ICS 51: Introductory Computer Organization **[Winter 2015]**

Lab#3 (lab3.zip)

(Due: Tuesday March 10 - 11:55pm)

1. x86 Calling Convention[1]:

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 protocol about how to call and return from routines. 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 set of calling convention rules, high-level language compilers can be made to follow the rules, thus allowing hand-coded assembly language routines and high-level language routines to call one another.

We will learn about the widely used C language calling convention in this lab. Following this convention will allow you to write assembly language subroutines that are safely callable from C (and C++) code.

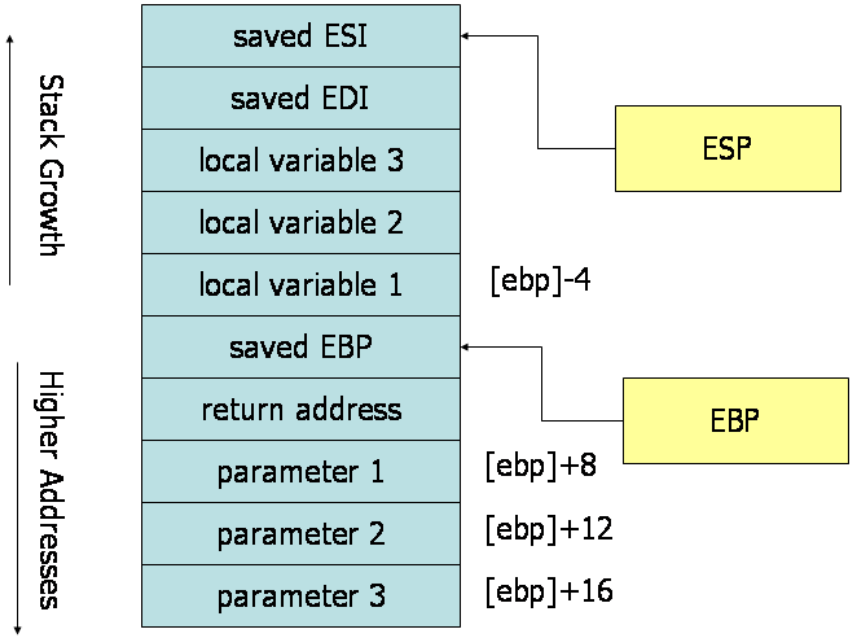
The C calling convention is based heavily on the use of the hardware-supported stack. It is based on the `push`, `pop`, `call`, and `ret` instructions. Subroutine parameters are passed on the stack. Registers are saved on the stack, and local variables used by subroutines are placed in memory on the stack. The vast majority of high-level procedural languages implemented on most processors have used similar calling conventions.

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 writer of the subroutine (the callee). It should be emphasized that mistakes 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 call convention in your own subroutines.

A good way to visualize the operation of the calling convention is to draw the contents of the nearby region of the stack during subroutine execution. The image below depicts the contents of the stack during the execution of a subroutine with three parameters and three local variables. The cells depicted in the stack are 32-bit wide memory locations, thus the memory addresses of the cells are 4 bytes apart. The first parameter resides at an offset of 8 bytes from the base pointer. Above the parameters on the stack (and below the base pointer), the `call` instruction placed the return address, thus leading to an extra 4 bytes of offset from the base pointer to the first parameter. When

[1] http://www.cs.virginia.edu/~evans/cs216/guides/x86.html#calling

the `ret` instruction is used to return from subroutine, it will jump to the return address stored on the stack.



Stack during Subroutine Call

1.1 Caller Rules

To make a subroutine call, the caller should:

R1. Before calling a subroutine, the caller should save the contents of certain registers that are designated *caller-saved*. The caller-saved registers are EAX, ECX, EDX. Since the called subroutine is allowed to modify these registers, if the caller relies on their values after the subroutine returns, the caller must push the values in these registers onto the stack (so they can be restore after the subroutine returns).

R2. To pass parameters to the subroutine, push them onto the stack before the call. The parameters should be pushed in inverted order (i.e. last parameter first). Since the stack grows down, the first parameter will be stored at the lowest address.

R3. To call the subroutine, use the `call` instruction. This instruction places the return address on top of the parameters on the stack, and branches to the subroutine code. This invokes the subroutine, which should follow the callee rules below.

After the subroutine returns (immediately following the `call` instruction), the caller can expect to find the return value of the subroutine in the register EAX. To restore the machine state, the caller should:

R4. Remove the parameters from stack. This restores the stack to its state before the call was performed.

R5. Restore the contents of caller-saved registers (EAX, ECX, EDX) 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 *callee* that takes three integer parameters. First parameter is in EAX, the second parameter is the constant 216. These two parameters are call-by-value parameters. The third parameter is the address of memory location *var*. Note that the third parameter is a call-by-reference parameter.

R1	`push eax`
	`push ecx`
	`push edx`

R2	`lea  ecx, var   // Get the effective address of sum1`
	`push ecx        // Push (the address of) last parameter first`
	`push 216        // Push the second parameter`
	`push eax        // Push first parameter last`

R3	`call callee     // Call the function`
R4	`add esp, 12     // Clear the stack`

R5	`pop edx`
	`pop ecx`
	`pop eax`

Note that after the call returns, the caller cleans up the stack using the `add` instruction. We have 12 bytes (3 parameters * 4 bytes each) on the stack, and the stack grows down. Thus, to get rid of the parameters, we can simply add 12 to the stack pointer.

The result produced by *callee* is now available for use in the register EAX. The values of the caller-saved registers (EAX, ECX and EDX), may have been changed by the *callee*. That's why the caller has saved them on the stack before the call and restore them after it.

1.2. Callee Rules

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

E1. Push the value of EBP onto the stack, and then copy the value of ESP into EBP using the following instructions:

```
push ebp
mov  ebp, esp
```

This initial action maintains the *base pointer*, EBP. 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 appropriate base pointer value for the caller when the subroutine returns. Remember, the caller is not expecting the subroutine to change the value of the base pointer. We then move the stack pointer into EBP to obtain our point of reference for accessing parameters and local variables.

E2. 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. 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 (4 bytes each) were required, the stack pointer would need to be decremented by 12 to make space for these local variables (i.e., `sub esp, 12`). As with parameters, local variables will be located at known offsets from the base pointer.

E3. Next, save the values of the *callee-saved* registers that will be used by the function. To save registers, push them onto the stack. The callee-saved registers are EBX, EDI, and ESI (ESP and EBP will also be preserved by the calling convention, but need not be pushed on the stack during this step).

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

E4. Leave the return value in EAX.

E5. Restore the old values of any callee-saved registers (EBX, EDI and ESI) that were modified. The register contents are restored by popping them from the stack. The registers should be popped in the inverse order that they were pushed.

E6. Deallocate 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 move the value in the base pointer into the stack pointer: `mov esp, ebp`. This works because the base pointer always contains the value that the stack pointer contained immediately prior to the allocation of the local variables.

E7. Immediately before returning, restore the caller's base pointer value by popping EBP off the stack. Recall that the first thing we did on entry to the subroutine was to push the base pointer to save its old value.

E8. 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:

E1	`// Subroutine Prologue`
	`push ebp  // Save the old base pointer value.`
	`mov ebp, esp  // Set the new base pointer value.`

E2	`sub esp, 4  // Make room for one 4-byte local variable.`

E3	`push edi  // Save the values of registers that the function`
	`push esi  // will modify. This function uses EDI and ESI and`
	`push ebx  // EBX`

	`// Subroutine Body`
	`mov eax, [ebp+8]  // Move value of parameter 1 into EAX`
	`mov esi, [ebp+12]  // Move value of parameter 2 into ESI`
	`mov edi, [ebp+16]  // Move the address of parameter 3 into EDI`
	`mov ebx, [edi]  // Get the value of parameter 3`
	`mov [ebp-4], ebx  // Move value of parameter 3 into the local variable`
	`add [ebp-4], esi  // Add ESI into the local variable`

E4	`add eax, [ebp-4]  // Add the contents of the local variable` `// into EAX (final result)`

E5	`// Subroutine Epilogue`
	`pop ebx`
	`pop esi  // Recover register values`
	`pop edi`

E6	`mov esp, ebp  // Deallocate local variables`
E7	`pop ebp  // Restore the caller's base pointer value`
E8	`ret`

The subroutine prologue performs the standard actions of saving a snapshot of the stack pointer in EBP (the base pointer), allocating local variables by decrementing the stack pointer, and saving 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 (i.e. at higher addresses) on the stack. The first parameter to the subroutine can always be found at memory location [EBP+8], the second at [EBP+12], the third at [EBP+16]. Similarly, since local variables are allocated after the base pointer is set, they always reside above the base pointer (i.e. at lower addresses) on the stack. In particular, the first local variable is always located at [EBP-4], the second at [EBP-8], 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.

The example used in this handout can be downloaded from here.

2. Your task:

Part 1) Modular selection sort

Part 2) Recursive factorial

Part 3) Count number of 1's in the binary representation of an integer

Part 4) Reverse the letter case for an array of characters

Part 5) Recursive fibonacci

The template code (lab3.c) and the tester (lab3-testing.c) are given to you. Download them from here. Remember you are only allowed to make modifications in the marked blocks of lab3.c. **For more details, read the comment sections of lab3.c and lab3-testing.c**.

VERY IMPORTANT:

Submit ONLY the completed lab3.c file
and
do NOT zip it!