UNIT – II **15 Periods**

Instruction set architecture of a CPU: Memory Locations and Addresses – Memory Operations - Instructions and Instruction Sequencing - Addressing Modes – Assembly Language - Stacks - Subroutines - Additional Instructions - CISC Instruction Sets.

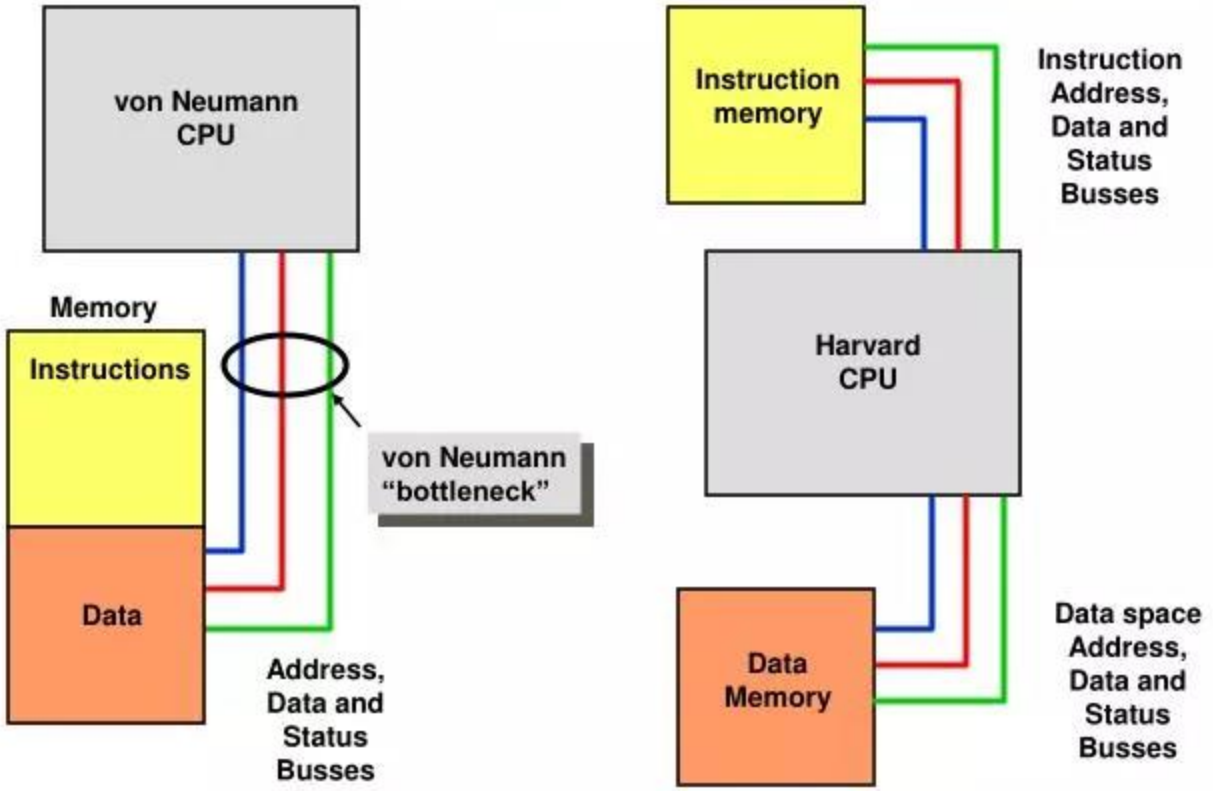
Introduction to x86 architecture: The Intel IA-32 Architecture: Memory Organization - Register Structure - Addressing Modes – Instruction Set – Interrupts and Exceptions.

MICROPR-OCESSOR	YEAR OF INTRO-DUCTION	WORD LENGTH	MEMORY ADDRESSING CAPA-CITY	PINS	CLOCK	REMARKS
4004	1971	4-BIT	1 KB	16	750 KHZ	FIRST MICROPROCESSOR
8085	1973	8-BIT	64 KB	40	3-6 MHZ	POPULAR 8-BIT PROCESSOR
8086	1978	16-BIT	1 MB	40	5-10 MHZ	
8088	1980	16-BIT	1 MB	40	5-8MHZ	
80286	1982	16-BIT	16 MB REAL, 4GB VIRTUAL	68	6-12.5 MHZ	WIDELY USED IN PC
80386	1985	32-BIT	4 GB REAL, 64TB VIRTUAL	100	20 MHZ	WIDELY USED IN PC
80486	1989	32-BIT	4 GB REAL, 64TB VIRTUAL	168	25-100 MHZ	WIDELY USED IN PC
PENTIUM	1993	32-BIT	4 GB REAL	237 PGA	233 MHZ	
PENTIUM PRO	1995	32BIT	64 GB REAL	387 PIN PGA	150-200 MHZ	
PENTIUM II		32BIT	64 GB REAL		450 MHZ	
CELERON	1998	32 BIT			2.6MHZ	
PENTIUM III	1999	32 BIT	64 REAL	370 PGA	500-1000 MHZ	
PENTIUM IV	2004	64 BIT	64 GB	423 PGA	1.3-3.2 GHZ	
PENTIUM 4EE And PENTIUM 6XX SERIES				-	3-3.7 GHZ	
ITANIUM	2001	64 BIT		423 PGA	-	EPIC PROCESSOR
CORE DUO	2006	64 BIT		775 LGA	1.6-2.66	

Von Neumann vs Harvard architecture

	Von Neumann Architecture	Harvard Architecture
Flexibility	High level of flexibility as the memory is shared between instructions and data so the level assigned to each can fluctuate depending on task	Limited flexibility as there is only a certain amount of memory that can be used for data and a certain amount for instructions.
Speed	Speed is limited when compared to harvard due to only having one memory location and set of buses	Two sets of memory and buses mean data can be handled more quickly which would result in decreasing execution time
Examples	Typically used in general purpose computers that will be used for many different purposes.	Typically embedded systems like washing machines, burglar alarms etc.

von Neumann and Harvard Architectures



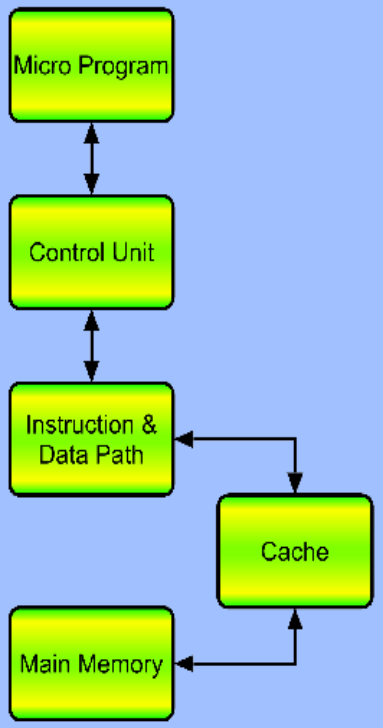
RISC vs. CISC

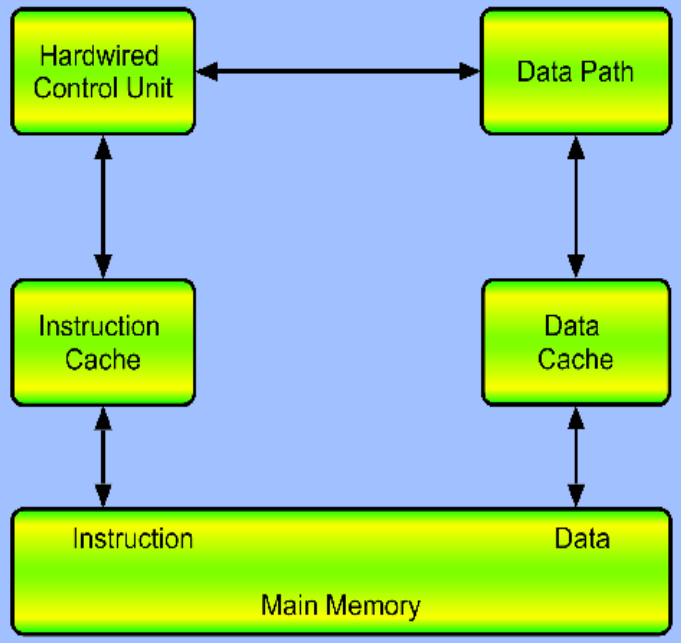
CISC	RISC
Emphasis on hardware	Emphasis on software
Multiple instruction sizes and formats	Instructions of same set with few formats
Less registers	Uses more registers
More addressing modes	Fewer addressing modes
Extensive use of microprogramming	Complexity in compiler
Instructions take a varying amount of cycle time	Instructions take one cycle time
Pipelining is difficult	Pipelining is easy

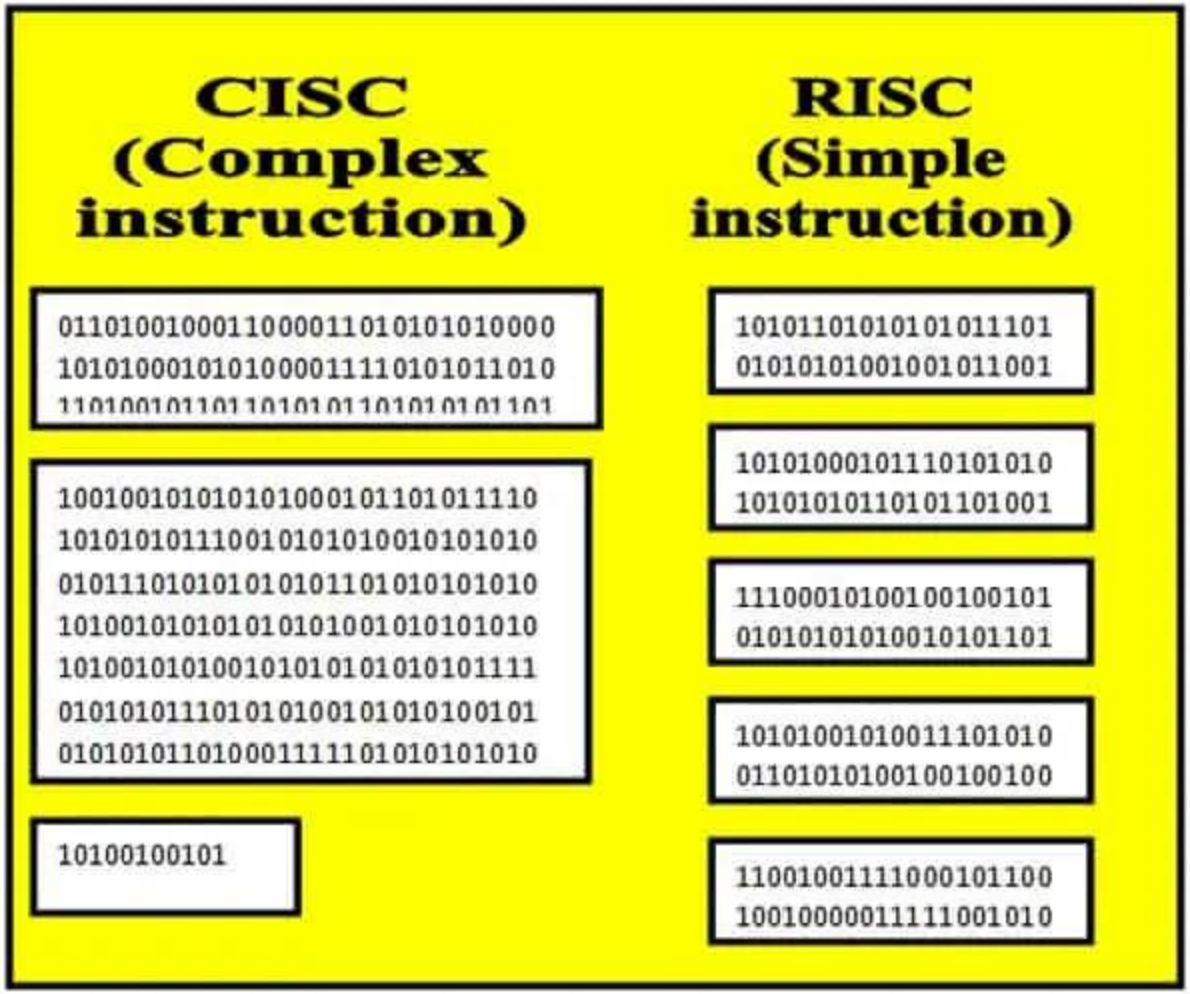
CISC versus RISC

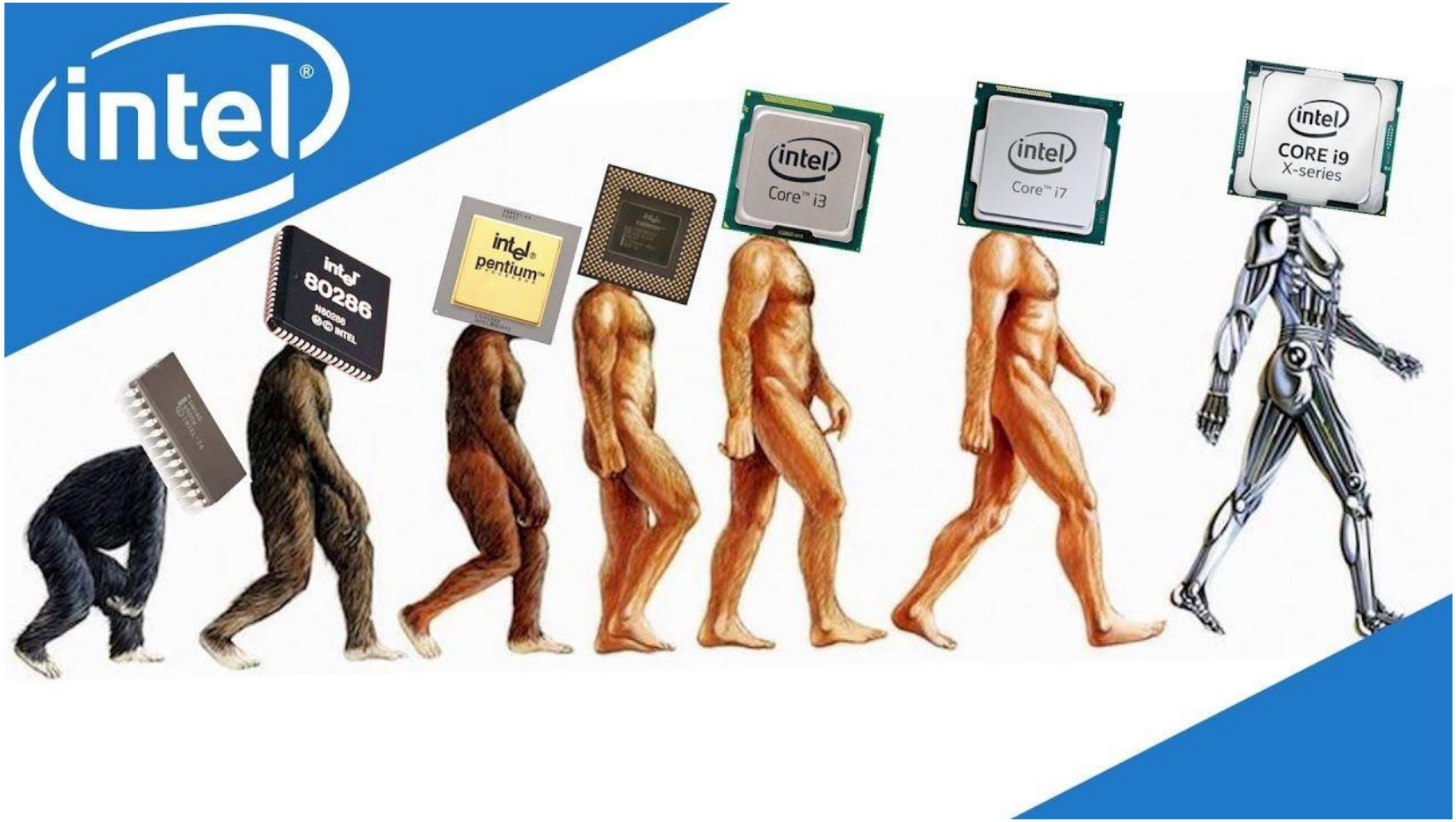
CISC	RISC
Emphasis on hardware	Emphasis on software
Includes multi-clock complex instructions	Single-clock, reduced instruction only
Memory-to-memory: "LOAD" and "STORE" incorporated in instructions	Register to register: "LOAD" and "STORE" are independent instructions
Small code sizes, high cycles per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

CISC vs RISC


Micro Program
Control Unit
Instruction & Data Path
Cache
Main Memory


Hardwired Control Unit
Data Path
Instruction Cache
Data Cache
Instruction
Data
Main Memory

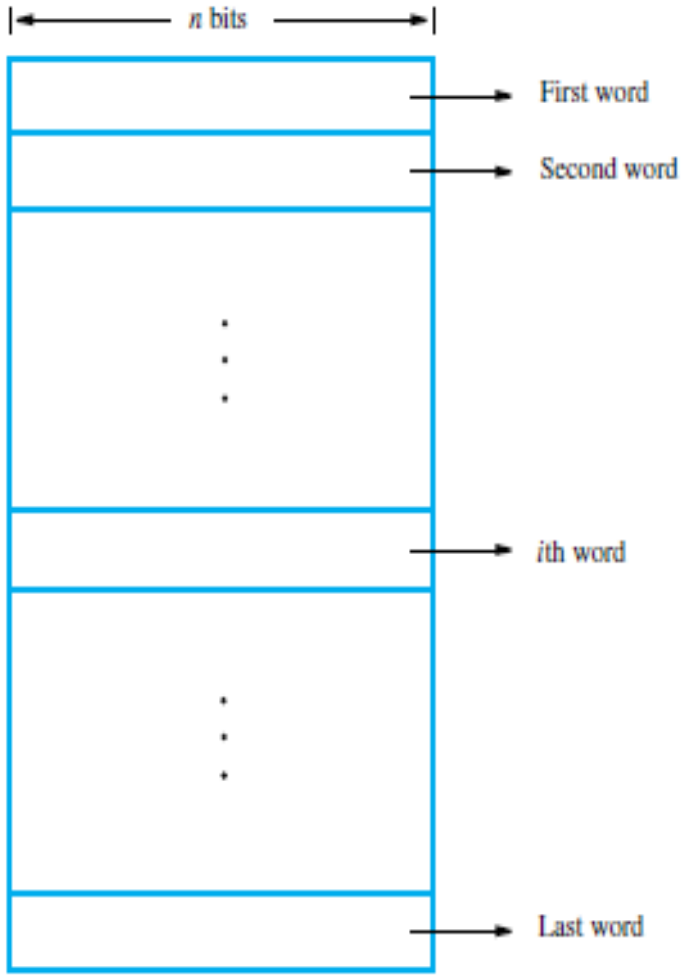

CISC (Complex instruction)
RISC (Simple instruction)
0110100100011000011010101010000
1010100010101000011110101011010
1001001010101010001011010111110
1010101011100101010100101010101 0
0101110101010101011010101010 10
1010010101010101010100101010101 0
1010010101001010101010101011 11
0101010111010101001010101001 01
0101010110100011111010101010 10
10100100101
1010110101010101011101
0101010100100101 1001
1010100010111010101 0
1010101011010110 1001
11100010100100100101
0101010101001010 1101
1010100101001110101 0
0110101010010010 0100
1100100111100010110 0
1001000000111110010 10


intel
intel
80286
intel
pentium
intel
Core™ i3
intel
Core™ i7
intel
CORE i9
X-series

Memory Locations and Addresses

Memory Locations and Addresses

- The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1.
- The memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation.
- Each group of n bits is referred to as a word of information, and n is called the word length.



Memory word

- Modern computers have word lengths that typically range from 16 to 64 bits.

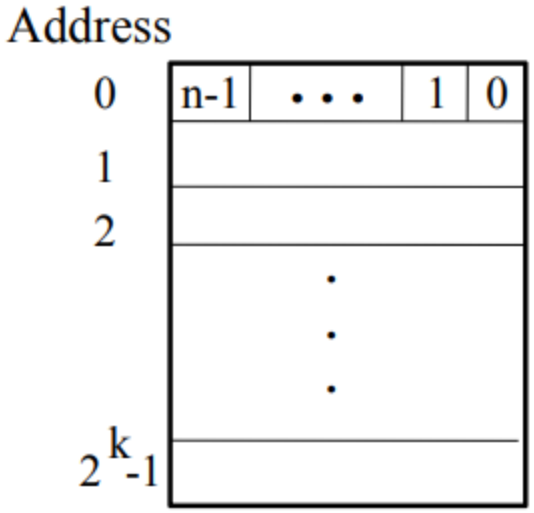
- If the word length of a computer is 32 bits, a single word can store a 32-bit signed number or four ASCII-encoded characters, each occupying 8 bits.

- A unit of 8 bits is called a byte.

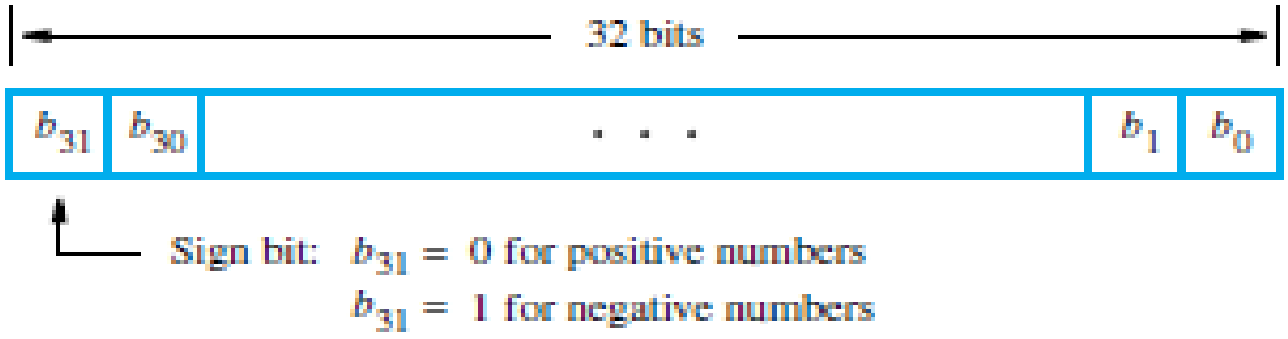
- Machine instructions may require one or more words for their representation.

- Memory holds both instructions and data.
- k address bits and n bits per location.

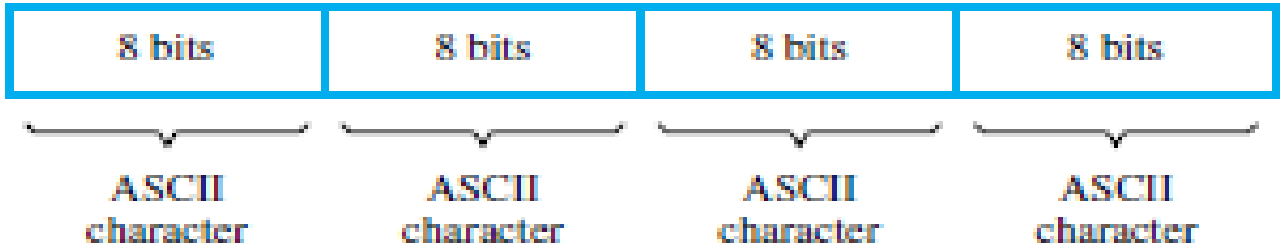
k	Number of locations
10	$2^{10} = 1024 = 1K$
16	$2^{16} = 65,536 = 64K$
20	$2^{20} = 1,048,576 = 1M$
24	$2^{24} = 16,777,216 = 16M$



- n is typically 8 (byte), 16 (word), 32 (long word), ….



(a) A signed integer



(b) Four characters

Examples of encoded information in a 32-bit word.

Byte Addressability

- A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits.

- It is impractical to assign distinct addresses to individual bit locations in the memory

- The term byte-addressable memory is used for this assignment.

- Byte locations have addresses 0, 1, 2, ...

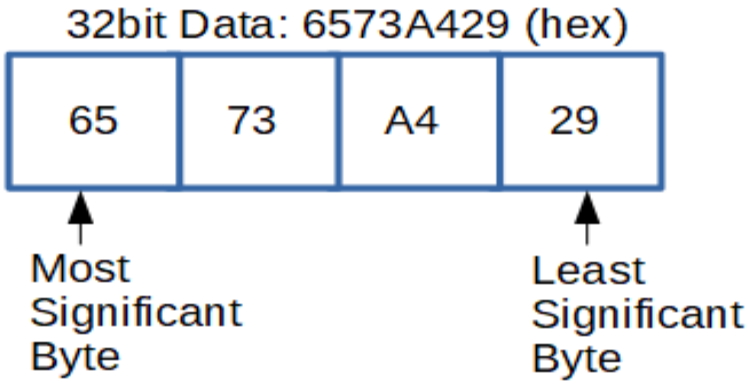
- Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0, 4, 8, . . . , with each word consisting of four bytes.

Big-Endian and Little-Endian Assignments

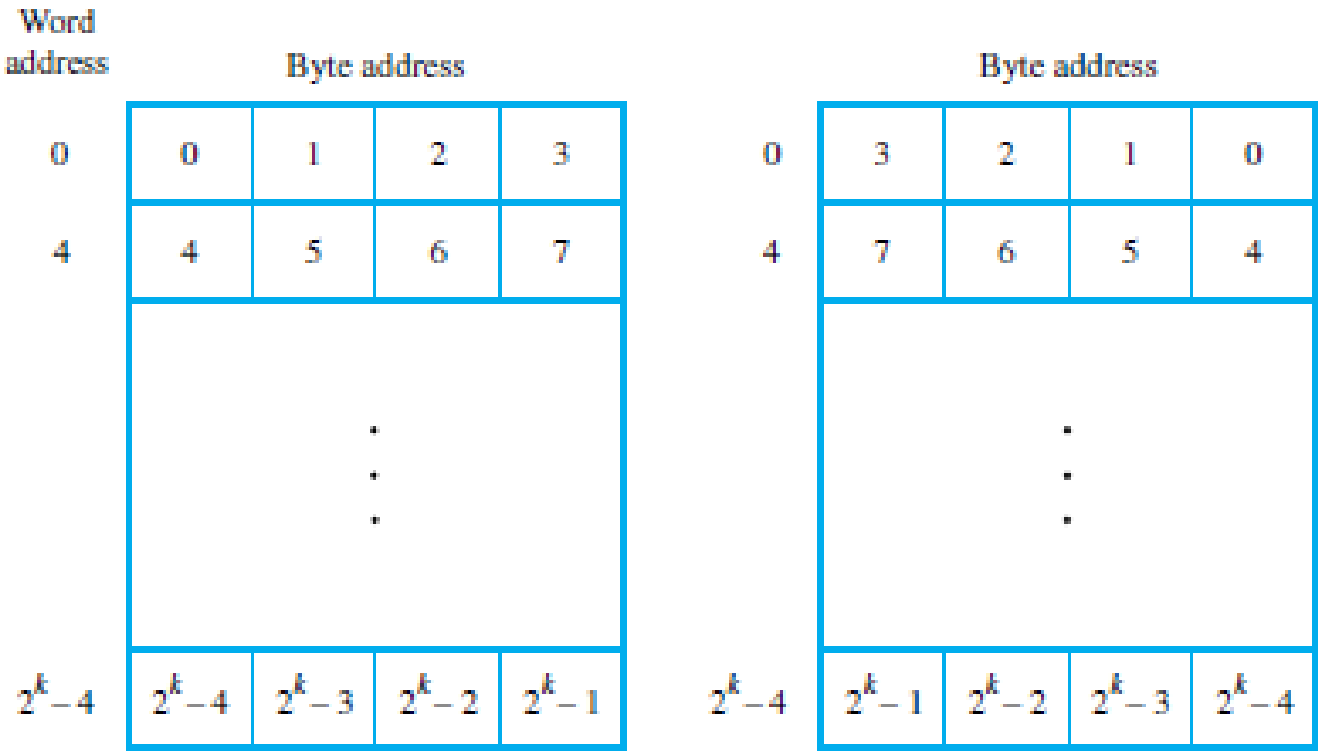
- There are two ways that byte addresses can be assigned across words.

- **Little-endian** – The bytes are ordered with the least significant byte placed at the lowest address.
- **Big-endian** – The bytes are ordered with the most significant byte placed at the lowest address.

- The words “more significant” and “less significant” are used in relation
- to the weights (powers of 2) assigned to bits when the word represents a number.



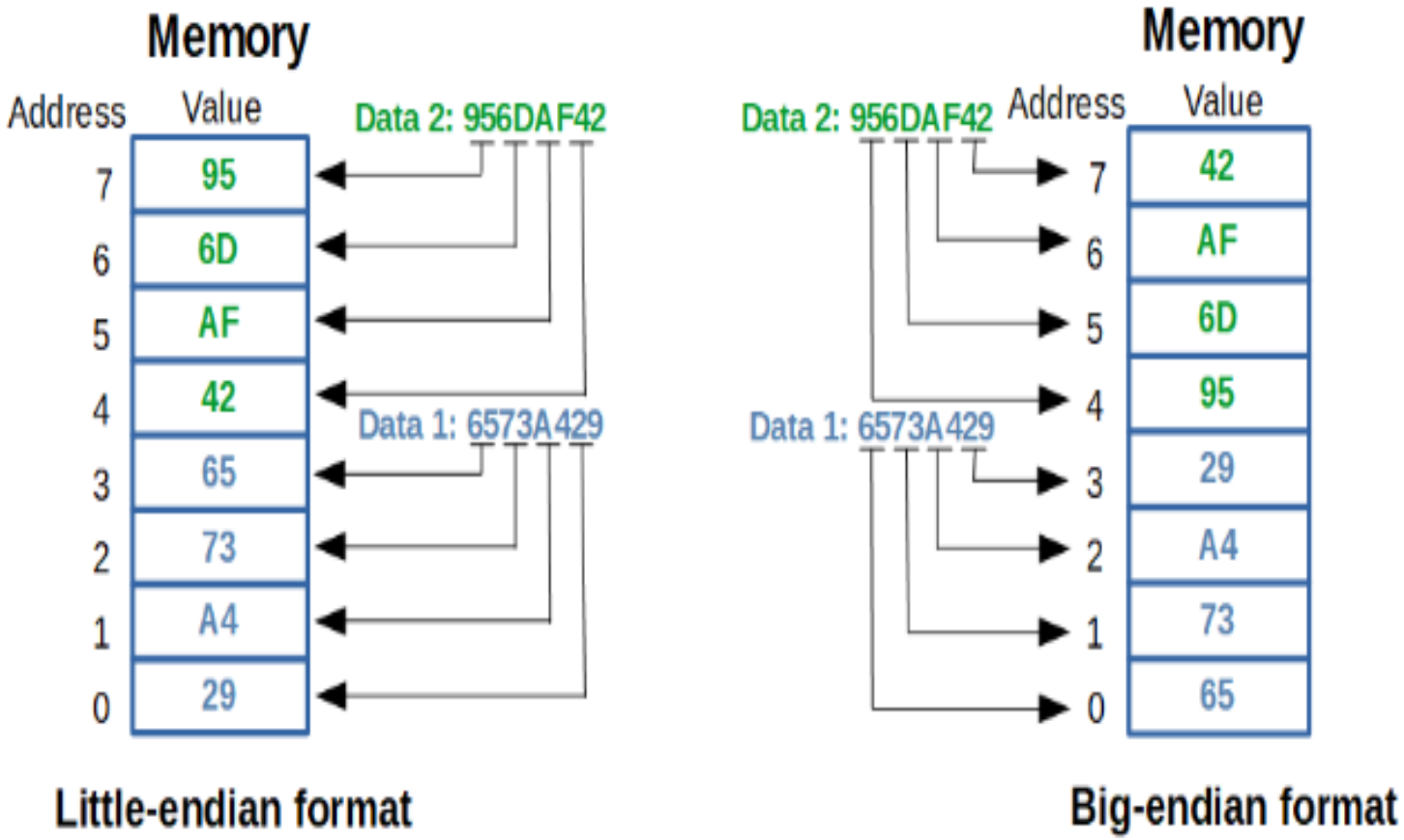
Visualization of a 32-bit data consisting of four bytes



(a) Big-endian assignment

(b) Little-endian assignment

Byte and word addressing.



Little-endian vs big-endian format

Memory Operations

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

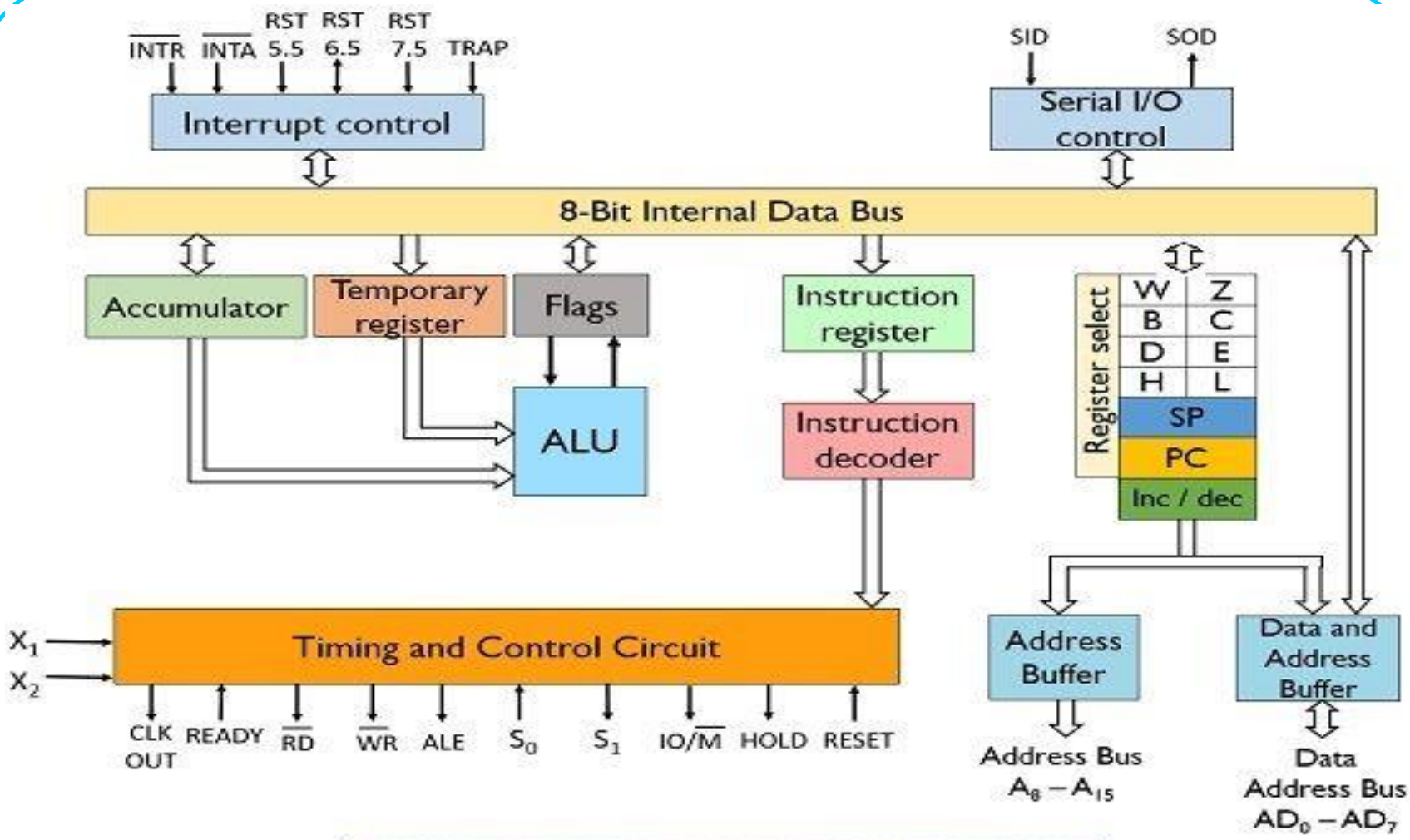
- Load operation (Read or Fetch)-Contents of specified memory location are read by processor
- Store operation (Write)- Data from the processor is stored in specified memory location

Instructions and Instruction Sequencing

INSTRUCTIONS AND INSTRUCTION SEQUENCING

A computer must have instructions capable of performing four types of operations:

- Data transfers between the memory and the processor registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers



Architecture of 8085 Microprocessor

Register Transfer Notation

To describe the transfer of information, the contents of any location are denoted by placing square brackets around its name.

$$R1 \leftarrow [LOC]$$

Thus, this expression means that the contents of memory location LOC are transferred into processor register R1.

As another example, consider the operation that adds the contents of registers R1 and R2, and places their sum into register R3.

This action is indicated as

$$R3 \leftarrow [R1] + [R2]$$

This type of notation is known as **Register Transfer Notation (RTN)**

Assembly-Language Notation

We need another type of notation to represent machine instructions and programs.

For this, we use assembly language. For example, a generic instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement

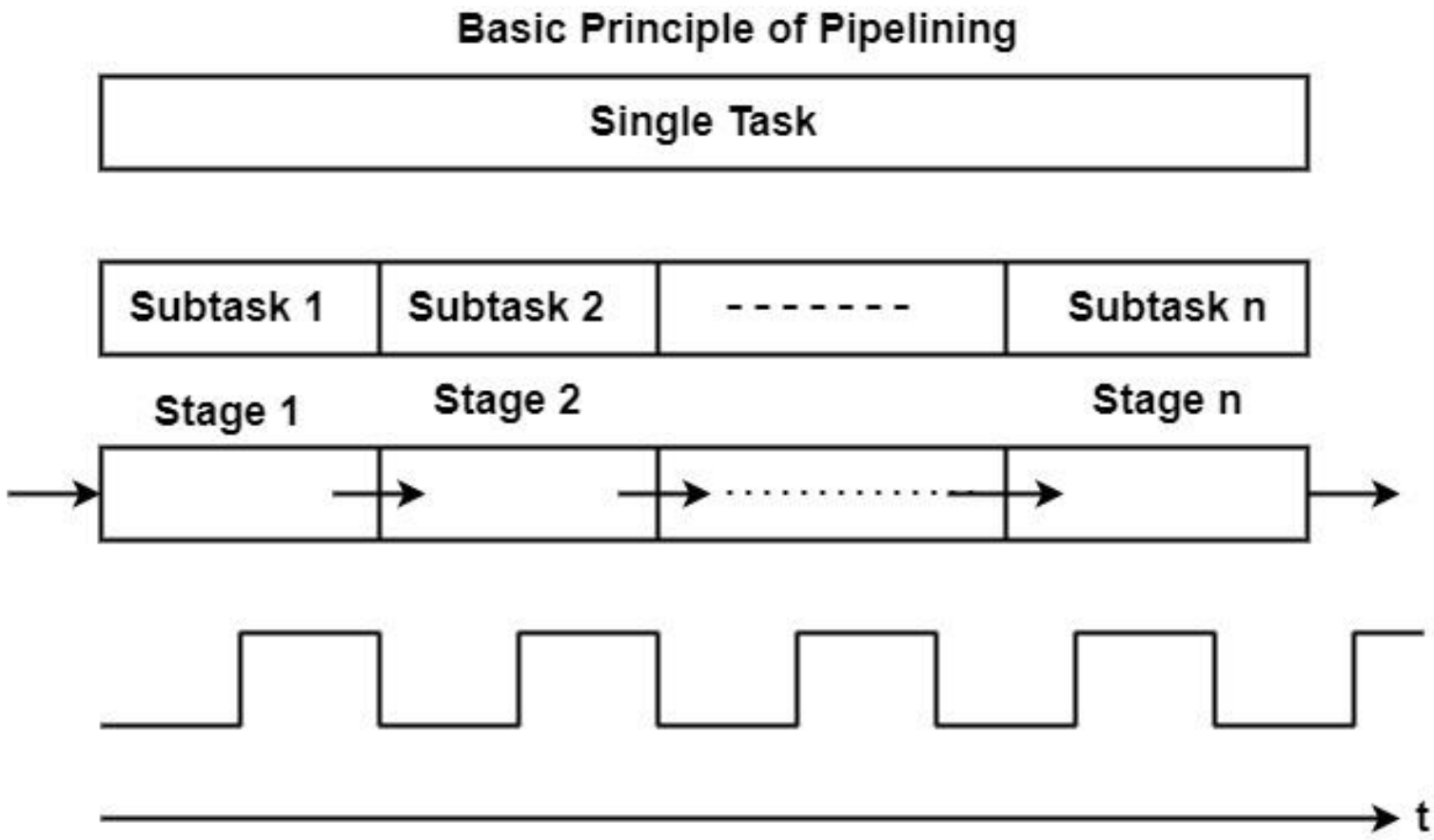
Move LOC, R1

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

The second example of adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly-language statement

Add R1, R2, R3

In this case, registers R1 and R2 hold the source operands, while R3 is the destination.

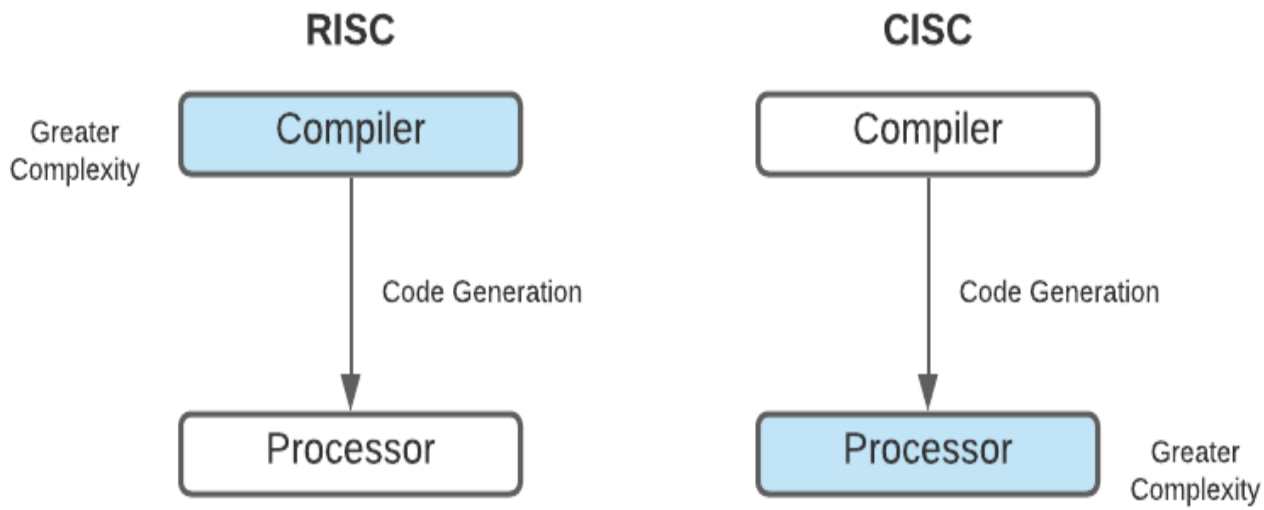

Basic Principle of Pipelining
Single Task
Subtask 1
Subtask 2
- - - - - - -
Subtask n
Stage 1
Stage 2
Stage n
t

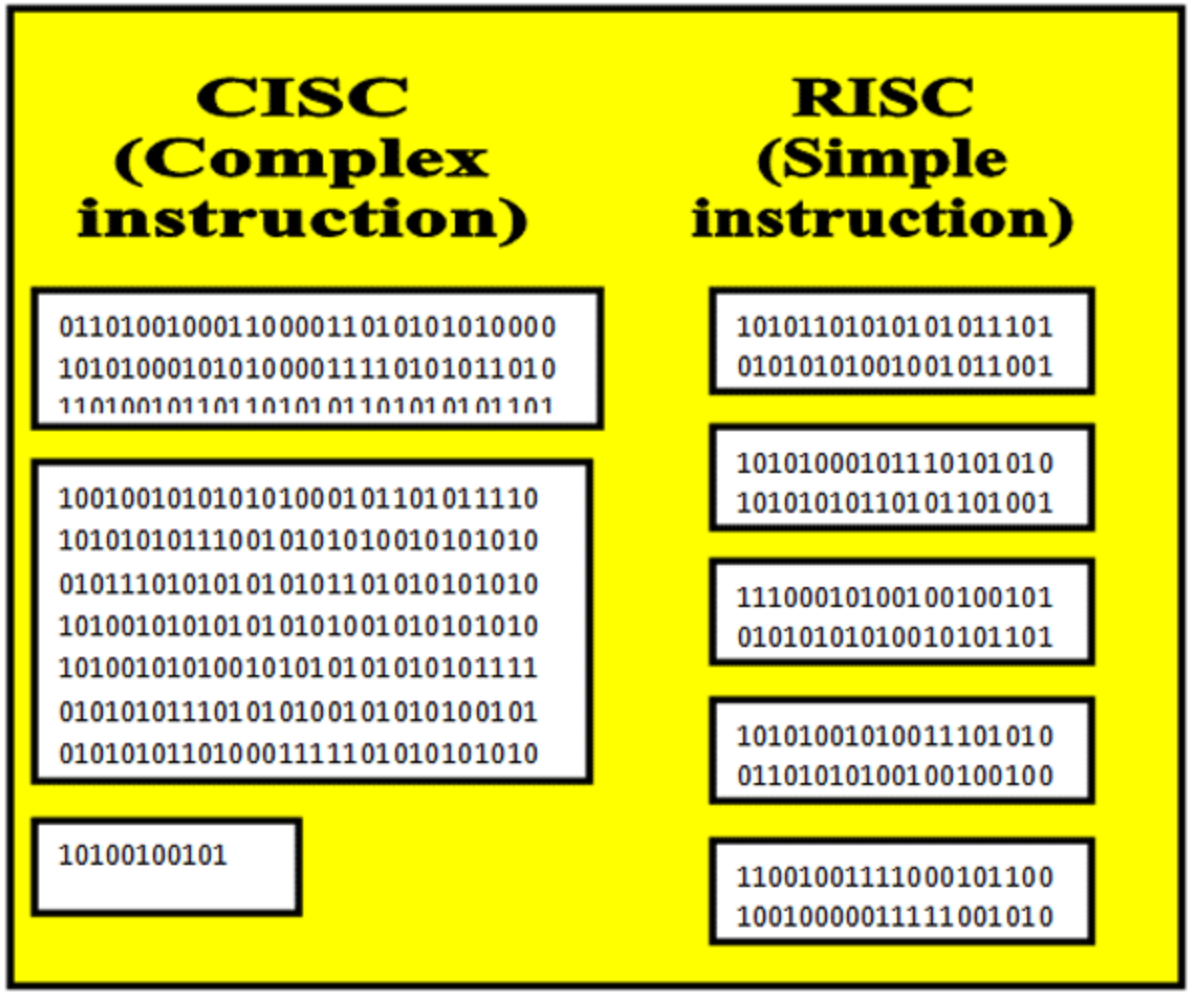
Pipelining:

	cycle 1	cycle 2	cycle 3	cycle 4	cycle 5	cycle 6
Instruction 1	Fetch	Decode	Execute			
Instruction 2		Fetch	Decode	Execute		
Instruction 3			Fetch	Decode	Execute	
Instruction 4				Fetch	Decode	Execute

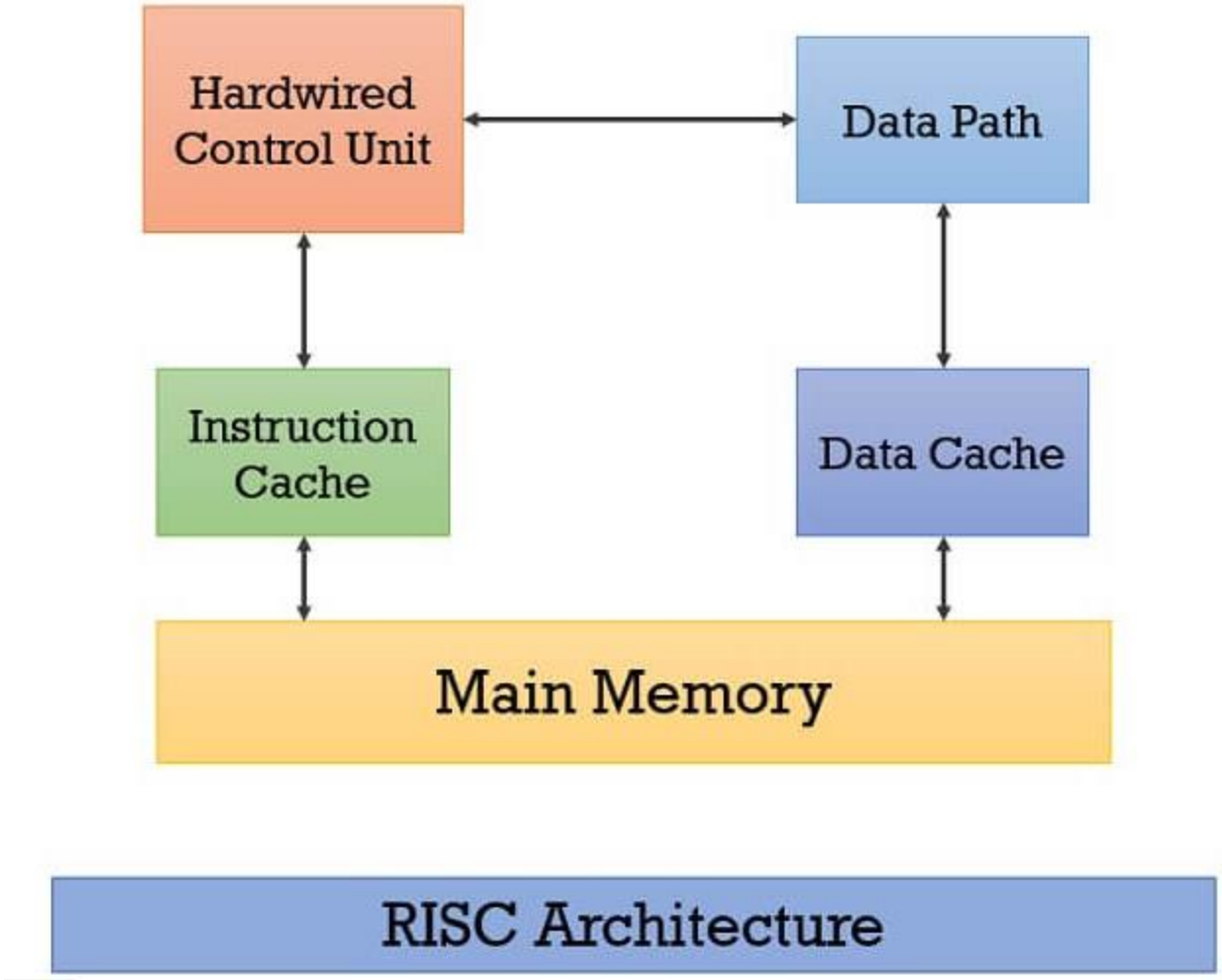
RISC	CISC
1. RISC stands for Reduced Instruction Set Computer.	1. CISC stands for Complex Instruction Set Computer.
2. RISC processors have simple instructions taking about one clock cycle. The average clock cycle per instruction (CPI) is 1.5	2. CSIC processor has complex instructions that take up multiple clocks for execution. The average clock cycle per instruction (CPI) is in the range of 2 and 15.
3. Performance is optimized with more focus on software	3. Performance is optimized with more focus on hardware.
4. It has no memory unit and uses separate hardware to implement instructions..	4. It has a memory unit to implement complex instructions.
5. It has a hard-wired unit of programming.	5. It has a microprogramming unit.
6. The instruction set is reduced i.e. it has only a few instructions in the instruction set.	**6.** The instruction set has a variety of different instructions that can be used for complex operations.
7. Multiple register sets are present	**7.** Only has a single register set
8. RISC processors are highly pipelined	8. They are normally not pipelined or less pipelined

RISC	CISC
11. The complexity of RISC lies with the compiler that executes the program - Compiler	11. The complexity lies in the microprogram - processor
12. Execution time is very less	12. Execution time is very high
13. Code expansion can be a problem	13. Code expansion is not a problem
14. The decoding of instructions is simple.	14. Decoding of instructions is complex
15. It does not require external memory for calculations	15. It requires external memory for calculations
16. The most common RISC microprocessors are Alpha, ARC, ARM, AVR, MIPS, PA-RISC, PIC, Power Architecture, and SPARC.	16. Examples of CISC processors are the System/360, VAX, PDP-11, Motorola 68000 family, AMD, and Intel x86 CPUs.
17. RISC architecture is used in high-end applications such as video processing, telecommunications, and image processing.	17. CISC architecture is used in low-end applications such as security systems, home automation, etc.


RISC
Greater Complexity
Compiler
Code Generation
Processor
CISC
Compiler
Code Generation
Processor
Greater Complexity


CISC
(Complex
instruction)
01101001000110000110101010000
10101000101010000111101011010
10010010101010100010110101111
10
10100100101
RISC
(Simple
instruction)
10101101010101011101
01010101001001011001
10101000101110101010
10101010110101101001
11100010100100100101
01010101010010101101
10101001010011101010
01101010100100100100
11001001111000101100
10010000011111001010

Introduction to RISC Instruction Sets



Two key characteristics of RISC instruction sets are:

- Each instruction fits in a single word.
- A load/store architecture is used, in which

– Memory operands are accessed only using Load and Store instructions.

– All operands involved in an arithmetic or logic operation must either be in processor registers, or one of the operands may be given explicitly within the instruction word.

RISC Instruction Sets

The instructions that have arithmetic and logic operation should have their operand either in the processor register or should be given directly in the instruction.

Like in both the instructions below we have the operands in registers

Add R2, R3
Add R2, R3, R4

The operand can be mentioned directly in the instruction as below:

Add R2, 100

At the start of execution of the program, all the operands are in memory. So, to access the memory operands, the RISC instruction set has load and store instructions.

The Load instruction loads the operand present in memory to the processor register. The load instruction is of the form:

Load destination, Source

Example Load R2, A // memory to register

The load instruction above will load the operand present at memory location A to the processor register R2.

The Store instruction stores the operand back to the memory.

Generally, the Store instruction is used to store the intermediate result or the final result in the memory. It is of the form:

Store source, destination

Example Store R2, A // register to memory

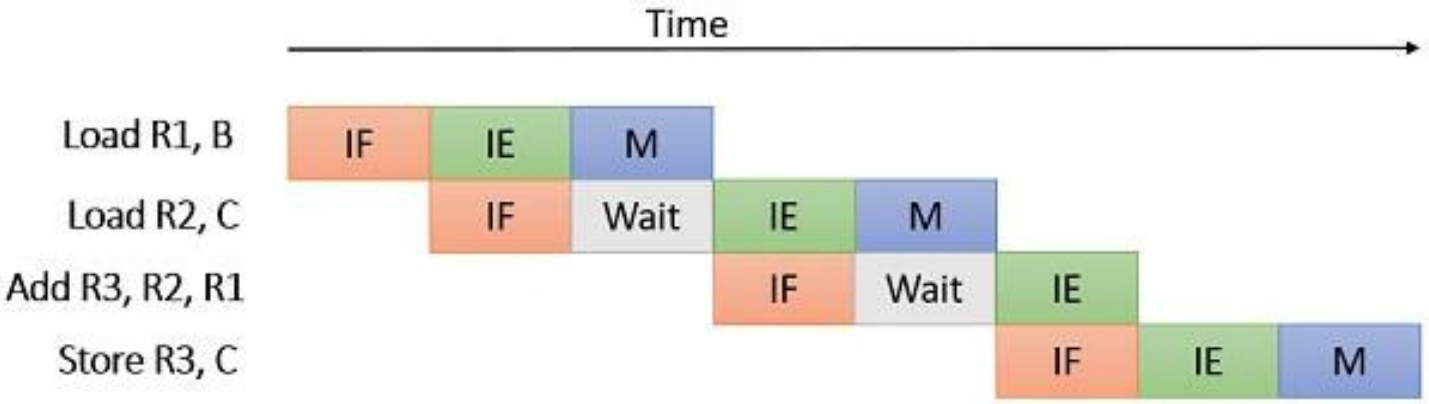
The Store instruction above will store the content in register **R2** into the **A** a memory location.

Consider the following instruction:

A = B + C

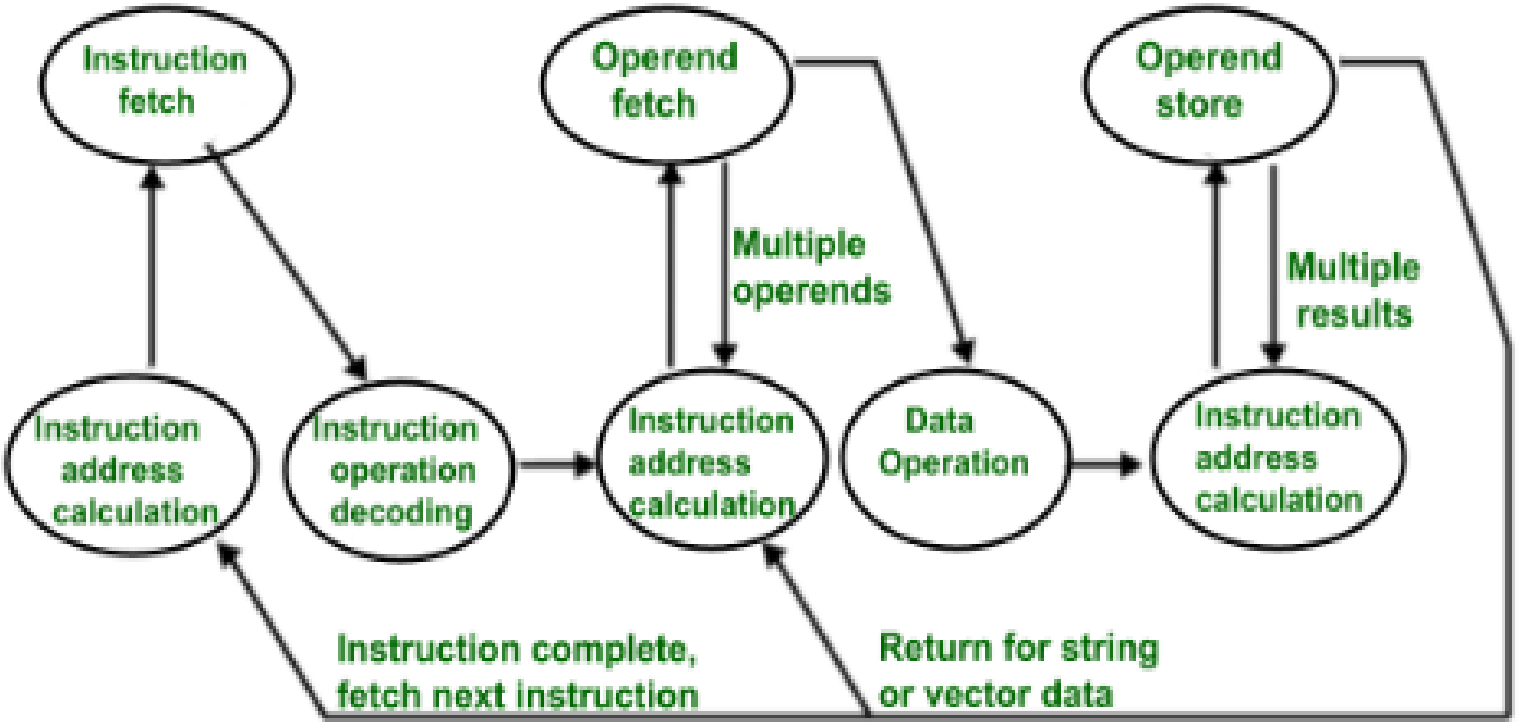
Creating a RISC instruction set for the above instruction will be.

Load R1, B
Load R2, C
Add R3, R2, R1
Store R3, C



Instruction Fetch (IF): Fetching the instruction
Instruction Execute (IE): Calculate memory address
Memory Store (M): Register to register operation or memory to memory operation

Instruction Execution and Straight-Line Sequencing



Instruction cycle state transition diagram

Instruction execution :

Instruction execution needs the following steps, which are

- PC (program counter) register of the processor gives the address of the next instruction which needs to be fetched from the memory.

- If the instruction is fetched then, the instruction opcode is decoded. On decoding, the processor identifies the number of operands.

- If there is any operand to be fetched from the memory, then that operand address is calculated.

- Operands are fetched from the memory.

- If there is more than one operand, then the operand fetching process may be repeated (i.e. address calculation and fetching operands).

- After this, the data operation is performed on the operands, and a result is generated.

- If the result has to be stored in a register, the instructions end here.

- If the destination is memory, then first the destination address has to be calculated. Then the result is then stored in the memory.

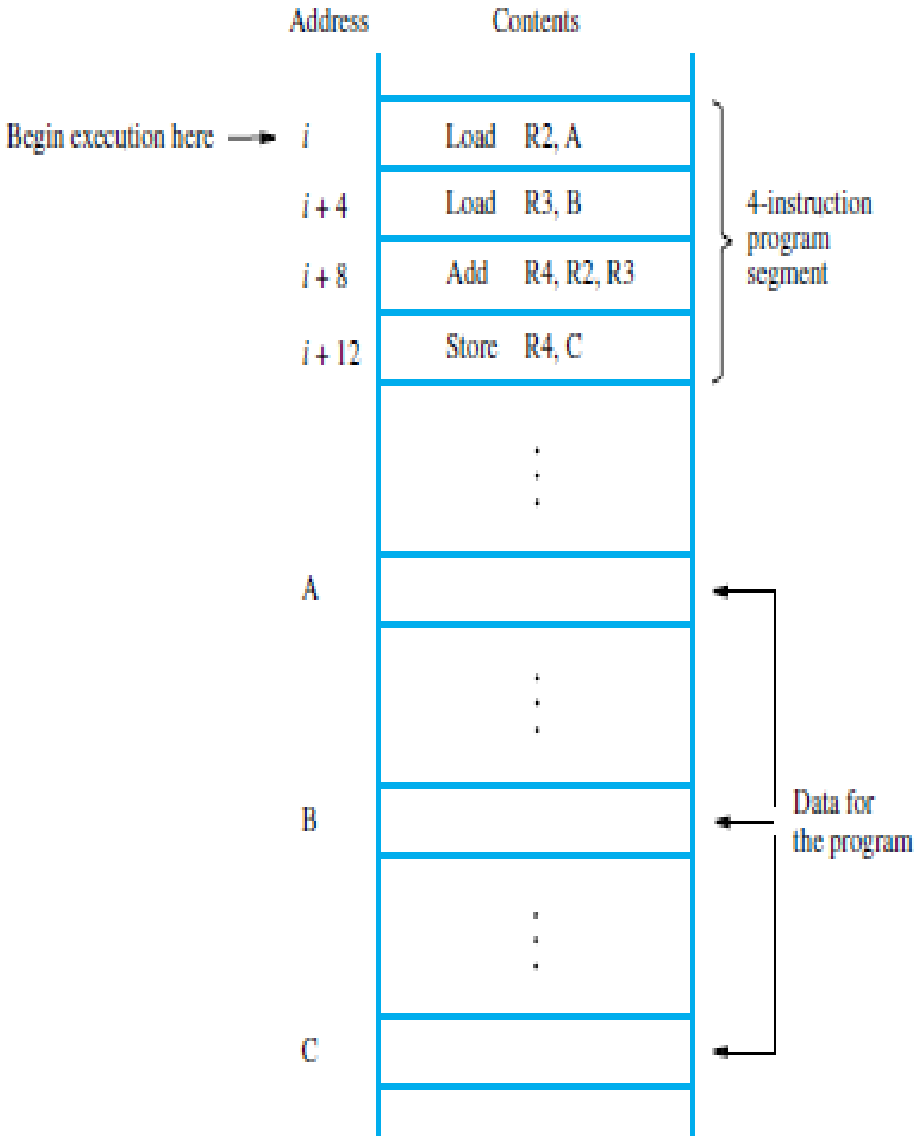
- If there are multiple results which need to be stored inside the memory, then this process may repeat (i.e. destination address calculation and store result).

- Now the current instructions have been executed.

- Side by side, the PC is incremented to calculate the address of the next instruction.

Straight line sequencing:

- Straight line sequencing means the instruction of a program is executed in a sequential manner(i.e. every time PC is incremented by a fixed offset).
- And no branch address is loaded on the PC.



A program for C ← [A] + [B].

Assembler

*	Address	Label	Mnemonics	Hexcode	Bytes	M-Cycles	T-States
√	0000		LDA 2000	3A	3	4	13
	0001			00			
	0002			20			
√	0003		MOV B,A	47	1	1	4
√	0004		LDA 2001	3A	3	4	13
	0005			01			
	0006			20			
√	0007		ADD B	80	1	1	4
√	0008		STA 2003	32	3	4	13
	0009			03			
	000A			20			
√	000B		HLT	76	1	2	5

Addressing Modes

Addressing modes:

- Programs are normally written in a high-level language, which enables the programmer to conveniently describe the operations to be performed on various data structures.

- When translating a high-level language program into assembly language, the compiler generates appropriate sequences of low-level instructions that implement the desired operations.

- The different ways for specifying the locations of instruction operands are known as *addressing modes*.

Effective Address (EA)

- Effective address is the address of the exact memory location where the value of the operand is present

- In the addressing modes that follow, the instruction does not give the operand or its address explicitly.

- Instead, it provides information from which an effective address (EA) can be derived by the processor when the instruction is executed.

- The effective address is then used to access the operand.

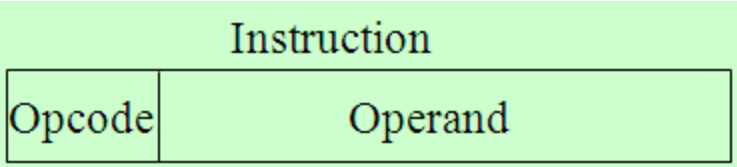
RISC-type addressing modes.

Name	Assembler syntax	Addressingfunction
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*) (LOC)	EA = [R*i*] EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Basewith index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Basewith index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*] ; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i* ; EA = [R*i*]

Addressing Modes

Immediate

- Operand is part of instruction
- Operand = address field



e.g. ADD 5
Add 5 to contents of accumulator
5 is operand

- No memory reference to fetch data
- The use of a constant in "MOV 5, R1" or

"MOV #5, R1" i.e. R1 ← 5
MOV #NUM1, R2 ; to copy the variable memory address

Direct Addressing

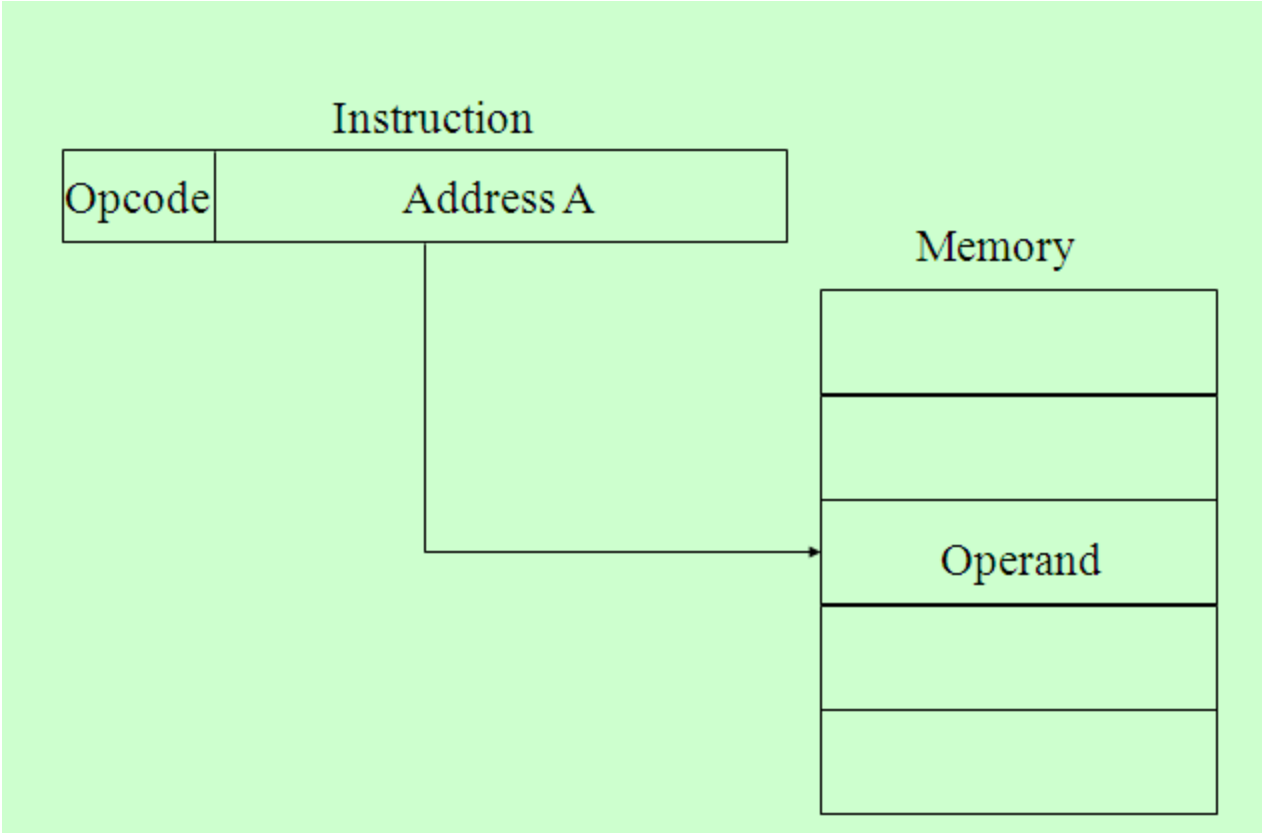
- Address field contains address of operand
- Effective address (EA) = address field (A)

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
- Use the given address to access a memory location

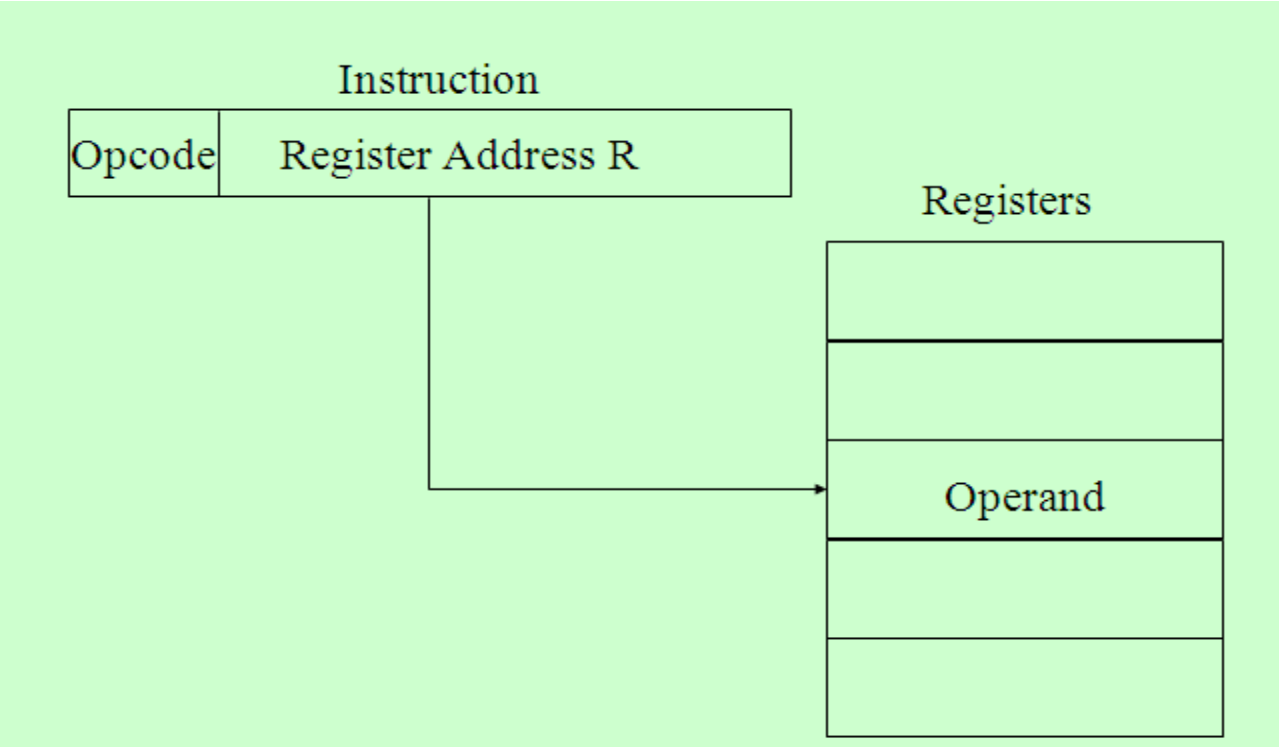
E.g. Move NUM1, R1
Move R0, SUM

Direct Addressing

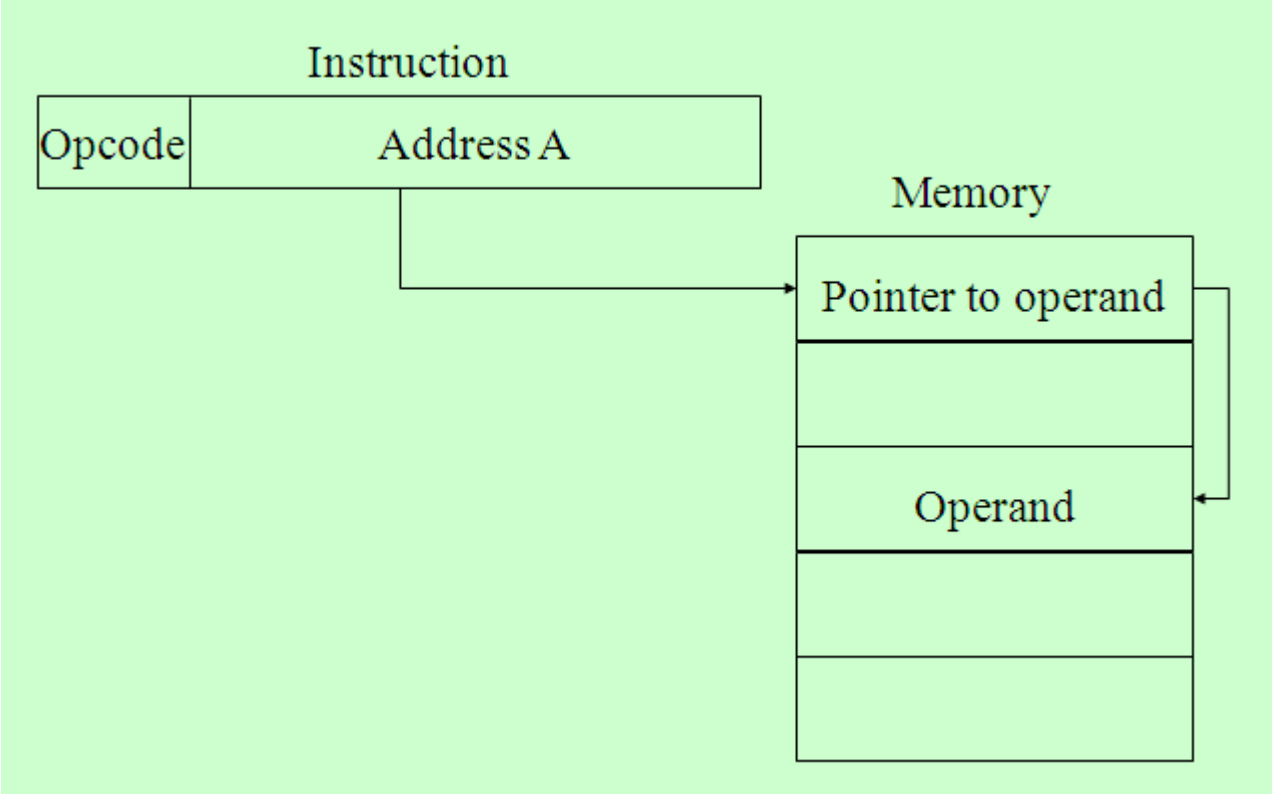


Register addressing

- Indicate which register holds the operand.
- EA = R
- Limited number of registers
- Very small address field needed



Indirect Addressing

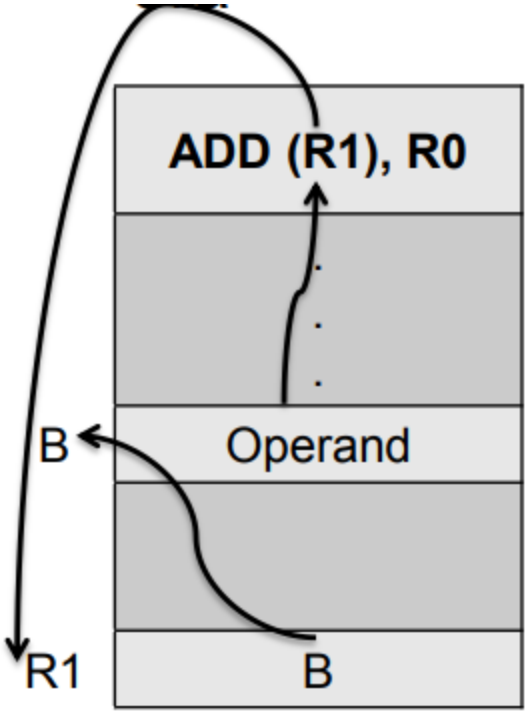


Indirect Addressing

- Indirect addressing through a general purpose register.
- Indicate the register (e.g. R1) that holds the address of the variable (e.g. B) that holds the operand.

  ADD (R1), R0

- The register or memory location that contain the address of an operand is called a pointer.

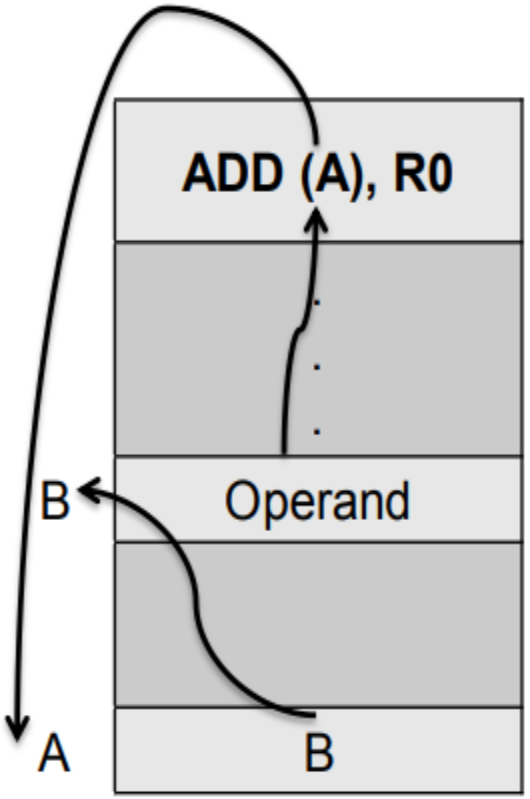


Indirect Addressing

- Indirect addressing through a memory addressing.

- Indicate the memory variable (e.g. A)that holds the address of the variable (e.g. B) that holds the operand

ADD (A), R0



Indirect Addressing

Example

Addition of N numbers

1. Move N,R1 ;	N = Numbers to add
2. Move #NUM1,R2 ;	R2= Address of 1st no.
3. Clear R0 ;	R0 = 00
4. Loop : Add (R2), R0 ;	R0 = [NUM1] + [R0]
5. Add #4, R2 ;	R2= To point to the next ; number
6. Decrement R1 ;	R1 = [R1] -1
7. Branch>0 Loop ;	Check if R1>0 or not if ; yes go to Loop
8. Move R0, SUM ;	SUM= Sum of all no.

Indexing and Arrays

- The EA of the operand is generated by adding a constant value to the contents of a register.

$$X(Ri) ; EA = X + (Ri) \quad X = \text{Signed number}$$

- X defined as offset or displacement

- Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.

$$X(Ri): EA = X + [Ri]$$

- The constant X may be given either as an explicit number or as a symbolic name representing a numerical value.

- If X is shorter than a word, sign-extension is needed

Indexing and Arrays

- In general, the Index mode facilitates access to an operand whose location is defined relative to a reference point within the data structure in which the operand appears.

- 2D Array

  (Ri, Rj) so EA = [Ri] + [Rj]
  Rj is called the base register

- 3D Array

  X(Ri, Rj) so EA = X + [Ri] + [Rj]

Indexing and Arrays

Address	Memory
	Add 20(R1), R2
	. . . .
10000H	
Offset=20 ↓	. . . .
10020H	Operand

R1	10000H

Offset is given as a Constant

Address	Memory
	Add 10000H(R1), R2
	. . . .
10000H	
Offset=20 ↓	. . . .
10020H	Operand

R1	20H

Offset is in the index register

Indexing and Arrays

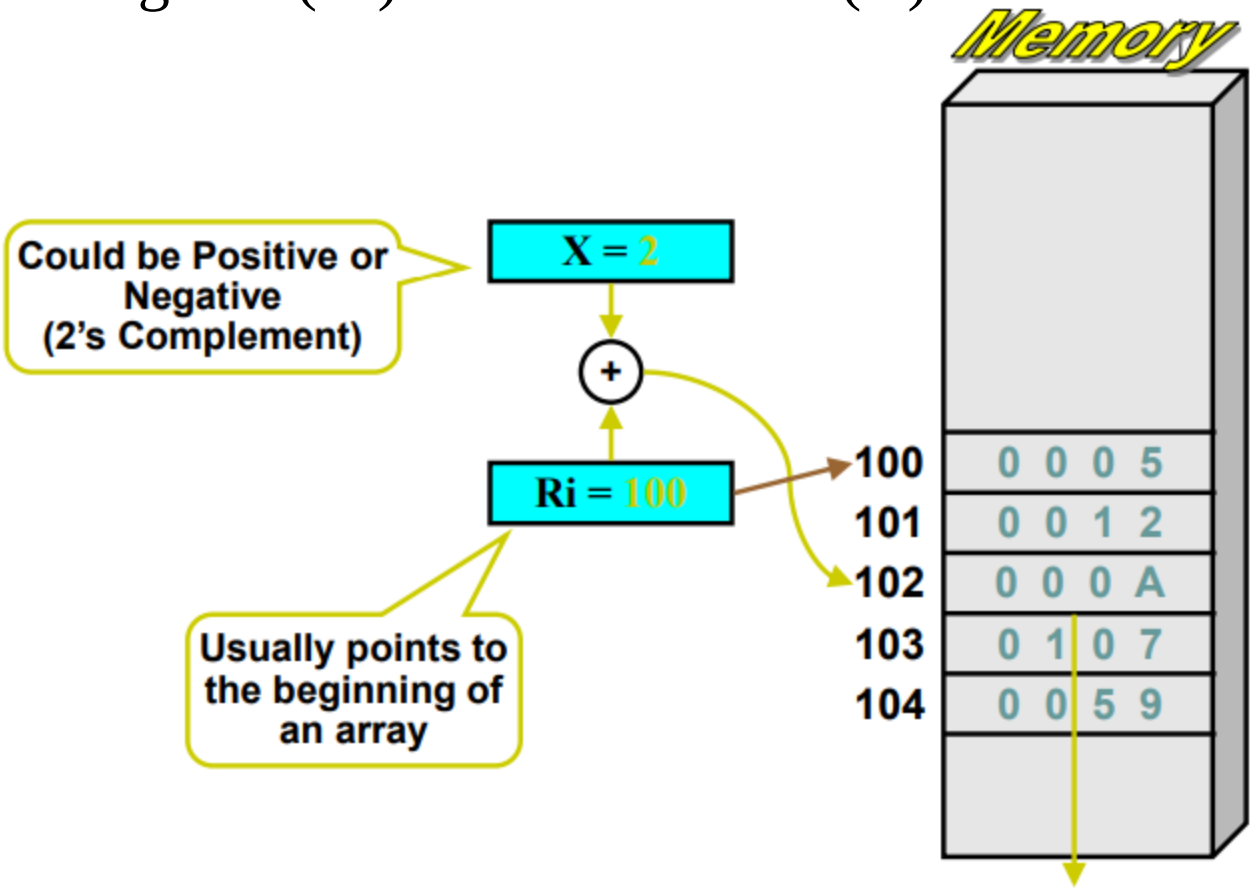
- Array
- E.g. List of students marks

Address	Memory	Comments
N	n	No. of students
LIST	Student ID1	Student 1
LIST+4	Test 1	
LIST+8	Test 2	
LIST+12	Test 3	
LIST+16	Student ID2	Student 2
LIST+20	Test 1	
LIST+24	Test 2	
LIST+28	Test 3	

- Indexed addressing used in accessing test marks from the list

Base Register

EA = Base Register (Ri) + Relative Addr (X)



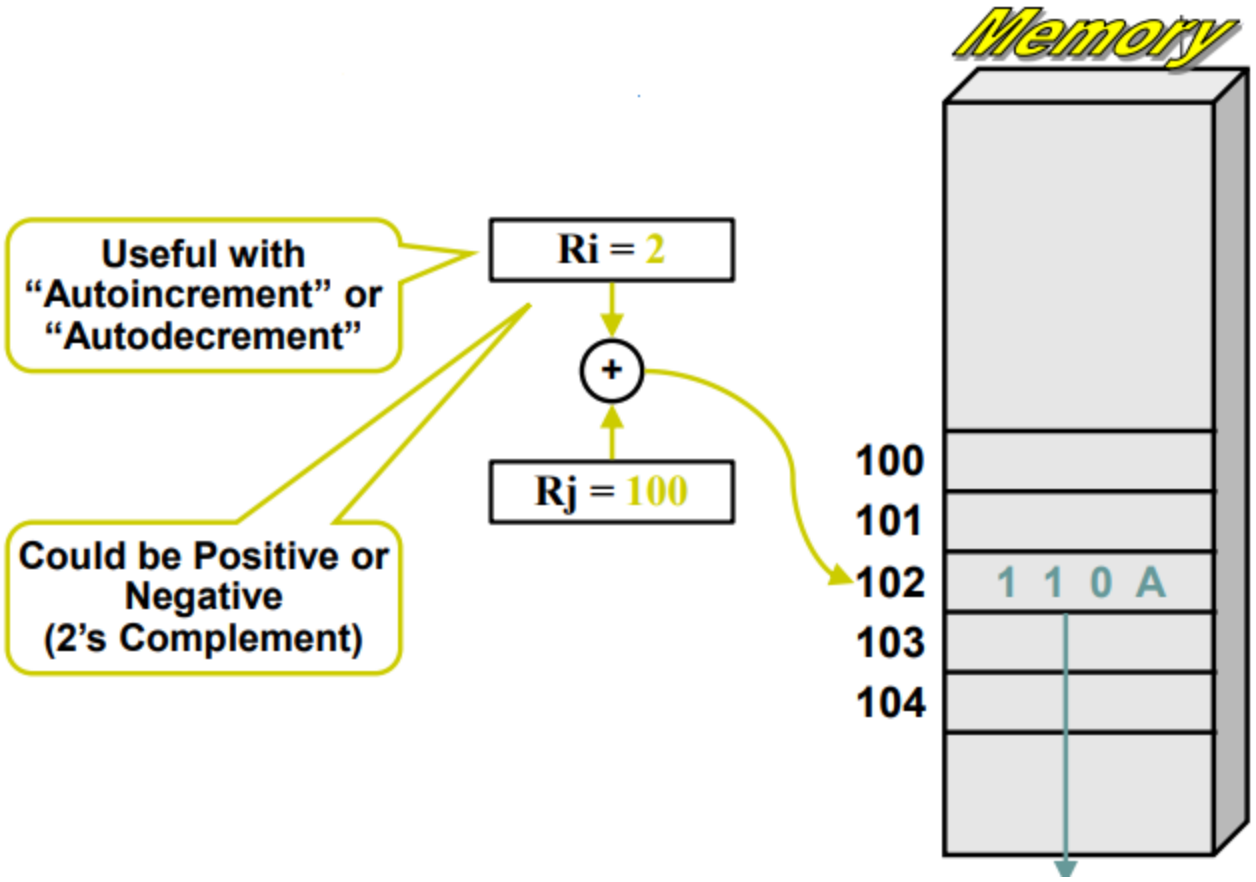
Indexing and Arrays

Program to find the sum of marks of all subjects of reach students and store it in memory.

1. Move #LIST, R0
2. Clear R1
3. Clear R2
4. Move #SUM, R2
5. Move N, R4
6. Loop : Add 4(R0), R1
7. Add 8(R0), R1
8. Add 12(R0),R1
9. Move R1, (R2)
10. Clear R1
11. Add #16, R0
12. Add #4, R2
13. Decrement R4
14. Branch>0 Loop

Indexed

EA = Index Register (Ri) + Relative Addr (Rj)



Relative Addressing

➢Relative mode – the effective address is determined by the Index mode using the program counter in place of the general-purpose register.

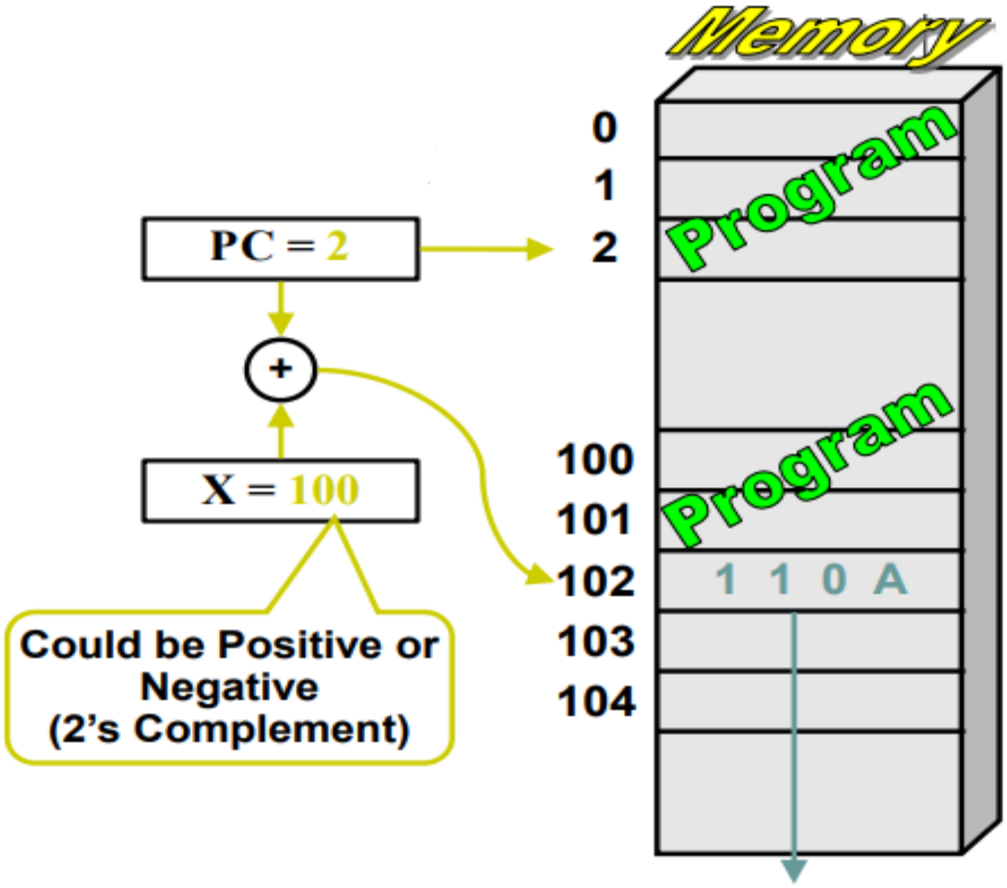
- ➢ X(PC) – note that X is a signed number
- ➢ Branch>0 LOOP

➢ This location is computed by specifying it as an offset from the current value of PC.

➢ Branch target may be either before or after the branch instruction, the offset is given as a singed num.

Relative Addressing

Relative Address EA = PC + Relative Addr (X)



Auto Increment

This addressing mode is a special case of Register Indirect Addressing Mode where-

Effective Address of the Operand = Content of Register

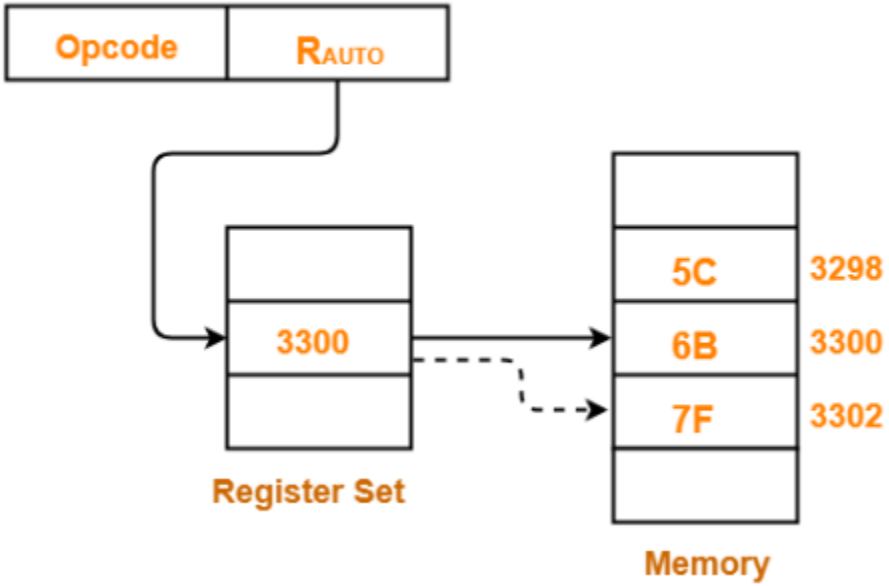
In this addressing mode,

- 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.

In auto-increment addressing mode,

- First, the operand value is fetched.
- Then, the instruction register R_{AUTO} value is incremented by step size ‘d’.

Auto Increment



Assume operand size = 2 bytes.

Here,

- After fetching the operand 6B, the instruction register R_{AUTO} will be automatically incremented by 2.
- Then, updated value of R_{AUTO} will be 3300 + 2 = 3302.
- At memory address 3302, the next operand will be found.

Auto Decrement

Effective Address of the Operand = Content of Register – Step Size

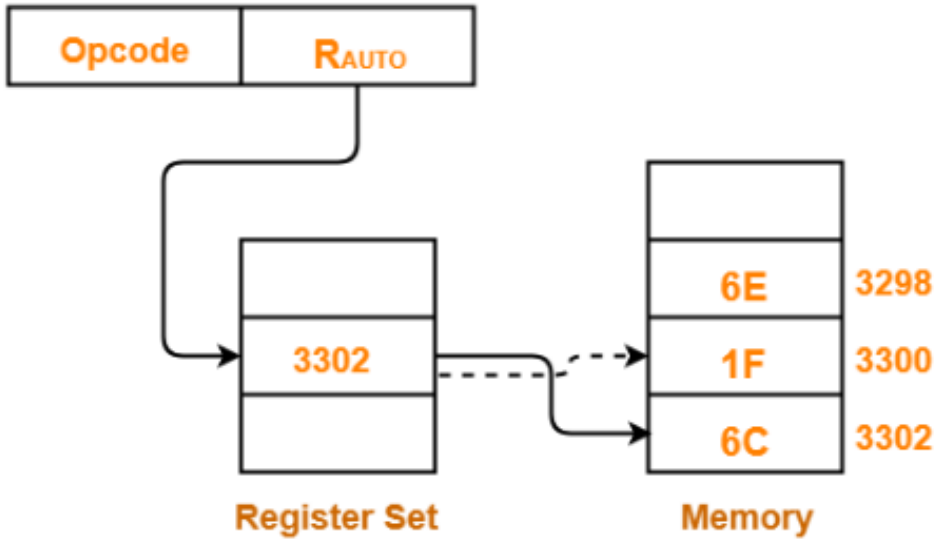
In this addressing mode,

- 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.

In auto-decrement addressing mode,

- First, the instruction register R_{AUTO} value is decremented by step size ‘d’.
- Then, the operand value is fetched.

Auto Decrement



Assume operand size = 2 bytes.

Here,

- First, the instruction register R_{AUTO} will be decremented by 2.
- Then, updated value of R_{AUTO} will be 3302 – 2 = 3300.
- At memory address 3300, the operand will be found.

Text and Reference Books

Text Books:

- Carl Hamacher, Zvonko Vranesic, Safwat Zaky: Computer Organization, 5th Edition, Tata McGraw Hill, 2002.

- Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Naraig Manjikian : Computer Organization and Embedded Systems, 6 th Edition, Tata McGraw Hill, 2012.

Reference Books:

- William Stallings: Computer Organization & Architecture, 9th Edition, Pearson, 2015.

ASSEMBLY LANGUAGE

Assembly language:

➢Machine instructions are represented by patterns of 0s and 1s. So these patterns represented by symbolic names called “mnemonics”

E.g. Load, Store, Add, Move, BR, BGTZ

➢A complete set of such symbolic names and rules for their use constitutes a programming language, referred to as an assembly language.

➢ The set of rules for using the mnemonics and for specification of complete instructions and programs is called the syntax of the language.

➢Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an assembler.

- The assembler program is one of a collection of utility programs that are a part of the system software of a computer.

- The user program in its original alphanumeric text format is called a source program, and the assembled machine-language program is called an object program.

- The assembly language for a given computer is not case sensitive.

E.g. MOVE R1, SUM

Assembler Directives

➢In addition to providing a mechanism for representing instructions in a program, assembly language allows the programmer to specify other information needed to translate the source program into the object program.

➢Assign numerical values to any names used in a program.

- For e,g, name TWENTY is used to represent the value 20. This fact may be conveyed to the assembler program through an equate statement such as TWENTY EQU 20

➢ If the assembler is to produce an object program according to this arrangement, it has to know

- How to interpret the names
- Where to place the instructions in the memory
- Where to place the data operands in the memory

Assembly language representation for the program

Label: Operation Operand(s) Comment

	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

Assembly language representation for the program

Assembly and Execution of Programs

- A source program written in an assembly language must be assembled into a machine language object program before it can be executed.

- This is done by the assembler program, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.

- A key part of the assembly process is determining the values that replace the names. Assembler keep track of Symbolic name and Label name, create table called symbol table.

- The symbol table created by scan the source program twice.

- A branch instruction is usually implemented in machine code by specifying the branch target as the distance (in bytes) from the present address in the Program Counter to the target instruction.

- The assembler computes this branch offset, which can be positive or negative, and puts it into the machine instruction.

- The assembler stores the object program on the secondary storage device available in the computer, usually a magnetic disk.

- The object program must be loaded into the main memory before it is executed. For this to happen, another utility program called a loader must already be in the memory.

- Executing the loader performs a sequence of input operations needed to transfer the machine-language program from the disk into a specified place in the memory.

- The loader must know the length of the program and the address in the memory where it will be stored.

- The assembler usually places this information in a header preceding the object code (Like start/end offset address).

- When the object program begins executing, it proceeds to completion unless there are logical errors in the program.

- The user must be able to find errors easily. The assembler can only detect and report syntax errors.

- To help the user find other programming errors, the system software usually includes a debugger program.

- This program enables the user to stop execution of the object program at some points of interest and to examine the contents of various processor registers and memory locations.

Number Notation

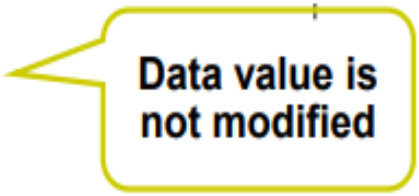
Decimal Number
ADD #93,R1

Binary Number
ADD #%0101110,R1

Hexadecimal Number
ADD #$5D,R1

Types of Instructions
Data Transfer Instructions

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



Data Transfer Instructions

Mode	Assembly	Register Transfer
Direct address	LD ADR	*AC* ← *M*[*ADR*]
Indirect address	LD @ADR	*AC* ← *M*[*M*[*ADR*]]
Relative address	LD $ADR	*AC* ← *M*[*PC*+*ADR*]
Immediate operand	LD #NBR	*AC* ← NBR
Index addressing	LD ADR(X)	*AC* ← *M*[*ADR*+*XR*]
Register	LD R1	*AC* ← *R1*
Register indirect	LD (R1)	*AC* ← *M*[*R1*]
Autoincrement	LD (R1)+	*AC* ← *M*[*R1*], *R1* ← *R1*+1

Data Manipulation Instructions

- Arithmetic
- Logical & Bit Manipulation
- Shift

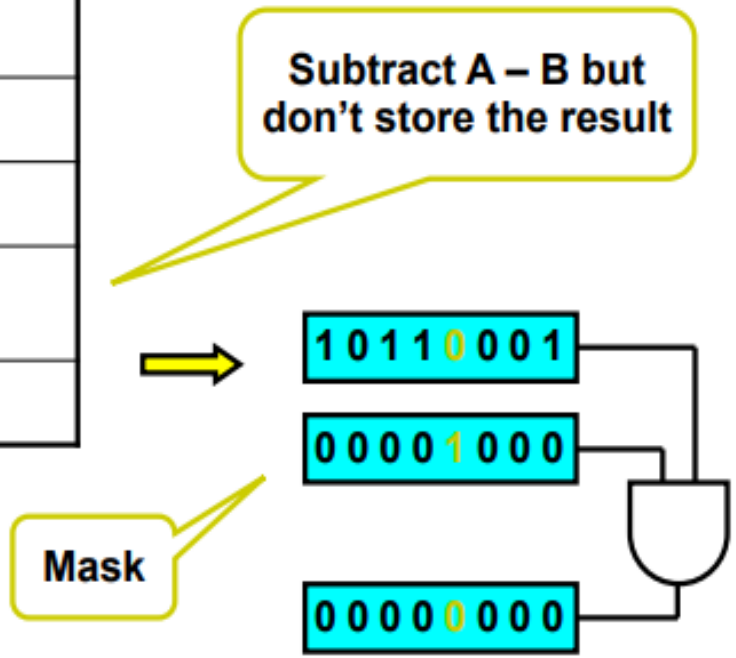
Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

Program Control Instructions

Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (Subtract)	CMP
Test (AND)	TST



Conditional Branch Instructions

Mnemonic	Branch Condition	Tested Condition
BZ	Branch if zero	Z = 1
BNZ	Branch if not zero	Z = 0
BC	Branch if carry	C = 1
BNC	Branch if no carry	C = 0
BP	Branch if plus	S = 0
BM	Branch if minus	S = 1
BV	Branch if overflow	V = 1
BNV	Branch if no overflow	V = 0

STACK

Stacks

- A stack is a list of data elements, usually words, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. The structure is sometimes referred to as a pushdown stack.

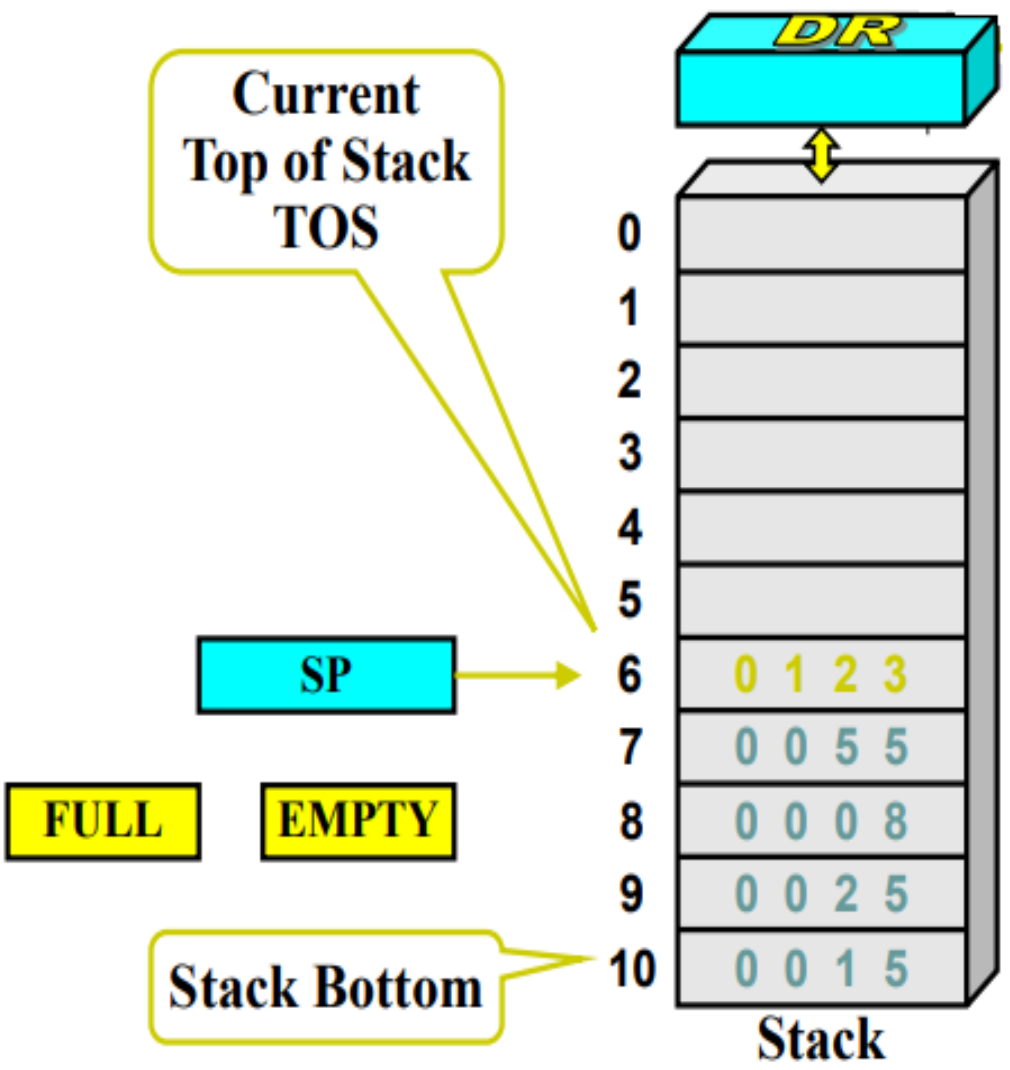
- Last-in–first-out (LIFO) stack working.

- The terms push and pop are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

- The stack pointer, SP, is used to keep track of the address of the element of the stack that is at the top at any given time.

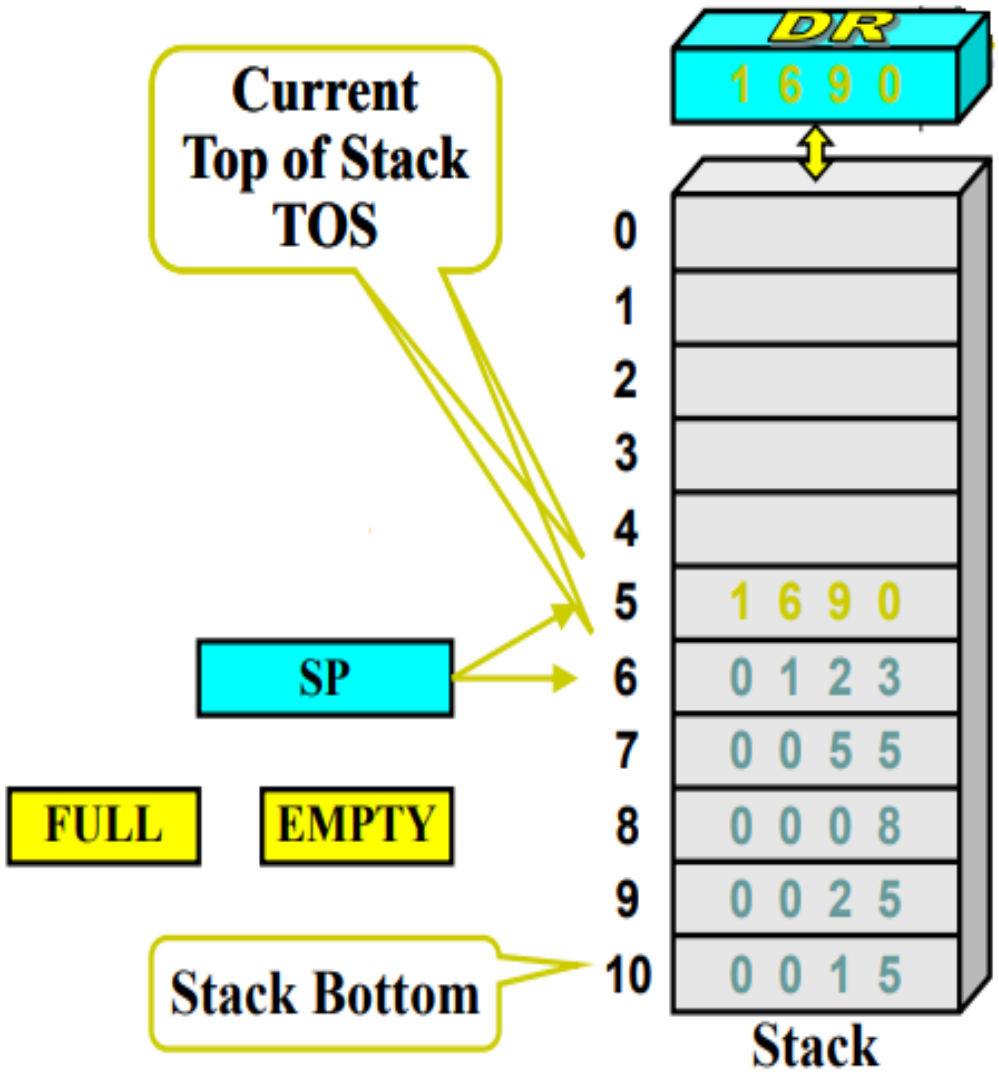
Stack Organization

LIFO
Last In First Out



Stack Organization

PUSH
SP ← SP – 1
M[SP] ← DR
If (SP = 0) then (FULI
EMPTY ← 0



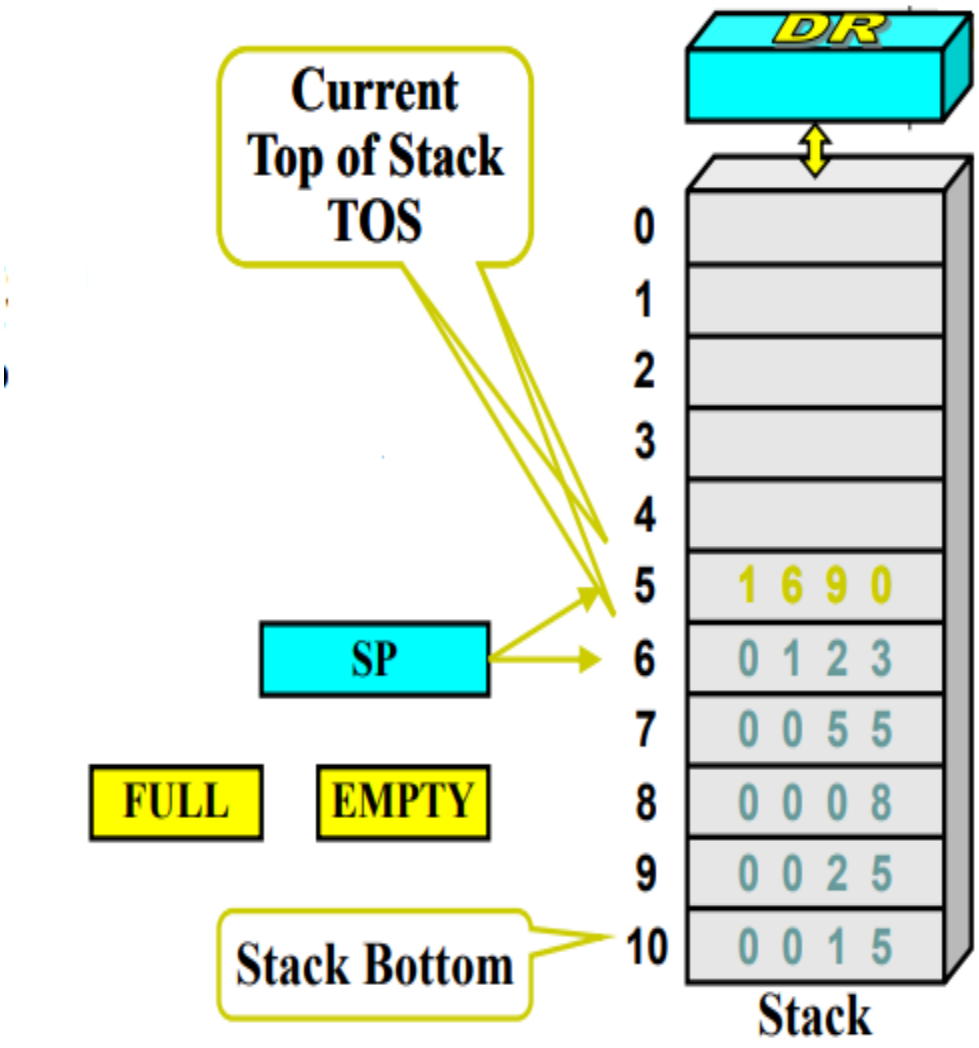
POP

DR ← M[SP]

SP ← SP + 1

If (SP = 11) then (EMPTY ← 1)

FULL ← 0



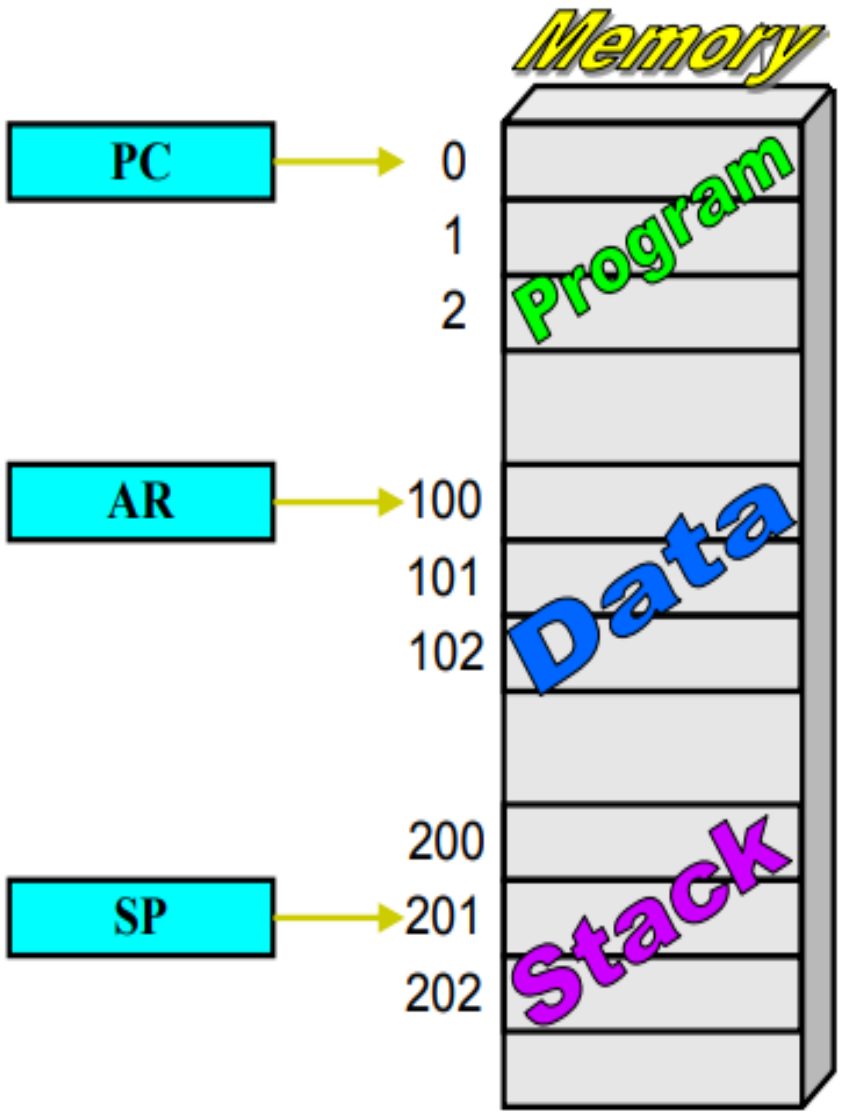
Memory Stack

PUSH SP ← SP – 1

M[SP] ← DR

POP DR ← M[SP]

SP ← SP + 1



SUBROUTINES

Subroutines

- In a given program, it is often necessary to perform a particular task many times on different data values.

- It is prudent to implement this task as a block of instructions that is executed each time the task has to be performed. Such a block of instructions is usually called a subroutine.

- However, to save space, only one copy of this block is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location.

- When a program branches to a subroutine we say that it is calling the subroutine.

- The instruction that performs this branch operation is named a Call instruction.

- After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine.

- The subroutine is said to return to the program that called it, and it does so by executing a Return instruction.

- Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location.

- The location where the calling program resumes execution is the location pointed to by the updated program counter (PC) while the Call instruction is being executed.

- Hence, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program.

- The way in which a computer makes it possible to call and return from subroutines is referred to as its subroutine linkage method.

- The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function.

- Such a register is called the link register. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

Subroutines

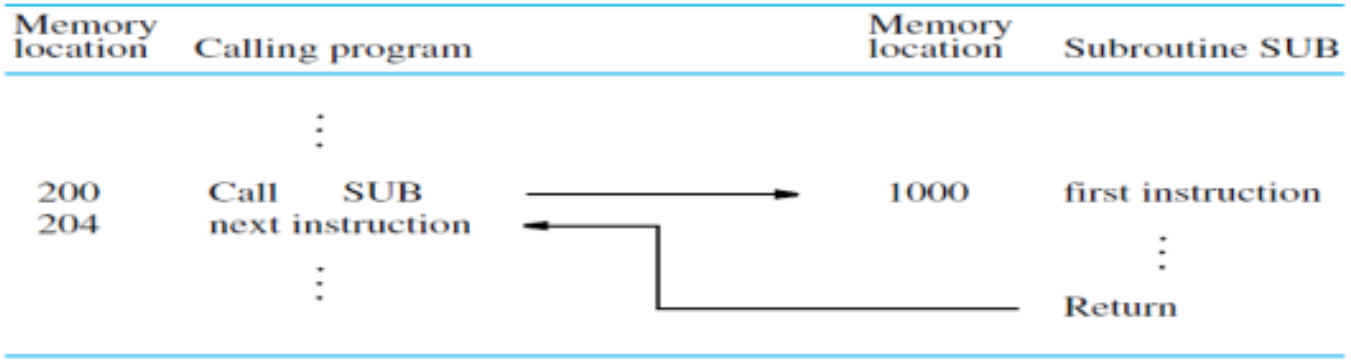
➢The Call instruction is just a special branch instruction that performs the following operations:

- Store the contents of the PC in the link register
- Branch to the target address specified by the Call instruction

➢ The Return instruction is a special branch instruction that performs the operation

- Branch to the address contained in the link register

Subroutines



Subroutine linkage using a link register.

Subroutine Nesting and the Processor Stack

- A common programming practice, called subroutine nesting, is to have one subroutine call another.

- In this case, the return address of the second call is also stored in the link register, overwriting its previous contents.

- Hence, it is essential to save the contents of the link register in some other location before calling another subroutine. Otherwise, the return address of the first subroutine will be lost.

- That is, return addresses are generated and used in a last-in–first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto the processor stack.

Parameter Passing

- When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation.

- Later, the subroutine returns other parameters, which are the results of the computation.

- This exchange of information between a calling program and a subroutine is referred to as parameter passing.

- Parameter passing may be accomplished in several ways. The parameters may be placed in registers, in memory locations, or on the processor stack where they can be accessed by the subroutine.

Program of subroutine
Parameters passed through registers.

- Calling Program

1. Move N, R1
2. Move #NUM1,R2
3. Call LISTADD
4. Move R0,SUM

- Subroutine

1. LISTADD: Clear R0
2. LOOP: Add (R2)+,R0
3. Decrement R1
4. Branch>0 LOOP
5. Return

Parameter Passing by Value and by Reference

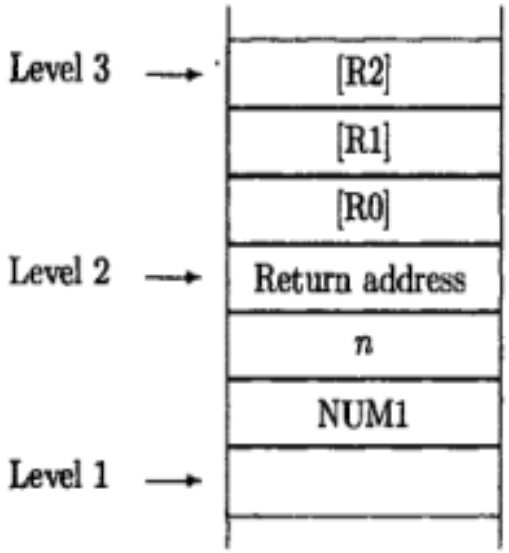
- Instead of passing the actual Value(s), the calling program passes the address of the Value(s). This technique is called passing by reference.

- The second parameter is passed by value, that is, the actual number of entries, is passed to the subroutine.

Program of subroutine
Parameters passed on the stack.

Assume top of stack is at level 1 below.

```
                Move        #NUM1,-(SP)     Push parameters onto stack.
                Move        N,-(SP)
                Call        LISTADD         Call subroutine
                                              (top of stack at level 2).
                Move        4(SP),SUM       Save result.
                Add         #8,SP           Restore top of stack
                                              (top of stack at level 1).
                :
LISTADD         MoveMultiple R0-R2,-(SP)    Save registers
                                              (top of stack at level 3).
                Move        16(SP),R1       Initialize counter to n.
                Move        20(SP),R2       Initialize pointer to the list.
                Clear       R0              Initialize sum to 0.
LOOP            Add         (R2)+,R0        Add entry from list.
                Decrement   R1
                Branch>0    LOOP
                Move        R0,20(SP)       Put result on the stack.
                MoveMultiple (SP)+,R0-R2    Restore registers.
                Return                      Return to calling program.
```



The Stack Frame

- If the subroutine requires more space for local memory variables, the space for these variables can also be allocated on the stack this area of stack is called Stack Frame.

- For e.g. during execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine.

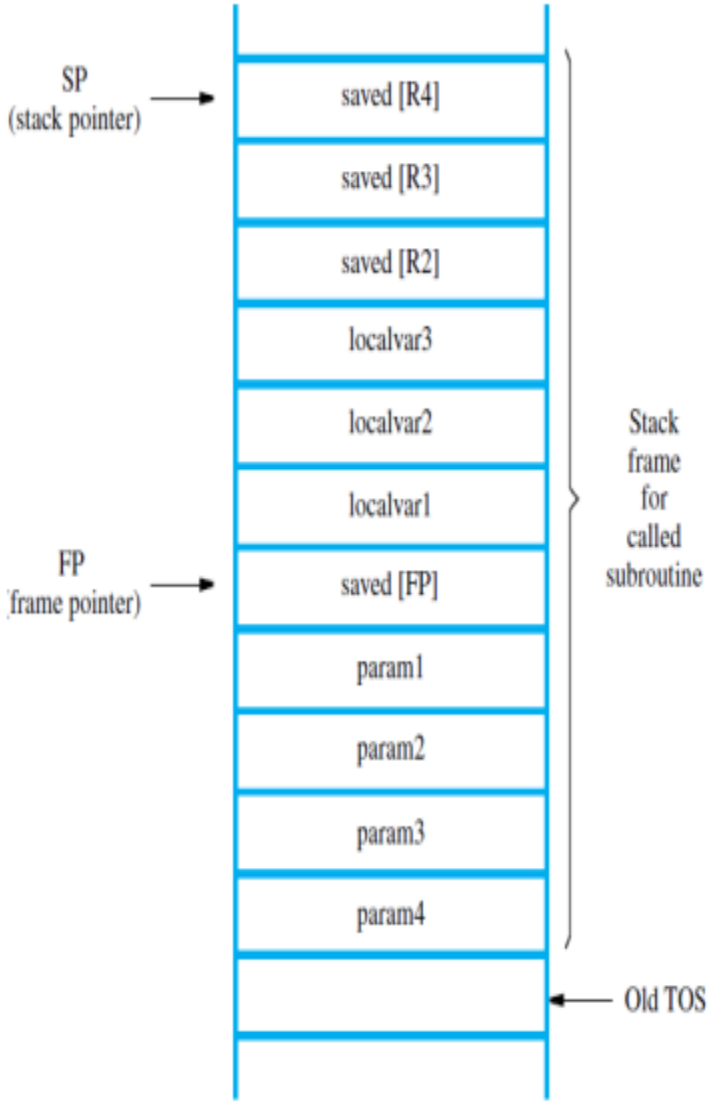
- These locations constitute a private work space for the subroutine, allocated at the time the subroutine is entered and deallocated when the subroutine returns control to the calling program.

The Stack Frame

- Frame pointer (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine .

- In the figure, we assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R 2 , R 3 , and R 4 need to be saved because they will also be used within the subroutine .

- When nested subroutines are used, the stack frame of the calling subroutine would also include the return address.



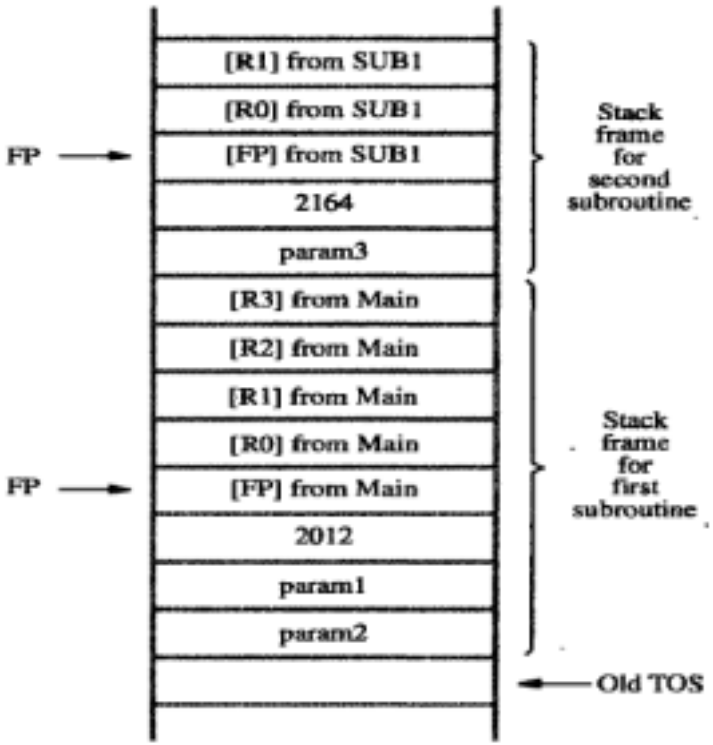
Stack Frames for Nested Subroutines

Main program

```
                :
2000    Move    PARAM2,-(SP)    Place parameters on stack.
2004    Move    PARAM1,-(SP)
2008    Call    SUB1
2012    Move    (SP),RESULT     Store result.
2016    Add     #8,SP           Restore stack level.
2020    next instruction
```

First subroutine

```
2100 SUB1  Move          FP,-(SP)      Save frame pointer register.
2104       Move          SP,FP         Load the frame pointer.
2108       MoveMultiple  R0-R3,-(SP)   Save registers.
2112       Move          8(FP),R0      Get first parameter.
           Move          12(FP),R1     Get second parameter.
                :
           Move          PARAM3,-(SP)  Place a parameter on stack.
2160       Call          SUB2
2164       Move          (SP)+,R2      Pop SUB2 result into R2.
                :
           Move          R3,8(FP)      Place answer on stack.
           MoveMultiple  (SP)+,R0-R3   Restore registers.
           Move          (SP)+,FP      Restore frame pointer register.
           Return                      Return to Main program.
```



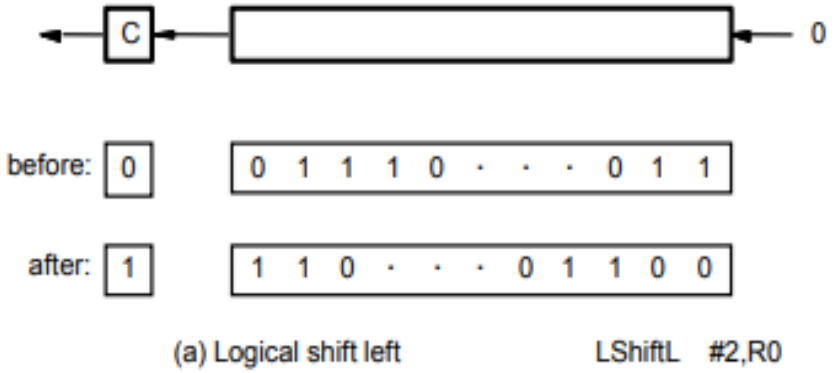
Second subroutine

```
3000 SUB2  Move          FP,-(SP)      Save frame pointer register.
           Move          SP,FP         Load the frame pointer.
           MoveMultiple  R0-R1,-(SP)   Save registers R0 and R1.
           Move          8(FP),R0      Get the parameter.
                :
           Move          R1,8(FP)      Place SUB2 result on stack.
           MoveMultiple  (SP)+,R0-R1   Restore registers R0 and R1.
           Move          (SP)+,FP      Restore frame pointer register.
           Return                      Return to Subroutine 1.
```

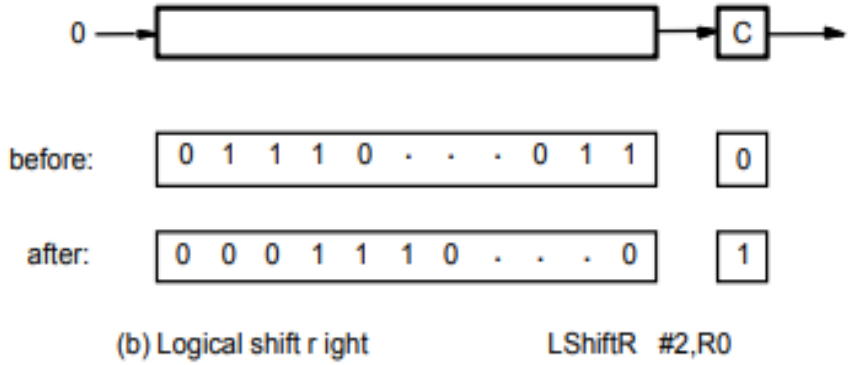
ADDITIONAL INSTRUCTIONS

Logical Shifts

Logical shift – shifting left (LShiftL) and shifting right (LShiftR)



(a) Logical shift left LShiftL #2,R0

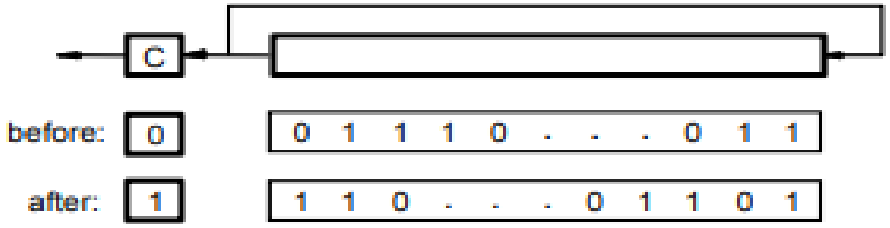


(b) Logical shift r ight LShiftR #2,R0

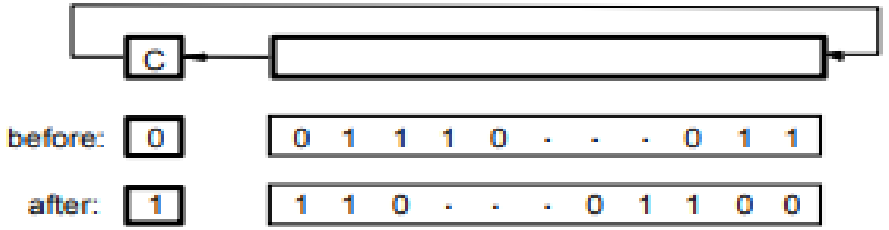
Arithmetic Shifts

(c) Arithmetic shift right AShiftR #2,R0

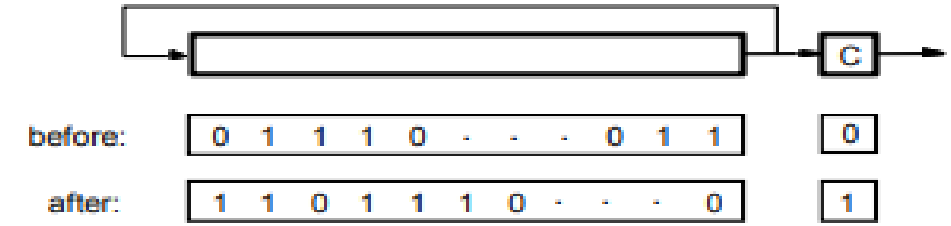
Rotate



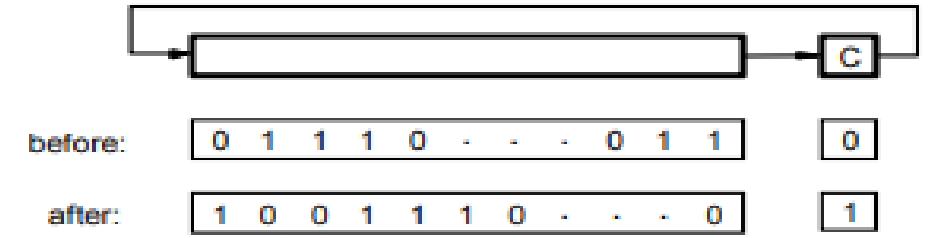
(a) Rotate left without carr y RotateL #2,R0



(b) Rotate left with carr y RotateLC #2,R0



(c) Rotate r ight without carr y RotateR #2,R0



(d) Rotate right with carr y RotateRC #2,R0

Multiplication and Division

- Not very popular (especially division)
- Multiply R_i, R_j
  $R_j \leftarrow [R_i] \times [R_j]$
- 2n-bit product case: high-order half in R(j+1)
- Divide R_i, R_j
  $R_j \leftarrow [R_i] / [R_j]$
  Quotient is in Rj, remainder may be placed in R(j+1)

Logic Instructions

- And R2, R3, R4
- And #Value, R4, R2
- And #$0FF, R2, R2,

ENCODING OF MACHINE INSTRUCTIONS

Encoding of Machine Instructions

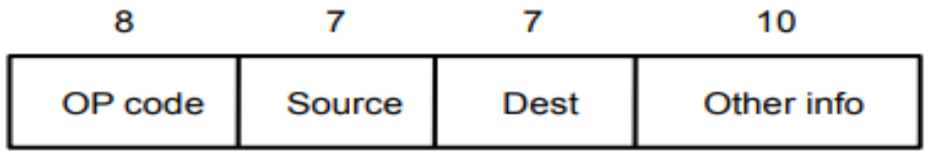
- Assembly language program needs to be converted into machine instructions. (ADD = 0100 in ARM instruction set)

- In the previous section, an assumption was made that all instructions are one word in length.

- OP code: the type of operation to be performed and the type of operands used may be specified using an encoded binary pattern

- Suppose 32-bit word length, 8-bit OP code (how many instructions can we have?), 16 registers in total (how many bits?), 3-bit addressing mode indicator.

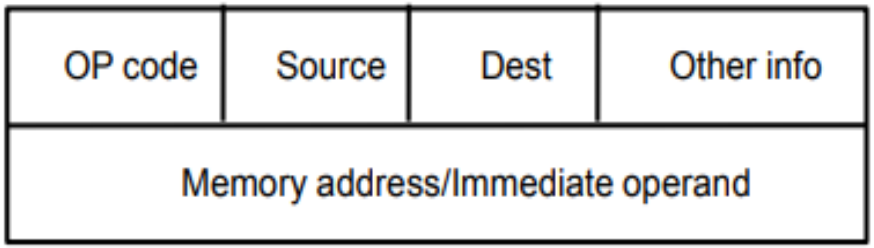
Add R1, R2
Move 24(R0), R5
LshiftR #2, R0
Move #$3A, R1



One-word instruction

Encoding of Machine Instructions

- What happens if we want to specify a memory operand using the Absolute addressing mode?
- Move R2, LOC
- 14-bit for LOC – insufficient
- Solution – use two words



Two-word instruction

Encoding of Machine Instructions

- Then what if an instruction in which two operands can be specified using the Absolute addressing mode?

- Move LOC1, LOC2

- Solution – use two additional words

- This approach results in instructions of variable length.

- Complex instructions can be implemented, closely resembling operations in high-level programming languages – Complex Instruction Set Computer (CISC)

Encoding of Machine Instructions

- If we insist that all instructions must fit into a single 32-bit word, it is not possible to provide a 32-bit address or a 32-bit immediate operand within the instruction.

- It is still possible to define a highly functional instruction set, which makes extensive use of the processor registers.

Add R1, R2 ----- yes
Add LOC, R2 ----- no
Add (R3), R2 ----- yes

CISC Instruction Sets

- Instructions in modern CISC processors typically do not use a three-address format.

- Most arithmetic and logic instructions use the two-address format

  Operation destination, source

- An Add instruction of this type is

  Add B, A

  which performs the operation B ← [A] + [B] on memory operands

- In some CISC processors one operand may be in the memory but the other must be in a register. In this case, the instruction sequence for the required task would be

  Move Ri, A
  Add Ri, B
  Move C, Ri

Additional Addressing Modes
Autoincrement and Autodecrement Modes

Autoincrement mode—The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically incremented to point to the next operand in memory.

$$(Ri)+$$

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

$$-(Ri)$$

Relative mode—The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.

Condition Codes

➢Operations performed by the processor typically generate results such as numbers that are positive, negative, or zero.

➢ The processor can maintain the information about these results for use by subsequent conditional branch instructions.

➢ This is accomplished by recording the required information in individual bits, often called condition code flags.

➢ These flags are usually grouped together in a special processor register called the condition code register or status register.

➢ Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed.

Four commonly used flags are

- N (negative) Set to 1 if the result is negative; otherwise, cleared to 0
- Z (zero) Set to 1 if the result is 0; otherwise, cleared to 0
- V (overflow) Set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
- C (carry) Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

If condition codes are used, then the Subtract instruction would cause both N and Z flags to be cleared to 0 if the contents of register R2 are still greater than 0. The desired branching could be specified simply as

Branch>*0 LOOP*

without indicating the register involved in the test. This instruction causes a branch if neither N nor Z is 1, that is, if the result produced by the Subtract instruction is neither negative nor equal to zero.

Many conditional branch instructions are provided in the instruction set of a computer to enable a variety of conditions to be tested. The conditions are defined as logic expressions involving the condition code flags.

Text and Reference Books

Text Books:

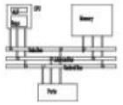
- Carl Hamacher, Zvonko Vranesic, Safwat Zaky: Computer Organization, 5th Edition, Tata McGraw Hill, 2002.

- Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Naraig Manjikian : Computer Organization and Embedded Systems, 6 th Edition, Tata McGraw Hill, 2012.

Reference Books:

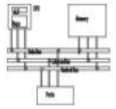
- William Stallings: Computer Organization & Architecture, 9th Edition, Pearson, 2015.

Intel IA-32 Family



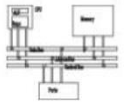
- Intel386 (1985)
  - 4 GB addressable RAM
  - 32-bit registers
  - paging (virtual memory)
  - Up to 33MHz
- Intel486 (1989)
  - instruction pipelining
  - Integrated FPU
  - 8K cache
- Pentium (1993)
  - Superscalar (two parallel pipelines)

IA32 Processors



- Totally Dominate Computer Market
- Evolutionary Design
  - Starting in 1978 with 8086
  - Added more features as time goes on
  - Still support old features, although obsolete
- Complex Instruction Set Computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!

General-purpose registers



32-bit General-Purpose Registers

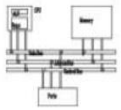
EAX	EBP
EBX	ESP
ECX	ESI
EDX	EDI

EFLAGS
EIP

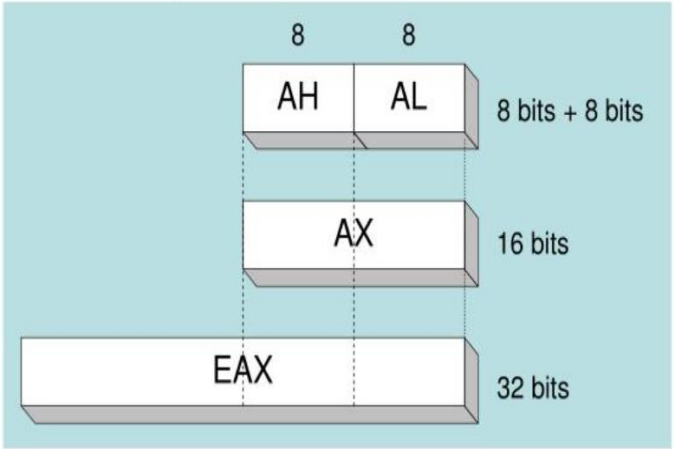
16-bit Segment Registers

CS	ES
SS	FS
DS	GS

Accessing parts of registers



- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to EAX, EBX, ECX, and EDX



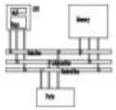
32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Index and base registers

- Some registers have only a 16-bit name fo
  their lower half (no 8-bit aliases). The 16-
  registers are usually used only in real-addr
  mode.

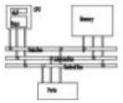
32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Some specialized register uses (1 of 2)



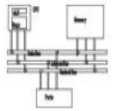
- General-Purpose
  - EAX – accumulator (automatically used by division and multiplication)
  - ECX – loop counter
  - ESP – stack pointer (should never be used for arithmetic or data transfer)
  - ESI, EDI – index registers (used for high-speed memory transfer instructions)
  - EBP – extended frame pointer (stack)

Status flags



- Carry
  - unsigned arithmetic out of range
- Overflow
  - signed arithmetic out of range
- Sign
  - result is negative
- Zero
  - result is zero
- Auxiliary Carry
  - carry from bit 3 to bit 4
- Parity
  - sum of 1 bits is an even number

Programmer's model



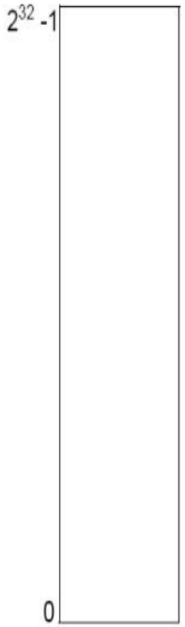
Basic Program Execution Registers

Eight 32-bit Registers	General-Purpose Registers
Six 16-bit Registers	Segment Registers
32-bits	EFLAGS Register
32-bits	EIP (Instruction Pointer Register)

FPU Registers

Eight 80-bit Registers	Floating-Point Data Registers
16 bits	Control Register
16 bits	Status Register
16 bits	Tag Register

Address Space*



*The address space can be flat or segmented. Using the physical address extension mechanism, a physical address space of 2^{36} – 1 can be addressed.

Intel IA-32 Architecture

Memory Organization

- IA-32 architecture, memory is byte-addressable using 32-bit addresses, and instructions typically operate on data operands of 8 or 32 bits.

- Operand sizes are called byte and doubleword in Intel terminology.

- 16-bit operand was called a word in earlier 16-bit Intel processors

- A larger 64-bit operand size called a quadword for double-precision floating-point numbers and packed integer data.

- Multiple-byte data operands may start at any byte address location.

- They need not be aligned with any particular address boundaries in the memory.

Register Structure

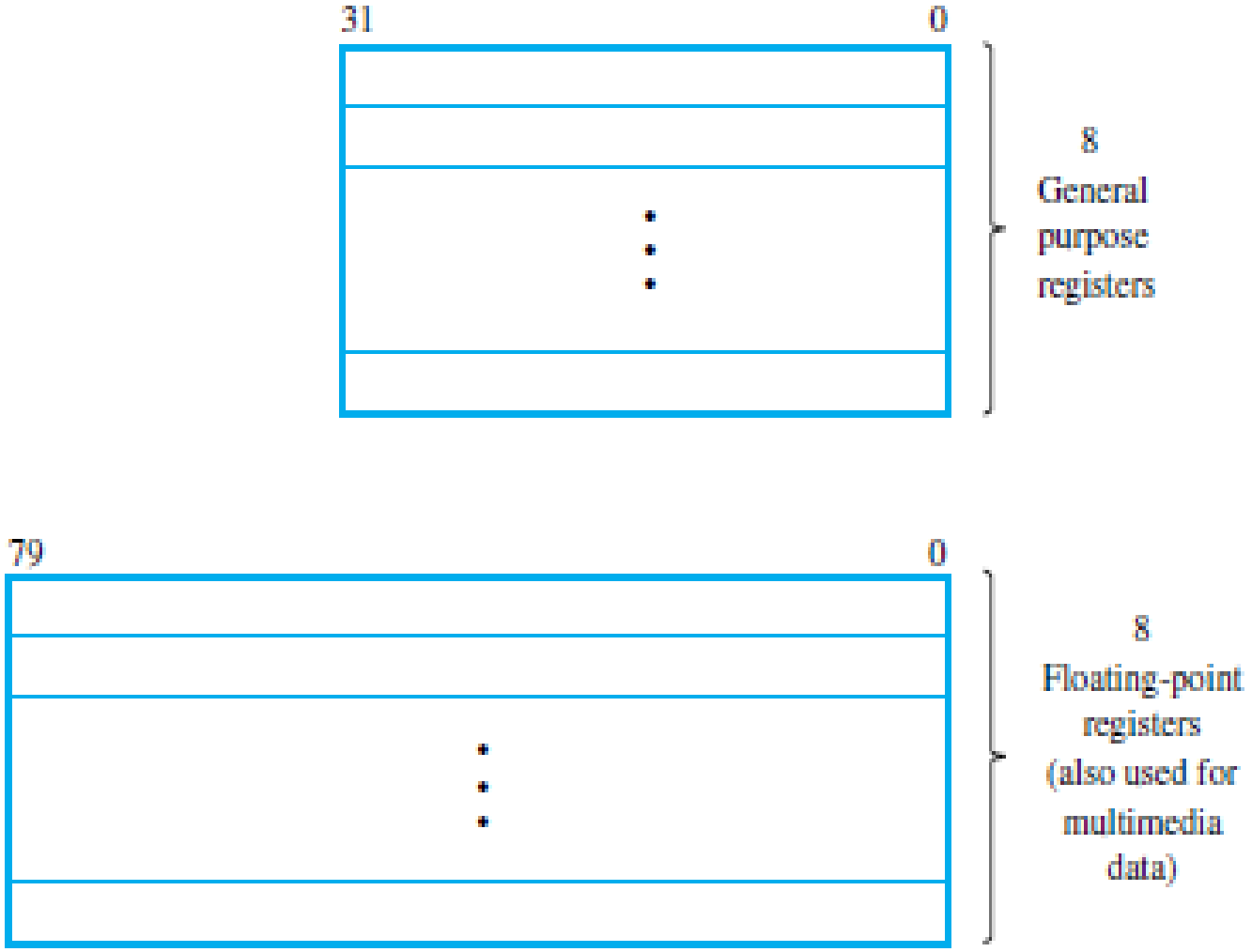
- IA-32 architecture has different models for accessing the memory.

- Segmented memory model associates different areas of the memory, called segments, with different usages.

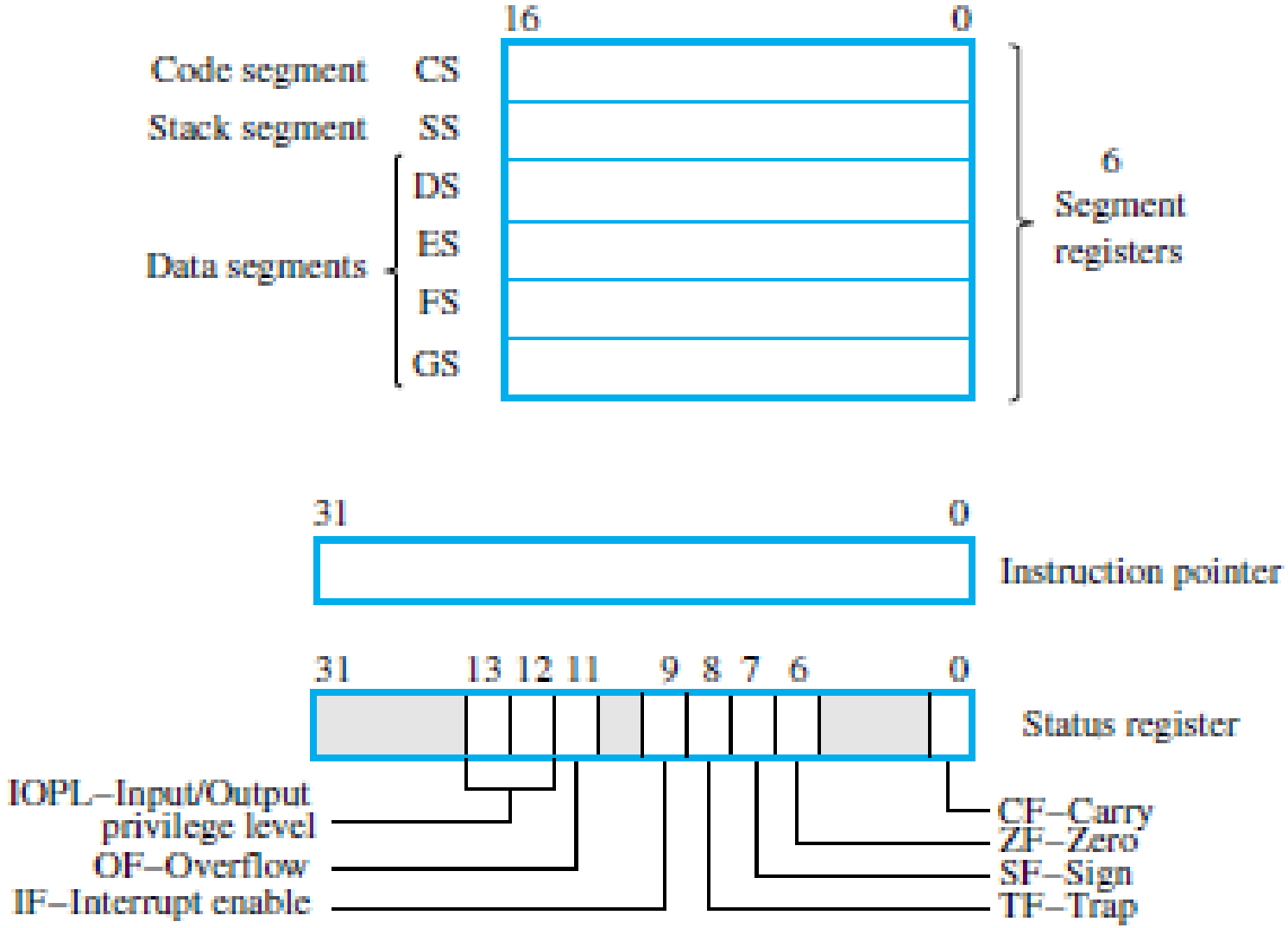
- Code segment holds the instructions of a program.

- Stack segment contains the processor stack, and four data segments are provided for holding data operands. (DS, ES, FS, GS)

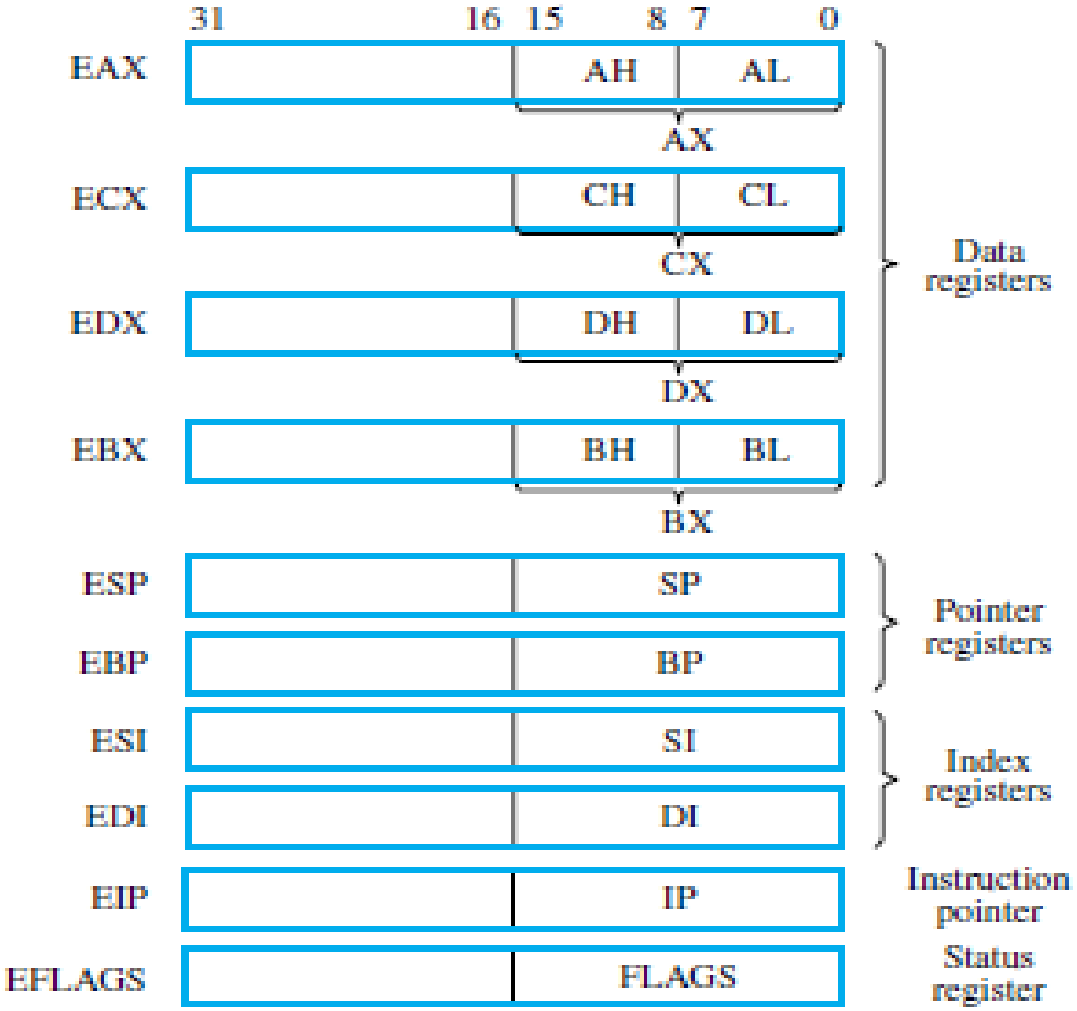
- Six segment registers contain selector values that identify where these segments begin in the memory address space.



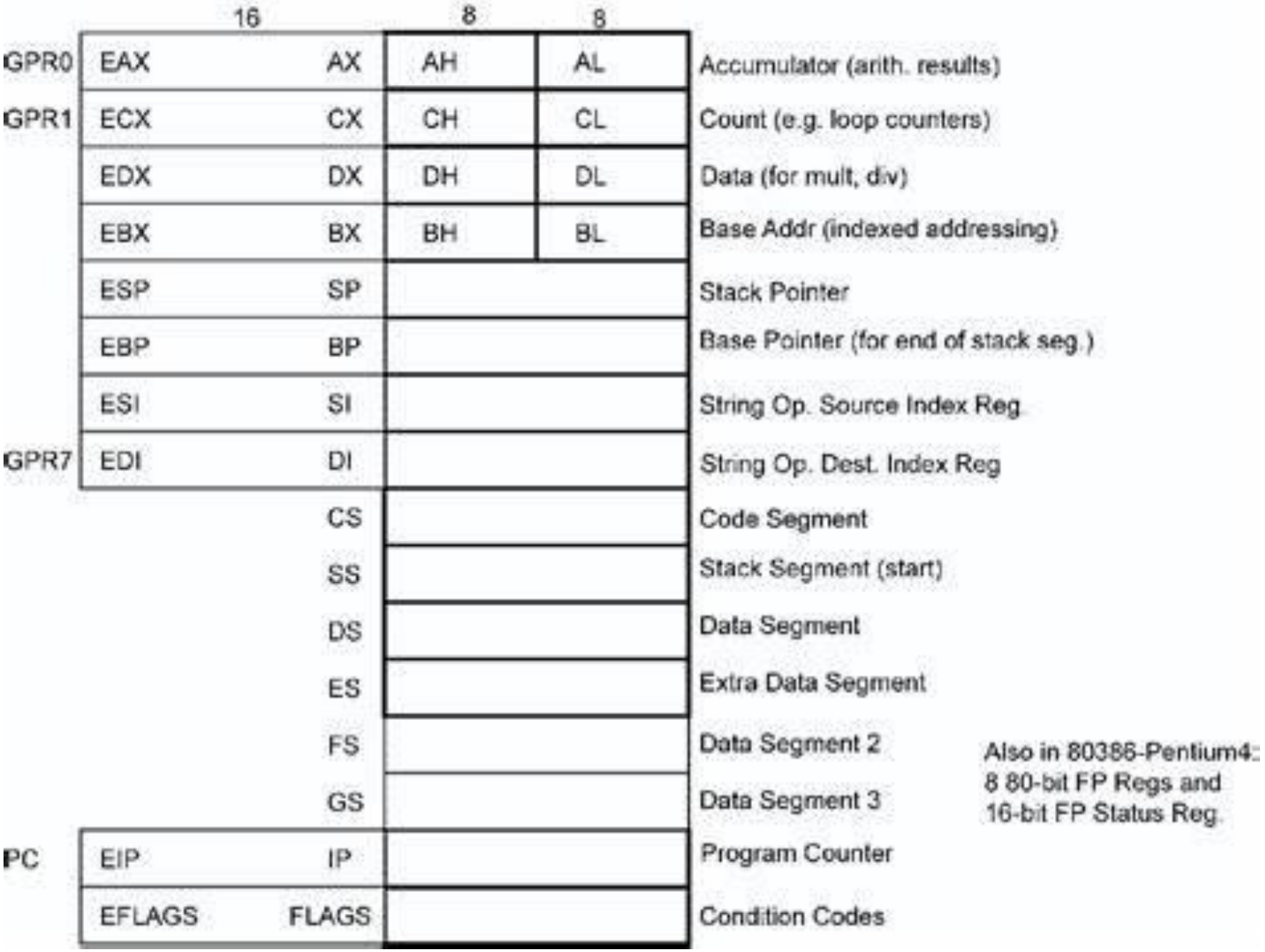
IA-32 register structure.



IA-32 register structure.



Compatibility of the IA-32 register structure with earlier Intel processor register structures.


16
8
8
GPR0
EAX
AX
AH
AL
Accumulator (arith. results)
GPR1
ECX
CX
CH
CL
Count (e.g. loop counters)
EDX
DX
DH
DL
Data (for mult, div)
EBX
BX
BH
BL
Base Addr (indexed addressing)
ESP
SP
Stack Pointer
EBP
BP
Base Pointer (for end of stack seg.)
ESI
SI
String Op. Source Index Reg
GPR7
EDI
DI
String Op. Dest. Index Reg
CS
Code Segment
SS
Stack Segment (start)
DS
Data Segment
ES
Extra Data Segment
FS
Data Segment 2
GS
Data Segment 3
PC
EIP
IP
Program Counter
EFLAGS
FLAGS
Condition Codes
Also in 80386-Pentium4:
8 80-bit FP Regs and
16-bit FP Status Reg.

➢The IA-32 general-purpose registers allow for compatibility with the registers of earlier 8-bit and 16-bit Intel processors.

➢In those processors, there are some restrictions regarding the use of certain registers

➢The eight general-purpose registers are grouped into three different types:

- data registers for holding operands,
- pointer registers for holding addresses, and
- index registers for holding address indices.

➢ The pointer and index registers are used to determine the effective address of a memory operand.

Addressing Modes

- IA-32 architecture has a large and flexible set of addressing modes.

- They are designed to access individual data items, or data items that are members of an ordered list that begins at a specified memory address.

- Effective address of the operand,

  EA, is calculated as

  $$EA = [Ri] + [Rj] + X$$

  where Ri and Rj are general-purpose registers and X is a constant..

- Instructions have zero, one, or two operands.

- In two-operand instructions, the source(src) and destination (dst) operands are specified in assembly language in the order

  OP dst, src

Addressing Modes

Table E.1 IA-32 addressing modes.

Name	Assembler syntax	Addressing function
Immediate	Value	Operand = Value
Direct	Location	EA = Location
Register	Reg	EA = Reg that is, Operand = [Reg]
Register indirect	[Reg]	EA = [Reg]
Base with displacement	[Reg + Disp]	EA = [Reg] + Disp
Index with displacement	[Reg * S + Disp]	EA = [Reg] × S + Disp
Base with index	[Reg1 + Reg2 * S]	EA = [Reg1] + [Reg2] × S
Base with index and displacement	[Reg1 + Reg2 * S + Disp]	EA = [Reg1] + [Reg2] × S + Disp

Value = an 8- or 32-bit signed number
Location = a 32-bit address
Reg, Reg1, Reg2 = one of the general purpose registers EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, with the exception that ESP cannot be used as an index register.
Disp = an 8- or 32-bit signed number, except that in the Index with displacement mode it can only be 32 bits.
S = a scale factor of 1, 2, 4, or 8

Immediate addressing mode

It is convenient to use the Move instruction to illustrate the IA-32 addressing modes and their notation in assembly language.

The instruction

MOV EAX, 25

uses the Immediate addressing mode for the source operand to move the decimal value 25 into the destination register EAX.

Direct addressing mode

Symbolic names may also be used as operands. If the name LOCATION has been defined as an address label, the instruction

MOV EAX, LOCATION

implicitly uses the Direct addressing mode to move the doubleword at memory address LOCATION into register EAX.

The Direct addressing mode can also be made explicit. The instruction

MOV EAX, DWORD PTR LOCATION

uses the keywords DWORDPTR to indicate that the label LOCATION should be interpreted as the address of a 32-bit operand.

Register addressing mode

MOV EAX, EBX

moves the value of the address label EBX into the EAX register using the Register addressing mode.

Register indirect addressing mode

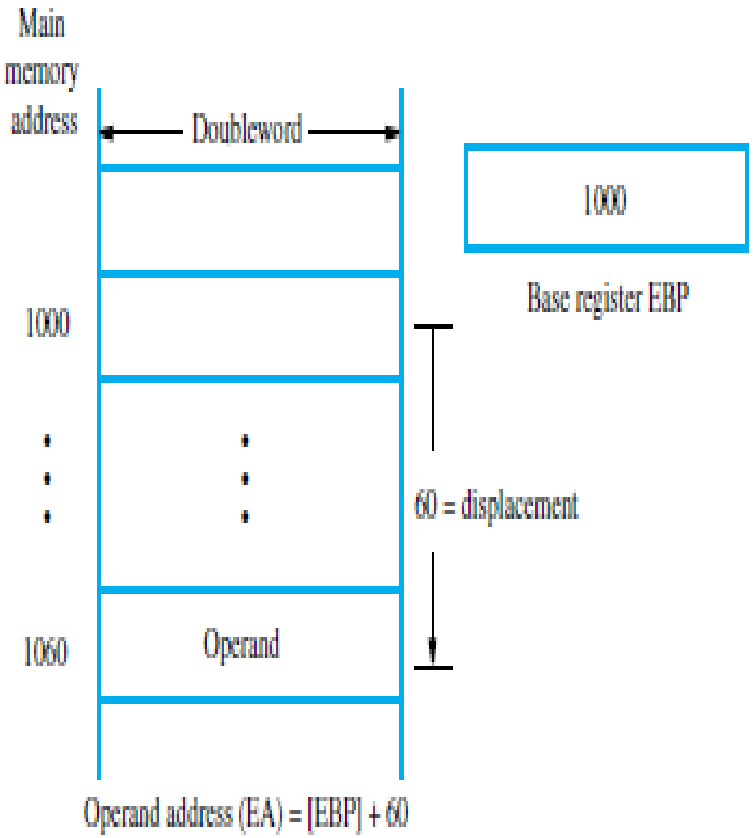
Once an address is loaded into a register, the Register indirect mode can be used to access the operand in memory.
The instruction

MOV EAX, [EBX]

moves the contents of the memory location whose address is contained in register EBX into register EAX.

Base with displacement mode

- Register EBP is used as the base register.
- A doubleword operand at address 1060, which is 60 byte locations away from the base address of 1000, can be moved into register EAX by the instruction
- MOV EAX, [EBP + 60]



Base with displacement mode, expressed as [EBP + 60]

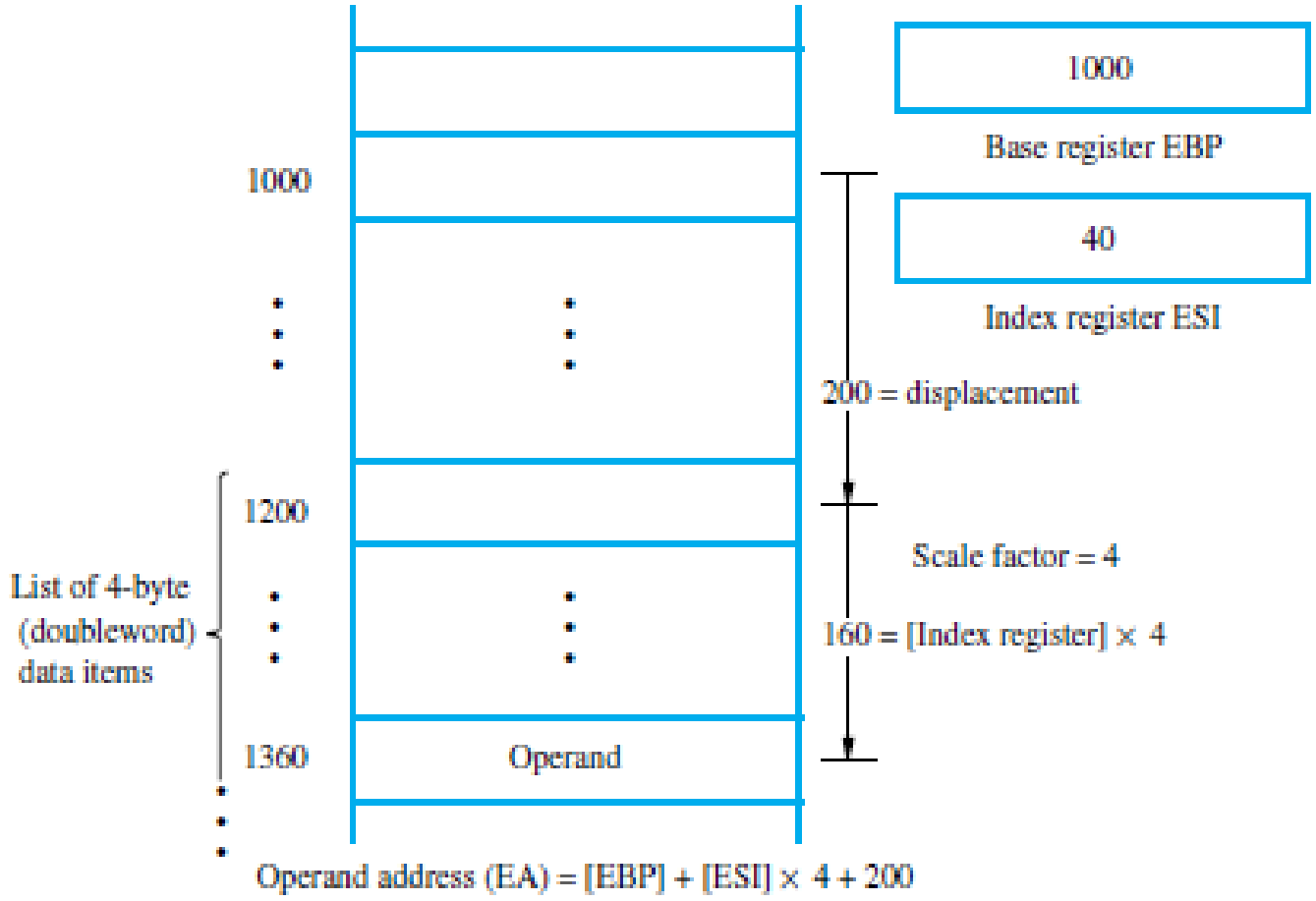
Base with index and displacement mode

- An 8-bit or 32-bit signed displacement, two of the eight general-purpose registers, and a scale factor of 1, 2, 4, or 8, are specified in the instruction. The registers are used as base and index registers.

- Effective address of the operand is determined by first multiplying the contents of the index register by the scale factor and then adding the result to the contents of the base register and the displacement.

- The addressing mode that provides the most flexibility

MOV EAX, [EBP + ESI * 4 + 200]



Base with displacement and index mode, expressed as [EBP + ESI * 4 + 200]

Instructions

- IA-32 instruction set is extensive.

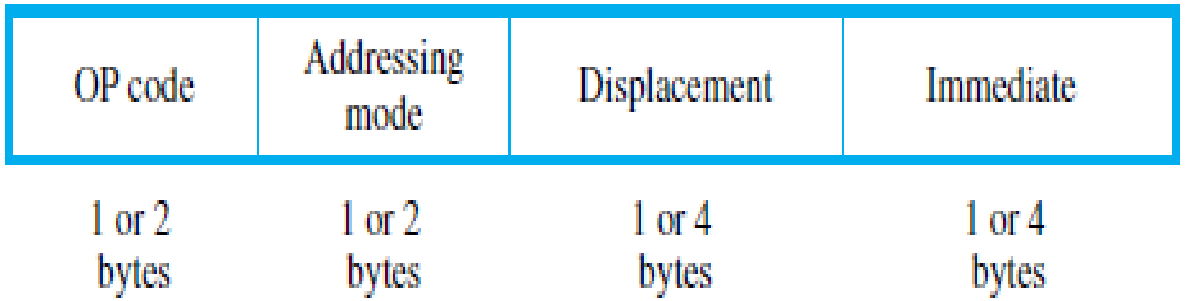
- Instructions are provided for moving data between the memory and the processor registers, performing arithmetic operations, and performing logical and shift/rotate operations.

- In the two-operand case, only one of the operands can be in the memory.

- The other must either be in a processor register or be an immediate value in the instruction.

➢Jump instructions and subroutine call/return instructions are included.

➢Push and pop operations for manipulating the processor stack are also directly supported in the instruction set.



IA-32 instruction format.

Machine Instruction Format

- Instructions are variable in length, ranging from 1 to 12 bytes and consisting of up to four fields.

- The OP-code field consists of one or two bytes, with most instructions requiring only one byte.

- Addressing mode information is contained in one or two bytes immediately following the OP code.

- For instructions that involve the use of only one register in generating the effective address of an operand in memory, only one byte is needed in the addressing mode field.

- Some simple instructions, such as those that increment or decrement a register, occupy only one byte. For example, the instruction

INC EDI

Move Instruction

➢Register contents may also be transferred to memory or to another register. The instruction

MOV LOCATION, ECX

moves the doubleword in register ECX into the memory location at address LOCATION.

➢The instruction

MOV EBP, EDI

moves the doubleword in register EDI to register EBP. The contents in register EDI are not changed.

The MOV instruction cannot be used with two memory operands, but it can be used to move an immediate value into a memory location, as in

MOV DWORD PTR [EAX + 16], 100

Note that the assembler requires the keywords DWORD PTR (or BYTE PTR) to specify the operand size in this instruction.

Load-Effective-Address Instruction

MOV EAX, OFFSET LOCATION

MOV instruction can be used to load an address into a register by using the keyword OFFSET. Alternatively, the LEA (Load-effective-address) instruction may be used

LEA EAX, LOCATION

The LEA instruction can be used to load an effective address that is computed at execution time.

The desired operand is an element of an array, located at an offset of 12 bytes from the start of the array. If register EBP contains the starting address of the array, the instruction

LEA EBX, [EBP + 12]

computes the desired effective address and places it in register EBX. The operand can then be accessed by a Move or other instruction using the Register indirect mode with EBX.

Arithmetic Instructions
Addition, Subtraction, Comparison, and Negation

Two-operand arithmetic instructions are:

- ADD (Add)
- ADC (Add with carry; for multiple-precision arithmetic)
- SUB (Subtract)
- SBB (Subtract with borrow; for multiple-precision arithmetic)
- CMP (Compare; value of destination operand remains unchanged)

One-operand arithmetic instructions are:

- INC (Increment)
- DEC (Decrement)
- NEG (Negate)

The NEG instruction affects all condition code flags, but the INC and DEC instructions do not affect the CF flag. These instructions must include keywords to specify the operand size unless the Register mode is used for the operand.

These instructions affect all of the condition code flags based on the result of the operation that is performed.

The instruction ADD EAX, EBX performs the 32-bit operation

EAX←[EAX] + [EBX]

The instruction CMP [EBX + 10], AL performs the 8-bit operation

[[EBX] + 10] − [AL]

Using register AL implies an operand size of one byte

The instruction

INC DWORD PTR [EDX]

increments the doubleword at the memory location whose address is contained in register EDX.

Multiplication

The signed integer multiplication instruction, IMUL, performs 32-bit multiplication. Depending on the form of the instruction that is used, the destination may be implicit and the 64-bit product may be truncated to 32 bits.

One form of this instruction is

IMUL src

which implicitly uses the EAX register as the multiplicand.

A second form of this instruction is

IMUL REG, src

The destination operand, REG, must be one of the eight general-purpose registers. The source operand can be in a register or in the memory. The product is truncated to 32 bits before it is placed in the destination register REG.

Division

The integer divide instruction, IDIV, operates on a 64-bit dividend and a 32-bit divisor to generate a 32-bit quotient and a 32-bit remainder.

The format of the instruction is

IDIV src

The source operand is the divisor. The 64-bit dividend is formed by the contents of register EDX (high-order half) and register EAX (low-order half).

After performing the division, the quotient is placed in EAX and the remainder is placed in EDX. All of the condition code flags are undefined. Division by zero causes an exception.

Jump and Loop Instructions

Conditional Jump Instructions and Condition Code Flags

Table E.2 IA-32 conditional jump instructions.

Mnemonic	Condition name	Condition test
JS	Sign (negative)	SF = 1
JNS	No sign (positive or zero)	SF = 0
JE/JZ	Equal/Zero	ZF = 1
JNE/JNZ	Not equal/Not zero	ZF = 0
JO	Overflow	OF = 1
JNO	No overflow	OF = 0
JC/JB	Carry/Unsigned below	CF = 1
JNC/JAE	No carry/Unsigned above or equal	CF = 0
JA	Unsigned above	CF $\vee$ ZF = 0
JBE	Unsigned below or equal	CF $\vee$ ZF = 1
JGE	Signed greater than or equal	SF $\oplus$ OF = 0
JL	Signed less than	SF $\oplus$ OF = 1
JG	Signed greater than	ZF $\vee$ (SF $\oplus$ OF) = 0
JLE	Signed less than or equal	ZF $\vee$ (SF $\oplus$ OF) = 1

Unconditional Jump Instruction

- An unconditional Jump instruction, JMP, causes a branch to the instruction at the target address.

- In addition to using short (one-byte) or long (four-byte) relative signed offsets to determine the target address, as is done in conditional Jump instructions, the JMP instruction also allows the use of other addressing modes.

- This flexibility in generating the target address can be very useful.

- At execution time, the index of the selected case is loaded into index register ESI.

- A jump to the routine for the selected case is performed by executing the instruction

JMP [JUMPTABLE + ESI * 4]

which uses the Index with displacement addressing mode.

Loop Instruction

A loop can be implemented as

```
        MOV ECX, NUM_PASSES
START:
        ...
        DEC ECX
        JG START
```

Loops of this form can be expressed in a more compact manner by using the LOOP instruction.

It combines the functionality of the DEC and JG instructions, and it also implicitly uses register ECX for the counter variable. Using this instruction, the loop can be implemented as

```
        MOV ECX, NUM_PASSES
START:
        ...
        LOOP START
```

Condition code flags are not affected by the LOOP instruction.

Logic Instructions

➢The IA-32 architecture has instructions that perform the logic operations AND, OR, and XOR.

➢ The operation is performed bitwise on two operands, and the result is placed in the destination location.

➢ For example, suppose register EAX contains the hexadecimal pattern 0000FFFF and register EBX contains the pattern 02FA62CA. The instruction

➢AND EBX, EAX

clears the left half of EBX to all zeroes, and leaves the right half unchanged. The result in EBX will be 000062CA.

➢There is also a NOT instruction which generates the logical complement of all bits of the operand, that is, it changes all 1s to 0s and all 0s to 1s.

Shift and Rotate Instructions

An operand can be shifted right or left, using either logical or arithmetic shifts, by a number of bit positions determined by a specified count. The format of the shift instructions is

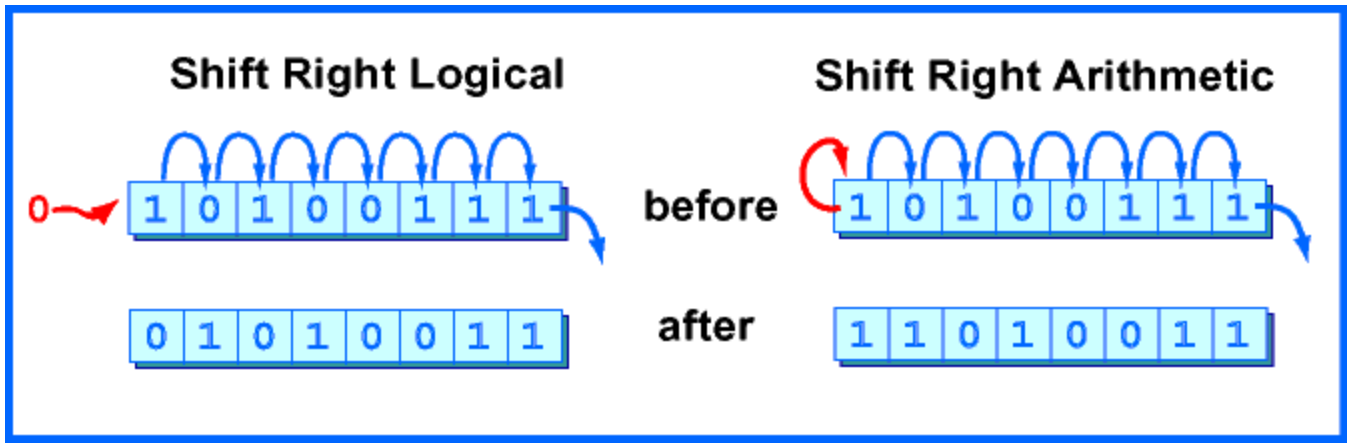
OP dst, count

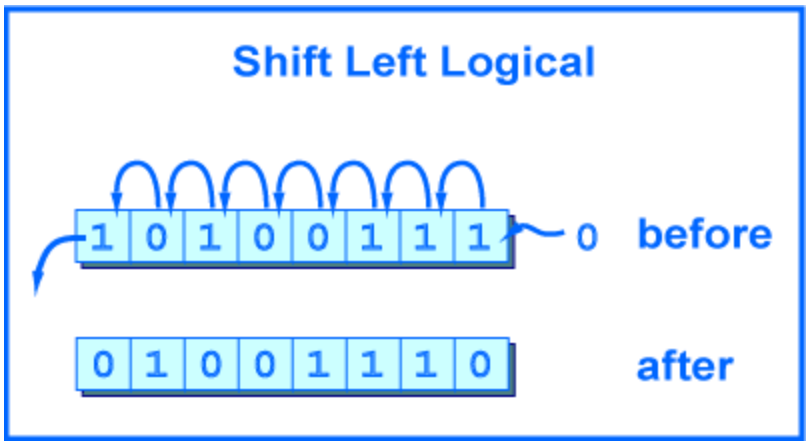
There are four shift instructions:

- SHL (Shift left logical)
- SHR (Shift right logical)
- SAL (Shift left arithmetic; operation is identical to SHL)
- SAR (Shift right arithmetic)

There are four rotate instructions:

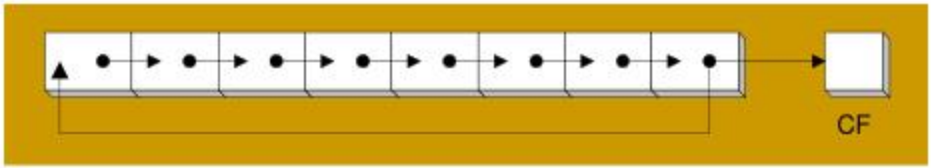
- ROL (Rotate left without the carry flag CF)
- ROR (Rotate right without the carry flag CF)
- RCL (Rotate left including the carry flag CF)
- RCR (Rotate right including the carry flag CF)


Shift Right Logical
Shift Right Arithmetic
0
1 0 1 0 0 1 1 1
before
1 0 1 0 0 1 1 1
0 1 0 1 0 0 1 1
after
1 1 0 1 0 0 1 1


Shift Left Logical
1 0 1 0 0 1 1 1
0
before
0 1 0 0 1 1 1 0
after

ROR Instruction

- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost

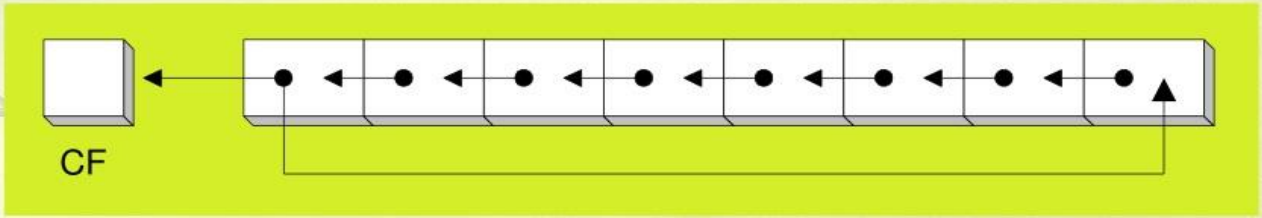



```
mov al,11110000b
ror al,1                ; AL = 01111000b

mov dl,3Fh
ror dl,4                ; DL = F3h
```

ROL Instruction

- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost



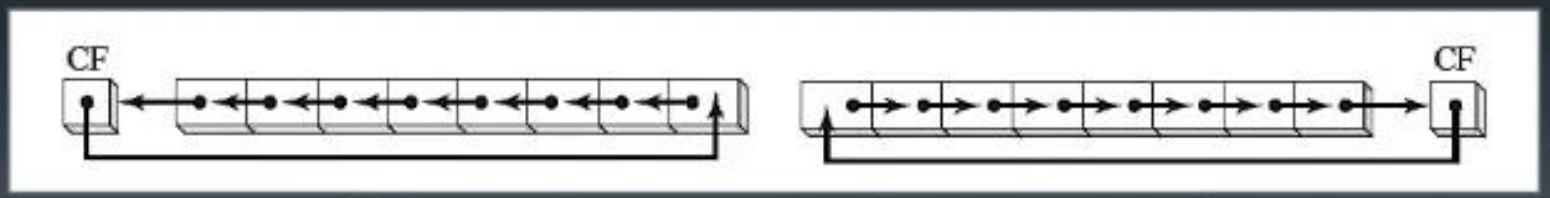

```
MOV Al,11110000b
ROL Al,1                    ; AL = 11100001b

MOV Dl,3Fh
ROL Dl,4                    ; DL = F3h
```

RCL & RCR Instruction

- There is another set of rotation instructions which are RCL (rotate carry left) and RCR (rotate carry right).

- They assume the carry bit is an extra bit in the operand and rotate the whole bits including the carry bit.



RCL *dest, cnt*

RCR *dest, cnt*

Subroutine Linkage Instructions

➢The stack grows toward lower numbered addresses. The width of the stack is 32 bits, that is, all stack entries are doublewords.

➢There are two instructions for pushing and popping individual elements onto and off the stack. The instruction

PUSH src

decrements ESP by 4, and then stores the doubleword at location src into the memory location pointed to by ESP.

➢ The instruction

POP dst

reverses this process by retrieving the doubleword from the location pointed to by ESP, storing it at location dst, and then incrementing ESP by 4. These instructions implicitly use ESP as the stack pointer.

➢There are also two more instructions that push or pop the contents of multiple registers. The instruction

PUSHAD

POPAD

➢ These two instructions are used to efficiently save and restore the contents of all registers as part of implementing subroutines

➢The subroutine is called by the instruction

CALL LISTADD

which first pushes the return address onto the stack and then jumps to LISTADD. The return address is the address of the MOV instruction that immediately follows the CALL instruction.

➢The subroutine saves the contents of register EDI on the stack.

Calling program

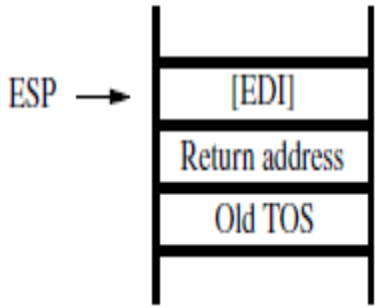
```
                ⋮
        LEA     EBX, NUM1       Load parameters
        MOV     ECX, N            into EBX, ECX.
        CALL    LISTADD         Branch to subroutine.
        MOV     SUM, EAX        Store sum into memory.
                ⋮
```

Subroutine

```
LISTADD:    PUSH    EDI                 Save EDI.
            MOV     EDI, 0              Use EDI as index register.
            MOV     EAX, 0              Use EAX as accumulator register.

STARTADD:   ADD     EAX, [EBX + EDI * 4]  Add next number.
            INC     EDI                 Increment index.
            DEC     ECX                 Decrement counter.
            JG      STARTADD            Branch back if [ECX] > 0.
            POP     EDI                 Restore EDI.
            RET                         Return to calling program.
```

Calling program and subroutine



Stack contents after saving EDI in subroutine

Operations on Large Numbers

```
MOV   EAX, 0A72C10F8H   EAX contains A72C10F8.
MOV   EBX, 10H          EBX contains 10.
MOV   ECX, 5C00FE04H    ECX contains 5C00FE04.
MOV   EDX, 4AH          EDX contains 4A.
ADD   EAX, ECX          Add low-order 32 bits; carry-out sets CF flag.
ADC   EBX, EDX          Add high-order bits with CF flag as carry-in bit.
```

Addition of numbers larger than 32 bits using the ADC instruction.

Interrupts and Exceptions

- Processors implementing the IA-32 architecture use two interrupt-request lines, a nonmaskable interrupt, NMI, and a maskable interrupt, also called the user interrupt request, INTR.

- Interrupt requests on NMI are always accepted by the processor. Requests on INTR are accepted only if they have a higher privilege level than the program currently running.

- INTR interrupts can be enabled or disabled by setting an interrupt-enable bit, IF, in the status register.

- The status register, contains the Interrupt Enable Flag (IF), the Trap flag (TF) and the I/O Privilege Level (IOPL).

- When IF = 1, INTR interrupts are accepted. The Trap flag enables trace interrupts after every instruction.

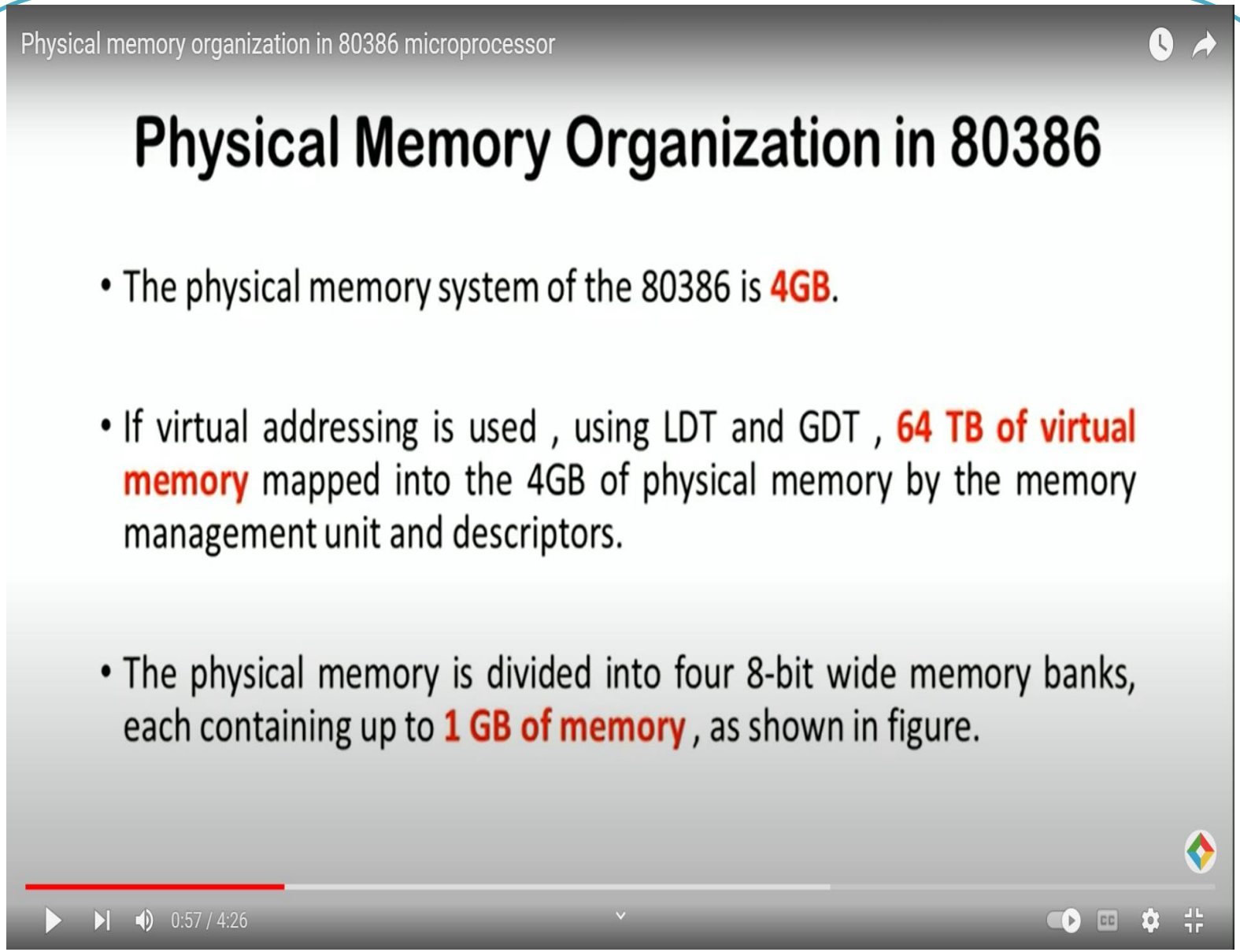
When an interrupt request is received or when an exception occurs, the processor performs the following actions:

1. It pushes the status register, the Code Segment register (CS), and the Instruction Pointer (EIP) onto the processor stack.

2. In the case of an exception resulting from an abnormal execution condition, it pushes a code on the stack describing the cause of the exception.

3. It clears the corresponding interrupt-enable flag so that further interrupts from the same source are disabled.

4. It fetches the starting address of the interrupt-service routine from the Interrupt Descriptor Table based on the vector number of the interrupt, loads this value into EIP, and then resumes execution.


Physical memory organization in 80386 microprocessor
Physical Memory Organization in 80386
• The physical memory system of the 80386 is 4GB.
• If virtual addressing is used , using LDT and GDT , 64 TB of virtual memory mapped into the 4GB of physical memory by the memory management unit and descriptors.
• The physical memory is divided into four 8-bit wide memory banks, each containing up to 1 GB of memory , as shown in figure.
0:57 / 4:26

Physical memory organization in 80386 microprocessor

Physical Memory Organization in 80386

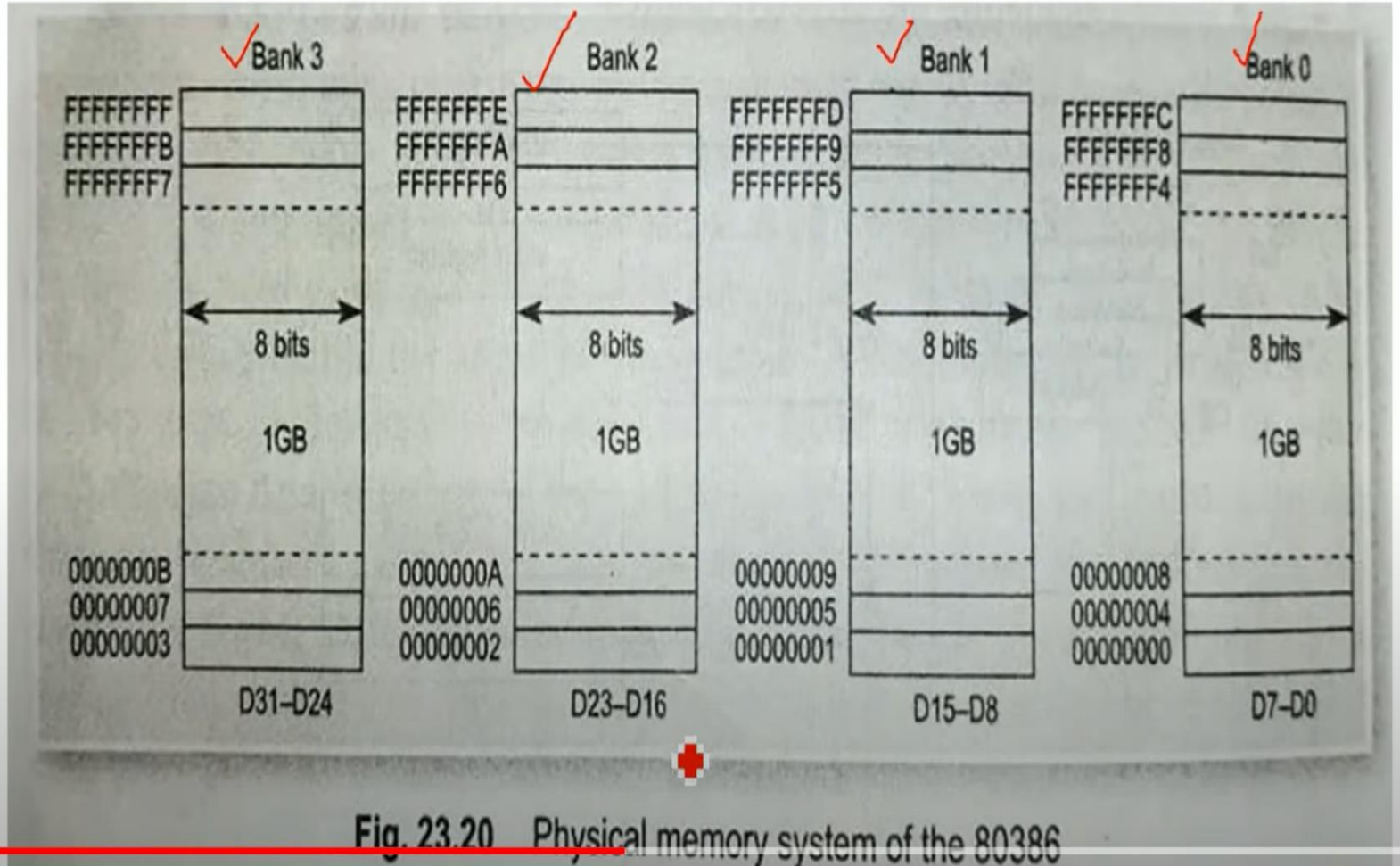
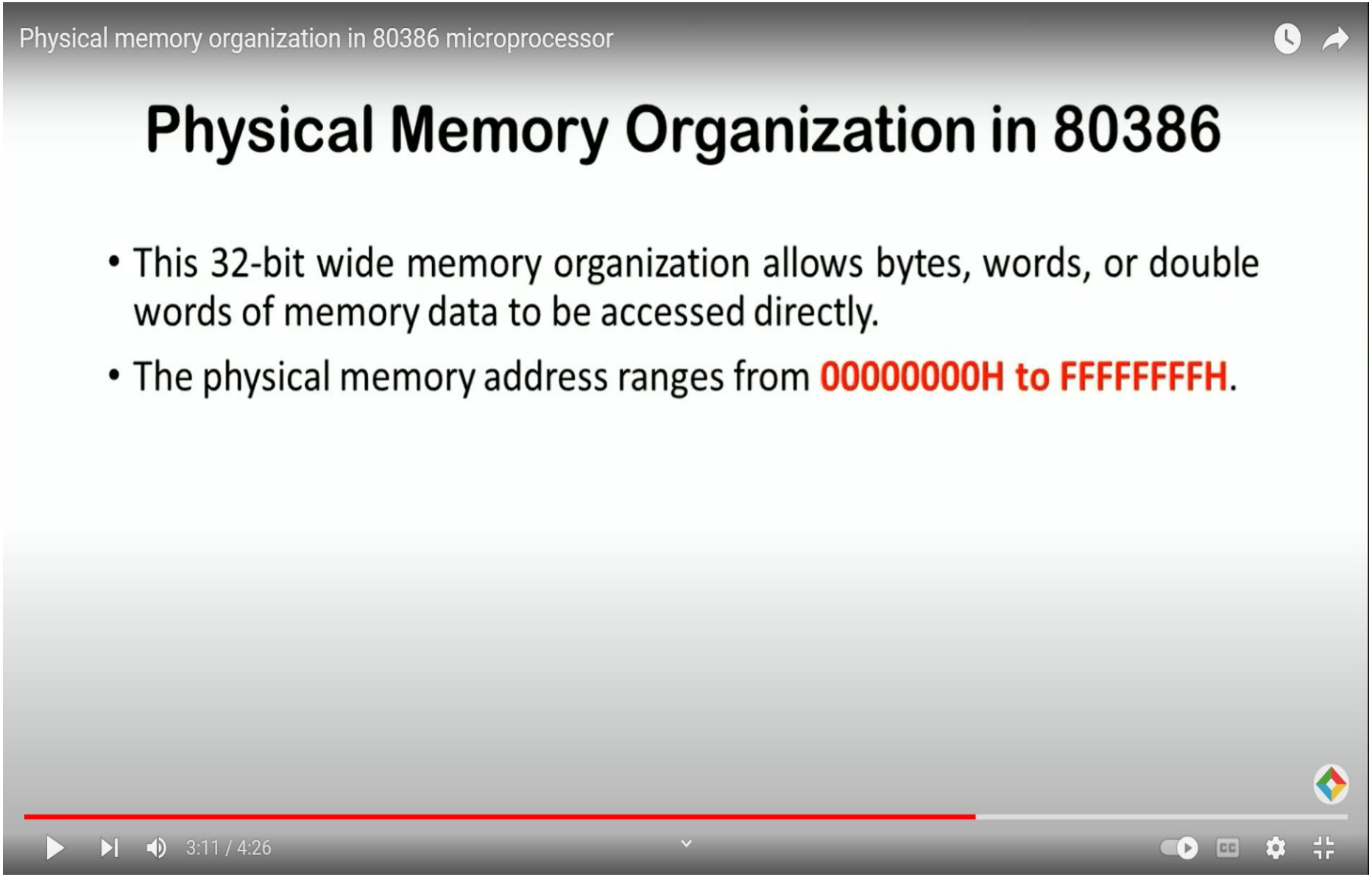


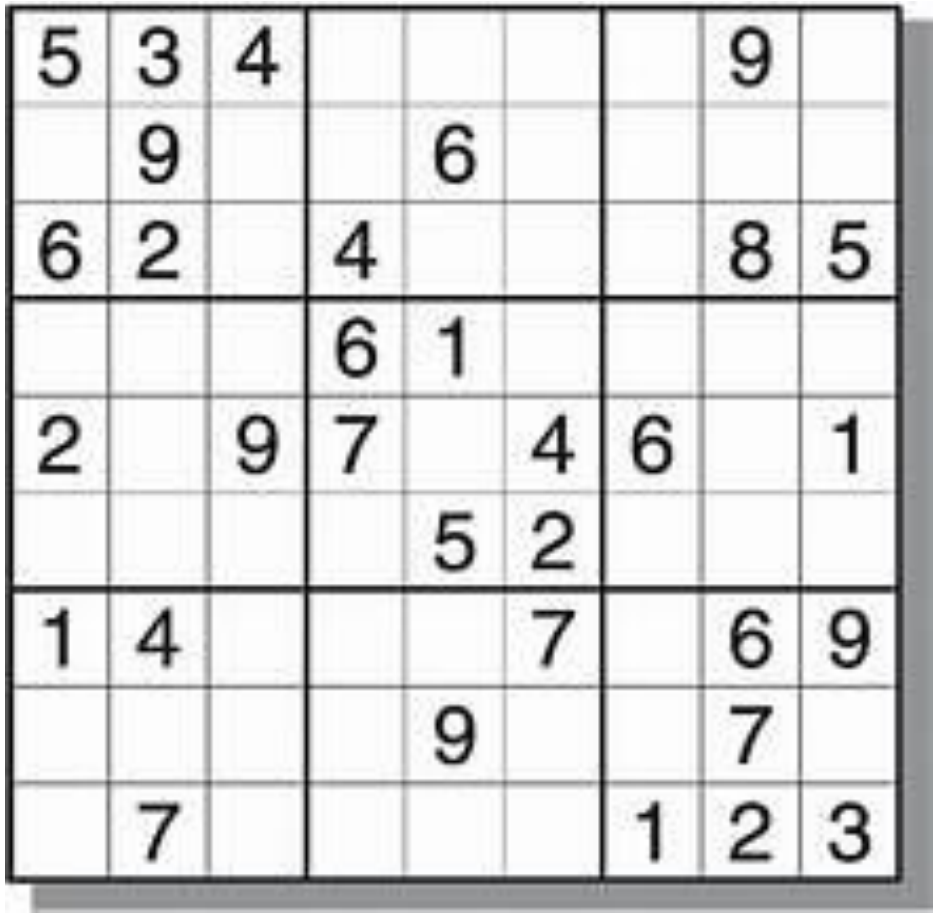
Fig. 23.20 Physical memory system of the 80386

2:04 / 4:26


Physical memory organization in 80386 microprocessor
Physical Memory Organization in 80386
• This 32-bit wide memory organization allows bytes, words, or double words of memory data to be accessed directly.
• The physical memory address ranges from 00000000H to FFFFFFFFH.
3:11 / 4:26

Physical Memory Organization in 80386

- The physical memory location with address 00000000H is in Bank 0 , 00000001H in Bank 1 , 00000002H in Bank 2 , 00000003H in Bank 3, etc.,
- The memory banks 3,2,1, and 0 are accessed via four bank enable signals - **BE3* , BE2* , BE1* , and BE0*** respectively.
- In most cases , a word is addressed in banks 0 and 1 , or in banks 2 and 3.



5	3	4					9	
	9			6				
6	2		4				8	5
			6	1				
2		9	7		4	6		1
				5	2			
1	4				7		6	9
				9			7	
	7					1	2	3

a puzzle in which players insert the numbers one to nine into a grid consisting of nine squares subdivided into a further nine smaller squares in such a way that every number appears once in each horizontal line, vertical line, and square.