

Chapter 1

x86 Assembly

...

This chapter was derived from a document written by Adam Ferrari and later updated by Alan Batson, Mike Lack and Anita Jones

1.1 Introduction

This small guide, in combination with the material covered in the class lectures on assembly language programming, should provide enough information to do the assembly language labs for this class. In this guide, we describe the basics of 32-bit x86 assembly language programming, covering a small but useful subset of the available instructions and assembler directives. However, real x86 programming is a large and extremely complex universe, much of which is beyond the useful scope of this class. For example, the vast majority of real (albeit older) x86 code running in the world was written using the 16-bit subset of the x86 instruction set. Using the 16-bit programming model can be quite complex – it has a segmented memory model, more restrictions on register usage, and so on. In this guide we restrict our attention to the more modern aspects of x86 programming, and delve into the instruction set only in enough detail to get a basic feel for programming x86 compatible chips at the hardware level.

1.2 Registers

Modern (i.e 386 and beyond) x86 processors have 8 32-bit general purpose registers, as depicted in Figure 1.1. The register names are mostly historical in nature. For example, EAX used to be called the “accumulator” since it was used by a number of arithmetic operations, and ECX was known as the “counter” since it was used to hold a loop index. Whereas most of the registers have lost their special purposes in the modern instruction set, by convention, two are reserved for special purposes – the stack pointer (ESP) and the base pointer (EBP).

In some cases, namely EAX, EBX, ECX, and EDX, subsections of the registers may be used. For example, the least significant 2 bytes of EAX can be treated as a 16-bit register called AX. The least significant byte of AX can be used as a single 8-bit register called AL, while the most significant byte of AX can be used as a single 8-bit register called AH. It is important to realize that these names refer to the same

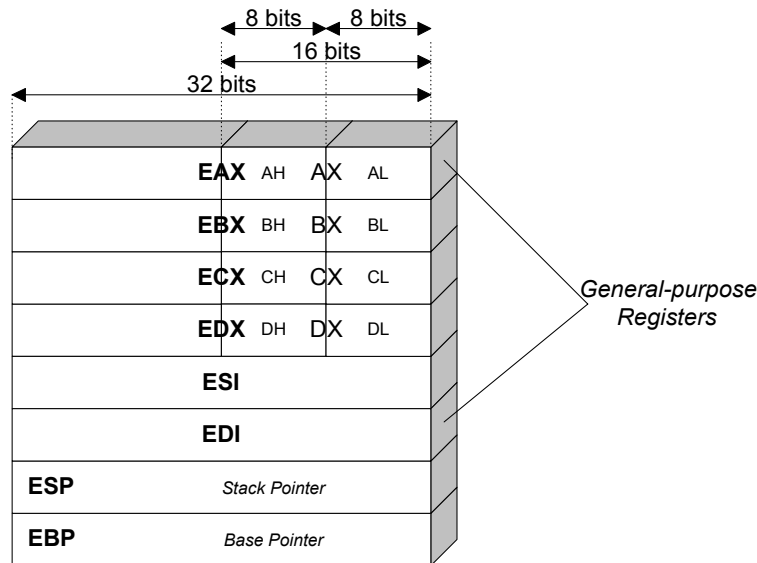


Figure 1.1: The x86 register set

physical register. When a two-byte quantity is placed into DX, the update affects the value of EDX (in particular, the least significant 16 bits of EDX). These “sub-registers” are mainly hold-overs from older, 16-bit versions of the instruction set. However, they are sometimes convenient when dealing with data that are smaller than 32-bits (e.g. 1-byte ASCII characters).

When referring to registers in assembly language, the names are not case-sensitive. For example, the names EAX and eax refer to the same register.

1.3 Memory and Addressing Modes

1.3.1 Declaring Static Data Regions

You can declare static data regions (analogous to global variables) in x86 assembly using special assembler directives for this purpose. Data declarations should be preceded by the `.DATA` directive. Following this directive, the directives `DB`, `DW`, and `DD` can be used to declare one, two, and four byte data locations, respectively. Declared locations can be labeled with names for later reference - this is similar to declaring variables by name, but abides by some lower level rules. For example, locations declared in sequence will be located in memory next to one another. Some example declarations are depicted in Listing 1.1.

The last example in Listing 1.1 illustrates the declaration of an array. Unlike in high level languages where arrays can have many dimensions and are accessed by indices, arrays in assembly language are simply a number of cells located contiguously in memory. Two other common methods used for declaring arrays of data are the `DUP` directive and the use of string literals. The `DUP` directive tells the assembler to duplicate an expression a given number of times. For example, the statement “4 DUP(2)” is equivalent to “2, 2, 2, 2”. Some examples of declaring arrays are depicted in Listing 1.2.

Listing 1.1: Declaring x86 memory regions

```

section .data
var      DB      64      ; Declare a byte containing the value 64. Label the
                        ; Memory location "var".
var2     DB      ?       ; Declare an uninitialized byte labeled "var2".
          DB      10      ; Declare an unlabeled byte initialized to 10. This
                        ; byte will reside at the memory address var2+1.
X        DW      ?       ; Declare an uninitialized two-byte word labeled "X".
Y        DD      3000     ; Declare 32 bits of memory starting at address "Y"
                        ; initialized to contain 3000.
Z        DD      1,2,3    ; Declare three 4-byte words of memory starting at
                        ; address "Z", and initialized to 1, 2, and 3,
                        ; respectively. E.g. 3 will be stored at address Z+8.

```

Listing 1.2: Declaring x86 arrays in memory

```

section .data
bytes    DB      10      DUP(?) ; Declare 10 uninitialized bytes starting at
                        ; the address "bytes".
arr      DD      100     DUP(0) ; Declare 100 4 bytes words, all initialized
                        ; to 0, starting at memory location "arr".
str      DB      'hello', 0    ; Declare 5 bytes starting at the address
                        ; "str" initialized to the ASCII character
                        ; values for the characters 'h', 'e', 'l',
                        ; 'l', 'o', and '\0' (NULL), respectively.

```

1.3.2 Addressing Memory

Modern x86-compatible processors are capable of addressing up to 2^{32} bytes of memory; that is, memory addresses are 32-bits wide. For example, in Listings 1.1 and x86-declaring-arrays.lst, where we used labels to refer to memory regions, these labels are actually replaced by the assembler with 32-bit quantities that specify addresses in memory. In addition to supporting referring to memory regions by labels (i.e. constant values), the x86 provides a flexible scheme for computing and referring to memory addresses:

X86 Addressing Mode Rule – Up to two of the 32-bit registers and a 32-bit signed constant can be added together to compute a memory address. One of the registers can be optionally pre-multiplied by 2, 4, or 8.

To see this memory addressing rule in action, we'll look at some example `mov` instructions. As we'll see later in Section 1.4.1, the `mov` instruction moves data between registers and memory. This instruction has two operands – the first is the destination (where we're moving data *to*) and the second specifies the source (where we're getting the data *from*). Some examples of `mov` instructions using address computations that obey the above rule are shown in Listing 1.3.

Some examples of incorrect address calculations are shown in Listing 1.4.

Listing 1.3: Valid x86 addressing modes

mov eax, [ebx]	<i>; Move the 4 bytes in memory at the address contained ; in EBX into EAX</i>
mov [var], ebx	<i>; Move the contents of EBX into the 4 bytes at memory ; address "var" (Note, "var" is a 32-bit constant).</i>
mov eax, [esi-4]	<i>; Move 4 bytes at memory address ESI+(-4) into EAX</i>
mov [esi+eax], cl	<i>; Move the contents of CL into the byte at address ; ESI+EAX</i>
mov edx, [esi+4*ebx]	<i>; Move the 4 bytes of data at address ESI+4*EBX into ; EDX</i>

Listing 1.4: Invalid x86 addressing modes

mov eax, [ebx-ecx]	<i>; Can only add register values</i>
mov [eax+esi+edi], ebx	<i>; At most 2 registers in address computation</i>

1.3.3 Size Directives

In general, the intended size of the of the data item at a given memory address can be inferred from the assembly code instruction in which it is referenced. For example, in all of the above instructions, the size of the memory regions could be inferred from the size of the register operand – when we were loading a 32-bit register, the assembler could infer that the region of memory we were referring to was 4 bytes wide. When we were storing the value of a one byte register to memory, the assembler could infer that we wanted the address to refer to a single byte in memory. However, in some cases the size of a referred-to memory region is ambiguous. Consider the instruction `mov [ebx], 2`.

Should this instruction move the value 2 into the single byte at address EBX? Perhaps it should move the 32-bit integer representation of 2 into the 4-bytes starting at address EBX. Since either is a valid possible interpretation, the assembler must be explicitly directed as to which is correct. The size directives `BYTE PTR`, `WORD PTR`, and `DWORD PTR` serve this purpose. For examples, see Listing 1.5.

Listing 1.5: x86 size directive usage

mov BYTE PTR [ebx], 2	<i>; Move 2 into the single byte at memory ; location EBX</i>
mov WORD PTR [ebx], 2	<i>; Move the 16-bit integer representation of 2 ; into the 2 bytes starting at address EBX</i>
mov DWORD PTR [ebx], 2	<i>; Move the 32-bit integer representation of 2 ; into the 4 bytes starting at address EBX</i>

1.4 Instructions

Machine instructions generally fall into three categories: data movement, arithmetic/logic, and control-flow. In this section, we will look at important examples of x86 instructions from each category. This section should not be considered an exhaustive list of x86 instructions, but rather a useful subset.

In this section, we will use the following notation:

- `<reg32>` - means any 32-bit register described in Section 2, for example, ESI.
- `<reg16>` - means any 16-bit register described in Section 2, for example, BX.
- `<reg8>` - means any 8-bit register described in Section 2, for example AL.
- `<reg>` - means any of the above.
- `<mem>` - will refer to a memory address, as described in Section 1.3.2, for example `[EAX]`, or `[var+4]`, or `DWORD PTR [EAX+EBX]`.
- `<con32>` - means any 32-bit constant.
- `<con16>` - means any 16-bit constant.
- `<con8>` - means any 8-bit constant.
- `<con>` - means any of the above sized constants.

1.4.1 Data Movement Instructions

Instruction: mov

Syntax:

```
mov <reg>, <reg>
mov <reg>, <mem>
mov <mem>, <reg>
mov <reg>, <const>
mov <mem>, <const>
```

Semantics: The `mov` instruction moves the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.

Examples:

```
mov eax, ebx           ; transfer ebx to eax
mov BYTE PTR [var], 5  ; store the value 5 into the byte at
                        ; memory location ``var``
```

Instruction: push

Syntax:

```
push <reg32>
push <mem>
push <con32>
```

Semantics: The `push` instruction places its operand onto the top of the hardware supported stack in memory. Specifically, `push` first decrements ESP by 4, then places its operand into the contents of the 32-bit location at address `[ESP]`. ESP (the stack pointer) is decremented by `push` since the x86 stack grows down – i.e. the stack grows from high addresses to lower addresses.

Examples:

```
push eax      ; push the contents of eax onto the stack
push [var]    ; push the 4 bytes at address ``var`` onto stack
```


Instruction: imul

Syntax:

```
imul <reg32>, <reg32>
imul <reg32>, <mem>
imul <reg32>, <reg32>, <con>
imul <reg32>, <mem>, <con>
```

Semantics: The imul instruction has two basic formats: two-operand (first two syntax listings above) and three-operand (last two syntax listings above). The two-operand form multiplies its two operands together and stores the result in the first operand. The result (i.e. first) operand must be a register. The three operand form multiplies its second and third operands together and stores the result in its first operand. Again, the result operand must be a register. Furthermore, the third operand is restricted to being a constant value.

Examples:

```
imul eax, [var]          ; multiply the contents of EAX by the
                        ; 32-bit contents of the memory location
                        ; 'var'. Store the result in EAX.
imul esi, edi, 25        ; multiply the contents of EDI by 25.
                        ; Store the result in ESI.
```

Instruction: idiv

Syntax:

```
idiv <reg32>
idiv <mem>
```

Semantics: The idiv instruction is used to divide the contents of the 64 bit integer EDX:EAX (constructed by viewing EDX as the most significant four bytes and EAX as the least significant four bytes) by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX. This instruction must be used with care. Before executing the instruction, the appropriate value to be divided must be placed into EDX and EAX. Clearly, this value is overwritten when the idiv instruction is executed.

Examples:

```
idiv ebx                ; divide the contents of EDX:EAX by the
                        ; contents of EBX. Place the quotient
                        ; in EAX and the remainder in EDX.
idiv DWORD PTR [var]    ; same as above, but divide by the
                        ; 32-bit value stored at memory
                        ; location 'var'.
```

Instruction: and, or, xor

Syntax:

```
and <reg>, <reg>      or <reg>, <reg>      xor <reg>, <reg>
and <reg>, <mem>      or <reg>, <mem>      xor <reg>, <mem>
and <mem>, <reg>      or <mem>, <reg>      xor <mem>, <reg>
and <reg>, <con>      or <reg>, <con>      xor <reg>, <con>
and <mem>, <con>      or <mem>, <con>      xor <mem>, <con>
```

Semantics: These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.

Examples:

```
and eax, 0fH           ; clear all but the last 4 bits of EAX.
xor edx, edx           ; set the contents of EDX to zero.
```

Instruction: not

Syntax: not <reg>
 not <mem>

Semantics: Performs the logical negation of the operand contents (i.e. flips all bit values).

Examples: not BYTE PTR [var] ; negate all bits in the byte at the
 ; memory location ``var``.

Instruction: neg

Syntax: neg <reg>
 neg <mem>

Semantics: Performs the arithmetic (i.e. two's complement) negation of the operand contents.

Examples: neg eax ; negate the contents of EAX.
 neg [var] ; negate the contests of ``var``

Instruction: shl, shr

Syntax: shl <reg>, <con8> shr <reg>, <con8>
 shl <mem>, <con8> shr <mem>, <con8>
 shl <reg>, cl shr <reg>, cl
 shl <mem>, cl shr <mem>, cl

Semantics: These instructions shift the bits in their first operand's contents left and right (shl and shr, respectively), padding the resulting empty bit positions with zeros. The shifted operand can be shifted up to 31 places. The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL. In either case, shifts counts of greater than 31 are performed modulo 32.

Examples: shl eax 5 ; shift the contents of eax left by 5 bit
 ; positions
 shr [var] 3 ; shift the contents of ``var`` right by 3
 ; bit positions

1.4.3 Control Flow Instructions

In this section, we will refer to labeled locations in the program text as <label>. Labels can be inserted anywhere in x86 assembly code text by entering a label name followed by a colon. For example, consider the code fragment in Listing 1.6. The second instruction in this code fragment is labeled "begin". Elsewhere in the code, we can refer to the memory location that this instruction is located at in memory using the more convenient symbolic name "begin" instead of having to refer to the memory address as an integer.

Listing 1.6: x86 labeled code location

	mov esi, [ebp+8]
begin:	xor ecx, ecx
	mov eax, [esi]

Instruction: jmp

Syntax: `jmp <label>`

Semantics: Transfers program control flow to the instruction at the memory location indicated by the operand.

Examples: `jmp begin` ; jumps to the ``begin'' label

Instruction: jCC

Syntax: `je <label>` ; Jump when equal
`jne <label>` ; Jump when not equal
`jz <label>` ; Jump when last result was zero
`jg <label>` ; Jump when greater than
`jge <label>` ; Jump when greater than or equal to
`jle <label>` ; Jump when less than
`jle <label>` ; Jump when less than or equal to

Semantics: These instructions are conditional jumps that are based on the status of a set of **condition codes** that are stored in a special register called the *machine status word*. The contents of the machine status word include information about the last arithmetic operation performed. For example, one bit of this word indicates if the last result was zero. Another indicates if the last result was negative. Based on these condition codes, a number of conditional jumps can be performed. For example, the `jz` instruction performs a jump to the specified operand label if the result of the last arithmetic operation (e.g. `add`, `sub`, etc.) was zero. Otherwise, control proceeds to the next instruction in sequence after the `jz`. These conditional jumps are the underlying support needed to implement high-level language features such as “if” statements and loops (e.g. “while” and “for”).

A number of the conditional branches are given names that are intuitively based on the last operation performed being a special compare instruction, `cmp` (see below). For example, conditional branches such as `jle` and `jne` are based on first performing a `cmp` operation on the desired operands.

Examples: `cmp eax, ebx` ; if the contents of `eax` are less than or
`jle done` ; equal to the contents of `EBX`, jump to the
; code location labeled ``done''.

Instruction: cmp

Syntax: `cmp <reg>, <reg>`
`cmp <reg>, <mem>`
`cmp <mem>, <reg>`
`cmp <reg>, <con>`
`cmp <mem>, <con>`

Semantics: Compares the two specified operands, setting the condition codes in the machine status word appropriately. In fact, this instruction is equivalent to the `sub` instruction, except the result of the subtraction is discarded.

Examples: `cmp DWORD PTR [var], 10` ; if the 4 bytes stored at memory
`jeq loop` ; location ``var'' equal the 4-byte
; integer value 10, then jump to the
; code location labeled `loop`

(i.e. integer-sized) memory region labeled “var”.

Next in each file, we find the declaration of the function named `returnTwo`. In the C++ file we have declared the function to be `extern “C”`. This declaration indicates that the C++ compiler should use C naming conventions when labeling the function `returnTwo` in the resulting object file that it produces. In fact, this naming convention means that the function `returnTwo` should map to the label `_returnTwo` in the object code. In the assembly code, we have labeled the beginning of the subroutine `_returnTwo` using the `PROC` directive, and have declared the label `_returnTwo` to be public. Again, the result of these actions will be that the subroutine will map to the symbol `_returnTwo` in the object code that the assembler generates.

The function bodies are straight-forward. As we will see in more detail in Section 6, return values for functions are placed into EAX by convention, hence the instruction to move the contents of “var” into EAX in the assembly code.

Given these equivalent function definitions, use of either version of the function is the same. A sample call to the function `returnTwo` is depicted in Listing 1.9. This C++ code could be linked to either definition of the function and would produce the same results (note, we could not link to both definitions, or the linker would produce a “multiply defined symbol” error. The mechanics of program linking will be discussed in an associated document that relates to the specific programming environment that you will use to assemble and run programs.

Listing 1.8: C++ code to return 2

```
int var = 2;

extern "C" returnTwo();

int returnTwo() {
    return var;
}
```

Listing 1.9: Calling `returnTwo()` from C++

```
#include <iostream>
using namespace std;

extern "C" int returnTwo();

int main() {
    cout << "calling _returnTwo() _returned: "
         << returnTwo() << endl;
    return 0;
}
```