Note of Computer Systems: A Programmer's Perspective, 2/E (CS:APP2e)

[p161] 3.2.1 Machine-Level Code

The ***program counter*** (commonly referred to as the “PC,” and called **%eip** in IA32) indicates the address in memory of **the next instruction** to be executed.

[p166]

To provide a clearer presentation of assembly code, we will show it in a form that omits most of the directives, while including line numbers and explanatory annotations. For our example, an annotated version would appear as follows:

```
1 simple:
2     pushl %ebp               Save frame pointer
3     movl  %esp, %ebp         Create new frame pointer
4     movl  8(%ebp), %edx      Retrieve xp
5     movl  12(%ebp), %eax     Retrieve y
6     addl  (%edx), %eax       Add *xp to get t
7     movl  %eax, (%edx)       Store t at xp
8     popl  %ebp               Restore frame pointer
9     ret                      Return
```

[p168]

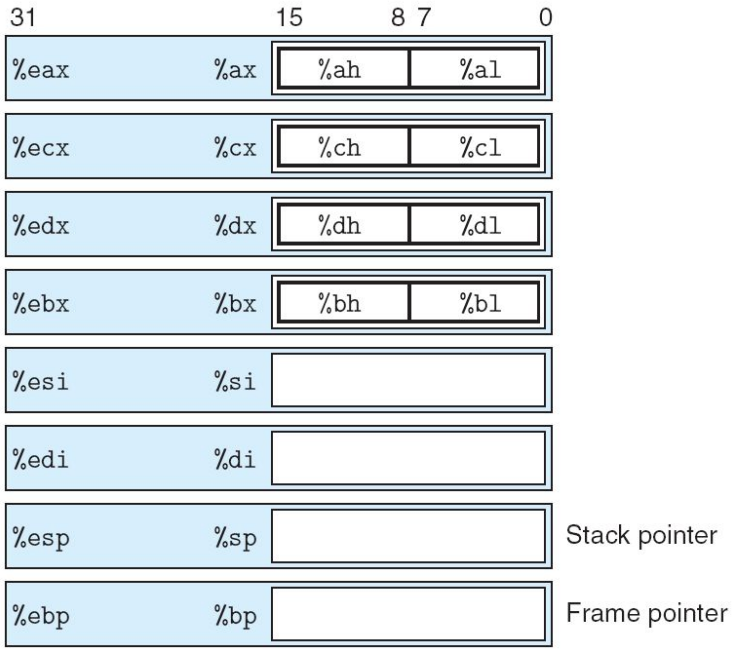

Figure 3.2 IA32 integer registers.

All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The 2 low-order bytes of the first four registers can be accessed independently.

Why esp is called “stack pointer” while ebp is called “frame pointer” ? [ref 1]

%ESP - Stack Pointer

This 32-bit register is implicitly manipulated by several CPU instructions (PUSH, POP, CALL, and RET among others), **it always points to the last element used on the stack** (not the first *free* element): this means that the PUSH and POP operations would be specified in pseudo-C as:

```
*--ESP = value;    // push
value = *ESP++;  // pop
```

The "Top of the stack" is an occupied location, not a free one, and is at the **lowest** memory address.

%EBP - Base Pointer
This 32-bit register is used to reference all the function parameters and local variables in the current stack frame. Unlike the %esp register, the base pointer is manipulated only *explicitly*. **This is sometimes called the "Frame Pointer"**.

%EIP - Instruction Pointer
This holds the address of the next CPU instruction to be executed, and it's saved onto the stack as part of the **CALL** instruction. As well, any of the "jump" instructions modify the %EIP directly.

[p219]
3.7.1 Stack Frame Structure
IA32 programs make use of the program stack to support procedure calls. The machine uses the stack to pass procedure arguments, to store return information, to save registers for later restoration, and for local storage.The portion of the stack allocated for a single procedure call is called a stack frame. Figure 3.21 diagrams the general structure of a stack frame. The top-most stack frame is delimited by two pointers, with register %ebp serving as the frame pointer, and register %esp serving as the stack pointer. The stack pointer can move while the procedure is executing, and hence most information is accessed relative to the frame pointer.

Suppose procedure P (*the caller*) calls procedure Q (*the callee*). The arguments to Q are contained within the stack frame for P. In addition, when P calls Q, the return address within P where the program should resume execution when it returns from Q is pushed onto the stack, forming the end of P's stack frame. The stack frame for Q starts with the saved value of the frame pointer (a copy of register %ebp), followed by copies of any other saved register values.

Procedure Q also uses the stack for any local variables that cannot be stored in registers. This can occur for the following reasons:

- There are not enough registers to hold all of the local data.
- Some of the local variables are arrays or structures and hence must be accessed by array or structure references.
- The address operator '&' is applied to a local variable, and hence we must be able to generate an address for it.

In addition, Q uses the stack frame for storing arguments to any procedures it calls. As illustrated in Figure 3.21, within the called procedure, the first argument is positioned at offset 8 relative to %ebp, and the remaining arguments (assuming their data types require no more than 4 bytes) are stored in successive 4-byte blocks, so that argument i is at offset 4 + 4i relative to %ebp. Larger arguments (such as structures and larger numeric formats) require larger regions on the stack.

As described earlier, the stack grows toward lower addresses and the stack pointer %esp points to the top element of the stack. Data can be stored on and retrieved from the stack using the *pushl* and *popl* instructions. Space for data with no specified initial value can be allocated on the stack by simply decrementing the stack pointer by an appropriate amount. Similarly, space can be deallocated by incrementing the stack pointer.

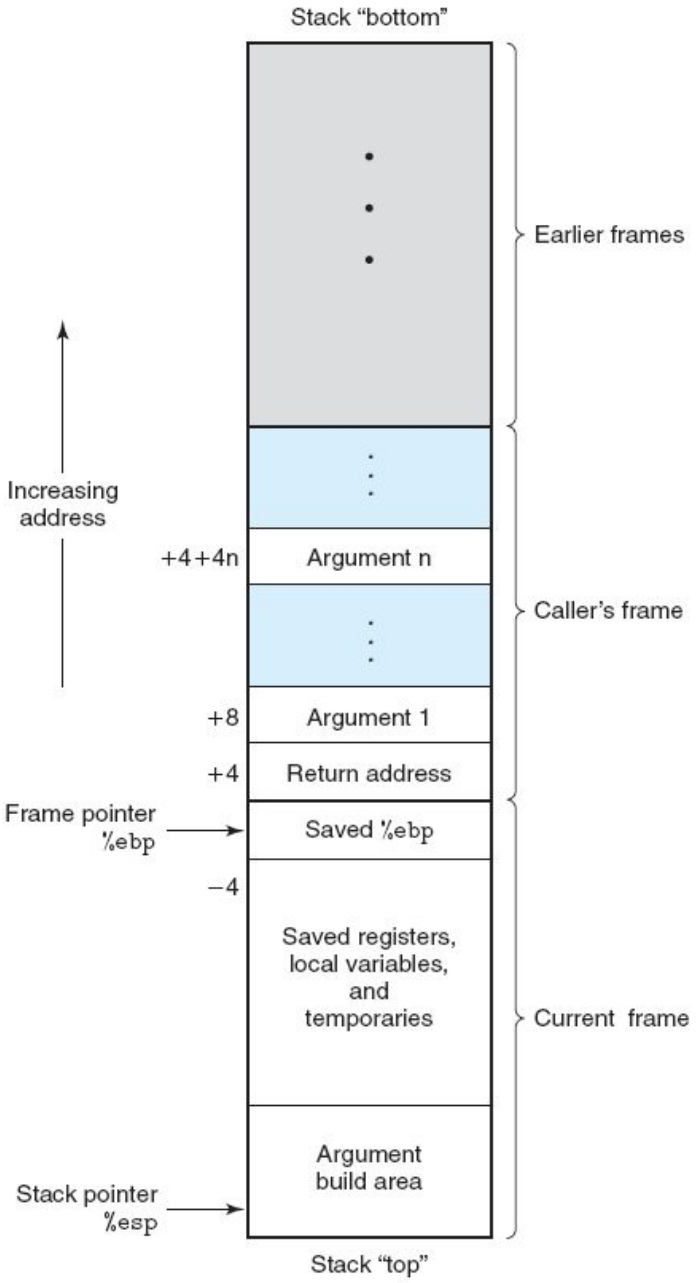

Figure 3.21 Stack frame structure.
The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.

[p221]
3.7.2 Transferring Control
The `call` instruction has a target indicating the address of the instruction where the called procedure starts.
The effect of a `call` instruction is to push a return address on the stack and jump to the start of the called procedure. The return address is the address of the instruction immediately following the `call` in the program, so that execution will resume at this location when the called procedure returns. The `ret` instruction pops an address off the stack and jumps to this location. The proper use of this instruction is to have prepared the stack so that the stack pointer points to the place where the preceding `call` instruction stored its return address.

[ref]
1. Intel x86 Function-call Conventions - Assembly View

http://www.unixwiz.net/techtips/win32-callconv-asm.html
2. Assorted Reference Material on the ia32 Architecture
http://inst.eecs.berkeley.edu/~cs164/sp11/ia32-refs/