Chapter Four

Instruction Set of 8086 Microprocessor

By: Gemmachis T.

College of Computing and Informatics

Topics to be covered.....

- Introduction
- Instruction Set Architecture (ISA)
- Instruction Set
- Classifications of Instruction Set
  - Data Transfer Instructions
  - Arithmetic Instructions
  - Bit Manipulation Instructions
  - Program Execution Transfer Instructions
  - String Manipulation Instructions
  - Processor Control Instructions

Introduction

- **Microprocessor** is an integrated circuit (IC) that contains all the functions of a central processing unit of a computer.
- A microprocessor executes a collection of machine instructions that tell the processor what to do.
- The instruction set architecture (**ISA**) is the set of basic instructions that a **processor** understands.
- The instruction set is a portion of what makes up an architecture.
- Based on the instructions, a microprocessor does **three** basic things:
  - Using its ALU (Arithmetic/Logic Unit), a microprocessor can perform mathematical operations like addition, subtraction, multiplication and division. Modern microprocessors contain complete floating-point processors that can perform extremely sophisticated operations on large floating-point numbers.
  - A microprocessor can move data from one memory location to another.
  - A microprocessor can make decisions and jump to a new set of instructions based on those decisions.

Instruction set architecture (ISA)

- ISA is an abstract model of a computer that defines how the CPU is controlled by the software.
- A realization of an ISA is called an *implementation*.
- An ISA permits multiple implementations that may vary in performance, physical size, and monetary cost (among other things); because the ISA serves as the interface between **software** and **hardware**.
- Software that has been written for an ISA can run on different implementations of the same ISA.
- This has enabled binary compatibility between different generations of computers to be easily achieved, and the development of computer families.

Cont'd...

- Both of these developments have helped to lower the cost of computers and to increase their applicability.
- For these reasons, the ISA is one of the most important abstractions in computing today.
- An ISA defines everything a machine language programmer needs to know in order to program a computer.
- What an ISA defines differs between ISAs. In general ISAs define:
  - The supported **data types**
  - What state there is (such as the **main memory** and **registers**) and their semantics (such as the memory consistency and addressing modes),
  - The instruction set (the set of machine instructions that comprises a computer's machine language), and
  - The input/output model.

Cont'd…

- An **instruction set architecture** is distinguished from a **microarchitecture**, which is the set of processor design techniques used, in a particular processor, to implement the instruction set.
- Processors with different microarchitectures can share a common instruction set.
- For example, the **Intel Pentium** and the **Advanced Micro Devices (AMD) Athlon** implement nearly identical versions of the x86 instruction set but have radically different internal designs.

Classification of ISAs

- An ISA may be classified in a number of different ways.
- A common classification is by **architectural *complexity***.
- Based on architectural complexity, there are two types of ISA:
  1. A **complex instruction set computer** (CISC): this ISA has many specialized instructions, some of which may only be rarely used in practical programs.
  2. **A reduced instruction set computer (RISC**: this ISA simplifies the processor by efficiently implementing only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines, having their resulting additional processor execution time offset by infrequent use.

Instruction Set

- The **instruction set** is part of a computer that pertains to programming, which is basically machine language.
- An instruction is a binary pattern designed inside a microprocessor to perform a specific function.
- The entire group of instructions that a microprocessor supports is called **Instruction Set**.

- The instruction set provides **commands** to the processor, to tell it what it needs to do.
- The instruction set consists of addressing modes, instructions, native data types, registers, memory architecture, interrupt, exception handling, and external I/O.
- 8086 Microprocessor has more than **20,000** instructions.
- Hence this microprocessor is called CISC processor

Cont'd...

Classification of Instruction Set

- Data Transfer Instructions
- Arithmetic Instructions
- Bit Manipulation Instructions
- Program Execution Transfer Instructions
- String Instructions
- Processor Control Instructions

Data Transfer Instructions

- These instructions are used to transfer data from source (src) to destination (des).
- The operand can be a constant, memory location, register or I/O port address.

MOV des, src:

- MOV instruction is used to copy the byte or word from the provided source to the provided destination
- src operand can be register, memory location or immediate operand.
- des can be register or memory operand.
- Both Src and Des cannot be memory location at the same time.

Example:

```
MOV CX, 037AH;
MOV AL, BL;
MOV BX, [0301H];
```

Cont'd…

PUSH Operand:

- PUSH instruction is used to put a word at the top of the stack
- It pushes the operand into top of stack.

Example:

```
PUSH BX;
```

POP des:

- POP instruction is used to get a word from the top of the stack to the provided location.
- It pops the operand from top of stack to des.
- Des can be a general-purpose register, segment register (except CS) or memory location.

Example:

```
POP AX;
```

Cont'd…

XCHG des, src:

- XCHG is used to exchange the data from two locations.
- This instruction exchanges src with des.
- It cannot exchange two memory locations directly.

Example:

```
XCHG DX, AX;
```

IN Accumulator, Port Address:

- IN instruction is used to read a byte or word from the provided port to the accumulator.
- It transfers the operand from specified port to accumulator register.

Example:

```
IN AX, 0028 H
```

Cont'd...

OUT Port Address, Accumulator:

- OUT instruction is used to send out a byte or word from the accumulator to the provided port.
- It transfers the operand from accumulator to specified port.

Example:

```
OUT 0028H, AX;
```

LEA Register, src:

- LEA (Load Effective Address) instruction is used to load the address of operand into the provided register.
- It loads a 16-bit register with the offset address of the data specified by the src.

Example:

```
LEA BX, [DI];
```

- This instruction loads the contents of DI (offset) into the BX register.

Cont'd...

LDS des, src:

- LDS (Load Data Segment) instruction is used to load DS register and other provided register from the memory
- It loads 32-bit pointer from memory source to destination register and DS.
- The offset is placed in the destination register and the segment is placed in DS.
- To use this instruction the word at the lower memory address must contain the **offset** and the word at the higher address must contain the **segment**.

Example:

```
LDS BX, [0301H];
```

Cont'd...

LES Des, Src:

- LES (Load Extra Segment) instruction is used to load ES register and other provided register from the memory.
- It loads 32-bit pointer from memory source to destination register and ES.
- The offset is placed in the destination register and the segment is placed in ES.
- This instruction is very similar to LDS except that it initializes ES instead of DS.

Example:

```
LES BX, [0301H];
```

Cont'd…

- The following are instructions to transfer flag registers.
- These instructions **do not** accept any operands
  - **LAHF:** It copies the lower byte of flag register to AH.
  - **SAHF:** It copies the contents of AH to lower byte of flag register.
  - **PUSHF:** Pushes flag register to top of stack.
  - **POPF:** Pops the stack top to flag register.

Arithmetic Instructions

- These instructions are used to perform arithmetic operations like addition, subtraction, multiplication, division, etc.

ADD des, src:

- ADD instruction is used to add the provided byte to byte/word to word.
- This instruction affects AF, CF, OF, PF, SF, ZF flags.

Example:

```
ADD AL, 74H;
ADD DX, AX;
ADD AX, [BX];
```

ADC Des, Src:

- ADC (ADD with Cary) adds the two operands with CF.
- It affects AF, CF, OF, PF, SF, ZF flags.

Example:

- `ADC AL, 74H;`
- `ADC DX, AX;`
- `ADC AX, [BX];`

Cont'd...

SUB Des, Src:

- SUB instruction is used subtracts a byte from byte or a word from word.
- It effects AF, CF, OF, PF, SF, ZF flags.
- For subtraction, CF acts as borrow flag.

Example:
- SUB AL, 74H;
- SUB DX, AX;
- SUB AX, [BX];

SBB Des, Src:

- SBB (Subtract with Borrow) subtracts the two operands and also the borrow from the result.
- It effects AF, CF, OF, PF, SF, ZF flags.

Example:
- SBB AL, 74H
- SBB DX, AX
- SBB AX, [BX]

Cont'd...

INC Src:

- It increments the byte or word by one.
- The operand can be a register or memory location.
- It effects AF, OF, PF, SF, ZF flags.
- CF is not effected.

Example:

```
INC AX;
```

DEC Src:

- It decrements the byte or word by one.
- The operand can be a register or memory location.
- It effects AF, OF, PF, SF, ZF flags.
- CF is not effected.

Example:

```
DEC AX;
```

Cont'd…

AAA (ASCII Adjust after Addition):

- The data entered from the terminal is in ASCII format.
- In ASCII, 0 – 9 are represented by 30H – 39H.
- This instruction allows us to add the ASCII codes
- This instruction does not have operands.

Other ASCII Instructions:

- **AAS** (ASCII Adjust after Subtraction)
- **AAM** (ASCII Adjust after Multiplication)
- **AAD** (ASCII Adjust Before Division)

Cont'd...

DAA (Decimal Adjust after Addition)

- It is used to make sure that the result of adding two BCD numbers is adjusted to be a correct BCD number.
- It only works on AL register.

DAS (Decimal Adjust after Subtraction)

- It is used to make sure that the result of subtracting two BCD numbers is adjusted to be a correct BCD number.
- It only works on AL register.

NEG Src:

- It creates 2's complement of a given number.
- That means, it changes the sign of a number.

Example:

```
NEG AL;
```

Cont'd…

CMP des, src:

- It compares two specified bytes or words.
- The src and des can be a constant, register or memory location.
- Both operands cannot be a memory location at the same time.
- The comparison is done simply by internally subtracting the source from destination.
- The value of source and destination does not change, but the flags are modified to indicate the result.

Cont'd...

MUL src:

- It is an unsigned multiplication instruction.
- It used to multiply unsigned byte by byte/word by word.
- It multiplies two bytes to produce a word or two words to produce a double word.

  AX = AL * Src — Src=8-bit register/memory

  DX : AX = AX * Src — Src=16-bit register/memory

Example:

```
MUL BH;
MUL CX;
```

- This instruction assumes one of the operand in AL or AX.
- Src can be a register or memory location.

IMUL Src:

- It is a signed multiplication instruction.

Example:

```
IMUL BH;
IMUL CX;
```

Cont'd...

DIV Src:

- It is an unsigned division instruction.
- It divides word by byte or double word by word.
- The operand is stored in AX, divisor is Src and the result is stored as:
- AH = remainder AL = quotient

Example:

- **DIV BL;**

AX = AX / 8-bit reg. ; AL = quotient ; AH = remainder

- **DIV CX;**

DX AX / 16-bit reg. ; AX = quotient ; DX = remainder

IDIV Src:

- It is a signed division instruction.
- It is used to divide the signed word by byte or signed double word by word.

Cont'd…

CBW (Convert Byte to Word):

- This instruction converts byte in AL to word in AX.
- The conversion is done by extending the sign bit of AL throughout AH.
- This instruction does not have any operand.

CWD (Convert Word to Double Word):

- This instruction converts word in AX to double word in DX:AX.
- The conversion is done by extending the sign bit of AX throughout DX.
- This instruction does not have any operand.

Bit Manipulation Instructions

- These instructions are used to perform operations where data bits are involved, i.e. operations like logical, shift, etc.
- These instructions are used at the bit level.
- These instructions can be used for:
  - Testing a zero bit
  - Set or reset a bit
  - Shift bits across registers

Cont'd…

NOT Src:

- It is used to invert each bit of a byte or word.
- It complements each bit of Src to produce **1's complement** of the specified operand.
- The operand can be a register or memory location.

Example:

```
NOT AL;
NOT DL;
```

Cont'd...

AND des, src:

- It is used to multiply each bit in a byte/word with the corresponding bit in another byte/word.
- It performs AND operation of des and src.
- Src can be immediate number, register or memory location.
- Des can be register or memory location.
- Both operands cannot be memory locations at the same time.
- CF and OF become zero after the operation.
- PF, SF and ZF are updated.

Example:

```
AND AX, 0010H;
AND AH, DH;
```

Cont'd…

OR des, Src:

- It is used for adding each bit in a byte/word with the corresponding bit in another byte/word.
- It performs OR operation of des and src.
- Src can be immediate number, register or memory location.
- Des can be register or memory location.
- Both operands cannot be memory locations at the same time.
- CF and OF become zero after the operation.
- PF, SF and ZF are updated.

Example:

```
AND AX, 0010H;
AND AH, DH;
```

Cont'd…

XOR des, src:

- It is used to perform Exclusive-OR operation over each bit in a byte/word with the corresponding bit in another byte/word.
- It performs XOR operation of des and src.
- Src can be immediate number, register or memory location.
- Des can be register or memory location.
- Both operands cannot be memory locations at the same time.
- CF and OF become zero after the operation.
- PF, SF and ZF are updated.

Example:

```
XOR AL, BL;
XOR AX, BX;
XOR AX, 0010H;
```

Cont'd…

SHL des, count:

- It is used to shift bits of a byte/word towards left and put zero(S) in LSBs.
- It shift bits of byte or word left, by count.
- It puts zero(s) in LSBs.
- MSB is shifted into carry flag.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Example:

```
SHL AX, BL;
```

Cont'd…

SHR Des, Count:

- It is used to shift bits of a byte/word towards the right and put zero(S) in MSBs.
- It shift bits of byte or word right, by count.
- It puts zero(s) in MSBs.
- LSB is shifted into carry flag.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Example:

```
SHR AL, 04;
```

Cont'd…

ROL des, count:

- It is used to rotate bits of byte/word towards the left, i.e. MSB to LSB and to Carry Flag [CF].
- It rotates bits of byte or word left, by count.
- MSB is transferred to LSB and also to CF.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Example:

```
ROL BX, 06;
```

Cont'd…

ROR des, count:

- It is used to rotate bits of byte/word towards the right, i.e. LSB to MSB and to Carry Flag [CF].
- It rotates bits of byte or word right, by count.
- LSB is transferred to MSB and also to CF.
- If the number of bits desired to be shifted is 1, then the immediate number 1 can be written in Count.
- However, if the number of bits to be shifted is more than 1, then the count is put in CL register.

Example:

```
ROR BL, CL
```

Program Execution Transfer Instructions (Branch and Loop Instructions)

- These instructions cause change in the sequence of the execution of instruction.
- These instructions are used to transfer/branch the instructions during an execution
- This change can be through a condition or sometimes unconditional.
- The conditions are represented by flags.

Cont'd…

CALL Des:

- This instruction is used to call a subroutine or function or procedure.
- The address of next instruction after CALL is saved onto stack.

RET:

- It returns the control from procedure to calling program.
- Every CALL instruction should have a RET.

- **JMP des (Unconditional jump):**
  - This instruction is used for unconditional jump from one place to another.

Example:

```
Call 2050H;
RET
JMP 2050H;
```

Cont'd…

Loop des:

- This is a looping instruction.
- The number of times looping is required is placed in the CX register.
- With each iteration, the contents of CX are decremented.
- ZF is checked whether to loop again or not.

Example:

```
LOOP 2050
```

Jxx des (Conditional Jump):

- All the conditional jumps follow some conditional statements or any instruction that affects the flag.
- These instructions only execute when the specified condition is true.

Opcode	Operand	Explanation	Example
JC	address	jump if CF = 1	JC 2050
JNC	address	jump if CF = 0	JNC 2050
JZ	address	jump if ZF = 1	JZ 2050
JNZ	address	jump if ZF = 0	JNZ 2050
JO	address	jump if OF = 1	JO 2050
JNO	address	jump if OF = 0	JNO 2050
JP	address	jump if PF = 1	JP 2050
JNP	address	jump if PF = 0	JNP 2050
JPE	address	jump if PF = 1	JPE 2050
JPO	address	jump if PF = 0	JPO 2050
JS	address	jump if SF = 1	JS 2050
JNS	address	jump if SF = 0	JNS 2050
JA	address	jump if CF=0 and ZF=0	JA 2050
JNBE	address	jump if CF=0 and ZF=0	JNBE 2050

Opcode	Operand	Explanation	Example
JAE	address	jump if CF=0	JAE 2050
JNB	address	jump if CF=0	JNB 2050
JBE	address	jump if CF = 1 or ZF = 1	JBE 2050
JNA	address	jump if CF = 1 or ZF = 1	JNA 2050
JE	address	jump if ZF = 1	JE 2050
JG	address	jump if ZF = 0 and SF = OF	JG 2050
JNLE	address	jump if ZF = 0 and SF = OF	JNLE 2050
JGE	address	jump if SF = OF	JGE 2050
JNL	address	jump if SF = OF	JNL 2050
JL	address	jump if SF != OF	JL 2050
JNGE	address	jump if SF != OF	JNGE 2050
JLE	address	jump if ZF = 1 or SF != OF	JLE 2050
JNG	address	jump if ZF = 1 or SF != OF	JNG 2050
JCXZ	address	jump if CX = 0	JCXZ 2050

Opcode	Operand	Explanation	Example
LOOPE	address	loop while ZF = 1 and CX = 0	LOOPE 2050
LOOPZ	address	loop while ZF = 1 and CX = 0	LOOPZ 2050
LOOPNE	address	loop while ZF = 0 and CX = 0	LOOPNE 2050
LOOPNZ	address	loop while ZF = 0 and CX = 0	LOOPNZ 2050

String Manipulation Instructions

- String is s **series** of data byte or word available in memory at consecutive locations.
- It is either referred as **byte** string or **word** string.
- Their memory is always allocated in a **sequential** order.
- Instructions used to manipulate strings are called **string manipulation instructions.**
- String in assembly language is just a sequentially stored bytes or words.
- It is a group of bytes/words and their memory is always allocated in a sequential order.
- There are very strong set of string instructions in 8086.
- By using these string instructions, the size of the program is considerably reduced.

Cont'd…

CMPS/CMPSB/CMPSW des, src:

- They compare the string bytes or words.
- CMPSB compares byte at ES:DI with byte at DS:SI and sets flags
- CMPSW compares word at ES:DI with word at DS:SI and sets flags
- These instructions do not have operands.

Example:

```
CMPS
CMPSB
CMPSW
```

Cont'd…

SCAS/SCASB/SCASW string:

- These instructions scan a string.
- They compare the string with byte in AL or with word in AX.
- SCASB compares byte at ES:DI with AL and sets flags according to result.
- SCASW compares word at ES:DI with AX and sets flags.

Example:

```
SCAS
SCASB
SCASW
```

Cont'd…

MOVS / MOVSB / MOVSW:

- These instructions are used to move the byte/word from one string to another.
- They cause moving of byte or word from one string to another.
- In these instructions, the source string is in data segment and destination string is in extra segment.
- SI and DI store the offset values for source and destination index.
- MOVSB moves contents of byte given by DS:SI into ES:DI
- MOVSW moves contents of word given by DS:SI into ES:DI

Example:

```
MOVS
MOVSB
MOVSW
```

Cont'd…

REP/REPE/REPZ/REPNE/REPNZ (Repeat):

- These are an instruction prefix.
- REP Used to repeat the given instruction till CX ≠ 0
- REPE repeat the given instruction while CX = 0
- REPZ repeat the given instruction while ZF = 1
- REPNE repeat the given instruction while CX != 0
- REPNZ repeat the given instruction while ZF = 0

Example:

```
REPE MOVSB STR1, STR2;
```

- It copies byte by byte contents.
- REP repeats the operation MOVSB until CX becomes zero.

Processor Control Instructions

- Process control instructions are the instructions which control the processor's action by setting(1) or resetting(0) the values of flag registers.
- These instructions control the processor itself.
- 8086 allows to control certain control flags that causes the processing in a certain direction
- These instructions are used for processor synchronization if more than one microprocessor attached.

Cont'd…

STC: It sets the carry flag CF to 1.

CLC: It clears the carry flag CF to 0.

CMC: It complements the carry flag CF.

STD: It sets the direction flag DF to 1.

- If it is set, string bytes are accessed from higher memory address to lower memory address.

CLD: It clears the direction flag DF to 0.

- If it is reset, the string bytes are accessed from lower memory address to higher memory address.

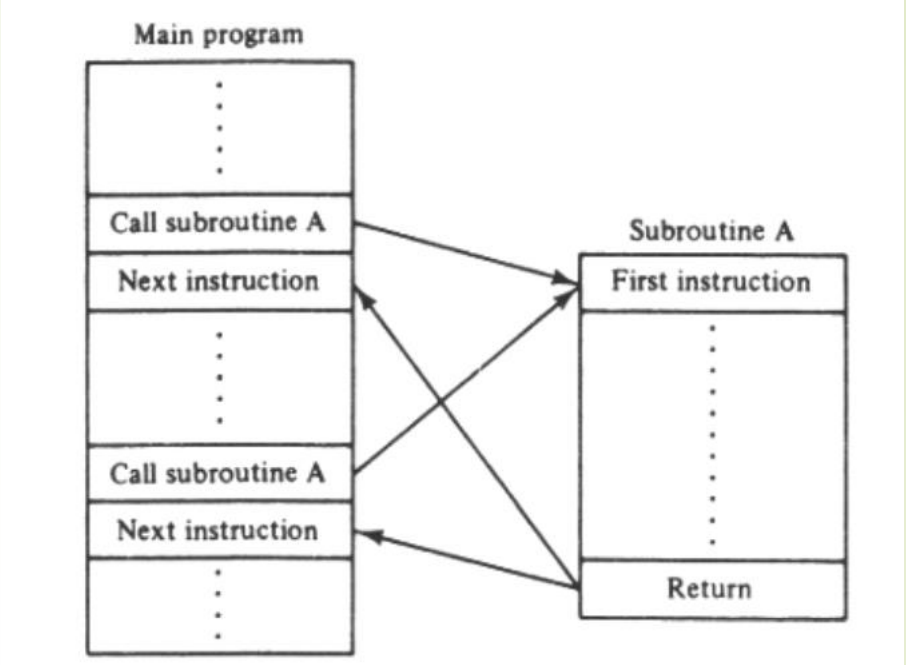
STI: It sets the interrupt flag to 1.

CLI: It clears/resets the interrupt flag to 0

~~~~~~~ The End ~~~~~~~
~~~~~~~

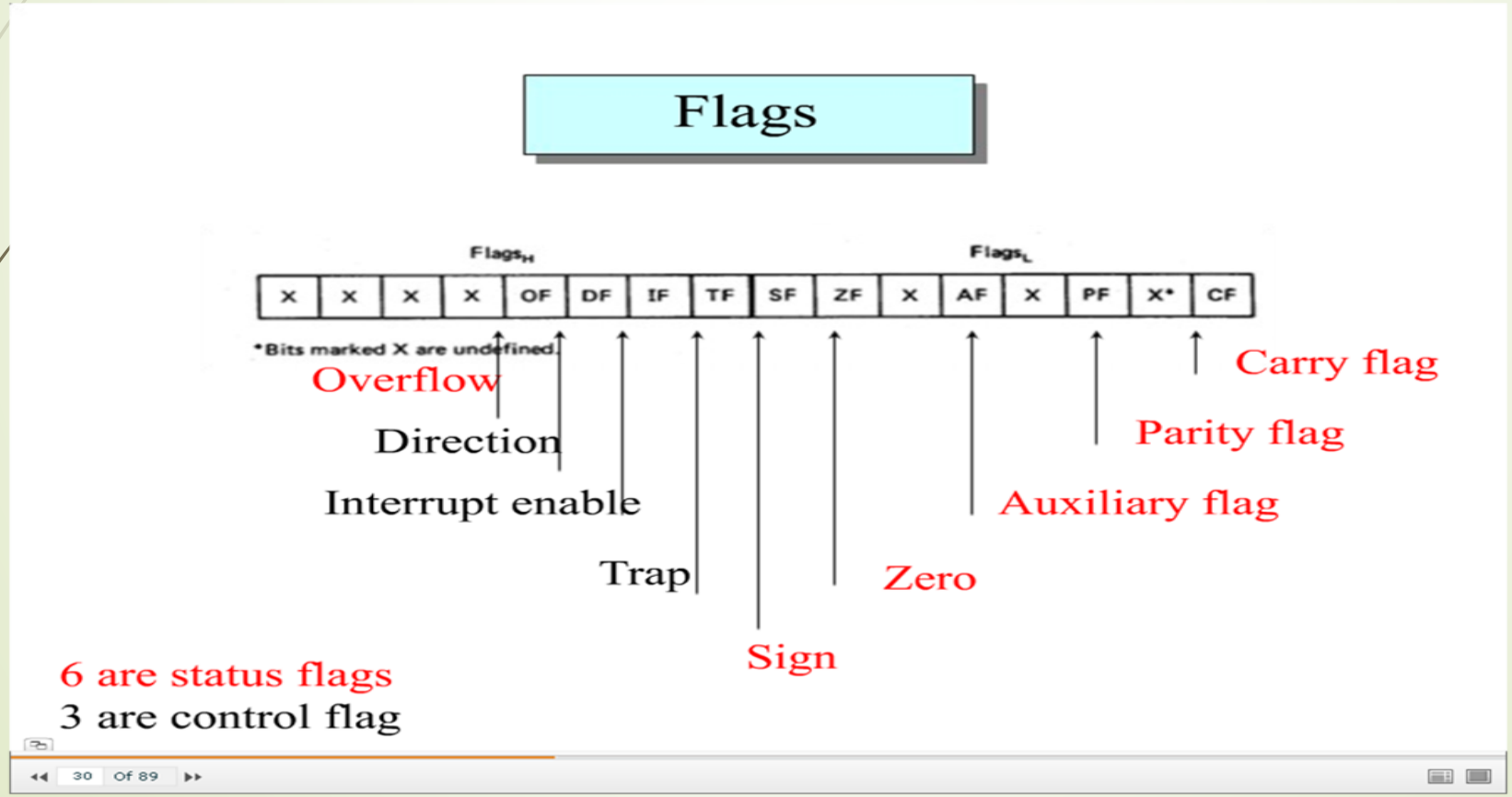
Subroutine/procedure

- A subroutine is a special segment of program that can be called for execution from any point in a program.
- The subroutine is written to provide a function that must be performed at various points in the main program.
- An assembly language subroutine is also referred to as a procedure.
- There are two basic instructions in the instruction set of the 8086 for subroutine handling:
- The call (CALL) and return (RET) instructions.
- CALL instruction is used to call the subroutine.
- RET instruction must be included at the end of the subroutine to initiate the return sequence to the main program environment.



Flag Registers

- **Flag Registers:** The 8086 Flag register contents indicate the results of computation in the *ALU*.



The Stack

- Stack is an area of memory for keeping temporary data.
- Stack is used by **CALL instruction** to keep return address for procedure, RET instruction gets this value from the stack and returns to that offset.
- Quite the same thing happens when INT instruction calls an interrupt, it stores in stack flag register, code segment and offset.
- IRET instruction is used to return from interrupt call.
- We can also use the stack to keep any other data,
- there are **two instructions** that work with the stack:
  - PUSH - stores 16-bit value in the stack.
  - POP - gets 16-bit value from the stack.
- PUSH and POP work with 16-bit values only!
- Note: PUSH immediate works only on 80186 CPU and later!
- The stack uses LIFO (Last In First Out) algorithm

```
Syntax for POP instruction:

POP REG
POP SREG
POP memory
REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, (except CS).

memory: [BX], [BX+SI+7], 16 bit variable, etc...
```

```
Syntax for PUSH instruction:

PUSH REG
PUSH SREG
PUSH memory
PUSH immediate
REG: AX, BX, CX, DX, DI, SI, BP, SP.

SREG: DS, ES, SS, CS.

memory: [BX], [BX+SI+7], 16 bit variable, etc...

immediate: 5, -24, 3Fh, 10001101b, etc...
```