

Subroutines on Intel CPUs

Norman Matloff

March 18, 2007

© 2002-2007, N.S. Matloff

Contents

1	Overview	3
2	Stacks	3
3	CALL, RET Instructions	4
4	Arguments	4
5	Ensuring Correct Access to the Stack	5
6	Cleaning Up the Stack	5
7	Full Examples	6
7.1	First Example	6
7.2	Second Example	9
8	Interfacing C/C++ to Assembly Language	10
8.1	Example	10
8.2	Cleaning Up the Stack?	12
8.3	More Segments	13
8.4	Multiple Arguments	14
8.5	Nonvoid Return Values	14
8.6	Calling C and the C Library from Assembly Language	14
8.7	Local Variables	15
8.8	Use of EBP	16
8.8.1	GCC Calling Convention	16

8.8.2	The Stack Frame for a Given Call	16
8.8.3	The Stack Frames Are Chained	17
8.8.4	ENTER and LEAVE Instructions	18
8.8.5	Application of Stack Frame Structure	19
8.9	The LEA Instruction Family	20
8.10	The Function main() IS a Function, So It Too Has a Stack Frame	21
8.11	Once Again, There Are No Types at the Hardware Level!	23
8.12	What About C++?	24
8.13	Putting It All Together	24
9	Subroutine Calls>Returns Are “Expensive”	27
10	Debugging Assembly Language Subroutines	27
10.1	Focus on the Stack	27
10.2	A Special Consideration When Interfacing C/C++ with Assembly Language	28
11	Macros	28

1 Overview

Programming classes usually urge the students to use **top-down** and **modular** design in their coding. This in turn means using a lot of calls to **functions** or **methods** in C-like languages, which are called **subroutines** at the machine/assembly level.¹

This document serves as a brief introduction to subroutine calls in Intel machine language. It assumes some familiarity with Intel 32-bit assembly language, using the AT&T syntax with Linux.

2 Stacks

Most CPU types base subroutine calls on the notion of a **stack**. An executing program will typically have an area of memory defined for use as a **stack**. Most CPUs have a special register called the **stack pointer** (SP in general, ESP on 32-bit Intel machines), which points to the **top** of the stack.

Note carefully that the stack is not a “special” area of memory, say with a “fence” around; wherever SP points, that is the stack. And that is true even if we aren’t using a stack at all, in which case the “stack” is garbage.

Stacks typically grow toward memory address 0.² If an item is added—**pushed**—onto the stack, the SP is decremented to point to the word preceding the word it originally pointed to, i.e. the word with the next-lower address than that of the word SP had pointed to beforehand. Similarly SP is incremented if the item at the top of the stack is **popped**, i.e. removed. Thus the words following the top of the stack, i.e. with numerically larger addresses, are considered the interior of the stack.

Say for example we wish to push the value 35 onto the stack. Intel assembly language code to do this would be, for instance,

```
subl $4,%esp # expand the stack by one word
movl $35,(%esp) # put 35 in the word which is the new top-of-stack
```

However, typically a CPU will include a special instruction to do pushes. On Intel machines, for example, we could do the above using just one instruction:³

```
push $35
```

(There is also a **pushl** instruction, for consistency with instructions like **movl**, but it is just another name for **push**. Note that this is because the items on the stack must be of word size anyway.)

By the way, instructions like

```
pushl w
```

where **w** is a label in the **.data** section, are legal, a rare exception to Intel’s theme of not allowing direct memory-to-memory operations.

Similarly, if we wish to pop the stack and have the popped value placed in, say, ECX, we would write

¹This latter term is also used in various high-level languages, such as FORTRAN and Perl.

²For this reason, we usually draw the stack with memory address 0 at the top of the picture, rather than the bottom.

³This would be not only easier to program, but more importantly would also execute more quickly.

```
pop %ecx
```

Note carefully that it is the *stack* which is popped, not the ECX register.

Keep in mind that a pop does not change memory. The item is still there; the only change is that it is not considered part of the stack anymore. And the stack pointer merely is a means for indicating what is considered part of the stack; by definition, the stack consists of the word pointed to by ESP plus all words at higher addresses than that. The popped item will still be there in memory until such time, if any, that another push is done, overwriting the item.

3 CALL, RET Instructions

The actual call to a subroutine is done via a `CALL` instruction. Execution of this instruction consists of two actions:

- The current value of the PC is pushed on the stack.
- The address of the subroutine is placed into the PC.

We say that the first of these two actions pushes the **return address** onto the stack, meaning the place that we wish to return to after we finish executing the subroutine.

Note that both of the actions above, like those of any instruction, are performed by the hardware.

For example, consider the code

```
call abc
addl %eax,%ebx
```

At first the PC is pointing to the **call** instruction. After the instruction is fetched from memory, the CPU, as usual, increments the PC to point to the next instruction, i.e. the **addl**. From the first bullet item above, we see that that latter instruction's address will be pushed onto the stack — which is good, because we do want to save that address, as it is the place we wish to return to when the subroutine is done. So, the terminology used is that the **call** instruction saves the return address on the stack.

What about the second bullet item above? Its effect is that we do a jump to the subroutine. So, what **call** does is save the return address on the stack and then start execution of the subroutine.

The very last instruction which the programmer puts into the subroutine will be **ret**, a return instruction. It pops the stack, and places that popped value into the PC. Since the return address had been earlier pushed onto the stack, we now pop it back off, and the return address will now be in the PC — so we are executing the instruction immediately following the **call**, such as the **addl** in the example above, just as desired.⁴

4 Arguments

Functions in C code usually have arguments (another term used is *parameters*, and that is also true in assembly language. The typical way the arguments are passed to the subroutine is again via the stack. We

⁴This assumes we've been careful to first pop anything we pushed onto the stack subsequent to the call. More on this later.

simply push the arguments onto the stack before the call, and then pop them back off after the subroutine is done.

For instance, in the example above, suppose the subroutine **abc** has two integer arguments, and in this particular instance they will have the values 3 and 12. Then the code above might look like

```
push $12
push $3
call abc
pop %edx # assumes EDX isn't saving some other data at this time
pop %edx
addl %eax,%ebx
```

The subroutine then accesses these arguments via the stack. Say for example **abc** needs to add the two arguments and place their sum in ECX. It could be written this way:

```
...
movl 4(%esp),%ecx
addl 8(%esp),%ecx
...
ret
```

In that **movl**, the source operand is 4 bytes past where SP is pointing to, i.e. the next-to-top element of the stack. In the call example above, this element will be the 3. (The top element of the stack is the return address, since it was the last item pushed.) The second **addl** picks up the 12.

Here's what the stack looks like at the time we execute that **movl**:

```
argument 2
argument 1
ESP → saved return address
towards 0 ↓
```

5 Ensuring Correct Access to the Stack

But wait a minute. What if the calling program above had also been storing something important in ECX? We must write **abc** to avoid a clash. So, we first save the old value of ECX—where else but the stack?—and later restore it. So, the beginning and end of **abc** might look like this:

```
push %ecx
...
movl 8(%esp),%ecx
addl 12(%esp),%ecx
... # ECX used here
pop %ecx
ret
```

Note that we had to replace the stack offsets 4 and 8 by 8 and 12, to adjust for the fact that one more item will be on the stack. No one but the programmer can watch out for this kind of thing. The assembler and hardware, for example, would have no way of knowing that we are wrong if we were to fail to make this adjustment; we would access the wrong part of the stack, and neither the assembler nor hardware would complain.

6 Cleaning Up the Stack

Also, note that just before the `ret`, we “clean up” the stack by popping it. This is important for several reasons:

- Whatever we push onto the stack, we should eventually pop, to avoid inordinate growth of the stack. For example, we may be using memory near 0 for something else, so if the stack keeps growing, it will eventually overlap that other data, overwriting it.
- We need to restore ECX to its state prior to the call to `abc`.
- We need to ensure that the return instance does return to the correct place.⁵

Note that in our call to `abc` above, we followed the call with some stack cleanup as well:

```
pop %edx # assumes EDX isn't saving some other data at this time
pop %edx
```

We needed to remove the two arguments which we had pushed before the call, and these two pops do that. The pop operation insists that the popped value be placed somewhere, so we use EDX in a “garbage can” role here, since we won’t be using the popped values. Or, we could do it this way:

```
addl $8,%esp
```

This would be faster-executing and we wouldn’t have to worry about EDX.

7 Full Examples

7.1 First Example

In the following modification of an example from our unit introducing assembly language, we again sum up elements of an array, but we handle initialization of the registers by a subroutine. We also allow starting the summation in the middle of the array, by specifying the starting point as an argument to the subroutine, and allow specification that only a given number of elements be summed:

```
1  .data
2  x:
3      .long 1
4      .long 5
5      .long 2
6      .long 18
7      .long 8888
8      .long 168
9  n:  .long 3
10 sum:
11     .long 0
12
13  .text
```

⁵The reader should ponder what would happen if the programmer here forgot to include that pop just before the return.

```

14 # EAX will contain the number of loop iterations remaining
15 # EBX will contain the sum
16 # ECX will point to the current item to be summed
17
18 .globl _start
19 _start:
20     # push the arguments before call: place to start summing, and
21     # number of items to be summed
22     push $x+4 # here we specify to start summing at the 5
23     push n # here we specify how many to sum
24     call init
25     addl $8, $esp # clean up the stack
26 top:  addl (%ecx), %ebx
27     addl $4, %ecx
28     decl %eax
29     jnz top
30 done: movl %ebx, sum
31
32 init:
33     movl $0, %ebx
34     # pick up arguments from the stack
35     mov 8(%esp), %ecx
36     mov 4(%esp), %eax
37     ret

```

Here is the output assembly listing:

```

GAS LISTING full.s                               page 1
 1          .data
 2          x:
 3 0000 01000000          .long 1
 4 0004 05000000          .long 5
 5 0008 02000000          .long 2
 6 000c 12000000          .long 18
 7 0010 B8220000          .long 8888
 8 0014 A8000000          .long 168
 9 0018 03000000          n:  .long 3
10          sum:
11 001c 00000000          .long 0
12
13          .text
14          # EAX will contain the number of loop iterations remaining
15          # EBX will contain the sum
16          # ECX will point to the current item to be summed
17
18          .globl _start
19          _start:
20              # push the arguments before call: place to start summing, and
21              # number of items to be summed
22 0000 68040000          push $x+4 # here we specify to start summing at the 5
22          00
23 0005 FF351800          push n # here we specify how many to sum
23          0000
24 000b E8110000          call init
24          00
25 0010 83C408          addl $8, %esp # clean up the stack
26 0013 0319          top:  addl (%ecx), %ebx
27 0015 83C104          addl $4, %ecx
28 0018 48             decl %eax
29 0019 75F8          jnz top
30 001b 891D1C00       done: movl %ebx, sum
30          0000
31
32          init:
33 0021 BB000000          movl $0, %ebx
33          00

```

```

34                               # pick up arguments from the stack
35 0026 8B4C2408                 mov 8(%esp), %ecx
36 002a 8B442404                 mov 4(%esp), %eax
37 002e C3                       ret

```

DEFINED SYMBOLS

```

full.s:2      .data:00000000 x
full.s:9      .data:00000018 n
full.s:10     .data:0000001c sum
full.s:19     .text:00000000 _start
full.s:32     .text:00000021 init
full.s:26     .text:00000013 top
full.s:30     .text:0000001b done

```

NO UNDEFINED SYMBOLS

Note from line 24 of the assembly output listing that the CALL instruction assembled to e811000000. That breaks down to an op code of e8 and a distance field of 11000000. The latter (after accounting for little-endianness) is the distance from the instruction after the CALL to the subroutine, which from lines 25 and 33 can be seen to be $0x0021-0x0010 = 0x11$.⁶

To gain a more concrete understanding of how the stack is used in subroutine calls, let's use GDB to inspect what happens with the stack when this program is run:

```

% gdb a.out
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
...
Breakpoint 1, _start () at SubArgsEx.s:24
24      call init
Current language: auto; currently asm
(gdb) p/x $eip
$1 = 0x804807f
(gdb) p/x $esp
$2 = 0xbfffcec8
(gdb) si
33      movl $0, %ebx
(gdb) p/x $eip
$3 = 0x8048093
(gdb) p/x $esp
$4 = 0xbfffcec4
(gdb) x/3w $esp
0xbfffcec4:      0x08048084      0x00000003      0x080490a8
(gdb) p/x &x
$5 = 0x80490a4

```

So, the PC value changed from 0x804807f to 0x0x8048093, reflecting the fact that a CALL does do a jump.

The CALL also does a push (of the return address), and sure enough, the stack did expand by one word, as can be seen by the fact that ESP decreased by 4, from 0xbfffcec8 to 0xbfffcec4.

The stack should now have the return address at the top, followed by the value of **n** and the address of **x** plus 4. Let's check that return address. The PC value just before the call had been, according our GDB output above, 0x804807f, and since (as discovered above from the output of **as -a**) the CALL was a 5-byte instruction, the saved return address should be $0x804807f+5 = 0x8048084$, which is indeed what we see on the top of the stack. By the way, you should be able to deduce the address at which the **.text** section begins.

Now look at the rest of our GDB session:

⁶This is what is known for Intel machines as a **near call**. Back in the days when Intel CPUs could only access 64K segment of memory at a time, they needed “near” (within-segment) and “far” (inter-segment) calls, but with the flat 32-bit addressing model used today, this is outmoded, and everything is “near.”

```

(gdb) b done
Breakpoint 2 at 0x804808d: file SubArgsEx.s, line 30.
(gdb) c
Continuing.

Breakpoint 2, done () at SubArgsEx.s:30
30     done: movl %ebx, sum
(gdb) si
33     movl $0, %ebx
(gdb) p sum
$6 = 25
(gdb) si
35     mov 8(%esp), %ecx
(gdb) q
The program is running.  Exit anyway? (y or n) y

```

After executing the instruction at **done**, I checked to see if **sum** was correct, which it was. At that point, I really should have stopped, since the program was indeed done. But I continued anyway, issuing another **si** command to GDB. Did GDB refuse? Did “the little man inside the computer” refuse? Heck, no! Remember, it’s just a dumb machine. The CPU has no idea that the instruction at **done** was the last instruction in the program. So, it keeps going, executing the next instruction in memory. That instruction is the MOV at the beginning of the subroutine **init()**! So, you can see that there is nothing “special” about a subroutine. There is no physical boundary between modules of code, in this case between the calling module and **init()**. It’s all just a bunch of instructions.

7.2 Second Example

The next example speaks for itself, through the comments at the top of the file. By the way, when you read through it, make sure you understand what we are doing with the SHR instruction.

```

1 # the subroutine findfirst(v,w,b) finds the first instance of a value v in a
2 # block of w consecutive words of memory beginning at b, returning
3 # either the index of the word where v was found (0, 1, 2, ...) or -1 if
4 # v was not found; beginning with _start, we have a short test of the
5 # subroutine
6
7 .data # data section
8 x:
9     .long 1
10    .long 5
11    .long 3
12    .long 168
13    .long 8888
14 .text # code section
15 .globl _start # required
16 _start: # required to use this label unless special action taken
17     # push the arguments on the stack, then make the call
18     push $x+4 # start search at the 5
19     push $168 # search for 168 (deliberately out of order)
20     push $4 # search 4 words
21     call findfirst
22 done:
23     movl %edi, %edi # dummy instruction for breakpoint
24 findfirst:
25     # finds first instance of a specified value in a block of words
26     # EBX will contain the value to be searched for
27     # ECX will contain the number of words to be searched
28     # EAX will point to the current word to search
29     # return value (EAX) will be index of the word found (-1 if not found)
30     # fetch the arguments from the stack

```

```

31     movl 4(%esp), %ebx
32     movl 8(%esp), %ecx
33     movl 12(%esp), %eax
34     movl %eax, %edx # save block start location
35     # top of loop; compare the current word to the search value
36 top:  cml ( %eax), %ebx
37     jz found
38     decl %ecx # decrement counter of number of words left to search
39     jz notthere # if counter has reached 0, the search value isn't there
40     addl $4, %eax # otherwise, go on to the next word
41     jmp top
42 found:
43     subl %edx, %eax # get offset from start of block
44     shrl $2, %eax # divide by 4, to convert from byte offset to index
45     ret
46 notthere:
47     movl $-1, %eax
48     ret

```

8 Interfacing C/C++ to Assembly Language

Programming in assembly language is very slow, tedious and unclear, so we try to avoid it, sticking to high-level languages such as C. But in some cases we need to write *part* of a program in assembly language, either because there is something which is highly machine-dependent,⁷ or because we need extra program speed and this part of the program is the main time consumer.

A good example is Linux. Most of it is written in C, for convenience and portability across machines, but small portions are written in assembly language, to get access to certain specific features of the given hardware. When Linux is ported to a new type of hardware, these portions must be rewritten, but fortunately they are small.

At first it might seem “unnatural” to combine C and assembly language. But remember, both `.c` and `.s` files are translated to machine language, so we are just combining machine language with machine language, not unnatural at all.

8.1 Example

So, here we will see how we can interface C code to an assembly language subroutine. Here is our C code:

```

1 // TryAddOne.c, example of interfacing C to assembly language
2 // paired with AddOne.s, which contains the function addone()
3 // compile by assembling AddOne.s first, and then typing
4 //
5 // gcc -g -o tryaddone TryAddOne.c AddOne.o
6 //
7 // to link the two .o files into an executable file tryaddone
8 // (recall the gcc invokes ld)
9
10 int x;
11
12 main()
13 {
14     x = 7;

```

⁷Note that to a large extent we can deal with machine-dependent aspects even from C. We can deal with the fact that different machines have different word sizes, for example, by using C’s `sizeof()` construct. However, this is not the case for some other situations.

```

15     addone(&x);
16     printf("%d\n",x); // should print out 8
17     exit(1);
18 }
19

```

I wrote the function **addone()** in assembly language. In order to do so, I needed to know how the C compiler was going to translate the call to **addone()** to machine/assembly language. In order to determine this, I compiled the C module with the **-S** option, which produces an assembly language version of the compiled code:

```

1  % gcc -S TryAddOne.c
2  % more TryAddOne.s
3      .file    "TryAddOne.c"
4      .version    "01.01"
5  gcc2_compiled.:
6      .section    .rodata
7  .LC0:
8      .string    "%d\n"
9  .text
10     .align 4
11     .globl main
12     .type      main,@function
13  main:
14     pushl    %ebp
15     movl    %esp, %ebp
16     subl    $8, %esp
17     movl    $7, x
18     subl    $12, %esp
19     pushl    $x
20     call    addone
21     addl    $16, %esp
22     subl    $8, %esp
23     pushl    x
24     pushl    $.LC0
25     call    printf
26     addl    $16, %esp
27     subl    $12, %esp
28     pushl    $1
29     call    exit
30  .Lfel:
31     .size    main,.Lfel-main
32     .comm    x,4,4
33     .ident   "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.1 2.96-85)"
34 %

```

There is quite a lot in that **.s** output file, but the part of interest to us here, for the purpose of writing **addone()**, is this:

```

    pushl    $x
    call     addone

```

This confirms what we expected, that the compiler would produce code that transmits the argument to **addone()** by pushing it onto the stack before the call.⁸ So, we write **addone()** so as to get the argument from the stack:

```

1  # AddOne.s, example of interfacing C to assembly language
2  #

```

⁸Note, though, that the compiler also first expanded the stack by 12 bytes, for no apparent reason. More on this below.

```

3 # paired with TryAddOne.c, which calls addone
4 #
5 # assemble by typing
6 #
7 #   as --gstabs -o AddOne.o AddOne.s
8 #
9
10 # note that we do not have a .data section here, and normally would not;
11 # but we could do so, and if we wanted that .data section to be visible
12 # from the calling C program, we would have .globl lines for whatever
13 # labels we have in that section
14
15 .text
16
17 # need .globl to make addone visible to ld
18 .globl addone
19
20 addone:
21
22     # will use EBX for temporary storage below, and since the calling
23     # module might have a value there, better save the latter on the stack
24     # and restore it when we leave
25     push %ebx
26
27     # at this point the old EBX is on the top of the stack, then the
28     # return address, then the argument, so the latter is at ESP+8
29
30     movl 8(%esp), %ebx
31
32     incl (%ebx) # need the (), since the argument was an address
33
34     # restore value of EBX in the calling module
35     pop %ebx
36
37     ret

```

The comments here say we should save, and later restore, the EBX register contents as they were prior to the call. Actually, our **main()** here does not use EBX in this case (see the assembly language version above), but since we might in the future want to call **addone()** from another program, it's best practice to protect EBX as we have done here.

So, by the time we reach the **movl** instruction, the stack looks like this:

```

          argument
          saved return address
ESP →    saved EBX value
          towards 0 ↓

```

The **movl** instruction thus needs to go 8 bytes deep into the stack to pick up the argument, which was the address of the word to be incremented. Then the **incl** instruction does the incrementing via indirect addressing, again since the argument is an address.

The reader is urged to compile/assemble/link as indicated in the comments in the **.c** and **.s** files above, and then execute the program, to see that it does indeed work. Seeing it actually happen will increase your understanding!

Note carefully that **TryAddOne.c** and **AddOne.s** are not “programs” by themselves. They are each the source code for a portion of the same program, the executable file **tryaddone**. This is no different from the situation in which we have C source code in two **.c** files, compile them into two **.o** files, and then link to make a single executable.

8.2 Cleaning Up the Stack?

Let's check whether the compiler fulfilled its "civic responsibility" by cleaning up the stack after the call. In the `.s` file produced by `gcc -S` above, we saw this code:

```
subl    $12, %esp
pushl   $x
call    addone
addl    $16, %esp
subl    $8, %esp
```

This is a bit odd. Since we had only pushed one argument, i.e. one word, the natural cleanup would have been to then pop one word, as follows:

```
call    addone
addl    $4, %esp
```

(or use a **pop** instruction instead of the **addl**). In other words, we should shrink the stack by 4 bytes after the call. Yet the net effect of the code generated by the compiler is to shrink the stack by $16 - 8 = 8$ bytes, not 4.

On the other hand, recall that with the instruction

```
subl    $12, %esp
```

the compiler expanded the stack by 12 bytes *too much* before the call. In other words, the overall effect of the call has been to expand the stack by $12 - (16-8) = 4$ bytes!

Supposedly the GCC people designed things this way, to leave gaps between one call and the next on the stack. This has makes things a little safer, as it is harder for a return address to be accidentally overwritten.

8.3 More Segments

By the way, the compiler has used the `.comm` directive to set up storage for our global variable `x`. It asks for 4 bytes of space, aligned on an address which is a multiple of 4. The linker will later set things up so that the variable `x` will be stored in the `.bss` section, which is like the `.data` section except that the data is uninitialized. Recall that in our program above, the declaration of `x`, a global variable, was simply

```
int x;
```

If instead the declaration had been

```
int x = 28;
```

then `x` would be in the `.data` section.

Also, the label `.LC0` is in yet another kind of section, `.rodata` ("read-only data").

The reader is strongly encouraged to run the Unix `nm` command on any executable file, say the compiled version of the C program here. In the output of that command, symbols (names of functions and global

variables) are marked as T, D, B, R and so on, indicating that the item is in the **.text** section, **.data** section, **.bss** section, **.rodata** section, etc., and the addresses assigned to them by the linker are shown.

In Linux, the stack section begins at address 0xbfffffff, and as mentioned, grows toward 0. The **heap**, from which space is allocated dynamically when your program calls **malloc()** or invokes the C++ operator **new**, starts at 0xbffff000 and grows away from 0.

8.4 Multiple Arguments

What if **addone()** were to have two arguments?⁹ A look at the output of the compilation of the call to **printf()** above shows that arguments are pushed in reverse order, in this case the second before the first. So, if **addone()** were to have two arguments, we would have to write the code for the function accordingly, making use of the fact that the first argument will be closer to the top of the stack than the second.

8.5 Nonvoid Return Values

The above discussion presumes that return value for the assembly language function is **void**. If this is not the case, then the assembly language function must place the return value in the EAX register. This is because the C compiler will place code after the call to the function which picks up the return value from EAX.

That assumes that the return value fits in EAX, which is the case for integer, character or pointer values. It is not the case for the type **long long** in GCC, implemented on 32-bit Intel machines in 8 bytes. If a function has type **long long**, GCC will return the value in the EDX:EAX pair, similar to the IMUL case.

The case of **float** return values is more complicated. The Intel chip has separate registers and stack for floating-point operations. See our unit on arithmetic and logic.

8.6 Calling C and the C Library from Assembly Language

The same principles apply if one has a C function which one wishes to call from assembly language. We merely have to take into account the order of parameters, etc., verifying by running **gcc -S** on the C function.

A bit more care must be taken if we wish to call a C library function, e.g. **printf()**, from assembly language. Your call is the same, of course, but the question is how to do the linking. The easiest way to do this is to actually use GCC, because it will automatically handle linking in the C library, etc.

Here is an example, again from our first assembly language example of summing up four array elements:

```
1  .data
2
3  x:
4      .long  1
5      .long  5
6      .long  2
7      .long  18
8
9  sum:
10     .long  0
11
```

⁹Note that when the C compiler compiles **TryAddOne.c** above, it does not know how many arguments **addone()** really has, since **addone()** is in a separate source file. (The compiler won't even know whether that source file is C or assembly code.) It simply knows how many arguments we have used in this call, which need not be the same.

```

12 fmt:  .string "%d\n"
13
14 .text
15
16 .globl main
17 main:
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 printsum:
28     pushl %ebx
29     pushl $fmt
30     call printf
31 done: movl %eax, %eax

```

We wanted to print out the sum which was in EBX. We know that **printf()** has as its first parameter the output format, which is a character string, with the other parameters being the items to print. We only have one such item here, EBX. So we push it, then push the address of the format string, then call.

We then run it through GCC.

```
gcc -g -o sum sum.s
```

Given that we have chosen to use GCC (which, recall, we did in order to avoid having to learn where the C library is, etc.), this forced us to chose **main** as the label for the entry point in our program, instead of **_start**. This is because GCC will link in the C startup library. The label **_start** is actually in that library, and when our program here, or any other C program, begins execution, it actually will be at that label. The code at that label will do some initialization and then will call **main**. So, we had better have a label **main** in our code!

Note that the C library function that you call may use EAX, ECX and EDX! For example, most C library functions have return values, typically success codes, and of course they are returned in EAX. Recall from Section 8.8.1 that if you write a C function, you don't have to worry about the calling module having "live" values in EAX, ECX and EDX—that is, if the calling module is in C. Here it isn't.

8.7 Local Variables

Not only does the compiler use the stack for storage of arguments, it also stores local variables there. Consider for example

```

int sum(int *x, int n)
{ int i=0,s=0;

  for (i = 0; i < n; i++)
    s += x[i];
  return s;
}

```

As mentioned in a previous unit, the locals will be stored in "reverse" order: The two variables will be in adjacent words of memory, but the first one, i.e. the one at the lower address, will be **s**, not **i**. As you will see below, this will be done by in essence pushing **i** and then pushing **s** onto the stack.

Let's explore this by viewing the compiled code (in assembly language form) for this by running `gcc -S`. Here is an excerpt from the output:¹⁰

```
...
sum:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    movl    $0, -8(%ebp)
    movl    $0, -4(%ebp)
```

You can see that the compiler first put in code to copy ESP to EBP, a standard operation which is explained below in Section 8.8, and then put in code to expand the stack by 8 bytes for the two local variables. set both locals to 0.¹¹ Note that the local variables are not really “pushed” onto the stack; there is simply room made for them on the stack. Note also that later in the compiled code for `sum()` (not shown here), the compiler needed to insert code to pop off those local variables from the stack before the `ret` instruction; without this, the `ret` would try to jump to the place pointed to by the first local variable—total nonsense.

8.8 Use of EBP

8.8.1 GCC Calling Convention

Recall that in our unit on assembly language programming, we mentioned that you should not use ESP for general storage if you are using subroutines. It should now be clear why we noted that restriction. Now, here is a new restriction: If you are interfacing assembly language to C/C++, you should avoid using the EBP register for general storage. In this section, we'll see why.

Note the first three instructions in the implementation of `sum()` above:

```
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
```

This **prologue** is standard for C compilers on Intel machines. The old value of EBP is pushed on the stack; the current value of ESP is saved in EBP; and the stack is expanded (by decreasing ESP) to allocate space for the locals.

If you use GCC, then GCC will make sure that the calling module will not have any “live” values in EAX, ECX or EDX. So, the called module need not save the values in any of these registers. But if the called module uses ESI, EDI or EBX, the called module must save these in the prologue too.

These are referred in the computer world as **calling conventions**, a set of guidelines to follow in writing subroutines for a given machine under a given compiler.

8.8.2 The Stack Frame for a Given Call

The portion of the stack which begins at the place pointed to by EBP immediately after execution of the prologue of any given C function `g()`, and ends at the place pointed to by ESP, is called the **stack frame** for

¹⁰Different versions of GCC may produce somewhat different code.

¹¹Different versions of `gcc` may not produce the same code. Always run `gcc -S` before doing any interfacing of C to assembly language.

g(). This is basically **g()**'s current portion of the stack. Let's see what this consists of. Let's say that **g()** had been called by **f()**.

From the prologue code, we can see that the beginning of **g()**'s frame will consist of the saved value that had been in EBP before the call to **g()**. Well, that was the address of the beginning of **f()**'s stack frame! So, we see that the first element in **g()**'s frame will be a pointer to **f()**'s frame (which of course is at higher addresses than **g()**'s).

We also see that **g()**'s stack frame will contain **g()**'s local variables. And if **g()** saves some other register values on the stack, to protect **f()**, those will be part of **g()**'s frame too.

During the time **g()** is executing, the end of **g()**'s stack frame is pointed to by ESP. However, if **g()** in turn makes a call to some other function, say **h()**, that function will also have a stack frame, at lower addresses than **g()**'s frame. However, As **g()** executes, this position may move. This can occur for instance, if **g()** does call another function, say **h()**. ESP will decrease due to expansion of the stack by the called function's arguments and return address, which are considered part of **g()**'s stack frame, not **h()**'s. Here is what **g()**'s stack frame (and a bit of **f()**'s and **h()**'s) will consist of, just after the CALL is executed but before **h()**'s prologue begins executing:

```
EBP →    address in f() to return to when g() is done (end of f()'s frame)
         address of f()'s stack frame (start of g()'s frame)
         g()'s first declared local variable
         g()'s second declared local variable
         ...
         g()'s last declared local variable
         any stack space g() is currently using as "scratch area"
         last argument in g()'s call to h()
         ...
         second argument in g()'s call to h()
         first argument in g()'s call to h()
ESP →    address in g() to return to when h() is done (end of g()'s frame)
         address of g()'s stack frame (start of h()'s frame)
towards 0 ↓
```

Of course, after we execute **h()**'s prologue, ESP will change again, and will demarcate the end of **h()**'s stack frame (and EBP will demarcate the beginning of that frame). Once **h()** finishes execution, the stack will shrink back again, and ESP will increase and demarcate the end of **g()**'s frame again.

8.8.3 The Stack Frames Are Chained

An implication of the structure depicted above is that in any function's stack frame, the first (i.e. highest-address) element will contain a pointer to the beginning of the caller's stack frame. In that sense, the beginning elements of the various stack frames form a linked list, enabling one to trace through the stack information in a chain of nested calls. And since we know that EBP points to the current frame, we can use that as our starting point in traversing this chain.

The authors of GDB made use of this fact when they implemented GDB's **bt** ("backtrace") command. Let's review that command. Consider the following example:

```
1 void h(int *w)
2 { int z;
```

```

3     *w = 13 * *w;
4 }
5
6 int g(int u)
7 { int v;
8   h(&u);
9   v = u + 12;
10  return v;
11 }
12
13 main()
14 { int x,y;
15   x = 5;
16   y = g(x);
17 }

```

Let's execute it in GDB:

```

(gdb) b h
Breakpoint 1 at 0x804833a: file bt.c, line 3.
(gdb) r
Starting program: /root/50/a.out

Breakpoint 1, h (w=0xfef5d2ac) at bt.c:3
3     *w = 13 * *w;
(gdb) bt
#0 h (w=0xfef5d2ac) at bt.c:3
#1 0x08048360 in g (u=5) at bt.c:8
#2 0x0804839c in main () at bt.c:16

```

We are now in **h()** (shown as frame 0 in the **bt** output), having called it from location 0x08048360 in **g()** (frame 1), which in turn had been called from location 0x0804839c in **main()** (frame 2).

And GDB allows us to temporarily change our context to another frame, say frame 1, and take a look around:

```

(gdb) f 1
#1 0x08048360 in g (u=5) at bt.c:8
8     h(&u);
(gdb) p u
$1 = 5
(gdb) p v
$2 = 0

```

Make sure you understand how GDB—which, remember, is itself a program—is able to do this. It does it by inspecting the stack frames of the various functions, and the way it gets from one frame to another within the stack is by the fact that the first element in a function's stack frame is a pointer to the first element of the caller's stack frame.

By using the structure shown above, the compiler is ensuring that we will always be able to get to return addresses, arguments and so on of “ancestral” calls.

8.8.4 ENTER and LEAVE Instructions

We've noted before that after return from a function call, the caller should clean up the stack, i.e. remove the arguments it had pushed onto the stack before the call. Similarly, within the function itself there should be a cleanup, to remove the locals and make sure that EBP is adjusted properly. That means: restoring ESP and EBP to the values they had before the prologue:

```
movl %ebp, %esp
popl %ebp
```

Let's call this the "epilogue." The prologue and epilogue codes are some common that Intel included ENTER and LEAVE instructions in the chip to do them. The above prologue, a three-instruction sequence which set 8 bytes of space for the locals, would be done, for instance, by the single instruction

```
enter 8, 0
```

(The 0 is never used in flat memory mode.)

The two-instruction epilogue above can be done with the single instruction

```
leave
```

Currently GCC uses LEAVE but not ENTER. The latter actually turns out to be slower on modern machines than just using the original three-instruction sequence, so it isn't used anymore.

8.8.5 Application of Stack Frame Structure

The following code implements a "backtrace" like GDB's. A user program calls **getchain()**, which will return a list of the return addresses of all the current stack frames. It is required that the user program first call the function **initbase()**, which determines the address of **main()**'s stack frame and stores it in **base**. It is assumed that **getchain()** will not be called from **main()** itself.

The C prototype is

```
void getchain(int *chain, int *nchain)
```

with the space for the array **chain** and its length **nchain** provided by the caller.

In reading the code, keep in mind this picture of the stack after the three pushes are done:

```

    address of nchain
    address of chain
    saved return address from getchain() back to the caller
    saved EAX
    saved EBX
ESP →  saved ECX
towards 0 ↓
```

```
1  .data
2  base: # address of main()'s stack frame
3      .long 0
4
5  .text
6  .globl initbase, getchain
7  initbase: # initializes base
8      movl %ebp, base
9      ret
10 getchain:
11     # register usage:
```

```

12 # EAX will point to the current element of chain to be filled
13 # EBX will point to the frame currently being examined
14 push %eax
15 push %ebx
16 push %ecx
17 movl 16(%esp), %eax
18 # put in the return address from this subroutine, getchain() as
19 # the first element of the chain
20 movl 12(%esp), %ecx
21 movl %ecx, (%eax)
22 addl $4, %eax
23 # EBP still points to the caller's frame, perfect since we don't want
24 # to include the frame for this subroutine
25 movl %ebp, %ebx
26 top:
27 # if this is main()'s frame, then leave
28 cmpl %ebx, base
29 jz done
30 # get return address and add it to chain
31 movl 4(%ebx), %ecx
32 movl %ecx, (%eax)
33 addl $4, %eax
34 # go to next frame
35 movl (%ebx), %ebx
36 jmp top
37 done:
38 # find nchain, by subtracting start of chain from EAX and adjusting
39 subl 16(%esp), %eax
40 shrl $2, %eax
41 movl 20(%esp), %ebx
42 movl %eax, (%ebx)
43 pop %ecx
44 pop %ebx
45 pop %eax
46 ret

```

Example of calls:

```

1 int chain[5],nchain;
2
3 void h(int *w)
4 { int z;
5   int chain[10],nchain;
6   z = *w +28;
7   getchain(chain,&nchain);
8   *w = 13 * z;
9 }
10
11 int g(int u)
12 { int v;
13   h(&u);
14   v = u + 12;
15   getchain(chain,&nchain);
16   return v;
17 }
18
19 main()
20 { int x,y;
21   initbase();
22   x = 5;
23   y = g(x);
24 }

```

8.9 The LEA Instruction Family

The name of the LEA instruction family stands for Load Effective Address. Its action is to compute the memory address of the first operand, and store that address in the second.

For example, the instruction

```
leal    -4(%ebp), %eax
```

computes $-4 + c(\text{EBP})$ and places the result in EAX. If we had a single local variable **z** within a C function, the above instruction would be computing **&z** and placing it in EAX. The compiler often uses this technique.

8.10 The Function `main()` IS a Function, So It Too Has a Stack Frame

It's important to keep in mind that `main()` is a function too, so it has information stored on the stack. Recall that a typical declaration of `main()` is

```
int main(int argc, char **argv)
```

This clearly shows that `main()` is a function. Note that we've given `main()` an `int` return value, typically used as a success code, again illustrating the fact that `main()` is a function.

Let's take a closer look. When you compile your program, the compiler (actually linker) puts in some C library code which is used for startup. Just like your assembly language programs, there is a label there, `_start`, at which execution begins. The code there will prepare the stack, including the `argc` and `argv` arguments, and will then call `main()`.

Note by the way that we are not required to give these formal parameters the names `argc` and `argv`. It is merely customary. As a veteran C programmer, you know that you can name the formal parameters whatever you want. When any subroutine is called, the caller, in this case the C library startup code, neither knows nor cares what the names of the formal parameters are in the module in which the subroutine is defined.

Accordingly, upon entry to `main()`, the stack will consist of the return address, then `argc`, then `argv`, and space will be made on the stack for any local variables `main()` may have. To illustrate that, consider the code

```
1 main(int argc, char **argv)
2 { int i;
3   printf("%d %s\n",argc,argv[1]);
4 }
```

After applying `gcc -S` to this, we get

```
1      .file      "argv.c"
2      .section   .rodata
3  .LC0:
4      .string    "%d %s\n"
5      .text
6      .align 2
7  .globl main
8      .type      main,@function
9  main:
10     pushl     %ebp
```

```

11     movl    %esp, %ebp
12     subl    $8, %esp
13     andl    $-16, %esp
14     movl    $0, %eax
15     subl    %eax, %esp
16     subl    $4, %esp
17     movl    12(%ebp), %eax
18     addl    $4, %eax
19     pushl   (%eax)
20     pushl   8(%ebp)
21     pushl   $.LC0
22     call    printf
23     addl    $16, %esp
24     leave
25     ret
26 .Lfel:
27     .size   main,.Lfel-main
28     .ident  "GCC: (GNU) 3.2 20020903 (Red Hat Linux 8.0 3.2-7)"

```

We see that **argc** and **argv** are 8 and 12 bytes from EBP, respectively. This makes sense, since from Section 8.8 we know that upon entry to **main()**, the stack headed by the place pointed to by EBP will look like this:

```

argv
argc
return address
saved EBP

```


At the end of **main()**, the GCC compiler will put the return value in EAX, and will produce a LEAVE instruction.

Let's look at this from one more angle, by running this program via GDB:

```

1 % gcc -S argv.c
2 % as --gstabs -o argv.o argv.s
3 % gcc -g argv.o
4 % gdb -q a.out
5 (gdb) b 10
6 Breakpoint 1 at 0x8048328: file argv.s, line 10.
7 (gdb) r abc def
8 Starting program:
9 /fandrhome/matloff/public_html/matloff/public_html/50/PLN/a.out abc def
10
11 Breakpoint 1, main () at argv.s:10
12 10          pushl   %ebp
13 Current language:  auto; currently asm
14 (gdb) x/3x $esp
15 0xbfffe33c:  0x42015967      0x00000003      0xbfffe384
16 (gdb) x/3x 0xbfffe384
17 0xbfffe384:  0xbfffe95c      0xbfffe99c      0xbfffe9a0
18 (gdb) x/s 0xbfffe95c
19 0xbfffe95c:
20 "/fandrhome/matloff/public_html/matloff/public_html/50/PLN/a.out"
21 (gdb) x/s 0xbfffe99c
22 0xbfffe99c:  "abc"
23 (gdb) x/s 0xbfffe9a0
24 0xbfffe9a0:  "def"

```

So we see that the return address to the C library is 0x42015967, **argc** is 3 and **argv** is 0xbfffe384. The latter should be a pointer to an array of three strings, which is confirmed in the subsequent GDB commands. 

By the way, the C library code provides a third argument to **main()** as well, which is a pointer to the environment variables (current working directory, executable search path, username, etc.). Then the declaration would be

```
int main(int argc, char **argv, char **envp)
```

8.11 Once Again, There Are No Types at the Hardware Level!

You have been schooled—properly so—in your beginning programming classes about the importance of **scope**, meaning which variables are accessible or inaccessible from which parts of a program. But if a variable is inaccessible from a certain point in a program, that is merely the compiler doing its gatekeeper work for that language. It is NOT a restriction by the hardware. There is no such thing as scope at the hardware level. ANY instruction in a program—remember, all C/C++ and assembly code gets translated to machine instructions—can access ANY data item ANYWHERE in the program.

In C++, you were probably taught a slogan of the form

“A **private** member of a class cannot be accessed from anywhere outside the class.”

But that is not true. The correct form of the statement should be

“The compiler *will refuse to compile* any C++ code you write which attempts to access *by name* a **private** member of a class from anywhere outside the class.”

Again, the gatekeeper here is the compiler, not the hardware. The compiler’s actions are desirable, because the notion of scope helps us to organize our data, but it has no physical meaning. Consider this example:

```
1  #include <iostream.h>
2
3  class c {
4      private:
5          int x;
6      public:
7          c(); // constructor
8          // printx() prints out x
9          void printx() { cout << x << endl; }
10 };
11
12 c::c()
13 { x = 5; }
14
15 int main(int argc, char *argv[])
16 { c ci;
17   ci.printx(); // prints 5
18   // now point p to ci, thus to the first word at ci, i.e. x
19   int *p = (int *) &ci;
20   *p = 29; // change x to 29
21   ci.printx(); // prints 29
22 }
```

The point is that the member variable **x** in the class **c**, though **private**, is nevertheless in memory, and we can get to any memory location by setting a pointer variable to that location. That is what we have done in this example. We’ve managed to change the value of **x** in an instance of the class through code which is outside the class.

The point also applies to local variables. We can actually access a local variable in one function from code in another function. The following example demonstrates that (make sure to review Section 8.8 before reading the example):

```

1 void g()
2 { int i,*p;
3   p = &i;
4   p = p + 1; // p now points to main()'s EBP
5   p = *p; // p now equals main()'s EBP value
6   p = p - 1; // p now points to main()'s x
7   *p = 29; // changes x to 29
8 }
9
10 main()
11 { int x = 5; // x is local to main()
12   g();
13   printf("%d\n",x); // prints out 29
14 }

```

8.12 What About C++?

If the calling program is C++ instead of C, you must inform the compiler that the assembly language routine is “C style,” by inserting

```
extern "C" void addone(int *);
```

in the C++ source file. You need to do this, because your assembly language routine will be in the C style, i.e. utilize C conventions such as that concerning EBP above.

For **instance** (i.e. non-**static**) functions, note that the **this** pointer is essentially an argument too. The convention is that it is pushed last, i.e. it is treated as the first argument.

8.13 Putting It All Together

To illustrate a number of the concepts we’ve covered here, let’s look at the full assembly code produced from the function **sum()** in Section 8.7. Here is the original C and then the compiled code:

```

1 int sum(int *x, int n)
2 { int i=0,s=0;
3
4   for (i = 0; i < n; i++)
5     s += x[i];
6   return s;
7 }

```

```

1      .file   "sum.c"
2      .text
3      .align 2
4 .globl sum
5      .type   sum,@function
6 sum:
7      pushl  %ebp
8      movl   %esp, %ebp
9      subl   $8, %esp
10     movl   $0, -4(%ebp)
11     movl   $0, -8(%ebp)
12     movl   $0, -4(%ebp)
13 .L2:
14     movl   -4(%ebp), %eax
15     cmpl  12(%ebp), %eax

```

```

16         jl     .L5
17         jmp     .L3
18  .L5:
19         movl   -4(%ebp), %eax
20         leal  0(,%eax,4), %edx
21         movl   8(%ebp), %eax
22         movl   (%eax,%edx), %edx
23         leal  -8(%ebp), %eax
24         addl  %edx, (%eax)
25         leal  -4(%ebp), %eax
26         incl  (%eax)
27         jmp     .L2
28  .L3:
29         movl   -8(%ebp), %eax
30         leave
31         ret
32  .Lfe1:
33         .size  sum, .Lfe1-sum
34         .ident "GCC: (GNU) 3.2 20020903 (Red Hat Linux 8.0 3.2-7)"

```

We see that the compiler has first placed the standard prologue at the beginning of the code, allowing for two local variables:

```

1  sum:
2      pushl   %ebp
3      movl   %esp, %ebp
4      subl   $8, %esp

```

It then initializes both of those locals to 0.

```

1      movl   $0, -4(%ebp)
2      movl   $0, -8(%ebp)

```

Since our code sets **i** to 0 twice, the compiler does so too, since we didn't ask the compiler to optimize.

Our **for** loop compares **i** to **n**, which is done here:

```

1      movl   -4(%ebp), %eax
2      cmpl  12(%ebp), %eax
3      jl     .L5
4      jmp     .L3

```

The code must add **x[i]** to the sum:

```

1  .L5:
2      movl   -4(%ebp), %eax
3      leal  0(,%eax,4), %edx
4      movl   8(%ebp), %eax
5      movl   (%eax,%edx), %edx
6      leal  -8(%ebp), %eax
7      addl  %edx, (%eax)

```

Note the use of the LEA instruction and advanced addressing modes.

We need to increment **i** and go to the top of the loop:

```

1      leal  -4(%ebp), %eax
2      incl  (%eax)
3      jmp     .L2

```

When the function is finished, it needs to put the sum in EAX, as described in Section 8.5, clean up the stack and return:

```
1 .L3:
2     movl    -8(%ebp), %eax
3     leave
4     ret
```

Note the use of the LEAVE instruction.

Now, what if we declare that local variable `s` as **static**?

```
1 int sum(int *x, int n)
2 { int i=0; static s=0;
3
4   for (i = 0; i < n; i++)
5     s += x[i];
6   return s;
7 }
```

Recall that this means that the variable will retain its value from one call to the next.¹² That means that the compiler can't use the stack for storage of this variable, as it would likely get overwritten by calls from other subroutines. So, it must be stored in a **.data** section:

```
1     .file      "Sum.c"
2     .data
3     .align 4
4     .type      s.0,@object
5     .size      s.0,4
6 s.0:
7     .long      0
8     .text
9     .align 2
10    .globl sum
11    .type      sum,@function
12 sum:
13    pushl     %ebp
14    movl     %esp, %ebp
15    subl     $4, %esp
16    movl     $0, -4(%ebp)
17    movl     $0, -4(%ebp)
18 .L2:
19    movl     -4(%ebp), %eax
20    cmpl     12(%ebp), %eax
21    jnl     .L5
22    jmp      .L3
23 .L5:
24    movl     -4(%ebp), %eax
25    leal     0(,%eax,4), %edx
26    movl     8(%ebp), %eax
27    movl     (%eax,%edx), %eax
28    addl     %eax, s.0
29    leal     -4(%ebp), %eax
30    incl     (%eax)
31    jmp      .L2
32 .L3:
33    movl     s.0, %eax
34    leave
35    ret
36 .Lfel:
37    .size     sum,.Lfel-sum
38    .ident   "GCC: (GNU) 3.2 20020903 (Red Hat Linux 8.0 3.2-7)"
```

¹²Our code initializes `s` to 0, but this is done only once, as you can see from the assembly language here.

You can tell from the code

```
movl    s.0, %eax
leave
ret
```

that our C variable `s` is being stored at a label `s.0` in the `.data` section.

9 Subroutine Calls>Returns Are “Expensive”

On most machines, subroutine calls and returns add overhead. Not only does it mean that extra instructions must be executed, but also stack access, being memory access, is slow. So, it makes sense to ask how CPUs could be designed to speed this up.

One approach would be to make a special cache for the stack. Note that since the stack is in memory, some of it may be in an ordinary cache, but by devoting a special cache to the stack, we would decrease the cache miss rate for the stack.

But a more aggressive approach would be to design the CPU so that the top few stack elements are in the CPU itself. This is done in Sun Microsystems’ SPARC chip, and was done back in the 60s and 70s on Burroughs mainframes. This would be better than a special cache, since the latter, with its complex circuitry for miss processing and so on, would have more delay.

On the other hand, space on a chip is precious. It may be that such usage of the space may not be as good as some other usage.

10 Debugging Assembly Language Subroutines

10.1 Focus on the Stack

At first, debugging assembly language subroutines would seem to be similar to debugging assembly language in general, and for that matter, debugging any kinds of code. However, what makes the assembly-language subroutine case special is the use of the stack. Many bugs involve errors in accessing the stack, such as failure to restore a register value which had been saved on the stack.

Recall that the essence of good debugging is confirmation. One uses the debugger, say **ddd**, to step through one’s code line by line, and in each line confirms that our stored data (variables in C/C++, register and memory values in assembly language) have the values we expect them to. In stepping through our code, we also confirm that we execute the lines we expect to be executed, such as a conditional branch being taken when we expect it to be taken. Eventually this process will lead us to a line in which we find that our confirmation fails. This then will pinpoint where our bug is, and we can start to analyze what is wrong with that line.

For assembly language subroutines, our confirmation process consists of confirming that the contents of the stack are exactly as we expect them to be. Eventually we will find a line of code at which that confirmation fails, and we then will have pinpointed the location of our bug, and we can start to analyze what is wrong with that line.

To do that, you have to know how to inspect the stack from within your debugger, e.g. **ddd**. The key point is that the stack is part of memory, and the debugger allows you to inspect memory. So, first find out from

the debugger the current value of ESP, and then inspect a few words of memory starting at wherever ESP points to, e.g.

```
x/4w $esp
```

to inspect the first four words of the stack in GDB.¹³

Note that if during a debugging session you run your program several times without exiting GDB,¹⁴ the stack will keep growing. Thus any stack addresses which you jot down for later use may become incorrect.

10.2 A Special Consideration When Interfacing C/C++ with Assembly Language

When you are interfacing C/C++ to assembly language and while debugging you reach a function call in your C/C++ code, it is often useful to view the assembly language, especially the addresses of the instructions. You may find GDB's **disassemble** command useful in this regard. It reverse-translates the machine code for the program into assembly language (regardless of whether the program source was originally assembly language or C/C++), and lists the assembly language instructions and their addresses. Note that the latter will be addresses with respect to the linker's assignments of the **.text** section location, not offsets as in the output of **as -a**. Also note that **.data** section symbols won't appear; the actual addresses will show up.

The **disassemble** command has several formats. If you run it without arguments, it will disassemble the code near your current location. You can also run it on any label in the **.text** section, e.g.

```
(gdb) disas _start
```

and

```
(gdb) disas main
```

In the latter case, where we have a function name, the entire function will be disassembled.

You can also specify a range of memory addresses, e.g.

```
(gdb) disas 0x08048083 0x08048099
```

11 Macros

A **macro** is like a subroutine in the sense that it makes one's programming more modular, but it actually is **inline code**, meaning that it produces code at the place where it is invoked.

To see what this means, consider this example from an earlier unit:

```
1 .data
2 x:
3     .long 1
```

¹³You can now see an advantage of designing the hardware so that stacks grow toward 0. This causes consecutive elements of the stack to be in consecutive locations, making it easier to inspect the stack.

¹⁴And this is the proper way to do it. You SHOULD stay in GDB from one run to the next, so as to retain your breakpoints etc.

```

4      .long  5
5      .long  2
6      .long  18
7  sum:
8      .long  0
9  .text
10 .globl _start
11 _start:
12     movl $4, %eax
13     movl $0, %ebx
14     movl $x, %ecx
15 top:  addl (%ecx), %ebx
16     addl $4, %ecx
17     decl %eax
18     jnz top
19 done: movl %ebx, sum

```

Those first three lines of executable code,

```

    movl $4, %eax
    movl $0, %ebx
    movl $x, %ecx

```

perform various initializations. To make our program more modular, we could make a subroutine out of them, say as **init()** in Section 7 (either with or without the argument). This would be fine, but it would slow down the execution of the program a bit, since the CALL and RET instructions would use some clock cycles and their stack access, being memory accesses, are slow. (And there is further slowdown if **init()** uses an argument).

An alternative would be to convert those three instructions to a macro:

```

1  .macro init
2      movl $4, %eax
3      movl $0, %ebx
4      movl $x, %ecx
5  .endm
6
7  .data
8  x:
9      .long  1
10     .long  5
11     .long  2
12     .long  18
13 sum:
14     .long  0
15
16 .text
17 .globl _start
18 _start:
19     init
20 top:  addl (%ecx), %ebx
21     addl $4, %ecx
22     decl %eax
23     jnz top
24 done: movl %ebx, sum

```

Though the “init” at **_start** has the look of a subroutine call, it really isn’t. Instead, when the assembler sees this line at **_start**, it replaces that line with the three lines of code defined for “init” at the top of the file.

In other words, the macro version of the program will produce exactly the same machine code as did the original nonmodular version. We can see this by running both files through **as -a**; here are excerpts from the outputs of this command from the original and macro versions of the program:

```

...
23          _start:
24 0000 B8040000      movl $4, %eax
24          00
25 0005 BB000000      movl $0, %ebx
25          00
26 000a B9000000      movl $x, %ecx
26          00
27 000f 0319          top:  addl (%ecx), %ebx
...

```

```

...
20          _start:
21 0000 B8040000      init
21          00BB0000
21          0000B900
21          000000
22 000f 0319          top:  addl (%ecx), %ebx
...

```

So, both versions of the program would produce the same `.o` file. By contrast, the subroutine version produces different code:

```

...
23          _start:
24 0000 E80E0000      call init
24          00
25 0005 0319          top:  addl (%ecx), %ebx
...
31          init:
32 0013 B8040000      movl $4, %eax
32          00
33 0018 BB000000      movl $0, %ebx
33          00
34 001d B9000000      movl $x, %ecx
34          00
35 0022 C3            ret

```

Note the machine code for the `call` instruction and the `ret`. So, by using a macro we get the efficiency of the nonmodular version while still getting a modular view for the programmer.¹⁵

Macros allow arguments too. Within a macro itself, the macro's arguments are rereferred to using backslashes.

For example, the macro `yyy` has two arguments, `u` and `v`:

```

.macro yyy u,v
    movl \u,%eax
    addl \v,%eax
.endm

```

So the call

```
yyy $5,$12
```

will produce the same code as we had written

```
movl $5, %eax
addl $12, %eax
```

¹⁵Though this modular view will not appear in a debugging tool, a drawback.