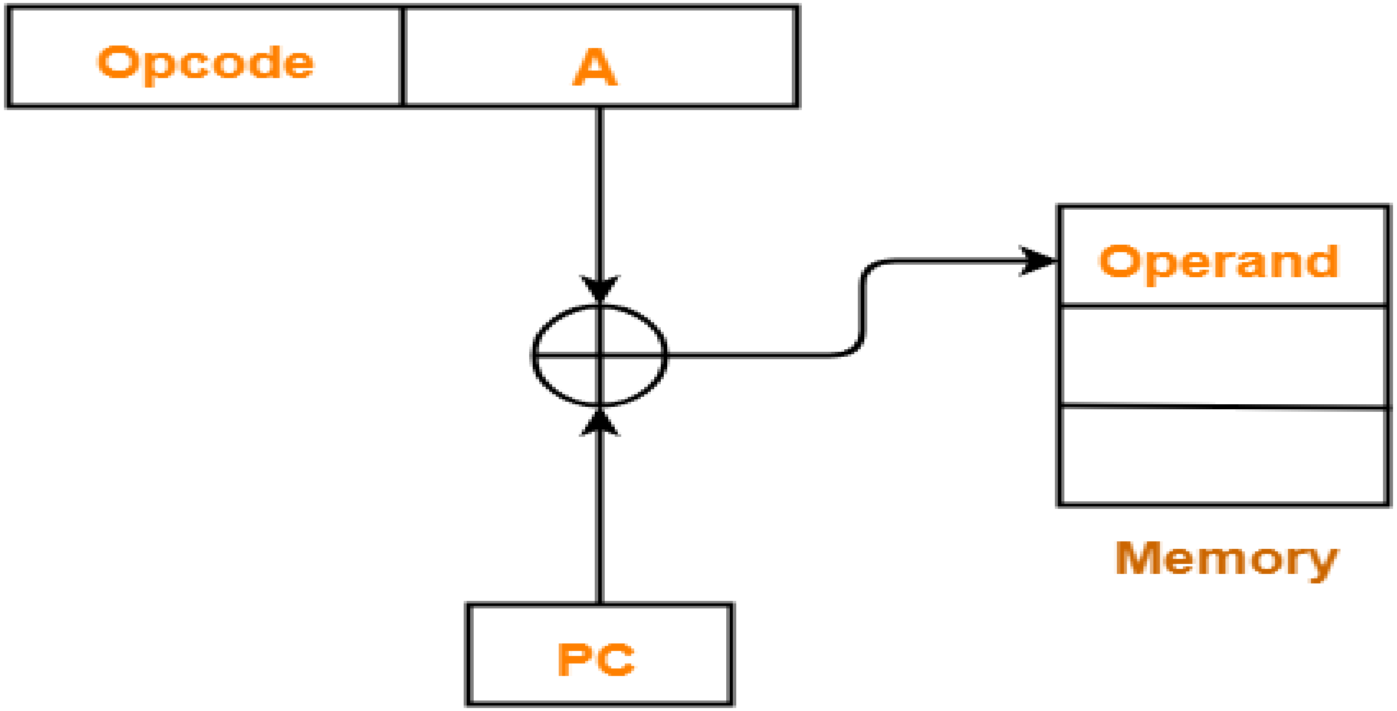
In this addressing mode,

- Effective address of the operand is obtained by adding the content of program counter with the address part of the instruction.

Effective Address

= Content of Program Counter +

Address part of the instruction



Relative Addressing Mode

Auto increment mode

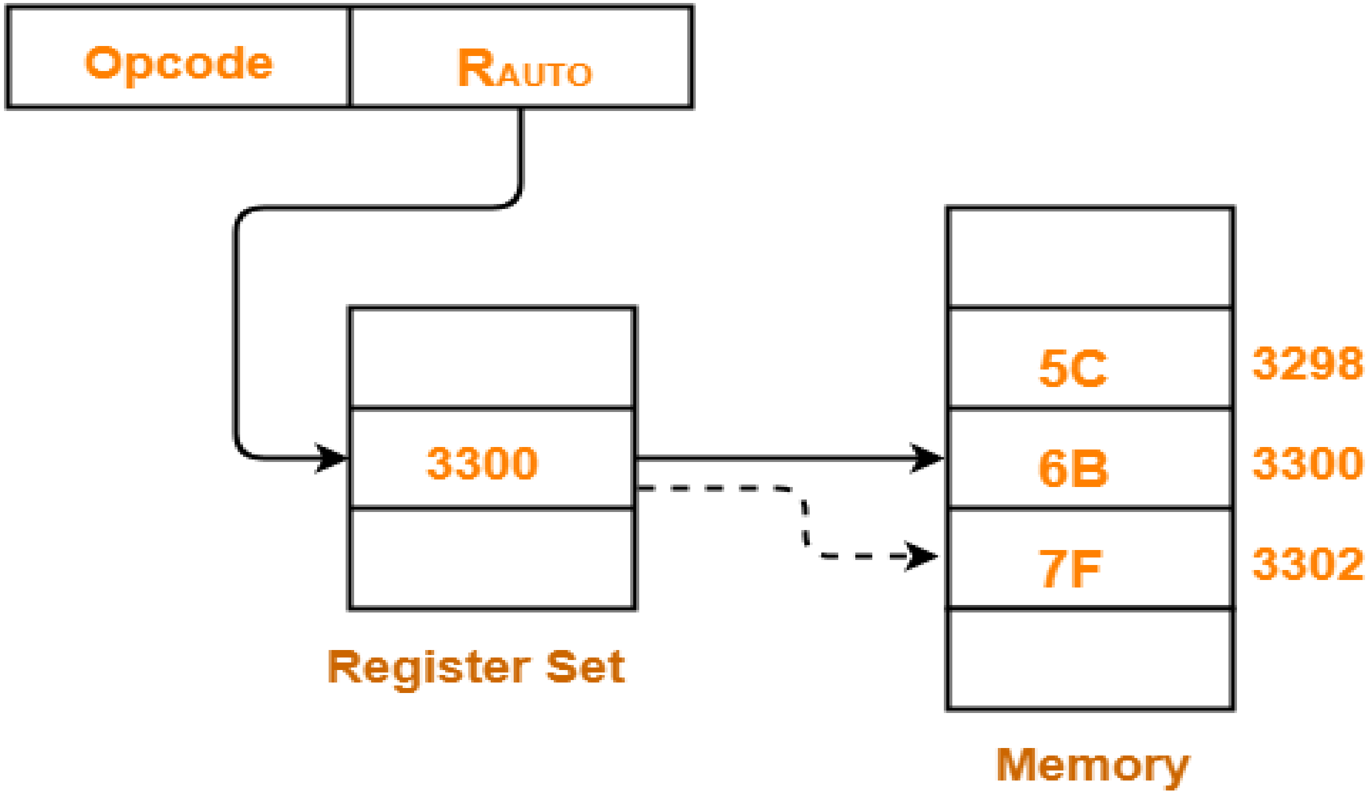
A special case of Register Indirect Addressing Mode where

Effective Address of the Operand

= Content of Register

In this addressing mode,

- After accessing the operand, the content of the register is automatically incremented by step size ‘d’.
- Step size ‘d’ depends on the size of operand accessed.
- Only one reference to memory is required to fetch the operand.



Auto-Increment Addressing Mode

Auto decrement mode

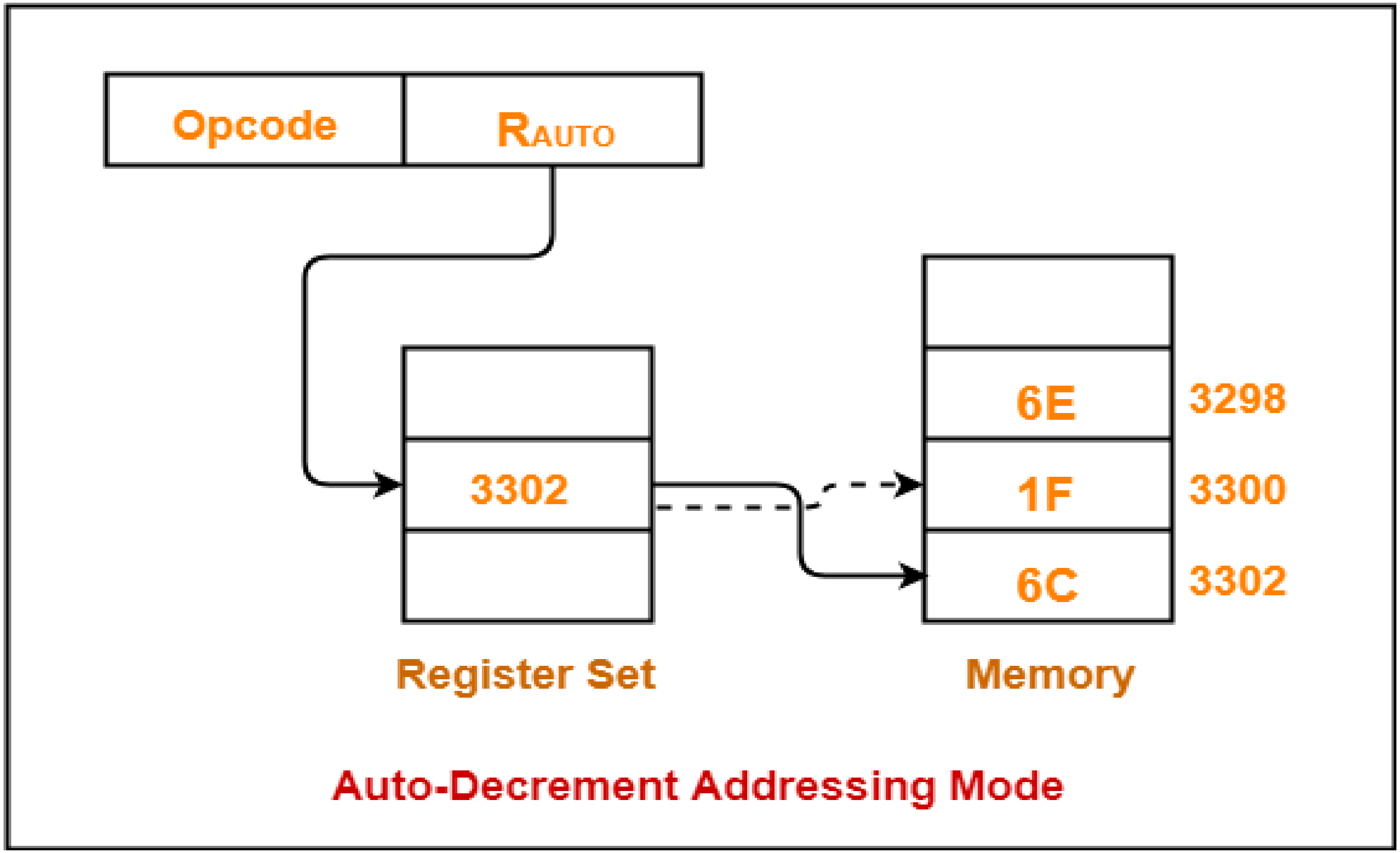
- A special case of Register Indirect Addressing Mode where

  Effective Address of the Operand

  = Content of Register – Step Size

In this addressing mode

- First, the content of the register is decremented by step size ‘d’.
- Step size ‘d’ depends on the size of operand accessed.
- After decrementing, the operand is read.
- Only one reference to memory is required to fetch the operand.



Auto-Decrement Addressing Mode

Microprocessors

Microprocessor : is a CPU on a single chip.

Microcontroller: If a microprocessor,

its associated support circuitry,

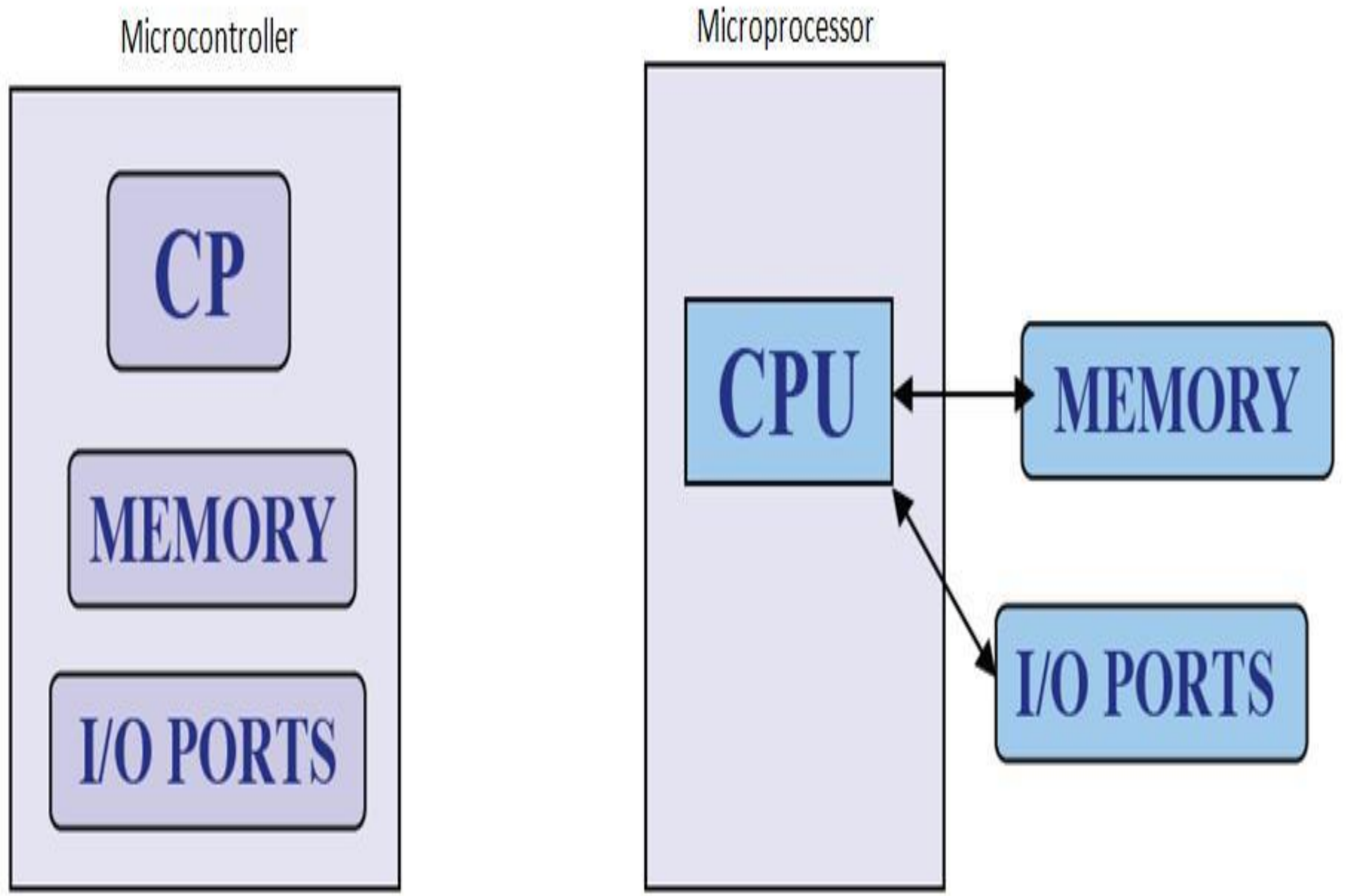
memory and

peripheral I/O components

are implemented on a single chip, it is a *microcontroller*.

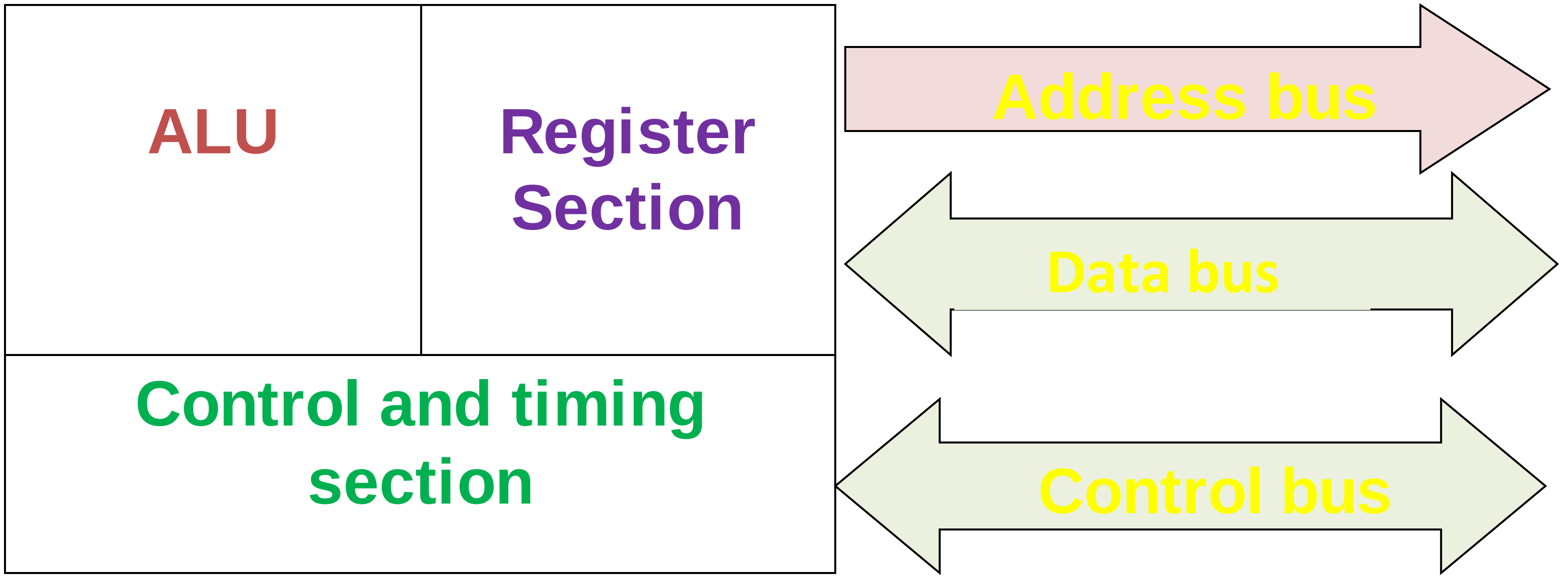
- We use AVR microcontroller as the example in our course study

What is Microprocessor and Microcontroller?



Sl No	Microprocessor	Microcontroller
1	CPU is stand-alone, RAM, ROM, I/O, timer are separate	CP,RAM,ROM,I/O and timer are all on single chip
2	Designer can decide on the amount of ROM,RAM and I/O ports	Fix amount of on-chip ROM, RAM, I/O ports
3	Expansive	Not Expansive
4	General purpose	Single purpose
5	Microprocessor based system design is complex and expensive	Microcontroller based system design is rather simple and cost effective
6	The instruction set of microprocessor is complex with large number of instructions.	The instruction set of a Microcontroller is very simple with the less number of instructions.

Internal structure and basic operation of microprocessor



Block diagram of a Microprocessor

- Microprocessor performs three main tasks:
  - data transfer between itself and the memory or I/O systems
  - simple arithmetic and logic operations
  - program flow via simple decisions

Microprocessor types

- Microprocessors can be characterized based on
  - the word size
    - 8 bit, 16 bit, 32 bit, etc. processors
  - Instruction set structure
    - RISC (Reduced Instruction Set Computer), CISC (Complex Instruction Set Computer)
  - Functions
    - General purpose, special purpose such image processing, floating point calculations
  - And more …

Evolution of Microprocessors

- The first **microprocessor** was **introduced** in 1971 by Intel Corp.
- It was named Intel 4004 as it was a 4 bit processor.

Categories according to the generations or size

First Generation (4 - bit Microprocessors)

- could perform simple arithmetic such as addition, subtraction, and logical operations like Boolean OR and Boolean AND.
- had a control unit capable of performing control functions like
  - fetching an instruction from storage memory,
  - decoding it, and then
  - generating control pulses to execute it.

Second Generation (8 - bit Microprocessor)

- The second generation microprocessors were introduced in 1973 again by Intel.
- the first 8 - bit microprocessor which could perform arithmetic and logic operations on 8-bit words.

Third Generation (16 - bit Microprocessor)

- introduced in 1978
- represented by **Intel's 8086, Zilog Z800 and 80286**,
- 16 - bit processors with a performance like minicomputers.

Fourth Generation (32 - bit Microprocessors)

- Several different companies introduced the 32-bit microprocessors
- the most popular one is the **Intel 80386**

Fifth Generation (64 - bit Microprocessors)

- Introduced in 1995
- After 80856, Intel came out with a new processor namely Pentium processor followed by **Pentium Pro CPU**
- allows multiple CPUs in a single system to achieve multiprocessing.
- Other improved 64-bit processors are **Celeron, Dual, Quad, Octa Core processors**.

Typical microprocessors

- Most commonly used
  - 68K
    - Motorola
  - x86
    - Intel
  - IA-64
    - Intel
  - MIPS
    - Microprocessor without interlocked pipeline stages
  - ARM
    - Advanced RISC Machine
  - PowerPC
    - Apple-IBM-Motorola alliance
  - Atmel AVR
- A brief summary will be given later

8086 Microprocessor

- designed by Intel in 1976
- 16-bit Microprocessor having
- 20 address lines
- 16 data lines
- provides up to 1MB storage
- consists of powerful instruction set, which provides operations like multiplication and division easily.

supports two modes of operation

Maximum mode :

suitable for system having multiple processors

Minimum mode :

suitable for system having a single processor.

Features of 8086

- Has an instruction queue, which is capable of storing six instruction bytes
- First 16-bit processor having
  - 16-bit ALU
  - 16-bit registers
  - internal data bus
  - 16-bit external data bus

uses two stages of pipelining

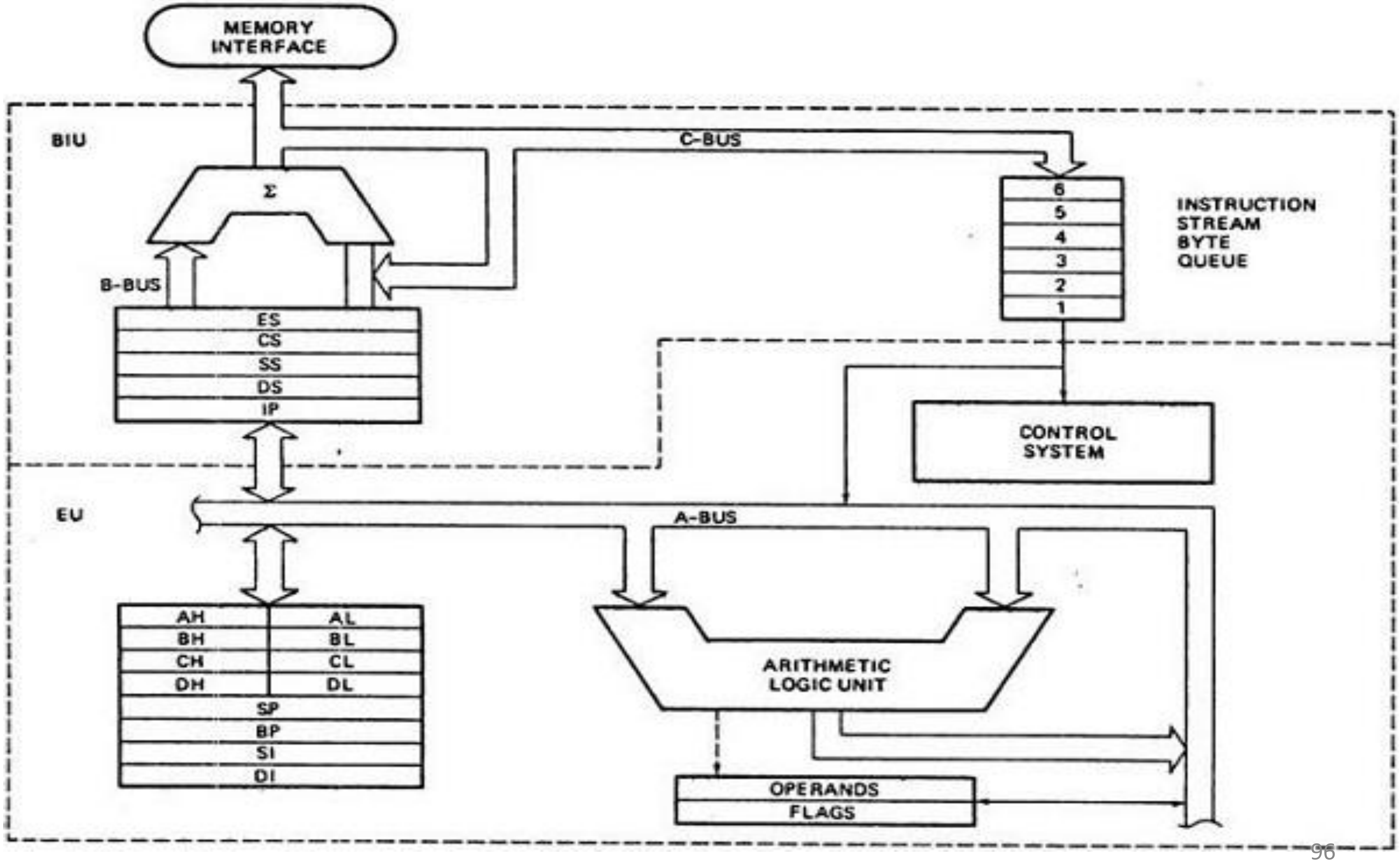
1. Fetch Stage and
2. Execute Stage

which improves performance.

Fetch stage : can pre-fetch up to 6 bytes of instructions and stores them in the queue.

Execute stage : executes these instructions.

Architecture of 8086



Segments in 8086

memory is divided into various sections called segments

Code segment : where you store the program.

Data segment : where the data is stored.

Extra segment : mostly used for string operations.

Stack segment : used to push/pop

General purpose registers

used to store temporary data within the microprocessor

AX – Accumulator

16 bit register

divided into two 8-bit registers AH and AL to perform 8-bit instructions also

generally used for arithmetical and logical instructions

BX – Base register

16 bit register

divided into two 8-bit registers BH and BL to perform 8-bit instructions also

Used to store the value of the offset.

CX – Counter register

16 bit register

divided into two 8-bit registers CH and CL to perform 8-bit instructions also

Used in looping and rotation

DX – Data register

16 bit register

divided into two 8-bit registers DH and DL to perform 8-bit instructions also

Used in multiplication an input/output port addressing

Pointers and Index Registers

SP – Stack pointer

16 bit register

points to the topmost item of the stack

If the stack is empty the stack pointer will be (FFFE)H

It’s offset address relative to stack segment

BP –Base pointer

16 bit register

used in accessing parameters passed by the stack

It’s offset address relative to stack segment

SI – Source index register

16 bit register

used in the pointer addressing of data and

as a source in some string related operations

It's offset is relative to data segment

DI – Destination index register

16 bit register

used in the pointer addressing of data and

as a destination in string related operations

It's offset is relative to extra segment.

IP - Instruction Pointer

16 bit register

stores the address of the next instruction to be executed

also acts as an offset for CS register.

Segment Registers

CS - Code Segment Register:

user cannot modify the content of these registers

Only the microprocessor's compiler can do this

DS - Data Segment Register:

The user can modify the content of the data segment.

SS - Stack Segment Registers:

used to store the information about the memory segment.

operations of the SS are mainly Push and Pop.

ES - Extra Segment Register:

By default, the control of the compiler remains in the DS where the user can add and modify the instructions

If there is less space in that segment, then ES is used

Also used for copying purpose.

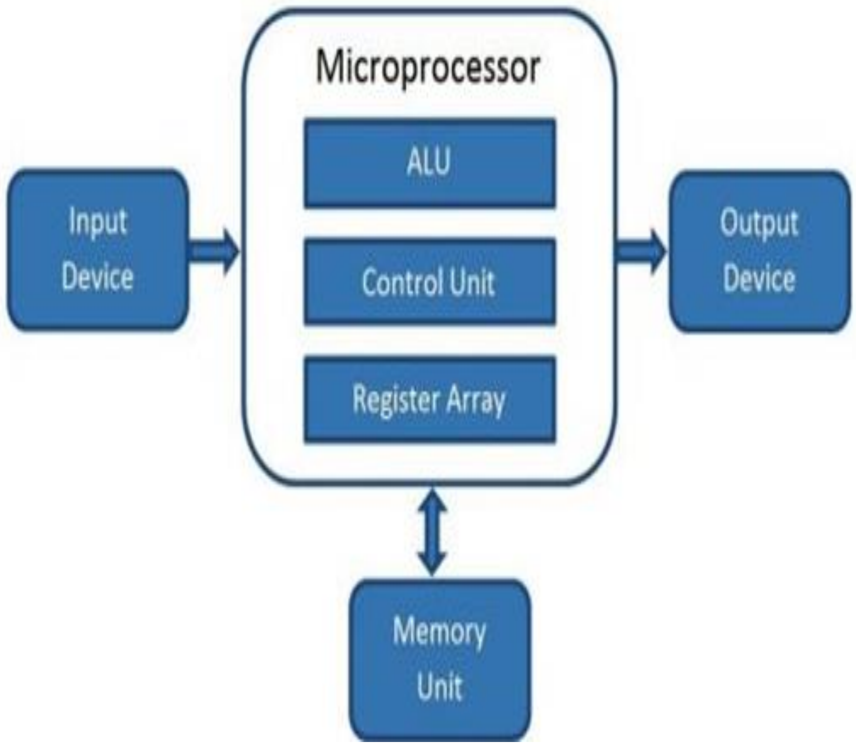
Flag or Status Register

- 16-bit register
- contains 9 flags
- remaining 7 bits are idle in this register
- These flags tell about the status of the processor after any arithmetic or logical operation
- IF the flag value is 1, the flag is set, and if it is 0, it is said to be reset.

Microcomputer

Block Diagram

- A digital computer with one microprocessor which acts as a CPU
- A complete computer on a small scale, designed for use by one person at a time
- called a personal computer (PC)
- a device based on a single-chip microprocessor
- includes laptops and desktops



Introduction to 8086 Assembly Language

Assembly Language Programming

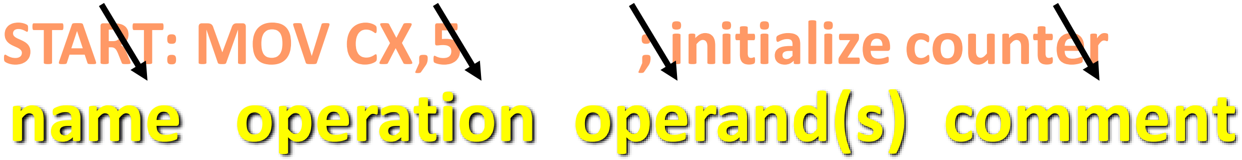
Program Statements

- Program consist of statement, one per line.
- Each statement is either an instruction, which the assembler translate into machine code, or assembler directive, which instructs the assembler to perform some specific task, such as allocating memory space for a variable or creating a procedure.
- Both instructions and directives have up to four fields:
- At least one blank or tab character must separate the fields

name operation operand(s) comment

Program Statements

- An example of an instruction is

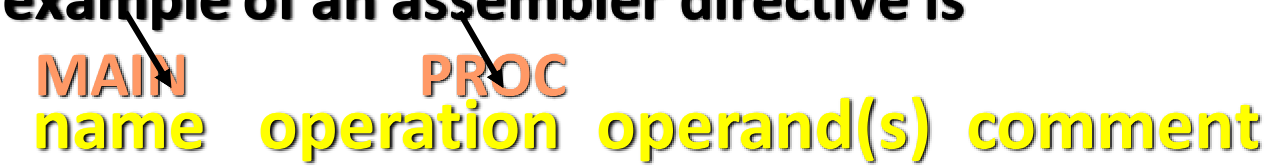


START: MOV CX,5 ;initialize counter

name operation operand(s) comment

  - The name field consists of the label START:
  - The operation is MOV, the operands are CX and 5
  - And the comment is ; initialize counter

- An example of an assembler directive is



MAIN PROC

name operation operand(s) comment

  - MAIN is the name, and the operation field contains PROC
  - This particular directive creates a procedure called MAIN

Program Statements

name operation operand(s) comment

- A Name field identifies a label, variable, or symbol.
  It may contain any of the following character :
  A,B…..Z ; a,b….z ; 0,1….9 ; ? ;
  _ (underline) ; @ ; $; . (period)

- Only the first 31 characters are recognized
- There is no distinction between uppercase and lower case letters.
- The first character may not be a digit
- If it is used, the period (.) may be used only as the first character.
- A programmer-chosen name may not be the same as an assembler reserved word.

Program Statements

name operation operand(s) comment

- **Operation field is a predefined or reserved word**
  - **mnemonic - symbolic operation code.**
    - **The assembler translates a symbolic opcode into a machine language opcode.**
    - **Opcode symbols often discribe the operation's function; for example, MOV, ADD, SUB**
  - **assemler directive - pseudo-operation code.**
    - **In an assembler directive, the operation field contains a pseudo-operation code (pseudo-op)**
    - **Pseudo-op are not translated into machine code; for example the PROC pseudo-op is used to create a procedure**

Program Statements

name   operation operand(s) comment

- An operand field specifies the data that are to be acted on by the operation.
- An instruction may have zero, one, or two operands. For example:
  - NOP — No operands; does nothing
  - INC AX — one operand; adds 1 to the contents of AX
  - ADD WORD1,2 — two operands; adds 2 to the contents of memory word WORD1

Program Statements

name operation operand(s) comment

- **The comment field is used by the programmer to say something about what the statement does.**
- **A semicolon marks the beginning of this field, and the assembler ignores anything typed after semicolon.**
- **Comments are optional, but because assembly language is low level, it is almost impossible to understand an assembly language program without comments.**

Program Data and Storage

- Pseudo-ops to define data or reserve storage
  - DB - byte(s)
  - DW - word(s)
  - DD - doubleword(s)
  - DQ - quadword(s)
  - DT - tenbyte(s)

- These directives require one or more operands
  - define memory contents
  - specify amount of storage to reserve for run-time data

Defining Data

- Numeric data values
  - 100 - decimal
  - 100B - binary
  - 100H - hexadecimal
  - '100' - ASCII
  - "100" - ASCII
- Use the appropriate DEFINE directive (byte, word, etc.)
- A list of values may be used - the following creates 4 consecutive words

```
DW 40CH,10B,-13,0
```

- A ? represents an uninitialized storage location

```
DB 255,?,-128,'X'
```

Naming Storage Locations

- Names can be associated with storage locations

```
ANum DB -4
  DW 17
ONE
UNO DW 1
X DD ?
```

- These names are called variables

- ANum refers to a byte storage location, initialized to FCh
- The next word has no associated name
- ONE and UNO refer to the same word
- X is an unitialized doubleword

- Multiple definitions can be abbreviated

  Example:

```
message     DB ’B’
            DB ’y’
            DB ’e’
            DB 0DH
            DB 0AH
```

  can be written as

```
message     DB ’B’,’y’,’e’,0DH,0AH
```

- More compactly as

```
message DB ’Bye’,0DH,0AH
```

Arrays

- Any consecutive storage locations of the same size can be called an array

```
X DW 40CH,10B,-13,0
Y DB 'This is an array'
Z DD -109236, FFFFFFFFH, -1, 100B
```

- Components of X are at X, X+2, X+4, X+6
- Components of Y are at Y, Y+1, …, Y+15
- Components of Z are at Z, Z+4, Z+8, Z+12

DUP

- Allows a sequence of storage locations to be defined or reserved
- Only used as an operand of a define directive

```
DB   40  DUP (?)   ; 40 words, uninitialized
DW  10h  DUP (0)   ; 16 words, initialized as 0
Table1  DW  10  DUP (?) ; 10 words, uninitialized

message  DB  3  DUP (’Baby’) ; 12 bytes, initialized
                             ; as BabyBabyBaby
Name1  DB  30  DUP (’?’) ; 30 bytes, each
                         ; initialized to ?
```

DUP

- **The DUP directive may also be nested**

Example

stars DB 4 DUP(3 DUP (’*’),2 DUP (’?’),5 DUP (’!’))

Reserves 40-bytes space and initializes it as

??!!!!!??!!!!!***??!!!!!***??!!!!!

matrix DW 10 DUP (5 DUP (0))

defines a 10X5 matrix and initializes its elements to zero.

This declaration can also be done by

matrix DW 50 DUP (0)

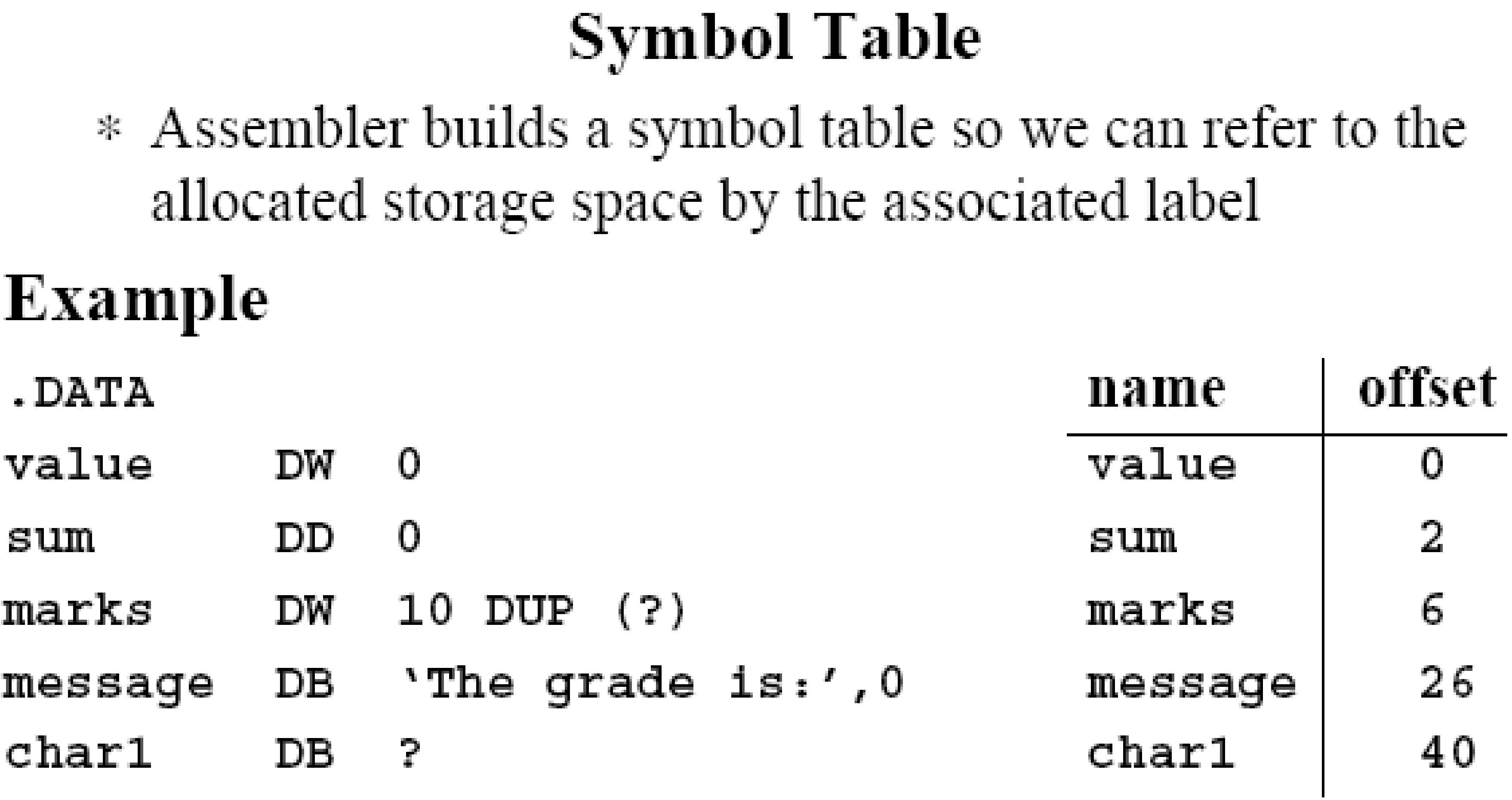
Word Storage

- Word, doubleword, and quadword data are stored in reverse byte order (in memory)

Directive	Bytes in Storage
DW 256	00 01
DD 1234567H	67 45 23 01
DQ 10	0A 00 00 00 00 00 00 00
X DW 35DAh	DA 35

Low byte of X is at X, high byte of X is at X+1

Word Storage



Symbol Table

- Assembler builds a symbol table so we can refer to the allocated storage space by the associated label

Example

```
.DATA
value     DW   0
sum       DD   0
marks     DW   10 DUP (?)
message   DB   'The grade is:',0
char1     DB   ?
```

name	offset
value	0
sum	2
marks	6
message	26
char1	40

Named Constants

- Symbolic names associated with storage locations represent addresses
- Named constants are symbols created to represent specific values determined by an expression
- Named constants can be numeric or string
- Some named constants can be redefined
- No storage is allocated for these values

Equal Sign Directive

- name = expression
  - expression must be numeric
  - these symbols may be redefined at any time

```
maxint = 7FFFh
count = 1
DW count
count = count * 2
DW count
```

EQU Directive

- name EQU expression
  - expression can be string or numeric
  - Use < and > to specify a string EQU
  - these symbols cannot be redefined later in the program

```
sample EQU 7Fh
aString EQU <1.234>
message EQU <This is a message>
```

Data Transfer Instructions

- **MOV *target, source***
  - **reg, reg**
  - **mem, reg**
  - **reg, mem**
  - **mem, immed**
  - **reg, immed**
- Sizes of both operands must be the same

- reg can be any non-segment register except IP cannot be the target register
- MOV's between a segment register and memory or a 16-bit register are possible

Sample MOV Instructions

```
b db 4Fh
w dw 2048

mov bl,dh
mov ax,w
mov ch,b
mov al,255
mov w,-100
mov b,0
```

- When a variable is created with a define directive, it is assigned a default size attribute (byte, word, etc)
- You can assign a size attribute using LABEL

```
LoByte LABEL BYTE
aWord DW 97F2h
```

Program Segment Structure

- Data Segments
  - Storage for variables
  - Variable addresses are computed as offsets from start of this segment
- Code Segment
  - contains executable instructions
- Stack Segment
  - used to set aside storage for the stack
  - Stack addresses are computed as offsets into this segment
- Segment directives

```
.data
.code
.stack size
```

Instruction types

Data transfer instructions

8086 instruction set

IN	Input byte or word from port
LAHF	Load AH from flags
LDS	Load pointer using data segment
LEA	Load effective address
LES	Load pointer using extra segment
MOV	Move to/from register/memory
OUT	Output byte or word to port
POP	Pop word off stack
POPF	Pop flags off stack
PUSH	Push word onto stack
PUSHF	Push flags onto stack
SAHF	Store AH into flags
XCHG	Exchange byte or word
XLAT	Translate byte

Additional 80286 instructions

INS	Input string from port
OUTS	Output string to port
POPA	Pop all registers
PUSHA	Push all registers

Additional 80386 instructions

LFS	Load pointer using FS
LGS	Load pointer using GS
LSS	Load pointer using SS
MOVSX	Move with sign extended
MOVZX	Move with zero extended
POPAD	Pop all double (32 bit) registers
POPD	Pop double register
POPFD	Pop double flag register
PUSHAD	Push all double registers
PUSHD	Push double register
PUSHFD	Push double flag register

Additional 80486 instruction

BSWAP	Byte swap

Additional Pentium instruction

MOV	Move to/from control register

Arithmetic instructions

8086 instruction set

AAA	ASCII adjust for addition
AAD	ASCII adjust for division
AAM	ASCII adjust for multiply
AAS	ASCII adjust for subtraction
ADC	Add byte or word plus carry
ADD	Add byte or word
CBW	Convert byte or word
CMP	Compare byte or word
CWD	Convert word to double-word
DAA	Decimal adjust for addition
DAS	Decimal adjust for subtraction
DEC	Decrement byte or word by one
DIV	Divide byte or word
IDIV	Integer divide byte or word
IMUL	Integer multiply byte or word
INC	Increment byte or word by one
MUL	Multiply byte or word (unsigned)
NEG	Negate byte or word
SBB	Subtract byte or word and carry (borrow)
SUB	Subtract byte or word

Additional 80386 instructions

CDQ	Convert double-word to quad-word
CWDE	Convert word to double-word

Additional 80486 instructions

CMPXCHG	Compare and exchange
XADD	Exchange and add

Additional Pentium instruction

CMPXCHG8B	Compare and exchange 8 bytes

Bit manipulation instructions

8086 instruction set

AND	Logical AND of byte or word
NOT	Logical NOT of byte or word
OR	Logical OR of byte or word
RCL	Rotate left trough carry byte or word
RCR	Rotate right trough carry byte or word
ROL	Rotate left byte or word
ROR	Rotate right byte or word
SAL	Arithmetic shift left byte or word
SAR	Arithmetic shift right byte or word
SHL	Logical shift left byte or word
SHR	Logical shift right byte or word
TEST	Test byte or word
XOR	Logical exclusive-OR of byte or word

Additional 80386 instructions

BSF	Bit scan forward
BSR	Bit scan reverse
BT	Bit test
BTC	Bit test and complement
BTR	Bit test and reset
BTS	Bit test and set
SETcc	Set byte on condition
SHLD	Shift left double precision
SHRD	Shift right double precision

String instructions

8086 instruction set

CMPS	Compare byte or word string
LODS	Load byte or word string
MOVS	Move byte or word string
MOVSB(MOVSW)	Move byte string (word string)
REP	Repeat
REPE (REPZ)	Repeat while equal (zero)
REPNE (REPNZ)	Repeat while not equal (not zero)
SCAS	Scan byte or word string
STOS	Store byte or word string

Program Skeleton

```
.model small
.stack 100H
.data
   ;declarations
.code
main proc
   ;code
main endp
   ;other procs
end main
```

- Select a memory model
- Define the stack size
- Declare variables
- Write code
  - organize into procedures
- Mark the end of the source file
  - optionally, define the entry point

EXAMPLE : Adding two 8 bit numbers

```
DATA SEGMENT                    ; Data Segment
N1
3n2 DB 12H
N2 DB 21H
RES DB ?
DATA ENDS
CODE SEGMENT                    ; Code segment
ASSUME CS: CODE, DS: DATA
START: MOV AX, DATA
MOV DS, AX
MOV AL, N1
MOV BL, N2
ADD AL, BL
MOV RES, AL
INT 21H
CODE ENDS
END START
```

The ARM Architecture

ARM Ltd

- Founded in November 1990
- Designs the ARM range of RISC processor cores
- Licenses ARM core designs to semiconductor partners who fabricate and sell to their customers.
- Also develop technologies to assist with the design-in of the ARM architecture

ARM Partnership Model

ATAP Partners
WIN-FINITY
BARCO SILEX
SOTA
DNP
Infinite Technology Corporation
SIEMENS
NSW
STEPMIND
Think
MacroTech
COMIT SYSTEMS
YOGITECH
ARCADIA
SIDSA
MAZeT
TOPPAN
WIPRO
SEODU INCHIF
Z A I Q
HOYA
INI CORE
MAZ
SCIWORX
FARADAY
nordic
TALITY
parthus
SYNOPSYS
FLEXTRONICS Semiconductor
Tools Partners
EPI
ASHLING
CoWare
YOKOGAWA
virtio
Green Hills
INNOVEDA
Computex
ADS
Tektronix
WindRiver
Sophia systems
Axis systems
Verisity
Aptix
Agilent Technologies
ALCATEL
COGENT
Aliant
IKOS
SYNOPSYS
Mentor Graphics
LAUTERBACH
cadence
QUALCOMM
ZTEIC
GOODRICH
MITSUBISHI ELECTRIC
ATMEL
FUJITSU
ERICSSON
OKI
MOTOROLA
ADMtek
NEC
NEOMAGIC
TOSHIBA
intersil
SAMSUNG
AMI
MICRONAS
PHILIPS
UMC
EPSON
Agilent Technologies
Chartered
SANYO
ROHM
ANALOG DEVICES
ALCATEL
ARM in Partnership
TEXAS INSTRUMENTS
AKM
CIRRUS LOGIC
ST
Panasonic
3Com
LSI LOGIC
agere systems
BROADCOM
SHARP
ZEEVO
intel
HYUNDAI
YAMAHA
OAK
CONEXANT
IBM
Virata
SONY
cogency
PRAIRIECOMM
Triscend
ALTERA
Infineon
ZARLINK
ArrayComm
KAWASAKI MICRO
Silicon Wave
RESONEXT
National Semiconductor
FIRMWARE SYSTEMS
realogy
esol
jmi
LINEO
QNX
Accelerated Technology Inc
ACCESS
PRECISE
US Software
GEOWORKS
Tao Systems
KADAK Products Ltd.
JAVA
OSE systems
Microsoft
AXE
FirmWorks
Mentor Graphics
Eonic
symbian
WindRiver
MICROWARE
Sun microsystems
ETNOTEAM
Embedded System Products
LYNUXWORKS
CMX COMPANY
RTOS Partners
interniche technologies, inc.
Microsoft
EMBLAZE Cellular Technology by C&C
FRONTIER
PacketVideo
INTERTRUST
ERICSSON
ZI corporation
liquid audio
Bluetooth
symbian
QNX
Dolby
Symmetricom
CPS
JAVA
ASAHI CHEMICAL INDUSTRY CO.,LTD.
Software Partners

ARM Powered Products

Data Sizes and Instruction Sets

- The ARM is a 32-bit architecture.

- When used in relation to the ARM:
  - Byte means 8 bits
  - Halfword means 16 bits (two bytes)
  - Word means 32 bits (four bytes)

- Most ARM's implement two instruction sets
  - 32-bit ARM Instruction Set
  - 16-bit Thumb Instruction Set

- Jazelle cores can also execute Java bytecode

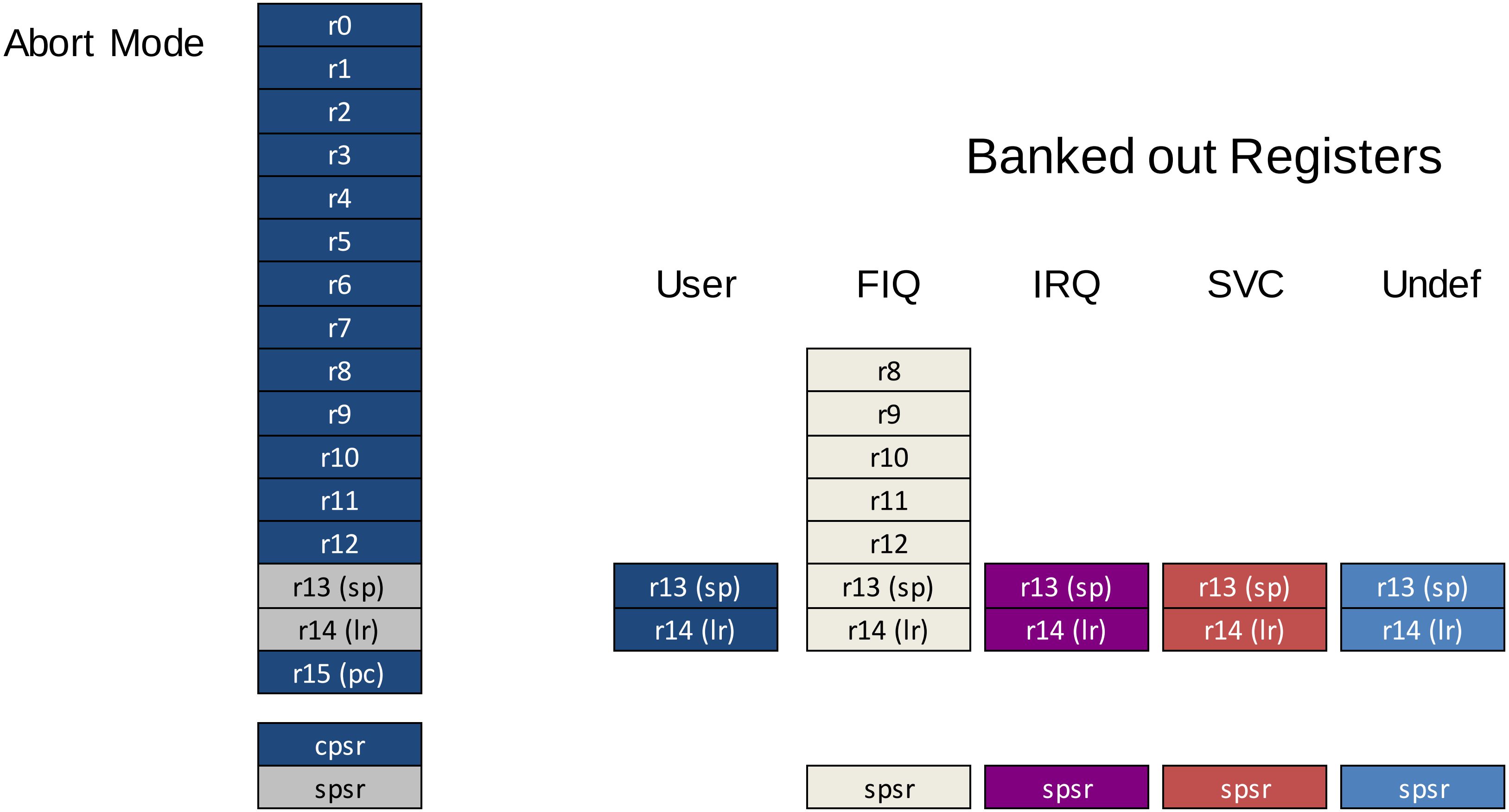
Processor Modes

- The ARM has seven basic operating modes:
  - User : unprivileged mode under which most tasks run
  - FIQ : entered when a high priority (fast) interrupt is raised
  - IRQ : entered when a low priority (normal) interrupt is raised
  - Supervisor : entered on reset and when a Software Interrupt instruction is executed
  - Abort : used to handle memory access violations
  - Undef : used to handle undefined instructions
  - System : privileged mode using the same registers as user mode

The ARM Register Set

Current Visible Registers

Banked out Registers



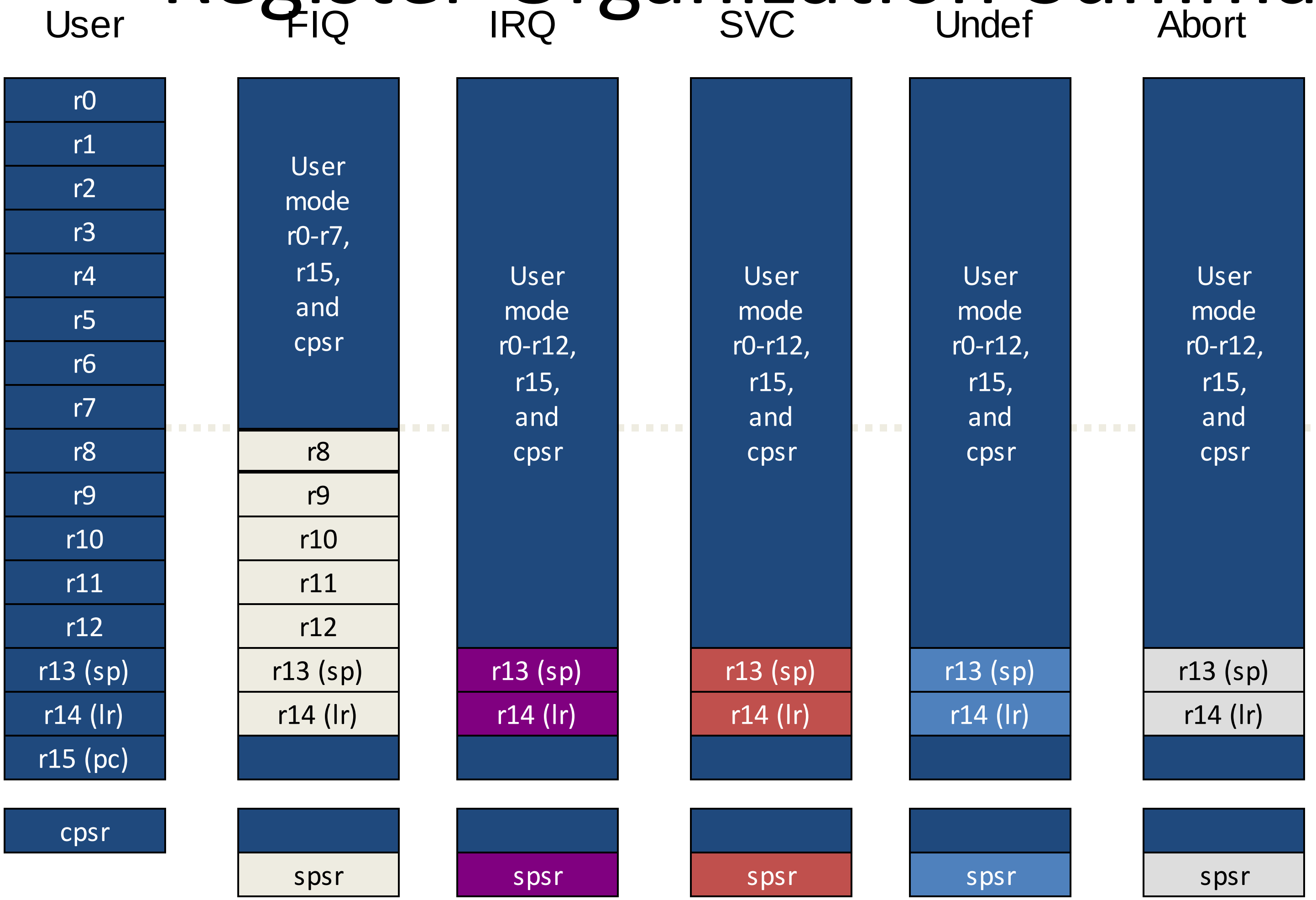
Abort Mode

r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)

cpsr
spsr

User	FIQ	IRQ	SVC	Undef
	r8			
	r9			
	r10			
	r11			
	r12			
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
	spsr	spsr	spsr	spsr

Register Organization Summary



User	FIQ	IRQ	SVC	Undef	Abort
r0	User mode r0-r7, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr	User mode r0-r12, r15, and cpsr
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8	r8				
r9	r9				
r10	r10				
r11	r11				
r12	r12				
r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)	r13 (sp)
r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)	r14 (lr)
r15 (pc)					
cpsr					
	spsr	spsr	spsr	spsr	spsr

Thumb state
Low registers

Thumb state
High registers

Note: System mode uses the User mode register set

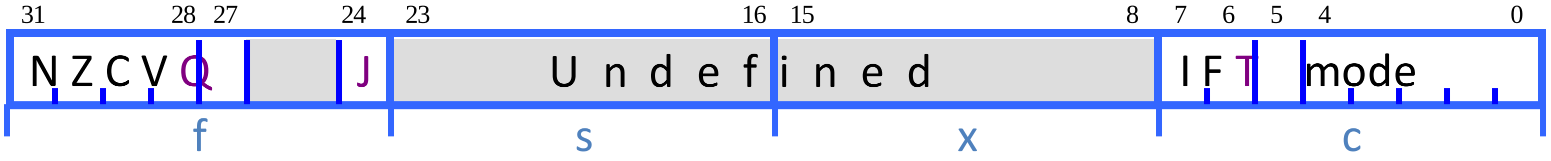
The Registers

- ARM has 37 registers all of which are 32-bits long.
  - 1 dedicated program counter
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers

- The current processor mode governs which of several banks is accessible. Each mode can access
  - a particular set of r0-r12 registers
  - a particular r13 (the stack pointer, sp) and r14 (the link register, lr)
  - the program counter, r15 (pc)
  - the current program status register, cpsr

  Privileged modes (except System) can also access
  - a particular spsr (saved program status register)

Program Status Registers


31 28 27 24 23 16 15 8 7 6 5 4 0

N Z C V Q J U n d e f i n e d I F T mode

f s x c

- Condition code flags
  - N = Negative result from ALU
  - Z = Zero result from ALU
  - C = ALU operation Carried out
  - V = ALU operation oVerflowed

- Sticky Overflow flag - Q flag
  - Architecture 5TE/J only
  - Indicates if saturation has occurred

- J bit
  - Architecture 5TEJ only
  - J = 1: Processor in Jazelle state

- Interrupt Disable bits.
  - I = 1: Disables the IRQ.
  - F = 1: Disables the FIQ.

- T Bit
  - Architecture xT only
  - T = 0: Processor in ARM state
  - T = 1: Processor in Thumb state

- Mode bits
  - Specify the processor mode

Program Counter (r15)

- When the processor is executing in ARM state:
  - All instructions are 32 bits wide
  - All instructions must be word aligned
  - Therefore the **pc** value is stored in bits [31:2] with bits [1:0] undefined (as instruction cannot be halfword or byte aligned).

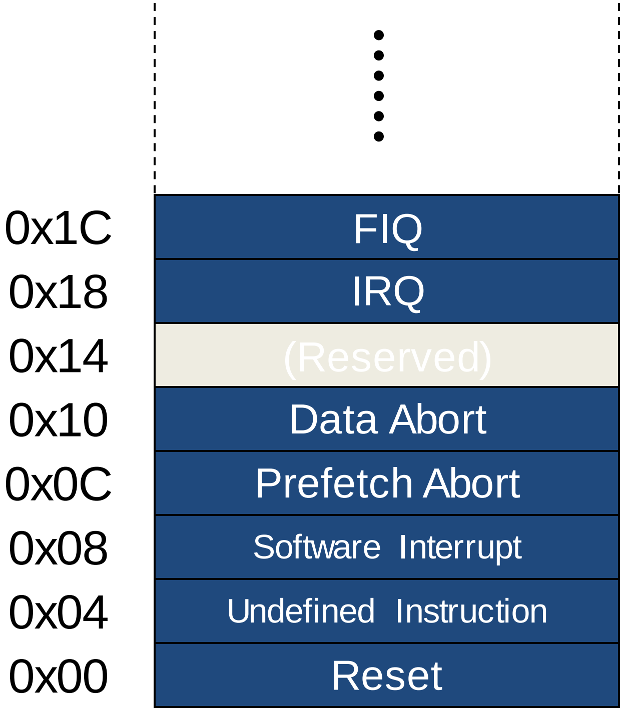
- When the processor is executing in Thumb state:
  - All instructions are 16 bits wide
  - All instructions must be halfword aligned
  - Therefore the **pc** value is stored in bits [31:1] with bit [0] undefined (as instruction cannot be byte aligned).

- When the processor is executing in Jazelle state:
  - All instructions are 8 bits wide
  - Processor performs a word access to read 4 instructions at once

Exception Handling

- When an exception occurs, the ARM:
  - Copies CPSR into SPSR_<mode>
  - Sets appropriate CPSR bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in LR_<mode>
  - Sets PC to vector address
- To return, exception handler needs to:
  - Restore CPSR from SPSR_<mode>
  - Restore PC from LR_<mode>

This can only be done in ARM state.



	⋮
0x1C	FIQ
0x18	IRQ
0x14	(Reserved)
0x10	Data Abort
0x0C	Prefetch Abort
0x08	Software Interrupt
0x04	Undefined Instruction
0x00	Reset

Vector Table

Vector table can be at 0xFFFF0000 on ARM720T and on ARM9/10 family devices

Conditional Execution and Flags

- ARM instructions can be made to execute conditionally by postfixing them with the appropriate condition code field.
  - This improves code density *and* performance by reducing the number of forward branch instructions.

```
 CMP    r3,#0
  BEQ    skip
  ADD    r0,r1,r2
skip
```

```
CMP    r3,#0
 ADDNE r0,r1,r2
```

- By default, data processing instructions do not affect the condition code flags but the flags can be optionally set by using "S". CMP does not need "S".

```
loop
  …
  SUBS r1,r1,#1
  BNE  loop
```

decrement r1 and set flags

if Z flag clear then branch

Condition Codes

- The possible condition codes are listed below:
  - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N=!V
AL	Always	

Examples of conditional execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);
    CMP      r0,#0
    MOVEQ    r0,#1
    BLEQ     func
```

- Set the flags, then use various condition codes

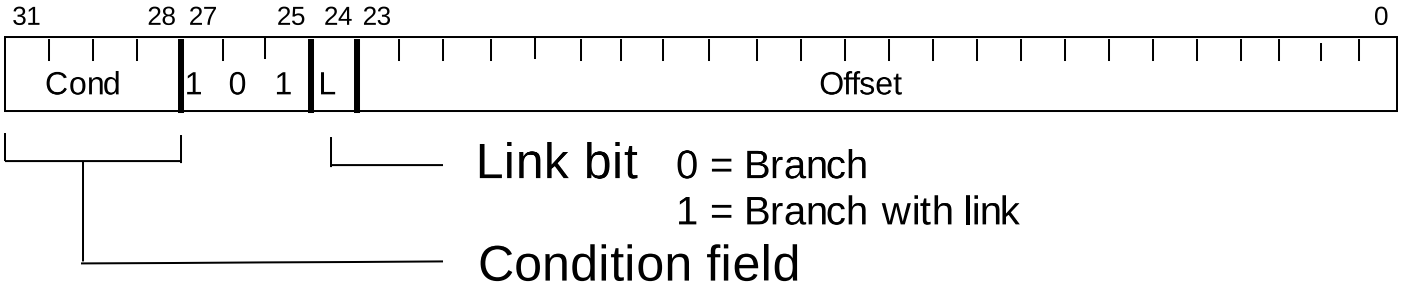
```
if (a==0) x=0;
if (a>0)  x=1;
    CMP      r0,#0
    MOVEQ    r1,#0
    MOVGT    r1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
    CMP      r0,#4
    CMPNE    r0,#10
    MOVEQ    r1,#0
```

Branch instructions

- Branch : B{<cond>} label
- Branch with Link : BL{<cond>} subroutine_label



- The processor core shifts the offset field left by 2 positions, sign-extends it and adds it to the PC
  - ± 32 Mbyte range
  - How to perform longer branches?

Data processing Instructions

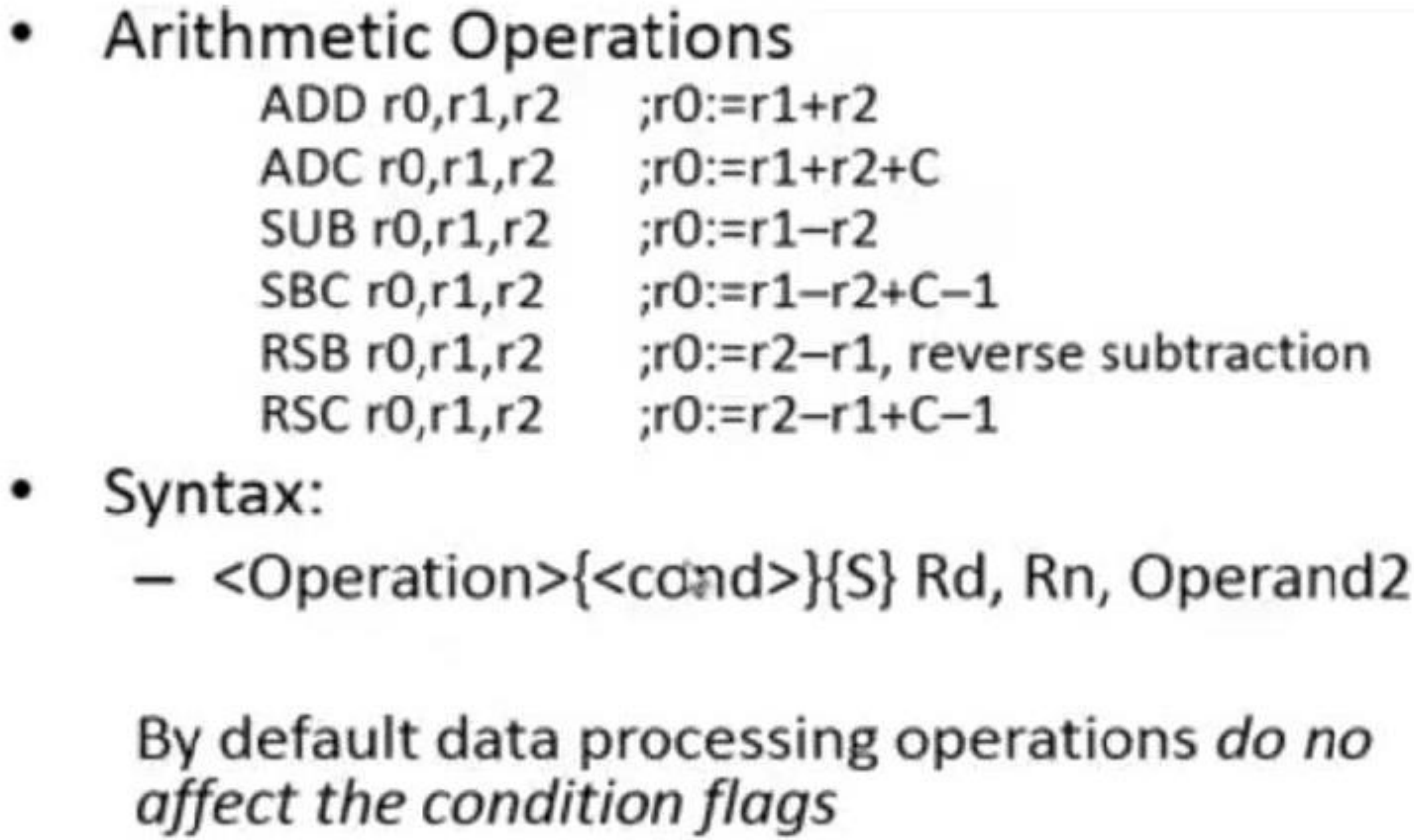
- Consist of :

– Arithmetic:	ADD	ADC	SUB	SBC	RSB	RSC
– Logical:	AND	ORR	EOR	BIC		
– Comparisons:	CMP	CMN	TST	TEQ		
– Data movement:	MOV	MVN				

- These instructions only work on registers, NOT memory.

- Second operand is sent to the ALU via barrel shifter.

EXAMPLE



- Arithmetic Operations

```
ADD r0,r1,r2     ;r0:=r1+r2
ADC r0,r1,r2     ;r0:=r1+r2+C
SUB r0,r1,r2     ;r0:=r1–r2
SBC r0,r1,r2     ;r0:=r1–r2+C–1
RSB r0,r1,r2     ;r0:=r2–r1, reverse subtraction
RSC r0,r1,r2     ;r0:=r2–r1+C–1
```

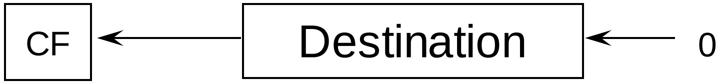
- Syntax:
  - <Operation>{<cond>}{S} Rd, Rn, Operand2

By default data processing operations *do no affect the condition flags*

Barrel Shifter & Memory Instructions

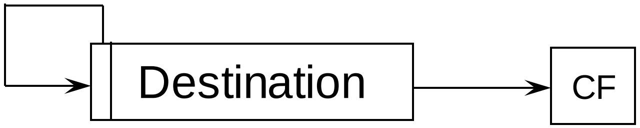
The Barrel Shifter

LSL : Logical Left Shift



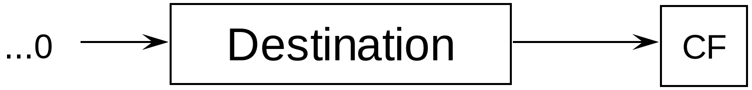
Multiplication by a power of 2

ASR: Arithmetic Right Shift



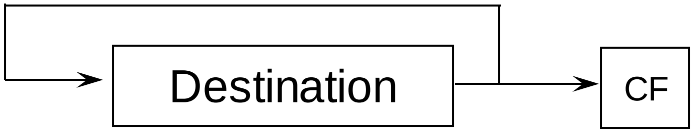
Division by a power of 2, preserving the sign bit

LSR : Logical Shift Right



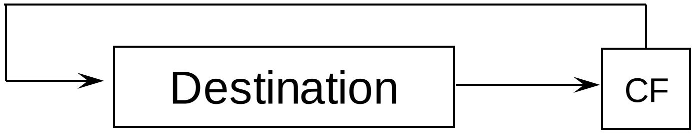
Division by a power of 2

ROR: Rotate Right



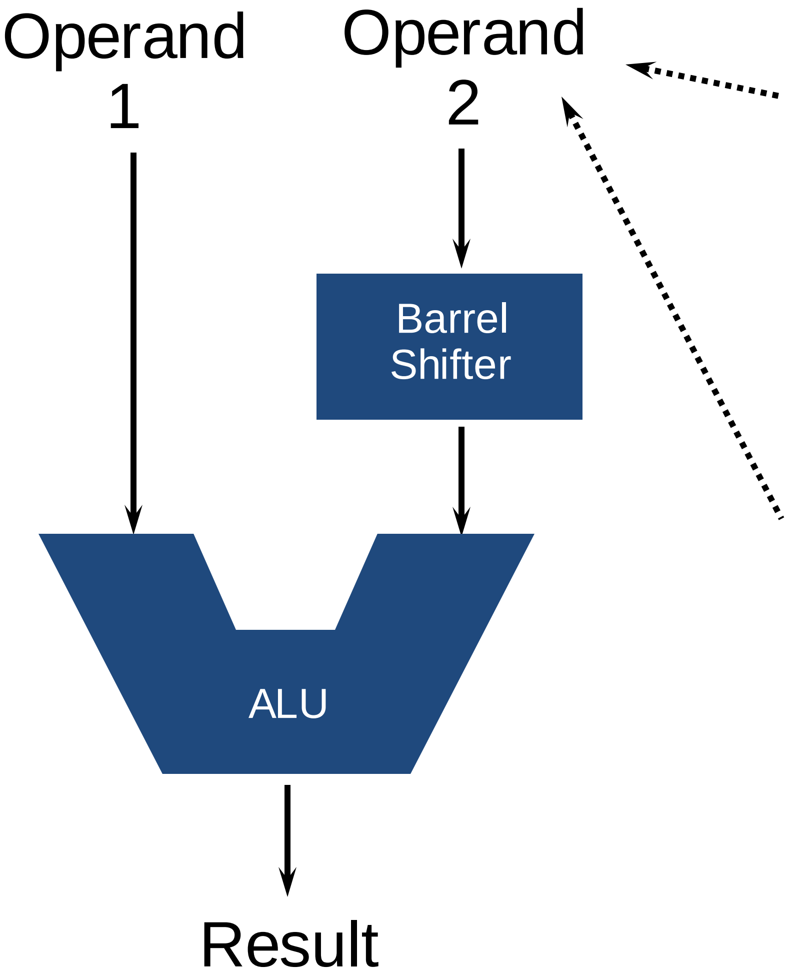
Bit rotate with wrap around from LSB to MSB

RRX: Rotate Right Extended



Single bit rotate with wrap around from CF to MSB

Using the Barrel Shifter: The Second Operand



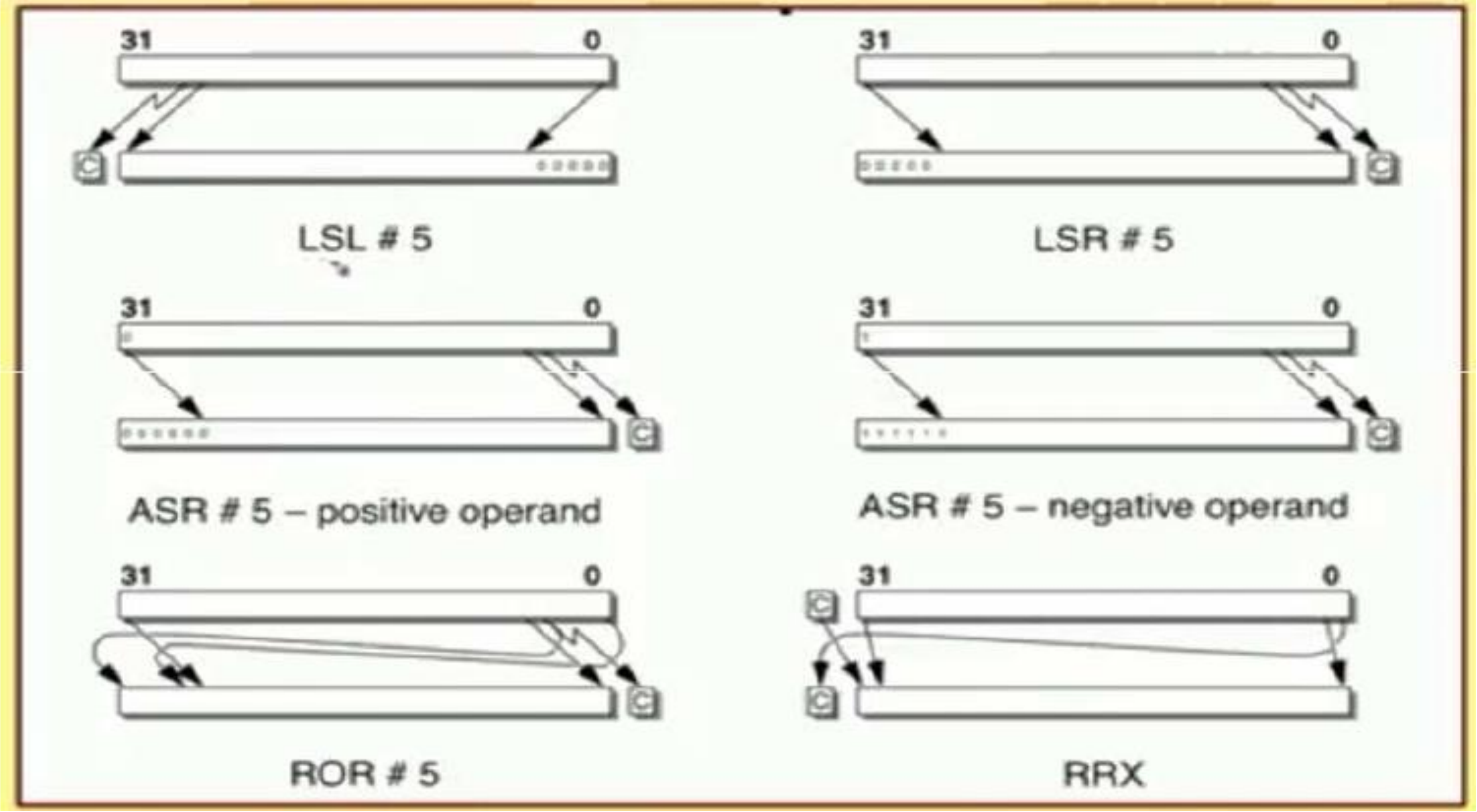
Register, optionally with shift operation

- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

- 8 bit number, with a range of 0-255.
  - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers

EXAMPLE



LSL # 5

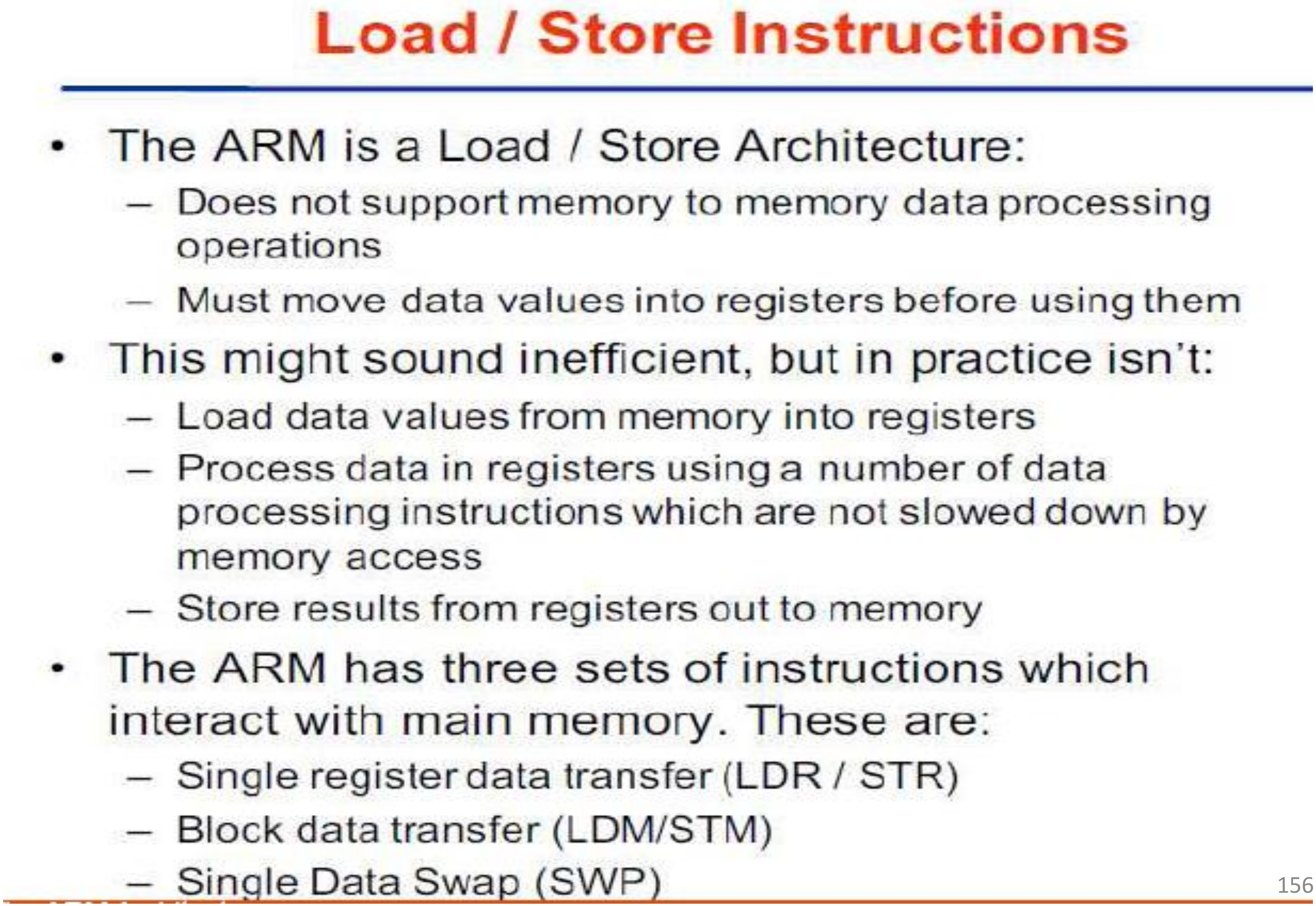
LSR # 5

ASR # 5 – positive operand

ASR # 5 – negative operand

ROR # 5

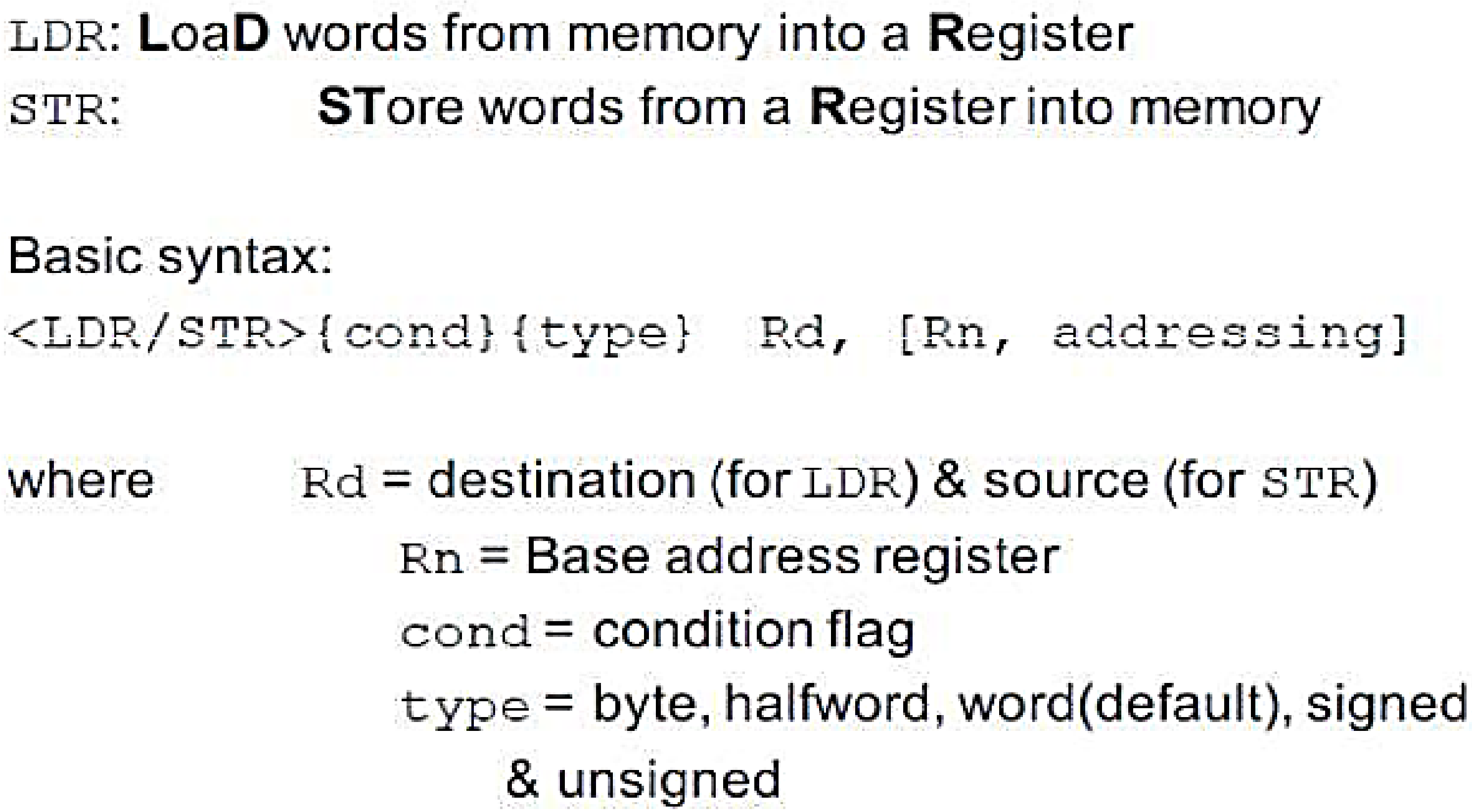
RRX



Load / Store Instructions

- The ARM is a Load / Store Architecture:
  - Does not support memory to memory data processing operations
  - Must move data values into registers before using them
- This might sound inefficient, but in practice isn't:
  - Load data values from memory into registers
  - Process data in registers using a number of data processing instructions which are not slowed down by memory access
  - Store results from registers out to memory
- The ARM has three sets of instructions which interact with main memory. These are:
  - Single register data transfer (LDR / STR)
  - Block data transfer (LDM/STM)
  - Single Data Swap (SWP)

Single register data transfer



LDR: **L**oa**D** words from memory into a **R**egister

STR: **ST**ore words from a **R**egister into memory

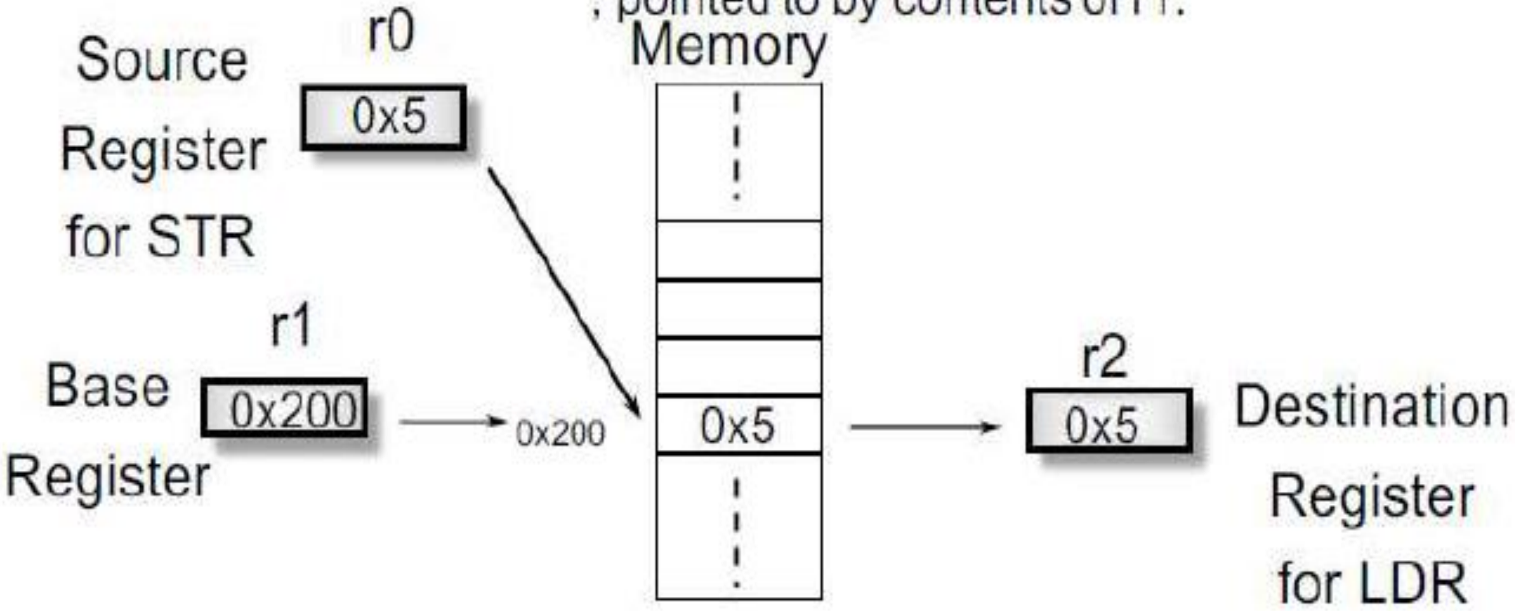
Basic syntax:

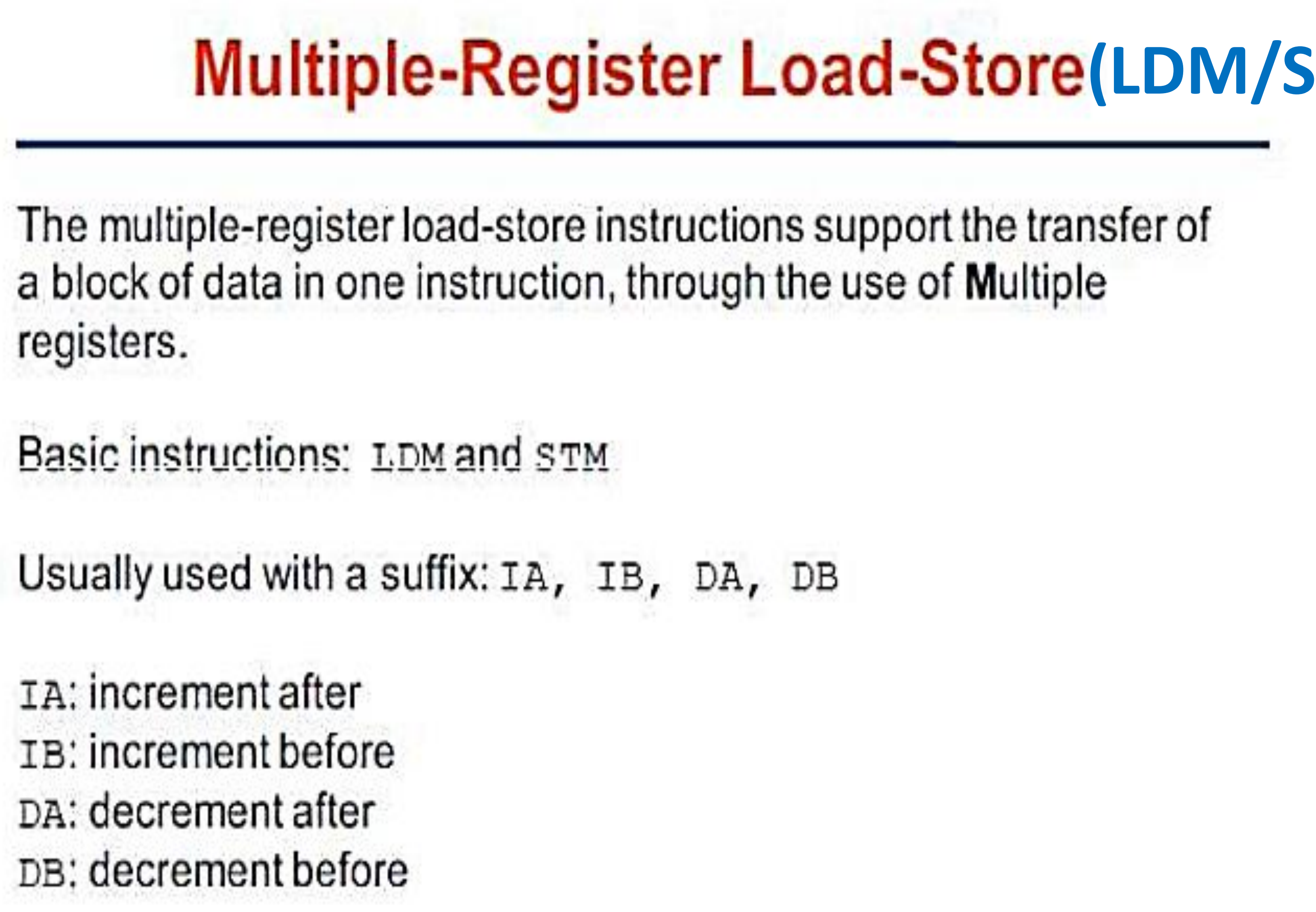
```
<LDR/STR>{cond}{type}  Rd,  [Rn,  addressing]
```

where Rd = destination (for LDR) & source (for STR)
Rn = Base address register
cond = condition flag
type = byte, halfword, word(default), signed & unsigned

Base Register

- The memory location to be accessed is held in a base register
  - STR r0, [r1] ; Store contents of r0 to location pointed to ; by contents of r1.
  - LDR r2, [r1] ; Load r2 with contents of memory location ; pointed to by contents of r1.





Multiple-Register Load-Store(LDM/STM)

The multiple-register load-store instructions support the transfer of a block of data in one instruction, through the use of **M**ultiple registers.

Basic instructions: `LDM` and `STM`

Usually used with a suffix: `IA, IB, DA, DB`

`IA`: increment after
`IB`: increment before
`DA`: decrement after
`DB`: decrement before

(LDM/STM) OPERATIONS

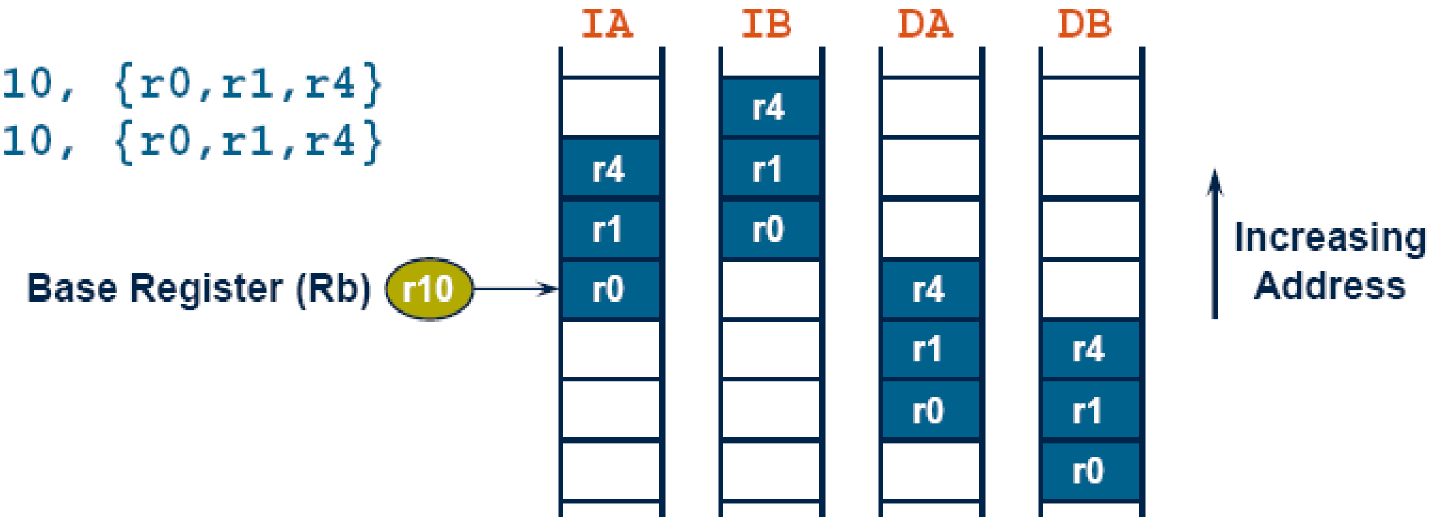
- Syntax:

  <LDM|STM>{<cond>}<addressing_mode> Rb{!}, <register list>

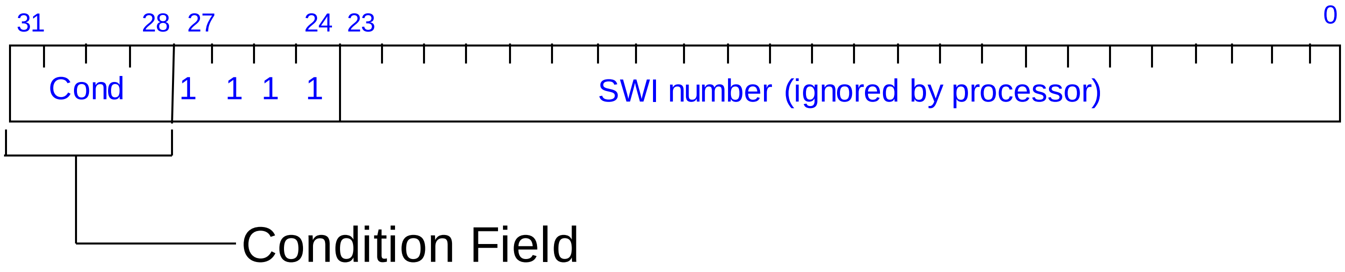
- 4 addressing modes:

LDMIA / STMIA	increment after
LDMIB / STMIB	increment before
LDMDA / STMDA	decrement after
LDMDB / STMDB	decrement before

```
LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}
```



Software Interrupt (SWI)

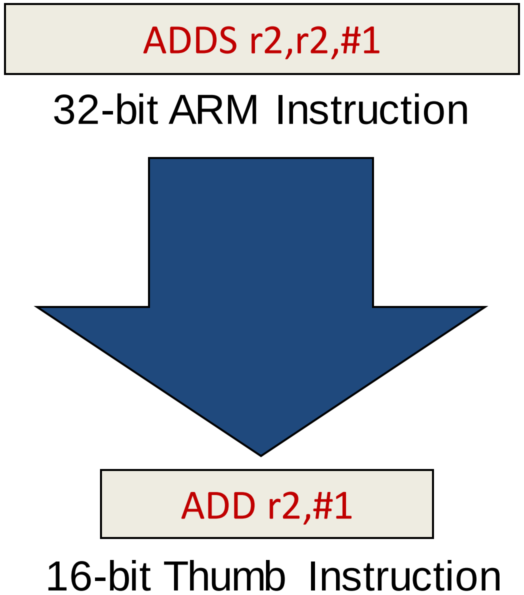


- Causes an exception trap to the SWI hardware vector
- The SWI handler can examine the SWI number to decide what operation has been requested.
- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- Syntax:

```
SWI{<cond>} <SWI number>
```

Thumb

- Thumb is a 16-bit instruction set
  - Optimized for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set
- Core has additional execution state - Thumb
  - Switch between ARM and Thumb using BX instruction



ADDS r2,r2,#1

32-bit ARM Instruction

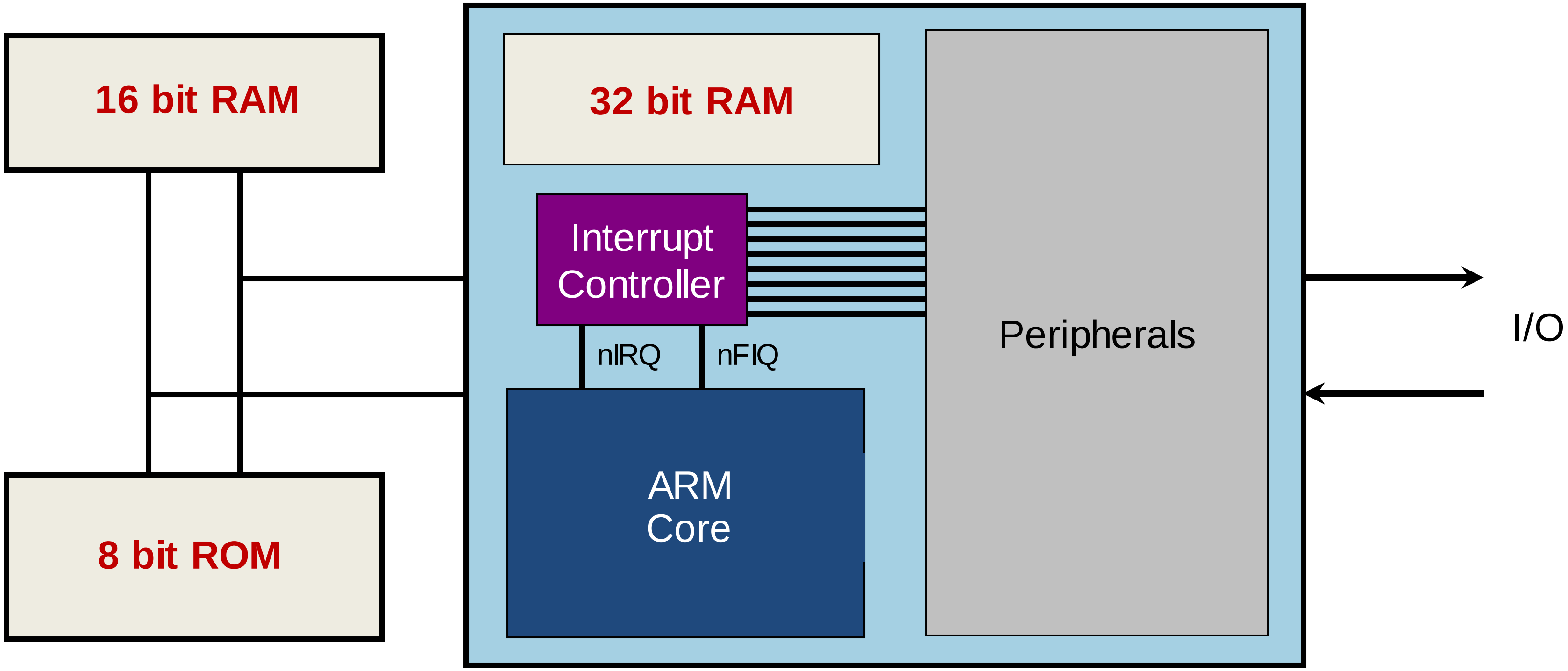
ADD r2,#1

16-bit Thumb Instruction

For most instructions generated by compiler:

- Conditional execution is not used
- Source and destination registers identical
- Only Low registers used
- Constants are of limited size
- Inline barrel shifter not used

Example ARM-based System

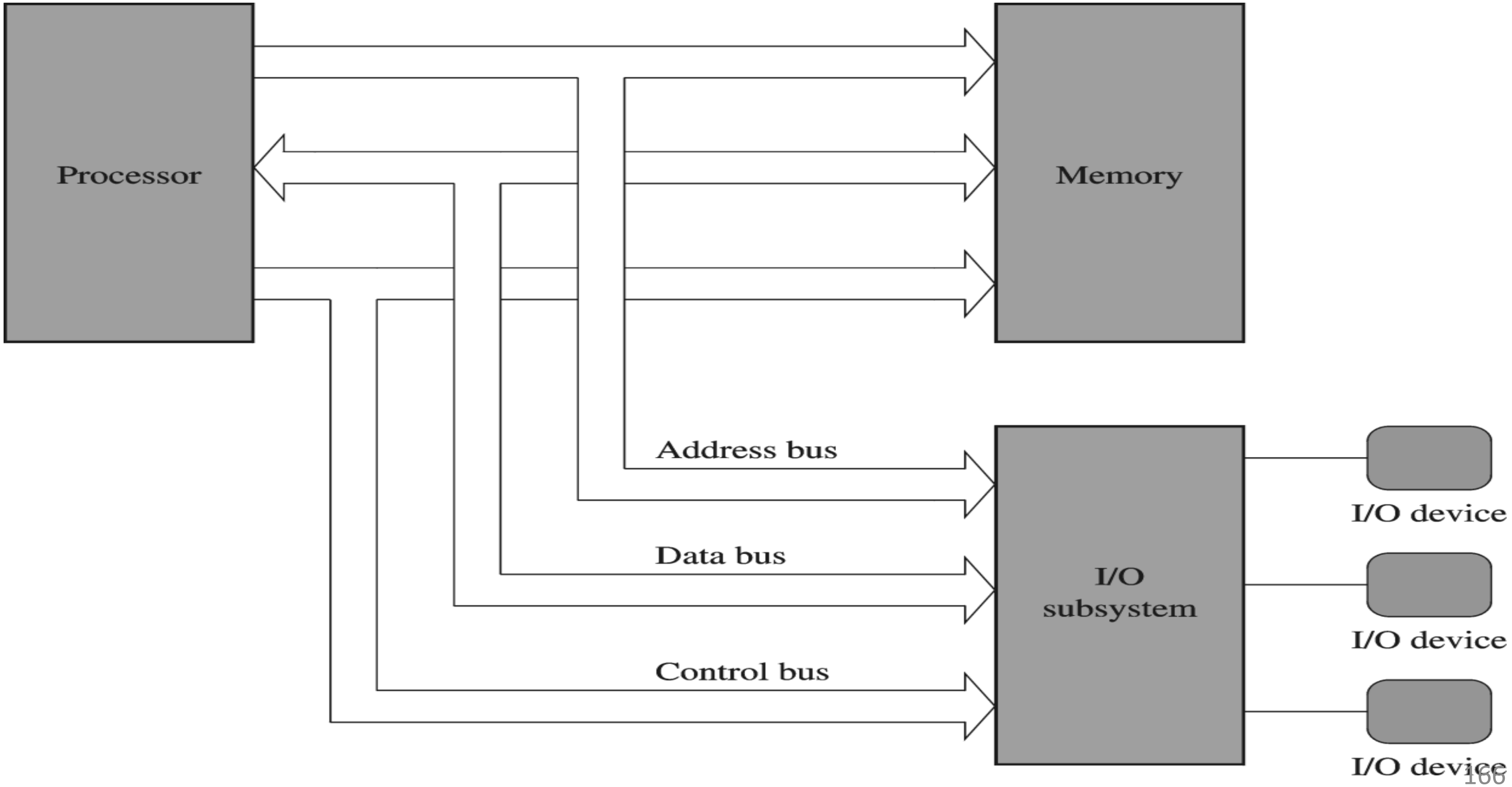


Basics of Input - Output Operations

Input - Output Interface

- Input Output Interface provides a method for transferring information between internal storage and external I/O devices.
- Peripherals connected to a computer need special communication links for interfacing them with the central processing unit.

COMMUNICATION BETWEEN CPU, MEMORY AND I/O DEVICES



BUS

- A bus is a bunch of wires through which data or address or control signals flow.
- The microprocessor communicates with the memory and the Input/Output devices via the three buses, viz**.,** **data bus, address bus and control bus.**
- Data flow through the DB, while address comes out of the AB and CB controls the activities of the microprocessor system at any instant of time.

Input - Output Interface

The purpose of communication link is to resolve the differences that exist between the central computer and each peripheral.

The Major Differences are:-

1. Peripherals are electromechnical and electromagnetic devices and their manner of operation of the CPU and memory, will differ. Therefore, a conversion of signal values may be needed.

2. The data transfer rate of peripherals is usually slower than the transfer rate of CPU and consequently, a synchronization mechanism may be needed.

3. Data codes and formats in the peripherals differ from the word format in the CPU and memory.
4. The operating modes of peripherals are different from each other and must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

Input - Output Interface

- **To Resolve these differences**, computer systems include special hardware components between the CPU and Peripherals to supervise and synchronize all input and out transfers.
- These components are called **Interface Units** because they interface between the processor bus and the peripheral devices.

I/O BUS and Interface Module

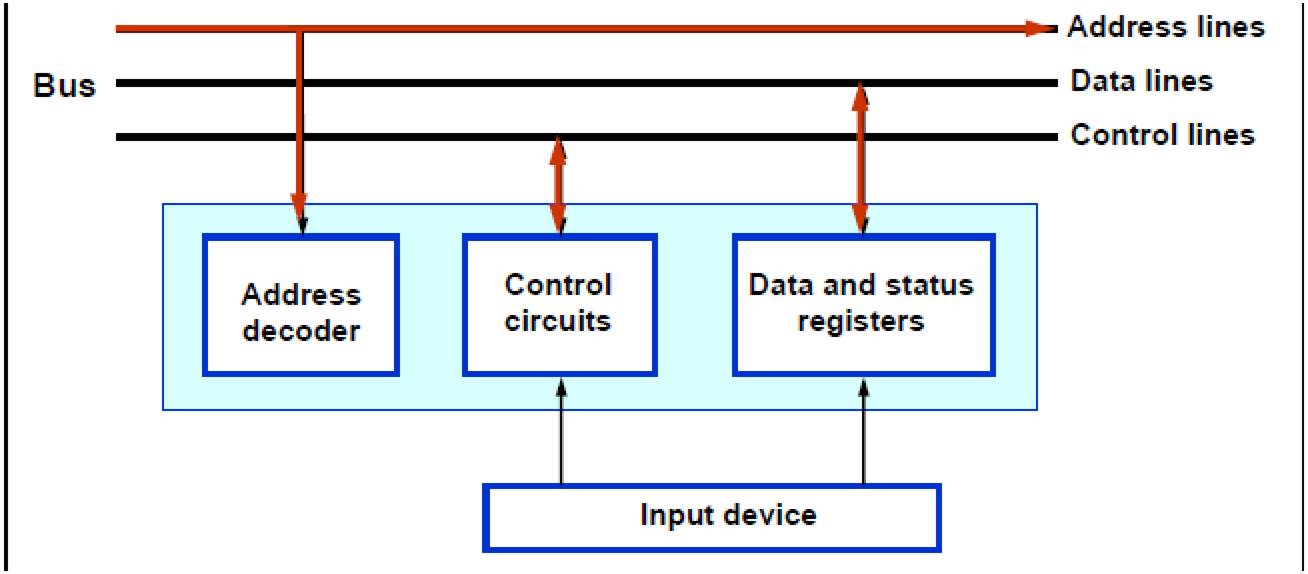
❑ It defines the typical link between the processor and several peripherals.

❑ The I/O Bus consists of **data lines, address lines** and **control lines**.

❑ The I/O bus from the processor is attached to all peripherals interface.

❑ To communicate with a particular device, the processor places a device address on address lines.

Interface Module



I/O BUS and Interface Module

- ❑ Each Interface **decodes** the address and control received from the I/O bus, interprets them for peripherals and provides signals for the **peripheral controller**.
- ❑ It is also synchronizes the data flow and supervises the transfer between peripheral and processor.
- ❑ Each peripheral has its own controller. **For example**, the printer controller controls the paper motion, the print timing.

The control lines are referred as an I/O command. The commands are as following:

Control command- A control command is issued to activate the peripheral and to inform it what to do.

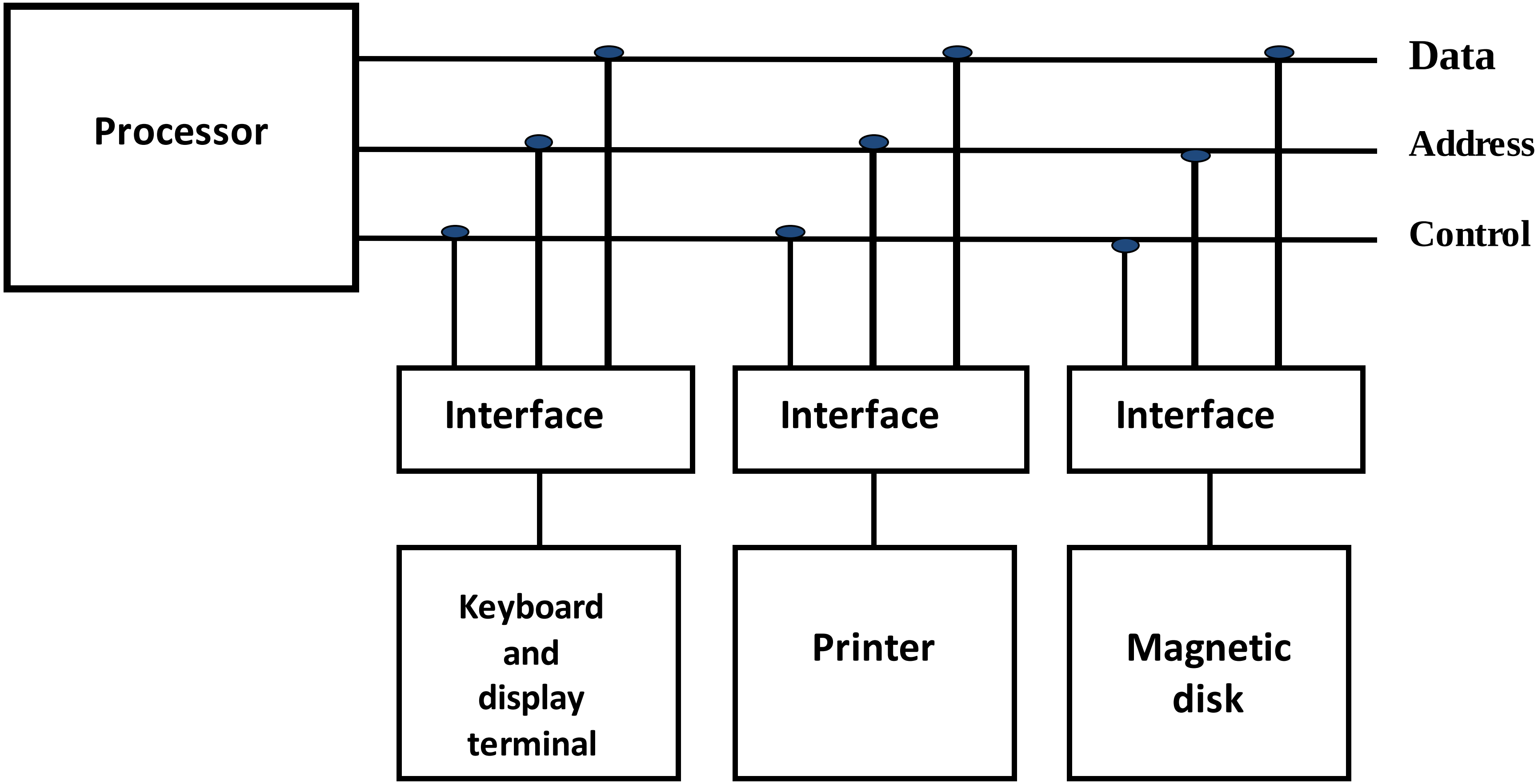
Status command- A status command is used to test various status conditions in the interface and the peripheral.

I/O BUS and Interface Module

Output data command- A data output command causes the interface to respond by transferring data from the bus into one of its registers.

Input data command- The data input command is the opposite of the data output. In this case the interface receives on item of data from the peripheral and places it in its buffer register.

I/O BUS and Interface Module



Connection of I/O bus to input-output devices

I/O Versus Memory Bus

There are 3 ways that computer buses can be used to communicate with memory and I/O:

i. Use **two Separate buses** , one for memory and other for I/O.

ii. Use **one common bus** for both memory and I/O but separate control lines for each.

iii. Use **one common bus for memory and I/O with common control lines.**