IA-32 Function Calling Conventions

When functions (routines) call other functions (subroutines), they may need to pass arguments to them. The called subroutines access these arguments as *parameters*. Conversely, some subroutines pass a *result* or return value to their callers. In the IA-32 environment most arguments are passed on the runtime stack; some vector arguments are passed in registers. Results are returned in registers or in memory. To efficiently pass values between callers and callees, GCC follows strict rules when it generates a program's object code.

This article describes the data types that can be used to manipulate the arguments and results of subroutine calls, how routines pass arguments to the subroutines they call, and how subroutines that provide a return value pass the result to their callers. This article also lists the registers available in the IA-32 architecture and whether their value is preserved after a subroutine call.

The function calling conventions used in the IA-32 environment are the same as those used in the System V IA-32 ABI, with the following exceptions:

- Different rules for returning structures
- The stack is 16-byte aligned at the point of function calls
- Large data types (larger than 4 bytes) are kept at their natural alignment
- Most floating-point operations are carried out using the SSE unit instead of the x87 FPU, except when operating on `long double` values. (The IA-32 environment defaults to 64-bit internal precision for the x87 FPU.)

The content of this article is largely based in *System V Application Binary Interface: Intel386 Architecture Processor Supplement*, available at http://www.sco.com/developers/devspecs/abi386-4.pdf.

Data Types and Data Alignment

Using the correct data types for your variables helps to maximize the performance and portability of your programs. Data alignment specifies how data is laid out in memory. A data type's *natural alignment* specifies the default alignment of values of that that type.

Table 1 lists the ANSI C scalar data types and their sizes and natural alignment in this environment.

Table 1 Size and natural alignment of the scalar data types

Data type	Size (in bytes)	Natural alignment (in bytes)
`_Bool`, `bool`	1	1
`unsigned char`	1	1
`char`, `signed char`	1	1
`unsigned short`	2	2
`signed short`	2	2
`unsigned int`	4	4
`signed int`	4	4
`unsigned long`	4	4

`signed long`	4	4
`unsigned long long`	8	4
`signed long long`	8	4
`float`	4	4
`double`	8	4
`long double`	16	16
pointer	4	4

Table 2 shows the vector types available in this environment.

Table 2 Size and alignment of the vector types

Vector type	Element data type	Size (in bytes)	Alignment (in bytes)
`__m64`	`int`	8	8
`__m128i`	`int`	16	16
`__m128`	`float`	16	16
`__m128d`	`double`	16	16

These are some important details about this environment:

- A byte is 8 bits long.
- A null pointer has a value of 0.
- This environment doesn't require 8-byte alignment for double-precision values.
- This environment requires 16-byte alignment for 128-bit vector elements.

These are the alignment rules followed in this environment:

1. Scalar data types use their natural alignment.
2. Composite data types (arrays, structures, and unions) take on the alignment of the member with the highest alignment. An array assumes the same alignment as its elements. The size of a composite data type is a multiple of its alignment (padding may be required).

Function Calls

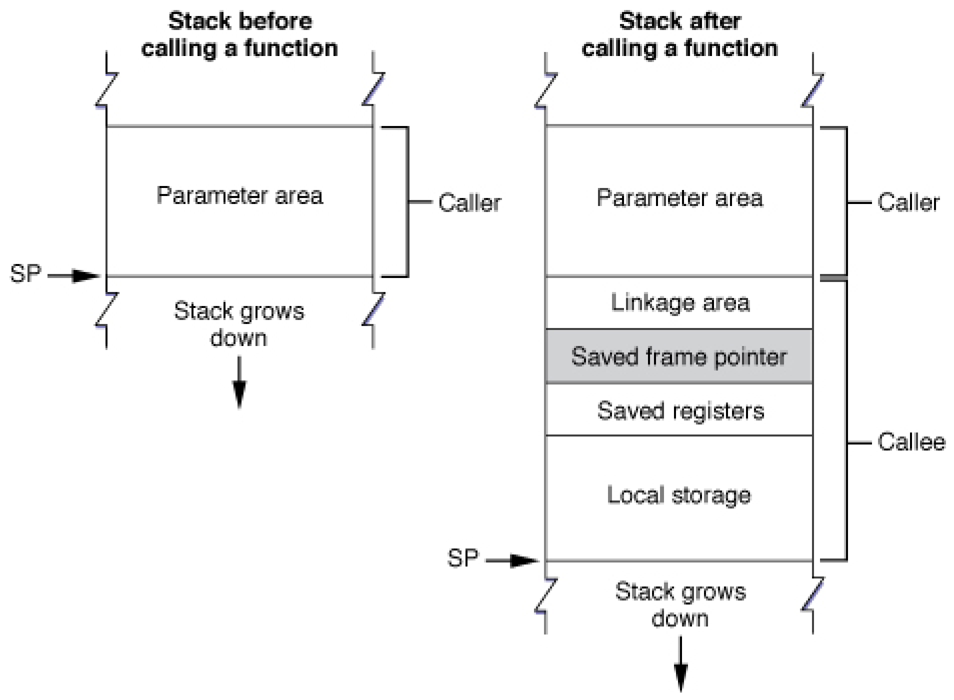
This section details the process of calling a subroutine and passing parameters to it, and how subroutines return values to their callers.

Note: These parameter-passing conventions are part of the Apple standard for procedural programming interfaces. Object-oriented languages may use different rules for their own method calls. For example, the conventions for C++ virtual function calls may be different from those for C functions.

Stack Structure

The IA-32 environment uses a stack that—at the point of function calls—is 16-byte aligned, grows downward, and contains local variables and a function's parameters. Each routine may add linkage information to its stack frame, but it's not required to do so. Figure 1 shows the stack before and during a subroutine call. (To help prevent the execution of malicious code on the stack, GCC protects the stack against execution.)

Figure 1 Stack layout



The *stack pointer* (SP) points to the bottom of the stack. Stack frames contain the following areas:

- *The parameter area* stores the arguments the caller passes to the called subroutine. This area resides in the caller's stack frame.
- *The linkage area* contains the address of the caller's next instruction.
- *The saved frame pointer* (optional) contains the base address of the caller's stack frame.

  You can use the `gcc -fomit-frame-pointer` option to make the compiler not save, set up, and restore the frame pointer in function calls that don't need one, making the EBP register available for general use. However, doing so may impair debugging.
- The *local storage area* contains the subroutine's local variables and the values of the registers that must be restored before the called function returns. See Register Preservation for details.
- The *saved registers area* contains the values of the registers that must be restored before the called function returns. See Register Preservation for details.

In this environment, the stack frame size is not fixed.

Prologs and Epilogs

The called subroutine is responsible for allocating its own stack frame. This operation is accomplished by a section of code called the *prolog*, which the compiler places before the body of the function. After the body of the function, the compiler places an *epilog* to restore the process to the state it was prior to the subroutine call.

The prolog performs the following tasks:

1. Pushes the value of the stack frame pointer (EBP) onto the stack.

2. Sets the stack frame pointer to the value of the stack pointer (ESP).
3. Pushes the values of the registers that must be preserved (EDI, ESI, and EBX) onto the stack.
4. Allocates space in the stack frame for local storage.

The epilog performs these tasks:

1. Deallocates the space used for local storage in the stack.
2. Restores the preserved registers (EDI, ESI, EBX, EBP) by popping the values saved on the stack by the prolog.
3. Returns.

Note: Functions called during signal handling have no unusual restrictions on their use of registers. When a signal-handling function returns, the process resumes its original path with the registers restored to their original values.

Listing 1 shows the definition of the `simp` function.

Listing 1 Definition of the `simp` function

```
#include <stdio.h>
void simp(int i, short s, char c) {
    printf("Hi!\n");
}
```

Listing 2 shows a possible prolog for the `simp` function.

Listing 2 Example prolog

```
pushl   %ebp            ; save EBP
movl    %esp,%ebp       ; copy ESP to EBP
pushl   %ebx            ; save EBX
subl    $0x24,%esp      ; allocate space for local storage
```

Listing 3 shows a possible epilog for the `simp` function.

Listing 3 Example epilog

```
addl    $0x24,%esp      ; deallocate space for local storage
popl    %ebx            ; restore EBX
popl    %ebp            ; restore EBP
ret                     ; return
```

Passing Arguments

The compiler adheres to the following rules when passing arguments to subroutines:

1. The caller ensures that the stack is 16-byte aligned at the point of the function call.
2. The caller aligns nonvector arguments to 4-byte (32 bits) boundaries.

   The size of each argument is a multiple of 4 bytes, with tail padding when necessary. Therefore, 8-bit and 16-bit integral data types are promoted to 32-bit before they are pushed onto the stack.

3. The caller places arguments in the parameter area in reverse order, in 4-byte chunks. That is, the rightmost argument has the highest address.
4. The caller places all the fields of structures (or unions) with no vector elements in the parameter area. These structures are 4-byte aligned .
5. The caller places structures with vector elements on the stack, 16-byte aligned. Each vector within the structure is 16-byte aligned.
6. The caller places 64-bit vectors (`__m64`) on the parameter area, aligned to 8-byte boundaries.
7. The caller places vectors 128-bit vectors (`__m128`, `__m128d`, and `__m128i`) in registers XMM0 through XMM3). When the usable XMM registers are exhausted, the caller places 128-bit vectors in the parameter area. The caller aligns 128-bit vectors in the parameter area to 16-byte boundaries.

In general, the caller is responsible for removing all the parameters used in a function call after the called function returns. The only exception are parameters that are generated automatically by GCC. When a function returns a structure or union larger than 8 bytes, the caller passes a pointer to appropriate storage as the first argument to the function. GCC adds this parameter automatically in the code generated. See Returning Results for more information. For example, the compiler would translate the code shown in Listing 4 to machine language as if it were written as shown in Listing 5.

Listing 4 Using a large structure—source code

```
typedef struct {
    float ary[8];
} big_struct;
big_struct callee(int a, float b) {
    big_struct callee_struct;
    ...
    return callee_struct;
}
caller() {
    big_struct caller_struct;
    caller_struct = callee(3, 42.0);
}
```

Listing 5 Using a large structure—compiler interpretation

```
typedef struct {
    float ary[8];
} big_struct;
void callee(big_struct *p, int a, float b)
{
    big_struct callee_struct;
    ...
    *p = callee_struct;
    return;
}
caller() {
    big_struct caller_struct;
    callee(&caller_struct, 3, 42.0);
}
```

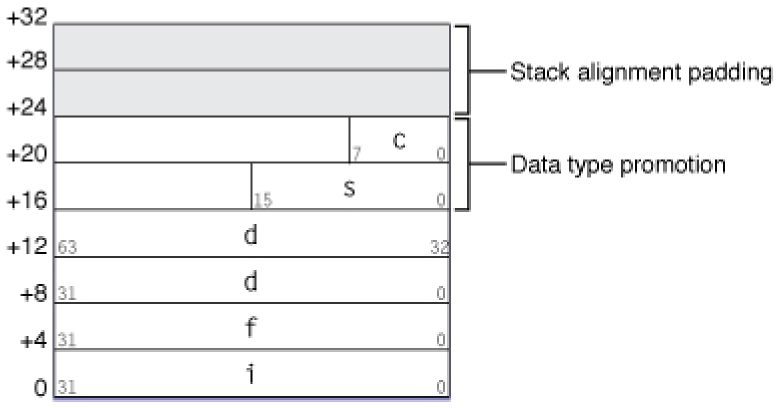
Passing Arguments of the Fundamental Data Types

Assume the function `foo` is declared like this:

```
void foo(SInt32 i, float f, double d, SInt16 s, UInt8 c);
```

Figure 2 illustrates the placement of arguments in the parameter area at the point of the function call. Note the padding added to align the stack at 16 bytes.

Figure 2 Argument assignment with arguments of the fundamental data types



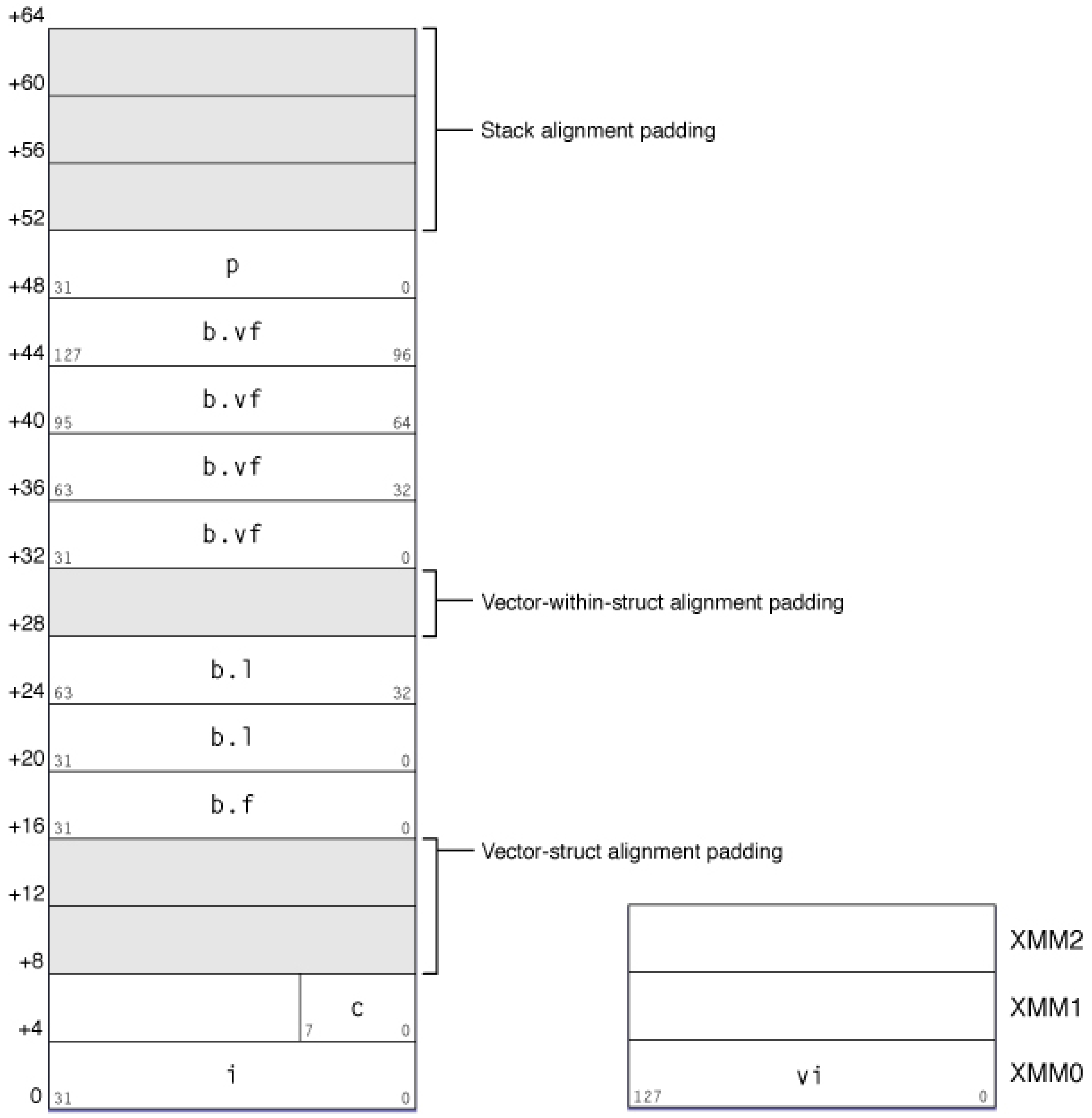
Passing Structures and Vectors

Assume the structure `data` and the function `bar` are declared like this:

```
struct data {
    float f;
    long long l;
    __m128 vf;
};
void bar(SInt32 i, UInt8 c, struct data b, __m128i vi, void* p);
```

Figure 3 illustrates the placement of arguments in the parameter area, the stack's 16-byte alignment padding at the point of the call, and the XMM registers.

Figure 3 Argument assignment with structure and vector arguments



Returning Results

This is how functions pass results to their callers:

- **Scalar values.** Scalar values include structures that contain only one scalar value.
  - The called function places integral or pointer results in EAX.
  - The called function places floating-point results in ST0.
  - The caller removes the value from this register, even when it doesn't use the value.
- **Structures.** The called function returns structures according to their aligned size.
  - Structures 1 or 2 bytes in size are placed in EAX.
  - Structures 4 or 8 bytes in size are placed in: EAX and EDX.
  - Structures of other sizes are placed at the address supplied by the caller. For example, the C++ language occasionally forces the compiler to return a value in memory when it would normally be returned in registers. See Passing Arguments for more information.
- **Vectors.**
  - The called function places vectors at the address supplied by the caller.

Register Preservation

Table 3 lists the IA-32 architecture registers used in this environment and their volatility in procedure calls. Registers that must preserve their value after a function call are called *nonvolatile*.

Table 3 Processor registers in the IA-32 architecture

Type	Name	Preserved	Notes
General-purpose register	EAX	No	Used to return integral and pointer values. The caller may also place the address to storage where the callee places its return value in this register.
	EDX	No	Dividend register (divide operation). Available for general use for all other operations.
	ECX	No	Count register (shift and string operations). Available for general use for all other operations.
	EBX	Yes	Position-independent code base register. Available for general use in non-position-independent code.
	EBP	Yes	Stack frame pointer. Optionally holds the base address of the current stack frame. A routine's parameters reside in the previous frame as positive offsets of this register's value. Local variables reside at negative offsets.
	ESI	Yes	Available for general use.
	EDI	Yes	Available for general use.
Stack-pointer register	ESP	Yes	Holds the address of the bottom of the stack.
Floating-point register	ST0	No	Used to return floating-point values. When the function doesn't return a floating-point value, this register must be empty. This register must also be empty on function entry.
	ST1-ST7	No	Available for general use. These registers must be empty on routine entry and exit.
64-bit register	MM0-MM7	No	Used to execute single-instruction, multiple-data (SIMD) operations on 64-bit packed byte, 2-byte, and 4-byte integers.
128-bit register	XMM0-XMM7	No	Used to execute 32-bit and 64-bit floating-point arithmetic. Also used to execute single-instruction, multiple-data (SIMD) operations on 128-bit packed single-precision and double-precision scalar and floating-point values, and 128-bit packed byte, 2-byte, and 4-byte integers. XMM0-XMM3 are used to pass the first four vectors in a function call.
System-flags register	EFLAGS	No	Contains system flags, such as the direction flag and the carry flag. The direction flag must be set to the "forward" direction (that is, `0`) before entry to and upon exit from a routine. Other user flags have no specified role in the standard calling sequence and are not preserved.