Assembly Language for x86 Processors

- Assembly Language fundamentals

CS 271 Computer Architecture
Purdue University Fort Wayne

Outline

- Statements
- Pseudo-instructions
- Directives

Assembly-Language Statement Structure

- The heart of any assembly language program are statements

label: **mnemonic** **operand(s)** **;comment**

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

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

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

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.
IMUL Op	Signed integer multiplication.
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 (AH:AL), 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, 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.

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.

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 one or more of the 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 specified interrupt program

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.
XOR 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

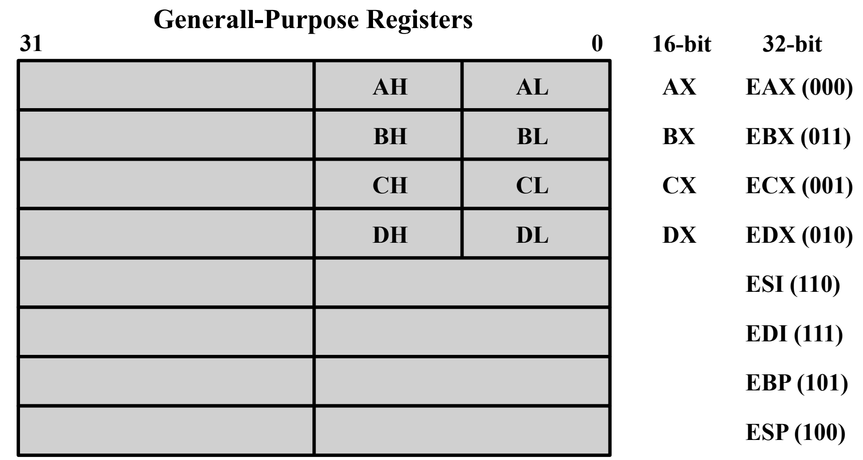
- An assembly language statement includes zero or more operands
- Each operand identifies:
  - immediate value,
  - a register value, or
  - a memory location
- Typically the assembly language provides conventions:
  - for distinguishing among the three types of operand references,
  - for indicating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

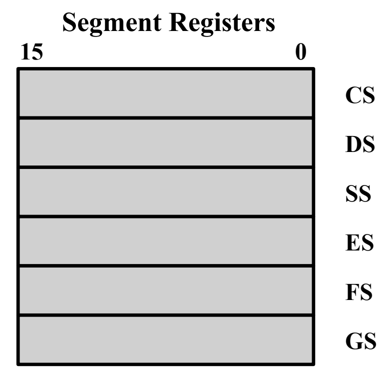
Immediate values

- 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



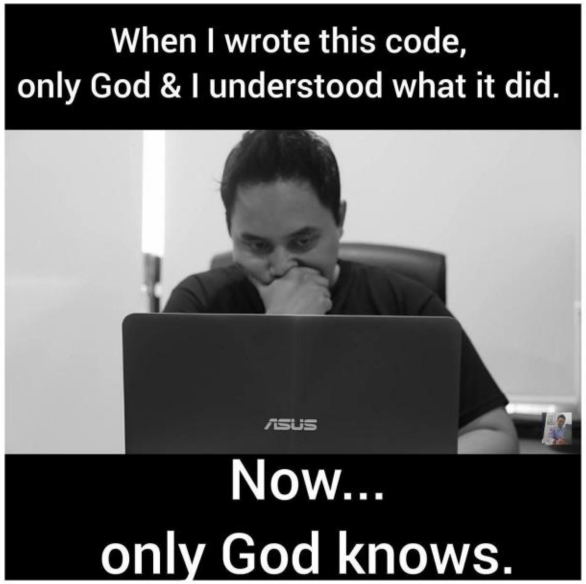


Identifiers and Reserved Words

- Identifiers
  - 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 and 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

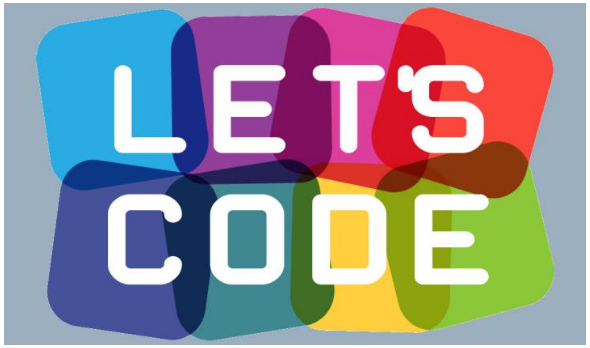
Program Template

```
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

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
```

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

```
end main
```

Example 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
```

Statements, Pseudo-instructions

- Pseudo-instructions/ 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
```

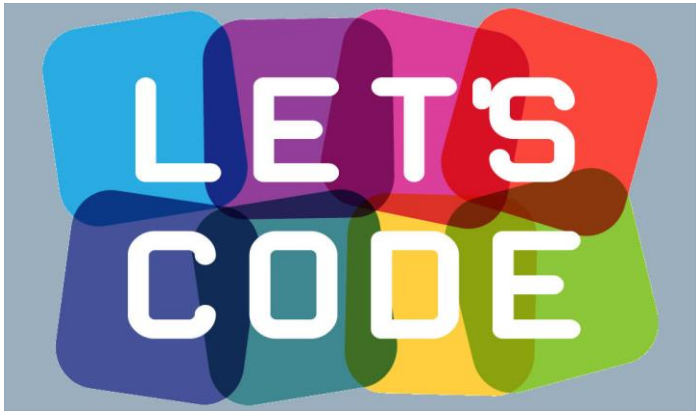
- All initializers become binary data in memory

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.

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
```

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
```