Alternatively, this preparation can be performed by an explicit sequence of move and pop operations. Register %eax is used for returning the value from any function that returns an integer or pointer.

Practice Problem 3.30

The following code fragment occurs often in the compiled version of library routines:

```
1     call next
2   next:
3     popl %eax
```

A. To what value does register %eax get set?

B. Explain why there is no matching ret instruction to this call.

C. What useful purpose does this code fragment serve?

3.7.3 Register Usage Conventions

The set of program registers acts as a single resource shared by all of the procedures. Although only one procedure can be active at a given time, we must make sure that when one procedure (the *caller*) calls another (the *callee*), the callee does not overwrite some register value that the caller planned to use later. For this reason, IA32 adopts a uniform set of conventions for register usage that must be respected by all procedures, including those in program libraries.

By convention, registers %eax, %edx, and %ecx are classified as *caller-save* registers. When procedure Q is called by P, it can overwrite these registers without destroying any data required by P. On the other hand, registers %ebx, %esi, and %edi are classified as *callee-save* registers. This means that Q must save the values of any of these registers on the stack before overwriting them, and restore them before returning, because P (or some higher-level procedure) may need these values for its future computations. In addition, registers %ebp and %esp must be maintained according to the conventions described here.

As an example, consider the following code:

```
1   int P(int x)
2   {
3       int y = x*x;
4       int z = Q(y);
5       return y + z;
6   }
```

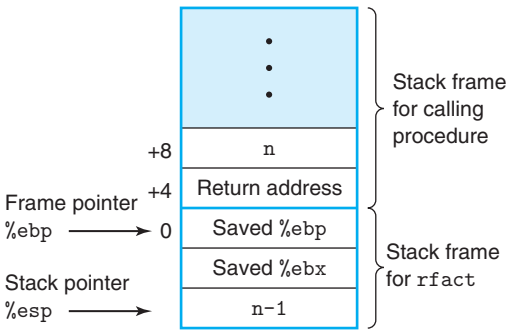

Figure 3.27
Stack frame for recursive factorial function. The state of the frame is shown just before the recursive call.

the value of $(n-1)!$ and (2) callee-save register %ebx holds the parameter n. It therefore multiplies these two quantities (line 13) to generate the return value of the function.

For both cases—the terminal condition and the recursive call—the code proceeds to the completion section (lines 15–17) to restore the stack and callee-saved register, and then it returns.

We can see that calling a function recursively proceeds just like any other function call. Our stack discipline provides a mechanism where each invocation of a function has its own private storage for state information (saved values of the return location, frame pointer, and callee-save registers). If need be, it can also provide storage for local variables. The stack discipline of allocation and deallocation naturally matches the call-return ordering of functions. This method of implementing function calls and returns even works for more complex patterns, including mutual recursion (for example, when procedure P calls Q, which in turn calls P).

Practice Problem 3.34

For a C function having the general structure

```
int rfun(unsigned x) {
    if ( __________ )
        return __________;
    unsigned nx = __________;
    int rv = rfun(nx);
    return __________;
}
```

GCC generates the following assembly code (with the setup and completion code omitted):

```
1     movl    8(%ebp), %ebx
2     movl    $0, %eax
3     testl   %ebx, %ebx
4     je      .L3
```

```
 5      movl    %ebx, %eax
 6      shrl    %eax              Shift right by 1
 7      movl    %eax, (%esp)
 8      call    rfun
 9      movl    %ebx, %edx
10      andl    $1, %edx
11      leal    (%edx,%eax), %eax
12    .L3:
```

A. What value does `rfun` store in the callee-save register `%ebx`?

B. Fill in the missing expressions in the C code shown above.

C. Describe in English what function this code computes.

3.8 Array Allocation and Access

Arrays in C are one means of aggregating scalar data into larger data types. C uses a particularly simple implementation of arrays, and hence the translation into machine code is fairly straightforward. One unusual feature of C is that we can generate pointers to elements within arrays and perform arithmetic with these pointers. These are translated into address computations in machine code.

Optimizing compilers are particularly good at simplifying the address computations used by array indexing. This can make the correspondence between the C code and its translation into machine code somewhat difficult to decipher.

3.8.1 Basic Principles

For data type T and integer constant N, the declaration

T `A[`N`];`

has two effects. First, it allocates a contiguous region of $L \cdot N$ bytes in memory, where L is the size (in bytes) of data type T. Let us denote the starting location as $x_{\mathtt{A}}$. Second, it introduces an identifier `A` that can be used as a pointer to the beginning of the array. The value of this pointer will be $x_{\mathtt{A}}$. The array elements can be accessed using an integer index ranging between 0 and $N-1$. Array element i will be stored at address $x_{\mathtt{A}} + L \cdot i$.

As examples, consider the following declarations:

```
char    A[12];
char   *B[8];
double  C[6];
double *D[5];
```

of type T, and the value of `p` is x_p, then the expression `p+i` has value $x_p + L \cdot i$, where L is the size of data type T.

The unary operators `&` and `*` allow the generation and dereferencing of pointers. That is, for an expression *Expr* denoting some object, `&`*Expr* is a pointer giving the address of the object. For an expression *AExpr* denoting an address, `*`*AExpr* gives the value at that address. The expressions *Expr* and `*&`*Expr* are therefore equivalent. The array subscripting operation can be applied to both arrays and pointers. The array reference `A[i]` is identical to the expression `*(A+i)`. It computes the address of the *i*th array element and then accesses this memory location.

Expanding on our earlier example, suppose the starting address of integer array `E` and integer index i are stored in registers `%edx` and `%ecx`, respectively. The following are some expressions involving `E`. We also show an assembly-code implementation of each expression, with the result being stored in register `%eax`.

Expression	Type	Value	Assembly code
`E`	`int *`	x_E	`movl %edx,%eax`
`E[0]`	`int`	$M[x_E]$	`movl (%edx),%eax`
`E[i]`	`int`	$M[x_E + 4i]$	`movl (%edx,%ecx,4),%eax`
`&E[2]`	`int *`	$x_E + 8$	`leal 8(%edx),%eax`
`E+i-1`	`int *`	$x_E + 4i - 4$	`leal -4(%edx,%ecx,4),%eax`
`*(E+i-3)`	`int *`	$M[x_E + 4i - 12]$	`movl -12(%edx,%ecx,4),%eax`
`&E[i]-E`	`int`	i	`movl %ecx,%eax`

In these examples, the `leal` instruction is used to generate an address, while `movl` is used to reference memory (except in the first and last cases, where the former copies an address and the latter copies the index). The final example shows that one can compute the difference of two pointers within the same data structure, with the result divided by the size of the data type.

Practice Problem 3.36

Suppose the address of `short` integer array `S` and integer index i are stored in registers `%edx` and `%ecx`, respectively. For each of the following expressions, give its type, a formula for its value, and an assembly code implementation. The result should be stored in register `%eax` if it is a pointer and register element `%ax` if it is a `short` integer.

Expression	Type	Value	Assembly code
`S+1`			
`S[3]`			
`&S[i]`			
`S[4*i+1]`			
`S+i-5`			

3.56 ◆◆

Consider the following assembly code:

```
    x at %ebp+8, n at %ebp+12
 1    movl    8(%ebp), %esi
 2    movl    12(%ebp), %ebx
 3    movl    $-1, %edi
 4    movl    $1, %edx
 5  .L2:
 6    movl    %edx, %eax
 7    andl    %esi, %eax
 8    xorl    %eax, %edi
 9    movl    %ebx, %ecx
10    sall    %cl, %edx
11    testl   %edx, %edx
12    jne     .L2
13    movl    %edi, %eax
```

The preceding code was generated by compiling C code that had the following overall form:

```
1  int loop(int x, int n)
2  {
3      int result = _____;
4      int mask;
5      for (mask = _____; mask _____ ; mask = _____) {
6          result ^= _____;
7      }
8      return result;
9  }
```

Your task is to fill in the missing parts of the C code to get a program equivalent to the generated assembly code. Recall that the result of the function is returned in register `%eax`. You will find it helpful to examine the assembly code before, during, and after the loop to form a consistent mapping between the registers and the program variables.

A. Which registers hold program values `x`, `n`, `result`, and `mask`?

B. What are the initial values of `result` and `mask`?

C. What is the test condition for `mask`?

D. How does `mask` get updated?

E. How does `result` get updated?

F. Fill in all the missing parts of the C code.

3.57 ◆◆

In Section 3.6.6, we examined the following code as a candidate for the use of conditional data transfer:

A. What would be the offsets (in bytes) of the following fields:

```
e1.p:
e1.y:
e2.x:
e2.next:
```

B. How many total bytes would the structure require?

C. The compiler generates the following assembly code for the body of `proc`:

```
   up at %ebp+8
1     movl    8(%ebp), %edx
2     movl    4(%edx), %ecx
3     movl    (%ecx), %eax
4     movl    (%eax), %eax
5     subl    (%edx), %eax
6     movl    %eax, 4(%ecx)
```

On the basis of this information, fill in the missing expressions in the code for `proc`. **Hint:** Some union references can have ambiguous interpretations. These ambiguities get resolved as you see where the references lead. There is only one answer that does not perform any casting and does not violate any type constraints.

3.68 ◆

Write a function `good_echo` that reads a line from standard input and writes it to standard output. Your implementation should work for an input line of arbitrary length. You may use the library function `fgets`, but you must make sure your function works correctly even when the input line requires more space than you have allocated for your buffer. Your code should also check for error conditions and return when one is encountered. Refer to the definitions of the standard I/O functions for documentation [48, 58].

3.69 ◆

The following declaration defines a class of structures for use in constructing binary trees:

```
1   typedef struct ELE *tree_ptr;
2
3   struct ELE {
4       long val;
5       tree_ptr left;
6       tree_ptr right;
7   };
```

For a function with the following prototype:

```
long trace(tree_ptr tp);
```

GCC generates the following x86-64 code:

```
1   trace:
    tp in %rdi
2     movl    $0, %eax
3     testq   %rdi, %rdi
4     je      .L3
5   .L5:
6     movq    (%rdi), %rax
7     movq    16(%rdi), %rdi
8     testq   %rdi, %rdi
9     jne     .L5
10  .L3:
11    rep
12    ret
```

A. Generate a C version of the function, using a `while` loop.

B. Explain in English what this function computes.

3.70 ◆◆

Using the same tree data structure we saw in Problem 3.69, and a function with the prototype

```
long traverse(tree_ptr tp);
```

GCC generates the following x86-64 code:

```
1   traverse:
    tp in %rdi
2     movq    %rbx, -24(%rsp)
3     movq    %rbp, -16(%rsp)
4     movq    %r12, -8(%rsp)
5     subq    $24, %rsp
6     movq    %rdi, %rbp
7     movabsq $-9223372036854775808, %rax
8     testq   %rdi, %rdi
9     je      .L9
10    movq    (%rdi), %rbx
11    movq    8(%rdi), %rdi
12    call    traverse
13    movq    %rax, %r12
14    movq    16(%rbp), %rdi
15    call    traverse
```