



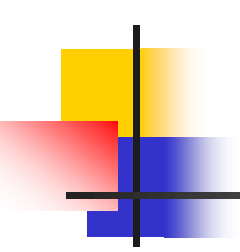
Assembly Language Programming

Assembly language programming is writing machine instructions in mnemonic form, using an assembler to convert these mnemonics into actual processor instructions and associated data.

- An assembly language is a low-level programming language for microprocessors and other programmable devices.

Features of Assembly Language Programming

- Assembly language is the most basic programming language available for any processor.
- Assembly languages generally lack high-level conveniences such as variables and functions.
- It has the same structures and set of commands as machine language, but it allows a programmer to use names instead of numbers.
- This language is still useful for programmers when speed is necessary or when they need to carry out an operation that is not possible in high-level languages.
- Some important features of assembly language programming are given below:
- Allows the programmer to use mnemonics when writing source code programs, like 'ADD' (addition), 'SUB' (subtraction), JMP (jump) etc
- Variables are represented by symbolic names, not as memory locations, like MOV A, here 'A' is the variable.
- Symbolic code
- error checking
- Changes can be quickly and easily incorporated with a re-assembly
- Programming aids are included for relocation and expression evaluation.

- 
-
- Variables are represented by symbolic names, not as memory locations, like MOV A, here 'A' is the variable.
 - Symbolic code
 - error checking
 - Changes can be quickly and easily incorporated with a re-assembly
 - Programming aids are included for relocation and expression evaluation.



Advantages of assembly language programming

- Easy to understand and use
- Easy to locate and correct errors
- Easy to modify
- No worry to address.
- Efficient than machine language programming

Disadvantages of assembly language programming

- The programmer requires knowledge of the processor architecture and instruction set.
- Machine language coding
 - Many instructions are required to achieve small tasks
- Source programs tend to be large and difficult to follow

Instructions set of Intel 8085 microprocessor according to assembly language programming

- ADD r:- add
- ADD M:- Add to Memory
- ADC r:-add with carry
- ADC M:-add with carry to memory
- LDI :- This instruction stands for load immediate and it means load' (put) into a particular named register a certain value named in the instruction
- LDA a:- load accumulator direct
- SLA: -means shift the contents of a register one place to the left.
- MOV:-means to move data from one place to desired place
- MOV M A:- means to move the result from memory to register A
- STA:- Store on register(accumulator or others)
- STA a: - store to accumulator.
- JMP- Jump to somewhere
- SUB r :- subtract
- SUB M:- subtract memory
- CY:-carry



Assembly Language Basic Syntax

An assembly program can be divided into three sections:

- The data section
- The bss section
- The text section

The data Section : The data section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names or buffer size etc. in this section. The syntax for declaring data section is:

section .data

The bss Section: The bss section is used for declaring variables. The syntax for declaring bss section is:

section .bss

The text section: The text section is used for keeping the actual code. This section must begin with the declaration global main, which tells the kernel where the program execution begins.

The syntax for declaring text section is:

section .text

global main

main:



Comments

Assembly language comment begins with a semicolon (;). It may contain any printable character including blank.

It can appear on a line by itself, like:

```
; This program displays a message on screen
```

or, on the same line along with an instruction, like:

```
add eax,ebx ; adds ebx to eax
```



Assembly Language Statements

Assembly Language programs consist of three types of statements:

- Executable instructions or instructions
 - Assembler directives or pseudo-ops
 - Macros
- The executable instructions or simply instructions tell the processor what to do. Each instruction consists of an operation code (opcode). Each executable instruction generates one machine language instruction.
 - The assembler directives or pseudo-ops tell the assembler about the various aspects of the assembly process.
 - These are non-executable and do not generate machine language instructions.
 - Macros are basically a text substitution mechanism.



Syntax of Assembly Language Statements

Assembly language statements are entered one statement per line. Each statement follows the following format:

[label] mnemonic [operands] [;comment]

- The fields in the square brackets are optional. A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic) which is to be executed, and the second are the operands or the parameters of the command. The following are some examples of typical assembly language statements:

INC COUNT ; Increment the memory variable COUNT

MOV TOTAL, 48 ; Transfer the value 48 in the
; memory variable TOTAL

ADD AH, BH ; Add the content of the
; BH register into the AH register

AND MASK1, 128 ; Perform AND operation on the
; variable MASK1 and 128

ADD MARKS, 10 ; Add 10 to the variable MARKS

MOV AL, 10 ; Transfer the value 10 to the AL register



The Hello World Program in Assembly

The following assembly language code displays the string 'Hello World' on the screen:

```
INC COUNT                                ; Increment the memory variable COUNT

section .text

global main ;must be declared for linker (ld)

main:                                    ;tells linker entry point

mov edx,len                             ;message length

mov ecx,msg                             ;message to write

mov ebx,1                               ;file descriptor (stdout)

mov eax,4                               ;system call number (sys_write)

int 0x80                                ;call kernel

mov eax,1                               ;system call number (sys_exit)

int 0x80                                ;call kernel

section .data

msg db 'Hello, world!', 0xa             ;our dear string

len equ $ - msg                        ;length of our dear string
```

When the above code is compiled and executed, it produces following result;

Hello, world!

The 80x86 MICROPROCESSOR

Historical Background

The Mechanical Age

The Electrical Age

1946

The first general purpose programmable electronic computer system - **ENIAC** (Electronics Numerical Integrator and Calculator) was developed

- 17,000 vacuum tubes
- 500 miles of wires
- weighted over 30 tons
- performed about 100,000 operations per second
- programmed by rewiring its circuits

1948

Development of the transistor (Bell Labs)

1958

Invention of the integrated circuit (Texas Instruments)

1960s

Development of digital integrated circuits

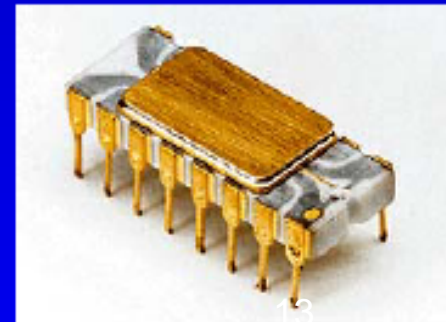
The Microprocessor Age

<http://www.computerhistory.org/exhibits/microprocessors/index.page>

November 15, 1971

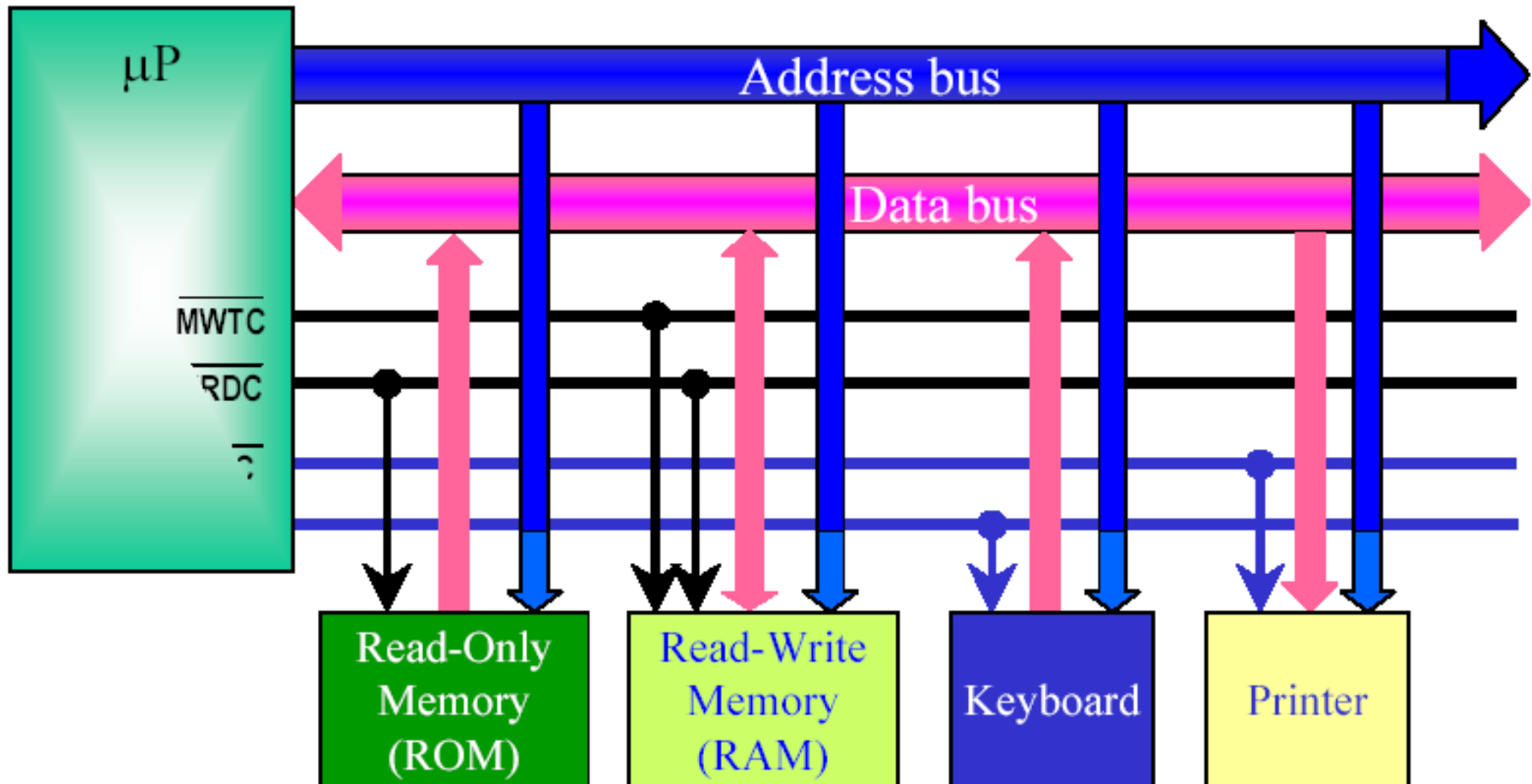
First Advertisement for Microprocessor Appears.

Intel 4004 has 2,250 transistors,
weighted less than an ounce,
handling data in 4-bit chunks, 45 instructions,
and could perform 50-60K instructions per second.



Year	name	Data size	memory size	#instructions	
1971	4004	4	4096 4-bit	45	first microprocessor
1973	8008	8	16K bytes	48	1st 8-bit μ P
1973	8080	8	64K bytes		10 times faster than 8008
1973	MC6800	8	64K bytes		1st Motorola μ P
1977	8085	8	64K bytes	246	Intel's most successful 8-bit general-purpose μ P due to its low cost
	Z80	8			Zilog's most successful microprocessor
1978	8086	8,16	1M bytes	>20,000	1st 16-bit μ P
1979	8088	8,16	1M bytes		prefetch instruction using cache
1981	IBM decided to use 8088 in its personal computer				
1983	80286	8,16	16M		
1986	80386	8,16,32	4G		
1989	80486	8,16,32	4G		
1993	Pentium	8,16,32	4G		
1995	Pentium Pro	64	64G		
1997	Pentium II	64	64G		
1999	Pentium III	?	?		
2000	Pentium 4	?	?		

Block Diagram of a Microprocessor-based computer system



Some buzz words



CISC – Complex Instruction Set Computers

- Refers to number and complexity of instructions
- Improvements was: Multiply and Divide
- The number of instruction increased from
 - 45 on 4004 to:
 - 246 on 8085
 - 20,000 on 8086 and 8088

RISC – Reduced Instruction Set Computer

- Executes one instruction per clock

Newer RISC - Superscaler Technology

- Execute more than one instruction per clock

Inside The 8088/8086

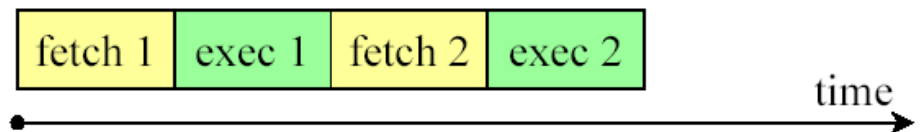
Concepts important to the internal operation
of 8088/8086

- Pipelining
- Registers

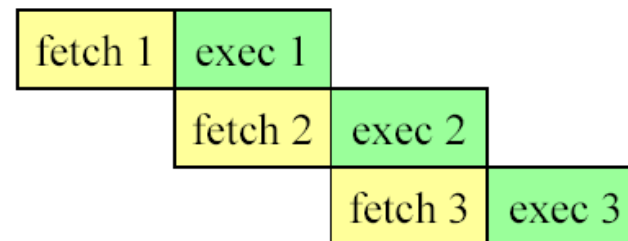
Inside The 8088/8086...*pipelining*

- **Pipelining**
 - Two ways to make CPU process information faster:
 - Increase the working frequency – technology dependent
 - Change the internal architecture of the CPU
 - Pipelining is to allow CPU to fetch and execute at the same time

non-pipelined 8085



pipelined 8086

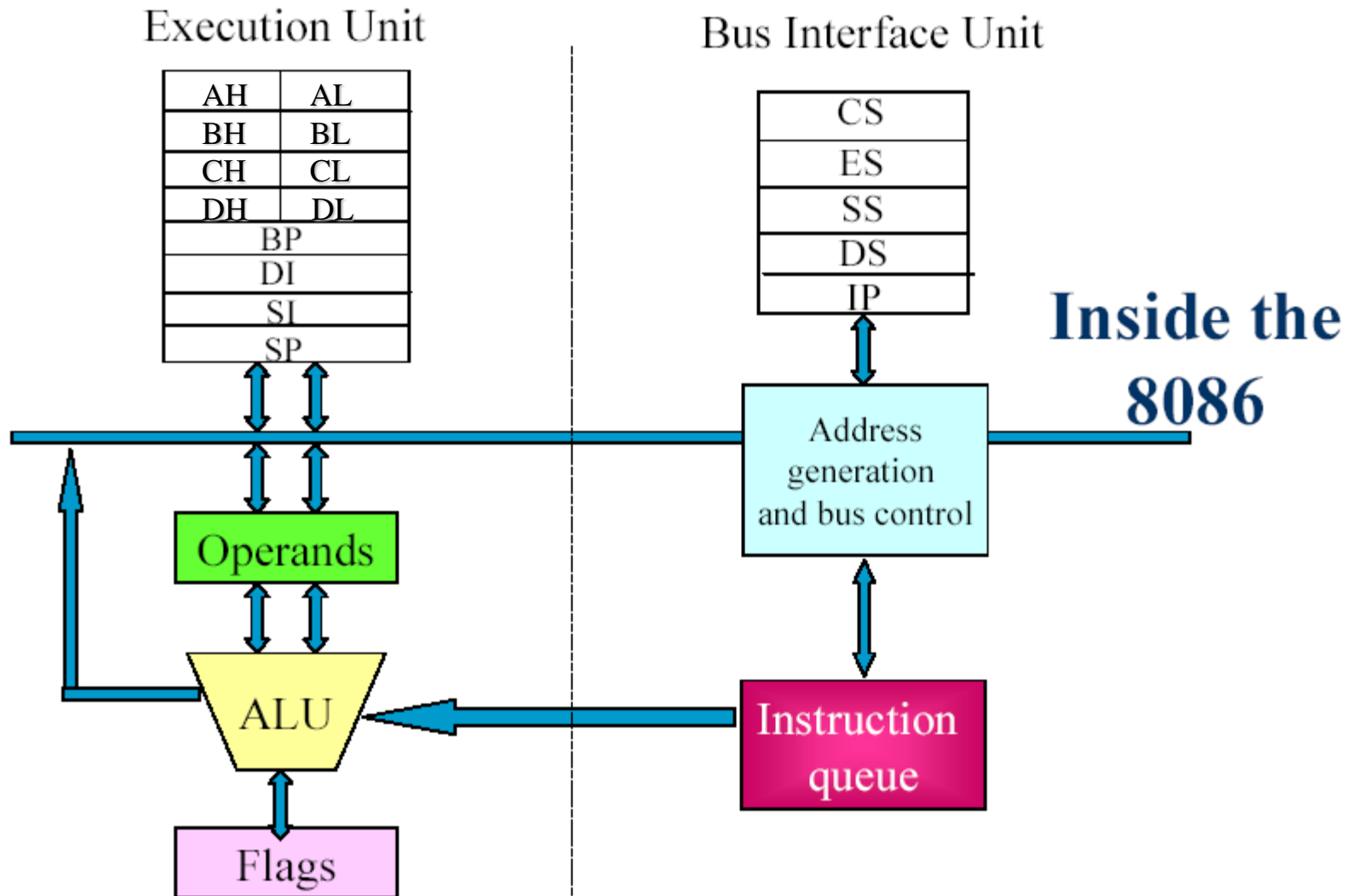


Inside The 8088/8086...*pipelining*

Intel implemented the concept of pipelining by splitting the internal structure of the 8088/8086 into two sections that works simultaneously:

- Execution Unit (EU) — executes instructions previously fetched
- Bus Interface Unit (BIU) — accesses memory and peripherals

Inside The 8088/8086



Registers

- Registers are used to store information temporarily
- 8086, 8088, 80286 contains 8-bit, 16-bit registers
- 80386, 80486, Pentium, Pentium Pro, and Pentium II contain 8, 16, and 32-bit registers.
- How about Pentium 4?

Overview

- Registers
 - General purpose registers (8)
 - Operands for logical and arithmetic operations
 - Operands for address calculations
 - Memory pointers
 - Segment registers (6)
 - EFLAGS register
 - The instruction pointer register
- The stack

Inside The 8088/8086...*registers*

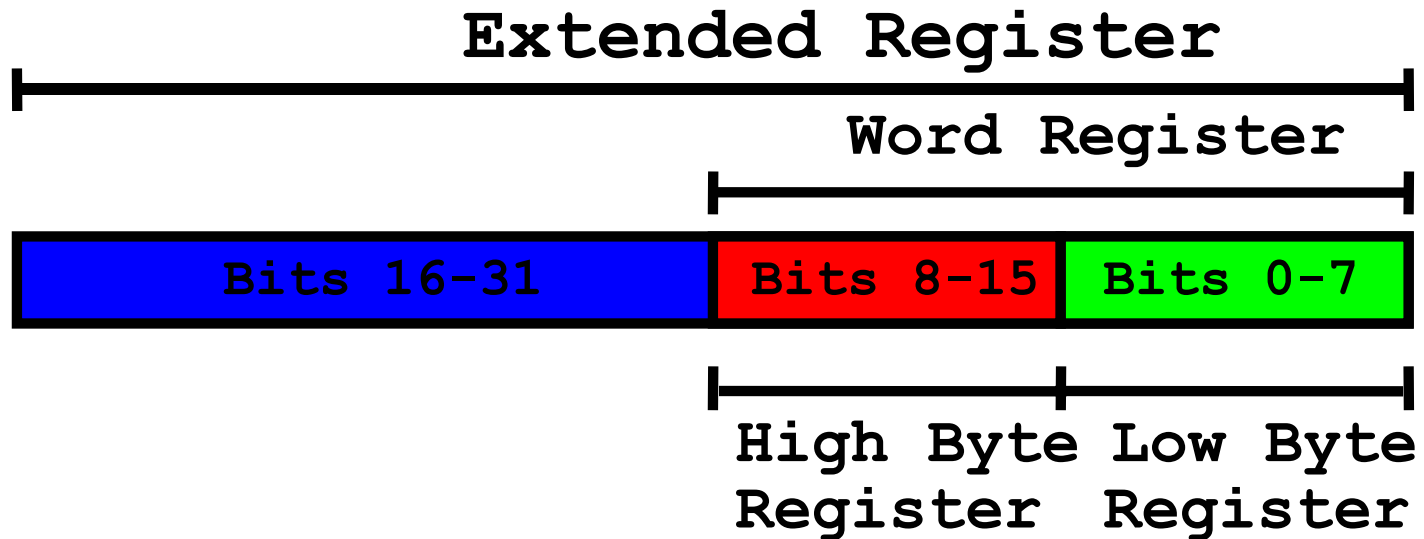
- Registers

- To store information temporarily

AX 16-bit register	
AH 8-bit reg.	AL 8-bit reg.

Category	Bits	Register Names
General	16	AX, BX, CX, DX
	8	AH, AL, BH, BL, CH, CL, DH, DL
Pointer	16	SP (stack pointer), BP (base pointer)
Index	16	SI (source index), DI (destination index)
Segment	16	CS (code segment), DS (data segment)
		SS (stack segment), ES (extra segment)
Instruction	16	IP (instruction pointer)
Flag	16	FR (flag register)

Anatomy of a Register



General Registers

32 bit Registers		16 bit Registers		8 bit Registers	
EAX	EBP	AX	BP	AH	AL
EBX	ESI	BX	SI	BH	BL
ECX	EDI	CX	DI	CH	CL
EDX	ESP	DX	SP	DH	DL
Bits 16-31		Bits 8-15		Bits 0-7	

Registers

6 Category

- General
- Pointer
- Index
- Segment
- Instruction
- Flag

EAX		AH	AX	AL
EBX		BH	BX	BL
ECX		CH	CX	CL
EDX		DH	DX	DL
EBP		BP		
ESI		SI		
EDI		DI		
ESP		SP		
EIP		IP		
EFLAGS		FLAGS		

CS
DS
ES
SS
FS
GS

Register names

- Accumulator
 - Base index
 - Count
 - Data
 - Stack Pointer
 - Base Pointer
 - Destination index
 - Source index
 - Instruction Pointer
 - Flags
- Segment registers
- Code
 - Data
 - Extra
 - Stack

General Registers I

- **EAX** – ‘Accumulator’
 - accumulator for operands and results data
 - usually used to store the return value of a procedure
- **EBX** – ‘Base Register’
 - pointer to data in the DS segment
- **ECX** – ‘Counter’
 - counter for string and loop operations
- **EDX** – ‘Data Register’
 - I/O pointer

General Registers II

- **ESI** – ‘Source Index’
 - source pointer for string operations
 - typically a pointer to data in the segment pointed to by the DS register
- **EDI** – ‘Destination Index’
 - destination pointer for string operations
 - typically a pointer to data/destination in the segment pointed to by the ES register

General Registers III

- **EBP** – ‘Base Pointer’
 - pointer to data on the stack
 - points to the current stack frame of a procedure
- **ESP** – ‘Stack Pointer’
 - pointer to the top address of the stack
 - holds the stack pointer and as a general rule should not be used for any other purpose

Segment Registers

- **CS** – ‘Code Segment’
 - contains the segment selector for the code segment where the instructions being executed are stored
- **DS (ES , FS , GS)** – ‘Data Segment’
 - contains the segment selectors for the data segment where data is stored
- **SS** – ‘Stack Segment’
 - contains the segment selector for the stack segment, where the procedure stack is stored

The EFLAGS Register I

- Carry Flag – CF (bit 0)
 - **Set** if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; **cleared** otherwise.
- Parity Flag – PF (bit 2)
 - **Set** if the least-significant byte of the result contains an even number of 1 bits; **cleared** otherwise.
- Adjust Flag – AF (bit 4)
 - Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; **cleared** otherwise.

The EFLAGS Register II

- Zero Flag – ZF (bit 6)
 - **Set** if the result is zero; **cleared** otherwise
- Sign Flag – SF (bit 7)
 - **Set** equal to the most-significant bit of the result, which is the sign bit of a signed integer
- Overflow Flag – OF (bit 11)
 - **Set** if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; **cleared** otherwise

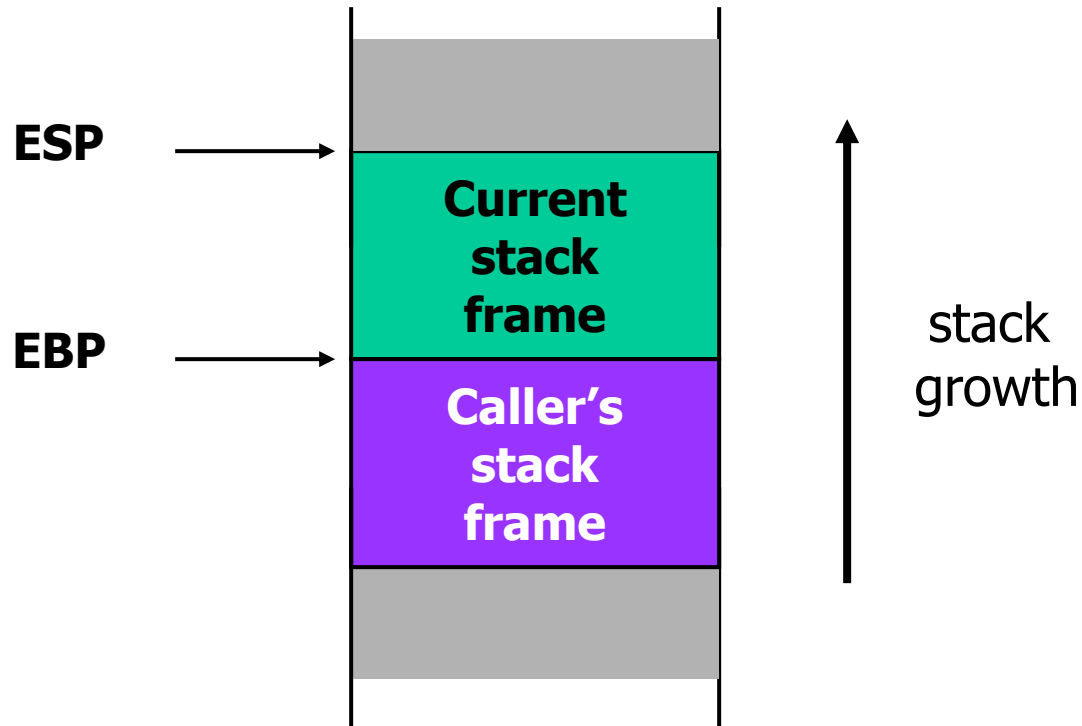
Instruction Pointer

- **EIP**

- ‘Instruction Pointer’
- Contains the offset within the code segment of the next instruction to be executed
- Cannot be accessed directly by software

The Stack

The stack starts in high memory and grows toward low memory



Intel Assembly

Intel Assembly

Goal: to gain a knowledge of Intel 32-bit assembly instructions

References:

- M. Pietrek, “Under the Hood: Just Enough Assembly Language to Get By”
 - MSJ Article, February 1998 www.microsoft.com/msj
 - Part II”, MSJ Article, June 1998 www.microsoft.com/msj
- IA-32 Intel® Architecture Software Developer’s Manual,
 - Volume 1: Basic Architecture
www.intel.com/design/Pentium4/documentation.htm#manuals
 - Volume 2A: Instruction Set Reference A-M
www.intel.com/design/pentium4/documentation.htm#manuals
 - Volume 2B: Instruction Set Reference N-Z
www.intel.com/design/pentium4/documentation.htm#manuals

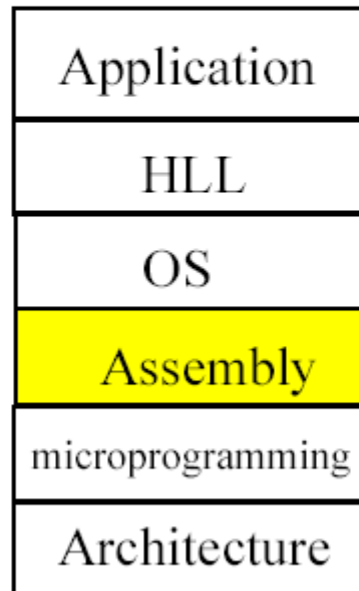
Assembly Programming

- Machine Language
 - binary
 - hexadecimal
 - machine code or object code
- Assembly Language
 - mnemonics
 - assembler
- High-Level Language
 - Pascal, Basic, C
 - compiler

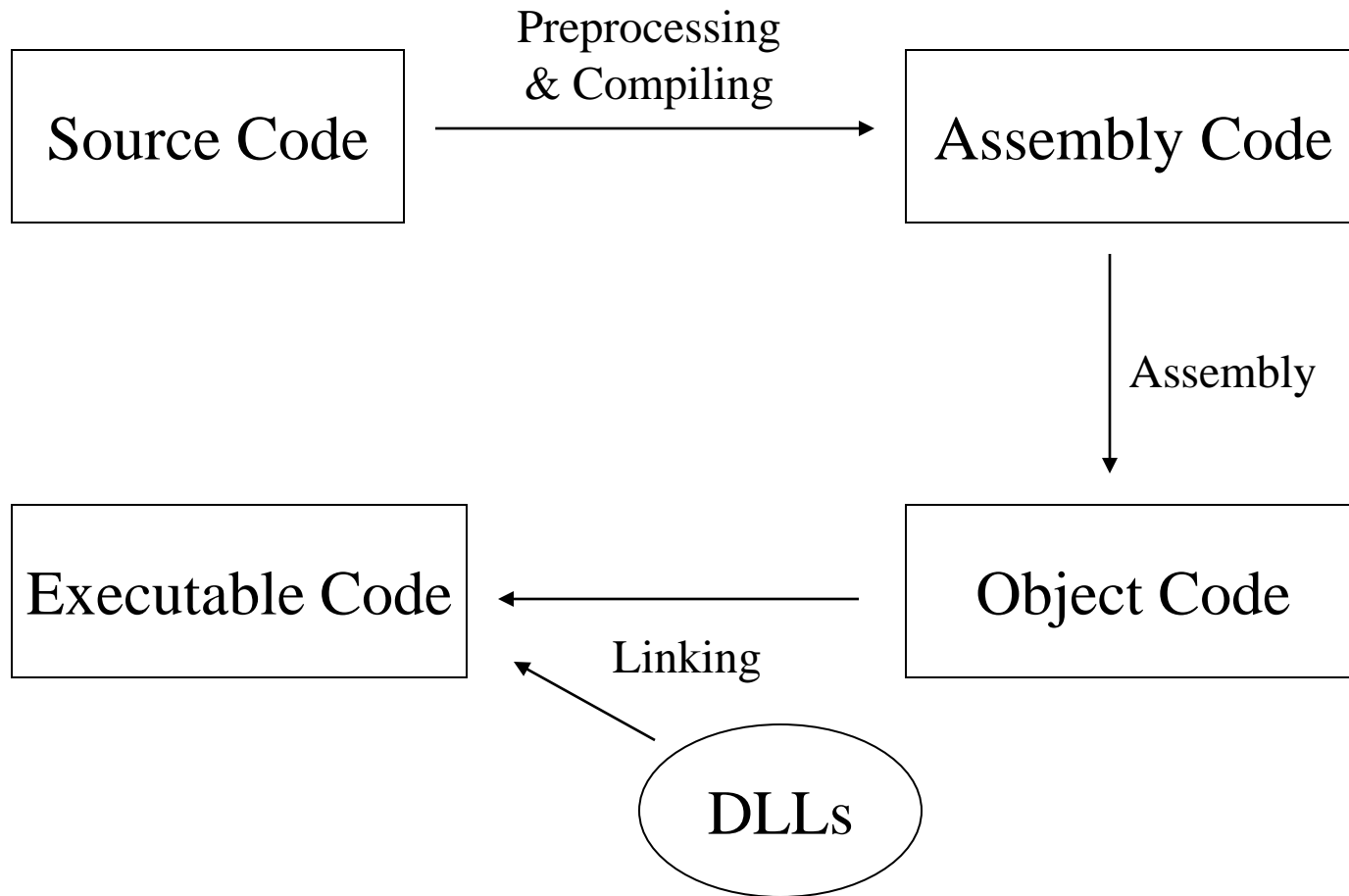
Assembly Language Programming

Motivations

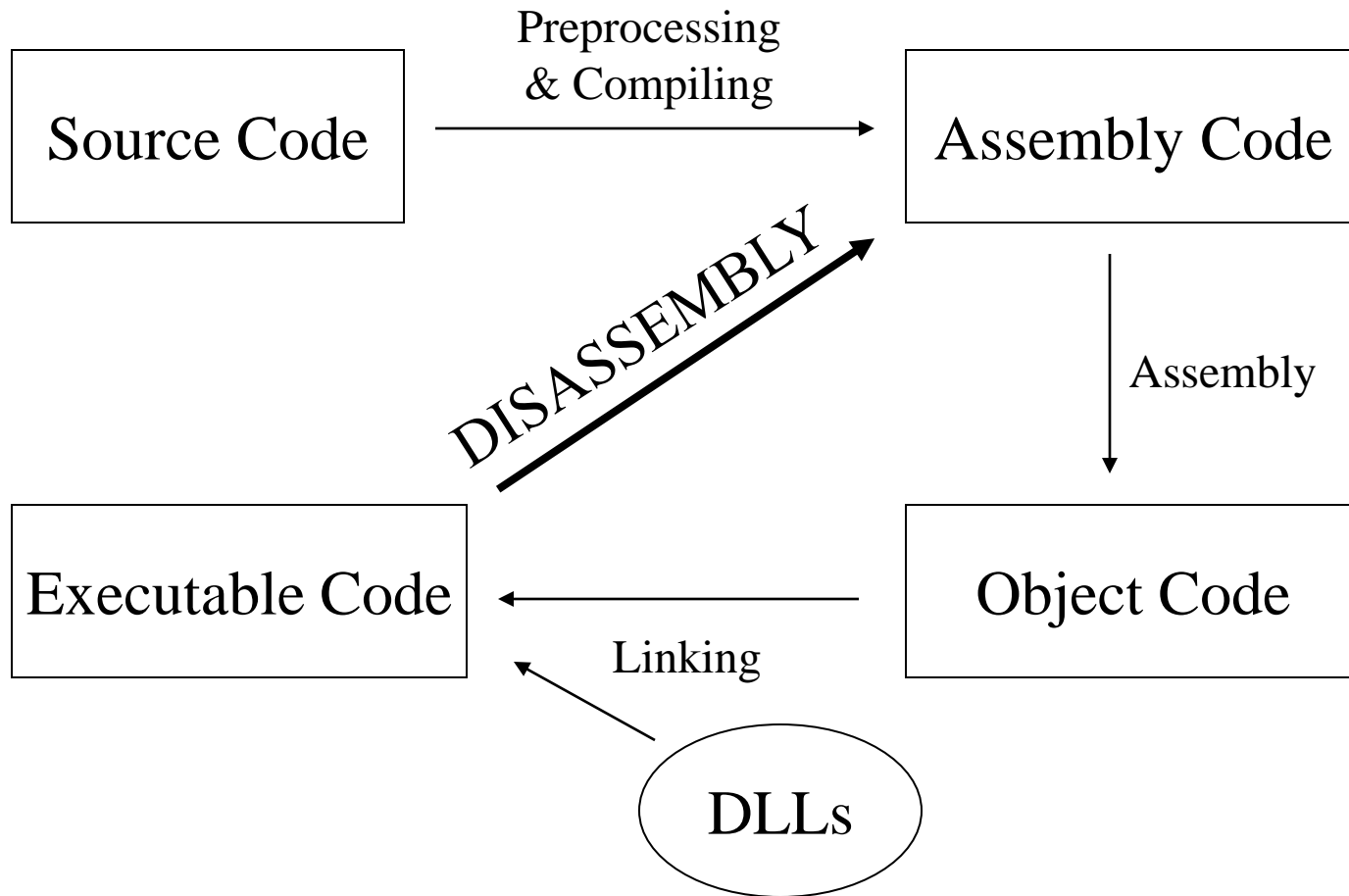
- Why do you learn assembly language?



What Does It Mean to Disassemble Code?



What Does It Mean to Disassemble Code?



Why is Disassembly Useful in Malware Analysis?

- It is not always desirable to execute malware: disassembly provides a static analysis.
- Disassembly enables an analyst to investigate all parts of the code, something that is not always possible in dynamic analysis.
- Using a disassembler and a debugger in combination creates synergy.

32-bit Instructions

- Instructions are represented in memory by a series of “opcode bytes.”
- A variance in instruction size means that disassembly is position specific.
- Most instructions take zero, one, or two arguments:

instruction destination, source

For example: `add eax, ebx`

is equivalent to the expression $\text{eax} = \text{eax} + \text{ebx}$

Assembly Instruction format

General format

mnemonic

operand(s)

;comments

MOV destination,source ;copy source operand to destination

Example:

MOV DX,CX

Example 2:

MOV CL,55H

MOV DL,CL

MOV AH,DL

MOV AL,AH

MOV BH,CL

MOV CH,BH

AH	AL
BH	BL
CH	CL
DH	DL

Section B.1: The 8086 Instruction Set

Page 865

MOV Move

Flags: Unchanged

Format: **MOV dest, source ; copy source to dest**

Function: Copy a word or byte from a register, memory location, or immediate number to a register or memory location. Source and destination must be of the same size and **cannot both be memory locations.**

MOV instruction (16 bits)

```
MOV CX,468H
MOV AX,CX
MOV DX,AX
MOV BX,DX
MOV DI,BX
MOV SI,DI
MOV DS,SI
MOV BP,DI
```

AH	AL
BH	BL
CH	CL
DH	DL
SP	
BP	
DI	
SI	
IP	
FLAGS	
CS	
DS	
ES	
SS	

What if ...

MOV AL,DX

Rule #1:

moving a value that is too large into a register will cause an error

MOV BL,7F2H ;Illegal: 7F2H is larger than 8 bits

MOV AX,2FE456H ;Illegal

Rule #2:

Data can be moved **directly** into **nonsegment** registers only

(Values cannot be loaded directly into any segment register.

To load a value into a segment register, first load it to a nonsegment register and then move it to the segment register.)

MOV AX,2345H

MOV DI,1400H

MOV DS,AX

MOV ES,DI

Rule #3:

If a value less than FFH is moved into a 16-bit register, the rest of the bits are assumed to be all zeros.

MOV BX, 5

BX = 0005

BH = 00, BL = 05

ADD instruction

ADD	destination,source	;ADD the source operand to the destination
-----	--------------------	--

MOV AL,25H

MOV BL,34H

ADD AL,BL

You can do this,

MOV DH,25H

ADD DH,34H

- What is the corresponding C++ code ?
- The way C++ compiler implements '+' operation is fixed
- Assembly language has more flexibility
- Assembly language can tailor the code closer to the hardware
(That is where the efficiency comes from)

ADD Signed or Unsigned ADD

Flags: Affected: OF,SF,ZF,AF,PF,CF

Format: **ADD dest, source ; dest=dest+source**

Function: Adds source operand to destination operand and places the result in destination. Both source and destination operands must match (e.g., both byte size or word size) and only one of them can be in memory.

Program Segments

- A segment is an area of memory that includes up to 64K bytes
- Begins on an address evenly divisible by 16
- 8085 could address a max. of 64K bytes of physical memory
 - it has only 16 pins for the address lines ($2^{16} = 64K$)
- 8088/86 stayed compatible with 8085
 - Range of 1MB of memory, it has 20 address pins ($2^{20} = 1 \text{ MB}$)
 - Can handle 64KB of code, 64KB of data, 64KB of stack
- A typical Assembly language program consist of three segments:
 - Code segments
 - Data segment
 - Stack segment

Program Segments...*a sample*

```
;THE FORM OF AN ASSEMBLY LANGUAGE PROGRAM
;NOTE: USING SIMPLIFIED SEGMENT DEFINITION
.MODEL SMALL
.STACK 64
.DATA

DATA1      DB      52H
DATA2      DB      29H
SUM        DB      ?

.CODE

MAIN        PROC    FAR      ;this is the program entry point
            MOV     AX,@DATA  ;load the data segment address
            MOV     DS,AX     ;assign value to DS
            MOV     AL,DATA1   ;get the first operand
            MOV     BL,DATA2   ;get the second operand
            ADD     AL,BL      ;add the operands
            MOV     SUM,AL     ;store the result in location SUM
            MOV     AH,4CH     ;set up to
            INT     21H        ;return to DOS
MAIN        ENDP
            END     MAIN      ;this is the program exit point
```

Program Segments



Why segment?

- Fig. 9-1a. How many address pins in 8086 chip?
- In 8085, only 16 pins for address lines. What's the size of physical memory can be pointed to in one instruction?
- What is the size of the physical memory 8086 can handle?
- How does 8086 handle the physical memory using the extra pins it is given?

Program Segments

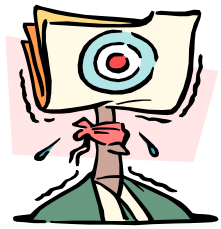
Code segment

The 8086 fetches the instructions (opcodes and operands) from the code segments.

The 8086 address types:

- Physical address
- Offset address
- Logical address
- Physical address
 - 20-bit address that is actually put on the address pins of 8086
 - Decoded by the memory interfacing circuitry
 - A range of 00000H to FFFFFH
 - It is the actual physical location in RAM or ROM within 1 MB mem. range
- Offset address
 - A location within a 64KB segment range
 - A range of 0000H to FFFFH
- Logical address
 - consist of a segment value and an offset address

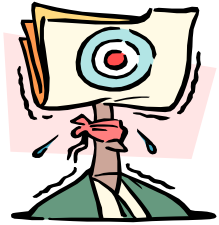
Program Segments...*example*



Define the addresses for the 8086 when it fetches the instructions (opcodes and operands) from the code segments.

- Logical address:
 - Consist of a **CS** (code segment) and an **IP** (instruction pointer)
format is **CS:IP**
- Offset address
 - **IP** contains the offset address
- Physical address
 - generated by shifting the **CS** left one hex digit and then adding it to the **IP**
 - the resulting 20-bit address is called the physical address

Program Segments...example



Suppose we have:

CS	2500
IP	95F3

- Logical address:

- Consist of a **CS** (code segment) and an **IP** (instruction pointer)
format is **CS:IP** **2500:95F3H**

- Offset address

- **IP** contains the offset address which is **95F3H**

- Physical address

- generated by shifting the **CS** left one hex digit and then adding it to the **IP**
25000 + 95F3 = 2E5F3H

Program Segments

Data segment

Data segment refers to an area of memory set aside for data

- Format DS:BX or DI or SI
- example:

DS:0200 = 25

DS:0201 = 12

DS:0202 = 15

DS:0203 = 1F

DS:0204 = 2B

Program Segments

Data segment

Example:

Add 5 bytes of data: 25H, 12H, 15H, 1FH, 2BH

Not using data segment

MOV	AL,00H	;clear AL
ADD	AL,25H	;add 25H to AL
ADD	AL,12H	
ADD	AL,15H	
ADD	AL,1FH	
ADD	AL,2BH	

Program Segments

Data segment

Example:

Add 5 bytes of data: 25H, 12H, 15H, 1FH, 2BH

using data segment with a constant offset

Data location in memory:

DS:0200 = 25

DS:0201 = 12

DS:0202 = 15

DS:0203 = 1F

DS:0204 = 2B

Program:

MOV AL,0

ADD AL,[0200]

ADD AL,[0201]

ADD AL,[0202]

ADD AL,[0203]

ADD AL,[0204]

Program Segments

Data segment

Example:

Add 5 bytes of data: 25H, 12H, 15H, 1FH, 2BH

using data segment with an offset register

Program:

```
MOV     AL,0
MOV     BX,0200H
ADD     AL,[BX]
INC     BX           ;same as "ADD BX,1"
ADD     AL,[BX]
INC     BX
ADD     AL,[BX]
INC     BX
ADD     AL,[BX]
```

Endian conversion

- **Little endian conversion:**

In the case of 16-bit data, the low byte goes to the low memory location and the high byte goes to the high memory address. (Intel, Digital VAX)

- **Big endian conversion:**

The high byte goes to low address. (Motorola)

Example:

Suppose DS:6826 = 48, DS:6827 = 22,

Show the contents of register BX in the instruction **MOV BX,[6826]**

Little endian conversion: BL = 48H, and BH = 22H

Program Segments

Stack segment

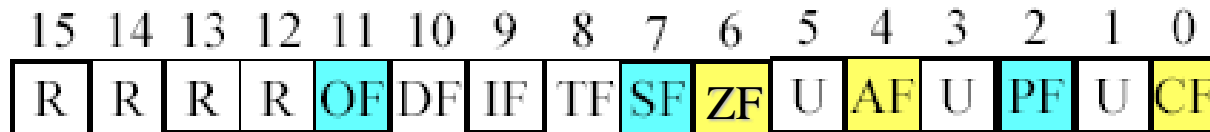
Stack

A section of RAM memory used by the CPU to store information temporarily.

- **Registers:** SS (Stack Segment) and SP (stack Pointer)
- **Operations: PUSH and POP**
 - **PUSH** – the storing of a CPU register in the stack
 - **POP** – loading the contents of the stack back into the CPU
- **Logical and offset address format:** SS:SP

Flag Register

- Flag Register (status register)
 - 16-bit register
 - Conditional flags: CF, PF, AF, ZF, SF, OF
 - Control flags: TF, IF, DF



R = reserved

U = undefined

OF = overflow flag

DF = direction flag

IF = interrupt flag

TF = trap flag

SF = sign flag

ZF = zero flag

AF = auxiliary carry flag

PF = parity flag

CF = carry flag

Flag Register and ADD instruction

- Flag Register that may be affected
 - Conditional flags: CF, PF, AF, ZF, SF, OF

Flow Control I

- **JMP *location***

Transfers program control to a different point in the instruction stream without recording return information.

```
jmp eax
```

```
jmp 0x00934EE4
```


Flow Control II

- ***CMP value, value / Jcc location***

The *compare* instruction compares two values, setting or clearing a variety of flags (e.g., ZF, SF, OF).

Various *conditional jump* instructions use flags to branch accordingly.

```
cmp  eax, 4
je   40320020
```

```
cmp  [ebp+10h], eax
jne  40DC0020
```

Flow Control III

- **TEST *value, value* / Jcc *location***

The *test* instruction does a logical AND of the two values. This sets the SF, ZF, and PF flags. Various *conditional jump* instructions use these flags to branch.

```
test    eax, eax
jnz     40DA0020
```

```
test    edx, 0056FCE2
jz      56DC0F20
```

Looping using zero flag

- The zero flag is set (ZF=1), when the counter becomes zero (CX=0)
- *Example:* add 5 bytes of data

	MOV	CX,05	;CX holds the loop count
	MOV	BX,0200H	;BX holds the offset data address
	MOV	AL,00	;initialize AL
ADD_LP:	ADD	AL,[BX]	;add the next byte to AL
	INC	BX	;increment the data pointer
	DEC	CX	;decrement the loop counter
	JNZ	ADD_LP	;jump to next iteration if counter ;not zero

Addressing Modes – Accessing operands (data) in various ways

80X86 Addressing Modes

1. Register
2. Immediate
3. Direct
4. Register indirect
5. Based relative
6. Indexed relative
7. Based indexed relative

Register Addressing Mode



- Registers are used to hold the data
- Memory is not accessed (hence is fast)
- Source and destination registers must match in size
- Exp: MOV BX,DX
 MOV ES,AX
 ADD AL,BH
 MOV CL,AX (error)

Immediate Addressing Mode



- The source operand is a constant
- can be used to load information to any registers except the segment registers and flag registers (can be done indirectly)
- operands come immediately after the opcode
- EXP: MOV AX,2550H
 MOV CX,625
 MOV BL,40H
- How about **MOV DS, 0123H** ? (page 42)

Direct Addressing Mode



- The data is in memory
- The address of the operand is provided in the instruction directly
- The address is the offset address
- The physical address can be calculated using the content in the DS register
- Exp1: `MOV DL,[2400]` ;move contents of DS:2400H into DL
- Exp2: `MOV AL,99H`
`MOV [3518],AL`



Register Indirect Addressing Mode

- The address of the memory location is in a register (SI, DI, or BX only)
- The physical address is calculated using the content of DS
- EXP: MOV CL,[SI] ;move contents of DS:SI into CL
 MOV [DI],AH ;move contents of AH into DS:DI
 MOV [SI],AX ; little endian is applied
 ;moves contents of AX into memory
 ;locations DS:SI and DS:SI +1



- 72



Index Relative Addressing Mode

- Similar to the based relative addressing mode
- DI and SI are used to hold the offset address
- DS is used for calculating physical address

EXP: **MOV DX,[SI]+5** ;PA = DS (sl) + SI + 5
 MOV CL,[DI]+20 ;PA = DS (sl) + DI + 20



Based Indexed Addressing Mode

- Combining the previous two modes
- One base register and one index register are used
- for physical address calculation, DS is used for BX; SS is used for BP
- EXP: MOV CL,[BX][DI]+8 ;PA=DS(s1)+BX+DI +8
 MOV AH,[BP][SI]+29 ;PA=SS(s1)+BP+SI +29