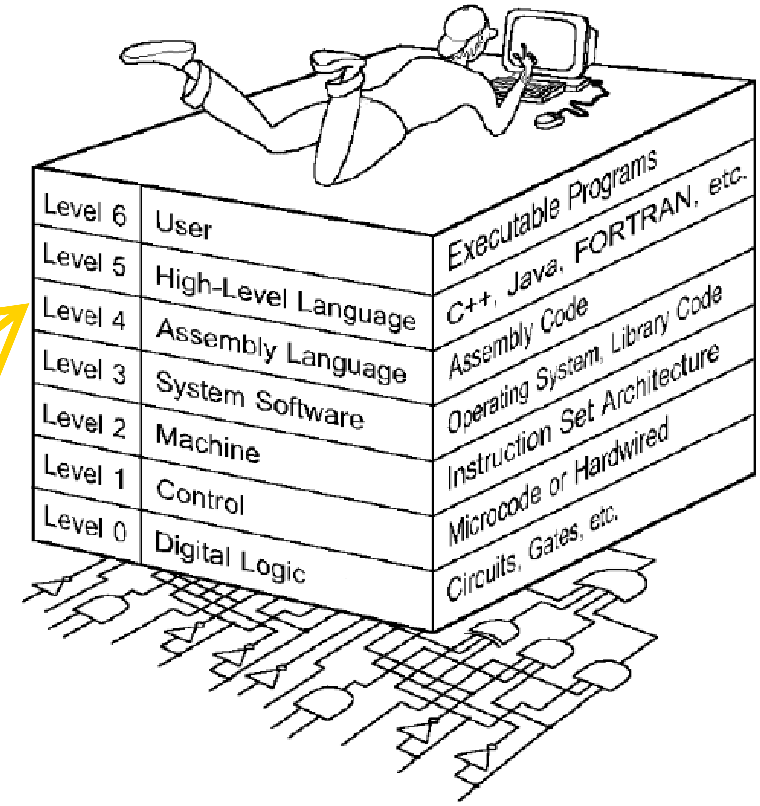
Assembly Language for x86 Processors



- X86 Processor Architecture

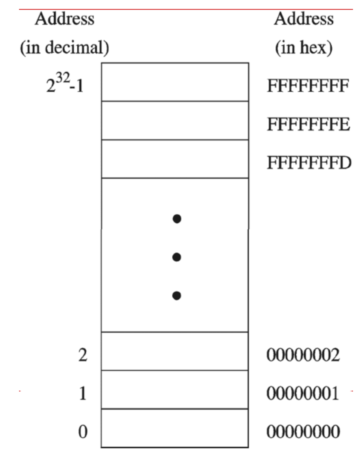
LO-4:

- how an instruction is **executed**.
- Implement basic **assembly-language programs**.
- Explain different **instruction formats**, various **addressing modes.**

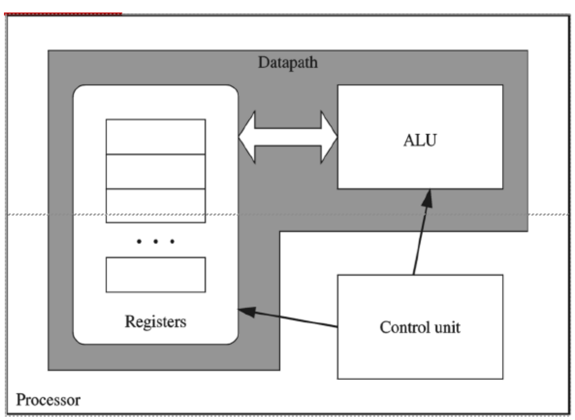
>>> Quiz-4 and **Test-4** (with Ch-6)

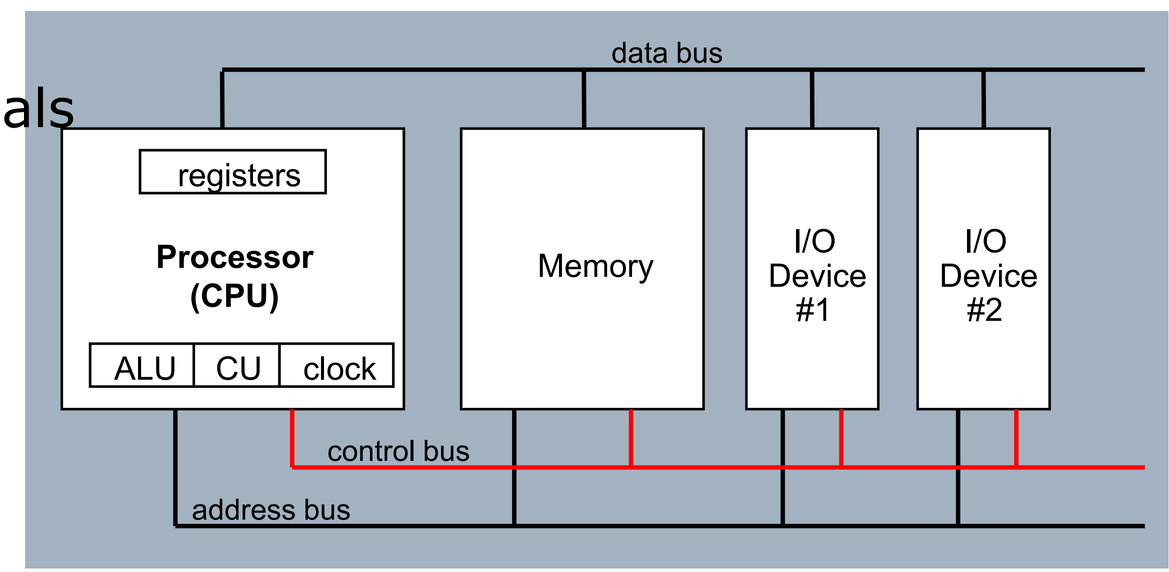
Basic IA32 Computer Organization

- Since the 1940's, the *Von Neumann* computers contains three key components:
  - Processor (CPU), Memory, I/O & Storage Devices
  - Interconnected with one or more buses
    - Data Bus, Address Bus, Control Bus



- The processor consists of
  - Datapath (ALU+Registers)
  - Control unit (generates signals to execute instructions)





Registers

- Registers are high speed memory inside the CPU
  - Eight 32-bit general-purpose registers
  - Six 16-bit segment registers
  - Processor Status Flags (EFLAGS) and Instruction Pointer (EIP)

Extended Instruction Pointer

32-bit General-Purpose Registers

* SFs of all but ESP = base & indexing; Also, multiple possible for integer size variation

AH/AL (8-bit)
AX=AH:AL (16b)
EAX=AY:AX (32b)

Accumulator EAX
data pointer EBX
counter ECX
I/O Pointer EDX

EBP
ESP
pointers for Stack segment
ESI
index for Data Segment
EDI

* GPRS for
- 32-bit offsets within a segment in protected mode, and
- the translation of segmented addresses to 32-bit linear addresses.

16-bit Segment Registers

EFLAGS

CS
SS
DS

ES
FS
GS

EIP like PC in MAARIE

- 16b segment+32b offset in it = 48b segmented address

-Linear/Virtual addresses 32b --> Page Table --> Physical address

Larger virtual address space: The IA-32 architecture defines a 48-bit segmented address format, with a 16-bit segment number and a 32-bit offset within the segment. Segmented addresses are mapped to 32-bit linear addresses.

General-Purpose Registers

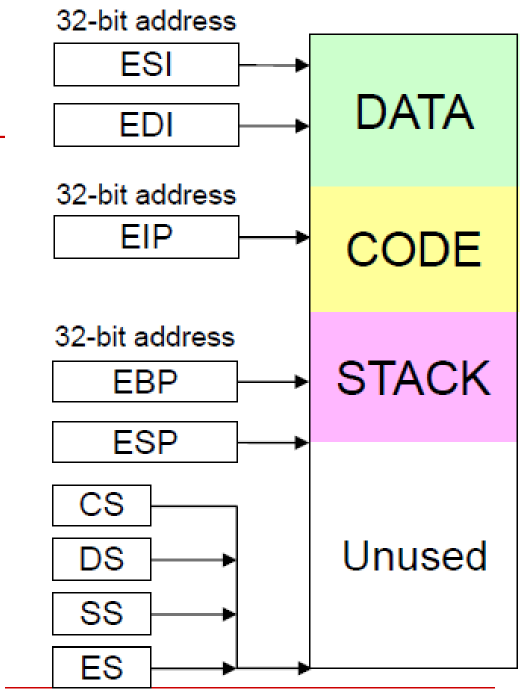
- Used primarily for arithmetic and data movement
  - `mov eax 10 ;`move constant integer 10 into register eax
- (But have) **specialized uses** (as well) of Registers:
  - **eax** – Accumulator register
    - Automatically used by multiplication and division instructions
  - **ecx** – Counter register
    - Automatically used by LOOP instructions
  - **esp**– Stack Pointer register
    - Used by PUSH and POP instructions, points to top of stack
  - **esi** and **edi** – Source Index and Destination Index register
    - Used by string and array instructions
  - **ebp** – Base Pointer register
    - Used to reference parameters and local variables on the stack

Special-Purpose & Segment Registers

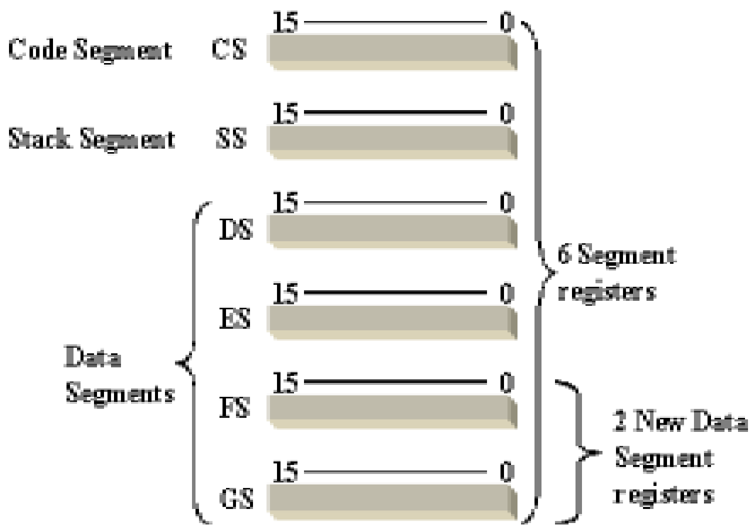
- EIP = Extended Instruction Pointer
  - Contains **next** instruction's address, **like PC**
- EFLAGS = Extended Flags Register
  - Contains status and control flags
  - Each flag is a single binary bit
- Six 16-bit Segment Registers
  - Support segmented memory
  - Segments contain distinct contents
    - Code
    - Data
    - Stack

Virtual memory: paged/segmented

Linear address space of a program (up to 4 GB)



base address = 0 for all segments



Background for EFLAGS

Overflow detection

- X, Y and Z are N-bit 2's-complement numbers and $Z_{2c}=X_{2c}+Y_{2c}$
- Overflow occurs if $X_{2c}+Y_{2c}$ exceeds the

A maximum value represented by N-bits.

- **If the signs of *X* and *Y* are different,**

B **don't detect overflow for $Z_{2c}=X_{2c}+Y_{2c}$.**

- **In case the signs of *X* and *Y* are the same, if the sign of $X_{2c}+Y_{2c}$ is opposite, overflow detected.** OF=C_{in} XOR C_{out} (of MSb)
  - Case 1: X, Y positive, Z sign bit ='1'
  - Case 2: X, Y negative, Z sign bit ='0'

C If X, Y and Z have same, dont detect overflow.

■ Case 1: X, Y positive, Z sign bit ='1'

Example

Overflow detected!

Case-B-1

- $X_{2c}=(01111010)_{2c}$, $Y_{2c}=(00001010)_{2c}$,
  $X_{2c}+Y_{2c}=(10000100)_{2c}$ Overflow detected

OF=C_{in} XOR C_{out} (of MSb)

Signed value = Unsigned value - 2^n
Signed value = Unsigned value - 256

0	1	1	1	1	0	1	0
0	0	0	0	1	0	1	0
1	0	0	0	0	1	0	0

122
+
10
=
132

A negative sum of positive operands (or vice versa) is an overflow.
Ignore the sign bit and depend on the overflow behavior

Sign=1 negative

BUT actual answer is -124 ==> problem.

No carry-out of MSb but there is carry-in ==>O.F. Hence, dont ignore it!

Complement it to fix this, once this OF is detected! i.e. 2^8+(-124)=132

- Case 2: X, Y negative, Z sign bit ='0'

Example

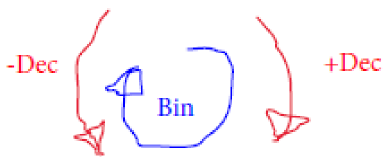
Overflow detected!

Case-B-2

- $X_{2c}=(10011010)_{2c}$, $Y_{2c}=(10001010)_{2c}$,
  $X_{2c}+Y_{2c}=(00100100)_{2c}$ Overflow detected

OF=C_{in} XOR C_{out} (of MSb)

Signed value = Unsigned value - 2^n
Signed value = Unsigned value - 256



	1	0	0	1	1	0	1	0	-102
									+
	1	0	0	0	1	0	1	0	-118
									=
1	0	0	1	0	0	1	0	0	-220

A negative sum of positive operands (or vice versa) is an overflow.
Ignore the sign bit and depend on the overflow behavior

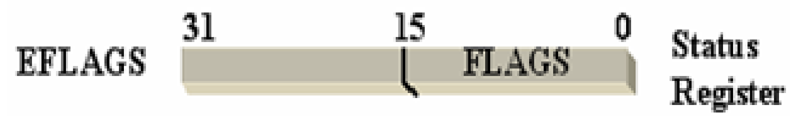
Sign =0

BUT actual answer is 36 ==> problem.

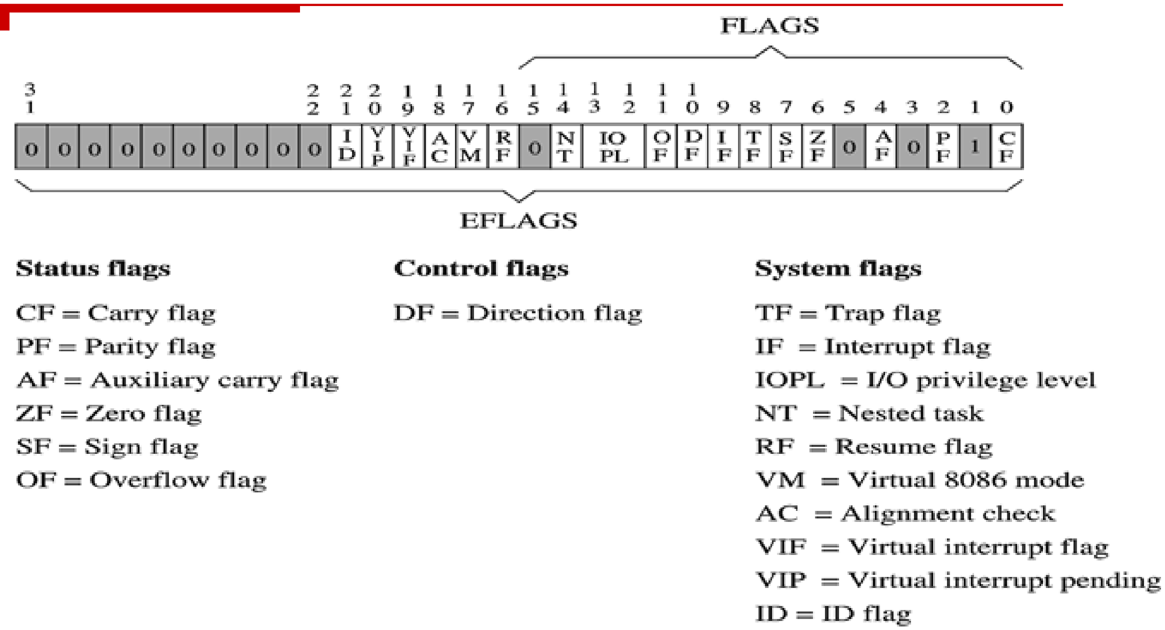
carry-out=1, carry-in=0
==> O.F.
dont ignore it!

Solution:
Once this OF is detected, Do this:
36-2^8 = -220

EFLAGS Register



- For each operation that performed in CPU, there must be *a mechanism* to determine if the operation is success or not
  - Contains status and control flags/bits for this purpose



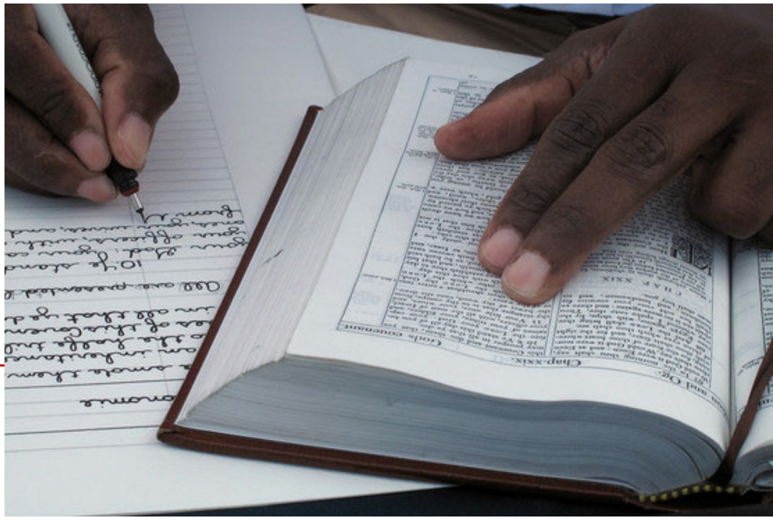
- ❖ Status Flags
  - ✧ Status of arithmetic and logical operations
- ❖ Control and System flags
  - ✧ Control the CPU operation
- ❖ Programs can set and clear individual bits in the EFLAGS register

Status Flags

- Carry Flag
  - Set when unsigned arithmetic result is out of range
- Overflow Flag
  - Set when signed arithmetic result is out of range
- Sign Flag
  - Copy of sign bit, set when result is negative
- Zero Flag
  - Set when result is zero
- Auxiliary Carry Flag (was designed for BCD arithmetic)
  - Set when there is a carry from bit 3 to bit 4
- Parity Flag (uses Odd parity!)
  - Set when parity is even
  - Least-significant byte in the result contains even number of 1s

IA EFLAGS uses this

Odd parity check



- Odd parity check
- Example: input: A(7…0), Output: odd_parity bit
  - If there are odd numbers of 1 in A, odd_parity = ‘0’,
  - If there are even numbers of 1 in A, odd_parity = ‘1’

e.g., A = “10100001”,

odd_parity = ‘0’

A = “10100011”,

Parity Bit is SET ===> odd_parity = ‘1’

64-Bit Processors

- 64-Bit Operation Modes
  - Compatibility mode – can run existing 16-bit and 32-bit applications (Windows supports only 32-bit apps in this mode)
  - 64-bit mode – Windows 64 uses this
- Basic Execution Environment
  - addresses can be 64 bits (48 bits, in practice)
  - 16 64-bit general purpose registers
  - 64-bit instruction pointer named RIP

Return Instruction Pointer

64-Bit General Purpose Registers

Compatibility mode

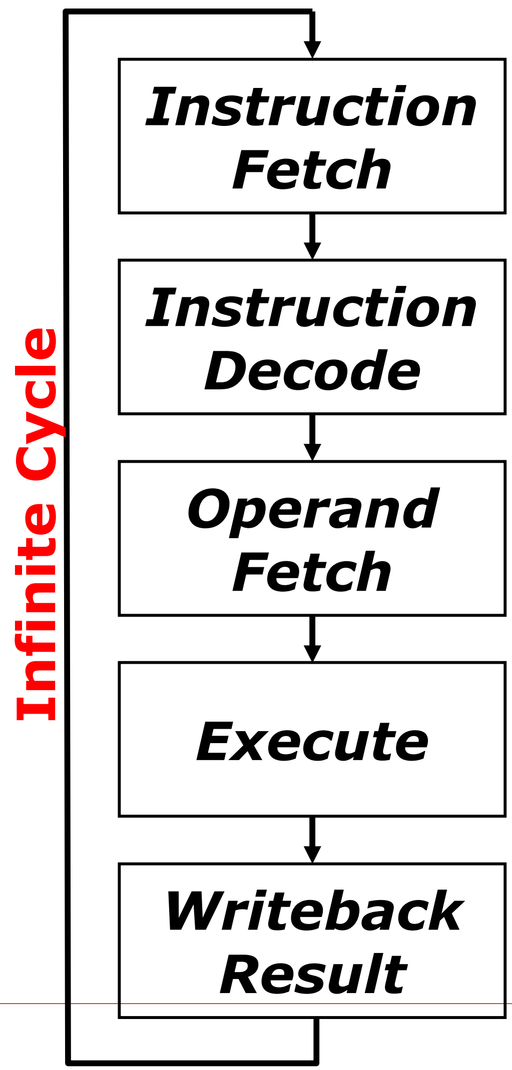
- 32-bit general purpose registers:
  - EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
- 64-bit general purpose registers:
  - RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8, R9, R10, R11, R12, R13, R14, R15

AH/AL (8-bit)
AX=AH:AL (16b)
EAX=AY:AX (32b)
RAX=EAY:EAX (64B)

FDE Cycle: The Heart-beat of CPU for Instruction Execution

using EIP,

CIR in CU holds instr. during FDE cycle



Obtain instruction from program storage

Determine required actions and instruction size

Locate and obtain operand data

Compute result value and status

Deposit results in storage for later use

Assembly Language for x86 Processors

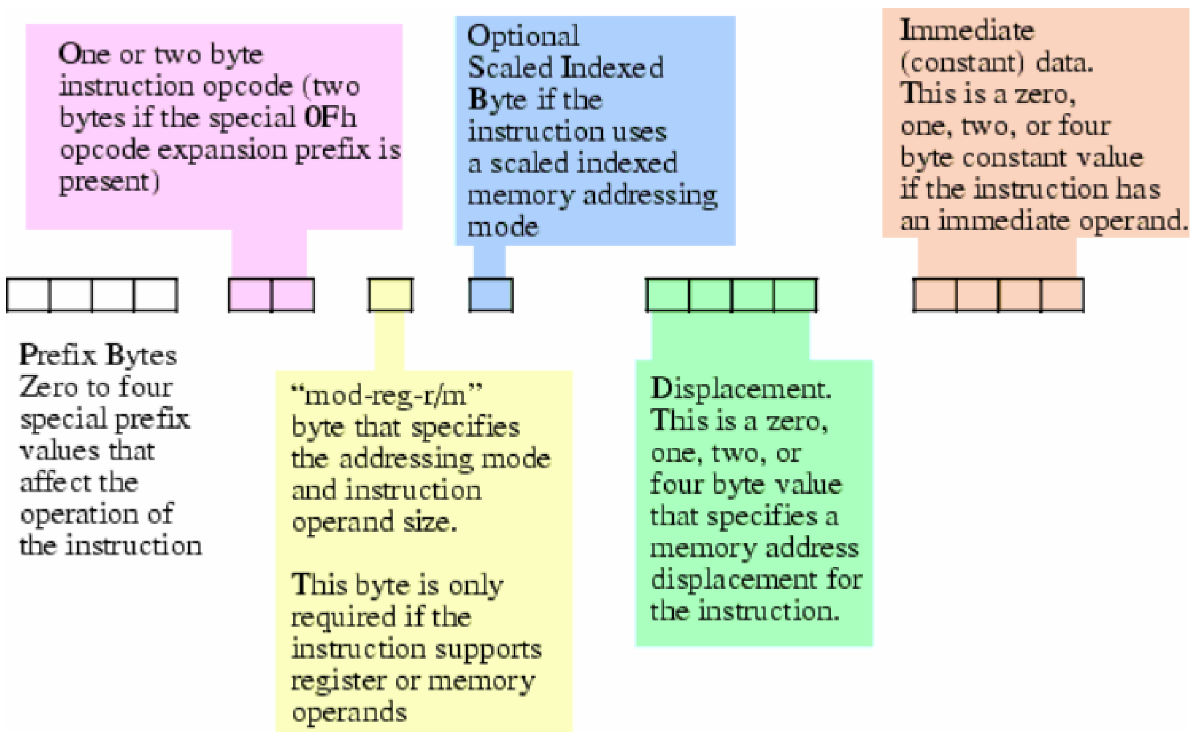
- Assembly Language: Fundamentals & Programming

x86 Assembly Language: Outline

- Statements
- Pseudo-instructions
- Directives

e.g. macro

To implement basic, selection, and program flow control structures.



http://www.c-jump.com/CIS77/CPU/x86/lecture.html#X77_0010_real_encoding

Assembly-Language Statement Structure

- The heart of any assembly language program are statements

$$\underbrace{\textbf{label:}}_{\text{optional}} \qquad \underbrace{\textbf{mnemonic}}_{\substack{\text{opcode name}\\ \text{or}\\ \text{directive name}\\ \text{or}\\ \text{macro name}}} \qquad \underbrace{\textbf{operand(s)}}_{\text{zero or more}} \qquad \underbrace{\textbf{;comment}}_{\text{optional}}$$

$$\underbrace{\textbf{label:}}_{\text{optional}} \quad \underbrace{\textbf{mnemonic}}_{\substack{\text{opcode name}\\ \text{or}\\ \text{directive name}\\ \text{or}\\ \text{macro name}}} \quad \underbrace{\textbf{operand(s)}}_{\text{zero or more}} \quad \underbrace{\textbf{;comment}}_{\text{optional}}$$

Statements, Label

- If a label is present, the assembler defines the label as equivalent to the address (as place markers)
  - the first byte of the object code generated for that instruction will be loaded
- Two types of labels
  - Data label
    - Just like the identifiers in Java and C →must be unique
    - example:

```
count DWORD 100 ;Define a variable named count
```

  - Code label
    - Mark of a memory location for jump and loop instructions
    - example:

```
L1:     (followed by colon)

        proc_name:
           procedure body
           ...
           ret
```

label:	mnemonic	operand(s)	;comment
optional	opcode name or directive name or macro name	zero or more	optional

Statements, Label

- The programmer may subsequently use the **label as an address or as data** in another instruction’s address field

- The assembler **replaces** the label with the assigned value when creating an **object program**

- Reasons for using a label:
  - Makes a program location **easier to find** and remember
  - Can easily be **moved** to correct a program
  - Programmer does **not have to calculate** relative or absolute memory addresses, but just uses labels as needed
    - Example: branch instructions

$$\underbrace{\textbf{label:}}_{\text{optional}} \quad \underbrace{\textbf{mnemonic}}_{\substack{\text{opcode name}\\ \text{or}\\ \text{directive name}\\ \text{or}\\ \text{macro name}}} \quad \underbrace{\textbf{operand(s)}}_{\text{zero or more}} \quad \underbrace{\textbf{;comment}}_{\text{optional}}$$

Statements, mnemonic

- The mnemonic is the **name of the operation or function** of the assembly language statement
- In the case of a **machine instruction**, a mnemonic is the **symbolic name** associated with a particular **opcode**

Common x86 Instruction Set Operations

Data transfer	Transfer data from one location to another If memory is involved: Determine memory address Perform virtual-to-actual-memory address transformation Check cache Initiate memory read/write
Arithmetic	May involve data transfer, before and/or after
	Perform function in ALU
	Set condition codes and flags
Logical	Same as arithmetic
Conversion	Similar to arithmetic and logical. May involve special logic to perform conversion
Transfer of control	Update program counter. For subroutine call/return, manage parameter passing and linkage
I/O	Issue command to I/O module
	If memory-mapped I/O, determine memory-mapped address

Common x86 Instruction Set Operations:
Data transfer, Arithmetic, Logical, Conversion, Control transfer, I/O

x86 Instruction Set, Data Transfer

Operation Name	**Description**
MOV Dest, Source	Move data between registers or between register and memory or immediate to register.
XCHG Op1, Op2	**Swap contents** between two registers or register and memory.
PUSH Source	Decrements stack pointer (ESP register), then copies the source operand to the top of stack.
POP Dest	Copies top of stack to destination and increments ESP.

Both operands must be the same size

Common x86 Instruction Set Operations:
Data transfer, Arithmetic, Logical, Conversion, Control transfer, I/O

x86 Instruction Set: Arithmetic

Operation Name	**Description**
ADD Dest, Source	Adds the destination and the source operand and stores the result in the destination. Destination can be register or memory. Source can be register, memory, or immediate.
SUB Dest, Source	Subtracts the **source from the destination** and stores the result in the destination.
MUL Op	**Unsigned integer multiplication** of the operand by the AL, AX, or EAX register and stores in the register. Opcode indicates size of register. AX=AH:AL (16-bit) EAX=AY:AX (32b)
IMUL Op	Signed integer multiplication.
DIV Op	W, 4 Ws, 4 DWs, 4 QWs concatenated! **Divides unsigned** the value in the AX, DX:AX, EDX:EAX, or RDX:RAX registers (dividend) by the source operand (divisor) and stores the result in the AX (Quotient in AL, Remainder in AH), DX:AX, EDX:EAX, or RDX:RAX registers.
IDIV Op	Signed integer division.
INC Op	Adds 1 to the destination operand, while preserving the state of the CF flag.
DEC Op	Subtracts 1 from the destination operand, while preserving the state of the CF flag.
NEG Op	Replaces the value of operand with (0 – operand), using twos complement representation.
CMP Op1, Op2	Compares the two operands by subtracting the second operand from the first operand and sets the status flags in the EFLAGS register according to the results.

Common x86 Instruction Set Operations:
Data transfer, Arithmetic, Logical, Conversion, Control transfer, I/O

x86 Instruction Set, Logical

Operation Name	Description
NOT Op	Inverts each bit of the operand.
AND Dest, Source	Performs a bitwise AND operation on the destination and source operands and stores the result in the destination operand.
OR Dest, Source	Performs a bitwise OR operation on the destination and source operands and stores the result in the destination operand.
XOR Dest, Source	Performs a bitwise XOR operation on the destination and source operands and stores the result in the destination operand.
TEST Op1, Op2	Performs a bitwise AND operation on the two operands and sets the S, Z, and P status flags. The operands are unchanged.

Common x86 Instruction Set Operations:
Data transfer, Arithmetic, Logical, Conversion, Control transfer, I/O

x86 Instruction Set, Shift and Rotate

Operation Name	Description
SAL Op, Quantity	Shifts the source operand left by from 1 to 31 bit positions. Empty bit positions are cleared. The CF flag is loaded with the last bit shifted out of the operand.
SAR Op, Quantity	Shifts the source operand right by from 1 to 31 bit positions. Empty bit positions are cleared if the operand is positive and set if the operand is negative. The CF flag is loaded with the last bit shifted out of the operand.
SHR Op, Quantity similar for SHL	Shifts the source operand right by from 1 to 31 bit positions. Empty bit positions are cleared and the CF flag is loaded with the last bit shifted out of the operand.
ROL Op, Quantity	Rotate bits to the left, with wraparound. The CF flag is loaded with the last bit shifted out of the operand.
ROR Op, Quantity	Rotate bits to the right, with wraparound. The CF flag is loaded with the last bit shifted out of the operand.
RCL Op, Quantity	Rotate bits to the left, including the CF flag, with wraparound. This instruction treats the CF flag as a one-bit extension on the upper end of the operand.
RCR Op, Quantity	Rotate bits to the right, including the CF flag, with wraparound. This instruction treats the CF flag as a one-bit extension on the lower end of the operand.

Arithmetic

for signed

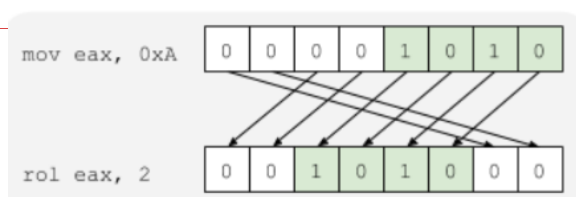
sign bit remains copied in the empty bits

Logical

for Unsigned

-ve sign bit DOES NOT copy in the empty bits

Only one w/o CF use



Common x86 Instruction Set Operations:
Data transfer, Arithmetic, Logical, Conversion, Control transfer, I/O

x86 Instruction Set, Transfer of Control

Operation Name	Description
CALL proc	**Saves procedure linking information** on the stack, and branches to the called procedure specified using the operand. The operand specifies the address of the first instruction in the called procedure.
RET	Transfers program control to a **return** address located on the **top of the stack**. The return is made to the instruction that follows the CALL instruction.
JMP Dest	Transfers program control **to a different point** in the instruction stream **without recording** return information. The operand specifies the address of the instruction being jumped to.
Jcc Dest	Checks the state of 1+ status flags in the EFLAGS register (CF, OF, PF, SF, and ZF) and, **if** the **flags** are in the specified state (**condition**), performs a **jump** to the target instruction specified by the **destination** operand. See Tables 13.8 and 13.9.
NOP	This instruction performs **no operation**. It is a one-byte or multi-byte NOP that takes up space in the instruction stream but does not impact machine context, **except** for the **EIP** register.
HLT	**Stops** instruction **execution** and places the **processor** in a **HALT state**. An enabled **interrupt**, a debug **exception**, the BINIT# signal, the **INIT#** signal, or the **RESET#** signal will **resume execution**.
WAIT	Causes the **processor** to **repeatedly check for** and **handle** pending, unmasked, floating-point **exceptions before proceeding**.
INT Nr	**Interrupts** current program, **runs (ISR**) specified interrupt program

Jmp
Cnd
Code

x86 Instruction Set, Input/Output

Operation Name	Description
IN Dest, Source	**Copies** the data from the I/O **port** specified by the source operand to the destination operand, which is a **register** location.
INS Dest, Source	Copies the data from the I/O **port** specified by the source operand to the destination operand, which is a **memory** location.
OUT Dest, Source	Copies the byte, word, or doubleword value from the source **register** to the I/O **port** specified by the destination operand.
OUTS Dest, Source	Copies byte, word, or doubleword from the source operand to the I/O port specified with the destination operand. The source operand is a **memory** location.

$$\underbrace{\textbf{label:}}_{\text{optional}}\quad \underbrace{\textbf{mnemonic}}_{\substack{\text{opcode name}\\ \text{or}\\ \text{directive name}\\ \text{or}\\ \text{macro name}}}\quad \underbrace{\textbf{operand(s)}}_{\text{zero or more}}\quad \underbrace{\textbf{;comment}}_{\text{optional}}$$

Statements, operands

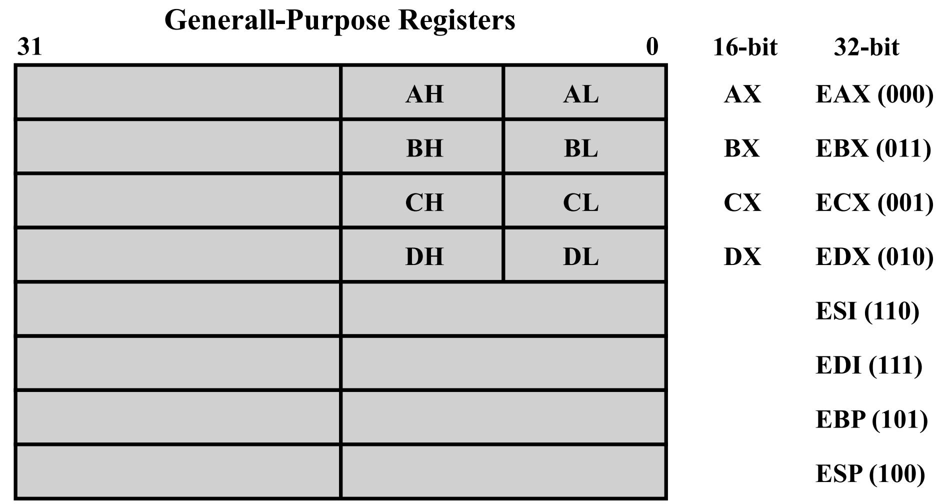
- An assembly language statement includes <u>zero or more operands</u>
- Each operand identifies:
  - **immediate** value,
  - a **register** value, or
  - a **memory** location
- Typically the assembly language **provides <u>conventions</u>**:
  - **for** distinguishing among the three **types of operand** references,
  - **for** indicating **addressing mode**

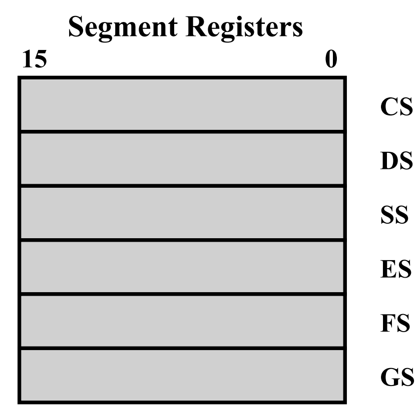
Immediate values

- **Radix** may be one of the following (upper or lower case):
  - h – hexadecimal
  - d – decimal (by default)
  - b – binary
  - r – encoded real
  - Hexadecimal must beginning with letter 0 → **0A5h**
  - **Optional** leading + or – **sign**
  - Enclose character in single or double **quotes**
- Examples:
  - `30d, 06Ah, 42, 1101b`
  - `'A', "x"`

Intel x86 Program Execution Registers

- statement may **refer** to a register operand **by name**.
- The assembler translates the **symbolic name into the binary identifier** for the register



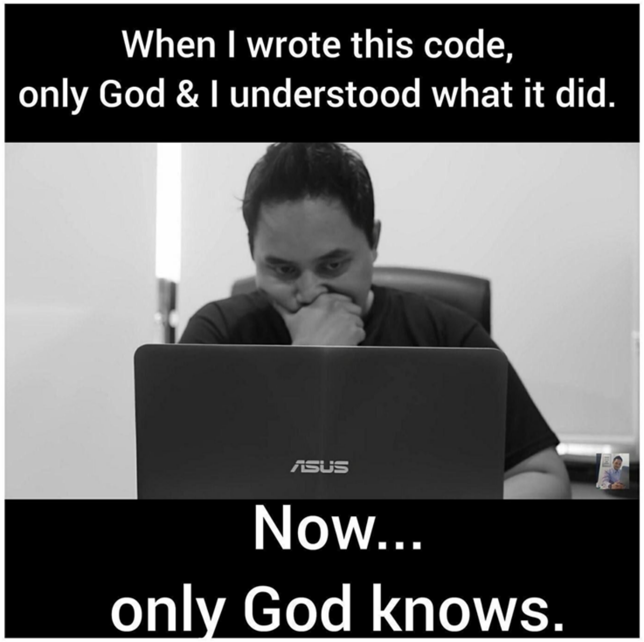


Identifiers and Reserved Words

- Identifiers
  - **Length:** Contains **1-247 characters**, including digits
  - **Not case sensitive**
  - The **first character** must be a letter, _ , @, ?, or $
  - examples: `var1, $first, _main`
- **Reserved words cannot be used as identifiers**
  - Instruction mnemonics, directives, register names, type attributes, operators, predefined symbols

Statements, comment

- All assembly languages allow the placement of comments in the program
- A comment can either :
  - occur at the **right-hand end** of an assembly statement or
  - occupy an **entire test line**
- The comment begins with a special character that signals to the assembler that the rest of the line is a comment and is to be ignored by the assembler
  - the x86 architecture use a **semicolon (;**) for the special character



Getting started with MASM

- Download Visual studio
- Setup Visual studio: https://www.youtube.com/watch?v=-fCyvipptZU
  - Start without debugging
  - C++ configuration

The Microsoft **Macro Assembler** is an x86 assembler that uses the Intel syntax for MS-DOS and Microsoft Windows. Beginning with MASM 8.0, there are **two versions** of the assembler: One for 16-bit & 32-bit assembly sources, and another for 64-bit sources only.

Program Template

```
TITLE Program Template                (Template.asm)

; Program Description:
; Author:
; Creation Date:
; Revisions:
; Date:                 Modified by:
.data
    ; (insert variables here)
.code
main PROC
    ; (insert executable instructions here)

    ;exit
main ENDP
    ; (insert additional procedures here)

END main
```

Program entry point

startup procedure

Example 1:

Write an assembly program to add the values 5 and 6 and store the value in eAx

```
TITLE Add (AddTwo.asm)
```

```
; This program adds two 32-bit integers.
```

```
.386
.model flat, stdcall
.stack 4096
ExitProcess proto, dwExitCode:dword
DumpRegs PROTO
```

(memory model, calling convention)

--> standard Windows service prototype declaration

~~Line 1 contains the .386 directive, which identifies this as a 32-bit program that can access 32-bit registers and addresses.~~

```
.code
main proc
    mov eax, 5
    add eax, 6
    invoke ExitProcess, 0
main endp
```

Line 2 selects the program's memory model (flat), and identifies the calling convention (named stdcall) for procedures. We use this because 32-bit Windows services require the stdcall convention to be used

Line 3 sets aside 4096 bytes of storage for the runtime stack, which every program must have.

Line 4 declares a prototype for the ExitProcess function, which is a standard Windows service

A prototype consists of the function name, the PROTO keyword, a comma, and a ~~list of input parameters. The input parameter for ExitProcess is named~~ dwExitCode.

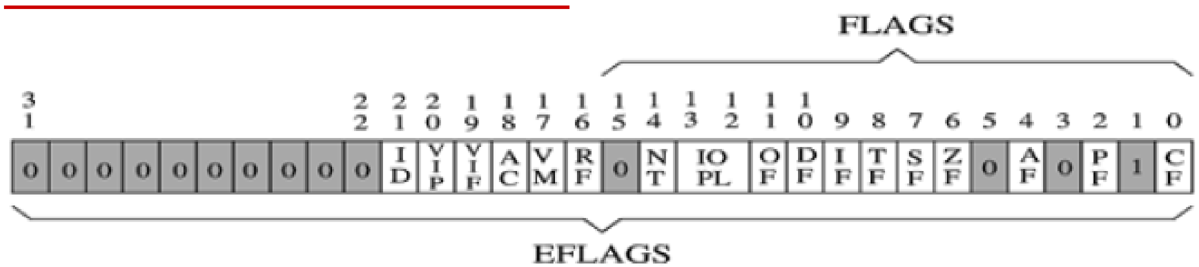
```
end main
```

Line 17 uses the end directive to mark the last line to be assembled, and it identifies the program entry point (main). The label main was declared on Line 10, and it marks the address at which the program will begin to execute.

Example 1: Output

Program output, showing registers and flags:

```
EAX = 0000000B EBX = 7EFDE000 ECX = 00000000 EDX = 00401005
ESI = 00000000 EDI = 00000000 EIP = 00401018 ESP = 0018FF8C
EBP = 0018FF94 EFL = 00000200

OV = 0 UP  = 0 EI = 1 PL = 0 ZR = 0 AC = 0 PE  = 0 CY = 0
OF     DF       IntF   SF     ZF      AF     PF       CF
       Dir.                        Aux. C.
```



Statements, Pseudo-instructions

- Pseudo-instructions and directives are statements which are:
  - **not** real x86 **machine instructions**.
    - **not directly translated** into machine language instructions
  - instructions to the assembler to perform specified **actions during the assembly** process
- Examples include:
  - Define **constants**
  - Designate areas of memory for **data** storage
    - MASM→`.data, .DATA`, and `.Data` are the same
  - Initialize **areas** of memory
  - Place tables or other **fixed data** in memory
  - Allow **references** to other programs

Intrinsic Data Types

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer (can also be a Near pointer in real-address mode)
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer (can also be a Near pointer in protected mode). D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

Data Definition Statement

- A data definition statement **sets aside** storage in memory for a **variable**.
- May optionally **assign** a name (**label**) to the data
- Syntax:

```
[name] directive initializer [,initializer] . . .
```

value1 BYTE 10

- All initializers become binary data in memory

Examples

```
value1 BYTE 'A'              ; character constant
value2 BYTE 0                ; smallest unsigned byte
value3 BYTE 255              ; largest unsigned byte
value4 SBYTE -128            ; smallest signed byte
value5 SBYTE +127            ; largest signed byte
value6 BYTE ?                ; uninitialized byte
word4  WORD  "AB"            ; double characters
val1 DWORD  12345678h        ; unsigned
val4 SDWORD -30.4            ; signed
```

MASM does not prevent you from initializing a BYTE with a negative value, but it's considered poor style.

Example 2:

Adding Variables to the AddSub Program

```
; AddVariables.asm - Chapter 3 example.

.386
.model flat,stdcall
.stack 4096
ExitProcess proto,dwExitCode:dword

.data
firstval      dword 20002000h
secondval     dword 11111111h
thirdval      dword 22222222h
sum           dword 0

.code
main proc
    mov eax, firstval
    add  eax, secondval
    add  eax, thirdval
    mov sum, eax

invoke ExitProcess,0
main endp
end main
```

Write an assembly program to add three DWORD variables named x, y and z

No more than one memory operand permitted

Arithmetic Expressions

- The compilers translate mathematical expressions into assembly language. You can do it also.
- For example:

```
Rval = -Xval + (Yval – Zval)
```

```
Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
    mov eax,Xval
    neg eax                 ; EAX = -26
    mov ebx,Yval
    sub ebx,Zval            ; EBX = -10
    add eax,ebx
    mov Rval,eax            ; -36
```

Symbolic Constants

- A symbolic constant (or symbol definition) is created by associating an **identifier** (a symbol) with an integer expression
  - Symbols do not reserve storage.
  - When a program is assembled, all occurrences of a symbol are replaced by expression
  - they **cannot change** at runtime.
- Syntax : `name = expression`
  - name is called a symbolic constant
  - The expression is a 32-bit integer (expression or constant)

```
COUNT = 500
...
mov ax,COUNT
```

label:	mnemonic	operand(s)	;comment
optional	opcode name or directive name or macro name	zero or more	optional

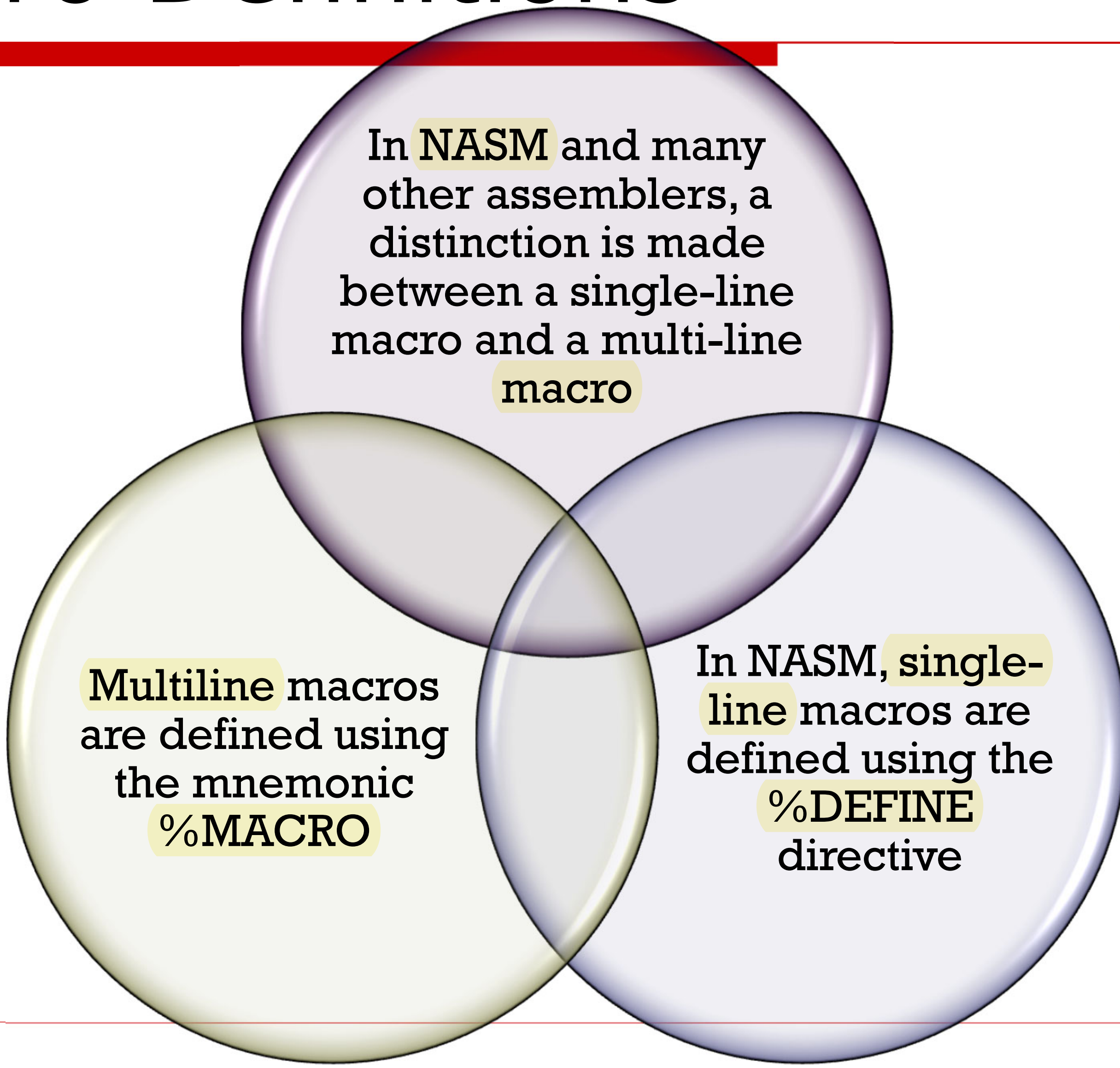
Macro Definitions

- A macro definition is similar to a subroutine in several ways
  Runtime handling
  - a section of a program that is **written once**, and can be used multiple times by calling the subroutine from any point in the program
  - When a program is compiled or assembled, the subroutine is **loaded only once**
  - A call to the subroutine **transfers control** to the subroutine and a return instruction in the subroutine **returns** control to the point of the call
- Similarly, a macro definition is a section of code that the programmer writes once, and then can use many times
  - The main difference is that when the assembler encounters a macro call, it **replaces the macro call with the macro itself**
    - **no runtime overhead of** a subroutine **call and return**
  - This process is call **macro expansion**
- Macros are handled by the assembler at **assembly time**

- Statements
- Pseudo-instructions
- Directives

label:	mnemonic	operand(s)	;comment
optional	opcode name or directive name or macro name	zero or more	optional

Macro Definitions



The Netwide Assembler (NASM) is an assembler and disassembler for the Intel x86 architecture. It can be used to write 16-bit, 32-bit and 64-bit programs. It is considered one of the most popular assemblers for Linux.

System Calls

software interrupt

- The assembler makes use of the x86 INT instruction to make system calls
- There are six **registers that store the arguments** of the system call used
  - EBX
  - ECX
  - EDX
  - ESI
  - EDI
  - EBP
- These registers **take the consecutive arguments, starting** with the **EBX** register
- If there are **more than six arguments**, then the **memory location of the first** argument is stored **in the EBX** register

Assembly Programs for Greatest Common Divisor

```
gcd:    mov     ebx,eax
        mov     eax,edx
        test    ebx,ebx
        jne     L1
        test    edx,edx
        jne     L1
        mov     eax,1
        ret
L1:     test    eax,eax
        jne     L2
        mov     eax,ebx
        ret
L2:     test    ebx,ebx
        je      L5
L3;     cmp     ebx,eax
        je      L5
        jae     L4
        sub     eax,ebx
        jmp     L3
L4:     sub     ebx,eax
        jmp     L3
L5:     ret
```

(a) Compiled program

```
gcd:    neg     eax   negation w/ 2's comp rep.
        je      L3
L1:     neg     eax
        xchg    eax,edx
L2:     sub     eax,edx
        jg      L2
        jne     L1
L3:     add     eax,edx
        jne     L4
        inc     eax
L4:     ret
```

(b) Written directly in assembly language

Euclid's Algo. for GCD:

- If m % n is 0, gcd (m, n) is n.
- Otherwise, gcd(m, n) is gcd(n, m % n).

x86 String Instructions

Operation Name	**Description**
MOVSB	**Moves the string byte addressed by** the ESI register to the location addressed by the EDI register.
CMPSB	**Subtracts** the **destination** string byte **from** the **source** string element and **updates** the status flags in the EFLAGS register according to the results.
SCASB	**Subtracts the destination string byte from** the contents of the **AL register** and **updates** the status flags according to the results.
LODSB	**Loads the source** string byte identified by the **ESI** register into the **EAX** register.
STOSSB	**Stores** the source string byte from the AL register into the memory location identified with the EDI register.
REP	**Repeat while the ECX** register is **not zero.**
REPE/REPZ	**Repeat while** the **ECX** register is not zero and the **ZF** flag is **set**.
REPNE/REPNZ	**Repeat** while the **ECX** register is not zero and the **ZF** flag is **clear**.

Assembly Program for Moving a String

```
section .text
    global  main                    ;must be declared for using gcc
main:                               ;tell linker entry point
    mov    ecx, len
    mov    esi, s1
    mov    edi, s2
    cld
    rep    movsb
    mov    edx,20                   ;message length
    mov    ecx,s2                   ;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
s1 db 'Hello, world!',0             ;string 1
len  equ    $-s1
section .bss
s2    resb 20                       ;destination
```

Assembly Language for x86 Processors

- Instructions:
  Branches and Conditions

To implement condition (if-else, switch-case), and loop (for, while) structures.

Outline

- Boolean and Comparison Instructions
- Conditional Jumps
- Conditional Structures
- The LOOP instruction

Boolean and Comparison Instructions

Perform bit-wise Boolean operations. Always clear the Overflow and Carry flags.

	Operation	Description	
AND destination, source	AND	Boolean AND operation between a source operand and a destination operand.	*Mask ==> 0 & x*
OR destination, source	OR	Boolean OR operation between a source operand and a destination operand.	*Set ==> 1 | x*
XOR destination, source	XOR	Boolean exclusive-OR operation between a source operand and a destination operand.	*Toggle==>1 XOR x*
NOT destination	NOT	Boolean NOT operation on a destination operand.	*Flip ==> X!*
TEST Input, test_value	TEST	Implied boolean AND operation between a source and destination operand, setting the CPU flags appropriately.	*ANDing ==> X & T (Flag only)*

```
              0 0 1 1 1 0 1 1
        AND   0 0 0 0 1 1 1 1
cleared ---   0 0 0 0|1 0 1 1 --- unchanged
```

```
                0 0 1 1 1 0 1 1
            OR  0 0 0 0 1 1 1 1
unchanged ---   0 0 1 1|1 1 1 1 --- set
```

```
                0 0 1 1 1 0 1 1
           XOR  0 0 0 0 1 1 1 1
unchanged ---   0 0 1 1|0 1 0 0 --- inverted
```

```
NOT   0 0 1 1 1 0 1 1
      1 1 0 0 0 1 0 0 --- inverted
```

```
0 0 1 0 0 1 0 0   <- input value (AL)
0 0 0 0 1 0 0 1   <- test value
0 0 0 0 0 0 0 0   <- result: ZF = 1
```

Dec	Hex	Oct		Char	Dec	Hex	Oct		Char
65	41	101	A	A	97	61	141	a	a
66	42	102	B	B	98	62	142	b	b
67	43	103	C	C	99	63	143	c	c

ASCII chart: case-difference=32

Applications

- Task: Convert the character in AL to upper case.
- Solution: Use the AND instruction to clear bit 5.

```
mov al,'a'                ; AL = 01100001b
and al,11011111b          ; AL = 01000001b
```

- Task: Convert a binary decimal byte into its equivalent ASCII decimal digit.
- Solution: Use the OR instruction to set bits 4 and 5.

```
mov al,6                  ; AL = 00000110b
or  al,00110000b          ; AL = 00110110b
```

The ASCII digit '6' = 00110110b

48	30	060	0	0
49	31	061	1	1
50	32	062	2	2
51	33	063	3	3
52	34	064	4	4
53	35	065	5	5
54	36	066	6	6
55	37	067	7	7
56	38	070	8	8
57	39	071	9	9

JMP Instruction

- JMP is an <u>unconditional</u> jump to a label that is usually within the same procedure.
  - Syntax: JMP *target*
- Logic: EIP ← *target*
- Example:

```
top:
    .
    .
    jmp top
```

Conditional jumps

- A conditional jump instruction branches to a label when specific register or flag conditions are met

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

Z = X & Y
// stores results in Z and affects EFLAGS

test X, Y
// dont forget, one operand must be a register
// Doesn't store result, and affects EFLAGS ONLY

TEST Instruction

- Performs a nondestructive AND operation between each pair of matching bits in two operands
  - No operands are modified, but the Zero flag is affected.
- Use a *bit mask*

```
test al,00001001b
jz   ValueNotFound
```

```
test al,00001001b
jnz   ValueFound
```

```
0 0 1 0 0 1 0 1   <- input value (AL)
0 0 0 0 1 0 0 1   <- test value
0 0 0 0 0 0 0 1   <- result: ZF = 0

0 0 1 0 0 1 0 0   <- input value (AL)
0 0 0 0 1 0 0 1   <- test value
0 0 0 0 0 0 0 0   <- result: ZF = 1
```

Applications

- Task: Jump to a label if the value in AL is not zero.
- Solution: OR the byte with itself, then use the JNZ (jump if not zero) instruction.

CF and OF are always cleared!
ZF, SF, PF are set accordingly.

```
or  al,al
jnz IsNotZero              ; jump if not zero
```

ORing any number with itself does not change its value.

Applications

- Task: Jump to a label if an integer is even.
- Solution: AND the lowest bit with a 1. If the result is Zero, the number was even.

CF and OF are always cleared!
ZF, SF, PF are set accordingly.

```
mov ax,wordVal
and ax,1                    ; low bit set?
jz  EvenValue               ; jump if Zero flag set
```

Your turn: Write code that jumps to a label if an integer is negative.

Applications

- Jump to label L2 if contents of eAx is even

```
test eAx,1
jz   L2
```

- Jump to label L1 if contents of eAx equals Zero

```
Test eAx,11111111x
jz  L1
```

CMP Instruction

EFLAGS <---- D - S

- Compares the destination operand to the source operand
  - Nondestructive <u>subtraction</u> of source from destination (destination operand is not changed)
- Syntax: CMP destination, source

CMP Results	ZF	CF
Destination < source	0	1
Destination > source	0	0
Destination = source	1	0

borrow!

When two unsigned operands are compared

CMP Results	Flags
Destination < source	SF ≠ OF
Destination > source	SF = OF
Destination = source	ZF = 1

When two signed operands are compared

x, y sign same but z sign different ==> invalid z; OF detected

+X +Y (CF=0, SF=1) ==> Z=result + 2^8

-X -Y (CF=1, SF=0) ==> Z=result - 2^8

Jumps Based on Equality

un-/signed

Mnemonic	Description
JE	Jump if equal ($leftOp = rightOp$)
JNE	Jump if not equal ($leftOp \neq rightOp$)
JCXZ	Jump if CX = 0 for counters and loops
JECXZ	Jump if ECX = 0 for counters and loops

Jumps Based on Signed Comparisons

greater/lesser

Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* >= *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* <= *rightOp*)
JNG	Jump if not greater (same as JLE)

Jumps Based on Unsigned Comparisons

above/below

Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* >= *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* <= *rightOp*)
JNA	Jump if not above (same as JBE)

Applications

- Task: Jump to a label if unsigned EAX is greater than EBX
- Solution: Use CMP, followed by JA

```
cmp eax,ebx
ja  Larger
```

- Task: Jump to a label if signed EAX is less than EBX
- Solution: Use CMP, followed by JL

```
cmp  eax,ebx
jl  Lesser
```

Applications

- Jump to label L1 if unsigned EAX is less than or equal to Val1

```
cmp eax,Val1
jbe L1                ; below or equal
```

- Jump to label L1 if signed EAX is equal to Val1

```
cmp eax,Val1
je L1
```

CONDITIONAL STRUCTURES

- Block-Structured IF Statements
- Compound Expressions with AND /OR
- WHILE Loops
- LOOP instruction

Block-Structured IF Statements

- Assembly language programmers can easily translate logical statements written in C++/Java into assembly language.

un-/signed !

- For example:

```
if( op1 == op2 )
  X = 1;
```

```
    mov eax,op1
    cmp eax,op2
    jne EndIF
    mov X,1
EndIf:
```

Applications

- Compare unsigned AX to BX, and copy the larger of the two into a variable named <u>Large</u>

```
    mov Large,bx
    cmp ax,bx
    jna Next
    mov Large,ax
Next:
```

- Compare signed AX to BX, and copy the smaller of the two into a variable named <u>Small</u>

```
    mov Small,ax
    cmp bx,ax
    jnl Next
    mov Small,bx
Next:
```

Your turn . . .

- Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx )
{
  eax = 5;
  edx = 6;
}
```

```
    cmp ebx,ecx
    ja  next
    mov eax,5
    mov edx,6
next:
```

(There are multiple correct solutions to this problem.)

Block-Structured IF Statements

- Assembly language programmers can easily translate logical statements written in C++/Java into assembly language.
- For example:

```
if( op1 == op2 )
  X = 1;
else
  X = 2;
```

un-/singed

```
    mov eax,op1
    cmp eax,op2
    jne Else
    mov X,1
    jmp EndIf
Else:
    mov X,2
EndIf :
```

Your turn . . .

- Implement the following pseudocode in assembly language. All values are 32-bit signed integers:

```
if( var1 <= var2 )
  var3 = 10;
else
{
  var3 = 6;
  var4 = 7;
}
```

```
    mov eax,var1
    cmp eax,var2
    jle L1
    mov var3,6
    mov var4,7
    jmp L2
L1: mov var3,10
L2:
```

(There are multiple correct solutions to this problem.)

Compound Expression with AND

- When implementing the logical AND operator, consider using short-circuit evaluation
- In the following example, if the first expression is false, the second expression is skipped:

```
if (al > bl) AND (bl > cl)
  X = 1;
```

Example

```
if (al > bl) AND (bl > cl)
  X = 1;
```

let, 8-bit registers are pre-loaded with unsigned integers

This is one possible implementation . . .

```
    cmp al,bl                ; first expression...
    ja  L1
    jmp next
L1:
    cmp bl,cl                ; second expression...
    ja  L2
    jmp next
L2:                          ; both are true
    mov X,1                  ; set X to 1
next:
```

Another !

```
if (al > bl) AND (bl > cl)
  X = 1;
```

```
    cmp al,bl                    ; first expression...
    jbe next                     ; quit if false
    cmp bl,cl                    ; second expression...
    jbe next                     ; quit if false
    mov X,1                      ; both are true
next:
```

`seven instructions before vs. five here`

the above implementation uses 29% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:

Your turn . . .

- Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx &&
    ecx > edx )
{
  eax = 5;
  edx = 6;
}
```

```
    cmp ebx,ecx
    ja  next
    cmp ecx,edx
    jbe next
    mov eax,5
    mov edx,6
next:
```

(There are multiple correct solutions to this problem.)

Compound Expression with OR

- When implementing the logical OR operator, consider using short-circuit evaluation
- In the following example, if the first expression is true, the second expression is skipped:

```
if (al > bl) OR (bl > cl)
  X = 1;
```

```
    cmp al,bl              ; is AL > BL?
    ja  L1                 ; yes
    cmp bl,cl              ; no: is BL > CL?
    jbe next               ; no: skip next statement
L1: mov X,1                ; set X to 1
next:
```

WHILE Loops

- A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump back to the top of the loop.
- Consider the following example:

```
while( eax < ebx)
    eax = eax + 1;
```

```
top: cmp eax,ebx          ; check loop condition
     jae next             ; false? exit loop
     inc eax              ; body of loop
     jmp top              ; repeat the loop
next:
```

Your turn . . .

- Implement the following loop, using unsigned 32-bit integers:

```
while( ebx <= val1)
{
    ebx = ebx + 5;
    val1 = val1 - 1
}
```

```
top:cmp ebx,val1          ; check loop condition
    ja  next              ; false? exit loop
    add ebx,5             ; body of loop
    dec val1
    jmp top               ; repeat the loop
next:
```

LOOP Instruction

- The LOOP instruction creates a counting loop
- Syntax:

```
LOOP target
```

- Logic:

```
ECX ← ECX – 1
if ECX != 0, jump to target
```

- Implementation:
  - The assembler calculates the distance, in bytes, between the offset of the following instruction and the offset of the target label. It is called the relative offset.
  - The relative offset is added to EIP.

Examples . . .

- *What will be the final value of AX?*

ax	ecx
6	4
7	3
8	2
9	1
10	0

```
    mov ax,6
    mov ecx,4
L1:
    inc ax
    loop L1
```

- *How many times will the loop execute?*

2^{32} times looping

```
    mov ecx,0
X2:
    inc ax
    loop X2
```

Write a program that sums the first 10 natural numbers

Solution

```
.data

.code
main PROC
    mov Ax, 0              ; zero the accumulator
    mov cx,10              ; loop counter

    L1:
        add ax,cx          ; add an integer
        LOOP L1            ; repeat until ECX = 0

    INVOKE ExitProcess, 0
main ENDP
END main
```

ax	cx
0	10
10	9
19	8
27	7
.	.
.	.
.	.
54	1
55	0

Nested Loops

- If you need to code a loop within a loop, you must save the outer loop counter's ECX value.

```
.data
count DWORD ?
.code
    mov ecx,10              ; set outer loop count
L1:
    mov count,ecx           ; save outer loop count
    mov ecx,20              ; set inner loop count
L2: .
    .
    loop L2                 ; repeat the inner loop
    mov ecx,count           ; restore outer loop count
    loop L1                 ; repeat the outer loop
```

Write a program that sums the first 10 natural numbers 5 times.

Solution

```
.data
; (insert variables here)
count DWORD 5

.code
main PROC
     mov Ax, 0       ; zero the accumulator; holds total result
     mov cx,count    ; outer loop counter
     L1:

       mov count, ecx      ; save the current iteration
      mov cx, 10           ; inner loop counter
       mov bx, 0         ; holds partial result
     L2:
      add bx,cx            ; add an integer
      LOOP L2              ; repeat until ECX = 0
      add Ax, bx           ;update the accumulator/total result
      mov ecx, count
      LOOP L1
```



```
INVOKE ExitProcess, 0

main ENDP
```

Summary

- Bitwise instructions (AND, OR, XOR, NOT, TEST)
  - manipulate individual bits in operands; update result/EFLAGS
- CMP – compares operands using implied subtraction
  - sets condition flags only
- Conditional Jumps
  - equality: JE, JNE
  - flag values: JC, JZ, JNC, JP, ...
  - signed: JG, JL, JNG, ...
  - unsigned: JA, JB, JNA, ...
- LOOP – branching instructions

Assembly Language for x86 Processors

- Arrray; Data-related Operators and Directives
- Addressing

A collection of data that has the same type

Defining Arrays

- Arrays use multiple initializers:

```
list1 BYTE 10,20,30,40
list2 BYTE 10,20,30,40
      BYTE 50,60,70,80
      BYTE 81,82,83,84
list3 BYTE ?,32,41h,00100010b
list4 BYTE 0Ah,20h,'A',22h
myList WORD  1,2,3,4,5 ; array of words
val4 SDWORD -3,-2,-1,0,1   ; signed array
```

Offset	Value
0000:	10
0001:	20
0002:	30
0003:	40

Using the DUP Operator

- Use DUP to allocate (create space for) an array or string.
- Syntax:
  - counter DUP (argument)
- Counter and argument must be constants or constant expressions

```
var1 BYTE 20 DUP(0)          ; 20 bytes, all equal to zero
var2 BYTE 20 DUP(?)          ; 20 bytes, uninitialized
var3 BYTE 4 DUP("STACK")     ; 20 bytes: "STACKSTACKSTACKSTACK"
var4 BYTE 10,3 DUP(0),20     ; 5 bytes
```

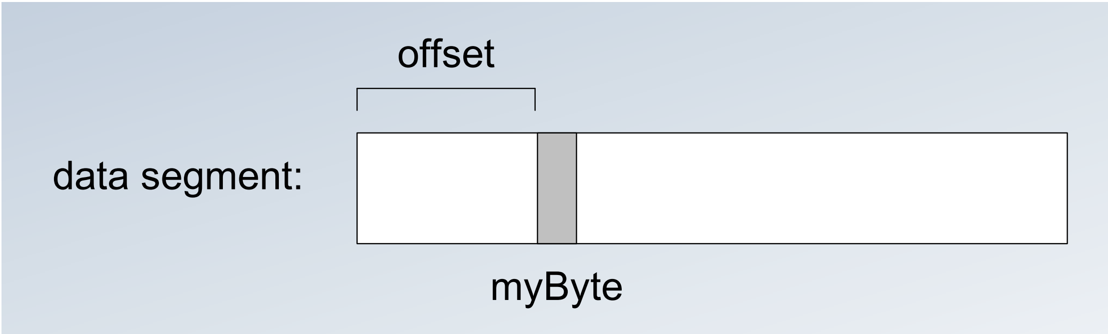
Defining Strings

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It is often null-terminated
- Examples:

```
str1 BYTE "Enter your name",0
str2 BYTE 'Error: halting program',0
str3 BYTE 'A','E','I','O','U'
greeting  BYTE "Welcome to the Encryption Demo program "
          BYTE "created by Kip Irvine.",0
```

OFFSET Operator

- OFFSET returns the distance in bytes, of a label from the beginning of its enclosing segment
- The value returned by OFFSET is a pointer.




```
// C++ version:

char array[1000];
char * p = array;
```

```
; Assembly language:

.data
array BYTE 1000 DUP(?)
.code
mov esi,OFFSET array
```

Examples

- Let's assume that the data segment begins at 00404000h:

```
.data
bVal BYTE ?
wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?
.code
mov esi,OFFSET bVal
                            ; ESI = 00404000
mov esi,OFFSET wVal
                            ; ESI = 00404001
mov esi,OFFSET dVal
                            ; ESI = 00404003
mov esi,OFFSET dVal2
                            ; ESI = 00404007
```

TYPE Operator

- The TYPE operator returns the size (in bytes) of a single element of a data declaration.

```
.data
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?

.code
mov eax,TYPE var1          ; 1
mov eax,TYPE var2          ; 2
mov eax,TYPE var3          ; 4
mov eax,TYPE var4          ; 8
```

LENGTHOF Operator

- The `LENGTHOF` operator counts the number of elements in a single data declaration.

```
.data                                           LENGTHOF
byte1  BYTE 10,20,30                            ; 3
array1 WORD 30 DUP(?),0,0                       ; 32
array2 WORD 5 DUP(3 DUP(?))                     ; 15
array3 DWORD 1,2,3,4                            ; 4
digitStr BYTE "12345678",0                      ; 9

.code
mov ecx,LENGTHOF array1                         ; 32
```

SIZEOF Operator

- The SIZEOF operator returns a value that is equivalent to multiplying `LENGTHOF` by `TYPE`.

```
.data                                       SIZEOF
byte1   BYTE 10,20,30                       ; 3
array1 WORD 30 DUP(?),0,0                   ; 64
array2 WORD 5 DUP(3 DUP(?))                 ; 30
array3 DWORD 1,2,3,4                        ; 16
digitStr BYTE "12345678",0                  ; 9

.code
mov ecx, SIZEOF array1                      ; 64
```

Spanning Multiple Lines

- A data declaration can span multiple lines if each line (except the last) ends with a comma.
- The LENGTHOF and SIZEOF operators include all lines belonging to the declaration:

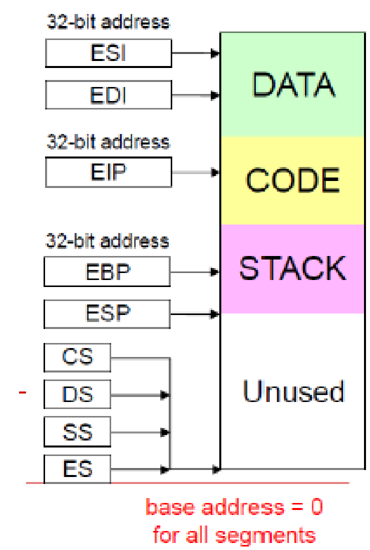
```
.data
array WORD 10,20,
           30,40,
           50,60

.code
mov eax,LENGTHOF array              ; 6
mov ebx,SIZEOF array                ; 12
```

ADDRESSING MODES

Review:

Ch-4 MARIE

Jump X: PC <-- X
JnS X: M[X] <-- PC ;   PC <-- X+1
JumpI X: PC <-- M[X]



Ch-5 ISA

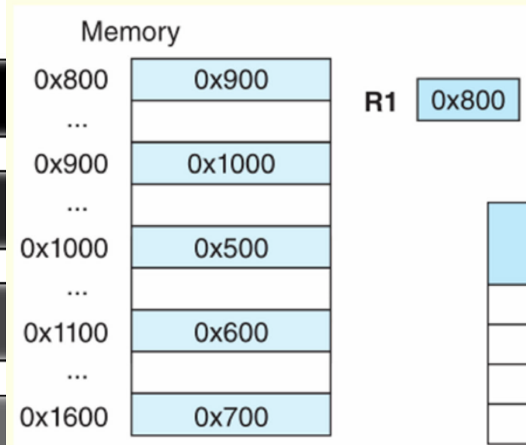
Immediate:	#	operand is the **value**
Direct:	X	operand is the address
Indirect:	M[X]	operand is the address of the address
Register:	R1	register is the address
Reg. Indir:	M[R1]	register data is the address
Indexed:	X+Roi	register is the index/offset to the address in the operand X
Base:	Rb+D	register is the base address and the operand D is the displacement

Addressing Modes

- The address field or fields in a typical instruction format are relatively small → various modes of addressing

Ch 5.4

- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack



LOAD 800

Mode	Value Loaded into AC
Immediate	0x800
Direct	0x900
Indirect	0x1000
Indexed	0x700

R1 is the index register

Direct Memory Operands

- A direct memory operand is a named reference (variable) to storage in memory
- The variable is automatically dereferenced by the assembler
  - After dereferencing, its value can be obtained

```
.data
var1 BYTE 010h
.code
mov al, var1                    ; After moving, AL = 010h
mov al,[var1]                   ; After moving, AL = 010h
```

alternate format

Direct-Offset Operands

Direct-Immediate offset

- A constant offset is added to a data label to produce an effective address (EA).
  - The offset are 0, 1, 2, …..
- The address is dereferenced to get the value inside its memory location.

```
.data
arrayB BYTE 010h,020h,030h,040h
.code
mov al,arrayB+1                ; AL = 020h
mov al,[arrayB+1]              ; alternative notation
```

Q : Why doesn't arrayB+1 produce 11h?

Your turn . . .

```
.data
arrayW  WORD 01000h,02000h,03000h
arrayD  DWORD 1,2,3,4
.code
mov ax, arrayW                  ;
mov ax,[arrayW+2]               ;
mov ax,[arrayW+4]               ;
mov eax,[arrayD+4]              ; EAX = 00000002h
```

What will happen when they run?

Write a program that sums the elements of a ~~WORD~~ array that is initialized with 080h,066h,0A5h

BYTE

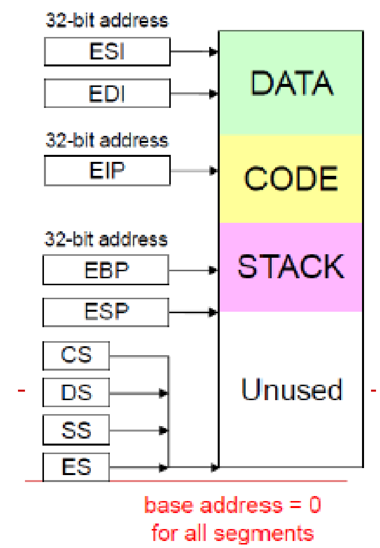
Use base addressing

Solution(s)

```
.data
myBytes BYTE 080h,066h,0A5h
```

```
mov al, myBytes          ;al=080h
add al,[myBytes+1]       ;al=0E6h
add al,[myBytes+2]       ;al=018bh
```

Any other possibilities?



Solution

- **Step1:** copy the 1st element into EAX and exchange it with the element in the 2nd position.
- **Step 2:** Exchange EAX with the 3rd element and copy the element in EAX to the first array position.

```
.data
   arrayD DWORD 1,2,3
.code
       mov eax,arrayD
       xchg eax,[arrayD+4]
       xchg eax,[arrayD+8]
       mov  arrayD,eax
```

Your turn...

- Show the value of the destination operand after each of the following instructions executes:

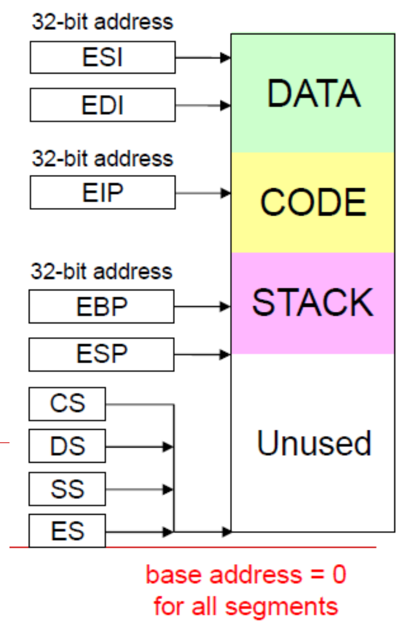
```
.data
myByte BYTE 0FFh, 0
.code
    mov al,myByte
    mov ah,[myByte+1]
    dec ah
    inc al
    dec ax
```

```
; AL =  FFh
; AH =  00h
; AH =  FFh
; AL =  00h
; AX =  FEFF
```

Indirect Operands

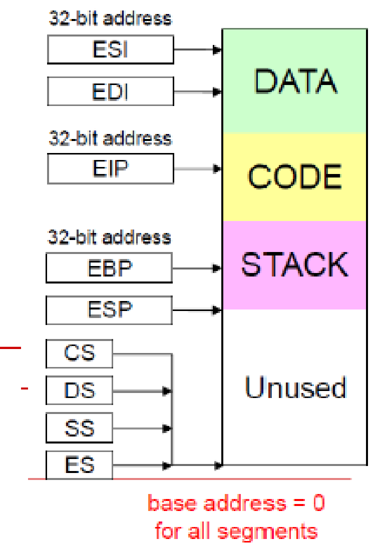
- An indirect operand holds the address of a variable, usually an array or string.
- It can be dereferenced by the assembler (just like a pointer).

```
.data
val1 BYTE 010h,020h,030h
.code
mov esi,OFFSET val1
mov al,[esi]                    ; dereference ESI (AL = 10h)

inc esi
mov al,[esi]                    ; AL = 020h

inc esi
mov al,[esi]                    ; AL = 030h
```



Write a program that sums the elements of a WORD array that is initialized with 01000h,02000h,03000h

Use indirect addressing



Solution

```
.data
    arrayW WORD 01000h,02000h,03000h
.code
    mov esi,OFFSET arrayW
    mov ax,[esi]
    add esi,2                    ; or: add esi,TYPE arrayW
    add ax,[esi]
    add esi,2
    add ax,[esi]                 ; AX = sum of the array
```

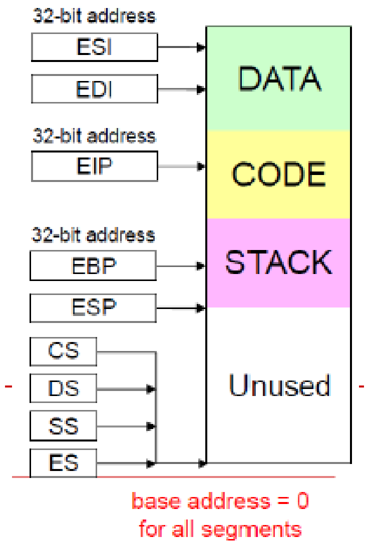
The register in brackets must be incremented by a value that matches the array type

Indexed Operands

- An indexed operand adds a constant to a register to generate an effective address.
- There are two notational forms:

  `[label + reg]`

  `label[reg]`



- example

```
.data
arrayW WORD 01000h,02000h,03000h
.code
    mov esi,0
    mov ax,[arrayW + esi]          ; AX = 1000h
    mov ax, arrayW[esi]            ; alternate format
    add esi,2
    add ax,[arrayW + esi]
    etc.
```

Index Scaling

- You can scale an indirect or indexed operand to the offset of an array element.
  - This is done by multiplying the index by the array's TYPE:

```
.data
arrayB BYTE  0,1,2,3,4,5
arrayW WORD  0,1,2,3,4,5
arrayD DWORD 0,1,2,3,4,5
.code
mov esi,4                              ; 5th element
mov al,arrayB[esi*TYPE arrayB]         ; 04
mov bx,arrayW[esi*TYPE arrayW]         ; 0004
mov edx,arrayD[esi*TYPE arrayD]    ; 00000004
```

Write a program that sums the elements of a WORD array
that is initialized with 100h,200h,300h,400h

Use index addressing

Solution

- calculate the sum of an array of 16-bit integers using LOOP

```
.data
intarray WORD 100h,200h,300h,400h
.code
    mov edi,OFFSET intarray         ; address of intarray
    mov ecx,LENGTHOF intarray       ; loop counter
    mov ax,0                        ; zero the accumulator
L1:
    add ax,[edi]                    ; add an integer
    add edi,TYPE intarray           ; point to next integer
    loop L1                         ; repeat until ECX = 0
```

What changes would you make to the program on the previous slide if you were summing a double-word array?

Solution

good use of SIZEOF

```
.data
source  BYTE  "This is the source string",0
target  BYTE  SIZEOF source DUP(0)

.code
    mov  esi,0                      ; index register
    mov  ecx,SIZEOF source          ; loop counter
L1:
    mov  al,source[esi]             ; get char from source
    mov  target[esi],al             ; store it in the target
    inc  esi                        ; move to next character
    loop L1                         ; repeat for entire string
```

Rewrite the program shown in the previous slide, using indirect addressing rather than indexed addressing.