Overview of IA-32 assembly programming

Lars Ailo Bongo

University of Tromsø

Contents

1 Introduction ... 2
2 IA-32 assembly programming ... 3
2.1 Assembly Language Statements ... 3
2.1 Modes ... 4
2.2 Registers ... 4
2.2.3 Data Registers ... 4
2.2.4 Pointer and Index Registers ... 4
2.2.5 Control Registers ... 5
2.2.6 Segment registers ... 7
2.3 Addressing ... 7
2.3.1 Bit and Byte Order ... 7
2.3.2 Data Types ... 7
2.3.3 Register Addressing Mode ... 7
2.3.4 Immediate Addressing Mode ... 8
2.3.5 Memory Addressing Modes ... 8
2.4 Stack ... 10
2.5 C procedure call convention ... 10
3 MASM Assembler directives ... 12
3.1 Data allocation ... 12
3.2 Defining Constants ... 13
3.2.1 The EQU directive ... 13
3.2.2 The = directive ... 13
3.3 Multiple Source Program Modules ... 13
3.3.1 The PUBLIC Directive ... 13
3.3.2 The EXTRN directive ... 14
4 Mixed language programming ... 15
4.1 Inline assembly ... 15
4.2 Using the Visual C++ debugger to test simple assembly programs ... 15
4.3 Using assembly files in Visual C++ ... 17
4.4 Understanding Assembly Listings ... 19
5 Examples ... 25
5.1 Arithmetic Instructions ... 25
5.2 Data Transfer (mov instruction) ... 25
5.2 Jumps ... 26
5.2.1 Unconditional Jump ... 26
5.2.2 Conditional Jumps ... 26
5.3 Function calls ... 27
5.4 The most useful IA-32 Instructions ... 28
6 References ... 31

1 Introduction

I think the best introduction to assembly programming is [Patterson, Hennesy] chapters 3 and 4. I assume you have read those chapters and know how to program in MIPS assembly, and are looking for a short guide on how to program in IA-32 assembly. Unfortunately all the books and tutorials I have read are:

1. Too long (as a student I didn't have time to read 1000 page books or tutorials).
2. Assuming that the reader is programming in MS-DOS.
3. Assuming that the reader needs to know about unimportant topics as BCD arithmetic.

Therefore I wrote this short paper that hopefully teaches you all the basic things you need to know to start programming in IA-32 assembly. The topics covered are:

- Most important aspects of the IA-32 architecture (registers, addressing modes, stack).
- MASM assembler directives (i.e. how to use MASM to write IA-32 assembly programs).
- How to use assembly code in your Visual C++ programs.
- How to read assembly listings produced by the Microsoft C compiler.

2 IA-32 assembly programming

This chapter is intended to be a reference you can use when programming in IA-32 assembly. It covers the most important aspects of the IA-32 architecture.

2.1 Assembly Language Statements

All assembly instructions, assembler directives and macros use the following format:

[label]   mnemonic   [operands]   [; comment]

Fields in square brackets are optional.

Label: used to represent either an identifier or a constant.

Mnemonic: Identifies the purpose of the statement. A Mnemonic is not required if a line contains only a label or a comment.

Operands: Specifies the data to be manipulated.

Comment: Text ignored by the assembler.

Example

```
    ; This is a comment
    jmp     label1  ; This is also a comment
    add     eax, ebx
label1:
    sub     edx, 32
```

Labels are in *italic*, mnemonics in **bold**, operands are underlined, and comments are in regular text.

Most instructions take two operands. Usually one of the operands is in a register, and the other can be in a register, memory or be an immediate value. In many instructions the first operand is used as source and destination.

Example:

```
add eax, ebx   ; EAX = EAX + EBX
```

2.1 Modes

Normally we only run in protected mode. But the Pentium processor can also run in real mode (for backward compatibility), system management mode (power management) and virtual 8086 mode (for backward compatibility).

2.2 Registers

This chapter is a summary of chapters 2, 3 and 5 from [Dandamudi]. Most of the figures and examples are taken from this book. If you want a more detailed explanation (or a better written one) you should buy and read this book.

2.2.3 Data Registers

The IA-32 processors provides four 32-bits data registers, they can be used as:

- Four 32-bits registers (EAX, EBX, ECX, EDX)
- Four 16-bits registers (AX, BX, CX, DX)
- Eight 8-bits registers (AL, AH, BL, BH, CL, CH, DL, DH)

32-bits registers (31...0)	Bits 31...16	Bits 15...8	Bits 7...0
EAX		AH	AL
EBX		BH	BL
ECX		CH	CL
EDX		DH	DL

The data registers can be used in most arithmetic and logical instructions. But when executing some instructions, some registers have special purposes.

2.2.4 Pointer and Index Registers

The IA-32 processors have four 32-bits index and pointer registers (ESI, EDI, ESP and EBP). These registers can also be used as four 16-bits registers (SI, DI, SP and EP).

Usually ESI and EDI are used as regular data registers. But when using the string instructions they have special functions.

ESP is the stack pointer, and EBP is the frame pointer. If you don't use stack frames, you can use EBP as a regular data register.

32-bits registers (31...0)	Bits 31...16	Bits 15...0	Special function
ESI		SI	Source index
EDI		DI	Destination index
ESP		SP	Stack pointer
EBP		BP	Frame pointer

2.2.5 Control Registers

The two most important control registers are the instruction pointer (EIP) and the EFlags register.

The Pentium has also many other control registers, which are not covered in this document (they control the operation of the processor, and applications cannot change them).

The Instruction Pointer Register (EIP)

EIP points to the next instruction to be executed. EIP cannot be accessed directly.

The EFlags register

Six of the flags in the EFlags register are status or arithmetic flags. They are used to record information about the most recently executed arithmetic or logical instruction. Three of the flags: SF, PF and AF are rarely used.

- Zero Flag (ZF). This flag is set when the result of the last executed arithmetic instruction was zero. ZF is used to test for equality or count down to a preset value. Related instructions are: *jz* and *jnz*.
- Carry Flag (CF). CF is set if the last arithmetic operation (on two unsigned integers) was either too big or too small (out of range). CF is used to propagate carry or borrow, detect overflow/ underflow or test a bit (using shift/ rotate). Related instructions are: *jc, jnc, stc, clc*, and *cmc*. Note that *inc* and *dec* does not affect the carry flag.
- Overflow Flag (OF). OF indicates when an operation on signed integers resulted in an overflow/underflow. Related instructions are: *jo* and *jno*.
- Sign Flag (SF). Indicates the sign of the result of an arithmetic operation. Related instructions are: *js* and *jns*.
- Parity Flag (PF). Indicates the parity of the 8-bit result produced by an operation. PF = 1 if the byte contains an even number 1 bits. It is used in data encoding programs. Related instructions are *jp* and *jnp*.

- Auxiliary Flag (AF). Indicates whether an operation has produced a result that has generated a carry, or borrow into the low-order four bits of 8- 16- or 32-bit operands. AF is used in arithmetic operations on BCD numbers.

One of the flags is a control flag:

- Direction flag (DF). It determines wetter string operations should scan the string forward or backward. It is only used in string instructions. DF can be set by *std* and cleared by *cld.*

The remaining ten flags are system flags. They are used to control the operation of the processor. Ordinary application programs cannot set these flags directly.

- TF (trap flag)
- IF (interrupt flag)
- IOPL (I/O privilege level)
- NT (nested task)
- RF (resume flag)
- VM (virtual 8086 mode)
- AC (alignment check)
- VIF (virtual interrupt flag)
- VIP (virtual interrupt pending)
- ID (ID flag)

Examples

```
mov     EAX, 8       ; ZF = 0
sub     EAX, 8       ; ZF = 1

cmp     char, 0      ; ZF = 1 if char == '\0'
cmp     EAX, EBX     ; ZF = 1 if EAX = EBX

; for (i = 0; i < 12; i++)
    mov     ECX, 12 ; ECX 12
loop:
    <do something>
    dec     ECX      ; ECX = ECX - 1
    jnz     loop     ; Jump if ZF = 0

mov     AL, 100
add     AL, 200      ; CF = 1

mov     AX, 100
sub     AX, 101      ; CF = 1 (any negative integer is out of range)

mov     AL, 100
add     AL, 30       ; OF = 1 (signed char range is -128...127)
```

Note that the processor does not know if you are using signed or unsigned integers. OF and CF are set for every arithmetic operation.

```
mov     AL, 15
add     AL, 100      ; SF = 0 (positive result)

mov     AL, 15
sub     AL, 100      ; SF = 1 (negative result)
```

2.2.6 Segment registers

The Pentium processor has six 16-bits segment registers:

- CS (code segment)
- DS (data segment)
- SS (stack segment)
- ES (extra data segment)
- FS (extra data segment)
- GS (extra data segment)

Modern applications and operating systems (including Windows 2000 and Linux) use the flat memory model (unsegmented memory model). In this model all segment registers are loaded with the same segment selector. So all memory references are to a single linear-address space.

2.3 Addressing

Most of the figures and examples are taken from [Dandamudi] chapter 5.

2.3.1 Bit and Byte Order

The Pentium processors uses little-endian byte order

2.3.2 Data Types

Data Type	Size
Byte	8 bits
Word	16 bits
Doubleword	32 bits
Quadword	64 bits

2.3.3 Register Addressing Mode

The operand is in a register.

```
mov     EAX, EBX ; move EBX to EAX
```

2.3.4 Immediate Addressing Mode

The operand is part of the instruction.

```
mov    EAX, 132 ; move 132 to EAX
```

2.3.5 Memory Addressing Modes

Direct addressing mode

The operand is in memory, and the address is specified as an offset.

```
a_letter DB 'c'     ; Allocate one byte of memory, initialize it to 'c'.
mov AL, a_letter    ; Move data at memory location "a_letter" into AL.
                    ; I.e. move 'c' to AL.
```

Register Indirect Addressing

The operand is found at the memory location specified by the register. The register is enclosed in square bracket.

```
mov EAX, ESP    ; Move stack pointer to EAX
mov EBX, [ESP]  ; Move value at top-of-stack to EBX
```

The first move uses register addressing, and the second uses register indirect addressing.

Indirect Addressing Mode

The offset of the data is in one of the eight general-purpose registers.

```
.DATA
  array DD 20 DUP (0)   ; Array of 20 integers initialized to zero
.CODE
  mov ECX, OFFSET array ; Move starting address of 'array' to ECX
```

The assembler directive OFFSET is used when we want to use the address of an element, and not the contents of the element.

Note that:

```
mov ECX, array
```

moves the first element in array (array[0]) into ECX, and not the address of the first element (&(array[0])).

Based Addressing

One of the eight general-purpose registers acts like a base register in computing the effective address of an operand. The address is computed by adding a signed (8-bit or 32-bit) number to the base address.

```
mov ECX, 20[EBP] ; ECX = memory[EBP + 20]
```

Indexed Addressing

The effective address is computed by:

(Index * scale factor) + signed displacement.

The beginning of the array is given by a displacement, and the value of the index register (EAX, EBX, ECX, EDX, ESI, EDI, EBP) selects an element within the array. The scale factor is used to specify how large the elements in the array are (in bytes). The scale factor can only be 1, 2, 4 or 8.

```
add AX, [DI + 20]   ; AX = AX + memory[DI + 20]
mov AX,table[ESI*4] ; AX = memory[ OFFSET table + ESI * 4 ]
add AX, table[SI]   ; AX = AX + memory[ OFFSET table + ESI * 1]
```

Based-Indexed Addressing

In this addressing mode, the effective address is computed as:

Base + (Index * Scale factor) + signed displacement.

The beginning of the array is given by a base register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP) and a displacement, and the value of the index register (EAX, EBX, ECX, EDX, ESI, EDI, EBP) selects an element within the array. The scale factor is used to specify how large the elements in the array are (in bytes). The scale factor can only be 1, 2, 4 or 8. The signed displacement must be either an 8, 16 or 32-bit value.

```
mov EAX, [EBX+ESI]     ; AX = memory[EBX + (ESI * 1) + 0]
mov EAX, [EBX+EPI*4+2] ; AX = memory[EBX + (EPP * 4) + 2]
```

The PTR directive

Sometimes the assembler does not know how large values it is supposed to use, as shown in the following example:

```
array SQWORD 20 DUP (0) ; int array[20];
mov ECX, OFFSET array   ; ECX = &(array[0])
mov [ECX], 25           ; memory[ECX] = 25, but is '25' a 1-byte,
                        ; 2-byte or 4-byte value?
```

To clarify we use the PTR directive (syntax: type-specifier PTR)

```
mov ECX, OFFSET array       ; ECX = &(array[0])
mov [ECX], SQWORD PTR 25    ; memory[ECX] = 25, and '25'  is a 4-byte
                            ; value (signed quad word)
```

You should use the PTR directive when the operand size is not implicit with the register name.

2.4 Stack

Properties:

- Pointed to by SS:ESP
- Only 32-bit data are pushed on the stack. (push al, uses 32-bits on the stack)
- The stack grows downward.
- ESP points to the last word saved on the stack.

Stack operation

push source:

1. ESP = ESP - 4
2. memory[ESP] = source

pop destination

1. destination = memory[ESP]
2. ESP = ESP + 4

Other stack operations are: pushfl (push EFlags), popfl (pop EFlags), pusha (push all general-purpose registers), popa (pop all general-purpose registers)

2.5 C procedure call convention

The convention below is used by MASM, I don't know if gas ("Linux" assembler) uses the same convention.

When doing a function call, the caller must:

- Save EAX, EBX, ECX and EDX if they must be preserved.
- Push all arguments on the stack. The arguments are pushed from right to left.
- Invoke the function, by using the instruction *call* (*call* will push the return address and jump to the called function)

Before the called function starts running it must:

- Save EBP, ESI, EDI, DS and SS if they are clobbered.
- Create a stack frame (if stack frames are used). This is done by setting:
    1. EBP = ESP
    2. ESP = ESP - frame size
    - The stack frame must contain space for local variables.
- Save the direction flag (EFlags.DF), if it is altered.

Before the called function returns it must:

- Restore all saved registers and the direction flag (if it was saved)
- Pop the stack frame by setting ESP = EBP
- A return value is stored according to the table below.
- Return to the caller by using the ret instruction (ret pops the return address, and jumps to it)

After returning from a function call, the caller must

- Pop all arguments. (Normally ESP is set to ESP + sizeof(arguments))
- Restore all saved registers.

Return Value Data Type	**Is Saved in Register**
char	AL
short (16-bit)	AX
int (32-bit)	EAX
64-bit	ECX:EAX

3 MASM Assembler directives

This chapter lists and explains the most important MASM directives.

The figures are from [MASM] and [Dandamudi]. Most of the examples are also taken from this book (this chapter is really a summary of chapter 3 from [Dandamudi]).

3.1 Data allocation

The general format of a storage allocator is:

[variable-name] define-directive initial-value [,initial-value],...

Variable-name: identify the storage space allocated.

Define-directive: the following table shows the directives that can be used, and the size in bytes:

The following directives indicate the size and value range of some integers and floating point numbers:

Directive	**Description of Initializers**
BYTE, **DB** (byte)	Allocates unsigned numbers from 0 to 255.
SBYTE (signed byte)	Allocates signed numbers from –128 to +127.
WORD, DW (word = 2 bytes)	Allocates unsigned numbers from 0 to 65,535 (64K).
SWORD (signed word)	Allocates signed numbers from –32,768 to +32,767.
DWORD, DD (doubleword = 4 bytes),	Allocates unsigned numbers from 0 to 4,294,967,295 (4 megabytes).
SDWORD (signed doubleword)	Allocates signed numbers from –2,147,483,648 to +2,147,483,647.
FWORD, DF (farword = 6 bytes)	Allocates 6-byte (48-bit) integers. These values are normally used only as pointer variables on the 80386/486 processors.
QWORD, DQ (quadword = 8 bytes)	Allocates 8-byte integers used with 8087-family coprocessor instructions.
TBYTE, DT (10 bytes),	Allocates 10-byte (80-bit) integers if the initializer has a radix specifying the base of the number.
REAL4	Short (32-bit) real numbers
REAL8	Long (64-bit) real numbers
REAL10	10-byte (80-bit) real numbers and BCD numbers

Examples

```
letter_c DB 'c'          ; Allocate a single byte of memory, and
                         ; initialize it to the letter 'c'.
an_integer DD 12425      ; Allocate memory for an integer (4-bytes), and
                         ; initialize it to 12425.
a_float REAL4 2.32       ; Allocate memory for a float, and initialize
                         ; it to 2.32
message DB 'Hello',13,0  ; Allocate memory for a null terminated string
                         ; "Hello\n"
marks DW 0, 0, 0, 0      ; Both allocates memory for an array of 4 * 2
                         ; bytes, and initialize all elements to zero.
marks DW 4 DUP (0)       ; DUP allows multiple initializations to the
                         ; same value
name DB 30 DUP(?)        ; Allocate memory for 30 bytes, uninitialized.
matrix QW 12*10          ; Allocate memory for a 12*10 quad-bytes matrix
```

We can also use the LABEL directive to name a memory location, the syntax is:

name LABEL type

3.2 Defining Constants

3.2.1 The EQU directive

Syntax: name EQU expression. It serves the same purpose as #define in C.

3.2.2 The = directive

Syntax: name = expression. The symbol that is defined by the = directive can be redefined, but it cannot be used to define strings.

3.3 Multiple Source Program Modules

3.3.1 The PUBLIC Directive

Syntax: PUBLIC label1, label2, label3...

This directive makes the labels public, and therefore available from other modules (source files).

Examples

```
PUBLIC error_msg, table
PUBLIC _a_C_function     ; All C functions begin with an underscore
```

3.3.2 The EXTRN directive

Syntax: EXTRN label:type

This directive can be used to declare extern labels (variables, functions, etc). The table below lists some types:

BYTE	Data variable (8-bits)
WORD	Data variable (16-bits)
DWORD	Data variable (32-bits)
QWORD	Data variable (64-bits)
PROC	A procedure name

Examples

```
EXTRN error_msg:BYTE, table:DW
EXTRN _printf:PROC              ; All C functions begin with an
                                ; underscore.
```

Normally source files are included when compiling, and object files (libraries) when linking.

4 Mixed language programming

This chapter covers three topics: how to write inline assembly in Visual C++, how you can use Visual C++ to debug your assembly programs, and how to read assembly listings (produced by the compiler).

4.1 Inline assembly

Inline assembly is used to insert assembly code into C source files.

In Visual C++ the keyword asm is placed before the inline assembly code, as shown in the examples.

Examples

```
asm pushf  ; Push the Eflags register

asm {
    mov  EAX, 0
    sub  EAX, 12
}
```

4.2 Using the Visual C++ debugger to test simple assembly programs

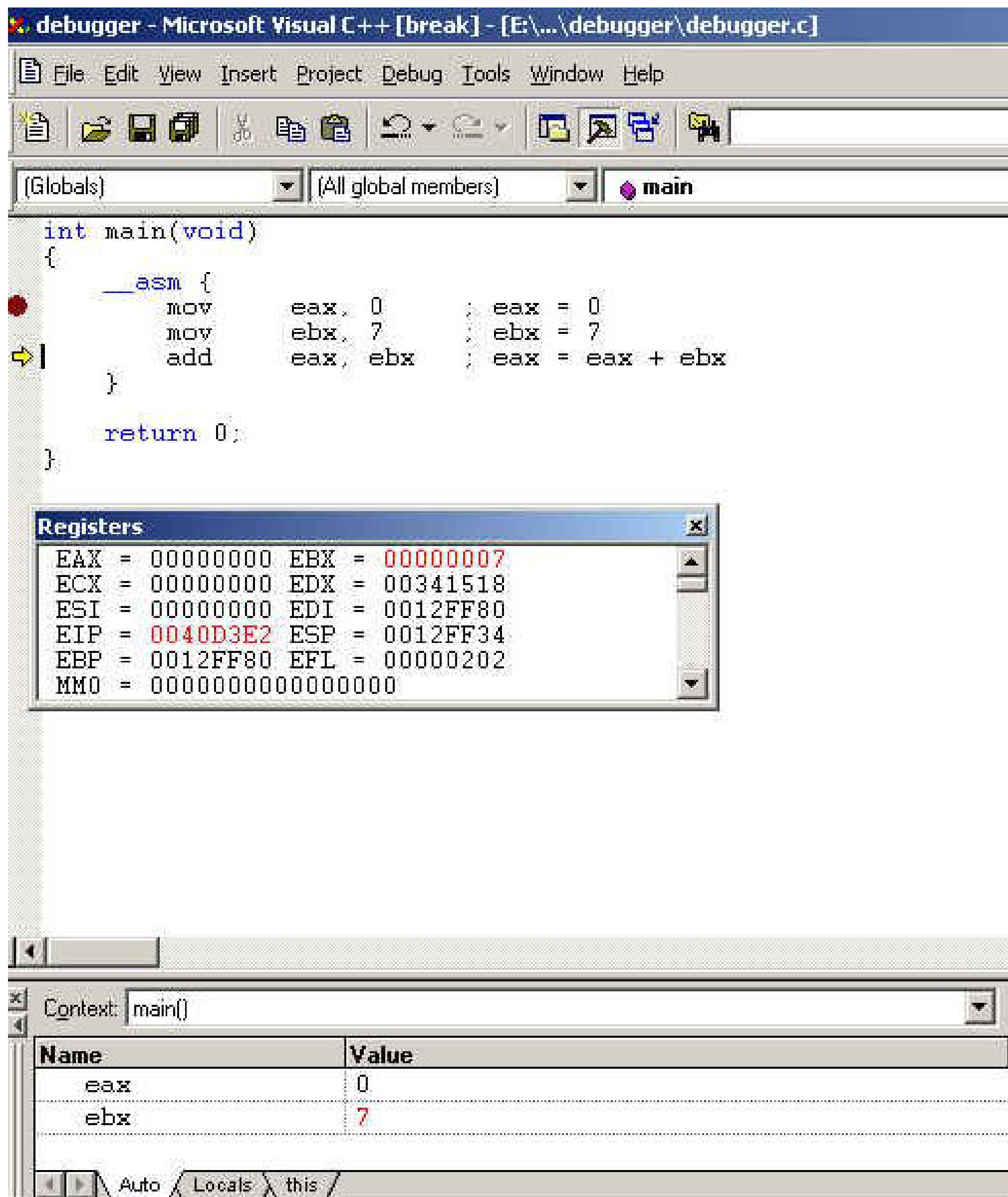
If you want to see what simple assembly programs do with the data registers and memory you can use the debugger in Visual C++.

To do this you need to:

1. Create a new (console) project in Visual C++.
2. Write your assembly code in inline assembly, as shown below.
3. Insert a breakpoint at the beginning of the assembly code (right click | Insert Breakpoint)
4. Start debugging (Build | Start Debug | Go (F5)).
5. View register window or/and memory window (View | Debug Windows | Registers or Memory).
6. Step trough the program (Debug | Step into (F11)). Then you can see what the registers and memory contain after each executed instruction.
7. When you are done, you stop the debugger (Debug | Stop Debugging)

We can see in the figure that:

- The breakpoint is set to the beginning of the assembly code (red bullet).
- The next instruction to be executed is add eax, ebx (yellow arrow)
- That the last instruction executed changed registers EBX, and EIP (red color in Registers window)



Note: By using this method your C programs will probably not function correctly, unless you save and restore all registers that are clobbered in the assembly code.

4.3 Using assembly files in Visual C++

This chapter tells you how to use assembly files in a Visual C++ project.

First we write a C source file, which calls the assembly function:

```
#include <stdio.h>

/* Return a + b
 * This function is in func.asm
 */
extern int assembly_function(int a, int b);

int main(void)
{
    printf("14 + 21 = %d\n", assembly_function(14, 21));

    return 0;
}
```

Then we write an assembly file that contains the function we are interested in:

```
.586                ; 32-bits (with Pentium instructions)
.MODEL flat         ; Flat memory model (no-segmentation)

EXTERN _printf:NEAR ; printf is an external function

; assembly_function is a public function
; Note that all C functions begins with an underscore
PUBLIC _assmebly_function

.DATA               ; Begin data segment

; printf() string (null terminated)
printf_msg DB 'Arguments: %d and %d\n', 0

.CODE               ; Begin code segment
; assembly_function in C:
; int assembly_function(int a, int b)
; {
;   int c = a + b;
;
;   printf("Arguments: %d and %d\n", a, b);
;
;   return c;
; }


; The = directive does the same as #define in C

; Location of arguments on the stack frame
_arg1 = 8           ; EBP + 8 = arg1 (a)
_arg2 = 12          ; EBP + 12 = arg2 (b)
; Location of local variables
```

```
_loc1 = -4          ; EBP - 4 = local variable (c)

_assembly_function:
  push ebp          ; Save old base pointer
  mov ebp, esp      ; Point EBP to top of stack
  sub esp, 4        ; Make space on stack for local variable

  mov eax, SDWORD PTR _arg1[ebp] ; Move argument 1 into eax
  ; Note that we specify that we are interested
  ; in moving 4 bytes (SDWORD PTR)
  mov ebx, SDWORD PTR _arg2[ebp] ; Move argument 2 into ebx
  mov ecx, eax ; ecx = eax
  add ecx, ebx ; ecx = eax + ebx

  ; Save caller saved registers
  ; Note that we don't need to save eax and ebx, because they don't
  ; need to be preserved
  push ecx
  ; Push printf arguments
  push ebx ; Push argument 3
  push eax ; Push argument 2
  push OFFSET printf_msg          ; Push address of string (argument 0)
  call _printf
  add esp, 12                     ; Pop arguments

  ; Restore caller saved registers
  pop ecx

  mov eax, ecx                    ; Store return value in eax

  mov esp, ebp                    ; ESP points to top of stack frame
  pop ebp                         ; Restore EBP register
  ret                             ; Return to caller

; End of source file
end
```

Then we need to assemble this file when building the project:

1. Insert file into project
2. Right-click on the filename in the FileView, and choose Settings.
3. Select the Custom Build tab.
4. In commands you write:
   c:\masm611\bin\ml /c /coff /Zd $(InputName).asm
5. And in Outputs you write:
   $(InputName).obj
6. Compile and run as usual.

4.4 Understanding Assembly Listings

One way to learn assembly programming is to study assembly listings produced by the compiler. In this chapter I have commented the assembly listing produced by the Microsoft compiler for the C program given in the next page.

C code:

```
#include <stdio.h>
#include "error_wrapper.h"

/* Just open the file given as the first command line
 * argument.
 */
int main(int argc, char *argv[])
{
  FILE *f;

  /* First argument is the name of the executable file */
  setprogname(argv[0]);

  /* Second argument is the file to be opened */
  if (argc < 2)
        eprintf("Usage: error_wrapper filename");

  f = fopen(argv[1], "r");
  if (f == NULL)
    eprintf("can't open file: %s", argv[1]);
  fclose(f);

  printf("File opened and closed without errors\n");

  return 0;
}
```

Assembly output (my comments begins with three semicolons):

```
;;; Name of the c file ?
        TITLE   H:\d241_a00\assembly_example\error_wrapper_test.c

;;; 386 processor mode (P: enable the instructions available only at
higher privilege levels)
        .386P

;;; This file contains assembler macros and is included by the files
;;; created with the -FA compiler switch to be assembled by MASM.
include listing.inc

;;; if MAMSM version > 5.1 then use flat memory model
;;; (no segmentation, code and data in the same segment)
;;; We use FLAT in Windows 2000
if @Version gt 510
.model FLAT
```

```
;;; Ignore this
else
_TEXT   SEGMENT PARA USE32 PUBLIC 'CODE'
_TEXT   ENDS
_DATA   SEGMENT DWORD USE32 PUBLIC 'DATA'
_DATA   ENDS
CONST   SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST   ENDS
_BSS    SEGMENT DWORD USE32 PUBLIC 'BSS'
_BSS    ENDS
$$SYMBOLS       SEGMENT BYTE USE32 'DEBSYM'
$$SYMBOLS       ENDS
$$TYPES SEGMENT BYTE USE32 'DEBTYP'
$$TYPES ENDS
_TLS    SEGMENT DWORD USE32 PUBLIC 'TLS'
_TLS    ENDS
;       COMDAT ??_C@_0BO@HHKP@Usage?3?5error_wrapper?5filename?$AA@
CONST   SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST   ENDS
;       COMDAT ??_C@_01LHO@r?$AA@
CONST   SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST   ENDS
;       COMDAT ??_C@_0BE@OCM@can?8t?5open?5file?3?5?$CFs?$AA@
CONST   SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST   ENDS
;       COMDAT ??_C@_0CH@BNAK@File?5opened?5and?5closed?5without?5e@
CONST   SEGMENT DWORD USE32 PUBLIC 'CONST'
CONST   ENDS
;       COMDAT _main
_TEXT   SEGMENT PARA USE32 PUBLIC 'CODE'
_TEXT   ENDS
FLAT    GROUP _DATA, CONST, _BSS
        ASSUME  CS: FLAT, DS: FLAT, SS: FLAT
endif

;;; Main is a public function (other modules can call it)
PUBLIC  _main

;;; These are static string labels.
;;; Static strings must be public so that other modules can use them.
;;; (the module which printf() is in must access string ABC when we
;;; use: printf("ABC");
PUBLIC  ??_C@_0BO@HHKP@Usage?3?5error_wrapper?5filename?$AA@ ; `string'
PUBLIC  ??_C@_01LHO@r?$AA@                                  ; `string'
PUBLIC  ??_C@_0BE@OCM@can?8t?5open?5file?3?5?$CFs?$AA@       ; `string'
PUBLIC  ??_C@_0CH@BNAK@File?5opened?5and?5closed?5without?5e@ ; `string'

;;; These are external functions
EXTRN   _fclose:NEAR
EXTRN   _fopen:NEAR
EXTRN   _printf:NEAR
EXTRN   _eprintf:NEAR
EXTRN   _setprogname:NEAR

;;; I think this one is a debugging function used to check that the
;;; stack frame is restored correctly.
```

```
EXTRN   __chkesp:NEAR

;       COMDAT ??_C@_0BO@HHKP@Usage?3?5error_wrapper?5filename?$AA@
; File H:\d241_a00\assembly_example\error_wrapper_test.c

;;; .CONST is used to define constant data that must be stored in
;;; memory.
;;; SEGMENT We define data in segments
;;; ??_C... I think is a label
;;; DB: byte aligned
;;; 'Usage...DB...filename, 00H: Data (a null terminated string)
;;; 'string': class, used to organize segments
;;; ENDS: end of this segment
CONST   SEGMENT
??_C@_0BO@HHKP@Usage?3?5error_wrapper?5filename?$AA@ DB 'Usage:
error_wra'
        DB      'pper filename', 00H                          ; `string'
CONST   ENDS

;;; More static string definitions
;       COMDAT ??_C@_01LHO@r?$AA@
CONST   SEGMENT
??_C@_01LHO@r?$AA@ DB 'r', 00H                         ; `string'
CONST   ENDS
;       COMDAT ??_C@_0BE@OCM@can?8t?5open?5file?3?5?$CFs?$AA@
CONST   SEGMENT
??_C@_0BE@OCM@can?8t?5open?5file?3?5?$CFs?$AA@ DB 'can''t open file:
%s', 00H ; `string'
CONST   ENDS
;       COMDAT ??_C@_0CH@BNAK@File?5opened?5and?5closed?5without?5e@
CONST   SEGMENT
??_C@_0CH@BNAK@File?5opened?5and?5closed?5without?5e@ DB 'File opened
and'
        DB      ' closed without errors', 0aH, 00H            ; `string'
CONST   ENDS

;       COMDAT _main
;;; Start of text (code) segment
_TEXT   SEGMENT

;;; To access argument argc in the stack frame we can add _argc$ to the
;;; address EBP points to.
;;; At memory location [EBP] + 4 is the return address of the function
;;; that called this function
;;; At memory location [EBP] is the old stack frame pointer.
_argc$ = 8
_argv$ = 12
;;; If we had a third argument it would be at memory location [EBP] +
;;; 16


;;; To access local variable f in the stack frame we can add _f$ to the
;;; address EBP points to.
_f$ = -4
;;; If we had more local variables they would be at memory location:
;;; [EBP] - 8, [EBP] - 12... (Note that even char's use 4 bytes, we
;;; cannot push one byte on the stack)
```

```
;;; Main is a public procedure, and the code starts here.
_main   PROC NEAR                                       ; COMDAT


;;; C code
; 8    : {

;;; Epilogue code
;;;
;;; Save the old stack frame pointer
        push    ebp
;;; Establish a new stack frame
        mov     ebp, esp
;;; Create room for local variables. I don't know why it subtracts 68
;;; bytes when there is only one local variable. Performance?
        sub     esp, 68                                 ; 00000044H
;;; Save callee saved registers used in this function.
        push    ebx
        push    esi
        push    edi

;;; I think the following code "clears" the stack area reserved for
;;; local variables.
;;;
;;; Compute the effective address of the old stack frame pointer and
;;; store it in EDI.
        lea     edi, DWORD PTR [ebp-68]
;;; We want to repeat the strings instruction (stosd) 17 times.
        mov     ecx, 17                                 ; 00000011H
;;; I have no idea why the value ccccccccH is used.
        mov     eax, -858993460                         ; ccccccccH
;;; for (i = 0; i < 17; i++)
;;;   Store EAX at address (EDI + 4 * i)
        rep stosd

; 9    :    FILE *f;
; 10   :
; 11   :    /* First argument is the name of the executable file */
; 12   :    setprogname(argv[0]);

;;; Move argument argv into eax
        mov     eax, DWORD PTR _argv$[ebp]
;;; Move argv[0] into ecx
        mov     ecx, DWORD PTR [eax]
;;; Push the first (and only) argument...
        push    ecx
;;; ...and call the function
        call    _setprogname
;;; Pop the argument.
        add     esp, 4

; 13   :
; 14   :    /* Second argument is the file to be opened */
; 15   :    if (argc < 2)

;;; In IA-32 on of the operands can be in memory
```

```
        cmp     DWORD PTR _argc$[ebp], 2
;;; If 1st operand >= 2nd operand then goto label $L363
        jge     SHORT $L363

; 16   :        eprintf("Usage: error_wrapper filename");

;;; Push static string as 1st argument
        push    OFFSET
FLAT:??_C@_0BO@HHKP@Usage?3?5error_wrapper?5filename?$AA@ ; `string'
;;; And call eprintf
        call    _eprintf
;;; Pop argument
        add     esp, 4

;;; Label to jump to, if argc >= 2
$L363:

; 17   :
; 18   :    f = fopen(argv[1], "r");

;;; Push second argument, the static string "r"
        push    OFFSET FLAT:??_C@_01LHO@r?$AA@        ; `string'
        mov     edx, DWORD PTR _argv$[ebp]
;;; Move argv[1] into eax...
        mov     eax, DWORD PTR [edx+4]
;;; ...and push it as the second argument...
        push    eax
;;; ...to function fopen, which is called.
        call    _fopen
;;; Pop the arguments
        add     esp, 8
;;; Save the reurn value on the stack
;;; When compiled with debugging on every time a local variable is
changed,
;;; is stored on the stack
        mov     DWORD PTR _f$[ebp], eax

; 19   :    if (f == NULL)

        cmp     DWORD PTR _f$[ebp], 0
        jne     SHORT $L367

; 20   :      eprintf("can't open file: %s", argv[1]);

;;; Push argv[1] (the second argument)
        mov     ecx, DWORD PTR _argv$[ebp]
        mov     edx, DWORD PTR [ecx+4]
        push    edx

;;; And push the static string as the first argument
        push    OFFSET
FLAT:??_C@_0BE@OCM@can?8t?5open?5file?3?5?$CFs?$AA@ ; `string'
        call    _eprintf
;;; Pop arguments
        add     esp, 8

;;; Jump to this label if (f != NULL)
```

```
$L367:

; 21   :   fclose(f);

;;; Move local variable f into eax, push it, call fclose, and pop the
;;; argument.
        mov     eax, DWORD PTR _f$[ebp]
        push    eax
        call    _fclose
        add     esp, 4

; 22   :
; 23   :   printf("File opened and closed without errors\n");

        push    OFFSET
FLAT:??_C@_0CH@BNAK@File?5opened?5and?5closed?5without?5e@ ; `string'
        call    _printf
        add     esp, 4

; 24   :
; 25   :   return 0;

;;; Return value is stored in eax
;;; xor eax, eaxc, is a fast way to store zero in eax
        xor     eax, eax

; 26   : }

;;; Epilouge code
;;;

;;; Restore saved calle save registers
        pop     edi
        pop     esi
        pop     ebx
;;; Pop the stack frame
        add     esp, 68                                 ; 00000044H

;;; I think this is a test to check that the old stack frame is
;;; restored
        cmp     ebp, esp
        call    __chkesp

;;; Restore the old stack frame.
        mov     esp, ebp
;;; Pop old stack frame pointer
        pop     ebp
;;; Return without poping any registers.
        ret     0

;;; End of procedure main
_main   ENDP
;;; End of code segment
_TEXT   ENDS
;; End of the source code
END
```

5 Examples

This chapter contains a lot of examples. You should read and understand them all.

5.1 Arithmetic Instructions

C program:

f = (g + h) - (i + j);

Assembly:

```
; We assume that f, g, h, i and j are assigned to registers EAX, EBX,
ECX, EDX and ESI
mov EDI, EBX ; EDI = g
add EDI, ECX ; EDI = g + h
mov EAX, EDI ; EAX = (g + h)
mov EDI, EDX ; EDI = i
add EDI, ESI ; EDI = i + j
sub EAX, EDI ; EAX = (g + h) - (i + j)
```

5.2 Data Transfer (mov instruction)

```
.DATA
  a_letter DB 'c'            ; Allocate one byte of memory, initialize
                             ; it to 'c'.
  array DD 20 DUP (0)        ; Array of 20 integers initialized to zero
  qwa   SQWORD 25 DUP (?)    ; Array of 25 quadwords (64 bits),
                             ; uninitialized
mov  EAX, EBX                ; EAX = EBX
mov  EAX, 132                ; EAX = 132
mov  a_letter, BYTE PTR EAX  ; memory[a_letter] = AL (8 lsb of EAX)
mov  EAX, [ESP]              ; EAX = memory[ESP]
mov  ECX, OFFSET array       ; ECX = &(array[0])
mov  ECX, array              ; ECX = array[0]
mov  EAX, array[ESI*4]       ; EAX = EAX + memory[ OFFSET table +
                             ;                      ESI * 4]
mov  EAX, [EBX+ESI]          ; EAX = memory[EBX + (ESI * 1) + 0]
mov  EAX, [EBX+ESI*4+2]      ; EAX = memory[EBX + (ESI * 4) + 2]
mov  ECX, OFFSET array       ; ECX = &(array[0])
mov  [ECX], SQWORD PTR 25    ; memory[ECX] = array[0] = 25
```

5.2 Jumps

5.2.1 Unconditional Jump

Infinite loop:

```
forever:
    jmp forever
```

5.2.2 Conditional Jumps

If then else

```
if (a < 0) {
        b = -5;
}
else if (a > 0) {
        b = 5;
}
else {
        b = 0;
}
; Assume that: a is in EAX, and that b is assigned to EBX

        cmp EAX, 0
        jge larger      ; if (a >= 0) goto larger;
smaller:                ; a < 0
        mov EBX, -5     ; b = -5
        jmp exit_if

larger:
        cmp EAX, 0
        jle equal       ; if (a == 0) goto (we know that a >= 0, so it
                        ;                   cannot be < 0)
        mov EBX, 5      ; b = 5
        jmp exit_if

equal:
        mov EBX, 0      ; b = 0

exit_if:                ; End of if then else
```

Loop

```
int i, vector[25];

for (i = 0; i < 25; i++)
  vector[i] = 0;
```

Assembly:

```
; vector[] is allocated memory on the stack
_vector = -112  ; Start address of vector is EBP - 112
; Remember that the stack grows downward, and that local
; variables are below the frame pointer.
; You should also note that:
; vector[0]  = memory[ebp + _vector]
; vector[13] = memory[ebp + _vector + 13*4]

start_loop:
        mov ECX, 0    ; i = 0
        jmp init_loop

loop:
        add ECX, 1    ; i++

init_loop:
        cmp ECX, 25
        jge exit_loop ; if (i >= 25) goto exit_loop

body:
        ; memory[ebp + _vector + ecx * 4] = 0
        ; DWORD PTR because we want to move a double word (remember
        ; that vector is an array of int's)
        mov DWORD PTR _vector[ebp + ecx * 4], 0
end_body:

        jmp loop

exit_loop:
```

5.3 Function calls

C code:

```
void *emalloc(size_t size)
{
        void *rp;

        if (((rp = malloc(size))) == NULL) {
                printf("Malloc error\n");
                exit(1);
        }

        return rp;
}
```

Assembly:

```
PUBLIC  _emalloc

.DATA
```

```
        malloc_string DB 'Malloc error',13,0 ; Allocate memory for a
                                             ; null terminated string
_size$ = 8    ; memory[ebp + _size$] = argument
_rp$ = -4     ; memory[ebp + _rp$]   = local variable
_emalloc:
        push    ebp                     ; Save old stack pointer
        mov     ebp, esp                ; Create a new stack frame
        sub     esp, 4                  ; Allocate memory for local
                                        ; variabels

        mov     eax, _size$[ebp]        ; Move argument 'size' to EAX
        push    eax                     ; Push malloc() argument
        call    _malloc                 ; Call malloc
        add     esp, 4                  ; Pop arguments
        mov     _rp$[ebp], eax          ; Save return value on the stack
        cmp     eax, 0                  ; Return value is in EAX (is
                                        ; compared to 0 = NULL)
        jne     no_error                ; if (return value != NULL) goto
                                        ;   no_error

error:
        push    OFFSET malloc_string    ; First argument to printf() is
                                        ; the address of the string
        call    _printf                 ; Call printf
        add     esp, 4                  ; Pop argument

        push    1                       ; First argument to exit()
        call    _exit                   ; Call exit
        ; Note that we never return from exit()

no_error:
        mov     eax, _rp$[ebp]          ; Move return value into eax
        mov     esp, ebp                ; Restore stack pointer
        pop     ebp                     ; Restore old stack frame
        ret                             ; Return to caller
```

5.4 The most useful IA-32 Instructions

Category	Instruction	Example	Meaning
Arithmetic	add	add EAX, EBX	EAX = EAX + EBX
	subtract	sub EAX, EBX	EAX = EAX - EBX
	add immediate	add EAX, 200	EAX = EAX + 200
	add unsigned[1]	add EAX, EBX	EAX = EAX + EBX
	add immediate unsigned	Don't exist	
	multiply	imul EBX	EDX:EAX = EAX * EBX
		imul ECX, EBX	ECX = ECX * EBX
		imul ECX, EBX, 200	ECX = EBX * 200

		imul ECX, 200	ECX = ECX * 200
	multiply unsigned	mul ECX	EDX:EAX = EAX * ECX
	divide	idiv ECX	EAX = EDX:EAX / ECX, EDX = EDX:EAX % ECX
	divide unsinged	div ECX	EAX = EDX:EAX / ECX, EDX = EDX: EAX % ECX
Logical	and	and EAX, EBX	EAX = EAX & EBX
	or	or EAX, EBX	EAX = EAX | EBX
	shift left logical	shl EAX, EBX	EAX = EAX << EBX
	shift right logical	shr EAX, EBX	EAX = EAX >> EBX
	xor	xor EAX, EBX	EAX = EAX ^ EBX
Data transfer	mov[2]	mov EAX, EBX	EAX = EBX
		mov EAX, 200	EAX = 200
		mov EAX, [ESP]	EAX = memory[ESP]
		mov EAX, label	EAX = memory[label]
		mov EAX, OFFSET array	EAX = &(array[0])
	[3]	mov EAX, table[ESI*4]	AX = memory[OFFSET table + ESI * 4]
	[3]	mov E AX, [EBX+ESI*4+2] ;	E AX = memory[E BX + (ESI * 4) + 2]
	push	push EAX	ESP = ESP - 4; memory[ESP] = EAX
	pop	pop EAX	EAX = memory[ESP]; ESP = ESP + 4
Compare	compare	cmp EAX, EBX	EAX - EBX. Set control flags.
Conditional jumps	jump if equal[4]	je label	Jump to label if Zero Flag (ZF) is set

	jump if zero[4]	jz label	Jump to label if ZF = 1
	jump if not equal[4]	jne label	Jump to label if ZF = 0
	jump if not zero[4]	jnz label	Jump to label if ZF = 0
	jump if CX is zero	jcxz label	Jump to label if CX = 0
	jump if carry	jc	Jump to label if Carry Flag (CF) is set
	jump if not carry	jnc	Jump to label if CF = 0
	jump if overflow	jo label	Jump to label if Overflow Flag (OF) is set
	jump if not overlow	jno label	Jump to label if OF = 0
	jump is sign	js label	Jump to label if Sign Flag (SF) is set (negative sign)
	jump is not sign	jns label	Jump to label if SF = 0 (postive sign)
Unconditional jump	jump	jmp label	Jump to label
	jump register	jmp EAX	Jump to address in EAX
Instruction call	call	call function_label	Push EIP and jump to function_label
Instruction return	ret	ret	Pop return address, and jump to it
Other	increment	inc EAX	EAX = EAX + 1
	decrement	dec EAX	EAX = EAX - 1
	no operation	nop	Do nothing
	test	test EAX, EBX	EAX & EBX. Set control flags.
	exchange values	xchg EAX, EBX	tmp = EAX; EAX = EBX; EBX = tmp

[1] The processor don't care if it is a signed value, it evaluates the result for both values.

[2] Note that IA-32 is not a load-store architecture; most of the instructions can have one of the operands in memory.

[3] Only ESI and EDI can be used as the displacement register.

[4] Use cmp or test to set control flags (ZF, CF, OF, SF, PF).

6 References

Sivarama, P. Dandamudi, *Introduction To Assembly Language Programming; From 8086 to Pentium Processor*, Springer 1998

Patterson, David A. / Hennesy, John L., *Computer Organization & Design The Hardware / Software Interface, Second Edition*, Morgan Kaufmann Publishers 1998

Intel Architecture Software Developers Manual Volume 1, 2 and 3, Intel

Microsoft MASM Programmers Guide, Microsoft

Shanley, Paul, *Protected Mode Software Architecture,* Mindshare Inc 1996

Li, Kay, *A Guide to Programming Pentium/Pentium Pro Processors*, Princeton University