CS:APP Web Aside ASM:IA32: IA32 Programming*

Randal E. Bryant
David R. O'Hallaron

January 21, 2015

Notice

The material in this document is supplementary material to the book Computer Systems, A Programmer's Perspective, Third Edition*, by Randal E. Bryant and David R. O'Hallaron, published by Prentice-Hall and copyrighted 2016. In this document, all references beginning with "CS:APP3e " are to this book. More information about the book is available at* `csapp.cs.cmu.edu`*.*

1 Introduction

Starting with the introduction of the Intel i386 in 1985, processors in the x86 family have been able to execute 32-bit programs in a machine language known as *IA32* (for "Intel architecture, 32 bits"). With these programs, all pointers are 32 bits long, and hence the addressable range of the code and data in a program is 4 gigabytes.

CS:APP3e Chapter 3 describes the x86-64 machine language, the 64-bit extension to IA32 introduced by AMD in 2000 (first implemented with their Opteron processor in 2003) and adopted by Intel with the Pentium 4E in 2004. Even though most x86 processors manufactured today are designed to execute x86-64 programs, they can also execute IA32 programs in a compatibility mode. Both IA32 and x86-64 programs can coexist on a single machine. GCC will generate IA32 code when given command-line option `-m32`, and it will generate x86-64 code when given command-line option `-m64`. Many application programs are still compiled for IA32, since most programs require far less than 4 gigabytes of memory, and the 32-bit pointers make IA32 data structures more compact.

In this document, we describe IA32 programs based on the assumption that you, the reader, are already familiar with x86-64. Although this reverses the order in which these two machine languages were devised,

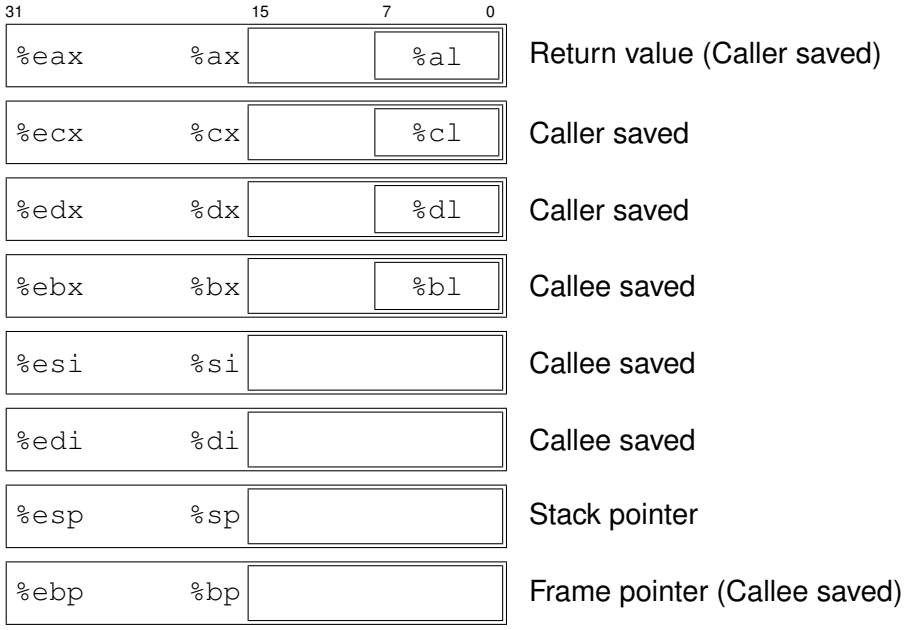

Figure 1: **IA32 integer registers.** All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The low-order bytes of the first four registers can be accessed independently.

you will find that having a background in x86-64 makes understanding IA32 fairly straightforward.

This document does not provide a comprehensive coverage of all aspects of IA32, but rather those features that are most prevalent in machine-language programs generated by GCC. The authoritative reference to IA32 is the documentation provided by Intel [1].

2 Registers

Figure 1 illustrates the set of 8 general-purpose registers used to hold both integer data and pointers. We see that, in several respects, these registers are a subset of those available in x86-64 programs (CS:APP3e Figure 3.2):

- There are only 8 registers instead of 16.
- The maximum register size is 32 bits.
- The low-order register bytes are only directly accessible for the first 4 registers.

There are many similarities between how registers are used in IA32 and x86-64:

- Both have a designated register to serve as the stack pointer—`%esp` for IA32 and `%rsp` for x86-64. This register should only be altered in accordance with the stack discipline used by programs.

- Both have a designated register that can optionally serve as the frame pointer—`%ebp` for IA32 and `%rbp` for x86-64. We will describe the use of frame pointers as part of our discussion of procedures in Section 4.
- Both use a designated register (`%eax` for IA32 and `%rax` for x86-64) for returning pointer or integer values from procedures.
- With Linux, some of the registers are used according to a caller-saved stack discipline, while others follow a callee-saved discipline.

3 Instructions

As one would expect, IA32 assembly has the same overall form as does x86-64. For the most part, the instructions are a subset of those available in x86-64. In particular, the '`q`' variants of the data movement and arithmetic instructions are not available, but the other versions are. So, for example, programs can use instructions `movl` (move double word), `movw` (move word), and `movb` (move byte), but not `movq` (move quad word).

For instruction arguments that reference memory locations, the base and index registers must be one of the 8 double-word registers shown in Figure 1. Thus, typical memory referencing instructions include:

```
movl   %eax, 8(%esp)
addl   $1, (%edx, %eax, 4)
movsbl -8(%ebp), %eax
```

IA32 has several instructions that are not allowed in x86-64, as well as others that have slightly different effects, due to the convention that addresses are 4 bytes rather than 8. Other differences for these instructions, of course, are that the stack pointer is `%esp` rather than `%rsp`, and the frame pointer (used and modified by the `leave` instruction) is `%ebp` rather than `%rbp`.

- Rather than `leaq`, the load effective address is called `leal`. The destination must be one of the double-word registers of Figure 1.
- The push instruction is called `push` or `pushl`, and the pop instruction is called either `pop` or `popl`. They decrement and increment the stack pointer by 4, respectively.
- Procedure calls and returns use the `call` and `ret` instructions, but these decrement and increment the stack pointer by 4.
- The `leave` instruction (described in CS:APP3e Section 3.10.5 for x86-64) operates on registers `%esp` and `%ebp`. It has no arguments and is equivalent to the IA32 instruction sequence

  ```
  movl %ebp, %esp   Set stack pointer to beginning of frame
  popl %ebp         Restore saved %ebp and set stack ptr to end of caller's frame
  ```

  It is used in procedures that make use of a frame pointer, as is described in Subsection 4.1.

Instruction	Effect	Description
`imull` S	R[`%edx`]:R[`%eax`] $\leftarrow$ $S \times$ R[`%eax`]	Signed full multiply
`mull` S	R[`%edx`]:R[`%eax`] $\leftarrow$ $S \times$ R[`%eax`]	Unsigned full multiply
`cltd`	R[`%edx`]:R[`%eax`] $\leftarrow$ SignExtend(R[`%eax`])	Convert to quad word
`idivl` S	R[`%edx`] $\leftarrow$ R[`%edx`]:R[`%eax`] mod S; R[`%eax`] $\leftarrow$ R[`%edx`]:R[`%eax`] $\div S$	Signed divide
`divl` S	R[`%edx`] $\leftarrow$ R[`%edx`]:R[`%eax`] mod S; R[`%eax`] $\leftarrow$ R[`%edx`]:R[`%eax`] $\div S$	Unsigned divide

Figure 2: **Special arithmetic operations.** These operations provide full 64-bit multiplication and division, for both signed and unsigned numbers. The pair of registers `%edx` and `%eax` are viewed as forming a single 64-bit quad word.

Figure 2 shows the IA32 instructions for arithmetic operations that involve 64-bit quantities. These include signed and unsigned multiplication instructions that compute the full 64-bit product of two 32-bit numbers. They also include signed and unsigned division, where the dividend is 64 bits. In both cases, the 64-bit quantity is held in registers `%edx` and `%eax`. The instruction `cltd` provides a way to extend the sign bit of `%eax` into `%edx`. Note the similarities to the x86-64 instructions that involve 128-bit quantities (CS:APP3e Figure 3.12).

4 IA32 Procedures

Perhaps the most striking difference between IA32 and x86-64 concerns the structure of procedure calls. Most significantly, IA32 requires all arguments to be passed on the stack, whereas x86-64 requires use of the stack for argument passing only if there are more than 6 arguments. As we saw in in CS:APP3e Chapter 3, many x86-64 procedures do not require passing arguments on the stack, and indeed they often use the stack only for saving the return pointer. IA32 programs also require greater use of the stack to hold local variables, due to the small number of registers.

We split our presentation of IA32 procedures into two forms, depending on whether or not the code makes use of the frame pointer `%ebp`. Traditionally, all IA32 code used frame pointers, and so we refer to this approach as *classic IA32*. More recent versions of GCC generate IA32 code that does not use frame pointers, and so we refer to this approach as *modern IA32*. Whichever approach is the default for a particular compiler, it is possible to force the other convention using command-line argument `-fomit-frame-pointer` (modern) or `-fno-omit-frame-pointer` (classic). Code using frame pointers can be freely mixed with code that does not, as long as `%ebp` follows a callee-saved convention. That is, any function that modifies `%ebp` must restore it to its former value before returning.

In addition to presenting general concepts, we will demonstrate how procedures are implemented in IA32 using the two C functions defined in Figure 3. These functions both have two arguments, but they differ slightly in their argument types. Function `call_fun` has both arguments of type `int` as given by the following prototype:

```
int call_fun(int x, int y);
```

(a) C code for function `call_fun`

```
int call_fun(int x, int y) {
    return fun(x, &y);
}
```

(b) C code for function `fun`

```
int fun(int a, int *bp) {
    return a + *bp;
}
```

Figure 3: **C code for example functions.**

Function `fun`, on the other hand, has one argument of type `int` and the other of type `int *`.

```
int fun(int a, int *bp);
```

Both functions return values of type `int` as results.

4.1 Classic IA32: Using Frame Pointers

The concept of a frame pointer is to have a designated register holding the address of the beginning of the stack frame for the currently executing function. With the stack pointer indicating the end of the stack, these two registers bracket the stack frame. Figure 4 shows the overall organization of the IA32 runtime stack, illustrating the case where a procedure `P` has called a procedure `Q`. This figure uses the same conventions as we showed for the x86-64 runtime stack (CS:APP3e Figure 3.25). In particular, both x86-64 and IA32 frames grow toward lower addresses. We draw our stack with the addresses increasing from bottom to top. This has the peculiar effect that the stack "top" is shown at the bottom of the diagram.

In this figure, register `%ebp` indicates the beginning of the stack frame for `Q`. By convention, the previous value of `%ebp` (in this case, the one for procedure `P`) is stored on the stack at the position indicated by `%ebp`. Having this value on the stack enables it to be restored as the current procedure (in this case `Q`) returns. Before calling `Q`, procedure `P` stores the arguments in its stack frame. These are retrieved within `Q` with memory references that use the appropriate offset from register `%ebp`. Such references are indicated in Figure 4 by the notation `%ebp` $+ k$.

Within the stack frame for `Q`, there may be space for saved registers, local variables, and to store the argument values for any procedure to be called by `Q`. Typical IA32 procedures save and restore registers using the `pushl` and `popl` instructions, and they allocate additional stack space by explicitly decrementing the stack pointer. Local variables are typically accessed with memory references having offsets to the frame pointer (indicated in the figure by the notation `%ebp` $- k$), and newly generated argument values are written with memory references having offsets to the stack pointer (indicated in the figure by the notation `%esp` $+ k$).

We can see how IA32 programs make use of the stack and the stack and frame pointers by studying the assembly code for function `call_fun`, as is shown in Figure 5(a). Figure 6 diagrams the state of the stack at two points in the execution of this function.

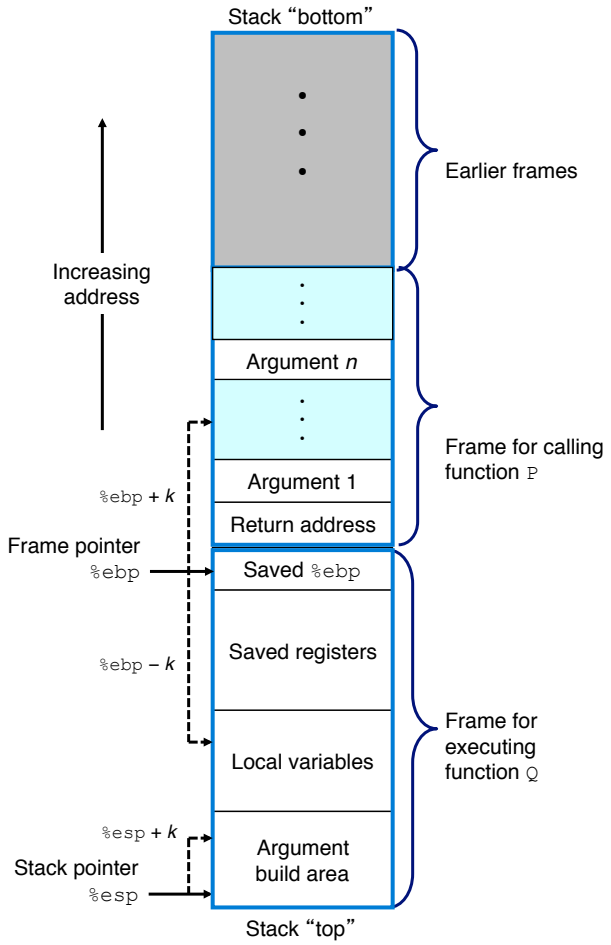

Figure 4: **General IA32 stack-frame structure with frame pointers.** The frame pointer, held in register `%ebp`, indicates the base of the current stack frame. Stack references are given as offsets to either the frame pointer or the stack pointer.

Upon entry, the state of the stack is illustrated in Figure 6(a). The `call` instruction has just pushed the return address onto the stack. Prior to that, the function that called `call_fun` put the values of arguments `y` and `x` on the stack. Each of the 3 topmost stack elements: the return address, `x` and `y` are 4 bytes long. Note also the ordering of the arguments on the stack—those that are earlier in the list of procedure parameters are closer to the top to the stack.

The first three instructions of `call_fun` (Figure 5(a)) set up its stack frame to construct the structure illustrated in Figure 6(b). The first instruction saves the current value of `%ebp` on the stack, and the second updates `%ebp` to point to this position. This combination establishes the convention for the frame pointer illustrated in Figure 4. The third instruction decrements the stack pointer by 8, allocating enough space on the stack to hold the two 4-byte arguments for function `fun`.

At this point, we can see that arguments `x` and `y` are stored at the stack at offsets 8 and 12 from `%ebp`, respectively, and that the arguments `a` and `bp` for `fun` should be stored on the stack at offsets 0 and 4 from

(a) Assembly code for `call_fun`

```
   int call_fun(int x, int y)
   x at %ebp+8, y at %ebp+12
1  call_fun:
2    pushl   %ebp                    Save %ebp
3    movl    %esp, %ebp              Set frame pointer
4    subl    $8, %esp                Allocate 8 bytes
   Store arguments at offsets 0 and 4 from %esp
5    leal    12(%ebp), %eax          Compute &y
6    movl    %eax, 4(%esp)           Store as 2nd argument
7    movl    8(%ebp), %eax           Get x
8    movl    %eax, (%esp)            Store as 1st argument
9    call    fun                     fun(x, &y)
10   leave                           Deallocate & restore %ebp
11   ret
```

(b) Assembly code for `fun`

```
   int fun(int a, int *bp)
   a at %ebp+8, bp at %ebp+12
1  fun:
2    pushl   %ebp                    Save %ebp
3    movl    %esp, %ebp              Set frame pointer
4    movl    12(%ebp), %eax          Get bp
5    movl    (%eax), %eax            Get *bp
6    addl    8(%ebp), %eax           Add a
7    popl    %ebp                    Restore %ebp
8    ret
```

Figure 5: **IA32 assembly code for example functions using frame pointers.** Arguments are passed to the functions on the stack.

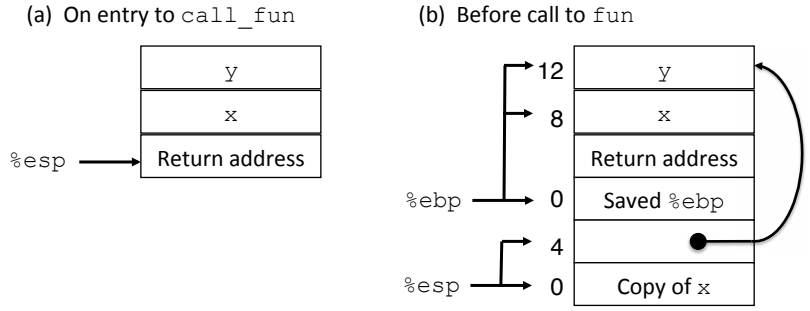

Figure 6: **Stack-frame structure for function `call_fun` using frame pointers.** The function allocates space on the stack and manages the frame pointer.

`%esp`, respectively. (These conventions follow those illustrated in Figure 4 for accessing the arguments to `call_fun`, as well as the argument build area.) The instructions at lines 5–8 then perform the actions specified in the code for `call_fun`. The `leal` instruction computes the address of argument `y`, and the following `movl` stores this address at the location for address `bp`. We show this in Figure 4(b) as an arrow from the memory location at offset 4 from `%esp` to the location of `y`. The next two `movl` instructions (lines 7–8) then read the value of `x` from memory and store a copy at offset 0 from the stack pointer.

Function `call_fun` can now call function `fun` (line 9). We will examine how this function operates shortly. Suffice it to say here that when `fun`, the stack will still have the same structure as is illustrated in Figure 4)(b), and that the value returned by `fun` will be stored in register `%eax`. The `leave` function (line 10) will then restore `%ebp` by (1) copying the value of `%ebp` to `%esp`, effectively deallocated 8 bytes of storage, and (2) popping the value at the new top of stack to `%ebp`. The result is therefore to return the stack state to that shown in Figure 4)(a), with the return address at the top of the stack. The `ret` instruction (line 11) will then cause the program to return back to the calling point.

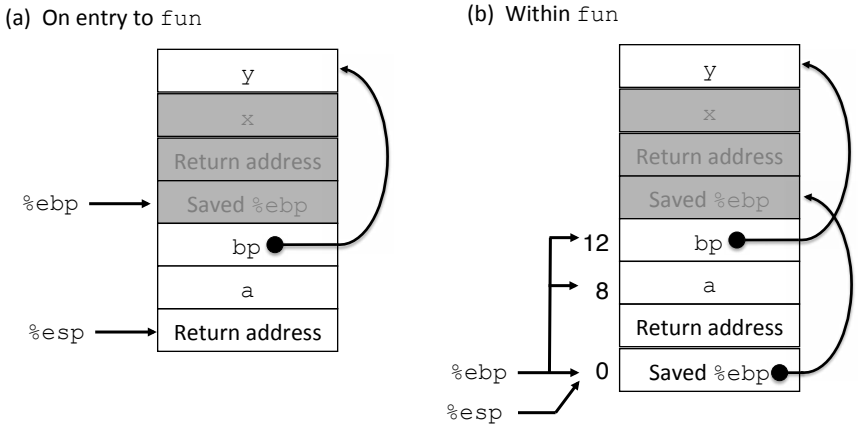

Figure 7: **Stack-frame structure for function `fun` using frame pointers.** The function manages the frame pointer and uses it to reference the function arguments.

Figure 5(b) shows the assembly code for function `fun`, while Figure 7 diagrams the stack at two points

during the execution of the function—when the function first begins execution (a), and after the first two instructions have set up the stack-frame structure for the function (b). The code for `fun` starts with the stack structure set up by `call_fun`, as shown in Figure 6(b), along with the return address pushed onto the stack by the `call` instruction. The first two instructions then set up register `%ebp` as the frame pointer, with the old value of the frame pointer stored on the stack. The function requires no additional storage, and so `%esp` and `%ebp` have the same value. Function arguments `a` and `bp` are now at offsets 8 and 12 from `%ebp`, respectively.

Implementing the expression `*bp` in `fun` requires two memory references—one (line 4) to retrieve the value of `bp` and the second (line 5) to dereference this pointer. The effect for this particular example will be to retrieve the value of `y`. The `addl` instruction (line 6) will read the value of `a` from the stack (which for this example is a copy of `x`), add it to the result of the earlier pointer dereference and store the value in register `%eax`, where it will serve as the return value for the function. The function completes by restoring `%ebp` (line 7) and executing the `ret` function.

These two examples illustrate the conventions followed by IA32 code at the beginning of a procedure call, for the case when a frame pointer is used. The general sequence is as follows:

1. A `pushl` instruction is used to push the value of the frame pointer onto the stack.
2. A `movl` instruction is used to copy the value of the stack pointer to the frame pointer, establishing the frame pointer for this procedure.
3. If any callee-saved registers need to be saved, these are pushed onto the stack via `pushl` instructions. (Neither of our examples require this.)
4. If any space needs to be allocated for local variables or for an argument build area, the stack pointer is decremented by the appropriate amount.

We also can see general patterns for how the final part of a procedure deallocates its stack frame and restores the register values. The details differ according to whether or not additional stack space was allocated and whether or not the values of callee-saved registers were pushed onto the stack, but the general idea is to restore all registers—`%esp`, `%ebp`, and any callee-saved registers to the values they had at the beginning of the procedure. The return address will then be at the top of the stack, ready for the `ret` instruction.

As a final illustration of IA32 code with frame pointers, the function `afun`, shown as both C code (a) and assembly code (b) in Figure 8, demonstrates the referencing of local variables. This function has a locally declared array `a` storing two values of type `int`. The stack structure for this function is diagrammed in Figure 9. We can see that the assembly code decrements the stack pointer by 16 (line 4), which would provide sufficient storage for 4 values of type `int`. We can see that the next two lines store the values 33 and 515 at offsets -8 and -4 from the frame pointer. We can therefore identify these locations as holding array elements `a[0]` and `a[1]`, respectively. The other two allocated locations are not used. The expression `a[t]` is then implemented with the `movl` instruction of line 8, using offset -8 from the frame pointer. These referencing patterns are consistent with the general rule of referencing local variables using negative offsets from the frame pointer, as is illustrated in Figure 4.

Practice Problem 1:

(a) C code for function `afun`

```
int afun(int t) {
    int a[2] = {33, 515};
    return a[t];
}
```

(b) Generated assembly code

```
    int afun(int t)
    t at %ebp+8
 1  afun:
 2    pushl   %ebp                        Save %ebp
 3    movl    %esp, %ebp                  Set frame pointer
 4    subl    $16, %esp                   Allocate 16 bytes
    Allocate: a[0] at %ebp-4, a[1] at %ebp-8
 5    movl    $33, -8(%ebp)               a[0] = 33
 6    movl    $515, -4(%ebp)              a[1] = 515
 7    movl    8(%ebp), %eax               Get t
 8    movl    -8(%ebp,%eax,4), %eax       Get a[t]
 9    leave                               Deallocate & restore %ebp
10    ret
```

Figure 8: **Code for function `afun` using frame pointers.** Local array `a` is referenced relative to the frame pointer.

Figure 10(a) shows C code for a function `proc` with 6 arguments. It is a simplified version of the program shown in CS:APP3e Figure 3.29. When compiled for IA32 using frame pointers, the following assembly code is generated:

```
   void proc(int a2, int *a2p, short a3, short *a3p, char a4, char *a4p)
 1 proc:
 2   pushl   %ebp
 3   movl    %esp, %ebp
 4   pushl   %ebx
 5   movl    12(%ebp), %ecx
 6   movl    20(%ebp), %edx
 7   movl    28(%ebp), %eax
 8   movl    8(%ebp), %ebx
 9   addl    %ebx, (%ecx)
10   movl    16(%ebp), %ecx
11   addw    %cx, (%edx)
12   movl    24(%ebp), %edx
13   addb    %dl, (%eax)
14   popl    %ebx
15   popl    %ebp
16   ret
```

A. What offsets, relative to `%ebp` does the function use in retrieving the arguments from the stack?

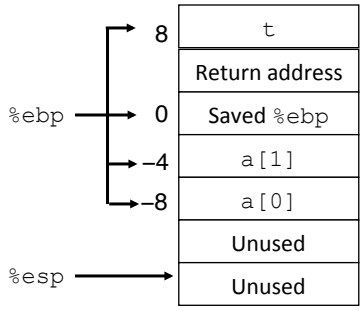

Figure 9: **Stack-frame structure for function `afun` using frame pointers.** Space for array `a` is allocated on the stack.

B. Annotate the assembly code to describe the match between its instructions and the actions described by the C code.

Practice Problem 2:

Figure 10(b) shows C code for a function that calls the function `proc` (Figure 10(a)). When compiled for IA32 using frame pointers, the following assembly code is generated:

```
   int call_proc()
1  call_proc:
2    pushl   %ebp
3    movl    %esp, %ebp
4    subl    $40, %esp
5    movl    $2, -4(%ebp)
6    movw    $3, -6(%ebp)
7    movb    $4, -7(%ebp)
8    leal    -7(%ebp), %eax
9    movl    %eax, 20(%esp)
10   movl    $4, 16(%esp)
11   leal    -6(%ebp), %eax
12   movl    %eax, 12(%esp)
13   movl    $3, 8(%esp)
14   leal    -4(%ebp), %eax
15   movl    %eax, 4(%esp)
16   movl    $2, (%esp)
17   call    proc
18   movswl  -6(%ebp), %eax
19   movsbl  -7(%ebp), %edx
20   subl    %edx, %eax
21   imull   -4(%ebp), %eax
22   leave
23   ret
```

A. How much space is allocated on the stack for the local variables and argument build area?

(a) C code for `proc`

```
void proc(int   a2, int   *a2p,
          short a3, short *a3p,
          char  a4, char  *a4p)
{
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

(b) C code for `call_proc`

```
int call_proc()
{
    int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x2, &x2, x3, &x3, x4, &x4);
    return x2*(x3-x4);
}
```

Figure 10: **C Code for practice problems.** These functions illustrate a number of issues regarding parameter passing.

B. What local variables are generated and where are they stored within the stack frame, relative to `%ebp`?

C. Where is each of the six arguments stored on the stack?

D. Annotate the assembly code to describe the match between its instructions and the actions described by the C code.

4.2 Modern IA32: No Frame Pointers

More recent versions of GCC generate IA32 code without frame pointers. Figure 11 diagrams the general structure of the stack when following this convention. There is no need to save register `%ebp` at the beginning of the stack frame. Instead, `%ebp` can be used as a callee-saved program register, pushed onto the stack by any procedure that uses it, and popped back off before the procedure returns. As indicated along the left-hand side of the figure, all references to elements of the stack frame—arguments, local variables, and the argument build area are done using offsets relative to the stack pointer.

Figure 12 shows the IA32 code generated for functions `call_fun` and `fun`, defined by the C code in Figure 3. Compared to the code using frame pointers (Figure 5), we can see that omitting frame pointers reduces the number of instructions required at the beginning and end of procedures.

Figure 13 shows the stack-frame structure for function `call_fun`. At the entry to the function (a), we see that the return address and the two arguments are at the top of the stack. The function decrements the stack pointer by 8 (line 2), having the effect of shifting the offsets for accessing arguments `x` and `y` to 12 and 16,

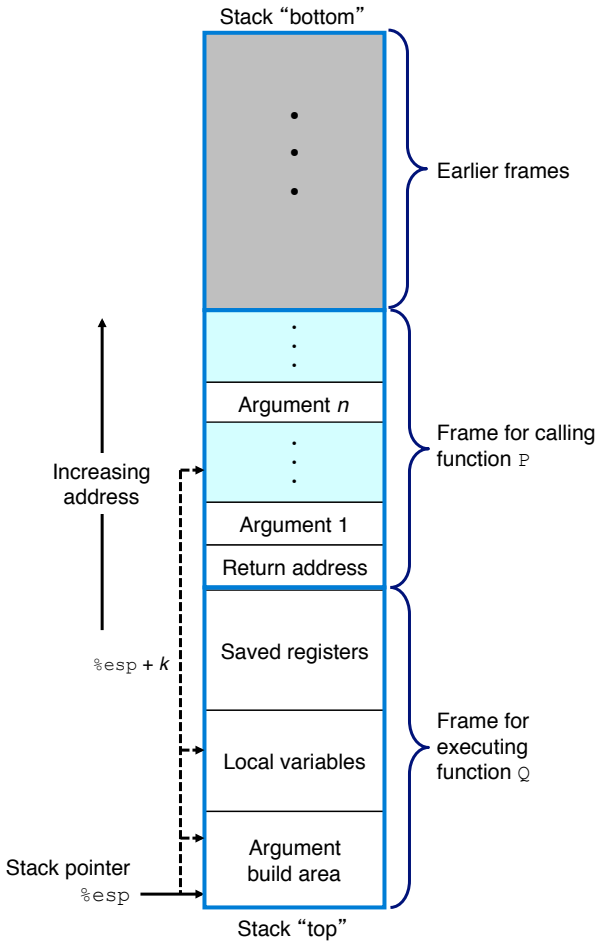

Figure 11: **General IA32 stack-frame structure without any frame pointers.** All stack references are relative to the stack pointer, held in register `%esp`.

respectively (b). We follow a practice of documenting such shifts in our annotated assembly code. In the subsequent code (lines 3–6), we see references to the arguments at offsets 12 and 16, and to the argument build area with offsets 0 an 4.

The generated assembly code for function `fun` (Figure 12(b)) is especially simple, since it does not need to allocate any additional space. We leave it to the reader to study this code.

Figure 14 shows the generated assembly code for our function having a local, two-element array. The stack structure for this function is illustrated in Figure 15 after the stack pointer has been decremented by 16 (line 2), and the array has been initialized with its two elements (lines 3–4). The main point to note here is that the references to both argument `t` and the two array elements are done relative to the stack pointer.

We can see by these examples that frame pointers are not necessarily required to support procedures. Compilers can readily keep track of the changing offsets of procedure arguments, local variables, and temporary items as stack space is allocated and deallocated. Omitting frame pointers reduces the number of instruc-

(a) Assembly code for `call_fun`

```
   int call_fun(int x, int y)
   x at %esp+4, y at %esp+8
1  call_fun:
2    subl    $8, %esp                  Allocate 8 bytes
   Now: x at %esp+12, y at %esp+16
   Store arguments at offsets 0 and 4 from %esp
3    leal    16(%esp), %eax            Compute &y
4    movl    %eax, 4(%esp)             Store as 2nd argument
5    movl    12(%esp), %eax            Get x
6    movl    %eax, (%esp)              Store as 1st argument
7    call    fun                       fun(x, &y)
8    addl    $8, %esp                  Deallocate 8 bytes
9    ret
```

(b) Assembly code for `fun`

```
   int fun(int a, int *bp)
   a at %esp+4, bp at %esp+8
1  fun:
2    movl    8(%esp), %eax             Get bp
3    movl    (%eax), %eax              Get *bp
4    addl    4(%esp), %eax             Add a
5    ret
```

Figure 12: **IA32 assembly code for example functions without frame pointers pointers.** Arguments are passed to the functions on the stack.

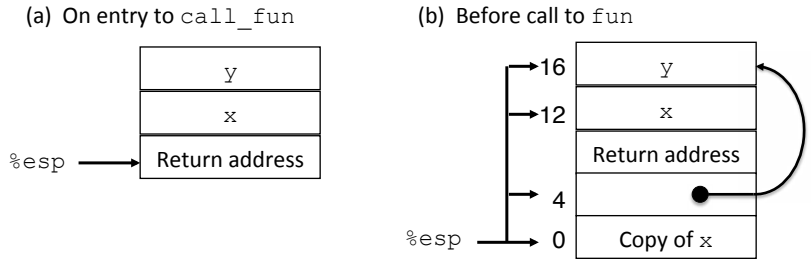

Figure 13: **Stack-frame structure for function `call_fun` without frame pointers.** The function allocates space on the stack.

tions to implement procedures, and it allows register `%ebp` to be used for other program data. It makes sense that modern compilers follow this convention when they can. The only case where frame pointers are truly needed is when the amount of stack space required for a procedure varies from one call to another, as we saw in CS:APP3e Section 3.10.5.

Practice Problem 3:

When the C code of Figure 10(a) is compiled for IA32 without frame pointers, the following assembly code is generated:

```
   void proc(int a2, int *a2p, short a3, short *a3p, char a4, char *a4p)
 1 proc:
 2   pushl   %ebx
 3   movl    12(%esp), %ecx
 4   movl    20(%esp), %edx
 5   movl    28(%esp), %eax
 6   movl    8(%esp), %ebx
 7   addl    %ebx, (%ecx)
 8   movl    16(%esp), %ecx
 9   addw    %cx, (%edx)
10   movl    24(%esp), %edx
11   addb    %dl, (%eax)
12   popl    %ebx
13   ret
```

A. What offsets, relative to `%esp`, does the function use in retrieving the arguments from the stack?

B. Annotate the assembly code to describe the match between its instructions and the actions described by the C code.

Practice Problem 4:

When the C code of Figure 10(b) is compiled for IA32 without frame pointers, the following assembly code is generated:

```
int call_proc()
```

(a) C code for function `afun`

```
int afun(int t) {
    int a[2] = {33, 515};
    return a[t];
}
```

(b) Generated assembly code

```
    int afun(int t)
    t at %esp+4
1 afun:
2   subl    $16, %esp                   Allocate 16 bytes
    Now: t at %esp+20
    Allocate: a[0] at %esp+8, a[1] at %esp+12
3   movl    $33, 8(%esp)                a[0] = 33
4   movl    $515, 12(%esp)              a[1] = 515
5   movl    20(%esp), %eax              Get t
6   movl    8(%esp,%eax,4), %eax        Get a[t]
7   addl    $16, %esp                   Deallocate 16 bytes
8   ret
```

Figure 14: **Code for function `afun` without frame pointers.** Local array `a` is referenced relative to the stack pointer.

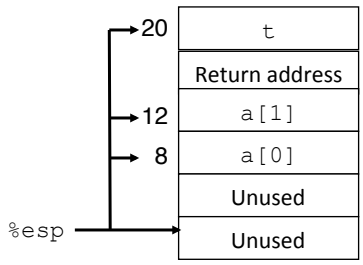

Figure 15: **Stack-frame structure for function `afun` without frame pointers.** Space for array `a` is allocated on the stack.

```
1  call_proc:
2    subl    $40, %esp
3    movl    $2, 36(%esp)
4    movw    $3, 34(%esp)
5    movb    $4, 33(%esp)
6    leal    33(%esp), %eax
7    movl    %eax, 20(%esp)
8    movl    $4, 16(%esp)
9    leal    34(%esp), %eax
10   movl    %eax, 12(%esp)
11   movl    $3, 8(%esp)
12   leal    36(%esp), %eax
13   movl    %eax, 4(%esp)
14   movl    $2, (%esp)
15   call    proc
16   movswl  34(%esp), %eax
17   movsbl  33(%esp), %edx
18   subl    %edx, %eax
19   imull   36(%esp), %eax
20   addl    $40, %esp
21   ret
```

A. How much space is allocated on the stack for the local variables and argument build area?

B. What local variables are generated and where are they stored within the stack frame, relative to `%esp`?

C. Where is each of the six arguments stored on the stack?

D. Annotate the assembly code to describe the match between its instructions and the actions described by the C code.

5 Summary

IA32 machine code is still widely used, both on small processors that only support 32-bit operation, and on x86-64 machines running in compatibility mode. As has been demonstrated in this document, people experienced in reading x86-64 code can rapidly gain the expertise needed to read and understand IA32 code. The most significant difference is that IA32 code tends to make much greater use of the stack, due to the argument-passing convention and to the fact that there are only eight general-purpose registers.

We have seen two variants: the classic approaching using a frame pointer and the more modern approach of relying solely on the stack pointer to reference objects within the stack frame.

Answers to Practice Problems

Problem 1 Solution: [Pg. 9]

Here is an annotated version of the code. The answers to all of the questions can be gleaned from the annotations.

```
   void proc(int a2, int *a2p, short a3, short *a3p, char a4, char *a4p)
   a2 at %ebp+8, a2p at %ebp+12, a3 at %ebp+16, a3p at %ebp+20
   a4 at %ebp+24, a4p at %ebp+28
 1 proc:
 2   pushl   %ebp                        Save %ebp
 3   movl    %esp, %ebp                  Set frame pointer
 4   pushl   %ebx                        Save %ebx
 5   movl    12(%ebp), %ecx              Get a2p
 6   movl    20(%ebp), %edx              Get a3p
 7   movl    28(%ebp), %eax              Get a4p
 8   movl    8(%ebp), %ebx               Get a2
 9   addl    %ebx, (%ecx)                *a2p += a2
10   movl    16(%ebp), %ecx              Get a3
11   addw    %cx, (%edx)                 *a3p += a3
12   movl    24(%ebp), %edx              Get a4
13   addb    %dl, (%eax)                 *a4p += a4
14   popl    %ebx                        Restore %ebx
15   popl    %ebp                        Restore %ebp
16   ret
```

Problem 2 Solution: [Pg. 11]

Here is an annotated version of the code:

```
   int call_proc()
 1 call_proc:
 2   pushl   %ebp                        Save %ebp
 3   movl    %esp, %ebp                  Set frame pointer
 4   subl    $40, %esp                   Allocate 40 bytes
   Allocation: x2 at %ebp-4, x3 at %ebp-6, x4 at %ebp-7
               Arguments at offsets 0, 4, 8, 12, 16, and 20 to %esp
 5   movl    $2, -4(%ebp)                Set x2 = 2
 6   movw    $3, -6(%ebp)                Set x3 = 3
 7   movb    $4, -7(%ebp)                Set x4 = 4
 8   leal    -7(%ebp), %eax              Compute &x4
 9   movl    %eax, 20(%esp)              Store as 6th argument
10   movl    $4, 16(%esp)                Store 4 as 5th argument
11   leal    -6(%ebp), %eax              Compute &x3
12   movl    %eax, 12(%esp)              Store as 4th argument
13   movl    $3, 8(%esp)                 Store 3 as 3rd argument
14   leal    -4(%ebp), %eax              Compute &x2
15   movl    %eax, 4(%esp)               Store as 2nd argument
16   movl    $2, (%esp)                  Store 2 as 1st argument
17   call    proc                        proc(x2, &x2, x3, &x3, x4, &x4)
18   movswl  -6(%ebp), %eax              Get x3
19   movsbl  -7(%ebp), %edx              Get x4
20   subl    %edx, %eax                  Compute x3-x4
```

```
21   imull   -4(%ebp), %eax            Multiply by x2
22   leave                             Deallocate & restore %ebp
23   ret
```

Problem 3 Solution: [Pg. 15]

Here is an annotated version of the code:

```
   void proc(int a2, int *a2p, short a3, short *a3p, char a4, char *a4p)
   a2 at %esp+4, a2p at %esp+8, a3 at %esp+12, a3p at %esp+16
   a4 at %esp+20, a4p at %esp+24
1  proc:
2    pushl   %ebx                      Save %ebx
   Now: a2 at %esp+8, a2p at %esp+12, a3 at %esp+16, a3p at %esp+20
        a4 at %esp+24, a4p at %esp+28
3    movl    12(%esp), %ecx            Get a2p
4    movl    20(%esp), %edx            Get a3p
5    movl    28(%esp), %eax            Get a4p
6    movl    8(%esp), %ebx             Get a2
7    addl    %ebx, (%ecx)              *a2p += a2
8    movl    16(%esp), %ecx            Get a3
9    addw    %cx, (%edx)               *a3p += a3
10   movl    24(%esp), %edx            Get a4
11   addb    %dl, (%eax)               *a4p += a4
12   popl    %ebx                      Restore %ebx
13   ret
```

Problem 4 Solution: [Pg. 15]

Here is an annotated version of the code:

```
1  call_proc:
2    subl    $40, %esp                 Allocate 40 bytes
   Allocation: x2 at %esp+36, x3 at %esp+34, x4 at %esp+33
               Arguments at offsets 0, 4, 8, 12, 16, and 20 to %esp
3    movl    $2, 36(%esp)              Set x2 = 2
4    movw    $3, 34(%esp)              Set x3 = 3
5    movb    $4, 33(%esp)              Set x4 = 4
6    leal    33(%esp), %eax            Compute &x4
7    movl    %eax, 20(%esp)            Store as 6th argument
8    movl    $4, 16(%esp)              Store 4 as 5th argument
9    leal    34(%esp), %eax            Compute &x3
10   movl    %eax, 12(%esp)            Store as 4th argument
11   movl    $3, 8(%esp)               Store 3 as 3rd argument
12   leal    36(%esp), %eax            Compute &x2
13   movl    %eax, 4(%esp)             Store as 2nd argument
14   movl    $2, (%esp)                Store 2 as 1st argument
15   call    proc                      proc(x2, &x2, x3, &x3, x4, &x4)
16   movswl  34(%esp), %eax            Get x3
17   movsbl  33(%esp), %edx            Get x4
```

```
18    subl    %edx, %eax              Compute x3-x4
19    imull   36(%esp), %eax          Multiply by x2
20    addl    $40, %esp               Deallocate 40 bytes
21    ret
```

References

[1] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*. Available at `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.