Assembly Language for x86 Processors

- Assembly Language fundamentals

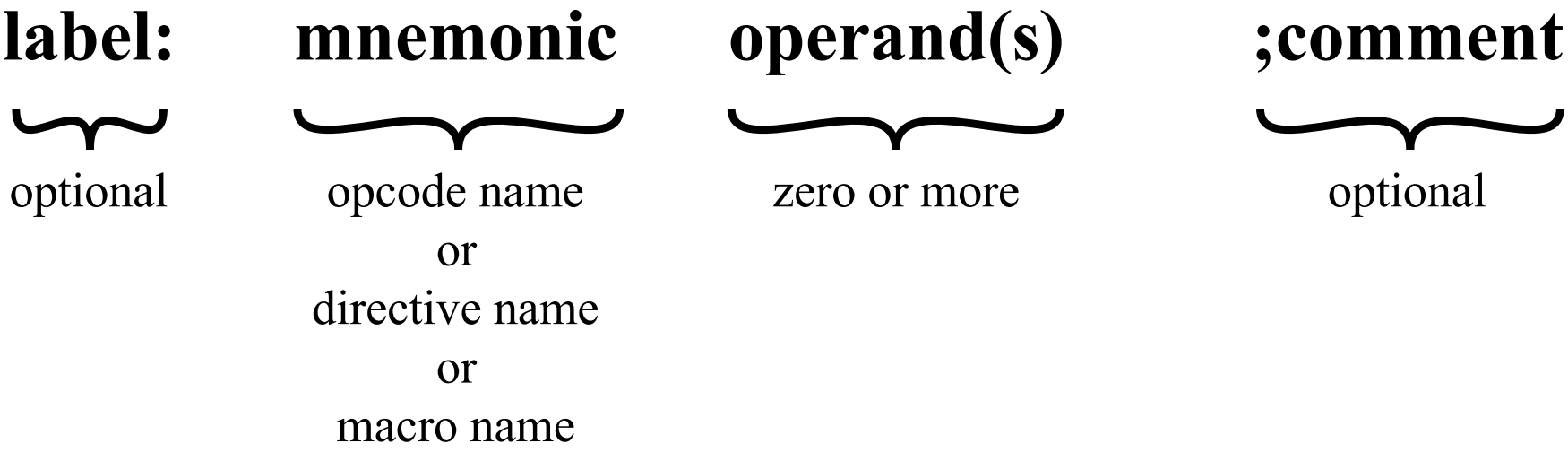
x86 Assembly Language: Outline

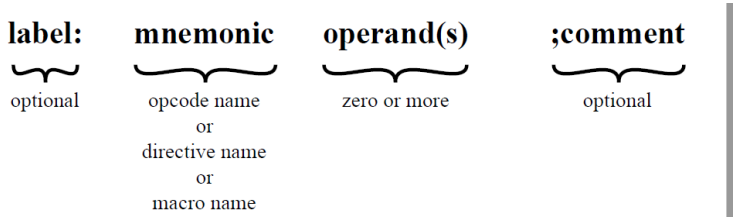
- Statements
- Pseudo-instructions
- Directives

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

Assembly-Language Statement Structure

- The heart of any assembly language program are statements





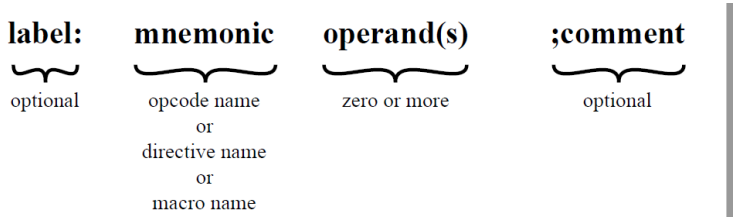
Statements, Label

- 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
```

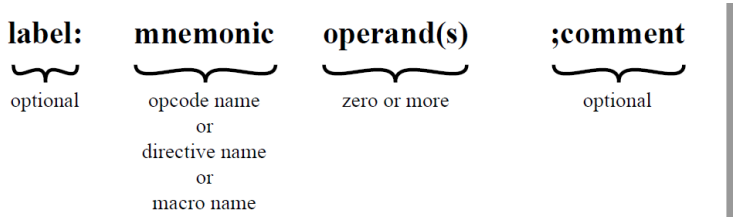
  - Code label
    - Mark of a memory location for jump and loop instructions
    - example:

```
L1:     (followed by colon)

        proc_name:
            procedure body
            ...
            ret
```



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



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
IMUL Op	Signed integer multiplication. W, 4 Ws, 4 DWs, 4 QWs concatenated!
DIV Op	**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.

x86 Instruction Set, Logical

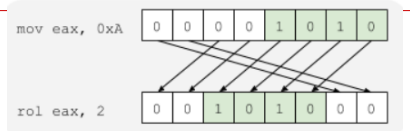
Operation Name	Description
NOT Op	Inverts each bit of the operand.
AND Dest, Source	Performs a bitwise AND operation on the destination and source operands and stores the result in the destination operand.
OR Dest, Source	Performs a bitwise OR operation on the destination and source operands and stores 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	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.

SAL: shift arithmetic left
SHL: shift logical left



ROL

RCL=Rotate with Carry to Left

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

x86 Instruction Set, Transfer of Control

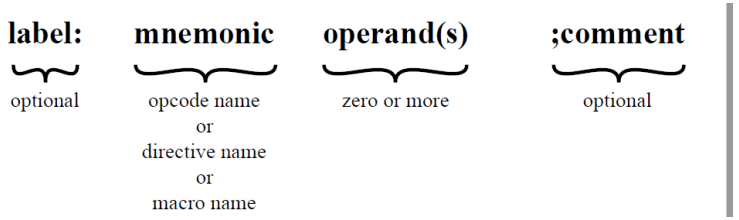
Operation Name	Description
CALL proc	**Saves procedure linking information** on the stack, and branches to the called procedure specified using the operand. The operand specifies the address of the first instruction in the called procedure.
RET	Transfers program control to a **return** address located 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

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

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.



Statements, operands

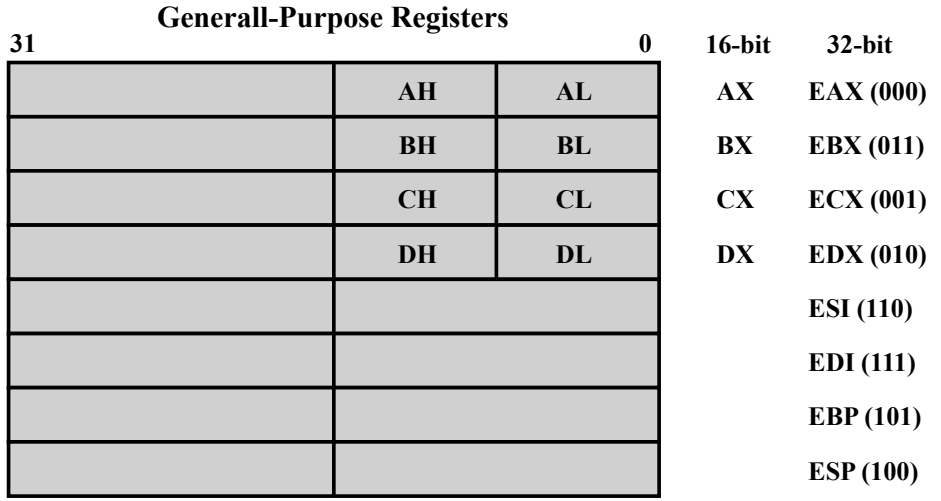
- An assembly language statement includes zero or more operands
- Each operand identifies:
  - **immediate** value,
  - a **register** value, or
  - a **memory** location
- Typically the assembly language provides **conventions**:
  - for distinguishing among the three **types** of operand references,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

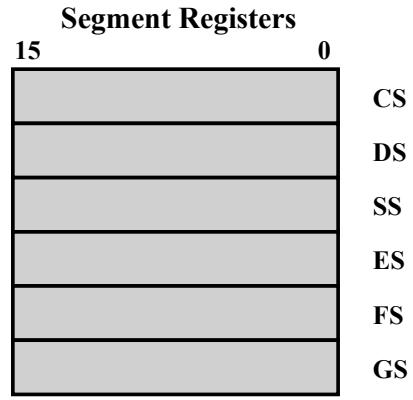
Immediate values

- **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



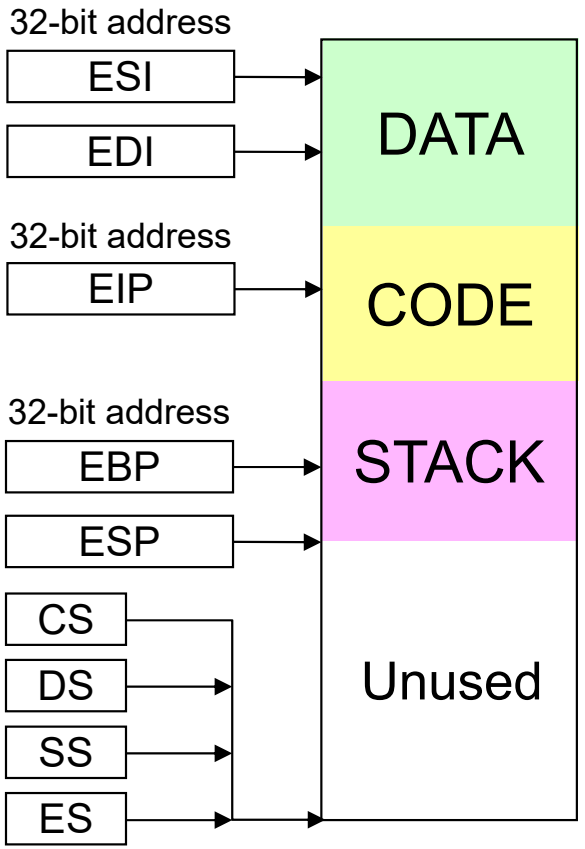


label:	mnemonic	operand(s)	;comment
optional	opcode name or directive name or macro name	zero or more	optional

Segment Registers

- Six 16-bit Segment Registers
  - Support segmented memory
  - Segments contain distinct contents
    - Code , Data , Stack
- EIP Register
  - Points at next instruction
- ESI and EDI Registers
  - Contain data addresses
  - Used also to index arrays
- ESP and EBP Registers
  - ESP points at top of stack
  - EBP is used to address parameters and variables on the stack

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



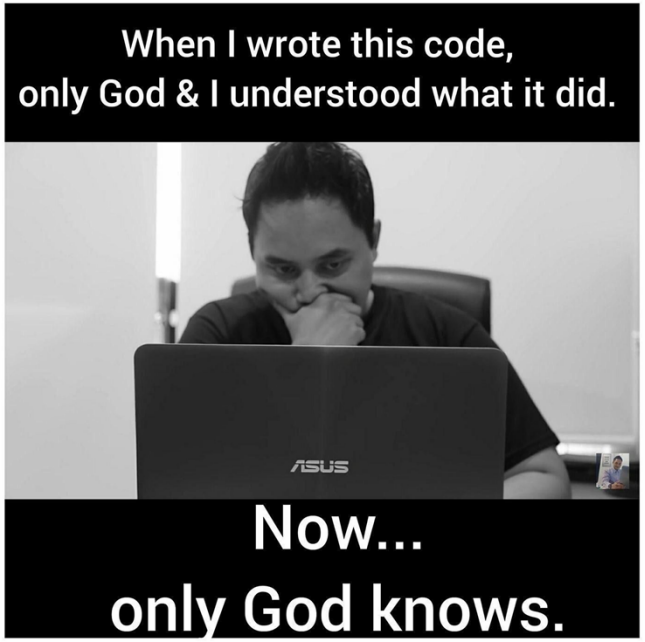
base address = 0 for all segments

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

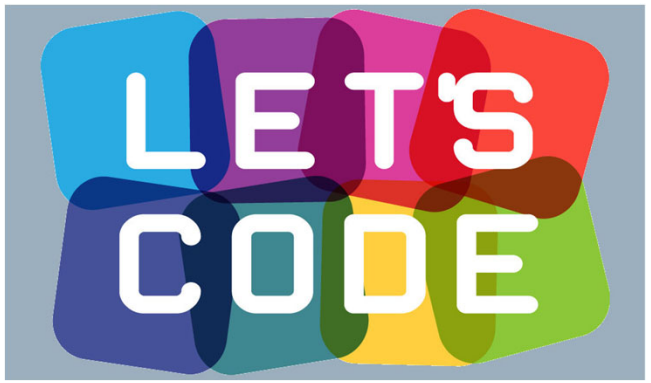
```
TITLE Program Template                (Template.asm)

; 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



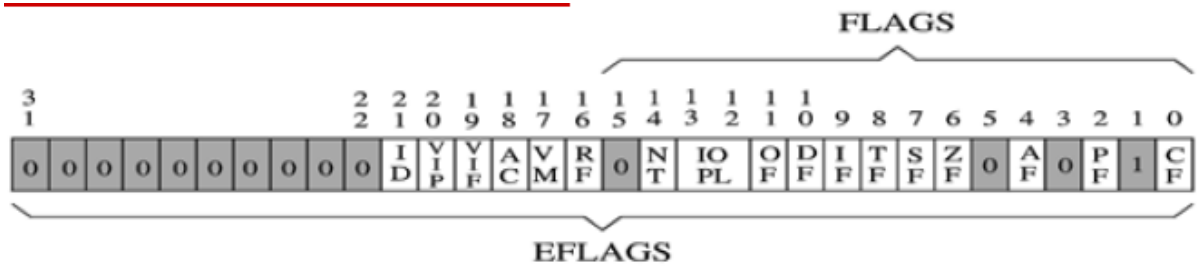
Write an assembly program to add the values 5 and 6 and store the value in eAx

```
TITLE Add (AddTwo.asm)

; This program adds two 32-bit integers.

.386
.model flat, stdcall          (memory model, calling convention)
.stack 4096
ExitProcess proto, dwExitCode:dword   --> standard Windows service
DumpRegs PROTO                        prototype declaration

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

end main
```

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] . . .
```

- All initializers become binary data in memory

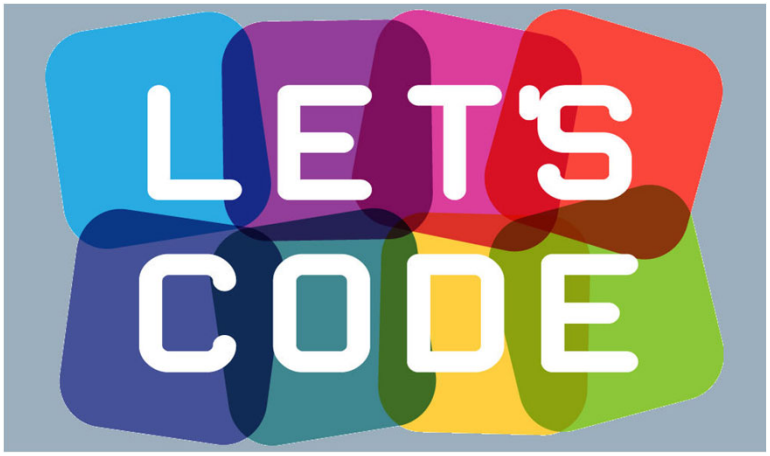
Examples

```
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:

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



No more than one memory operand permitted

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
```

No more than one memory operand permitted

Arithmetic Expressions

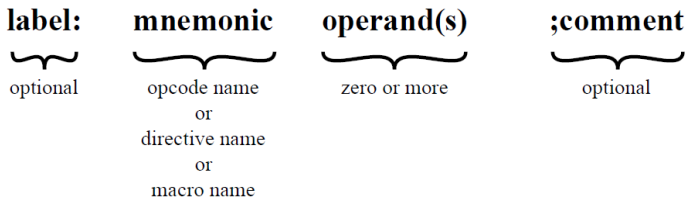
- The compilers translate mathematical expressions into assembly language. You can do it also.
- For example:

```
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
```



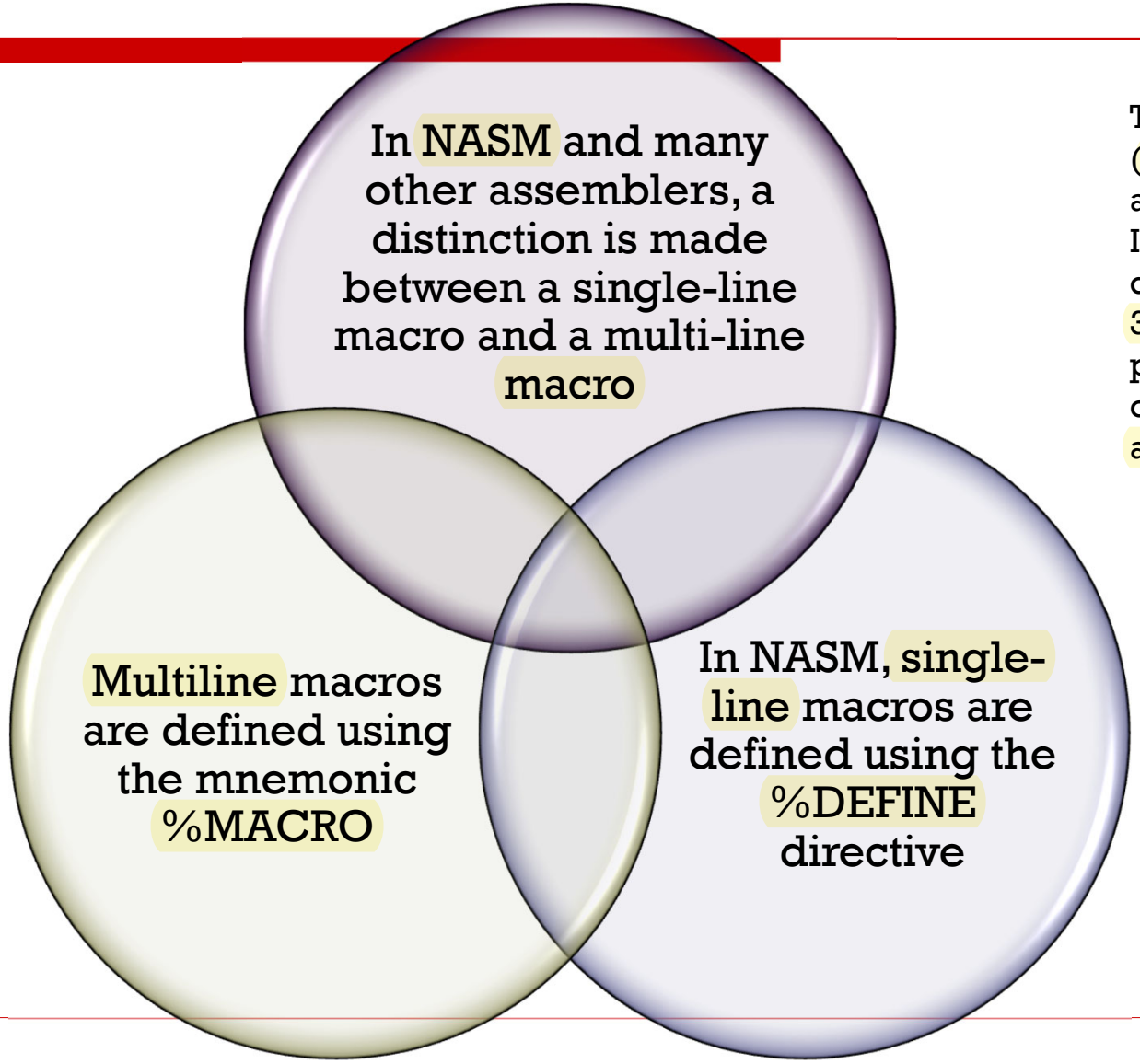
Macro Definitions

- 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

Example-1

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
```

Euclid's Algo. for GCD:

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

(b) Written directly in assembly language

Example-2a

C Program for Generating Prime Numbers

```
unsigned guess;              /* current guess for prime */
unsigned factor ;            /* possible factor of guess */
unsigned limit ;             /* find primes up to this value */

printf (”Find primes up to : ”);
scanf(”%u”, &limit);
printf (”2\n”);              /* treat first two primes as */
printf (”3\n”);              /* special case */
guess = 5;                   /* initial guess */
while ( guess <= limit ) {     /* look for a factor of guess */
   factor = 3;
   while ( factor * factor < guess && guess % factor != 0 )
   factor += 2;
   if ( guess % factor != 0 )
       printf (”%d\n”, guess);
   guess += 2;               /* only look at odd numbers */
}
```

Example-2b

Assembly Program for Generating Prime Numbers

```
%include "asm_io.inc"
segment .data
Message db "Find primes up to: ", 0

segment .bss
Limit resd 1                            ; find primes up to this limit
Guess resd 1                            ; the current guess for prime

segment .text
       global _asm_main
_asm_main:
       enter 0,0                        ; setup routine
       pusha

       mov eax, Message
       call print_string
       call read_int                    ; scanf("%u", & limit );
       mov [Limit], eax
       mov eax, 2                       ; printf("2\n");
       call print_int
       call print_nl
       mov eax, 3                       ; printf("3\n");
       call print_int
       call print_nl

       mov dword [Guess], 5             ; Guess = 5;
while_limit:                            ; while ( Guess <= Limit )
       mov eax,[Guess]
       cmp eax, [Limit]
       jnbe end_while_limit             ; use jnbe since numbers are unsigned

       mov ebx, 3                       ; ebx is factor = 3;
while_factor:
       mov eax,ebx
       mul eax                          ; edx:eax = eax*eax
       jo end_while_factor              ; if answer won't fit in eax alone
       cmp eax, [Guess]
       jnb end_while_factor             ; if !(factor*factor < guess)
       mov eax,[Guess]
       mov edx,0
       div ebx                          ; edx = edx:eax % ebx
       cmp edx, 0
       je end_while_factor              ; if !(guess % factor != 0)

       add ebx,2 ; factor += 2;
       jmp while_factor
end_while_factor:
       je end_if                        ; if !(guess % factor != 0)
       mov eax,[Guess]                  ; printf("%u\n")
       call print_int
       call print_nl
end_if:
       add dword [Guess], 2             ; guess += 2
       jmp while_limit
end_while_limit:

       popa
       mov eax, 0                       ; return back to C
       leave
       ret
```

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**.

Example-3

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
```