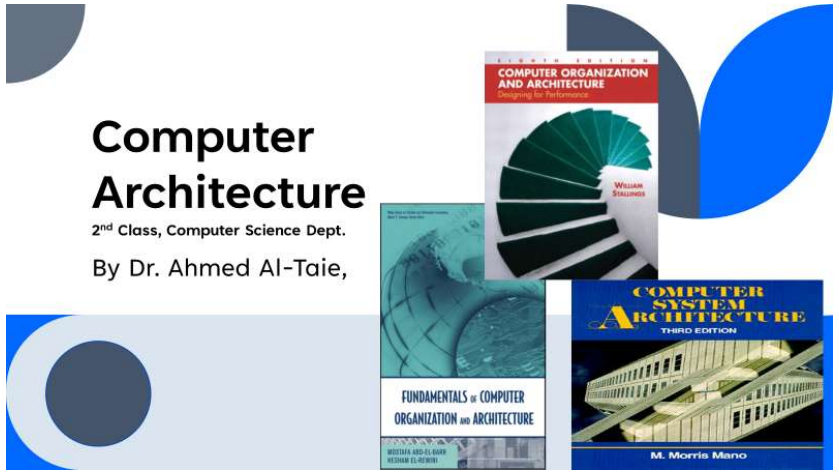
Computer Architecture

2nd Class, Computer Science Dept.

By Dr. Ahmed Al-Taie,

Chapter 3
Assembly Language Programming
Outline

- A Simple Machine
- Instructions Mnemonics and Syntax
- Assembler Directives and Commands
- Assembly and Execution of Programs
- **Example**: The X86 Family



ASSEMBLY AND EXECUTION OF PROGRAMS

- As you know by now, a program written in **assembly language** needs to be **translated** into **binary machine language** before it can be **executed.**
- In this section, we will learn how to get **from the point of writing** an **assembly program** to the **execution phase**.
  - **Figure 3.3** shows **three steps** in the **assembly** and **execution process**.
  - The assembler reads the source program in assembly language and generates the object program in binary form.
  - The object program is passed to the linker.
- **The linker** will check the object file for calls to procedures in the link library.
- **The linker** will combine the required procedures from the link library with the object program and produce the executable program.
- **The loader** loads the executable program into memory and branches the CPU to the starting address. The program begins execution.

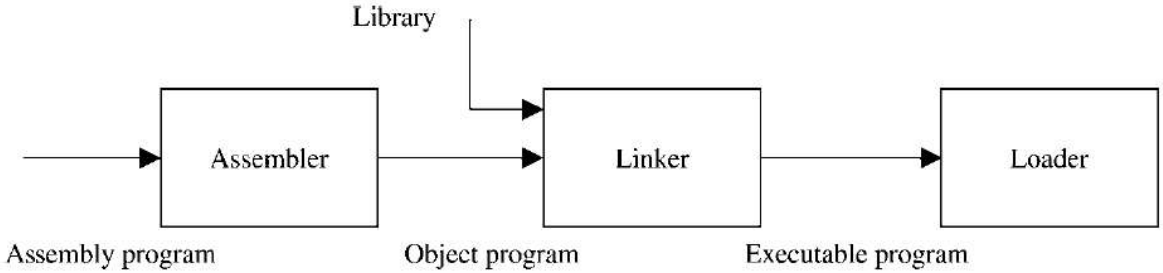


Figure 3.3 Assembly and execution process

ASSEMBLY AND EXECUTION OF PROGRAMS

- **Assemblers:**
- **Assemblers** are **programs** that generate **machine code instructions** from a **source code program** written in **assembly language**.
- The **assembler** will **replace symbolic addresses** by **numeric addresses**, **replace symbolic operation codes by machine operation codes, reserve storage** for **instructions** and **data,** and **translate constants into machine representation**.
- The **functions of the assembler** can be **performed** by **scanning** the **assembly program** and **mapping** its **instructions** to their **machine code equivalent**.
- Since **symbols** can be **used** in **instructions before** they are **defined in later ones**, a **single scanning** of the **program** might not be **enough to perform the mapping**.
- A **simple assembler scans** the **entire assembly program twice**, where **each scan** is called a **pass**. During the **first pass**, it **generates** a **table** that includes **all symbols** and **their binary values**.
- **This table** is **called** the **symbol table**.
- During the **second pass**, the **assembler** will use the **symbol table** and **other tables** to **generate** the **object program**, and **output some information** that will be **needed** by **the linker**.

ASSEMBLY AND EXECUTION OF PROGRAMS

- **Data Structures:** The **assembler uses at least three tables to perform its functions: symbol table, opcode table,** and **pseudo instruction table.**
- **The symbol table,** which is **generated** in **pass one,** has **an entry** for **every symbol in the program.**
- **Associated** with **each symbol** are its **binary value** and **other information.**

TABLE 3.5 Symbol Table for the Multiplication Segment (Example 2)

Symbol	Value (hexadecimal)	Other information
Loop	004	
EXIT	01E	
X	020	
Y	022	
Z	024	
ONE	026	
N	028	

ASSEMBLY AND EXECUTION OF PROGRAMS

- We **assume** that the **instruction** **LD X** is starting at **location 0** in the **memory**.
- Since **each instruction** takes **two bytes,** the **value** of the **symbol LOOP** is 4 **(004** in **hexadecimal)**.
- **Symbol N**, for **example**, will be stored at **decimal location 40** (**028** in **hexadecimal**).
- The **values** of the other **symbols** can be **obtained** in a **similar way**.
- The **opcode table** provides information about the **operation codes**.

- **Associated** with each **symbolic opcode** in the **table** are its **Numerical value** and **other information about its type**, its instruction **length**, and its **operands**.
- **Table 3.6** shows the **opcode Table** for the simple processor described in Section 3.1.

TABLE 3.6 The Opcode Table for the Assembly of our Simple Processor

Opcode	Operand	Opcode value (binary)	Instruction length (bytes)	Instruction type
STOP	—	0000	2	Control
LD	*Mem-adr*	0001	2	Memory-reference
ST	*Mem-adr*	0010	2	Memory-reference
MOVAC	—	0011	2	Register-reference
MOV	—	0100	2	Register-reference
ADD	—	0101	2	Register-reference
SUB	—	0110	2	Register-reference
AND	—	0111	2	Register-reference
NOT	—	1000	2	Register-reference
BRA	*Mem-adr*	1001	2	Control
BZ	*Mem-adr*	1010	2	Control

ASSEMBLY AND EXECUTION OF PROGRAMS

- As **an example**, we **explain** the **information associated** with the **opcode LD.**
- It has **one operand**, which is **a memory address** and its **binary value** is **0001**.
- The **instruction length** of **LD** is **2 bytes** and its type is memory-reference.
- The **entries** of the **pseudo instruction table** are the **pseudo instructions symbols**.
- **Each entry refers** the **assembler** to a **procedure** that **processes** the **pseudo instruction** when encountered in the program.
- For **example**, if **END** is **encountered,** the **translation process is terminated.**

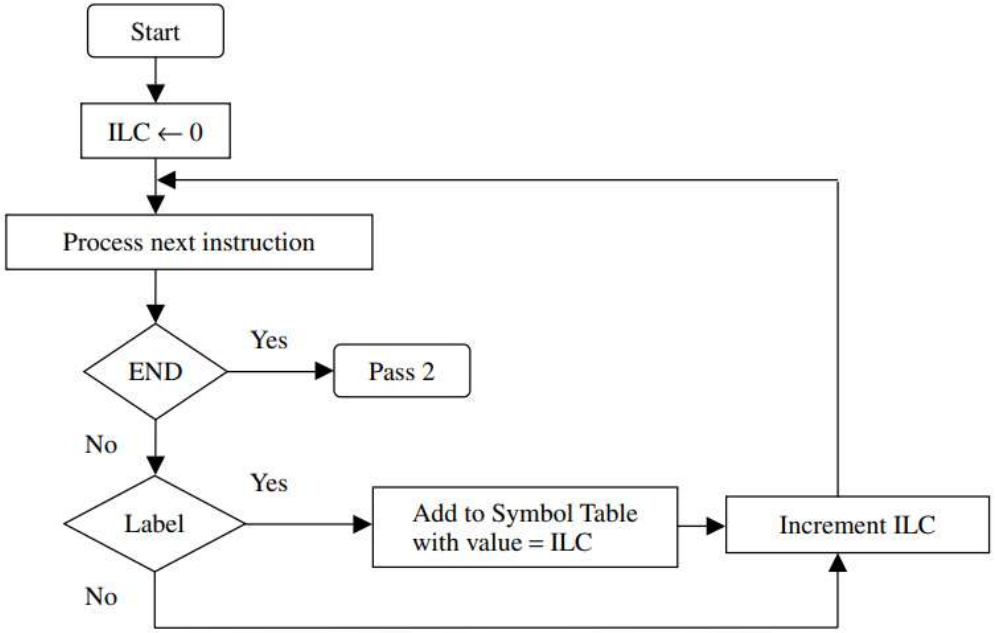


Figure 3.4 Simplified pass one in a two-pass assembler

ASSEMBLY AND EXECUTION OF PROGRAMS

- In order to **keep track** of the **instruction locations,** the **assembler maintains** a **variable called instruction location counter (ILC).**
- The **ILC** contains the **value of memory location assigned to the instruction or operand** being **processed.**
- The **ILC is initialized to 0** and is **incremented after processing each instruction.**
- The **ILC** is **incremented by** the **length** of the **instruction being processed,** or **the number of bytes allocated** as a result of a **data allocation pseudo instruction.**
- **Figures 3.4 and 3.5** show **simplified flowcharts** of **pass one and pass two** in a **two-pass assembler.**

- **Remember t**hat the **main function of pass one** is to build the **symbol table** while **pass two's main function** is to **generate** the **object code.**

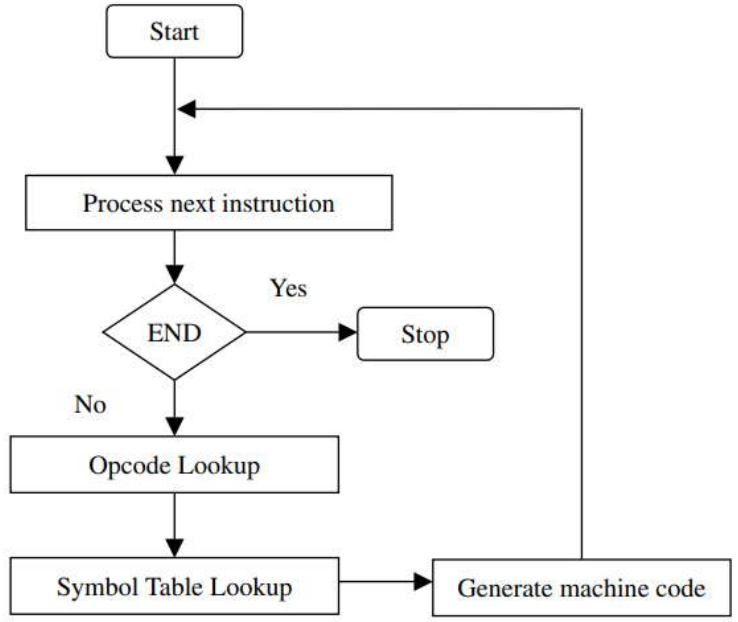


Figure 3.5 Simplified pass two in a two-pass assembler

Linker and Loader

- **The linker** is **the entity** that **can combine object modules** that may have **resulted from assembling multiple assembly modules separately**.
- **The loader** is the **operating system utility** that **reads (loads) the executable** into **memory** and **start execution**.
- **In summary, after assembly modules** are **translated into object modules**, the **functions of the linker** and **loader prepare the program** for **execution.**
- **These functions** include **combining object modules together, resolving addresses unknown at assembly time, allocating storage, and finally executing the program.**

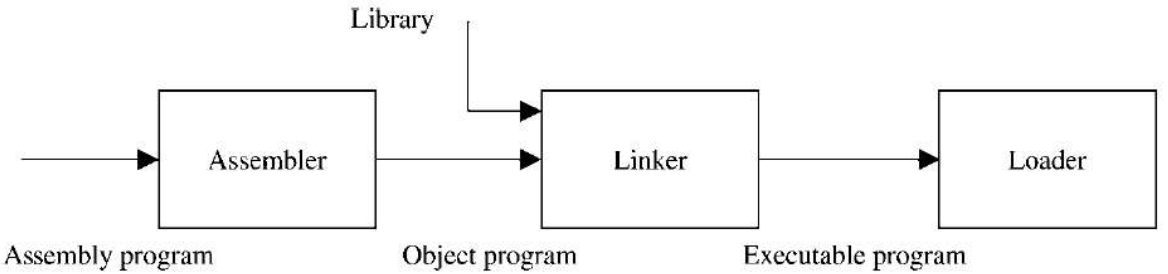


Figure 3.3 Assembly and execution process

EXAMPLE: THE X86 FAMILY

- In **this section**, we **discuss** the **assembly language features** and **use of the X86 family**.
- We **present** the **basic organizational features** of the **system**, the **basic programming model,** the **addressing modes**, **sample of** the **different instruction types used**, and **finally examples showing how to use the assembly language of the system in programming sample real-life problems**.
- In **the late 1970s, Intel introduced the 8086** as its **first 16-bit microprocessor**.
- **This processor has a 16-bit external bus**.
- The **8086 evolved into a series of faster** and **more powerful processors starting with** the **80286** and **ending with** the **Pentium.**
- The **latter was introduced in 1993**.
- This **Intel family of processors** is usually **called the X86 family**.

EXAMPLE: THE X86 FAMILY

- Table 3.7 summarizes the main features of the main members of such a family.

TABLE 3.7 Main Features of the Intel *X86* Microprocessor Family

Feature	8086	286	386	486	Pentium
Date introduced	1978	1982	1985	1991	1993
Data bus	8 bits	16 bits	32 bits	32 bits	64 bits
Address bus	20 bits	24 bits	32 bits	32 bits	32 bits
Operating speed	5,8,10 MHz	6,8,10, 12.5, 16, 20 MHz	16, 20,25, 33, 40, 50 MHz	25, 33, 50 MHz	50, 60, 66, 100 MHz
Instruction cache size	NA	NA	16 bytes	32 bytes	8 Kbytes
Data cache size	NA	NA	256 bytes	8 Kbytes	8 Kbytes
Physical memory	1 Mbytes	16 Mbytes	4 Gbytes	4 Gbytes	4 Gbytes
Data word size	16 bits	16 bits	16 bits	32 bits	32 bits

EXAMPLE: THE X86 FAMILY

- The **Intel Pentium processor** has about **three million transistors** and **its computational power ranges between two and five times that of its predecessor processor, the 80486.**
- **A number** of **new features** were **introduced in the Pentium processor**, **among which** is the **incorporation of a dual-pipelined superscalar architecture capable of processing more than one instruction per clock cycle.**
- The **basic programming model** of the **386, 486, and the Pentium** is shown in **Figure 3.6.**

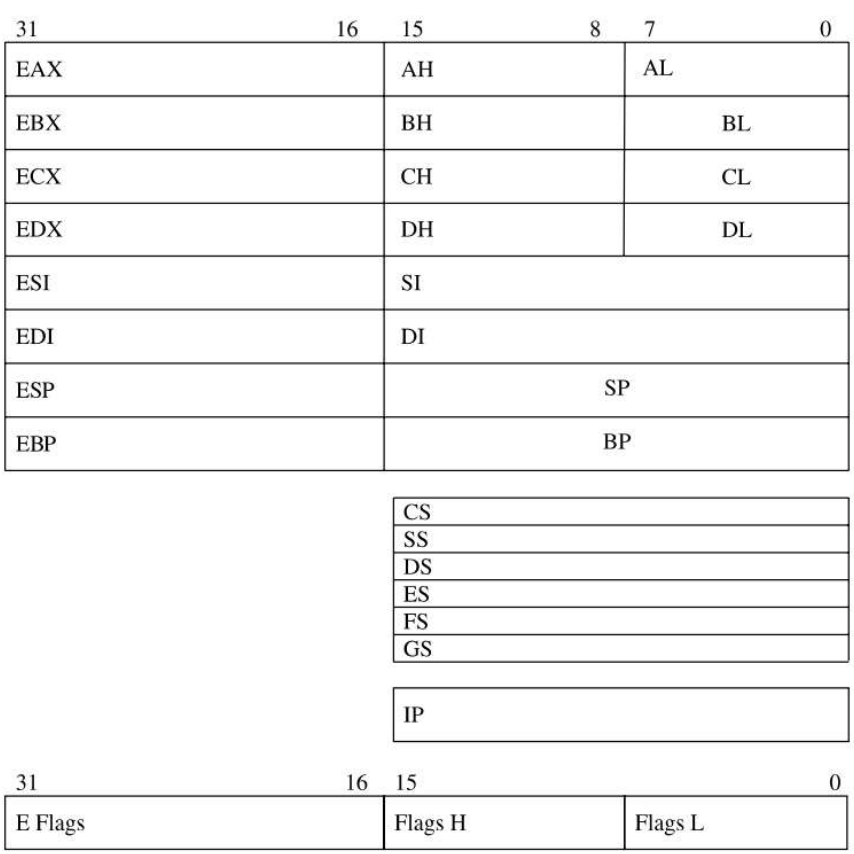


Figure 3.6 The base register sets of the *X86* programming model

EXAMPLE: THE X86 FAMILY

- It **consists of three register groups.**
- These are the **general purpose registers**, the **segment registers**, and the **instruction pointer (program counter)** and **the flag register**.
- The **first set** consists of **general purpose registers A, B, C, D, SI (source index), DI (destination index), SP (stack pointer), and BP (base pointer)**.
- It should be noted that in **naming these registers**, we used **X** to indicate **eXtended.**
- The **second set** of **registers** consists of **CS (code segment), SS (stack segment), and four data segment registers DS, ES, FS, and GS**.
- The **third set** of **registers consists** of the **instruction pointer** (**program counter**) and the **flags (status) register**. The **latter is shown** in **Figure 3.7**.

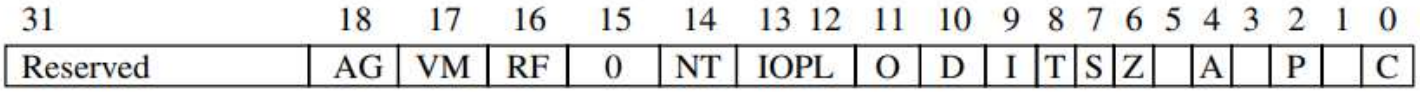


Figure 3.7 The *X86* flag register

EXAMPLE: THE X86 FAMILY

- **Among** the **status bits** shown in **Figure 3.7**, **the first five** are **identical to those bits introduced** as **early** as in the **8085 8-bit microprocessor**. **The next 6– 11 bits** are **identical to those introduced in the 8086**.
- The **flags in the bits 12– 14** were **introduced in the 80286** while **the 16 –17 bits** were **introduced in the 80386**. The **flag in bit 18** was **introduced in the 80486**. **Table 3.8 shows the meaning of those flags**.
- In the **X86 family an instruction can perform an operation on one or two operands**.
- In **two-operand instructions,** the **second operand** can be **immediate data** in **2's complement format**. **Data transfer, arithmetic and logical instructions** can act **on immediate data, registers, or memory locations.**

TABLE 3.8 ***X86*** **Status Flags**

Flag	Meaning	Processor	Flag	Meaning	Processor
C	Carry	All	P	Parity	All
A	Auxiliary	All	Z	Zero	All
S	Sign	All	T	Trap	All
I	Interrupt	All	D	Direction	All
O	Overflow	All	IOPL	I/O privilege level	286
NT	Nested task	286	RF	Resume	386
VM	Virtual mode	386	AC	Alignment check	486

EXAMPLE: THE X86 FAMILY

- In the **X86 family**, **direct** and **indirect memory addressing** can be **used**.
- In **direct addressing**, a **displacement address** **consisting of a 8-, 16-, or 32-bit word** is used as the **logical address**.
- This **logical address** is **added to** the **shifted contents** of the **segment register** (**segment base address**) **to give a physical memory address.**
- **Figure 3.8 illustrates the direct addressing process.**

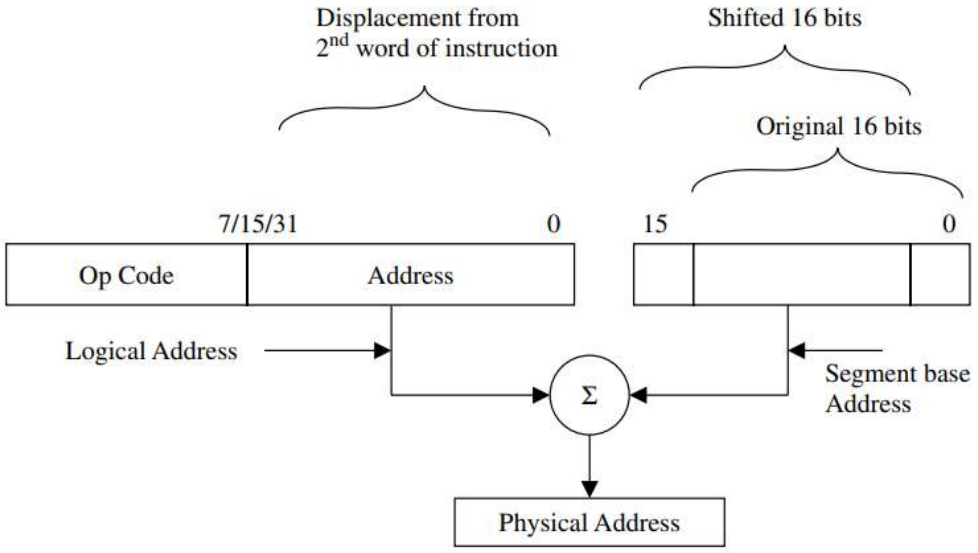


Figure 3.8 Direct addressing in the *X86* family

EXAMPLE: THE X86 FAMILY

- **Address indirection** in the **X86 family** can be **obtained using** the **content of a base pointer register (BPR), the content of an index register,** or the **sum of a base register and** an **index register**.
- **Figure 3.9 illustrates** indirect addressing using the **BPR**. The **X86 family of processors** defines a **number of instruction types**.
- Using **the naming convention** introduced before, **these instruction types are data movement, arithmetic and logic, and sequencing (control transfer**).
- **In addition**, the **X86 family defines** other **instruction types** such as **string manipulation**, **bit manipulation**, and **high-level language support.**

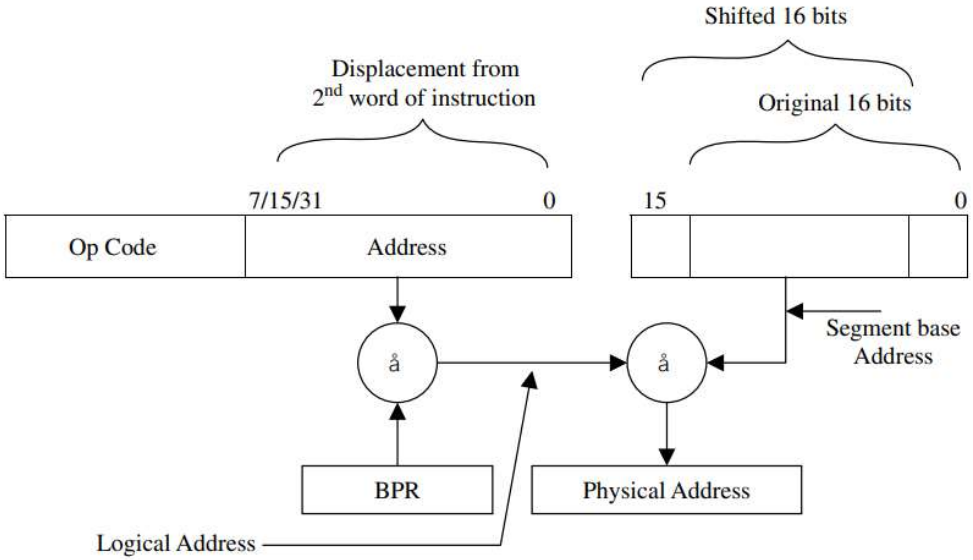


Figure 3.9 Indirect addressing using BPR in the *X86* family

EXAMPLE: THE X86 FAMILY

- **Data movement instructions** in the **X86 family** include **mainly four subtypes**.
- These are the **general-purpose**, **accumulator-specific**, **address-object**, and **flag instructions**. A sample of these instructions is shown in **Table 3.9**.

TABLE 3.9 Sample of the *X86* Data Movement Instructions

Mnemonic	Operation	Subtype
MOV	Move source to destination	General purpose
POP	Pop source from stack	General purpose
POPA	Pop all	General purpose
PUSH	Push source onto stack	General purpose
PUSHA	Push all	General purpose
XCHG	Exchange source with destination	General purpose
IN	Input to accumulator	Accumulator
OUT	Output from accumulator	Accumulator
XLAT	Table lookup to translate byte	Accumulator
LEA	Load effective address in register	Address-object
LMSW	Load machine status word	Address-object
SMSW	Store machine status word	Address-object
POPF	Pop flags off stack	Flag
PUSHF	Push flags onto stack	Flag

EXAMPLE: THE X86 FAMILY

- **Arithmetic and logic instructions** in the **X86 family** include **mainly five subtypes.**
- These are **addition, subtraction, multiplication, division, and logic instructions.**
- A sample of the **arithmetic instructions** is shown in **Table 3.10**.

TABLE 3.10 Sample of the *X86* Arithmetic Instructions

Mnemonic	Operation	Subtype
ADD	Add source to destination	Addition
ADC	Add source to destination with carry	Addition
INC	Increment operand by 1	Addition
SUB	Subtract source from destination	Subtraction
SBB	Subtract source from destination with borrow	Subtraction
DEC	Decrement operand by 1	Subtraction
MUL	Unsigned multiply source by destination	Multiply
IMUL	Signed multiply source by destination	Multiply
DIV	Unsigned division accumulator by source	Division
IDIV	Signed division accumulator by source	Division

EXAMPLE: THE X86 FAMILY

- **Logic instructions include** the **typical AND, OR, NOT, XOR, and TEST**.
- The latter **performs a logic compare** of **the source and the destination** and **sets the flags accordingly.**
- In addition, the **X86 family** has a **set of shift and rotate instructions**.
- A **sample of these** is shown in **Table 3.11**.

TABLE 3.11 Sample of the *X86* Shift and Rotate Instructions

Mnemonic	Operation
ROR	Rotate right
ROL	Rotate left
RCL	Rotate left through carry
RCR	Rotate right through carry
SAR	Arithmetic shift right
SAL	Arithmetic shift left
SHR	Logic shift right
SHL	Logic shift left

EXAMPLE: THE X86 FAMILY

- **Control transfer instructions** in the **X86 family** include **mainly four subtypes**.
- These are **conditional, iteration, interrupt, and unconditional**.
- A **sample of these instructions** is shown in **Table 3.12**.

TABLE 3.12 Sample of the *X86* Control Transfer Instructions

Mnemonic	Operation	Subtype
SET	Set byte to true or false based on condition	Conditional
JS	Jump if sign	Conditional
LOOP	Loop if CX does not equal zero	Iteration
LOOPE	Loop if CX does not equal zero & ZF = 1	Iteration
INT	Interrupt	Interrupt
IRET	Interrupt return	Interrupt
JMP	Jump unconditional	Unconditional
RET	Return from procedure	Unconditional

EXAMPLE: THE X86 FAMILY

- **Processor control instructions** in the **X86 family** include **mainly three subtypes**.
- These are **external synchronization, flag manipulation, and general control instructions.**
- A **sample** of **these instructions** is shown in **Table 3.13**.
- Having **introduced the basic features** of the **instruction set** of the **X86 processor family**, we now move on to present a **number of programming examples** to **show how the instruction set can be used.**
- The **examples presented** are the same as those **presented** at the end of **Chapter 2.**

TABLE 3.13 Sample of the *X86* Processor Control Instructions

Mnemonic	Operation	Subtype
HLT	Halt	External sync
LOCK	Lock the bus	External sync
CLC	Clear carry flag	Flag
CLI	Clear interrupt flag	Flag
STI	Set interrupt flag	Flag
INVD	Invalidate data cache	General control

EXAMPLE: THE X86 FAMILY

- **Example 3: Adding 100 numbers stored at consecutive memory locations starting at location 1000, the results should be stored in memory location 2000.**
- **LIST** is defined as an array of **N elements** each of **size byte**.
- **FLAG** is a memory variable used to indicate whether the list has been sorted or not.
- The **register CX** is used as a **counter** with the **Loop instruction**.
- The **Loop instruction decrements** the **CX register** and **branch if the result is not zero.**
- The **addressing mode** used to **access the array List [BX + 1]** is called **based addressing mode**.
- It should be noted that since we are using **BX** and **BX + 1** the **CX counter** is **loaded** with the value **999** in order **not to exceed the list**.

EXAMPLE: THE X86 FAMILY

```
        MOV CX, 1000 - 1 ; Counter = CX (1000 - 1)
        MOV BX, Offset LIST ; BX pointer to LIST
        CALL SORT.....

SORT PROC NEAR
  Again: MOV FLAG, 0 ; FLAG 0
   Next: MOV AL, [BX]
        CMP AL, [BX + 1] ;Compare current and next values
        JLE Skip ;Branch if current , next values
        XCHG AL, [BX + 1] ;If not, Swap the contents of the
        MOV [BX + 1], AL ;current location with the next one
        MOV FLAG, 1 ;Indicate the swap
   Skip: INC BX ; BX = BX + 1
        LOOP Next ;Go to next value
        CMP FLAG, 1 ;Was there any swap
        JE Again ;If yes Repeat process
        RET
SORT ENDP
```

EXAMPLE: THE X86 FAMILY

- **Example 4:** Here we implement the **SEARCH algorithm** in the **8086 instruction set**.
- **LIST** is defined as an array of **N elements** each of **size word**.
- **FLAG** is a **memory variable used** to indicate **whether the list** has been **sorted or not**.
- The **register CX** is used as a **counter** with the **loop instruction**.
- The **Loop instruction decrements** the **CX register and branch if the result is not zero**.
- The addressing mode used to access the array **List [BX + 1]** is **called based addressing mode**

```
MOV CX, 1000 ; Counter = CX + 1000
MOV BX, Offset LIST ; BX pointer to LIST
MOV SI, 0 ; SI used as an index
MOV AX, VAL ; AX <- VAL
CALL SEARCH
        ..... ; Test FLAG to check whether value found
```

EXAMPLE: THE X86 FAMILY

```
SEARCH PROC NEAR

        MOV FLAG, 0 ; FLAG <-0
Next:   CMP AX, [BX+SI] ;Compare current value to VAL
        JE Found ;Branch if equal
        ADD SI, 2 ; SI SI+2, next value
        LOOP Next ;Go to next value
        JMP Not_Found
Found:  MOV FLAG, 1 ;Indicate value found
        MOV POSITION, SI ;Return index of value in List
Not_Found: RET

SEARCH ENDP
```

EXAMPLE: THE X86 FAMILY

Example 5: This is the same as Example 4 but using the stack features of the X86.

```
PUSH DS ;See Table 3.9
MOV CX, 1000 ;Counter = CX =1000
MOV BX, OFFSET LIST ;Point to beginning of
LIST
PUSH BX
PUSH VAL ;VAL is a word variable
CALL SEARCH ;Test FLAG to check whether
value found  ;If found get index from SI register
using  POP SI
.....
```

```
SEARCH PROC NEAR
POP TEMP ;Save IP
POP AX ;AX VAL. Value to search for
POP SI ;SI OFFSET LIST and let BX =SI
POP ES ;Make ES = DS (See Table)
CLD ;Set auto-increment mode
REPNE SCASW ;Scan LIST for value in AX if not found;
        increment SI by 2, decrement CX and if; not zero scan
        next location in LIST. ;If occurrence found Zero flag
        is set
JNZ Not_Found ;If value not branch to Not_Found?
MOV FLAG, 1 ;Yes
SUB SI, BX
PUSH SI ;Save position
Not_Found: PUSH TEMP ;Restore IP
RET
SEARCH ENDP
```

Thank you