Chapter 9
Functions

In this chapter we will discuss how to write assembly functions which can be called from C or C++ and how to call C functions from assembly. Since the C or C++ compiler generally does a very good job of code generation, it is usually not important to write complete programs in assembly. There might be a few algorithms which are best done in assembly, so we might write 90% of a program in C or C++ and write a few functions in assembly language.

It is also useful to call C functions from assembly. This gives your assembly programs full access to all C libraries. We will use `scanf` to input values from `stdin` and we will use `printf` to print results. This will allow us to write more useful programs.

9.1 The stack

So far we have had little use for the run-time stack, but it is an integral part of using functions. We stated earlier that the stack extends to the highest possible address: `0x7fffffffffff`. This is not quite true. Inspection of the memory map using "`cat /proc/$$/maps`" shows the top stack address is `0x7fffa6b79000` for my `bash` process and different values for other processes always matching the pattern `0x7fffXXXXX000`. Perhaps this is a result of "stack randomization" which is an attempt to avoid rogue code which modifies stack values.

Items are pushed onto the stack using the `push` instruction. The effect of `push` is to subtract 8 from the stack pointer `rsp` and then place the value being pushed at that address. Initially the stack pointer would be set to `0x7ffffffff000` (or some address ending in `000`) by the operating system when a process is started. On the first `push`, `rsp` would be decreased to `0x7fffffffeff8` and an 8 byte value would be placed in bytes `0x7fffffffeff8` through `0x7fffffffefff`.

Many different values are pushed onto the stack by the operating system. These include the environment (a collection of variable names and values defining things like the search path) and the command line parameters for the program.

Values can be removed from the stack using the `pop` instruction. `pop` operates in the reverse pattern of `push`. It moves the value at the location specified by the stack pointer (`rsp`) to a register or memory location and then adds 8 to `rsp`.

You can push and pop smaller values than 8 bytes, at some peril. It works as long as the stack remains bounded appropriately for the current operation. So if you push a word and then push a quad-word, the quad-word push may fail. It is simpler to push and pop only 8 byte quantities.

9.2 Call instruction

The assembly instruction to call a function is `call`. A typical use would be like

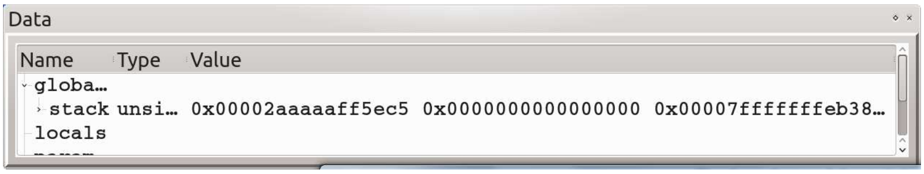
```
        call    my_function
```

The operand `my_function` is a label in the text segment of a program. The effect of the `call` instruction is to push the address of the instruction following the call onto the stack and to transfer control to the address associated with `my_function`. The address pushed onto the stack is called the "return address". Another way to implement a call would be

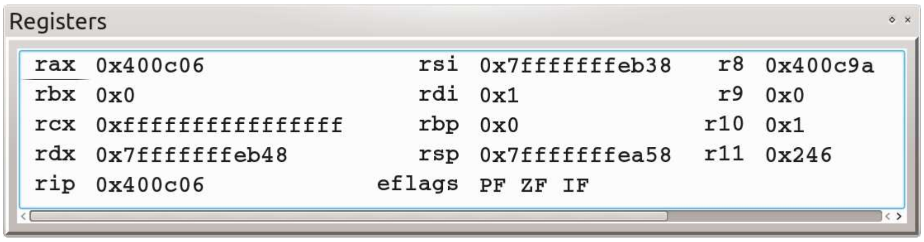
```
        push    next_instruction
        jmp     my_function
next_instruction:
```

While this does work, the `call` instruction has more capability which we will generally ignore.

Ebe shows the top of the stack (normally 6 values) as your program executes. Below are the top 3 quad-words on the stack upon entry to `main` in an assembly program. Immediately preceding this register display was a `call` instruction to call `main`.



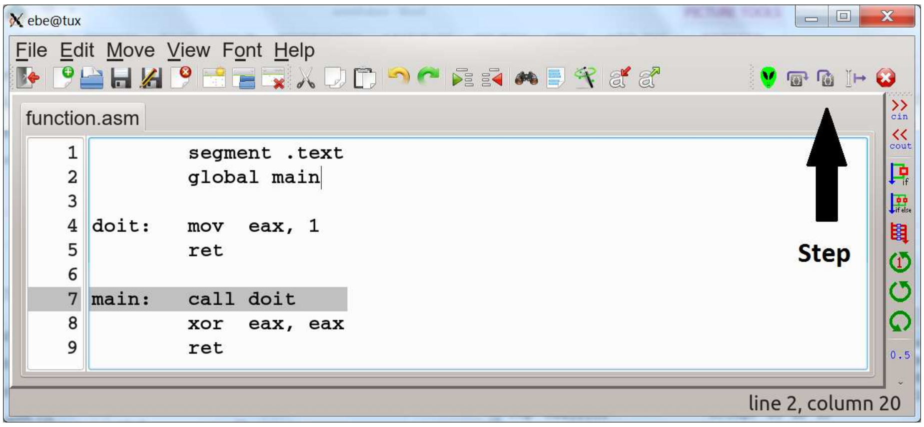
The first item on the stack is the return address, `0x2aaaaaff5ec5`. Normal text segment addresses tend to be a little past `0x400000` in Linux programs as illustrated by `rip` in the register display below taken from the same program when it enters `main`. The return address is an address in a shared object library, probably in the function `__libc_start_main` in libc.so. The same pattern occurs with OS X though the code addresses are larger numbers (bit 32 is set with addresses like `0x100400c06`).



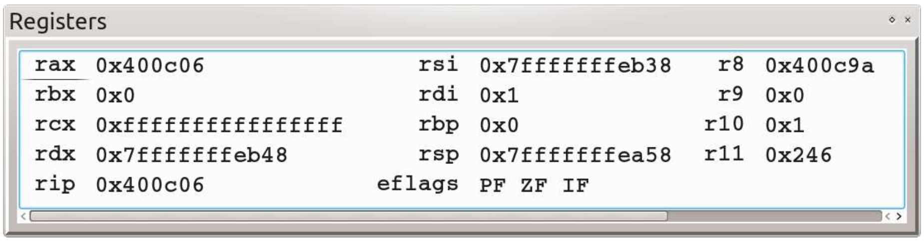
9.3 Return instruction

To return from a function you use the `ret` instruction. This instruction pops the address from the top of the stack and transfers control to that address. In the previous example `next_instruction` is the label for the return address.

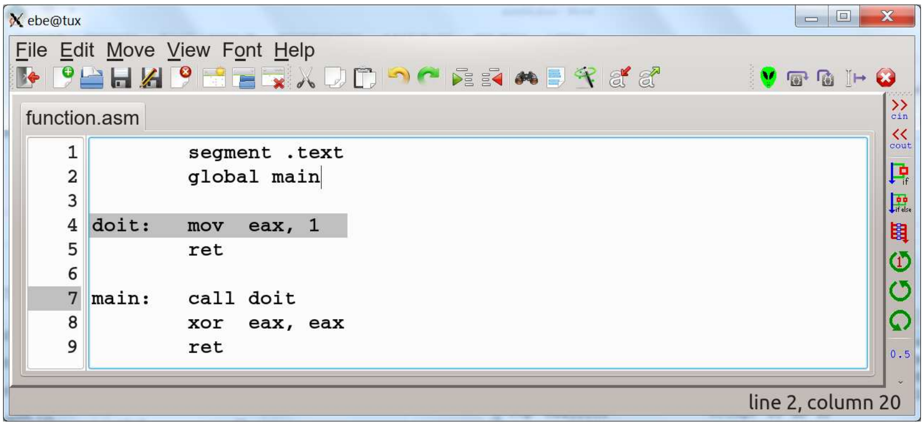
Below is shown a very simple program which illustrates the steps of a function call and return. The first instruction in `main` is a call to the `doit` function.



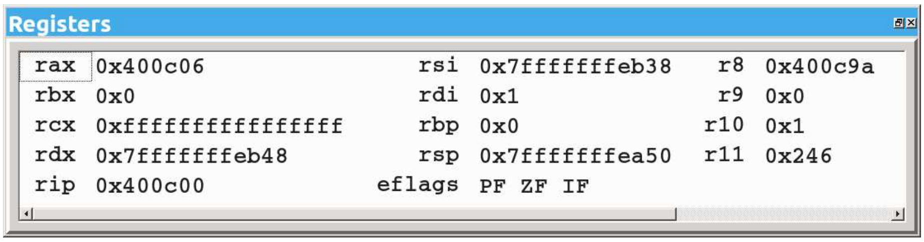
You can see that there is a breakpoint on line 7 and the call to `doit` has not yet been made. I have added an arrow pointing to the "Step" button which is immediately to the right of the "Next" button. In the register display below you can see that `rip` is `0x400c06`.



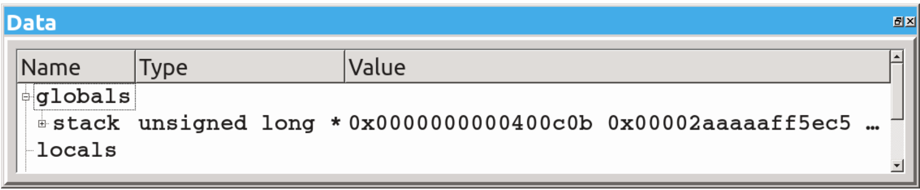
Previously we have used the “Next” button to execute the current instruction. However, if we use “Next” now, the debugger will execute the `doit` call and control will be returned after the function returns and the highlighted line will be line 8. In order to study the function call, I have clicked on “Step” which will step into the `doit` function.



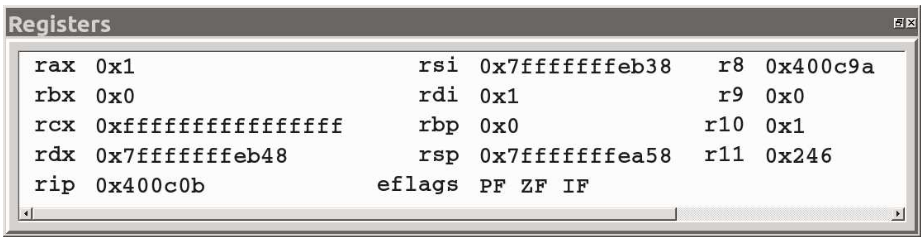
Now we see that the next instruction to execute is on line 4. It is instructive to view the registers at this point and the stack.



You can see that `rip` is now `4004c0` which is at a lower address than the call at line 7.



From the variable display we see that the first item on the stack is 4004cb which is the return address. After using "Step" two more times the debugger executes the return from doit. Below are the registers after executing the return.



Here we see that rip is now 0x4004cb which was the value placed on the stack by the call to doit.

9.4 Function parameters and return value

Most function have parameters which might be integer values, floating point values, addresses of data values, addresses of arrays, or any other type of data or address. The parameters allow us to use a function to operate on different data with each call. In addition most functions have a return value which is commonly an indicator of success or failure.

X86-64 Linux uses a function call protocol called the "System V Application Binary Interface" or System V ABI. The Apple Mac OS/X using x86-64 processing mode also uses the System V ABI, so the information in this book applies to Linux and to the Mac. Unfortunately Windows uses a different protocol called the "Microsoft x64 Calling Convention". In both protocols some of the parameters to functions are passed in registers. Linux allows the first 6 integer parameters to be passed in registers, while Windows allows the first 4 (using different registers). Linux and OS X allow the first 8 floating point parameters to be passed in floating pointer registers xmm0-xmm7, while Windows allows the first 4 floating point parameters to be passed in registers xmm0-xmm3.

Linux, OS X and Windows use register rax for integer return values and register xmm0 for floating point return values.

Both Linux and Windows expect the stack pointer to be maintained on 16 byte boundaries in memory. This means that the hexadecimal value for rsp should end in 0. The reason for this requirement is to allow local variables in functions to be placed at 16 byte alignments for SSE and AVX instructions. Executing a call would then decrement rsp leaving it ending with an 8. Conforming functions should either push something or subtract from rsp to get it back on a 16 byte boundary. It is common for a function to push rbp as part of establishing a stack frame which re-establishes the 16 byte boundary for the stack. If your function calls any external function, it seems wise to stick with the 16 byte bounding requirement.

The first 6 integer parameters in a function under Linux and OS X are passed in registers rdi, rsi, rdx, rcx, r8 and r9, while Windows uses rcx, rdx, r8 and r9 for the first 4 integer parameters. If a function requires more parameters, they are pushed onto the stack in reverse order.

Functions like scanf and printf which have a variable number of parameters pass the number of floating point parameters in the function call using the rax register.

For 32 bit programs the protocol is different. Registers r8-r15 are not available, so there is not much value in passing function parameters in registers. These programs use the stack for all parameters.

We are finally ready for “Hello World!”

```
     section .data
msg: db    “Hello World!”,0x0a,0
     section .text
     global  main
     extern  printf
main:
     push  rbp
     mov   rbp, rsp
     lea   rdi, [msg] ; parameter 1 for printf
     xor   eax, eax   ; 0 floating point parameters
     call  printf
     xor   eax, eax   ; return 0
     pop   rbp
     ret
```

We use the “load effective address” instruction (lea) to load the effective address of the message to print with printf into rdi. This could also be done with mov, but lea allows specifying more items in the brackets so that we could load the address of an array element. Furthermore, under OS X mov will not allow you to move an address into a register. There the problem is that static addresses for data have values which exceed the capacity of 32 bit pointers and the constant field of the mov instruction is 32 bits. The easy assessment is to use lea to load addresses.

Interestingly when the system starts a program in _start (or simply start in OS X) the parameters to start are pushed onto the stack. However, the parameters to main are in registers like any other C function.

9.5 Stack frames

One of the most useful features of the gdb debugger is the ability to trace backwards through the stack functions which have been called using command `bt` or `backtrace`. To perform this trick each function must keep a pointer in `rbp` to a 2 quad-word object on the stack identifying the previous value of `rbp` along with the return address. You might notice the sequence "`push rbp; mov rbp, rsp`" in the hello world program. The first instruction pushes `rbp` immediately below the return address. The second instruction makes `rbp` point to that object.

Assuming all functions obey this rule of starting with the standard 2 instructions, there will be a linked list of objects on the stack - one for each function invocation. The debugger can traverse through the list to identify the function (based on the location of the return address) called and use other information stored in the executable to identify the line number for this return address.

These 2 quad-word objects are simple examples of "stack frames". In functions which do not call other functions (leaf functions), the local variables for the function might all fit in registers. If there are too many local variables or if the function calls other functions, then there might need to be some space on the stack for these local variables. To allocate space for the local variables, you simply subtract from `rsp`. For example to leave 32 bytes for local variables in the stack frame do this:

```
push    rbp
mov     rbp, rsp
sub     rsp, 32
```

Be sure to subtract a multiple of 16 bytes to avoid possible problems with stack alignment.

To establish a stack frame, you use the following 2 instructions at the start of a function:

```
push    rbp
mov     rbp, rsp
```

The effect of the these 2 instructions and a possible subtraction from `rsp` can be undone using

```
leave
```

just before a `ret` instruction. For a leaf function there is no need to do the standard 2 instruction prologue and no need for the `leave` instruction. They can also be omitted in general though it will prevent gdb from being able to trace backwards though the stack frames.

When you have local variables in the stack frame it makes sense to access these variables using names rather than adding 8 or 16 to `rsp`. This can be done by using yasm's `equ` pseudo-op. The following sets up symbolic names for 0 and 8 for two local variables.

```
x       equ     0
y       equ     8
```

Now we can easily save 2 registers in `x` and `y` prior to a function call using

```
        mov     [rsp+x], r8
        mov     [rsp+y], r9
```

With any function protocol you must specify which registers must be preserved in a function. For the System V ABI (Linux and OS X), registers `rbx`, `rbp` and `r12-15` must be preserved, while the Windows calling convention requires that registers `rbx`, `rbp`, `rsi`, `rdi` and `r12-15` must be preserved.

Function to print the maximum of 2 integers

The program listed below calls a function named `print_max` to print the maximum of 2 longs passed as parameters. It calls `printf` so it uses the `extern` pseudo-op to inform yasm and ld that `printf` will be loaded from a library.

```
    segment .text
    global  main
    extern  printf

; void print_max ( long a, long b )
; {
a   equ  0
b   equ  8
print_max:
    push rbp;         ; normal stack frame
    mov  rbp, rsp
    sub  rsp, 32      ; leave space for a, b and max
;   int max;
max equ  16
    mov  [rsp+a], rdi ; save a
    mov  [rsp+b], rsi ; save b
;   max = a;
    mov  [rsp+max], rdi
;   if ( b > max ) max = b;
    cmp  rsi, rdi
    jng  skip
    mov  [rsp+max], rsi
skip:
;   printf ( "max(%ld,%ld) = %ldn", a, b, max );
    segment .data
fmt db   'max(%ld,%ld) = %ld',0xa,0
    segment .text
    lea  rdi, [fmt]
    mov  rsi, [rsp+a]
    mov  rdx, [rsp+b]
    mov  rcx, [rsp+max]
    xor  eax, eax     ; 0 floating point parameters
```

```
        call printf
  ; }
        leave
        ret

main:
        push rbp
        mov  rbp, rsp
  ;     print_max ( 100, 200 );
        mov  rdi, 100    ; first parameter
        mov  rsi, 200    ; second parameter
        call print_max
        xor  eax, eax    ; to return 0
        leave
        ret
```

In main you first see the standard 2 instructions to establish a stack frame. There are no local variables in main, so there is no need to subtract anything from rsp. On the other hand the print_max function has 2 parameters and 1 local variable. The required space is 24 bytes, which is rounded up the next multiple of 16. It would be possible avoid storing these variables in memory, but it would be more confusing and less informative.

Immediately after the comment for the heading for print_max, I have 2 equates to establish offsets on the stack for a and b. After the comment for the declaration for max, I have an equate for it too.

Before doing any of the work of print_max I have 2 mov instructions to save a and b onto the stack. Both variables will be parameters to the printf call, but they will be the second and third parameters so they will need to be different registers at that point.

The computation for max is done using the stack location for max rather than using a register. It would have been possible to use rcx which is the register for max in the printf call, but would be less clear and the goal of this code is to show how to handle parameters and local variables to functions simply.

The call to printf requires a format string which should be in the data segment. It would be possible to have a collection of data prior to the text segment for the program, but it is nice to have the definition of the format string close to where it is used. It is possible to switch back and forth between the text and data segments, which seems easier to maintain.

9.6 Recursion

One of the fundamental problem solving techniques in computer programming is recursion. A recursive function is a function which calls itself. The focus of recursion is to break a problem into smaller problems.

Frequently these smaller problems can be solved by the same function. So you break the problem into smaller problems repeatedly and eventually you reach such a small problem that it is easy to solve. The easy to solve problem is called a "base case". Recursive functions typically start by testing to see if you have reached the base case or not. If you have reached the base case, then you prepare the easy solution. If not you break the problem into sub-problems and make recursive calls. As you return from recursive calls you assemble solutions to larger problems from solutions to smaller problems.

Recursive functions generally require stack frames with local variable storage for each stack frame. Using the complete stack frame protocol can help in debugging.

Using the function call protocol it is easy enough to write recursive functions. As usual, recursive functions test for a base case prior to making a recursive call.

The factorial function can be defined recursively as

$$f(n) = \begin{cases} 1 & \text{if } n \le 1 \\ n * f(n-1) & \text{if } n < 1 \end{cases}$$

Here is a program to read an integer n, compute n! recursively and print n!.

```
    segment .data
x   dq      0
scanf_format:
    db     "%ld",0
printf_format:
    db     "fact(%ld) = %ld",0x0a,0

    segment .text
    global  main        ; tell world about main
    global  fact        ; tell world about fact
    extern  scanf       ; resolve scanf and
    extern  printf      ; printf from libc
main:
    push    rbp
    mov     rbp, rsp
    lea     rdi, [scanf_format] ; set arg 1
    lea     rsi, [x]    ; set arg 2 for scanf
    xor     eax, eax    ; set rax to 0
    call    scanf
    mov     rdi, [x]    ; move x for fact call
    call    fact
    lea     rdi, [printf_format]; set arg 1
    mov     rsi, [x]    ; set arg 2 for printf
    mov     rdx, rax    ; set arg 3 to be x!
    xor     eax, eax    ; set rax to 0
    call    printf
    xor     eax, eax    ; set return value to 0
    leave
    ret

fact:                   ; recursive function
n   equ     8
    push    rbp
```

```
    mov     rbp, rsp
    sub     rsp, 16
    ; make room for n
    cmp     rdi, 1      ; compare n with 1
    jg      greater     ; if n <= 1, return 1
    mov     eax, 1      ; set return value to 1
    leave
    ret
greater:
    mov     [rsp+n], rdi; save n
    dec     rdi         ; call fact with n-1
    call    fact
    mov     rdi, [rsp+n]; restore original n
    imul    rax, rdi    ; multiply fact(n-1)*n
    leave
    ret
```

You will notice that I have set `rax` prior to calling `scanf` and `printf`. The value of `rax` is the number of floating point parameters when you make a call to a function with a variable number of parameters.

In the `fact` function I have used an equate for the variable `n`. The `equ` statement defines the label `n` to have the value 8. In the body of the function I save the value of `n` on the stack prior to making a recursive call. The reference `[rsp+n]` is equivalent to `[rsp+8]`, but it allows more flexibility in coding while being clearer.

Exercises

1. Write an assembly program to produce a billing report for an electric company. It should read a series of customer records using scanf and print one output line per customer giving the customer details and the amount of the bill. The customer data will consist of a name (up to 64 characters not including the terminal 0) and a number of kilowatt hours per customer. The number of kilowatt hours is an integer. The cost for a customer will be \$20.00 if the number of kilowatt hours is less than or equal to 1000 or \$20.00 plus 1 cent per kilowatt hour over 1000 if the usage is greater than 1000. Use quotient and remainder after dividing by 100 to print the amounts as normal dollars and cents. Write and use a function to compute the bill amount (in pennies).

2. Write an assembly program to generate an array of random integers (by calling the C library function `random`), to sort the array using a bubble sort function and to print the array. The array should be stored in the `.bss` segment and does not need to be dynamically allocated. The number of elements to fill, sort and print should be stored in a memory location. Write a function to loop through the array elements filling the array with random integers. Write a function to print the array contents. If the array size is less than or equal to 20, call your print function before and after printing.

3. A Pythagorean triple is a set of three integers, a, b and c, such that $a^2 + b^2 = c^2$. Write an assembly program to print all the Pythagorean triples where $c \ <= \ 500$. Use a function to test whether a number is a Pythagorean triple.

4. Write an assembly program to keep track of 10 sets of size 1000000. Your program should read accept the following commands: "`add`", "`union`", "`print`" and "`quit`". The program should have a function to read the command string and determine which it is and return 0, 1, 2 or 3 depending on the string read. After reading "`add`" your program should read a set number from 0 to 9 and an element number from 0 to 999999 and insert the element into the proper set. You need to have a function to add an element to a set. After reading "`union`" your program should read 2 set numbers and make the first set be equal to the union of the 2 sets. You need a set union function. After reading "`print`" your program should print all the elements of the set. You

can assume that the set has only a few elements. After reading "`quit`" your program should exit.

5. A sequence of numbers is called bitonic if it consists of an increasing sequence followed by a decreasing sequence or if the sequence can be rotated until it consists of an increasing sequence followed by a decreasing sequence. Write an assembly program to read a sequence of integers into an array and print out whether the sequence is bitonic or not. The maximum number of elements in the array should be 100. You need to write 2 functions: one to read the numbers into the array and a second to determine whether the sequence is bitonic. Your bitonic test should not actually rotate the array.

6. Write an assembly program to read two 8 byte integers with `scanf` and compute their greatest common divisor using Euclid's algorithm, which is based on the recursive definition

$$\gcd(a,b) = \begin{cases} a & \text{if } b = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

7. Write an assembly program to read a string of left and right parentheses and determine whether the string contains a balanced set of parentheses. You can read the string with `scanf` using "`%79s`" into a character array of length 80. A set of parentheses is balanced if it is the empty string or if it consists of a left parenthesis followed by a sequence of balanced sets and a right parenthesis. Here's an example of a balanced set of parentheses: "`((()())())`".