Assembly Language Statements in MASM Visual Studio Community

Assembly Language Statements

```
; Program to add 158 to number in memory
; Author: R. Detmer    Date:  6/2013
.586
.MODEL FLAT
.STACK 4096                 ; reserve 4096-byte stack
.DATA                       ; reserve storage for data
number DWORD  -105
sum    DWORD  ?
.CODE                       ; start of main program code
main  PROC
      mov   eax, number     ; first number to EAX
      add   eax, 158        ; add 158
      mov   sum, eax        ; sum to memory
      mov   eax, 0          ; exit with return code 0
      ret
main  ENDP
END
```

Comments

- Start with a semicolon (;)
- Extend to end of line
- May follow other statements on a line

Instructions

- Each corresponds to a single instruction actually executed by the 80x86 CPU
- Examples
  - `mov eax, number`

    copies a doubleword from memory to the accumulator EAX
  - `add eax, 158`

    adds the doubleword representation of 158 to the number already in EAX, replacing the number in EAX

Directives

- Provide instructions to the assembler program
- Typically don’t cause code to be generated
- Examples
  - .586 tells the assembler to recognize 32-bit instructions
  - DWORD tells the assembler to reserve space for a 32-bit integer value

Macros

- Each is “shorthand” for a sequence of other statements – instructions, directives, or even other macros
- The assembler expands a macro to the statements it represents, and then it assembles these new statements.
- No macros in this sample program

Typical Statement Format

- name mnemonic operand(s) ; comment
  - In the data segment, a name field has no punctuation
  - In the code segment, a name field is followed by a colon (:)
- Some statements omit some of these fields.

Identifiers

- **Identifiers used in assembly language are formed from letters, digits, and special characters**
  - **Special characters are best avoided except for an occasional underscore**
- **An identifier may not begin with a digit.**
- **An identifier may have up to 247 characters.**
- **Restricted identifiers include instruction mnemonics, directive mnemonics, register designations, and other words that have a special meaning to the assembler.**

Program Format

- Indent for readability, starting names in column 1 and aligning mnemonics and trailing comments where possible
- Assembler code is not case-sensitive, but it is good practice to
  - Use lowercase letters for instructions
  - Use uppercase letters for directives

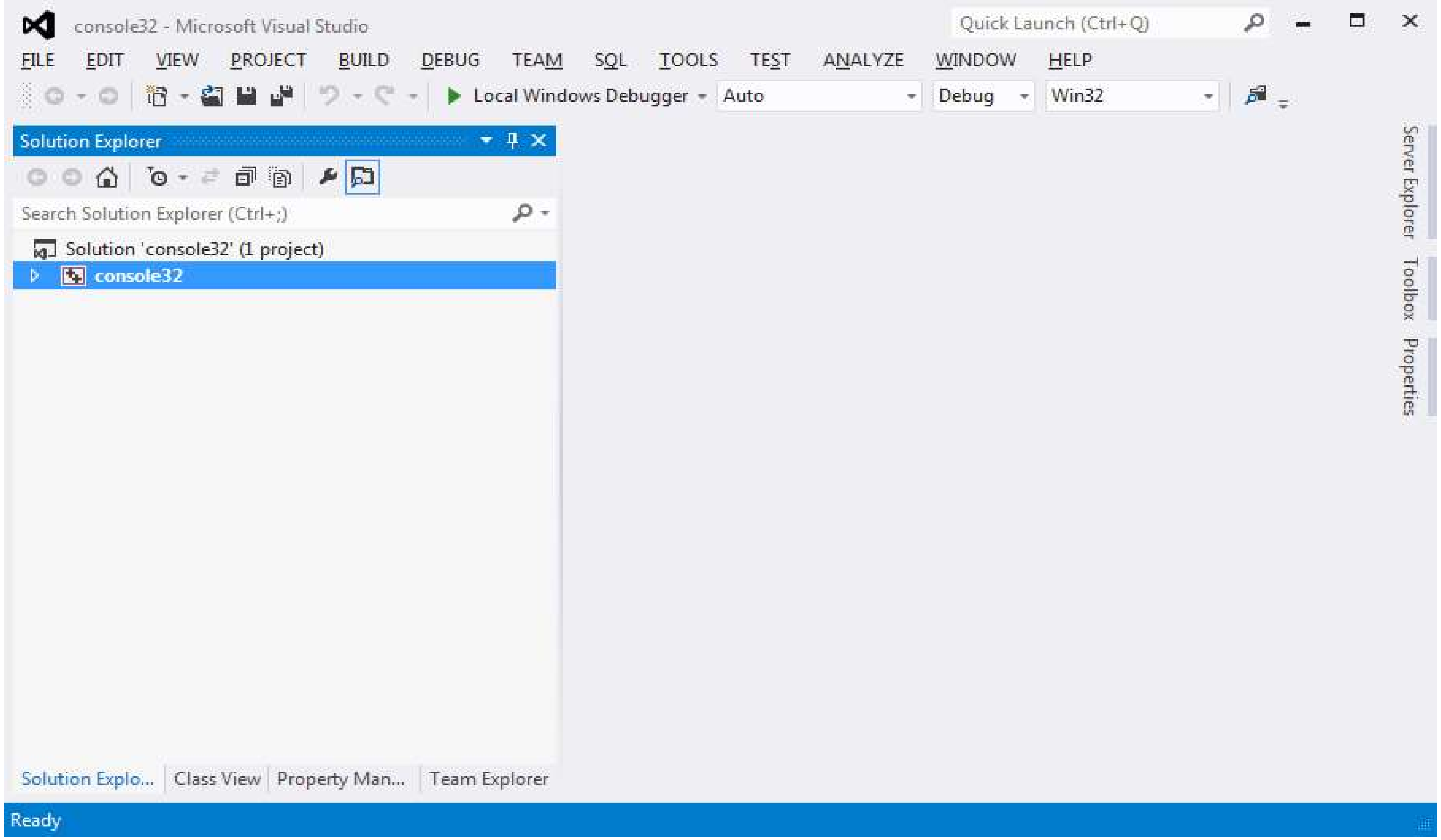
A Complete 32-bit Example Using the Debugger

Framework Packages

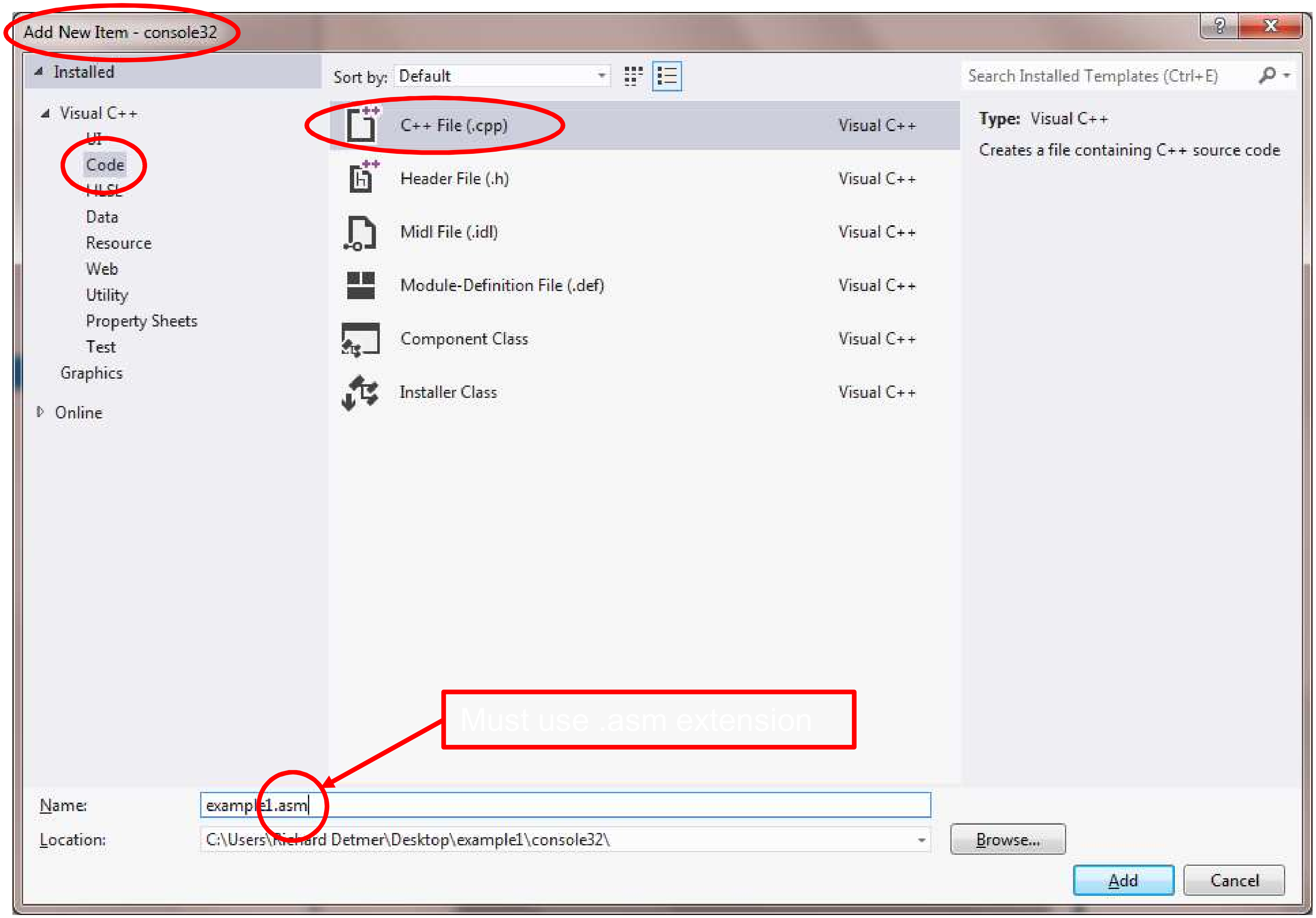
Name	**Use**
Name	Use
console32	• Produces 32-bit programs, even in a 64-bit operating system • No I/O provided • Debugger used to see register and memory contents
console64	• Produces 64-bit programs—64-bit operating system required • No I/O provided • Debugger used to see register and memory contents
windows32	• Produces 32-bit programs, even in a 64-bit operating system • Simple I/O using macros defined in the package • Debugger available to see register and memory contents
windows64	• Produces 64-bit programs—64-bit operating system required • Simple I/O using macros defined in the package • Debugger available to see register and memory contents

Using Visual Studio

- Open the *console32* project to see



Using Visual Studio


Add New Item - console32
?
X
Installed
Visual C++
UI
Code
HLSL
Data
Resource
Web
Utility
Property Sheets
Test
Graphics
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
C++ File (.cpp) Visual C++
Header File (.h) Visual C++
Midl File (.idl) Visual C++
Module-Definition File (.def) Visual C++
Component Class Visual C++
Installer Class Visual C++
Browse...
Add
Cancel

Type: Visual C++

Creates a file containing C++ source code

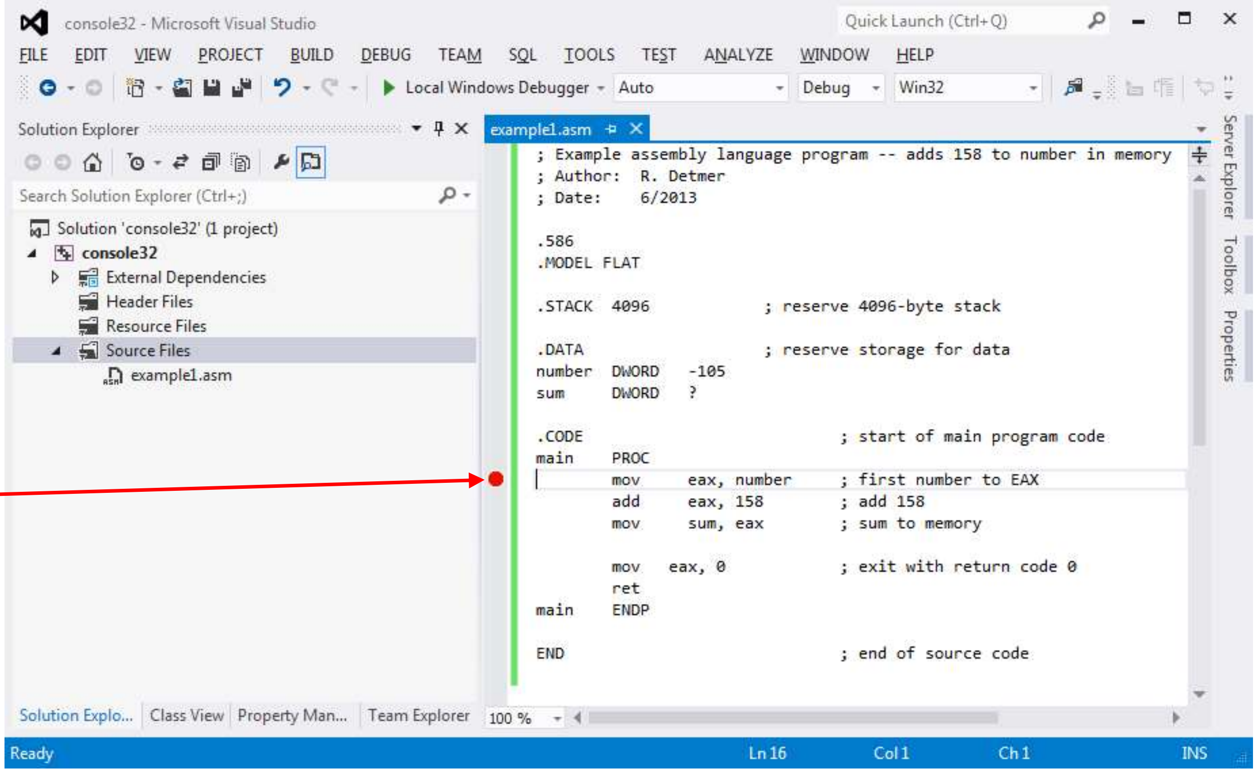
Must use .asm extension

Name: example1.asm

Location: C:\Users\Richard Detmer\Desktop\example1\console32\

Using Visual Studio

- Type or copy/paste source code
- Breakpoint at first instruction



```
; Example assembly language program -- adds 158 to number in memory
; Author:  R. Detmer
; Date:    6/2013

.586
.MODEL FLAT

.STACK  4096            ; reserve 4096-byte stack

.DATA                   ; reserve storage for data
number  DWORD   -105
sum     DWORD   ?

.CODE                           ; start of main program code
main    PROC
        mov     eax, number     ; first number to EAX
        add     eax, 158        ; add 158
        mov     sum, eax        ; sum to memory

        mov     eax, 0          ; exit with return code 0
        ret
main    ENDP

END                             ; end of source code
```

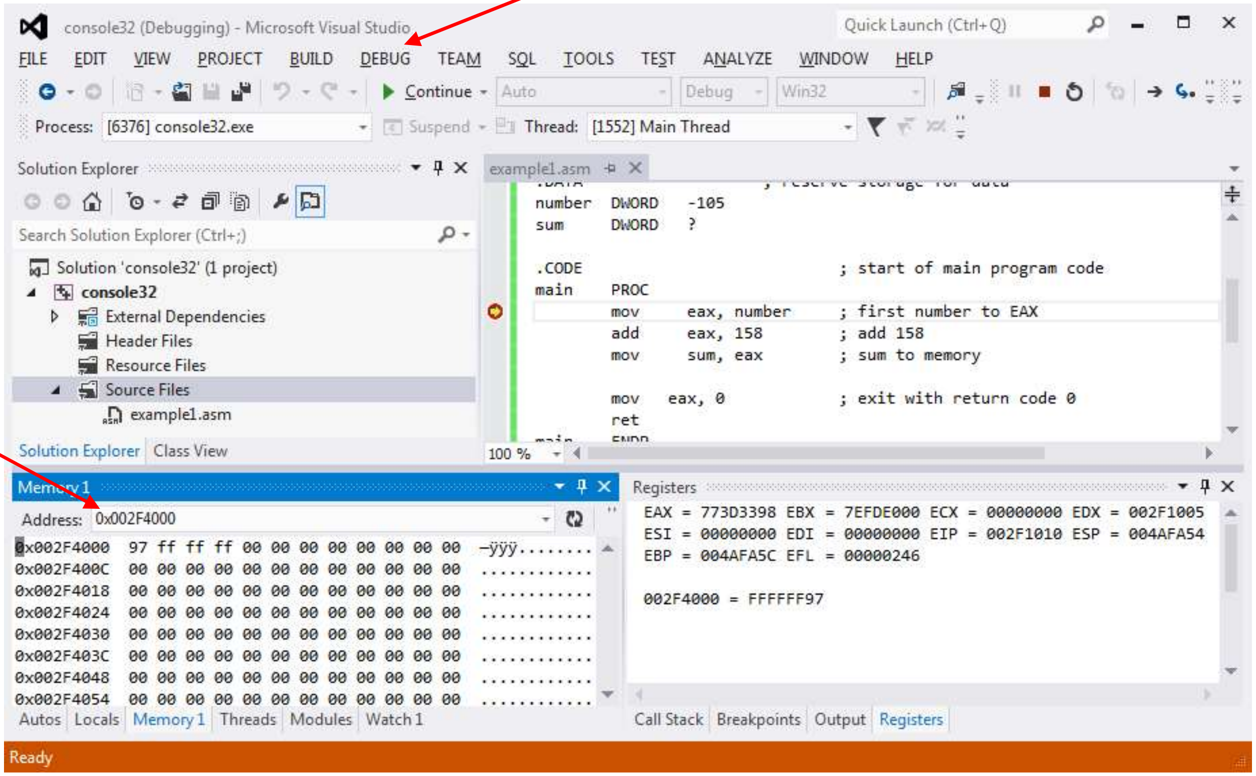
Click here to add breakpoint

Using Visual Studio

- **Launch execution with F5**

Use Debug/Window
to open debug windows

Enter address &number to see memory starting at *number*



console32 (Debugging) - Microsoft Visual Studio
Quick Launch (Ctrl+Q)
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP
Continue
Auto
Debug
Win32
Suspend
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'console32' (1 project)
console32
External Dependencies
Header Files
Resource Files
Source Files
example1.asm
Solution Explorer
Class View
100 %
Memory 1
Registers
Autos Locals Memory 1 Threads Modules Watch 1
Call Stack Breakpoints Output Registers
Ready

Process: [6376] console32.exe

Thread: [1552] Main Thread

example1.asm

```
.DATA                          ; reserve storage for data
number  DWORD   -105
sum     DWORD   ?

.CODE                          ; start of main program code
main    PROC
        mov     eax, number    ; first number to EAX
        add     eax, 158       ; add 158
        mov     sum, eax       ; sum to memory

        mov   eax, 0           ; exit with return code 0
        ret
main    ENDP
```

Address: 0x002F4000

```
0x002F4000  97 ff ff ff 00 00 00 00 00 00 00 00  —ÿÿÿ........
0x002F400C  00 00 00 00 00 00 00 00 00 00 00 00  ............
0x002F4018  00 00 00 00 00 00 00 00 00 00 00 00  ............
0x002F4024  00 00 00 00 00 00 00 00 00 00 00 00  ............
0x002F4030  00 00 00 00 00 00 00 00 00 00 00 00  ............
0x002F403C  00 00 00 00 00 00 00 00 00 00 00 00  ............
0x002F4048  00 00 00 00 00 00 00 00 00 00 00 00  ............
0x002F4054  00 00 00 00 00 00 00 00 00 00 00 00  ............
```

```
EAX = 773D3398 EBX = 7EFDE000 ECX = 00000000 EDX = 002F1005
ESI = 00000000 EDI = 00000000 EIP = 002F1010 ESP = 004AFA54
EBP = 004AFA5C EFL = 00000246

002F4000 = FFFFFF97
```

Using Visual Studio

- Step through program by pressing F10.
- Each time an instruction is executed, register or memory contents may change.
  - Changed values turn red
- The instruction pointer EIP will change each time to the address of the instruction to be executed.
- The flags register EFL (EFLAGS) will change if an instruction affects flags.

Debugger Memory Display

- Shows the starting memory address for each line
- Shows two hex digits for each byte memory byte
  - If the byte can be interpreted as a printable ASCII character, that character is displayed to the right.
  - Otherwise, a period is displayed to the right.

Output of Assembler

- Object file (e.g., example.obj)
  - Contains machine language statements almost ready to execute
- Listing file (e.g., example.lst)
  - Shows how the assembler translated the source program

Listing File

locations of data relative to start of data segment

8 bytes reserved for data, with first doubleword initialized to -105

```
00000000                    .DATA
00000000 FFFFFF97               number   DWORD    -105
00000004 00000000               sum      DWORD    ?
00000000                    .CODE
00000000                    main    PROC
00000000  A1 00000000 R                  mov      eax, number
00000005  05 0000009E                    add      eax, 158
0000000A  A3 00000004 R                  mov      sum, eax
```

object code for the three instructions

locations of instructions relative to start of code segment

Parts of an Instruction

- Instruction's object code begins with the opcode, usually one byte
  - Example, A1 for `mov eax, number`
- Immediate operands are constants embedded in the object code
  - Example, 0000009E for `add eax, 158`
- Addresses are assembly-time; must be fixed when program is linked and loaded
  - Example, 00000004 for `mov sum, eax`

Data Declarations

BYTE Directive

- Reserves storage for one or more bytes of data, optionally initializing storage
- Numeric data can be thought of as signed or unsigned.
- Characters are assembled to ASCII codes.
- Examples

```
byte1   BYTE  255        ; value is FF
byte2   BYTE  91         ; value is 5B
byte3   BYTE  0          ; value is 00
byte4   BYTE  -1         ; value is FF
byte5   BYTE  6 DUP (?)  ; 6 bytes each with 00
byte6   BYTE  'm'        ; value is 6D
byte7   BYTE  "Joe"      ; 3 bytes with 4A 6F 65
```

DWORD Directive

- **Reserves storage for one or more doublewords of data, optionally initializing storage**
- **Examples**

```
double1   DWORD   -1          ; value is FFFFFFFF
double2   DWORD   -1000       ; value is FFFFFC18
double3   DWORD   -2147483648 ; value is 80000000
double4   DWORD   0, 1        ; two doublewords
Double5   DWORD   100 DUP (?) ; 100 doublewords
```

WORD Directive

- Reserves storage for one or more words of data, optionally initializing storage

Multiple Operands

- Separated by commas
  - `DWORD 10, 20, 30 ; three doublewords`
- Using DUP
  - `DWORD 100 DUP (?) ; 100 doublewords`
- Character strings (BYTE directive only)
  - `BYTE "ABCD" ; 4 bytes`

Instruction Operands

Types of Instruction Operands

- Immediate mode
  - Constant assembled into the instruction
- Register mode
  - A code for a register is assembled into the instruction
- Memory references
  - Several different modes

Memory References

- Direct – at a memory location whose address is built into the instruction
  - Usually recognized by a data segment label, e.g., `mov sum, eax` (here eax is a register operand)
- Register indirect – at a memory location whose address is in a register
  - Usually recognized by a register name in brackets, e.g., `mov DWORD PTR [ebx], 10` (here 10 is an immediate operand)

A Complete 32-bit Example Using Windows Input/Output

windows32 Framework

- Program includes *io.h,* which defines input/output macros
- Main procedure must be called _MainProc
- Example prompts for and inputs two numbers, adds them, and displays sum

Example Program Data Segment

```
.DATA
number1 DWORD   ?
number2 DWORD   ?
prompt1 BYTE    "Enter first number", 0
prompt2 BYTE    "Enter second number", 0
string  BYTE    40 DUP (?)
resultLbl BYTE  "The sum is", 0
sum     BYTE    11 DUP (?), 0
```

Program Code Segment (1)

```
.CODE
_MainProc PROC
    input   prompt1, string, 40
```

Displays dialog box

Reads up to 40 characters into memory at *string*

```
    atod    string
```

Scans memory at *string*

Converts to doubleword integer in EAX

```
    mov     number1, eax
```

Program Code Segment (2)

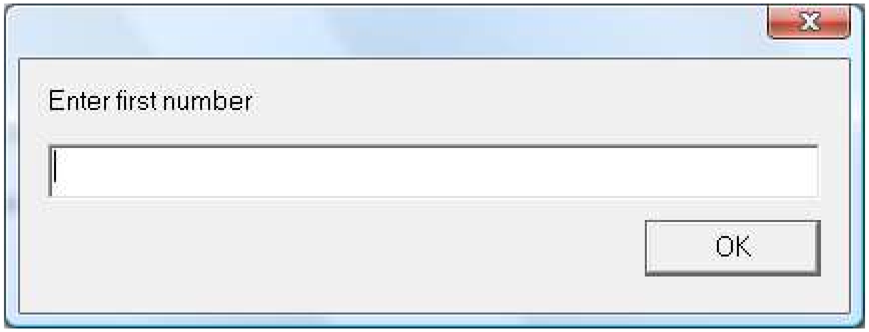
```
mov     eax, number1
add     eax, number2

dtoa    sum, eax
```

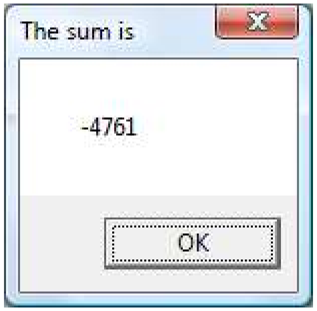
Convert doubleword integer in EAX to 11-byte-long string of spaces and decimal digits at *sum*

```
output  resultLbl, sum
```

Display message box showing two strings

Input and Output


Enter first number


The sum is

-4761

Input/Output and Data Conversion Macros Defined in IO.H

atod

- Format: atod *source*
- Scans the string starting at *source* for
  + or - followed by digits, interpreting
  these characters as an integer. The corresponding
  2's complement number
  is put in EAX.

dtoa

- **Format: dtoa *destination*, *source***
- **Converts the doubleword integer at *source* (register or memory) to an eleven-byte-long ASCII string at *destination*. The string represents the decimal value of the source number and is padded with leading spaces.**

input

- **Format: input *prompt, destination*, *length***
- **Generates a dialog box with label specified by *prompt,* where *prompt* references a string in the data segment. When OK is pressed, up to *length* characters are copied from the dialog box to memory at *destination.***

output

- Format: output *labelMsg, valueMsg*
- Generates a message box with the label *labelMsg*, and *valueMsg* in the message area. Each of *labelMsg* and *valueMsg* references a string in the data segment.

atow and wtoa

- Similar to atod and dtoa, but for words instead of doublewords
- Rarely needed because doublewords are the integer size of choice in current 80x86 systems.

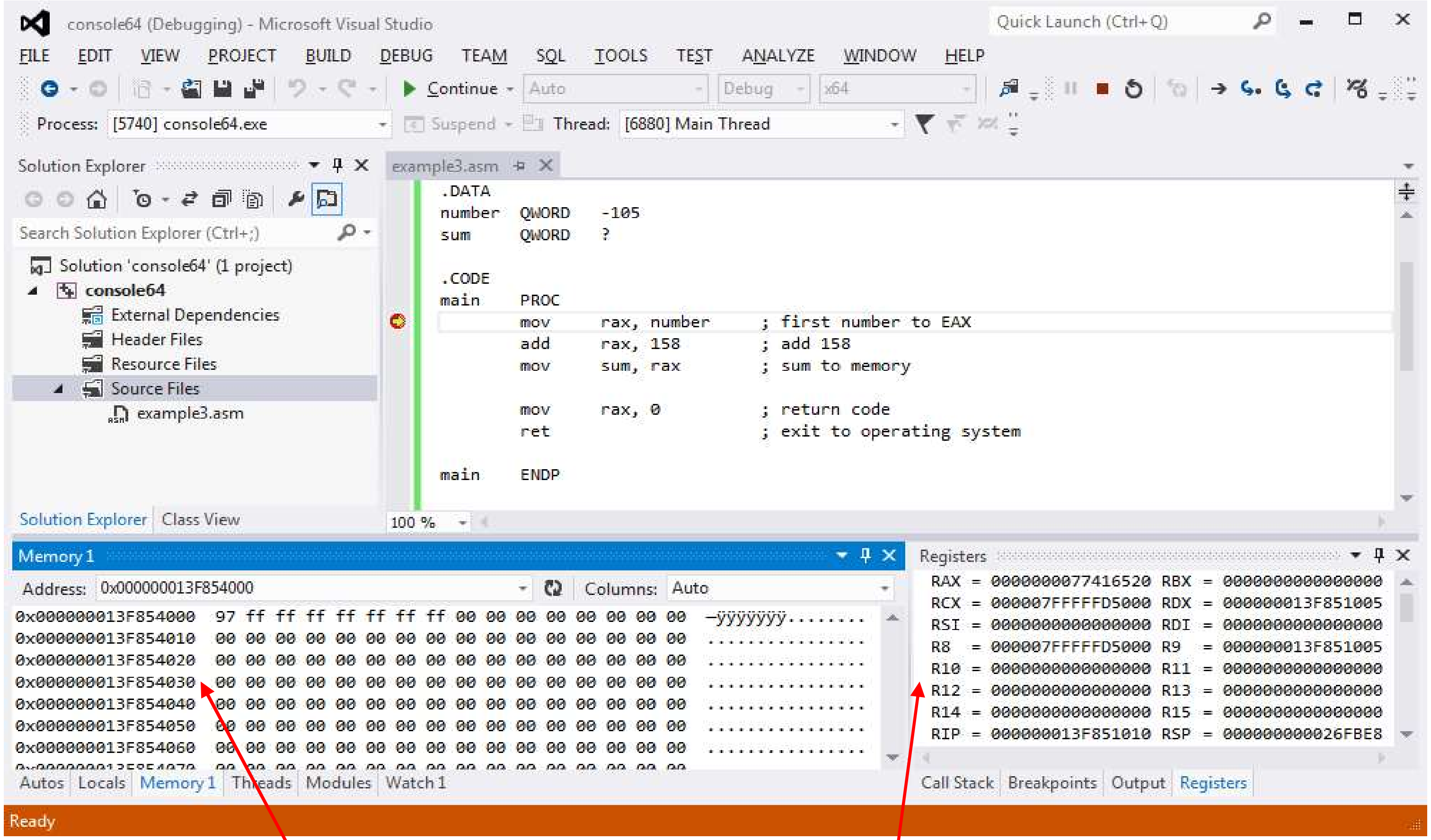
64-bit Examples

console64 Example

- Similar to *console32*, but fewer directives

```
; Example assembly language program
.DATA
number  QWORD   -105
sum     QWORD   ?
.CODE
main    PROC
        mov     rax, number
        add     rax, 158
        mov     sum, rax
        mov     rax, 0
        ret
main    ENDP
END
```

Debugger



```
.DATA
number  QWORD   -105
sum     QWORD   ?

.CODE
main    PROC
        mov     rax, number      ; first number to EAX
        add     rax, 158         ; add 158
        mov     sum, rax         ; sum to memory

        mov     rax, 0           ; return code
        ret                      ; exit to operating system

main    ENDP
```

```
0x000000013F854000  97 ff ff ff ff ff ff ff 00 00 00 00 00 00 00 00  —ÿÿÿÿÿÿÿ........
0x000000013F854010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x000000013F854020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x000000013F854030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x000000013F854040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x000000013F854050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x000000013F854060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x000000013F854070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

```
RAX = 0000000077416520 RBX = 0000000000000000
RCX = 000007FFFFFD5000 RDX = 000000013F851005
RSI = 0000000000000000 RDI = 0000000000000000
R8  = 000007FFFFFD5000 R9  = 000000013F851005
R10 = 0000000000000000 R11 = 0000000000000000
R12 = 0000000000000000 R13 = 0000000000000000
R14 = 0000000000000000 R15 = 0000000000000000
RIP = 000000013F851010 RSP = 000000000026FBE8
```

64-bit addresses

64-bit registers

64-bit Differences

- “Direct” memory addressing is actually RIP relative: the 32-bit offset stored in the instruction is added to RIP to get the operand address.
- Extra code is required in *windows64* programs

```
sub rsp,120  ; reserve stack space for MainProc
        ...
add rsp, 120 ; restore stack
```