Intel Assembly Programming Calling Convention

To allow separate programmers to share code and develop libraries for use by many programs, and to simplify the use of subroutines in general, programmers typically adopt a common *calling convention*. The calling convention is a strict protocol about how to **call** and **return** from subroutines. For example, given a set of calling convention rules, a programmer need not examine the definition of a subroutine to determine how parameters should be passed to that subroutine. Furthermore, given a strict set of calling convention rules, high-level language compilers can be made to follow the rules, thus allowing assembly language routines and high-level language routines to call one another.

In practice, many calling conventions are possible. Let's examine the widely used **C**-language calling convention. Strictly adhering to this convention will allow you to write assembly language subroutines that are **safely callable** from **C** (and **C++**) code, and will also enable you to call **C** library functions from your assembly language code.

The **C** calling convention is based heavily on the use of the hardware-supported program stack. It relies on the **push, pop, call,** and **ret** instructions. Parameters are passed to the subroutine by placing them on the stack. Any register to be used as a local variable internally by subroutines must have their contents first saved to the stack (to protect the caller's value).

The calling convention is broken into two sets of rules. The first set of rules is employed by the **caller** of the subroutine, and the second set of rules is observed by the subroutine itself (the **callee**). It should be emphasized that a single mistake in the observance of these rules quickly result in **fatal program errors** since the stack will be left in an inconsistent state; thus meticulous care should be used when implementing the calling convention in your own subroutines.

A good way to visualize the operation of the calling convention is to draw the contents of the stack during subroutine execution. The diagram (**Unix Program Stack**) provided depicts the contents of the stack during the execution of a subroutine. Let's assume an example of three parameters and using three local variables. The cells depicted in the stack are 64-bit memory locations, thus the memory addresses of the cells are 64 bits (**8** bytes) apart. The **stack** pointer (**RSP**) is initially set by the OS. The first parameter resides at an offset of 16 bytes from the **base** pointer (**RBP**). Above (,in the positive direction depicted below the base pointer) are the parameters on the stack used by the **call.** They are placed below the return address, thus the first parameter is an extra **8** bytes away from the base pointer (**RBP**). When the **ret** instruction is used to return from the called subroutine back to the caller routine, it will

branch (jump) to the return address that had been stored in that specific stack location **(RSP+ 8).**

Caller Rules

To pass control to a subroutine, the caller should:

1) Before calling a subroutine, the **caller** should save the contents of any registers that the subroutine will use to pass back a return value. The remaining registers become the responsibility of the **called** subroutine to save, since it may modify them as local variables. The subroutine must save the values in these registers by pushing them onto the stack (so they can be **restored just prior** to the subroutine returning control to the caller.

2) To pass parameters to the subroutine, push them onto the stack before the call. The parameters should be pushed in **reverse order** (i.e. last parameter first). The **top** of the stack is pointed to by the **RSP** register. Since the stack grows down (in the positive direction through greater memory locations), the first parameter will be stored at the lowest address. (This reverse order of parameters was historically used to allow functions to be passed a variable number of parameters).

3) To call the subroutine, use the **call** instruction. The call instruction places the return address on top of the parameters on the stack, and then branches to the subroutine location. Upon transfer of control, the subroutine, should follow the **callee** rules below.

After the subroutine returns control, the caller can expect to find the return value of the subroutine, if any, in the register(s) documented by the author of the subroutine. To restore the machine state, the caller should:

1) **Remove** the parameters from stack. This restores the stack to its state before the call was performed. The top of the stack is reset to its location prior to the call setup.

2) **Recover** the value, if any, returned by the subroutine.

3) **Restore** the contents of any caller-saved registers by popping them off of the stack. The caller can assume that no other registers were modified by the subroutine.

Example
The code below shows a function call that follows the **caller** rules. The caller is calling a function *myFunc* that takes three integer parameters. First parameter is in RAX, the second parameter is the constant 216; the third parameter is in memory location *var*.

```
push [var]     ; Push last parameter to stack first
push 216       ; Push the second parameter
push rax       ; Push first parameter last
call myFunc    ; Call the function
add rsp, 24    ; Rid stack and deallocate 3 entries
```

Note in this example, that after the call returns, the caller cleans up the stack using the **add** instruction. We have 24 bytes (3 parameters * 8 bytes each) on the stack, and the stack grows down (in the positive direction). Thus, to get rid of the parameters, we can simply add 24 to the stack pointer. This operation resets the stack pointer to its original value before the call.

The result produced by myFunc is now available for use in the **return register**. The values of any other registers may have been changed by the subroutine, but it was the subroutine's job to recover their contents prior to returning control to the caller.

Callee Rules

The definition of the subroutine should adhere to the following rules at the beginning of the subroutine:

1) Push the value of **RBP** onto the stack, and then copy the value of **RSP** into **RBP** using the following instructions:

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

This initial action maintains the base pointer, **RBP**. The base pointer is used by convention as a point of reference for finding parameters and local variables on the stack. When a subroutine is executing, the base pointer holds a copy of the stack pointer value from when the subroutine started executing. Parameters and local variables will always be located at known, constant offsets away from the base pointer value. We push the old base pointer value at the beginning of the subroutine so that we can later restore the original **RBP** base pointer value of the caller when the subroutine returns control. Remember, the caller is not expecting the subroutine to change the value of the base pointer. We then move the stack pointer into **RBP** to obtain our point of reference for accessing parameters and local variables.

2) Next, allocate local variables by making space on the stack. Recall, the stack grows down, so to make space on the top of the stack, the stack pointer should be decremented (negative direction). The amount by which the stack pointer is decremented depends on the number and size of local variables needed. For example, if 3 local integers (8 bytes each) were required, the stack pointer would need to be decremented by 24 to make space for these local variables (i.e., sub rsp, 24). As with parameters, local variables will be located at known offsets from the base pointer.

3) Next, save the values of the callee-saved registers that will be used by the function. To save registers, push them onto the stack. Any callee-saved registers will be preserved by the calling convention.

After these three actions are performed, the body of the subroutine may proceed. When the subroutine returns control, it must follow these steps:

1) **Place** the return value in the documented register.

2) **Restore** the old values of any callee-saved registers. The register contents are restored by popping them from the stack. The registers should be popped in the reverse order that they were pushed.

3) **De-allocate** local variables. The obvious way to do this might be to add the appropriate value to the stack pointer (since the space was allocated by subtracting the needed amount from the stack pointer). In practice, a less error-prone way to deallocate the variables is to copy the value in the original base pointer into the stack pointer:

```
mov rsp, rbp
```

This works because the base pointer always contains the value that the stack pointer contained immediately prior to the allocation of the local variables.

4) Immediately before returning, restore the caller's base pointer value by popping **RBP** off the stack. Recall, that the first thing we did on entry to the subroutine was to push the base pointer to save its value.

5) Finally, return to the caller by executing a **ret** instruction. This instruction will find and remove the appropriate return address from the stack.

Note that the callee's rules fall cleanly into two halves that are basically mirror images of one another. The first half of the rules apply to the beginning of the function, and are commonly said to define the **prologue** to the function. The latter half of the rules apply to the end of the function, and are thus commonly said to define the **epilogue** of the function.

Example
Here is an example function definition that follows the callee rules:

```
    myFunc equ     $
      ; Subroutine Prologue
      push rbp     ; Save the old base pointer value.
      mov rbp, rsp ; Set the new base pointer value.
      sub rsp, 8   ; Make room one 8-byte nonreg local variable
      push rdi     ; Save the values of registers that function
      push rsi     ; may modify internally (say RDI and RSI)
      :
      ; Subroutine Body Statements
      ; lets assume that some calculation puts result in RAX for
      ; return to caller.

      ; Subroutine Epilogue
      pop rsi           ; Recover register values
      pop rdi
      mov rsp, rbp      ; Deallocate local variables
      pop rbp           ; Restore the caller's base pointer value
      ret               ; result is in the RAX register
```

The subroutine prologue performs the standard actions of saving a snapshot of the stack pointer in RBP (the base pointer), allocating local variables by decrementing the stack pointer, and saving local work register values on the stack.

In the body of the subroutine we can see the use of the base pointer. Both parameters and local variables are located at constant offsets from the base pointer for the duration of the subroutines execution. In particular, we notice that since parameters were placed onto the stack before the subroutine was called, they are always located below the base pointer (at higher addresses) on the stack. The first parameter to the subroutine can always be found at memory location RBP + 16, the second at RBP + 24, the third at RBP + 32. Similarly, since local variables are allocated after the base pointer is set,

they always reside above the base pointer (at lower addresses) on the stack. In particular, the first local variable is always located at RBP - 8, the second at RBP - 16, and so on. This conventional use of the base pointer allows us to quickly identify the use of local variables and parameters within a function body.

The function epilogue is basically a mirror image of the function prologue. The caller's register values are recovered from the stack, the local variables are deallocated by resetting the stack pointer, the caller's base pointer value is recovered, and the **ret** instruction is used to return to the appropriate code location in the caller.