Chapter 1

x86 Assembly, 32 bit

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

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, there exists 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'll restrict our attention to the more modern aspects of 32-bit 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 eight 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 TIMES directive and the use of string literals. The TIMES directive tells the assembler to duplicate an expression a given number of times. For example, the statement "TIMES 4 DB 2" is equivalent to "2, 2, 2, 2". Some examples of declaring arrays are depicted in Listing 1.2.

			0 70
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 1.1: Declaring x86 memory regions

Listing 1.2: Declaring x86 arrays in memory

section	.data						
bytes	TIMES	10 DB	?	eclare 10 unini	tialized by	tes starting	a t
				e address "byte	<i>es</i> ″.		
arr	TIMES	100 DD	0	eclare 100 4 byt	tes words, a	all initializ	e d
				0, starting at	memory loc	cation "arr".	
str	DB	'hello '	, 0	eclare 5 bytes s	starting at	the address	
				tr" initialized	to the AS	CII character	r
				lues for the ch	naracters 'h	ı', 'e', 'l',	,
				', 'o', and ' $\setminus 0$)' (NULL), 1	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 1.2, 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.

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

Listing 1.3: Valid x86 addressing modes

Libring 1.1. Invalia Xoo daalebbing model	Listing 1.4:	Invalid	x86	addressi	ng	modes
---	--------------	---------	-----	----------	----	-------

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

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

1.4 Instructions

Machine instructions generally fall into three categories: data movement, arithmetic/logic, and controlflow. 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></reg></reg>	
	mov <reg>,<mem></mem></reg>	
	mov <mem>,<reg></reg></mem>	
	mov <reg>,<const></const></reg>	
	mov <mem>,<const></const></mem>	
Semantics:	The mov instruction moves the	data item referred to by its second operand (i.e. reg-
	ister contents, memory contents	, or a constant value) into the location referred to by
	its first operand (i.e. a register o	r memory). While register-to-register moves are pos-
	sible, direct memory-to-memory	moves are not. In cases where memory transfers are
	desired, the source memory con	tents must first be loaded into a register, then can be
	stored to the destination memor	y 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></reg32>
	push <mem></mem>
	push <con32></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 decre- mented by push since the x86 stack grows down – i.e. the stack grows from high
	addresses to lower addresses.
Examples:	<pre>push eax ; push the contents of eax onto the stack push [var] ; push the 4 bytes at address ``var'' onto stack</pre>

Instruction: pop

Syntax:	pop <reg32></reg32>
	pop <mem></mem>
Semantics:	The pop instruction removes the 4-byte data element from the top of the hardware
	supported stack into the specified operand (i.e. register or memory location). Specifi-
	cally, pop first moves the 4 bytes located at memory location [ESP] into the specified
	register or memory location, and then increments SP by 4.
Examples:	pop edi ; pop the top element of the stack into EDI.
,	pop [ebx] ; pop the top element of the stack into memory at
	; the four bytes starting at location EBX.
Instruction: lea	
Syntax:	lea <reg32>,<mem></mem></reg32>
Semantics:	The lea instruction places the address specified by its second operand into the regis-
	ter specified by its first operand. Note, the contents of the memory location are not
	loaded – only the effective address is computed and placed into the register. This is
	useful for obtaining a "pointer" into a memory region.
Examples:	<pre>lea eax, [var] ; the address of ``var'' is placed in EAX</pre>
,	lea edi, [ebx+4*esi] ; the value EBX+4*ESI is placed in EDI

1.4.2 Arithmetic and Logic Instructions

Instruction: add, sub

Syntax:	add <reg>,<reg></reg></reg>	<pre>sub <reg>, <reg></reg></reg></pre>			
	add <reg>,<mem></mem></reg>	<pre>sub <reg>,<mem></mem></reg></pre>			
	add <mem>,<reg></reg></mem>	<pre>sub <mem>,<reg></reg></mem></pre>			
	add <reg>,<con></con></reg>	<pre>sub <reg>,<con></con></reg></pre>			
	add <mem>,<con></con></mem>	<pre>sub <mem>,<con></con></mem></pre>			
Semantics:	The add instruction adds together its two operands, storing the result in its first operand. Similarly, the sub instruction subtracts its second operand from its first. Note whereas both operands may be registers, at most one operand may be a mem-				
	ory location.	lay be registers, at most one operation may be a ment			
Examples:	add eax, 10	; add 10 to the contents of EAX.			
·	sub [var], esi	; subtract the contents of ESI from ; the 32-bit integer stored at ; memory location ``var''.			
	add BYTE PTR [var], 10	; add 10 to the single byte stored ; at memory address ``var''.			

Instruction: inc, dec

Syntax:	inc <reg></reg>	dec <reg></reg>		
	inc <mem></mem>	dec <mem></mem>		
Semantics:	The inc instruction inc	crements the contents of its operand by one, and similarly dec		
	decrements the contents of its operand by one.			
Examples:	dec eax	; subtract one from the contents of EAX.		
	inc DWORD PTR [va	ar] ; add one to the 32-bit integer stored at		
		; memory location ``var''.		

1.4. INSTRUCTIONS

Instruction: imu	1					
Syntax:	<pre>imul <reg32>, <reg32></reg32></reg32></pre>	>				
	imul <reg32>,<mem></mem></reg32>					
	<pre>imul <reg32>, <reg32></reg32></reg32></pre>	>, <con></con>				
	<pre>imul <reg32>, <mem>, <con></con></mem></reg32></pre>					
Semantics:	The imul instruction has tw	vo basic formats: two-op	perand (first two syntax listings			
	above) and three-operand	(last two syntax listings	above). The two-operand form			
	multiplies its two operands	s together and stores the	result in the first operand. The			
	result (i.e., first) operand m	ust be a register. The th	ree operand form multiplies its			
	second and third operands	together and stores the r	esult 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 con	ntents of EAX by the			
		; 32-bit contents	of the memory location			
		; ``var''. Store	the result in EAX.			
	imul esi, edi, 25	; multiply the con	ntents of EDI by 25.			
		; Store the result	t in ESI.			
Instruction: idiv	,					
Syntax:	idiv <reg32></reg32>					
C C	idiv <mem></mem>					
Semantics:	The idiv instruction is used	to divide the contents of t	the 64 bit integer EDX:EAX (con-			
	structed by viewing EDX as	the most significant four	r bytes and EAX as the least sig-			
	nificant four bytes) by the sp	pecified operand value. T	he 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	nd EAX. Clearly, this val	ue is overwritten when the idiv			
	instruction is executed.					
Examples:	idiv ebx	; divide the con	tents of EDX:EAX by the			
		; contents of EB	X. Place the quotient			
		; in EAX and the remainder in EDX.				
	idiv DWORD PTR [var] ; same as above, but divide by the					
		; 32-bit value s	tored at memory			
		; location ``var	· · · .			
Instruction: and	, or, xor					
Syntax:	and <reg>,<reg></reg></reg>	or <reg>,<reg></reg></reg>	xor <reg>,<reg></reg></reg>			
C C	and <reg>, <mem></mem></reg>	or <reg>, <mem></mem></reg>	xor <reg>, <mem></mem></reg>			
	and <mem>,<reg></reg></mem>	or <mem>,<reg></reg></mem>	xor <mem>,<reg></reg></mem>			
	and <reg>,<con></con></reg>	or <reg>,<con></con></reg>	xor <reg>,<con></con></reg>			
	and <mem>,<con></con></mem>	or <mem>,<con></con></mem>	<pre>xor <mem>,<con></con></mem></pre>			
Semantics:	These instructions perform	the specified logical opera	tion (logical bitwise and, or, and			
	exclusive or, respectively) of	on their operands, placin	g the result in the first operand			
	location.		_			
Examples:	and eax, OfH	; clear all but the	e last 4 bits of EAX.			
	xor edx, edx	; set the contents	of EDX to zero.			

Instruction: not	
Syntax:	not <reg></reg>
	not <mem></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></reg>
	neg <mem></mem>
Semantics:	Performs the arithmetic (i.e., two's complement) negation of the operand contents.
Examples:	neg eax ; negate the contents of EAX.
	<pre>neg [var] ; negate the contests of ``var''</pre>
Instruction: shl,	shr
Syntax:	<pre>shl <reg>,<con8> shr <reg>,<con8></con8></reg></con8></reg></pre>
	<pre>shl <mem>,<con8> shr <mem>,<con8></con8></mem></con8></mem></pre>
	<pre>shl <reg>,cl shr <reg>,cl</reg></reg></pre>
	<pre>shl <mem>,cl shr <mem>,cl</mem></mem></pre>
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 then 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]				
1	begin:	xor	ecx,	ecx				
		mov	eax,	[esi]				

8

1.4. INSTRUCTIONS

Instruction: jmp

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

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	CC	nter	nts	of	eax	ar	e l	ess	tha	in c	or
	jle	done		;	equ	al	to	the	cor	nter	nts d	of	EBX	κ, j	ump	to	the
				;	coc	le l	oca	tior	n la	abel	ed	`` d	one	. ''			

Instruction: cmp

Syntax:	<pre>cmp <reg>,<reg></reg></reg></pre>	
	<pre>cmp <reg>, <mem></mem></reg></pre>	
	<pre>cmp <mem>,<reg></reg></mem></pre>	
	<pre>cmp <reg>,<con></con></reg></pre>	
	<pre>cmp <mem>,<con></con></mem></pre>	
Semantics:	Compares the two specified opera	ands, setting the condition codes in the machine sta-
	tus word appropriately. In fact, t	his instruction is equivalent to the sub instruction,
	except the result of the subtraction	n 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

Instruction: call

Examples:

Syntax:	call <label></label>				
Semantics:	This instruction implements subroutine return instruction current code location onto the instruction for details), and the indicated by the label operare the simple jmp instruction) is completes.	a subroutine call that operates in cooperation with the a, ret, described below. This instruction first pushes the ne hardware supported stack in memory (see the push nen performs an unconditional jump to the code location ad. The added value of this instruction (as compared to that it saves the location to return to when the subroutine			
Examples:	call my_subroutine	; jumps to the ``my_subroutine'' label, ; pushing the return address onto the ; stack			
Instruction: ret					
Syntax:	ret				
Semantics:	In cooperation with the call instruction, the ret instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack (see the pop instruction for details). It then performs an unconditional jump to the retrieved code location.				

1.5 **Basic Program Structure**

ret

Given the above repertoire of instructions, you are in a position to examine the basic skeletal structure of an assembly language subroutine suitable for linking into C++ code. Unlike C++, which is often used for the development of complete software systems, assembly language is most often used in cooperation with other languages such as Fortran, C, and C++. Commonly, most of a project is implemented in the more convenient high-level language, and assembly language is used sparingly to implement extremely low-level hardware interfaces or performance-critical "inner loops." Thus, in addition to understanding how to program in assembly language, it is equally important to understand how to link assembly language code into high-level language programs.

Before examining the linkage conventions, we must first ex-

amine the basic structure of an assembly language file. To do this, we can compare a very simple assembly language file to an equivalent C++ file. In Listings 1.7 and 1.8 we see two files, one in C++, the other in x86 assembly. Each file includes a function (albeit an ugly one) to return the integer value 2.

The top of the assembly file contains two directives that indicate the instruction set and memory model we will use for all work in this class (note, there are other possibilities – one might use only the older 80286 instruction set for wider compatibility, for example).

Next, where in the C++ file we find the declaration of the global variable "var", in the assembly file we find the use of the .DATA and DD directives (described in Section 3.1) to reserve and initialize a 4-byte

Listing 1.7: x86 code to return 2

```
global returnTwo
section .data
        var DD 2
section .text
returnTwo:
        mov eax, [var]
        ret
```

; returns to the address on the top of the stack

1.5. BASIC PROGRAM STRUCTURE

(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.

```
Listing 1.8: C++ code to return 2
int var = 2;
extern "C" returnTwo();
int returnTwo() {
  return var;
}
```

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 1.9: Calling returnTwo() from C++

```
#include <iostream>
using namespace std;
extern "C" int returnTwo();
int main() {
   cout << "calling_returnTwo()_returned:_"
        << returnTwo() << endl;
   return 0;
}</pre>
```