

Introduction to Linux Intel Assembly Language

Norman Matloff

March 18, 2007

©2001-2007, N.S. Matloff

Contents

1	Overview of Intel CPUs	4
1.1	Computer Organization	4
1.2	CPU Architecture	4
1.3	The Intel Architecture	4
2	What Is Assembly Language?	5
3	Different Assemblers	6
4	Sample Program	6
4.1	Analysis	7
4.2	Source and Destination Operands	11
4.3	Remember: No Names, No Types at the Machine Level	11
4.4	Dynamic Memory Is Just an Illusion	12
5	Use of Registers Versus Memory	12
6	Another Example	12
7	Addressing Modes	16
8	Assembling and Linking into an Executable File	17
8.1	Assembler Command-Line Syntax	17
8.2	Linking	18
8.3	Makefiles	18

9	How to Execute Those Sample Programs	18
9.1	“Normal” Execution Won’t Work	18
9.2	Running Our Assembly Programs Using GDB/DDD	19
9.2.1	Using DDD for Executing Our Assembly Programs	19
9.2.2	Using GDB for Executing Our Assembly Programs	20
10	How to Debug Assembly Language Programs	21
10.1	Use a Debugging Tool for ALL of Your Programming, in EVERY Class	21
10.2	General Principles	21
10.2.1	The Principle of Confirmation	21
10.2.2	Don’t Just <u>Write</u> Top-Down, But <u>Debug</u> That Way Too	22
10.3	Assembly Language-Specific Tips	22
10.3.1	Know Where Your Data Is	22
10.3.2	Seg Faults	22
10.4	Use of DDD for Debugging Assembly Programs	23
10.5	Use of GDB for Debugging Assembly Programs	24
10.5.1	Assembly-Language Commands	24
10.5.2	TUI Mode	25
10.5.3	CGDB	25
11	Some More Operand Sizes	26
12	Some More Addressing Modes	27
13	GCC/Linker Operations	29
13.1	GCC As a Manager of the Compilation Process	29
13.1.1	The C Preprocessor	29
13.1.2	The Actual Compiler, CC1, and the Assembler, AS	30
13.2	The Linker	30
13.2.1	What Is Linked?	30
13.2.2	Headers in Executable Files	31
13.2.3	Libraries	31
14	Inline Assembly Code for C++	33
15	“Linux Intel Assembly Language”: Why “Intel”? Why “Linux”?	34

16 Viewing the Assembly Language Version of the Compiled Code	34
A String Operations	35
B Useful Web Links	36
C Top-Down Programming	37

1 Overview of Intel CPUs

1.1 Computer Organization

Computer programs execute in the computer's **central processing unit** (CPU). Examples of CPUs are the Pentiums in PCs and the PowerPCs in Macs.¹ The program itself, consisting of both instructions and data are stored in **memory** (RAM) during execution. If you created the program by compiling a C/C++ source file (as opposed to writing in assembly language directly), the instructions are the machine language operations (add, subtract, copy, etc.) generated from your C/C++ code, while the data are your C/C++ variables. The CPU must fetch instructions and data from memory (as well as store data to memory) when needed; this is done via a **bus**, a set of parallel wires connecting the CPU to memory.

The components within the CPU include **registers**. A register consists of bits, with as many bits as the word size of the machine. Thus a register is similar to one word of memory, but with the key difference that a register is inside the CPU, making it much faster to access than memory. Accordingly, when programming in assembly language, we try to store as many of our variables as possible in registers instead of in memory. Similarly, if one invokes a compiler with an “optimize” command, the compiler will also attempt to store variables in registers instead of in memory (and will try other speedup tricks as well).

1.2 CPU Architecture

There are many different types of CPU chips. The most commonly-known to the public is the Intel Pentium, but other very common ones are the PowerPC chip used in Macintosh computers and IBM UNIX workstations, the MIPS chip used in SGI workstations and so on.

We speak of the **architecture** of a CPU. This means its instruction set and registers, the latter being units of bit storage similar memory words but inside the CPU.

It is highly important that you keep at the forefront of your mind that the term **machine language** does not connote “yet another programming language” like C or C++; instead, it refers to code consisting of instructions for a specific CPU type. A program in Intel machine language will be rejected as nonsense if one attempts to run it on, say, a MIPS machine. For instance, on Intel machines, 011101011111000 means to jump back 8 bytes, while on MIPS it would either mean something completely different or would mean nothing at all.

1.3 The Intel Architecture

Intel CPUs can run in several different modes. On Linux the CPU runs in **protected, flat 32-bit mode**, or the same for 64 bits on machines with such CPUs. For our purposes here, we will not need to define the terms *protected* and *flat*, but do keep in mind that in this mode word size is 32 bits.

For convenience, we will assume from this point onward that we are discussing 32-bit CPUs.

Intel instructions are of variable length. Some are only one byte long, some two bytes and so on.

The main registers of interest to us here are named EAX, EBX, ECX, EDX, ESI, EDI, EBP and ESP.²

¹Slated to be replaced by Intel chips.

²We will use the all-caps notation, EAX, EBX, etc. to discuss registers in the text, even though in program code they appear as `%eax`, `%ebx`, ...

Similarly, we will use all-caps notation when referring to instruction families, such as the MOV family, even though in program

These are all 32-bit registers, but the registers EAX through EDX are also accessible in smaller portions. The less-significant 16 bit portion of EAX is called AX, and the high and low 8 bits of AX are called AH and AL, respectively. Similar statements hold for EBX, ECX and EDX.

Note that EBP and ESP may not be appropriate for general use in a given program. In particular, we should avoid use of ESP in programs with subroutines, for reasons to be explained in our unit on subroutines. If we are writing some assembly language which is to be combined with some C/C++ code, we won't be able to use EBP either.

Also, there is another register, the PC, which for Intel is named EIP (Extended Instruction Pointer). We can never use it to store data, since it is used to specify which instruction is currently executing. Similarly, the flags register EFLAGS, to be discussed below, is reserved as well.

2 What Is Assembly Language?

Machine-language consists of long strings of 0s and 1s. Programming at that level would be extremely tedious, so the idea of assembly language was invented. Just as hex notation is a set of abbreviations for various bit strings in general, assembly language is another set of abbreviations for various bit strings which show up in machine language.

For example, in Intel machine language, the bit string 01100110100111000011 codes the operation in which the contents of the AX register are copied to the BX register. Assembly language notation is much clearer:

```
mov %ax, %bx
```

The abbreviation “mov” stands for “move,” which actually means “copy.”

An **assembler** is a program which translates assembly language to machine language. Unix custom is that assembly-language file names end with a **.s** suffix. So, the assembler will input a file, say **x.s**, written by the programmer, and produce from it an **object file** named **x.o**. The latter consists of the compiled machine language, e.g. the bit string 0111010111111000 we mentioned earlier for a jump-back-8-bytes instruction. In the Windows world, assembly language source code file names end with **.asm**, from which the assembler produces machine code files with names ending with **.obj**.

You probably noticed that an assembler is similar to a compiler. This is true in some respects, but somewhat misleading. If we program in assembly language, we are specifying exactly what machine-language instructions we want, whereas if we program in, say, C/C++, we let the compiler choose what machine-language instructions to produce.

For instance, in the case of the assembly-language line above, we know for sure which machine instruction will be generated by the assembler—we know it will be a 16-bit MOV instruction, that the registers AX and BX will be used, etc. And we know a single machine instruction will be produced. By contrast, with the C statement

```
z = x + y;
```

we don't know which—or even how many—machine instructions will be generated by the compiler. In fact, different compilers might generate different sets of instructions.

code they appear as **%movl**, **%movb**, etc.

3 Different Assemblers

Our emphasis will be on the GNU assembler AS, whose executable file is named **as**. This is part of the GCC package. Its syntax is commonly referred to as the “AT&T syntax,” alluding to Unix’s AT&T Bell Labs origins.

However, we will also be occasionally referring to another commonly-used assembler, NASM. It uses Intel’s syntax, which is similar to that of **as** but does differ in some ways. For example, for two-operand instructions, **as** and NASM have us specify the operands in orders which are the reverse of each other, as you will see below.

It is very important to note, though, that the two assemblers will produce the same machine code. Unlike a compiler, whose output is unpredictable, we know ahead of time what machine code an assembler will produce, because the assembly-language **mnemonics** are merely handy abbreviations for specific machine-language bit fields.

Suppose for instance we wish to copy the contents of the AX register to the BX register. In **as** we would write

```
mov %ax,%bx
```

while in NASM it would be

```
mov bx,ax
```

but the same machine-language will be produced in both cases, 01100110100111000011, as mentioned earlier.

4 Sample Program

In this very simple example, we find the sum of the elements in a 4-word array, **x**.

```
1  # introductory example; finds the sum of the elements of an array
2
3  .data # start of data section
4
5  x:
6      .long 1
7      .long 5
8      .long 2
9      .long 18
10
11  sum:
12      .long 0
13
14  .text # start of code section
15
16  .globl _start
17  _start:
18      movl $4, %eax # EAX will serve as a counter for
19                    # the number of words left to be summed
20      movl $0, %ebx # EBX will store the sum
21      movl $x, %ecx # ECX will point to the current
22                    # element to be summed
```

```

23 top:  addl (%ecx), %ebx
24      addl $4, %ecx # move pointer to next element
25      decl %eax  # decrement counter
26      jnz top   # if counter not 0, then loop again
27 done: movl %ebx, sum # done, store result in "sum"

```

4.1 Analysis

A source file is divided into **data** and **text** sections, which contain data and machine instructions, respectively.³ Data consists of variables, as you are accustomed to using in C/C++.

First, we have the line

```
.data    # start of data section
```

The fact that this begins with ‘.’ signals the assembler that this will be a **directive** (also known as a **pseudoinstruction**), meaning a command to the assembler rather than something the assembler will translate into a machine instruction. It is rather analogous to a note one might put in the margin, say “Please double-space here,” of a handwritten draft to be given to a secretary for typing; one does NOT want the secretary to type “Please double-space here,” but one does want the secretary to take some action there.

This directive here is indicating that what follows will be data rather than code.

The # character means that it and the remainder of the line are to be treated as a comment.

Next,

```

x:
    .long 1
    .long 5
    .long 2
    .long 18

```

tells the assembler to make a note in **x.o** saying that later, when the operating system (OS) loads this program into memory for execution, the OS should set up four consecutive “long” areas in memory, set with initial contents 1, 5, 2 and 18 (decimal). “Long” means 32-bit size, so we are asking the assembler to arrange for four words to be set up in memory, with contents 1, 5, 2 and 18.⁴ Moreover, we are telling the assembler that in our assembly code below, the first of these four long words will be referred to as **x**. We say that **x** is a **label** for this word.⁵ Similarly, immediately following those four long words in memory will be a long word which we will refer to in our assembly code below as **sum**.

By the way, what if **x** had been an array of 1,000 long words instead of four, with all words to be initialized to, say, 8? Would we need 1,000 lines? No, we could do it this way:

³There may be other related sections as well. We will describe some of these in a later unit.

⁴The term **long** here is a historical vestige from the old days of 16-bit Intel CPUs. The modern 32-bit word size is “long” in comparison to the old 16-bit size. Note that in the Intel syntax the corresponding term is **double**.

⁵Note that **x** is simply a name for the first word in the array, not the set of 4 words. Knowing this, you should now have some insight into why in C or C++, an array name is synonymous with a pointer to the first element of the array.

Keep in mind that whenever we are referring to **x**, both here in the program and also when we run the program via a debugger (see below), **x** will never refer to the entire array; it simply is a name for that first word in the array. Remember, we are working at the machine level, and there is no such thing as a data type here, thus no such thing as an array! Arrays exist only in our imaginations.

```
x:
    .rept 1000
    .long 8
    .endr
```

The **.rept** directive tells the assembler to act as if the lines following **.rept**, up to the one just before **.endr**, are repeated the specified number of times.

What about an array of characters? We can ask the assembler to leave space for this using the **.space** directive, e.g. for a 6-character array:

```
y: .space 6 # reserve 6 bytes of space
```

Or if we wish to have that space initialized to some string:

```
y: .string "hello"
```

This will take up six bytes, including a null byte at the end; the developers of the **.string** directive decided on such a policy in order to be consistent with C. Note carefully, though, that they did NOT have to do this; there is nothing sacrosanct about having null bytes at the ends of character strings. ✓

Getting back to our example here, we next have a directive signalling the start of the **text** section, meaning actual program code. Look at the first two lines:

```
_start:
    movl $4, %eax
```

Here **_start** is another label, in this case for the location in memory at which execution of the program is to begin, called the **entry point**, in this case that **movl** instruction. We did not choose the name for this label arbitrarily, in contrast to all the others; the Unix linker takes this as the default.⁶

The **movl** instruction copies the constant 4 to the EAX register.⁷ You can infer from this example that AS denotes constants by dollar signs.

The ‘l’ in “movl” means “long.” The corresponding Intel syntax,

```
mov eax,4
```

has no such distinction, relying on the fact that EAX is a 32-bit register to implicitly give the same message to the assembler, i.e. to tell the assembler that we mean a 32-bit 4, not say, a 16-bit 4.

Keep in mind that **movl** is just one member of the move family of instructions. Later, for example, you will see another member of that family, **movb**, which does the same thing except that it copies a byte instead of a word.

As noted earlier, we will generally use all-caps notation to refer to instruction families, for example referring to MOV to any of the instructions **movl**, **movb**, etc.

⁶The previous directive, **.globl**, was needed for the linker too. More on this in our unit on subroutines.

⁷We will usually use the present tense in remarks like this, but it should be kept in mind that the action will not actually occur until the program is executed. So, a more precise though rather unwieldy phrasing would be, “When it is later executed, the **movl** instruction will copy...”

By the way, integer constants are taken to be base-10 by default. If you wish to state a constant in hex instead, use the C “0x” notation.⁸

The second instruction is similar, but there is something noteworthy in the third:

```
movl $x, %ecx
```

In the token \$4 in the first instruction, the dollar sign meant a constant, and the same is true for \$x. The constant here is the address of **x**. Thus the instruction places the address of **x** in the ECX register, so that EAX serves as a pointer. A later instruction,

```
addl $4, %ecx
```

increments that pointer by 4 bytes, i.e. 1 word, each time we go around the loop, so that we eventually have the sum of all the words.

Note that \$x has a completely different meaning than **x** by itself. The instruction

```
movl x, %ecx
```

would copy the contents of the memory location **x**, rather than its address, to ECX.⁹

The next line begins the loop:

```
top: addl (%ecx), %ebx
```

Here we have another label, **top**, a name which we’ve chosen to remind us that this is the top of the loop. This instruction takes the word pointed to by ECX and adds it to EBX. The latter is where I am keeping the total.

Recall that eventually we will copy the final sum to the memory location labeled **sum**. **We don’t want to do so within the loop, though, because memory access is much slower than register access (since we must leave the CPU to go to memory), and we thus want to avoid it. So, we keep our sum in a register, and copy to memory only when we are done.**¹⁰

If we were not worried about memory access speed, we might store directly to the variable **sum**, as follows:

```
movl $sum,%edx # use %edx as a pointer to "sum"
movl $0,%ebx
top: addl (%ecx), %ebx # old sum is still in %ebx
     movl %ebx, (%edx)
```

Note carefully that we could NOT write

```
movl $sum,%edx # use %edx as a pointer to "sum"
top: addl (%ecx), (%edx)
```

because there is no such instruction in the Intel architecture. Intel chips (like most CPUs) do not allow an instruction to have both its operands in memory.¹¹ **Note that this is a constraint placed on us by the hardware, not by the assembler.**

⁸Note, though, that the same issues of endian-ness will apply in using the assembler as those related to the C compiler.

⁹The Intel syntax is quite different. Under that syntax, **x** would mean the address of **x**, and the contents of the word **x** would be denoted as **[x]**.

¹⁰This presumes that we need it in memory for some other reason. If not, we would not do so.

¹¹There are actually a couple of exceptions to this on Intel chips, as will be seen in Section A.

The bottom part of the loop is:

```
decl %eax
jnz top
```

The DEC instruction, **decl** (“decrement long”), in AT&T syntax, subtracts 1 from EAX. This instruction, together with the JNZ following it, provides our first illustration of the operation of the EFLAGS register in the CPU:

When the hardware does almost any arithmetic operation, it also records whether the result of this instruction is 0: It sets Zero flag, ZF, in the EFLAGS register to 1 if the result of the operation was 0, and sets that flag to 0 if not.¹² Similarly, the hardware sets the Sign flag to 1 or 0, according to whether the result of the subtraction was negative or not.

Most arithmetic operations do affect the flags, but for instance MOV does not. For a given instruction, you can check by writing a short test program, or look it up in the official Intel CPU manual.¹³

Now, here is how we make use of the Zero flag. The JNZ (“jump if not zero”) instruction says, “If the result of the last arithmetic operation was not 0, then jump to the instruction labeled **top**.” The circuitry for JNZ implements this by jumping if the Zero flag is 1. (The complementary instruction, JZ, jumps if the Zero flag is 0, i.e. if the result of the last instruction was zero.)

So, the net effect is that we will go around the loop four times, until EAX reaches 0, then exit the loop (where “exiting” the loop merely means going to the next instruction, rather than jumping to the line labeled **top**).

Even though our example jump here is backwards—i.e. to a lower-addressed memory location—forward jumps are just as common. The reader should think about why forward jumps occur often in “if-then-else” situations, for example.

By the way, in computer architecture terminology, the word “branch” is a synonym for “jump.” In many architectures, the names of the jump instructions begin with ‘B’ for this reason.

The EFLAGS register is 32 bits wide (numbered 31, the most significant, to 0, the least significant), like the other registers. Here are some of the flag and enable bits it includes:¹⁴

Data	Position
Overflow Flag	Bit 11
Interrupt Enable	Bit 9
Sign Flag	Bit 7
Zero Flag	Bit 6
Carry Flag	Bit 0

So, for example there are instructions like JC (“jump if carry”), JNC and so on which jump if the Carry Flag is set.

Note that the label **done** was my choice, not a requirement of **as**, and I didn’t need a label for that line at all,

¹²The reason why the hardware designers chose 1 and 0 as codes this way is that they want us to think of 1 as meaning yes and 0 as meaning no. So, if the Zero flag is 1, the interpretation is “Yes, the result was zero.”

¹³This is on the Intel Web site, but also available on our class Web page at <http://heather.cs.ucdavis.edu/~matloff/50/IntelManual.PDF>. Go to Index, then look up the page number for your instruction. Once you reach the instruction, look under the subheading Flags Affected.

¹⁴The meanings of these should be intuitive, except for the Interrupt Enable bit, which will be described in our unit on input/output.

since it is not referenced elsewhere in the program. I included it only for the purpose of debugging, as seen later.

Just as there is a DEC instruction for decrementing, there is INC for incrementing.

4.2 Source and Destination Operands

In a two-operand instruction, the operand which changes is called the **destination** operand, and the other is called the **source** operand. For example, in the instruction

```
movl %eax, %ebx
```

EAX is the source and EBX is the destination.

4.3 Remember: No Names, No Types at the Machine Level

Keep in mind that, just as our variable names in a C/C++ source file do not appear in the compiled machine language, in an assembly language source file labels—in this case, **x**, **sum**, **_start**, **top** and **done**—are just temporary conveniences for us humans. We use them only in order to conveniently refer AS to certain locations in memory, in both the **.data** and **.text** sections. These labels do NOT appear in the machine language produced by AS; only numeric memory addresses will appear there.

For instance, the instruction

```
jnz top
```

mentions the label **top** here in assembly language, but the actual machine language which comes from this is 011101011111000. You will find in our unit on machine language that the first 8 bits, 01110101, code a jump-if-not-zero operation, and the second 8 bits, 11111000, code that the jump target is 8 bytes backward. Don't worry about those codes for now, but the point at hand now is that neither of those bit strings makes any mention of **top**.

Again, there are no types at the machine level. So for example there would be no difference between writing

```
z:
    .long 0
w:
    .byte 0
```

and

```
z:
    .rept 5
    .byte 0
    .endr
```

Both of these would simply tell AS to arrange for 5 bytes of (zeroed-out) space at that point in the data section. Make sure to avoid the temptation of viewing the first version as “declaring” an integer variable **z** and a character variable **w**. True, the programmer may be interpreting things that way, but the two versions would produce exactly the same **.o** file.



4.4 Dynamic Memory Is Just an Illusion

One thing to note about our sample program above is that all memory was allocated statically, i.e. at assembly time.¹⁵ So, we have statically allocated five words in the **.data** section, and a certain number of bytes in the **.text** section (the number of which you'll see in our unit on machine language).

Since C/C++ programs are compiled into machine language, and since assembly language is really just machine language, you might wonder how it can be that memory can be dynamically allocated in C/C++ yet not in assembly language. The answer to this question is that one cannot truly do dynamic allocation in C/C++ either. Here is what occurs:

Suppose you call **malloc()** in C or invoke **new** in C++. The latter, at least for G++, in turn calls **malloc()**, so let's focus on that function. When you run a program whose source code is in C/C++, some memory is set up called the **heap**. Any call to **malloc()** returns a pointer to some memory in the heap; **malloc()**'s internal data structures then record that that portion of memory is in use. Later, if the program calls **free()**, those data structures now mark the area as available for use.

In other words, the heap memory itself was in fact allocated when the program was loaded, i.e. statically. However, **malloc()** and its sister functions such as **free()** manage that memory, recording what is available for use now and what isn't. So the notion of dynamic memory allocation is actually an illusion.

An assembly language programmer could link the C library into his/her program, and thus be able to call **malloc()** if that were useful. But again, it would not actually be dynamic allocation, just like it is not in the C/C++ case.

5 Use of Registers Versus Memory

Recall that registers are located inside the CPU, whereas memory is outside it. Thus, register access is much faster than memory access.

Accordingly, when you do assembly language programming, you should try to minimize your usage of memory, i.e. of items in the **.data** section (and later, of items on the stack), especially if your goal is program speed, which is a common reason for resorting to assembly language.

However, most CPU architectures have only a few registers. Thus in some cases you may run out of registers, and need to store at least some items in memory.¹⁶

6 Another Example

The following example does a type of sort. See the comments for an outline of the algorithm. (This program is merely intended as an example for learning assembly language, not for algorithmic efficiency.)

One of the main new ideas here is that of a **subroutine**, which is similar to the concept of a function in C/C++.¹⁷ In this program, we have subroutines **init()**, **findmin()** and **swap()**.

¹⁵Again, be careful here. The memory is not actually assigned to the program until the program is actually run. Our word *allocated* here refers to the fact that the assembler had already planned the memory usage.

¹⁶Recall that a list of usable registers for Intel was given in Section 1.3.

¹⁷In fact, the compiler translates C/C++ functions to subroutine at the machine-language level.

```

1  # sample program; does a (not very efficient) sort of the array x, using
2  # the algorithm (expressed in pseudo-C code notation):
3
4  # for each element x[i]
5  #   find the smallest element x[j] among x[i+1], x[i+2], ...
6  #   if new min found, swap x[i] and x[j]
7
8  .equ xlength, 7  # number of elements to be sorted
9
10 .data
11 x:
12     .long    1
13     .long    5
14     .long    2
15     .long    18
16     .long    25
17     .long    22
18     .long    4
19
20 .text
21     # register usage in "main()":
22     #   EAX points to next place in sorted array to be determined,
23     #   i.e. "x[i]"
24     #   ECX is "x[i]"
25     #   EBX is our loop counter (number of remaining iterations)
26     #   ESI points to the smallest element found via findmin
27     #   EDI contains the value of that element
28 .globl _start
29 _start:
30     call init  # initialize needed registers
31 top:
32     movl (%eax), %ecx
33     call findmin
34     # need to swap?
35     cmpl %ecx, %edi
36     jge nexti
37     call swap
38 nexti:
39     decl %ebx
40     jz done
41     addl $4, %eax
42     jmp top
43
44 done: movl %eax, %eax  # dummy, just for running in debugger
45
46 init:
47     # initialize EAX to point to "x[0]"
48     movl $x, %eax
49     # we will have xlength-1 iterations
50     movl $xlength, %ebx
51     decl %ebx
52     ret
53
54 findmin:
55     # does the operation described in our pseudocode above:
56     #   find the smallest element x[j], j = i+1, i+2, ...
57     # register usage:
58     #   EDX points to the current element to be compared, i.e. "x[j]"
59     #   EBP serves as our loop counter (number of remaining iterations)
60     #   EDI contains the smallest value found so far
61     #   ESI contains the address of the smallest value found so far
62     # haven't started yet, so set min found so far to "infinity" (taken
63     # here to be 999999; for simplicity, assume all elements will be
64     # <= 999999)
65     movl $999999, %edi
66     # start EDX at "x[i+1]"
67     movl %eax, %edx
68     addl $4, %edx

```

```

69      # initialize our loop counter (nice coincidence:  number of
70      #      iterations here = number of iterations remaining in "main()")
71      movl %ebx, %ebp
72      # start of loop
73 findminloop:
74      # is this "x[j]" smaller than the smallest we've seen so far?
75      cmpl (%edx), %edi  # compute destination - source, set EFLAGS
76      js nextj
77      # we've found a new minimum, so update EDI and ESI
78      movl (%edx), %edi
79      movl %edx, %esi
80 nextj:  # do next value of "j" in the loop in the pseudocode
81      # if done with loop, leave it
82      decl %ebp
83      jz donefindmin
84      # point EDX to the new "x[j]"
85      addl $4, %edx
86      jmp findminloop
87 donefindmin:
88      ret
89
90 swap:
91      # copy "x[j]" to "x[i]"
92      movl %edi, (%eax)
93      # copy "x[i]" to "x[j]"
94      movl %ecx, (%esi)
95      ret

```

Note that there are several new instructions used here, as well as a new pseudoinstruction, **.equ**.

Let's deal with the latter first:

```
.equ xlength, 7
```

This tells the assembler that, in every line in which it sees **xlength**, the assembler should assemble that line as if we had typed 7 there instead of **xlength**. In other words **.equ** works like **#define** in C/C++. Note carefully that **xlength** is definitely not the same as a label in the **.data** section, which is the name we've given to some memory location; in other words, **xlength** is not the name of a memory location.

At the beginning of the **.text** section, we see the instruction

```
call init
```

The CALL instruction is a type of jump, with the extra feature that it records the place the jump was made from. That record is made on the **stack**, which we will discuss in detail in our unit on subroutines. Here we will jump to the instruction labeled **init** further down in the source file:

```

init:
    movl $x, %eax
    movl $xlength, %ebx
    decl %ebx
    ret

```

After the CALL instruction brings us to **init**, the instructions there, i.e.

```

    movl $x, %eax
    ...

```

will be executed, just as we’ve seen before. But the RET (“return”) instruction **ret** is another kind of jump; it jumps back to the instruction immediately following the place at which the call to **init** was made. Recall that that place had been recorded at the time of the call, so the machine does know it, by looking at the stack. Keep in mind that execution of a RET will result in a return to the instruction *following* the CALL. That instruction is the one labeled **top**:

```
top:
    mov (%eax), %ecx
```

Again, you will find out how all this works later, in our unit on subroutines. But it is being introduced here, to encourage you to start using subroutines from the beginning of your learning of assembly language programming. It facilitates a top-down approach. (See Appendix C.)

As always, remember that the CPU is just a “dumb machine.” Suppose for example that we accidentally put a RET instruction somewhere in our code where we don’t intend to have one. If execution reaches that line, the RET will indeed be executed, no questions asked. The CPU has no way of knowing that the RET shouldn’t be there. It does not know that we are actually not in the midst of a subroutine.

Also, though you might think that the CPU will balk when it tries to execute the RET but finds no record of call location on the stack, the fact is that there is always *something* on the stack even if it is garbage. So, the CPU will return to a garbage point. This is likely to cause various problems, for instance possibly a seg fault, but the point is that the CPU will definitely **not** say, “Whoa, I refuse to do this RET.”

For that matter, even the assembler would not balk at our accidentally putting a RET instruction at some random place in our code. Remember, the assembler is just a clerk, so for example it does not care that that RET was not paired with a CALL instruction.

Note by the way that the code beginning at **__start** is analogous to **main()** in C/C++, and the comments in the code use this metaphor.

The **init()** subroutine does what it says, i.e. initialize the various registers to the desired values.

The next instruction is another subroutine call, to **findmin()**. As described in the pseudocode, it finds the smallest element in the remaining portion of the array. Let’s not go into the details of how it does this yet—not only should one *write* code in a top-down manner, but one should also *read* code that way. We then check to see whether a swap should be done, and if so, we do it.

Another new instruction is CMP (“compare”), **cmpl**, which is used within **findmin()**. As noted in the comment in the code, this instruction subtracts the source from the destination; the result, i.e. the difference, is not stored anywhere, but the key point is that the EFLAGS register is affected.

Since there is an instruction for addition, it shouldn’t be a surprise to know there is one for subtraction too. For example,

```
subl %ebx, %eax
```

subtracts c(EBX) from c(EAX), and places the difference back into EAX. This instruction does the same thing as CMP, except that the latter does not store the difference back anyway; the computation for the **cmpl** is done only for the purpose of setting the flags.

Following the CMP instruction is JS (“jump if the Sign flag is 1), meaning that we jump if the result of the last arithmetic computation was “signed,” i.e. negative. (The complementary instruction, JNS, jumps if the result of the last computation was not negative.)

In other words, the combined effect of the `CMP` and `JS` instructions here is that we jump to **nextj** if the contents of `EDI` is less than that of the memory word pointed to by `EDX`. In this case, we have not found a new minimum, so we just go to the next iteration of the loop.

The instruction

```
jmp top
```

is conceptually new. All the jump instructions we've seen so far have been **conditional**, i.e. the jump is made only if a certain condition holds. But `JMP` means to jump unconditionally.

Note my comments on the usage I intend for the various registers, e.g. in “`main()`”:

```
# register usage in "main()":
#   EAX points to next place in sorted array to be determined,
#       i.e. "x[i]"
#   ECX is "x[i]"
#   EBX is our loop counter (number of remaining iterations)
#   ESI points to the smallest element found via findmin
#   EDI contains the value of that element
```

I put these in **BEFORE** I started writing the program, to help keep myself organized, and I found myself repeatedly referring to them during the writing process. **YOU SHOULD DO THE SAME.**

Also note again that we absolutely needed to avoid using `ESP` as storage here, due to the fact that we are using **call** and **ret**. The reason for this will be explained in our unit on subroutines.

7 Addressing Modes

Consider the examples

```
movl $9, %ebx
movl $9, (%ebx)
movl $9, x
```

As you can see, the circuitry which implements **movl** allows for several different versions of the instruction. In the first example above, we copy 9 to a register. In the second and third examples, we copy to places in memory, but even then, we do it via different ways of specifying the memory location—using a register as a pointer to memory in the second example, versus directly stating the given memory location in the third example.

The manner in which an instruction specifies an operand is called the **addressing mode** for that operand. So, above we see three distinct addressing modes for the second operand of the instruction.

For example, let's look at the instruction

```
movl $9, %ebx
```

Let's look at the destination operand first. Since the operand is in a register, we say that this operand is expressed in **register mode**. We say that the source operand is accessed in **immediate** mode, which means

that the operand, in this case the number 9, is right there (i.e. “immediately within”) in the instruction itself.¹⁸

Now consider the instruction

```
movl $9, (%ebx)
```

Here the destination operand is the memory word pointed to by EBX. This is called **indirect mode**; we “indirectly” state the location, by saying, “It’s whatever EBX points to.”

By contrast, the destination operand in

```
movl $9, x
```

is accessed in **direct mode**; we have directly stated where in memory the operand is.

Other addressing modes will be discussed in Section 12.

8 Assembling and Linking into an Executable File

8.1 Assembler Command-Line Syntax

To assemble an AT&T-syntax source file, say **x.s**, we will type

```
as -a --gstabs -o x.o x.s
```

The **-o** option specifies what to call the object file, the machine-language file produced by the assembler. Here we are telling the assembler, “Please call our object file **x.o**.”¹⁹

The **-a** option tells the assembler to display to the screen the source (i.e. assembly) code, machine code and section offsets²⁰ side-by-side, for easier viewing by us humans.

The **--gstabs** option (note that there are two hyphens, not one) tells the assembler to retain in **x.o** the **symbol table**, a list of the locations of whatever labels are in **x.s**, in a form usable by symbolic debuggers, in our case GDB or DDD.²¹

Things are similar under other operating systems. The Microsoft assembler, MASM, has similar command-line options, though of course with different names and some difference in functionality.²²

¹⁸You’ll see this more explicitly when we discuss machine language.

¹⁹**Important note on disaster avoidance:** Suppose we are assembling more than one file, say

```
as -a --gstabs -o z.o x.s y.s
```

Suppose we inadvertently forgot to include the **z.o** on the command line here. Then we would be telling the assembler, “Please call our object file **x.s**.” This would result in the assembler overwriting **x.s** with the object file, trashing **x.s**! Be careful to avoid this.

²⁰These are essentially addresses, as will be explained later.

²¹The **--gstabs** option is similar to **-g** when running GCC.

²²By the way, NASM is available for both Unix and MS Windows. For that matter, even AS can be used under Windows, since it is part of the GCC package that is available for Windows under the name Cygwin.

8.2 Linking

Say we have an assembly language source file **x.s**, and then assemble it to produce an object file **x.o**. We would then type, say,

```
ld -o x x.o
```

Here **ld** is the linker, LD, which would take our object file **x.o** and produce an executable file **x**.

We will discuss more on linking in Section 13.2.

8.3 Makefiles

By the way, you can automate the assembly and linkage process using Makefiles, just as you would for C/C++. Keep in mind that makefiles have nothing to do with C or C++. They simply state which files depend on which other files, and how to generate the former files from the latter files.

So for example the form

```
x: y
<TAB> z
```

simply says, “The file **x** depends on **y**. If we need to make **x** (or re-make it if we have changed **y**), then we do **z**.”

So, for our source file **x.s** above, our Makefile might look like this:

```
x: x.o
<TAB> ld -o x x.o

x.o: x.s
<TAB> as --gstabs -o x.o x.s
```

9 How to Execute Those Sample Programs

9.1 “Normal” Execution Won’t Work

Suppose in our sum-up-4-words example above we name the source file **Total.s**, and then assemble and link it, with the final executable file named, say, **tot**. We could not simply type

```
% tot
```

at the Unix command line. The program would crash with a segmentation fault. Why is this?

The basic problem is that after the last instruction of the program is executed, the processor will attempt to execute the “instruction” at the next location of memory. There is no such instruction, but the CPU won’t know that. All the CPU knows is to keep executing instructions, one after the other. So, when your program marches right past its last real instruction, the CPU will try to execute the garbage there. I call this “going off the end of the earth.”

Actually, in Linux the linker will arrange for our **.data** section to follow our **.text** section in memory, almost immediately after the end of the latter. It's "almost" because we would like the **.data** section to begin on a word boundary, i.e. at an address which is a multiple of 4. The area in between is padding, consisting of 0s, in this case three bytes of 0s.²³ Thus the "garbage" which we execute when we "go off the end of the earth" is our own data!²⁴



This doesn't happen with your compiled C/C++ program, because the compiler inserts a **system call**, i.e. a call to a function in the operating system, which in this case is the **exit()** call. (Actually, good programming practice would be to insert this call in your C/C++ programs yourself.) This results in a graceful transition from your program to the OS, after which the OS prints out your familiar command-line prompt.

We could have inserted system calls in our sample assembly language programs above too, but did not do so because that is a topic to be covered later in the course. Note that that also means we cannot do input and output, which is done via system calls too—so, not only does our program crash if we run it in the straightforward manner above, but also we have no way of knowing whether it ran correctly before crashing!

So, in our initial learning environment here, we will execute our programs via a debugger, either DDD or GDB, which will allow us to have the program stop when it is done and to see the results.

9.2 Running Our Assembly Programs Using GDB/DDD

Since a debugger allows us to set breakpoints or single-step through programs, we won't "go off the end of the earth" and cause a seg fault as we would by running our programs directly.²⁵

Moreover, since the debuggers allow us to inspect registers and memory contents, we can check the "output" of our program. In our first array-summing program in Section 4, for example, the sum was in EBX, so the debugger would enable us to check the program's operation by checking whether EBX contained the correct sum.

9.2.1 Using DDD for Executing Our Assembly Programs

Starting the Debugger:

For our array-summing example, we would start by assembling and linking the source code, and then typing

```
% ddd tot
```

Your source file **Total.s** should appear in the DDD Source Window.

Making Sure You Don't "Go Off the End of the World":

You would set a breakpoint at the line labeled **done** by clicking on that line and then on the red stop sign icon at the top of the window. This arranges for your program to stop when it is done.

Running the Program:

You would then run the program by clicking on Run. The program would stop at **done**.

²³This can be determined using information presented in our unit on machine language.

²⁴Possibly preceded by 1-3 0 bytes.

²⁵Keep in mind, that if we don't set a breakpoint when we run a program within the debugger, we will still "go off the end of the earth."

Checking the “Output” of the Program:

You may have written your program so that its “output” is in one or more registers. If so, you can inspect the register contents when you reach **done**. For example, recall that in the **tot** program, the final sum (26) will be stored in the EBX register, so you would inspect the contents of this register in order to check whether the program ran correctly.

To do this, click on Status then Registers.

On the other hand, our program’s output may be in memory, as is the case for instance for the program in Section 6. Here we check output by inspecting memory, as follows:

Hit Data, then Memory. An Examine Memory window will pop up, asking you to state which data items you wish to inspect:

- Fill in the blank on the left with the number of items you want to inspect.
- In the second field, labeled “octal” by default, state how you want the contents of the items described—in decimal, hex or whatever.
- In the third field, state the size of each item—byte, word or whatever.
- In the last field, give the address of the first item.

For the purpose of merely checking a program’s output we would choose Print here rather than Display. The former shows the items just once, while the latter does so continuously; the latter is useful for actual debugging of the program, while the former is all we need for merely checking the program’s output.

For instance, again consider the example in Section 6. We wish to inspect all seven array elements, so we fill in the Examine Memory window to indicate that we wish to Print 7 Decimal Words starting at **&x**.

Note on endian-ness, etc.: If you ask the debugger to show a word-sized item (i.e. a memory word or a full register), it will be shown most-significant byte first. Within a byte, the most-significant bit will be shown first.

9.2.2 Using GDB for Executing Our Assembly Programs

In some cases, you might find it more convenient to use GDB directly, rather than via the DDD interface. For example, you might be using **telnet**.

(Note: It is assumed here that you have already read the material on using DDD above.)

Starting the Debugger:

For the **tot** program example here, you would start by assembling and linking, and then typing

```
gdb tot
```

Making Sure You Don’t “Go Off the End of the World”:

Set the breakpoint at **done**:

```
(gdb) b done
Breakpoint 1 at 0x804808b: file sum.s, line 28.
```

Running the Program:

Issue the **r** (“run”) command to GDB.

Checking the “Output” of the Program:

To check the value of a register, e.g. EBX, use the **info registers** command:

```
(gdb) info registers ebx
ebx                0x1a    26
```

To check the value of a memory location, use the **x** (“examine”) command. In the example in Section 6, for instance:

```
x/7w &x
```

This says to inspect 7 words, beginning at **x**, printing out the contents in hex. In some cases, e.g. if you are working with code translated from C to assembly language, GDB will switch to printing individual bytes, in which case use, e.g., **x/7x** instead of **x/7w**.

There are other ways to print out the contents, e.g.

```
x/12c &x
```

would treat the 12 bytes starting at **x** as characters and print them out.

10 How to Debug Assembly Language Programs

10.1 Use a Debugging Tool for ALL of Your Programming, in EVERY Class

I’ve found that many students are **shooting themselves in the foot** by not making use of debugging tools. They learn such a tool in their beginning programming class, but treat it as something that was only to be learned for the final exam, rather than for their own benefit. Subsequently they debug their programs with calls to `printf()` or `cout`, which is really a slow, painful way to debug. **You should make use of a debugging tool in all of your programming work – for your benefit, not your professors’.** (See my debugging-tutorial slide show, at <http://heather.cs.ucdavis.edu/~matloff/debug.html>.)

For C/C++ programming on Unix machines, many debugging tools exist, some of them commercial products, but the most commonly-used one is GDB. Actually, many people use GDB only indirectly, using DDD as their interface to GDB; DDD provides a very nice GUI to GDB.

10.2 General Principles

10.2.1 The Principle of Confirmation

Remember (as emphasized in the debugging slide show cited above) **the central principle of debugging is confirmation**. We need to step through our program, at each step confirming that the various registers and memory locations contain what we think they ought to contain, and confirming that jumps occur when we think they ought to occur, and so on. Eventually we will reach a place where something fails to be confirmed, and then that will give us a big hint as to where our bug is.

10.2.2 Don't Just Write Top-Down, But Debug That Way Too

Consider code like, say,

```
movl $12, %eax
call xyz
addl %ebx, %ecx
```

When you are single-stepping through your program with the debugging tool and reach the line with the **call** instruction, the tool will give you a choice of going in to the subroutine (e.g. the Step command in DDD) or skipping over it (the Next command in DDD). Choose the latter at first. When you get to the next line (with **addl**), you can check whether the “output” of the subroutine **xyz()** is correct; if so, you will have saved a lot of time and distraction by skipping the detailed line-by-line execution of **xyz()**. If on the other hand you find that the output of the subroutine is wrong, then you have narrowed down the location of your bug; you can then re-run the program, but in this case opt for Step instead of Next when you get to the call.

10.3 Assembly Language-Specific Tips

10.3.1 Know Where Your Data Is

The first thing you should do during a debugging session is write down the addresses of the labeled items in your **.data** section. And then use those addresses to help you debug.

Consider the program in Section 6, which included a data label **x**. We should first determine where **x** is. We can do this in GDB as follows:

```
(gdb) p/x &x
```

In DDD, we might as well do the same thing, issuing a command directly to GDB via DDD's Console window.

Knowing these addresses is extremely important to the debugging process. Again using the program in Section 6 for illustration, on the line labeled **top** the EAX register is serving as a pointer to our current position in **x**. In order to verify that it is doing so, we need to know the address of **x**.

10.3.2 Seg Faults

Many bugs in assembly language programs lead to seg faults. These typically occur because the programmer has inadvertently used the wrong addressing mode or the wrong register.

For example, consider the instruction

```
movl $39, %edx
```

which copies the number 39 to the EDX register.

Suppose we accidentally writes

```
movl $39, (%edx)
```

This copies 39 to the memory location pointed to by EDX. If EDX contains, say, 12, then the CPU will attempt to copy 39 to memory location 12, which probably will not be in the memory space allocated to this program when the OS loaded it into memory. In other words, we used the wrong addressing mode, which will cause a seg fault.

When we run the program in the debugger, the latter will tell us exactly where—i.e. at what instruction—the seg fault occurred. This is of enormous value, as it pinpoints exactly where our bug is. Of course, the debugger won't tell us, "You dummy, you used the wrong addressing mode"—the programmer then must determine why that particular instruction caused the seg fault—but at least the debugger tells us where the fault occurred.

Suppose on the other hand, we really did need to use indirect addressing mode, but accidentally specified ECX instead of EDX. In other words, we intended to write

```
movl $39, (%edx)
```

but accidentally wrote

```
movl $39, (%ecx)
```

Say $c(EDX) = 0x10402226$ but $c(ECX) = 12$,²⁶ and that $0x10402226$ is in a part of memory which was allocated to our program but 12 is not. Again we will get a seg fault, as above. Again, the debugger won't tell us, "You dummy, you should have written `"%edx"`, not `"%ecx"`", but at least it will pinpoint which instruction caused the seg fault, and we then can think about what we might have done wrong in that particular instruction.

10.4 Use of DDD for Debugging Assembly Programs

To examine register contents, hit Status and then Registers. All register contents will be displayed.

The display includes EFLAGS, the flags register. The Carry Flag is bit 0, i.e. the least-significant bit. The other bits we've studied are the Zero Flag, bit 6, the Sign Flag, bit 7, and the Overflow Flag, bit 11.

To display, say, an array **z**, hit Data, then Memory. State how many cells you want displayed, what kind of cells (whole words, individual bytes, etc.), and where to start, such as **&z**. It will also ask whether you want the information "printed," which means displayed just once, or "displayed," which means a continuing display which reflects the changes as the program progresses.

Breakpoints may be set and cleared using the stop sign icons.²⁷

You can step through your code line by line in the usual debugging-tool manner. However, in assembly language, make sure that you use StepI instead of Next or Step, since we are working at the machine instruction level (the 'i' stands for "instruction.")

Note that when you modify your assembly-language source file and reassemble and link, DDD will not automatically reload your source and executable files. To reload, click on File in DDD, then Open Program and click on the executable file.

²⁶Recall that $c()$ means "contents of."

²⁷For some reason, it will not work if we set a breakpoint at the very first instruction of a program, though any other instruction works.

10.5 Use of GDB for Debugging Assembly Programs

10.5.1 Assembly-Language Commands

Assuming you already know GDB (see the link to my Web tutorial), here are the two new commands you should learn.

- To view all register contents, type

```
info registers
```

You can view specific registers with the **p** (“print”) command, e.g.

```
p/x $pc
p/x $esp
p/x $eax
```

- To view memory, use the **x** (“examine”) command. If for example you have a memory location labeled **z** and wish to examine the first four words starting at a data-section label **z**, type

```
x/4w &z
```

Do not include the ampersand in the case of a text-section label. Note that the **x** command differs greatly from the **p** command, in that the latter prints out the contents of only one word.

Note too that you can do indirection. For example

```
x/4w $ebx
```

would display the four words of memory beginning at the word pointed to by EBX.

- As in the DDD case, use the StepI mode of single-stepping through code;²⁸ the command is

```
(gdb) stepi
```

or just

```
(gdb) si
```

Unlike DDD, GDB automatically reloads the program’s executable file when you change the source.

An obvious drawback of GDB is the amount of typing required. But this can be greatly mitigated by using the “define” command, which allows one to make abbreviations. For example, we can shorten the typing needed to print the contents of EAX as follows:

```
(gdb) define pa
Type commands for definition of "pa".
End with a line saying just "end".
>p/x $eax
>end
```

²⁸The NextI mode is apparently unreliable. Of course, you can still hop through the code using breakpoints.

From then on, whenever we type **pa** in this **gdb** session, the contents of EAX will be printed out.

Moreover, if we want these abbreviations to carry over from one session to another for this program, we can put them in the file **.gdbinit** in the directory containing the program, e.g. by placing these lines

```
define pa
p/x $eax
end
```

in **.gdbinit**, **pa** will automatically be defined in each debugging session for this program.

Use **gdb**'s online help facility to get further details; just type "help" at the prompt.

10.5.2 TUI Mode

As mentioned earlier, it is much preferable to use a GUI for debugging, and thus the DDD interface to GDB is highly recommended. As a middle ground, though, you may try GDB's new TUI mode. You will need a relatively newer version of GDB for this, and it will need to have been built to include TUI.²⁹

TUI may be invoked with the **-tui** option on the GDB command line. While running GDB, you toggle TUI mode on or off using **ctrl-x a**.

If your source file is purely in assembly language, i.e. you have no **main()**, first issue GDB's **l** ("list") command, and hit Enter an extra time or two. That will make the source-code subwindow appear.

Then, say, set a breakpoint and issue the **r** ("run") command to GDB as usual.

In the subwindow, breakpoints will be marked with asterisks, and your current instruction will be indicated by a **>** sign.

In addition to displaying a source code subwindow, TUI will also display a register subwindow if you type

```
(gdb) layout reg
```

This way you can watch the register values and the source code at the same time. TUI even highlights a register when it changes values.

Of course, since TUI just adds an interface to GDB, you can use all the GDB commands with TUI.

10.5.3 CGDB

Recall that the goal of TUI in our last subsection is to get some of the functionality of a GUI like DDD while staying within the text-only realm. If you are simply Telnetting into the machine where you are debugging a program, TUI is a big improvement over ordinary GDB. CGDB is another effort in this direction.

Whereas TUI is an integral part of GDB, CGDB is a separate front end to GDB, not developed by the GDB team. (Recall that DDD is also like this, but as a GUI rather than being text-based.) You can download it from <http://cgdb.sourceforge.net/>.

²⁹If your present version of GDB does not include TUI (i.e. GDB fails when you invoke it with the **-tui** option), you can build your own version of GDB. Download it from www.gnu.org, run **configure** with the option **--enable-tui**, etc.

Like TUI, CGDB will break the original GDB window into several subwindows, one of which is for GDB prompts and the other for viewing the debuggee’s source code. CGDB goes a bit further, by allowing easy navigation through the source-code subwindow, and by using a nice colorful interface.

To get into the source-code subwindow, hit Esc. You can then move through that subwindow using the **vi**-like commands, e.g. **j** and **k** to move down or up a line, **/** to search for text, etc.

To set a breakpoint at the line currently highlighted by the cursor, just hit the space bar. Breakpoints are highlighted in red,³⁰ and the current instruction in green.

Use the **i** command to get to the GDB command subwindow.

CGDB’s startup file is **cgdbrc** in a directory named **.cgdb** in your home directory. One setting you should make sure to have there is

```
set autosourcereload
```

which will have CGDB automatically update your source window when you recompile.

11 Some More Operand Sizes

Recall the following instruction from our example above:

```
addl (%ecx), %ebx
```

How would this change if we had been storing our numbers in 16-bit memory chunks?

In order to do that, we would probably find it convenient³¹ to use **.word** instead of **.long** for initialization in the **.data** section. Also, the above instruction would become

```
addw (%ecx), %bx
```

with the **w** meaning “word,” an allusion to the fact that earlier Intel chips had word size as 16 bits.

The changes here are self-explanatory, but the non-change may seem odd at first: Why are we still using ECX, not CX? The answer is that even though we are accessing a 16-bit item, its address is still 32 bits.

The corresponding items for 8-bit operations are **.byte** in place of **.long**, **movb** instead of **movl**, **%ah** or **%al** (high and low bytes of AX) in place of EAX, etc.

If you wish to have conditional jumps based on 16-bit or 8-bit quantities, be sure to use **cmpw** or **cmpb**, respectively.

If the destination operand for a byte instruction is of word size, the CPU will allow us to “expand” the source byte to a word. For example,

```
movb $-1, %eax
```

³⁰When you build CGDB, make sure you do **make install**, not just **make**. As of this early version of CGDB, in March 2003, this feature does not seem to work for assembly-language source code.

³¹Though not mandatory; recall Section 4.3.

will take the byte -1, i.e. 11111111, and convert it to the word -1, i.e. 11111111111111111111111111111111 which it will place in EAX. Note that the sign bit has been extended here, an operation known as **sign extension**.

By the way, though, in this situation the assembler will also give you a warning message, to make sure you really do want to have a word-size destination operand for your byte instruction. And the assembler will give an error message if you write something like, say,

```
movb %eax, %ebx
```

with its source operand being word-sized. The assembler has no choice but to give you the error message, since the Intel architecture has no machine instruction of this type; there is nothing the assembler can assemble the above line to.

12 Some More Addressing Modes

Following are examples of various addressing modes, using decrement and move for our example operations. The first four are ones we've seen earlier, and the rest are new. Here **x** is a label in the **.data** section.

```
decl %ebx           # register mode
decl (%ebx)         # indirect mode
decl x              # direct mode
movl $8888, %ebx    # source operand is in immediate mode

decl x(%ebx)        # indexed mode
decl 8(%ebx)        # based mode (really same as indexed)
decl (%eax,%ebx,4)  # scale-factor (my own name for it)
decl x(%eax,%ebx,4) # based scale-factor (my own name for it)
```

Let's look at the indexed mode first.

The expression **x(%ebx)** means that the operand is c(EBX) bytes past **x**. The name "indexed" comes from the fact that EBX here is playing the role of an array index.

For example, consider the C code

```
char x[100];
...
x[12] = 'g';
```

Since this is an array of type **char**, i.e. with each array element being stored in one byte, **x[12]** will be at the memory location 12 bytes past **x**. So, the C compiler might translate this to

```
movl $12, %ebx
movb $'g', x(%ebx)
```

Again, EBX is playing the role of the array index here, hence the name "indexed addressing mode." We say that EBX is the **index register** here. On Intel machines, almost any register can serve as an index register.

This won't work for **int** variables. Such variables occupy 4 bytes each. Thus our machine code would need an extra instruction in the **int** case. The C code

```
int x[100], i; // suppose these are global
...
x[i] = 8888;
```

would be translated to something like

```
movl i, %ebx
imull $4, %ebx
movl $8888, x(%ebx)
```

(Here **imull** is a multiplication instruction, to be discussed in detail in a later unit.)

So, Intel has another more general mode, which I have called “scale-factor mode” above. Here is how it works:

In the scale-factor mode, the syntax is

```
(register1, register2, scale_factor)
```

and the operand address is $\text{register1} + \text{scale_factor} * \text{register2}$.

We can now avoid that extra **imull** instruction by using scale-factor mode:

```
movl $x, %eax
movl i, %ebx
movl $8888, (%eax, %ebx, 4)
```

Of course, that still entailed an extra step, to set EAX. But if we were doing a lot of accesses to the array **x**, we would need to set EAX only once, and thus come out ahead.³²

By the way, if the instruction to be compiled had been

```
x[12] = 8888;
```

then plain old direct mode would have sufficed:

```
movl $8888, x+48
```

The point is that here the destination would be a fixed address, 48 bytes past **x** (remember that the address of **x** is fixed).

What about based mode? Indexed and based modes are actually identical, even though we think of them somewhat differently because we tend to use them in different contexts. In both cases, the syntax is

```
constant(register)
```

³² Actually, the product $4 \times i$ must still be computed, but it is now done as part of that third **movl** instruction, rather than as an extra instruction. This can speed things up in various ways, and it makes our code cleaner—EBX really does contain the index, not 4 times the index.

The action is then that the operand's location is constant+register contents. If the constant is, say, 5000 and the contents of the register is 240, then the operand will be at location 5240.

Note that in the case of `x(%ebx)`, the `x` is a constant, because `x` here means the address of `x`, which is a constant. Indeed, the expression `(x+200)(%ebx)` is also valid, meaning “EBX bytes past `x+200`,” or if you prefer thinking of it this way, “EBX+200 bytes past `x`.”

We tend to think of based mode a bit differently from indexed mode, though. We think of `x(%ebx)` as meaning “EBX bytes past `x`,” while we think of `8(%ebx)` as meaning “8 bytes past [the place in memory pointed to by] EBX.” The former is common in array contexts, as we have seen, while the latter occurs with stacks.

You will see full detail about stacks in our unit on subroutines later on. But for now, let's recall that local variables are stored on the stack. A given local variable may be stored, say, 8 bytes from the beginning of the stack. You will also learn that the ESP register points to the beginning of the stack. So, the local variable is indeed “8 bytes past ESP,” explaining why based mode is so useful.

13 GCC/Linker Operations

13.1 GCC As a Manager of the Compilation Process

Say our C program consists of source files `x.c` and `y.c`, and we compile as follows:

```
gcc -g x.c y.c
```

GCC³³ is basically a “wrapper” program, serving only a manager of the compilation process; it does not do the compilation itself. Instead, GCC will run other programs which actually do the work. We'll illustrate that here, using the `x.c/y.c` example throughout.

13.1.1 The C Preprocessor

First GCC will run the C preprocessor, `cpp`, which processes your `#define`, `#include` and other similar statements. For example:

```
% cat y.c
// y.c

#define ZZZZ 7

#include "w.h"

main()
{  int x,y,z;

    scanf("%d%d",&x,&y);
    x *= ZZZZ;
    y += ZZZZ;
    z = bigger(x,y);
    printf("%d\n",z);
}
```

³³The “English” name of this compiler is the GNU C Compiler, with acronym GCC, so it is customary to refer to it that way. The actual name of the file is `gcc`.

```

% cat w.h
// w.h

#define bigger(a,b) (a > b) ? (a) : (b)

% cpp y.c
# 1 "y.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "y.c"

# 1 "w.h" 1
# 6 "y.c" 2

main()
{ int x,y,z;

    scanf("%d%d",&x,&y);
    x *= 7;
    y += 7;
    z = (x > y) ? (x) : (y);
    printf("%d\n",z);
}

```

The preprocessor's output, seen above, now can be used in the next stage:

13.1.2 The Actual Compiler, CC1, and the Assembler, AS

Next, GCC will start up another program, **cc1**, which does the actual code translation. Even **cc1** does not quite do a full compile to machine code. Instead, it compile only to assembly language, producing a file **x.s** GCC will then start up the assembler, AS (file name **as**), which translates **x.s** to a true machine language (1s and 0s) file **x.o**. The latter is called an **object file**.

Then GCC will go through the same process for **y.c**, producing **y.o**.

Finally, GCC will start up the **linker** program, LD (file name **ld**), which will splice together **x.o** and **y.o** into an executable file, **a.out**. GCC will also delete the intermediate **.s** and **.o** files it produced along the way, as they are no longer needed.

13.2 The Linker

13.2.1 What Is Linked?

Recall our example above of a C program consisting of two source files, **x.c** and **y.c**. We noted that the compilation command,

```
gcc -g x.c y.c
```

would temporarily produce two object files, **x.o** and **y.o**, and that GCC would call the linker program, LD to splice these two files together to form the executable file **a.out**.

Exactly what does the linker do? Well, suppose **main()** in **x.c** calls a function **f()** in **y.c**. When the compiler sees the call to **f()** in **x.c**, it will say, "Hmm...There is no **f()** in **x.c**, so I can't really translate the call, since I don't know the address of **f()**." So, the compiler will make a little note in **x.o** to the linker saying, "Dear

linker: When you link **x.o** with other **.o** files, you will need to determine where **f()** is in one of those files, and then finish translating the call to **f()** in **x.c**.³⁴

Similarly, **x.c** may declare some global variable, say **z**, which the code in **y.c** references. The compiler will then leave a little note to the linker in **y.o**, etc.

So, the linker's job is to resolve issues like this before combining the **.o** files into one big executable file, **a.out**.

It doesn't matter whether our original source code was C/C++ or assembly language. Machine code is machine code, no matter what its source is, so the linker won't care which original language was the source of which **.o** file.

Though GCC invokes LD, we can run it directly too, which as we have seen is common in the case of writing assembly language code.³⁵

13.2.2 Headers in Executable Files

Even if our source code consists of just one file (as opposed to, for instance, our example of **x.c** and **y.c** above), so that there is nothing to link, we must still invoke the linker to produce an executable file. There are a couple of reasons for this.

First, in the case of C, you are linking in libraries without realizing it. If you run the **ldd** command on an executable which was compiled from C, you'll find that the executable uses the C library, **libc.so**. (More on **ldd** below.) At the very least, that library provides a place to begin execution, the label **_start** which we found in Section 4.1 is needed, and sets up your program's stack. That library also includes many basic functions, such as **printf()** and **scanf()**.

But beyond that, executable files consist not only of the actual machine code but also a **header**, a kind of introduction, at the start of the file. The header will state at what memory locations the various sections are to be loaded, how large the sections are, at what address execution is to begin, and so on. The linker will make these decisions, and place them in the header.

There are various standard formats for these headers. In Linux, the ELF format is used. If you are curious, the Linux **readelf** command will tell you what is in the header of any executable file. By running this command with, e.g., the **-s** option, you can find out the final addresses assigned to your labels by the linker.³⁶

There are headers in object files as well; the Linux command **objdump** will display these for you.

13.2.3 Libraries

A library is a conglomeration of several object files, collected together in one convenient place. Libraries can be either **static** or **dynamic**, as explained below. In Unix, library names end with **.a** (static) or **.so** (dynamic), possibly followed by a version number.³⁷ The names also usually start with "lib," so that for example **libc.so.6** is version 6 of the C library.

³⁴Or, in our GCC command line we may ask the linker to get other functions from libraries. See Section 13.2.3 below.

³⁵We can also apply GCC to the assembly-language file. GCC will notice that the file name has a **.s** extension, and thus will invoke the assembler, AS.

³⁶A more common command for this is **nm**. It's more general, as it does not apply only to ELF files, but it only gives symbol information.

³⁷You may have encountered dynamic libraries on Windows systems, which have the suffix **.dll**.

When you need code from a static library, the linker physically places the code in with your machine code. In the dynamic case, though, the linker merely places a note in your machine code, which states which libraries are needed; at run time, the OS will do the actual linking then. Dynamic libraries save a lot of disk space and memory (at the slight cost of a bit of a delay in loading the program into memory at run time), since we only need a single copy of any given library. Here is an overview of how the libraries are created and used:

One creates a static library by applying the **ar** command to the given group of **.o** files, and then possibly running **ranlib**. For example:

```
% gcc -c x.c
% gcc -c y.c
% ar lib8888.a x.o y.o
% ranlib lib8888.a
```

Later, if someone wants to check what's inside a static library, one runs **ar** on the **.a** file, with the **t** option, e.g.

```
% ar t lib8888.a
```

To create a dynamic library, one needs to use the **-fPIC** option when compiling to produce the **.o** files, and then one compiles the library by using GCC's **-shared** option, e.g.

```
% gcc -g -c -fPIC x.c
% gcc -g -c -fPIC y.c
% gcc -shared -o lib8888.so x.o y.o
```

In systems that use ELF, you can check what's in a dynamic library by using **readelf**, e.g.

```
% readelf -s lib8888.so
```

When you link to a library, GCC's **-l** option can be used to state which library to link in. For example, if you have a C program **w.c** which calls **sqrt()**, which is in the math library **libm.so**, you would type

```
% gcc -g w.c -lm
```

The notation "**-lxxx**" means "link the library named **libxxx.a** or **libxxx.so**."

One point to note, though, is that the library you need may not be in the default directories **/usr/lib**, **/lib** and those listed in **/etc/ld.so.cache**. If for example your library **libqrs.so** is in the directory **/a/b/c**, you would type

```
% gcc -g w.c -lqrs -L/a/b/c
```

This is fine in the static case, but remember that in the dynamic case, the library is not actually acquired at link time. All that is put in our executable file is the name of the library, but NOT its location. In other words, in the dynamic case, the **-L/a/b/c** in the example above is used by the linker to verify that the library does exist, but this location is NOT recorded. When the program is actually run, the OS will still look only in the default directories. There are various ways to tell it to look at others. One of the ways is to specify **/a/b/c** within the Unix environment variable **LD_LIBRARY_PATH**, e.g.


```
% setenv LD_LIBRARY_PATH /a/b/c
```

If you need to know which dynamic libraries an executable file needs, run **ldd**, e.g.

```
% ldd a.out
```

Clearly this is a complex topic. If you ever need to know more than what I have in this introduction, plug something like “shared library tutorial” into a Web search engine.

14 Inline Assembly Code for C++

The C++ language includes an **asm** construct, which allows you to embed assembly language source code right there in the midst of your C++ code.³⁸

This feature is useful, for instance, to get access to some of the fast features of the hardware. For example, say you wanted to make use of Intel’s fast MOVS string copy instruction. You could write an assembly language subroutine using MOVS and then link it to your C++ program, but that would add the overhead of subroutine call/return. (More on this in our unit on subroutines.) Instead, you could write the MOVS code there in your C++ source file.

Here’s a very short, overly simple example:

```
// file a.c

int x;

main()

{  scanf("%d",&x);
   __asm__("pushl x");
}
```

After doing

```
gcc -S a.c
```

the file **a.s** will be

```

        .file    "a.c"
        .section .rodata
.LC0:
        .string "%d"
        .text
.globl main
        .type    main, @function
main:
        leal     4(%esp), %ecx
        andl     $-16, %esp
        pushl    -4(%ecx)
        pushl    %ebp
        movl     %esp, %ebp
        pushl    %ecx
```

³⁸This is, as far as I know, available in most C++ compilers. Both GCC and Microsoft’s C++ compiler allow it.

```

        subl    $20, %esp
        movl    $x, 4(%esp)
        movl    $.LC0, (%esp)
        call    scanf
#APP
        pushl   x
#NO_APP
        addl    $20, %esp
        popl    %ecx

```

Our assembly language is bracketed by APP and NO_APP, and sure enough, it is


```
pushl x
```

For an introduction to how to use this feature, see the tutorials on the Web; just plug “inline assembly tutorial” into Google. For instance, there is one at <http://www.opennet.ru/base/dev/gccasm.txt.html>.

15 “Linux Intel Assembly Language”: Why “Intel”? Why “Linux”?

The title of this document is “Linux Intel Assembly Language”? Where do those qualifiers “Intel” and “Linux” come in?

First of all, the qualifier “Intel” refers to the fact, discussed earlier, that every CPU type has a different instruction set and register set. Machine code which runs on an Intel CPU will be rejected on a PowerPC CPU, and vice versa.

The “Linux” qualifier is a little more subtle. Suppose we have a C source file, **y.c**, and compile it twice on the same PC, once under Linux and then under Windows, producing executable files **y** and **y.exe**. Both files will contain Intel instructions. But for I/O and other OS services, the calls in **y** will be different from the calls in **y.exe**. 

16 Viewing the Assembly Language Version of the Compiled Code

We will have often occasion to look at the assembly language which the compiler produces as its first step in translating C code. In the case of GCC we use the **-S** option to do this. For example, if you type

```
gcc -S y.c
```

an assembly language file **y.s** will be created, as it would ordinarily, but the compiler will stop at that point, not creating object or executable files. You could even then apply **as** to this file, producing **y.o**, and then run **ld** on **y.o** to create the same executable file **a.out**, though you would also have to link in the proper C library code file.³⁹

³⁹In order to do the latter automatically, without having to know which file it is and where it is, use GCC to do the linking:

```
gcc y.o
```

GCC will call LD for you, also supplying the proper C library code file.

A String Operations

The STOS (STore String) family of instructions does an extremely fast copy of a given string to a range of consecutive memory locations, much faster than one could do with MOV instructions in a programmer-written loop.

For example, the **stosl** instruction copies the word in EAX to the memory word pointed to by the EDI register, and increments EDI by 4. If we add the **prefix, rep** (“repeat”), i.e. our line of assembly language is now

```
rep stosl
```

and then this is where the real power comes in. That single instruction effectively becomes the equivalent of a loop: The **stosl** instruction will be executed repeatedly, with ECX being decremented by 1 each time, until ECX reaches 0. This would mean that c(EAX) would be copied to a series of consecutive words in memory.

Note that the **rep** prefix is in effect an extra op code, prepended before the instruction itself. The instruction

```
stosl
```

translates to the machine code 0xab, but with **rep** prepended, the instruction is 0xabf3.⁴⁰

EDI, EAX and ECX are wired-in operands for this instruction. The programmer has no option here, and thus they do not appear in the assembly code.

The way to remember EDI is that the D stands for “destination.”

There is also the MOVS family, which copies one possibly very long string in memory to another place in memory. EDI again plays the destination role, i.e. points to the place to be copied to, and the ESI register points to the source string.⁴¹

Here is an example of STOS:

```
1  .data
2  x:
3      .space 20  # set up 5 words of space
4
5  .text
6
7  .globl _start
8
9  _start:
10     movl $x,%edi
11     movl $5,%ecx # we will copy our string to 5 words
12     movl $0x12345678,%eax # the string to be copied is 0x12345678
13     rep stosl
14 done:
```

Here is an example of MOVS, copying one string to another:

⁴⁰The lower-address byte will be f3, and the higher-address byte will be ab. Recall that the C notation “0x” describes a bit string by saying what the (base-16) number would be if the string were representing an integer. Since we are on a little-endian machine, the 0x notation for this instruction would be 0xabf3.

⁴¹Again, the S here refers to “source.”

```

1  .data
2  x: .string "abcde"  # 5 characters plus a null
3  y: .space 6
4
5  .text
6  .globl _start
7  _start:
8      movl $x, %esi
9      movl $y, %edi
10     movl $6, %ecx
11     rep movsb
12 done:

```

Again, you must use ESI, EDI and ECX for the source address, destination address and repeat count, respectively. No other registers may be used.

Warning: REP MOVS really is the equivalent (though much faster) of writing a loop to copy the characters from the source to the destination. An implication of this is that if the destination overlaps the source, you won't necessarily get a copy of the original source in the destination. If for instance the above code were

```

1  .data
2  x: .string "abcde"  # 5 characters plus a null
3  y: .space 9
4
5  .text
6  .globl _start
7  _start:
8      movl $x, %esi
9      movl %esi, %edi
10     addl $3, %edi
11     movl $6, %ecx
12     rep movsb
13 done:

```

then the resulting contents of the nine bytes starting at *y* would be ('a','b','c','a','b','c','a','b','c').

B Useful Web Links

- Linux assembly language Web page: <http://linuxassembly.org/>
- full **as** manual: http://www.gnu.org/manual/gas-2.9.1/html_mono/as.html (contains full list of directives, register names, etc.; op code names are same as Intel syntax, except for suffixes, e.g. "l" in "movl")
- Intel2gas, converter from Intel syntax to AT&T and vice versa: <http://www.niksula.cs.hut.fi/~mtiihone/intel2gas/>
- There are many Web references for the Intel architecture. Just plug "Intel instruction set" into any Web search engine.

One such site is <http://www.penguin.cz/~litrakl/intel/intel.html>.

For more detailed information, see the Intel manual, at <ftp://download.intel.com/design/Pentium4/manuals/24547108.pdf>. There is an index at the end.

Also, if you need to learn about a specific instruction, often you can get some examples by plugging the instruction name and the word *example* into Google or any other Web search engine. Use the

family name for the instruction, e.g. MOV instead of **movl**, **movb** etc. This will ensure that your search will pick up both Intel-syntax-oriented and AT&T-syntax-oriented sites.

By the way, if you do get information on the Web which is Intel-syntax-oriented, use the **intel2gas** program, mentioned above, to convert it to AT&T.

- NASM assembler home page: <http://nasm.2y.net/>
- the ALD debugger: <http://ellipse.mcs.drexel.edu/ald.html>
- my tutorials on debugging, featuring my slide show, using **ddd**: <http://heather.cs.ucdavis.edu/~matloff/debug.html>
- Unix tutorial: <http://heather.cs.ucdavis.edu/~matloff/unix.html>
- Linux installation guide: <http://heather.cs.ucdavis.edu/~matloff/linux.html>

C Top-Down Programming

Programming is really mind-boggling work. When one starts a large, complex program, it is really a daunting feeling. One may feel, “Gee, there is so much to do...” It is imperative that you deal with this by using the **top-down** approach to programming, also known as **stepwise refinement**. Here is what it is.

You start by writing **main()** (or in assembly language, **_start()**), and—this is key—making sure that it consists of no more than about a dozen lines of code.⁴² Since most programs are much bigger than just 12 lines, this of course means that some, many most, of these 12 lines will be calls to functions (or in the assembly language case, calls to subroutines). It is important that you give the functions good names, in order to remind yourself what tasks the various functions will perform.

In a large, complex program, some of those functions will also have a lot to do, so you should impose upon yourself a 12-line on them too! In other words, some functions will themselves consist of calls to even more functions.

The point is that this is something easy and somewhat quick to do, something that you can do without feeling overwhelmed. Each 12-line module which you write is simple enough so that you can really focus your thoughts.

⁴²I’m just using 12 lines as an example. You may prefer a smaller limit, especially when working at the assembly language level.