x86-64 Machine-Level Programming*

Randal E. Bryant
David R. O'Hallaron

September 9, 2005

Intel's IA32 instruction set architecture (ISA), colloquially known as "x86", is the dominant instruction format for the world's computers. IA32 is the platform of choice for most Windows and Linux machines. The ISA we use today was defined in 1985 with the introduction of the i386 microprocessor, extending the 16-bit instruction set defined by the original 8086 to 32 bits. Even though subsequent processor generations have introduced new instruction types and formats, many compilers, including GCC, have avoided using these features in the interest of maintaining backward compatibility.

A shift is underway to a 64-bit version of the Intel instruction set. Originally developed by Advanced Micro Devices (AMD) and named *x86-64*, it is now supported by high end processors from AMD (who now call it *AMD64*) and by Intel, who refer to it as *EM64T*. Most people still refer to it as "x86-64," and we follow this convention. Newer versions of Linux and GCC support this extension. In making this switch, the developers of GCC saw an opportunity to also make use of some of the instruction-set features that had been added in more recent generations of IA32 processors.

This combination of new hardware and revised compiler makes x86-64 code substantially different in form and in performance than IA32 code. In creating the 64-bit extension, the AMD engineers also adopted some of the features found in reduced-instruction set computers (RISC) [7] that made them the favored targets for optimizing compilers. For example, there are now 16 general-purpose registers, rather than the performance-limiting eight of the original 8086. The developers of GCC were able to exploit these features, as well as those of more recent generations of the IA32 architecture, to obtain substantial performance improvements. For example, procedure parameters are now passed via registers rather than on the stack, greatly reducing the number of memory read and write operations.

This document serves as a supplement to Chapter 3 of *Computer Systems: A Programmer's Perspective* (CS:APP), describing some of the differences. We start with a brief history of how AMD and Intel arrived at x86-64, followed by a summary of the main features that distinguish x86-64 code from IA32 code, and then work our way through the individual features.

1 History and Motivation for x86-64

Over the twenty years since the introduction of the i386, the capabilities of microprocessors have changed dramatically. In 1985, a fully configured, high-end personal computer had around 1 megabyte of random-access memory (RAM) and 50 megabytes of disk storage. Microprocessor-based "workstation" systems were just becoming the machines of choice for computing and engineering professionals. A typical microprocessor had a 5-megahertz clock and ran around one million instructions per second. Nowadays, a typical high-end system has 1 gigabyte of RAM, 500 gigabytes of disk storage, and a 4-gigahertz clock, running around 5 billion instructions per second. Microprocessor-based systems have become pervasive. Even today's supercomputers are based on harnessing the power of many microprocessors computing in parallel. Given these large quantitative improvements, it is remarkable that the world's computing base mostly runs code that is binary compatible with machines that existed 20 years ago.

The 32-bit word size of the IA32 has become a major limitation in growing the capacity of microprocessors. Most significantly, the word size of a machine defines the range of virtual addresses that programs can use, giving a 4-gigabyte virtual address space in the case of 32 bits. It is now feasible to buy more than this amount of RAM for a machine, but the system cannot make effective use of it. For applications that involve manipulating large data sets, such as scientific computing, databases, and data mining, the 32-bit word size makes life difficult for programmers. They must write code using *out-of-core* algorithms[1], where the data reside on disk and are explicitly read into memory for processing.

Further progress in computing technology requires a shift to a larger word size. Following the tradition of growing word sizes by doubling, the next logical step is 64 bits. In fact, 64-bit machines have been available for some time. Digital Equipment Corporation introduced its Alpha processor in 1992, and it became a popular choice for high-end computing. Sun Microsystems introduced a 64-bit version of its SPARC architecture in 1995. At the time, however, Intel was not a serious contender for high-end computers, and so the company was under less pressure to switch to 64 bits.

Intel's first foray into 64-bit computers were the Itanium processors, based on the IA64 instruction set. Unlike Intel's historic strategy of maintaining backward compatibility as it introduced each new generation of microprocessor, IA64 is based on a radically new approach jointly developed with Hewlett-Packard. Its *Very Large Instruction Word* (VLIW) format packs multiple instructions into bundles, allowing higher degrees of parallel execution. Implementing IA64 proved to be very difficult, and so the first Itanium chips did not appear until 2001, and these did not achieve the expected level of performance on real applications. Although the performance of Itanium-based systems has improved, they have not captured a significant share of the computer market. Itanium machines can execute IA32 code in a compatibility mode but not with very good performance. Most users have preferred to make do with less expensive, and often faster, IA32-based systems.

Meanwhile, Intel's archrival, Advanced Micro Devices (AMD) saw an opportunity to exploit Intel's misstep with IA64. For years AMD had lagged just behind Intel in technology, and so they were relegated to competing with Intel on the basis of price. Typically, Intel would introduce a new microprocessor at a price premium. AMD would come along 6 to 12 months later and have to undercut Intel significantly to get any sales—a strategy that worked but yielded very low profits. In 2002, AMD introduced a 64-bit

[1]The physical memory of a machine is often referred to as *core memory*, dating to an era when each bit of a random-access memory was implemented with a magnetized ferrite core.

microprocessor based on its "x86-64" instruction set. As the name implies, x86-64 is an evolution of the Intel instruction set to 64 bits. It maintains full backward compatibility with IA32, but it adds new data formats, as well as other features that enable higher capacity and higher performance. With x86-64, AMD has sought to capture some of the high-end market that had historically belonged to Intel. AMD's recent generations of Opteron and Athlon 64 processors have indeed proved very successful as high performance machines. Most recently, AMD has renamed this instruction set *AMD64*, but "x86-64" persists as the favored name.

Intel realized that its strategy of a complete shift from IA32 to IA64 was not working, and so began supporting their own variant of x86-64 in 2004 with processors in the Pentium 4 Xeon line. Since they had already used the name "IA64" to refer to Itanium, they then faced a difficulty in finding their own name for this 64-bit extension. In the end, they decided to describe x86-64 as an enhancement to IA32, and so they refer to it as *IA32-EM64T* for "Enhanced Memory 64-bit Technology."

The developers of GCC steadfastly maintained binary compatibility with the i386, even though useful features had been added to the IA32 instruction set. The PentiumPro introduced a set of conditional move instructions that could greatly improve the performance of code involving conditional operations. More recent generations of Pentium processors introduced new floating point operations that could replace the rather awkward and quirky approach dating back to the 8087, the floating point coprocessor that accompanied the 8086 and is now incorporated within the main microprocessors chips. Switching to x86-64 as a target provided an opportunity for GCC to give up backward compatibility and instead exploit these newer features.

In this document, we use "IA32" to refer to the combination of hardware and GCC code found in traditional, 32-bit versions of Linux running on Intel-based machines. We use "x86-64" to refer to the hardware and code combination running on the newer 64-bit machines from AMD and Intel. In the Linux world, these two platforms are referred to as "i386" and "x86_64," respectively.

2 Finding Documentation

Both Intel and AMD provide extensive documentation on their processors. This includes general overviews of the assembly language programmer's view of the hardware [2, 4], as well as detailed references about the individual instructions [3, 5, 6]. The organization `amd64.org` has been responsible for defining the *Application Binary Interface* (ABI) for x86-64 code running on Linux systems [8]. This interface describes details for procedure linkages, binary code files, and a number of other features that are required for object code programs to execute properly.

Warning: Both the Intel and the AMD documentation use the Intel assembly code notation. This differs from the notation used by the Gnu assembler GAS. Most significantly, it lists operands in the opposite order.

3 An Overview of x86-64

The combination of the new hardware supplied by Intel and AMD, as well as the new version of GCC targeting these machines makes x86-64 code substantially different from that generated for IA32 machines.

C declaration	Intel data type	GAS suffix	x86-64 Size (Bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`unsigned`	Double word	`l`	4
`long int`	Quad word	`q`	8
`unsigned long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`d`	8
`long double`	Extended precision	`t`	16

Figure 1: **Sizes of standard data types with x86-64** Both long integers and pointers require 8 bytes, as compared to 4 for IA32.

The main features include:

- Pointers and long integers are 64 bits long. Integer arithmetic operations support 8, 16, 32, and 64-bit data types.
- The set of general-purpose registers is expanded from 8 to 16.
- Much of the program state is held in registers rather than on the stack. Integer and pointer procedure arguments (up to 6) are passed via registers. Some procedures do not need to access the stack at all.
- Conditional operations are implemented using conditional move instructions when possible, yielding better performance than traditional branching code.
- Floating-point operations are implemented using a register-oriented instruction set, rather than the stack-based approach supported by IA32.

3.1 Data Types

Figure 1 shows the sizes of different C data types for x86-64. Comparing these to the IA32 sizes (CS:APP Figure 3.1), we see that pointers (shown here as data type `char *`) require 8 bytes rather than 4. In principal, this gives programs the ability to access 16 *exabytes* of memory (around 18.4×10^{18} bytes). That seems like an astonishing amount of memory, but keep in mind that 4 gigabytes seemed astonishing when the first 32-bit machines appeared in the late 1970s. In practice, most machines do not really support the full address range—the current generations of AMD and Intel x86-64 machines support 256 terabytes (2^{48}) bytes of virtual memory—but allocating this much memory for pointers is a good idea for long term compatibility.

We also see that the prefix "`long`" changes integers to 64 bits, allowing a considerably larger range of values. Whereas a 32-bit unsigned value can range up to 4,294,967,295 (CS:APP Figure 2.8), increasing the word size to 64 bits gives a maximum value of 18,446,744,073,709,551,615.

As with IA32, the `long` prefix also changes a floating point `double` to use the 80-bit format supported by IA32 (CS:APP Section 2.4.6.) These are stored in memory with an allocation of 16 bytes for x86-64, compared to 12 bytes for IA32. This improves the performance of memory read and write operations, which typically fetch 8 or 16 bytes at a time. Whether 12 or 16 bytes are allocated, only the low-order 10 bytes are actually used.

3.2 Assembly Code Example

Section 3.2.3 of CS:APP illustrated the IA32 assembly code generated by GCC for a function `simple`. Below is the C code for `simple_l`, similar to `simple`, except that it uses long integers:

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

When GCC is run on an x86-64 machine with the command line

```
unix> gcc -O2 -S -m32 code.c
```

it generates code that is compatible with any IA32 machine:

```
    IA32 version of function simple_l.
    Arguments in stack locations 8(%ebp) (xp) and 12(%ebp) (y)
1 simple_l:
2   pushl   %ebp              Save frame pointer
3   movl    %esp, %ebp        Create new frame pointer
4   movl    8(%ebp), %edx     Get xp
5   movl    (%edx), %eax      Retrieve *xp
6   addl    12(%ebp), %eax    Add y to get t (and return value)
7   movl    %eax, (%edx)      Store t at *xp
8   leave                     Restore stack and frame pointers
9   ret                       Return
```

This code is almost identical to that shown in CS:APP, except that it uses the single `leave` instruction (CS:APP Section 3.7.2), rather than the sequence `movl %ebp, %esp` and `popl %ebp` to deallocate the stack frame.

When we instruct GCC to generate x86-64 code

```
unix> gcc -O2 -S -m64 code.c
```

(on most machines, the flag `-m64` is not required), we get very different code:

```
    x86-64 version of function simple_l.
    Arguments in registers %rdi (xp) and %rsi (y)
```

```
1 simple_l:
2   addq    (%rdi), %rsi      Add *xp to y to get t
3   movq    %rsi, %rax        Set t as return value
4   movq    %rsi, (%rdi)      Store t at *xp
5   ret                       Return
```

Some of the key differences include

- Instead of `movl` and `addl` instructions, we see `movq` and `addq`. The pointers and variables declared as long integers are now 64 bits (quad words) rather than 32 bits (long words).
- We see the 64-bit versions of the registers, e.g., `%rsi`, `%rdi`. The procedure returns a value by storing it in register `%rax`.
- No stack frame gets generated in the x86-64 version. This eliminates the instructions that set up (lines 2–3) and remove (line 8) the stack frame in the IA32 code.
- Arguments `xp` and `y` are passed in registers `%rdi` and `%rsi`, rather than on the stack. These registers are the 64-bit versions of registers `%edi` and `%esi`. This eliminates the need to fetch the arguments from memory. As a consequence, the two instructions on lines 2 and 3 can retrieve `*xp`, add it to `y`, and set it as the return value, whereas the IA32 code required three lines of code: 4–6.

The net effect of these changes is that the IA32 code consists of 8 instructions making 7 memory references, while the x86-64 code consists of 4 instructions making 3 memory references. Running on an Intel Pentium 4 Xeon, our experiments show that the IA32 code requires around 17 clock cycles per call, while the x86-64 code requires 12 cycles per call. Running on an AMD Opteron, we get 9 and 7 cycles per call, respectively. Getting a performance increase of 1.3–1.4X on the same machine with the same C code is a significant achievement. Clearly x86-64 represents a important step forward.

4 Accessing Information

Figure 2 shows the set of general-purpose registers under x86-64. Compared to the registers for IA32 (CS:APP Figure 3.2), we see a number of differences:

- The number of registers has been doubled to 16. The new registers are numbered 8–15.
- All registers are 64 bits long. The 64-bit extensions of the IA32 registers are named `%rax`, `%rcx`, `%rdx`, `%rbx`, `%rsi`, `%rdi`, `%rsp`, and `%rbp`. The new registers are named `%r8`–`%r15`.
- The low-order 32 bits of each register can be accessed directly. This gives us the familiar registers from IA32: `%eax`, `%ecx`, `%edx`, `%ebx`, `%esi`, `%edi`, `%esp`, and `%ebp`, as well as eight new 32-bit registers: `%r8d`–`%r15d`.
- The low-order 16 bits of each register can be accessed directly, as is the case for IA32. The *word-size* versions of the new registers are named `%r8w`–`%r15w`.

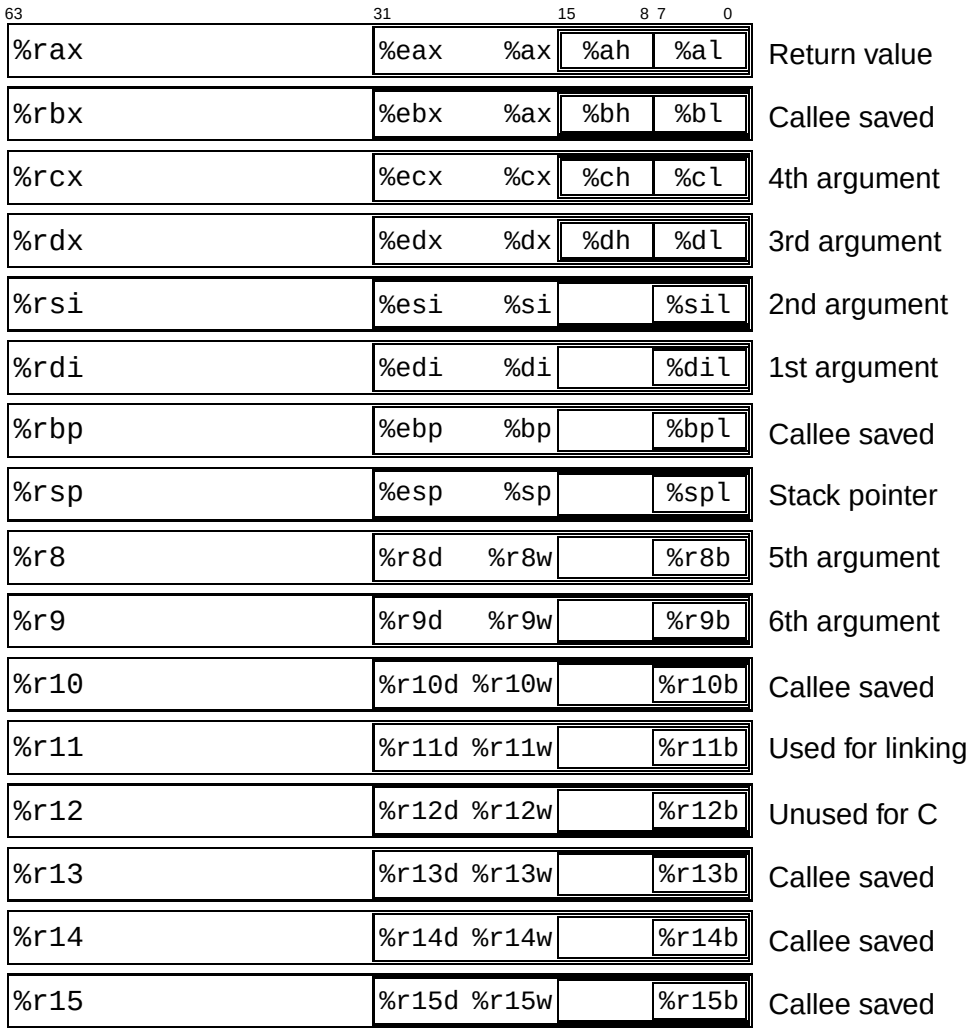


Figure 2: **Integer registers.** The existing eight registers are extended to 64-bit versions, and eight new registers are added. Each register can be accessed as either 8 bits (byte), 16 bits (word), 32 bits (double word), or 64 bits (quad word).

- The low-order 8 bits of each register can be accessed directly. This is true in IA32 only for the first 4 registers (`%al`, `%cl`, `%dl`, `%bl`). The byte-size versions of the other IA32 registers are named `%sil`, `%dil`, `%spl`, and `%bpl`. The byte-size versions of the new registers are named `%r8b`–`%r15b`.
- For backward compatibility, the second byte of registers `%rax`, `%rcx`, `%rdx`, and `%rbx` can be directly accessed by instructions having single-byte operands.

As with IA32, most of the registers can be used interchangeably, but there are some special cases. Register `%rsp` has special status, in that it holds a pointer to the top stack element. Unlike in IA32, however, there is no frame pointer register; register `%rbp` is available for use as a general-purpose register. Particular conventions are used for passing procedure arguments via registers and for how registers are to be saved and restored registers during procedure calls, as is discussed in Section 6. In addition, some arithmetic instructions make special use of registers `%rax` and `%rdx`.

For the most part, the operand specifiers of x86-64 are just the same as those in IA32 (see CS:APP Figure 3.3). One minor difference is that some forms of *PC-relative* operand addressing are supported. With IA32, this form of addressing is only supported for jump and other control transfer instructions (see CS:APP Section 3.6.3). This mode is provided to compensate for the fact that the offsets (shown in CS:APP Figure 3.3 as Imm) are only 32 bits long. By viewing this field as a 32-bit, two's complement number, instructions can access data within a window of around $\pm 2.15 \times 10^9$ relative to the program counter. With x86-64, the program counter is named `%rip`.

As an example of PC-relative data addressing, consider the following procedure, which calls the function `call_simple_l` examined earlier:

```
long int gval1 = 567;
long int gval2 = 763;

long int call_simple_l()
{
    long int z = simple_l(&gval1, 12L);
    return z + gval2;
}
```

This code references global variables `gval1` and `gval2`. When this function is compiled, assembled, and linked, we get the following executable code (as generated by the disassembler `objdump`)

```
1 0000000000400500 <call_simple_l>:
2   400500:  be 0c 00 00 00          mov    $0xc,%esi           Load 12 as 1st argument
3   400505:  bf 08 12 50 00          mov    $0x501208,%edi      Load &gval1 as 2nd argument
4   40050a:  e8 b1 ff ff ff          callq  4004c0 <simple_l>   Call simple_l
5   40050f:  48 03 05 ea 0c 10 00    add    1051882(%rip),%rax    Add gval2 to result
6   400516:  c3                      retq
```

The instruction on line 3 stores the address of global variable `gval1` in register `%rdi`. It does this by simply copying the constant value `0x501208` into register `%edi`. The upper 32 bits of `%rdi` are then automatically set to zero. The instruction on line 5 retrieves the value of `gval2` and adds it to the value returned

Instruction		Effect	Description
`movq`	S, D	$D \leftarrow S$	Move quad word
`movabsq`	I, R	$R \leftarrow I$	Move quad word
`movslq`	S, R	$R \leftarrow$ SignExtend(S)	Move sign-extended double word
`movsbq`	S, R	$R \leftarrow$ SignExtend(S)	Move sign-extended byte
`movzbq`	S, R	$R \leftarrow$ ZeroExtend(S)	Move zero-extended byte
`pushq`	S	R[%rsp] $\leftarrow$ R[%rsp] $- 8$; M[R[%rsp]] $\leftarrow S$	Push
`popq`	D	$D \leftarrow$ M[R[%rsp]]; R[%rsp] $\leftarrow$ R[%rsp] $+ 8$	Pop

Figure 3: **Data movement instructions.** These supplement the movement instructions of IA32. The `movabsq` instruction only allows immediate data (shown as I) as the source value. Others allow, immediate data, a register, or memory (shown as S). Some instructions require the destination to be a register (shown as R), while others allow both register and memory destinations (shown as D).

by the call to `simple_l`. Here we see PC-relative addressing—the immediate value `1051882` (hexadecimal `0x100cea`) is added to the address of the following instruction to get `0x100cea` + `0x400516` = `0x501200`.

Figure 3 documents some of the data movement instructions available with x86-64, beyond those found in IA32 (see CS:APP Figure 3.4). Some instructions require to the destination to be a register, indicated by R. The instructions shown include different variations of the `mov` instruction to move data into a 64-bit register or memory destination. Moving immediate data to a 64-bit register can be done either with the `movq` instruction, which will sign extend a 32-bit immediate value, or with the `movabsq` instruction, when a full 64-bit immediate is required.

Moving from a smaller data size to 64 bits can involve either sign extension (`movsbq`, `movslq`) or zero extension (`movzbq`). Perhaps unexpectedly, instructions that move or generate 32-bit register values also set the upper 32 bits of the register to zero. Consequently there is no need for an instruction `movzlq`. Similarly, the instruction `movzbq` has the exact same behavior as `movzbl` when the destination is a register—both set the upper 56 bits of the destination register to zero. This is in contrast to instructions that generate 8 or 16-bit values, such as `movb`; these instructions do not alter the other bits in the register. The new stack instructions `pushq` and `popq` allow pushing and popping of 64-bit values.

Practice Problem 1:

The following C function converts an argument of type `src_t` to a return value of type `dst_t`, where these two types are defined using `typedef`:

```
dest_t cvt(src_t x)
{
    return (dest_t) x;
}
```

Assume argument `x` is in the appropriately named portion of register `%rdi` (i.e., `%rdi`, `%edi`, `%di`, or `%dil`), and that some form of move instruction is to be used to perform the type conversion and to copy

Instruction		Effect	Description
`leaq`	S, D	$D \leftarrow$ &S	Load effective address
`incq`	D	$D \leftarrow D$ `+ 1`	Increment
`decq`	D	$D \leftarrow D$ `- 1`	Decrement
`negq`	D	$D \leftarrow$ `-`D	Negate
`notq`	D	$D \leftarrow$ `~`D	Complement
`addq`	S, D	$D \leftarrow D$ `+` S	Add
`subq`	S, D	$D \leftarrow D$ `-` S	Subtract
`imulq`	S, D	$D \leftarrow D$ `*` S	Multiply
`xorq`	S, D	$D \leftarrow D$ `^` S	Exclusive-or
`orq`	S, D	$D \leftarrow D$ `|` S	Or
`andq`	S, D	$D \leftarrow D$ `&` S	And
`salq`	k, D	$D \leftarrow D$ `<<` k	Left shift
`shlq`	k, D	$D \leftarrow D$ `<<` k	Left shift (same as `salq`)
`sarq`	k, D	$D \leftarrow D$ `>>` k	Arithmetic right shift
`shrq`	k, D	$D \leftarrow D$ `>>` k	Logical right shift

Figure 4: **Integer 64-bit arithmetic operations.** They resemble their 32-bit counterparts (CS:APP Figure 3.7).

the value to the appropriately named portion of register `%rax`. Fill in the following table indicating the instruction, the source register, and the destination register for the following combinations of source and destination type:

T_X	T_Y	Instruction	S	D
`long`	`long`	`movq`	`%rdi`	`%rax`
`int`	`long`			
`char`	`long`			
`unsigned int`	`unsigned long`			
`unsigned char`	`unsigned long`			
`long`	`int`			
`unsigned long`	`unsigned`			

As shown in Figure 4, the arithmetic and logical instructions for 64-bit data resemble their 32-bit counterparts (see CS:APP Figure 3.7). For example, we find the instruction `addq` in addition to `addl`, and `leaq` in addition to `leal`. As mentioned earlier, instructions that generate 32-bit register results, such as `addl`, also set the high-order bits of the register to 0.

When mixing 32 and 64-bit data, GCC must make the right choice of arithmetic instructions, sign extensions, and zero extensions. These depend on subtle aspects of type conversion and the interactions between the 32 and 64-bit instructions. This is illustrated by the following C function:

```
1 long int gfun(int x, int y)
2 {
```

```
3     long int t1 = (long) x + y;   /* 64-bit addition */
4     long int t2 = (long) (x + y); /* 32-bit addition */
5     return t1 | t2;
6 }
```

Assuming integers are 32 bits and long integers are 64, the two additions in this function proceed as follows. Recall that type conversion has higher precedence than addition, and so line 3 calls for `x` to be converted to 64 bits, and by operand promotion `y` is also converted. Value `t1` is then computed using 64-bit addition. On the other hand, `t2` is computed in line 4 by performing 32-bit addition and then extending this value to 64 bits.

The assembly code generated for this function is as follows

```
   x86-64 version of function gfun
   Arguments in registers %rdi (x) and %rsi (y)
1 gfun:
2   movslq  %edi,%rax          Convert x to long
3   movslq  %esi,%rdx          Convert y to long
4   addl    %esi, %edi         lower bits of t2 (32-bit addition)
5   addq    %rdx, %rax         t1 (64-bit addition)
6   movslq  %edi,%rdi          Sign extend to get t2
7   orq     %rdi, %rax         Return t1 | t2
8   ret
```

Local value `t1` is computed by first sign extending the arguments. The `movslq` instructions on lines 2–3 take the lower 32 bits of the arguments in registers `%rdi` and `%rsi` and sign extend them to 64 bits in registers `%rax` and `%rdx`. The `addq` instruction on line 5 then performs 64-bit addition to get `t1`. Value `t2` is computed by performing 32-bit addition on the lower 32 bits of the two operands (line 4). This value is sign extended to 64 bits on line 6 (within a single register) to get `t2`.

Practice Problem 2:

A C function `arithprob` with arguments `a`, `b`, `c`, and `d` has the following body:

```
return a*b + c*d;
```

Itcompiles to the following x86-64 code:

```
   Arguments: a in %edi, b in %sil, c in %rdx, d in %ecx
1 arithprob:
2   movslq  %ecx,%rcx
3   movsbl  %sil,%esi
4   imulq   %rdx, %rcx
5   imull   %edi, %esi
6   leal    (%rsi,%rcx), %eax
7   ret
```

The arguments and return value are all signed integers of various lengths. Based on this assembly code, write a function prototype describing the return and argument types for `arithprob`.

Instruction	Effect	Description
`imulq` S	R[%rdx]:R[%rax] $\leftarrow S \times$ R[%rax]	Signed full multiply
`mulq` S	R[%rdx]:R[%rax] $\leftarrow S \times$ R[%rax]	Unsigned full multiply
`cltq`	R[%rax] $\leftarrow$ SignExtend(R[%eax])	Convert %eax to quad word
`cqto`	R[%rdx]:R[%rax] $\leftarrow$ SignExtend(R[%rax])	Convert to oct word
`idivq` S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] $\bmod S$; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Signed divide
`divq` S	R[%rdx] $\leftarrow$ R[%rdx]:R[%rax] $\bmod S$; R[%rax] $\leftarrow$ R[%rdx]:R[%rax] $\div S$	Unsigned divide

Figure 5: **Special arithmetic operations.** These operations support full 64-bit multiplication and division, for both signed and unsigned numbers. The pair of registers %rdx and %rax are viewed as forming a single 128-bit oct word.

4.1 Special Arithmetic Instructions

Figure 5 show instructions used to generate the full 128-bit product of two 64-bit words, as well as ones to support 64-bit division. They are similar to their 32-bit counterparts (CS:APP Figure 3.9). Several of these instructions view the combination of registers %rdx and %rax as forming a 128-bit *oct word*. For example, the `imulq` and `mulq` instructions store the result of multiplying two 64-bit values—the first as given by the source operand, and the second from register %rax.

The two divide instructions `idivq` and `divq` start with %rdx:%rax as the 128-bit dividend and the source operand as the 64-bit divisor. They then store the quotient in register %rax and the remainder in register %rdx. Preparing the dividend depends on whether unsigned (`divq`) or signed (`idivq`) division is to be performed. In the former case, register %rdx is simply set to 0. in the latter case, the instruction `cqto` is used to perform sign extension, copying the sign bit of %rax into every bit of %rdx.[2]

Figure 5 also shows an instruction `cltq` to sign extend register %eax to %rax [3]. This instruction is just a shorthand for the instruction `movslq %eaxreg,%raxreg`.

Aside: What if 64-bit arithmetic isn't good enough?

GCC running on x86-64 supports arithmetic using 128-bit signed and unsigned arithmetic, via data types `__int128_t` and `__uint128_t`. For example, the following C code computes the 128-bit product of two 64-bit integers and stores the result in a 128-bit global variable. It also returns the lower 64 bits as the function result. It implements this function using the single-operand version of the `imull` instruction.

```
/* Global variable for 128-bit integer */
__int128_t prod;
/* Compute 128-bit product of two 64-bit integers */
long int oct_mul(long int x, long int y)
{
    prod = (__int128_t) x * y;
    return (long) prod;
```

[2] GAS instruction `cqto` is called `cqo` in Intel and AMD documentation.

[3] Instruction `cltq` is called `cdqe` in Intel and AMD documentation.

```
}
```

Integers of this size can represent numbers in the range $\pm 1.7 \times 10^{38}$, nearly the range of what can be represented as a aingle-precision fbating-point number. **End Aside.**

5 Control

The control instructions and methods of implementing control transfers in x86-64 are essentially the same as those in IA32 (CS:APP Section 3.6). Two new instructions `cmpq` and `testq` are added to compare and test quad words, augmenting those for byte, word, and double word sizes (CS:APP Section 3.6.1):

Instruction	Based on	Description
`cmpq` S_2, S_1	S_1 - S_2	Compare quad words
`testq` S_2, S_1	S_1 & S_2	Test quad word

5.1 Conditional Move Instructions

Starting with the PentiumPro in 1995, recent generations of IA32 processors have had *conditional move* instructions that either do nothing or copy a value to a register depending on the values of the condition codes. These provide an alternate way of implementing conditional operations that can be substantially faster than using branching code. For years, these instructions have been largely unused. GCC did not generate code that used them, because that would prevent backward compatibility. The advent of x86-64 allowed the compiler writers to forgo this compatibility requirement, and so we now find conditional moves in compiled x86-64 code. This section describes conditional moves and how they are used.

We saw in the IA32 code generated by GCC that conditional statements (using `if`) and conditional expressions (using `? :`) were always implemented with conditional jumps, where the condition code values determine whether or not a jump is taken.

As an example, consider the following general form of assignment and conditional expression

```
v = test-expr ? then-expr : else-expr;
```

The compiler generates code having a form shown by the following abstract code

```
    if (!test-expr)
      goto else;
    v = then-expr;
    goto done;
else:
    v = else-expr;
done:
```

This code contains two code sequences—one evaluating *then-expr* and one evaluating *else-expr*. A combination of conditional and unconditional jumps is used to ensure that just one of the sequences is evaluated.

As noted in the discussion of branch prediction and mispredication penalties (CS:APP Section 5.12), modern processors can execute code of the style shown above efficiently only in the case where they can predict the outcome of the test with high probability. The processor uses a deep pipeline that follows the predicted branch direction, executing the chosen code sequence, a process known as *speculative execution*. If it predicted the test outcome correctly, then it *commits* any speculative results and proceeds with no loss of efficiency. If the prediction is incorrect, it must discard the speculative results and restart execution with the other code sequence. This can incur a significant delay, perhaps 10–40 clock cycles. Processors employ sophisticated branch prediction hardware that attempt to detect any regular pattern for a given branch being taken or not taken.

As an extreme example of this inefficiency, consider the following C function for computing the maximum of two values

```
int max(int x, int y)
{
    return (x < y) ? y : x;
}
```

In a typical application, the outcome of the test `x < y` is highly unpredictable, and so even the most sophisticated branch prediction hardware will guess correctly only around 50% of the time. In addition, the computations performed in each of the two code sequences require only a single clock cycle. As a consequence, the branch misprediction penalty dominates the performance of this function. Running on an Intel Pentium 4 Xeon, we find the function requires around 10 cycles per call when the outcome is easily predictable, but around 31 cycles per call when the outcome is random. From this we can infer that the branch misprediction penalty is around 42 clock cycles.

> **Aside: How did you determine this penalty?**
> Assume the probability of misprediction is p, the time to execute the code without mispredication is T_{OK}, and the misprediction penalty is T_{MP}. Then the average time to execute the code is $T_{avg} = (1-p)T_{OK} + p(T_{OK} + T_{MP}) = T_{OK} + pT_{MP}$. So for $p = 0.5$, $T_{OK} = 10$, and $T_{avg} = 31$, we get $T_{MP} = 42$. **End Aside.**

An alternate method of implementing some forms of conditional operations is to use *conditional move* instructions. With this approach, both the *then-expr* and the *else-expr* are evaluated, with the final value chosen based on the evaluation *test-expr*. This can be described by the following abstract code:

```
vt = then-expr;
v = else-expr;
if (test-expr) v = vt;
```

The final statement in this sequence illustrates a conditional move—value `vt` is moved to `v` only if the tested condition holds.

Figure 6 illustrates some of the conditional move instructions added to the IA32 instruction set with the introduction of the PentiumPro microprocessor. Each of these instructions has two operands: a source register or memory location *S*, and a destination register *D*. As with the different `set` (CS:APP Section 3.6.2) and `jump` instructions (CS:APP Section 3.6.3), the outcome of these instructions depend on the values of the

Instruction		Synonym	Move condition	Description
`cmove`	*S*, *D*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *D*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *D*		`SF`	Negative
`cmovns`	*S*, *D*		`~SF`	Nonnegative
`cmovg`	*S*, *D*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *D*	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	*S*, *D*	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	*S*, *D*	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	*S*, *D*	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	*S*, *D*	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	*S*, *D*	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	*S*, *D*	`cmovna`	`CF | ZF`	below or equal (unsigned <=)

Figure 6: **The `cmov` instructions.** These instructions copy the source value *S* to its destination *D* when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

condition codes. The source value is read from either memory or the source register, but it is copied to the destination only if the specified condition holds.

The source and destination values can be 16, 32, or 64 bits long. Single byte conditional moves are not supported. Unlike the unconditional instructions, where the operand length is explicitly encoded in the instruction name (e.g., `movw`, `movl`, `movq`), conditional move operand lengths can be inferred from the names of the destination registers, and so the same instruction name can be used for all operand lengths.

As an example, GCC generates the following code for function `max`

```
   x86-64 code generated for function max
   x in register %edi, y in %esi
1  max:
2    cmpl    %esi, %edi        Compare x:y
3    cmovge  %edi, %esi        if >=, then y=x
4    movl    %esi, %eax        set y as return value
5    ret
```

This code uses a conditional move to overwrite the register containing `y` with `x` when `x` $\geq$ `y` (line 3).

The conditional move can be implemented with no need to predict the outcome of the test. The processor simply evaluates the code and then either updates the destination register or keeps it the same. Running the code on the same Intel Pentium 4 Xeon, we find it requires around 10 cycles per call regardless of the test outcome.

Practice Problem 3:

In the following C function, `OP` is some C operator declared earlier with `#define`.

```
int arith(int x)
```

```
{
    return x OP 4;
}
```

When compiled, GCC generates the following x86-64 code:

```
    x86-64 implementation of arith
    x in register %edi
1 arith:
2   leal    3(%rdi), %eax
3   cmpl    $-1, %edi
4   cmovle  %eax, %edi
5   sarl    $2, %edi
6   movl    %edi, %eax    return value in %eax
7   ret
```

A. What operation is `OP`?

B. Annotate the code to explain how it works.

Not all conditional expressions can be compiled using conditional moves. Most significantly, the abstract code shown earlier evaluates both *then-expr* and *else-expr* regardless of the test outcome. If one of those two expressions could possibly generate an error condition or a side effect, this could lead to invalid behavior. As an illustration, consider the following C function

```
int cread(int *xp)
{
    return (xp ? *xp : 0);
}
```

At first, this seems like a good candidate to compile using a conditional move to read the value designated by pointer `xp`, as shown in the following assembly code:

```
    Invalid implementation of function cread
    xp in register %rdi
1 cread:
2   xorl    %eax, %eax        Set 0 as return value
3   testq   %rdi, %rdi        Test xp
4   cmovne  (%rdi), %eax      if !0, dereference xp to get return value
5   ret
```

This implementation is invalid, however, since the dereferencing of `xp` by the `cmovne` instruction (line 4) occurs even when the test fails, causing a null pointer dereferencing error. Instead, this code must be compiled using branching code.

A similar case holds when either of the two branches causes a side effect, as illustrated by the following function

```
int lcount = 0;
int se_max(int x, int y)
{
    return (x < y) ? (lcount++, y) : x;
}
```

This function increments global variable `lcount` as part of *then-expr*. Thus, branching code must be used to ensure this side effect only occurs when the test condition holds.

Using conditional moves also does not always improve code efficiency. For example, if either the *then-expr* or the *else-expr* evaluation requires a significant computation, then this effort is wasted when the corresponding condition does not hold. Our experiments with GCC indicate that it only uses conditional moves when the two expressions can each be computed with a single instruction. This is being too conservative for current processors, given their high branch misprediction penalties.

5.2 Looping Constructs

In CS:APP Section 3.6.5, we found that the three looping constructs of C: `do-while`, `while`, and `for`, were all compiled for IA32 using a common template based on the structure of `do-while`. The other loop forms were transformed into `do-while` as illustrated in CS:APP Figure 3.14.

With x86-64, we find a richer variety of loop templates. The template for `do-while` remains the same, and some `while` and `for` statements are implemented in this form. For example, the code generated for `fib_w` and `fib_f` shown in CS:APP Section 3.6.5 follows the same pattern as that shown for IA32. In some cases, however, `while` and `for` loops are translated using a different template. Let us examine these two different loop templates.

As an illustration, consider the following C function for computing factorial using a `do-while` loop

```
int fact_dw(int x)
{
    int result = 1;
    do {
        result *= x;
        x--;
    } while (x > 0);
    return result;
}
```

GCC generates the following code for x86-64

```
    x86-64 implementation of fact_dw
    x in register %edi
1 fact_dw:
2   movl    $1, %eax    result = 1
3 .L2:                    loop:
4   imull   %edi, %eax      result *= x
5   decl    %edi            x--
```

```
6   testl   %edi, %edi     Test x
7   jg      .L2           if >0 goto loop
8   rep ; ret               else return
```

This code follows the loop template seen earlier. Within the loop, the `imull` and `decl` instructions implement the body, while the `testl` and `jg` instructions implement the test. The loop is always entered from the top. The general form of this implementation is

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

Aside: Why is there a `rep` instruction in this code?
On line 8 of the x86-64 code for `fact_dw`, we see that the procedure ends with the instruction combination `rep; ret`, rather than simply `ret`. Looking at the Intel and AMD documentation for the `rep` instruction, we see that it is normally used to implement a repeating string operation [3, 6]. It seems completely inappropriate here. The answer to this puzzle can be seen in AMD's guidelines to compiler writers [1]. They recommend this particular combination to avoid making the `ret` instruction be the target of a conditional jump instruction. This is the case here, because it is preceded by a a `jg` instruction, and in the event the jump condition does not hold, the program "falls through" to the return. According to AMD, the processor does a better job predicting the outcome of the branch if it does not have a `ret` instruction as a target. This `rep` instruction will have no effect, since it is not followed by a string manipulation instruction. Its only purpose is to serve as a branch target. **End Aside.**

Now consider a factorial function based on a `while` loop:

```
int fact_while(int x)
{
    int result = 1;
    while (x > 0) {
        result *= x;
        x--;
    }
    return result;
}
```

(These two functions yield different results when $x \leq 0$.) GCC generates the following code for x86-64

```
    x86-64 implementation of fact_while
    x in register %edi
1 fact_while:
2   movl    $1, %eax      result = 1
3   jmp     .L12          goto middle
4 .L13:                 loop:
5   imull   %edi, %eax    result *= x
6   decl    %edi          x--
```

```
 7 .L12:                    middle:
 8   testl   %edi, %edi      Test x
 9   jg      .L13            if >0 goto loop
10   rep ; ret               else return
```

In this loop, we see the exact same four instructions, but we also see a second entry point into the loop, labeled `middle` in the comments. When entering at this point, only the instructions implementing the loop test are executed. The program enters via this point at the start of the loop, providing the initial loop test. This code has the following general form:

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

Comparing this form to the form based on translating `while` into `do-while`, we see that this new code requires an additional unconditional jump at the beginning, but it then avoids having duplicate copies of the test code. On a modern machine, unconditional jumps have negligible performance overhead, and so this new form would seem better in most cases.

Practice Problem 4:

For the following C function, the expressions `EXPR1`–`EXPR5` were specified using `#define`:

```
long int puzzle(int a, int b)
{
    int i;
    long int result = EXPR1;
    for (i = EXPR2; i > EXPR3; i -= EXPR4)
        result *= EXPR5;
    return result;
}
```

GCC generates the following x86-64 code:

```
     x86-64 implementation of puzzle
     a in register %edi, b in register %esi
     return value in register %rax
 1 puzzle:
 2   movslq  %esi,%rdx
 3   jmp     .L60
 4 .L61:
 5   movslq  %edi,%rax
 6   subl    %esi, %edi
 7   imulq   %rax, %rdx
```

```
 8 .L60:
 9   testl   %edi, %edi
10   jg      .L61
11   movq    %rdx, %rax
12   ret
```

A. What register is being used for local variable `result`?

B. What register is being used for local variable `i`?

C. Give valid definitions for expressions `EXPR1`–`EXPR5`

6 Procedures

We have already seen in our code samples that the x86-64 implementation of procedure calls differs substantially from that of IA32. By doubling the register set, programs need not be so dependent on the stack for storing and retrieving procedure information. This can greatly reduce the overhead for procedure calls and returns.

Here are some of the highlights of how procedures are implemented with x86-64:

- Arguments (up to the first six) are passed to procedures via registers, rather than on the stack. This eliminates the overhead of storing and retrieving values on the stack.
- The `call` instruction stores a 64-bit return pointer on the stack.
- Many functions do not require a stack frame. Only functions that cannot keep all local variables in registers need to allocate space on the stack.
- Functions can access storage on the stack up to 128 bytes beyond (i.e., at a lower address than) the current value of the stack pointer. This allows some functions to store information on the stack without incrementing or decrementing the stack pointer.
- There is no frame pointer. Instead, references to stack locations are made relative to the stack pointer. Typical functions allocate their total stack storage needs at the beginning of the call and keep the stack pointer at a fixed position.
- As with IA32, some registers are designated as callee-save registers. These must be saved and restored by any procedure that modifies them.

6.1 Argument Passing

Up to six integral (i.e., integer and pointer) arguments can be passed via registers. The registers are used in a specified order, and the name used for a register depends on the size of the data type being passed. These are shown in Figure 7. Arguments are allocated to these registers according to their ordering in the argument

Argument Number	Operand Size (bits)			
	64	32	16	8
1	%rdi	%edi	%di	%dl
2	%rsi	%esi	%si	%sil
3	%rdx	%edx	%dx	%dl
4	%rcx	%ecx	%cx	%cl
5	%r8	%r8d	%r8w	%r8b
6	%r9	%r9d	%r9w	%r9b

Figure 7: **Registers for passing function arguments** The registers are used in a specified order and named according to the argument sizes.

list. Arguments smaller than 64 bits can be accessed using the appropriate subsection of the 64-bit register. For example, if the first argument is 32 bits, it can be accessed as `%edi`.

As an example of argument passing, consider the following C function having eight arguments

```
void proc(long  a1, long  *a1p,
          int   a2, int   *a2p,
          short a3, short *a3p,
          char  a4, char  *a4p)
{
    *a1p += a1;
    *a2p += a2;
    *a3p += a3;
    *a4p += a4;
}
```

The arguments include a range of different sized integers (64, 32, 16, and 8 bits) as well as different types of pointers, each of which is 64 bits.

This function is implemented in x86-64 as follows:

```
    x86-64 implementation of function proc
    Arguments passed as follows:
      a1  in %rdi     (64 bits)
      a1p in %rsi     (64 bits)
      a2  in %edx     (32 bits)
      a2p in %rcx     (64 bits)
      a3  in %r8w     (16 bits)
      a3p in %r9      (64 bits)
      a4  at  8(%rsp) ( 8 bits)
      a4p at 16(%rsp) (64 bits)
1 proc:
2   movq    16(%rsp), %r10    Fetch a4p  (64 bits)
3   addq    %rdi, (%rsi)      *a1p += a1 (64 bits)
4   addl    %edx, (%rcx)      *a2p += a2 (32 bits)
5   movzbl  8(%rsp), %eax     Fetch a4   (8 bits)
```

```
6   addw    %r8w, (%r9)        *a3p += a3 (16 bits)
7   addb    %al, (%r10)        *a4p += a4 (8 bits)
8   ret
```

The first six arguments are passed in registers, while the last two are at positions 8 and 16 relative to the stack pointer. Different versions of the `add` instruction are used according to the sizes of the operands: `addq` for `a1` (`long`), `addl` for `a2` (`int`), `addw` for `a3` (`short`), and `addb` for `a4` (`char`).

Practice Problem 5:

A C function `incrprob` with arguments `q`, `t`, and `x` (not given in that order) has the following body:

```
*t += x;
*q += *t;
```

It compiles to the following x86-64 code:

```
1 incrprob:
2   addl    (%rdx), %edi
3   movl    %edi, (%rdx)
4   movslq  %edi,%rdi
5   addq    %rdi, (%rsi)
6   ret
```

Determine all valid function prototypes for `incrprob` by determining the ordering and possible types of the three parameters.

6.2 Stack Frames

We have already seen that many compiled functions do not require a stack frame. If all of the local variables can be held in registers, and the function does not call any other functions (sometimes referred to as a *leaf procedure*, in reference to the tree structure of procedure calls), then the only need for the stack is to save the return pointer.

On the other hand, there are several reasons a function may require a stack frame:

- There are too many local variables to hold in registers.
- Some local variables are arrays or structures.
- The function uses the address-of operator (&) to compute the address of a local variable.
- The function must pass some arguments on the stack to another function.
- The function needs to save the state of a callee-save register before modifying it.

When any of these conditions hold, we find the compiled code for the function creating a stack frame. Unlike the code for IA32, where the stack pointer fluctuates back and forth as values are pushed and popped, the stack frames for x86-64 procedures usually have a fixed size, set at the beginning of the procedure by decrementing the stack pointer (register `%rsp`). The stack pointer remains at a fixed position during the call, making it possible to access data using offsets relative to the stack pointer. As a consequence, the frame pointer (register `%ebp`) seen in IA32 code is no longer needed.

In addition, whenever one function (the *caller*) calls another (the *callee*), the return pointer gets pushed on the stack. By convention, we consider this part of the caller's stack frame, in that it encodes part of the caller's state. But this information gets popped from the stack as control returns to the caller, and so it does not affect the offsets used by the caller for accessing values within the stack frame.

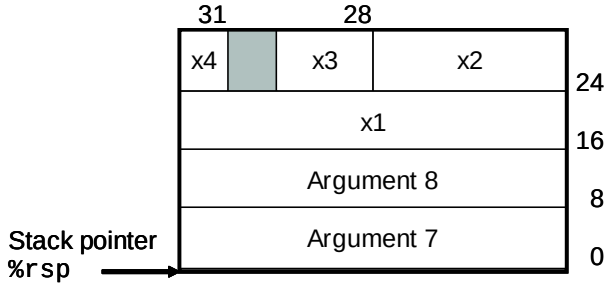
The following function illustrates many aspects of the x86-64 stack discipline.

```
long int call_proc()
{
    long x1 = 1;  int  x2 = 2;
    short x3 = 3; char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

GCC generates the following x86-64 code.

```
    x86-64 implementation of call_proc
 1  call_proc:
 2    subq    $32, %rsp          Allocate 32-byte stack frame
 3    movl    $2, %edx           Pass   2 as argument 3
 4    movl    $3, %r8d           Pass   3 as argument 5
 5    leaq    31(%rsp), %rax     Compute &x4
 6    leaq    24(%rsp), %rcx     Pass &x2 as argument 4
 7    leaq    28(%rsp), %r9      Pass &x3 as argument 6
 8    leaq    16(%rsp), %rsi     Pass &x1 as argument 2
 9    movl    $1, %edi           Pass   1 as argument 1
10    movq    $1, 16(%rsp)       x1 = 1
11    movq    %rax, 8(%rsp)      Pass &x4 as argument 8
12    movl    $2, 24(%rsp)       x2 = 2
13    movw    $3, 28(%rsp)       x3 = 3
14    movb    $4, 31(%rsp)       x4 = 4
15    movl    $4, (%rsp)         Pass   4 as argument 7
16    call    proc               Call
17    movswl  28(%rsp),%edx      Get x3
18    movsbl  31(%rsp),%ecx      Get x4
19    movslq  24(%rsp),%rax      Get x2
20    addq    16(%rsp), %rax     Compute x1+x2
21    addq    $32, %rsp          Deallocate stack frame
22    subl    %ecx, %edx         Compute (int) (x3-x4)
23    movslq  %edx,%rdx          Sign extend to long
24    imulq   %rdx, %rax         Return (x1+x2)*(x3-x4)
25    ret
```

A). Before call to `proc`



B). During call to `proc`

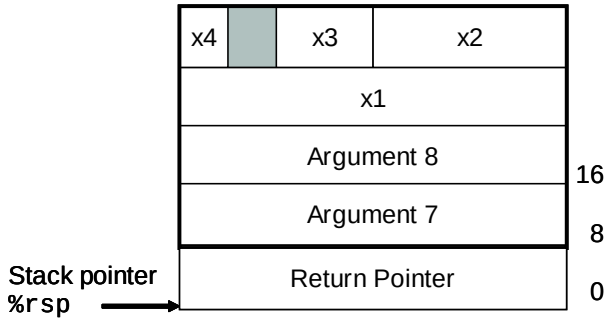


Figure 8: **Stack frame structure for `call_proc`.** The frame is required to hold local variables `x1`–`x4`, as well as for the seventh and eighth arguments to `proc`. During the execution of `proc` (B), the stack pointer is shifted down by 8.

Figure 8A illustrates the stack frame set up during the execution of `call_proc`. Function `call_proc` allocates 32 bytes on the stack by decrementing the stack pointer. It uses bytes 16–31 to hold local variables `x1` (bytes 16–23), `x2` (bytes 24–27), `x3` (bytes 28–29), and `x4` (byte 31). These allocations are sized according to the variable types. Byte 30 is unused. Bytes 0–7 and 8–15 of the stack frame are used to hold the seventh and eighth arguments to `call_proc`, since there are not enough argument registers. The parameters are allocated eight bytes each, even though parameter `x4` requires only a single byte. In the code for `call_proc`, we can see instructions initializing the local variables and setting up the parameters (both in registers and on the stack) for the call to `call_proc`. After `proc` returns, the local variables are combined to compute the final expression, which is returned in register `%rax`. The stack space is deallocated by simply incrementing the stack pointer before the `ret` instruction.

Figure 8B illustrates the stack during the execution of `proc`. The `call` instruction pushed the return pointer onto the stack, and hence the stack pointer is shifted down by 8 relative to its position during the execution of `call_proc`. Hence, within the code for `proc`, arguments 7 and 8 are accessed by offsets of 8 and 16 from the stack pointer.

Observe how `call_proc` changed the stack pointer only once during its execution. GCC determined that 32 bytes would suffice for holding all local variables and for holding the additional arguments to `proc`. Minimizing the amount of movement by the stack pointer simplifies the compiler's task of generating reference to stack elements using offsets from the stack pointer.

6.3 Register Saving Conventions

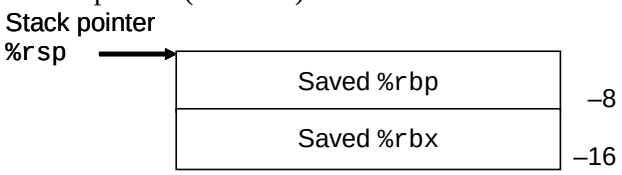
We saw in IA32 (CS:APP Section 3.73) that some registers used for holding temporary values are by designated as *caller saved*, where the callee is free to overwrite their values, while others are *callee saved*, where the callee must save their values on the stack before writing to them. With x86-64, the following registers are designated as being callee saved: `%rbx`, `%rbp`, and `%r12`–`%r15`.

> **Aside: Are there any caller-saved temporary registers?**
> Of the 16 general-purpose registers, we've seen that six are designated for passing arguments, six are for callee-saved temporaries, one (`%rax`) holds the return value for a function, and one (`%rsp`) serves as the stack pointer. In addition, `%r10` has a specifi c use in supporting languages that allow *static scoping*. This includes Pascal, but not C or C++. Only `%r12` is left as a caller-saved temporary register, but this is generally reserved for use by the linking code.
>
> Of course, some of the argument registers can be used when a procedure has less than six arguments, and `%rax` can be used within a procedure. Moreover, it's always possible to save the state of some callee-saved registers and then use them to hold temporary values. **End Aside.**

We illustrate the use of callee-saved registers with a somewhat unusual version of a recursive factorial function:

```
/* Compute x! and store at resultp */
void sfact_helper(long int x, long int *resultp)
{
    if (x <= 0)
        *resultp = 1;
    else {
```

A). Before decrementing the stack pointer (on line 4)



B). After decrementing the stack pointer

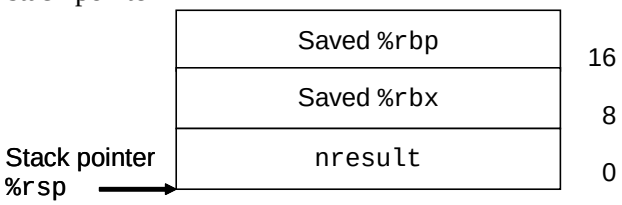


Figure 9: **Stack frame for function `sfact_helper`.** This function decrements the stack pointer *after* saving some of the state.

```
        long int nresult;
        sfact_helper(x-1, &nresult);
        *resultp = x * nresult;
    }
}
```

To compute the factorial of a value `x`, this function would be called at the top level as follows:

```
long int sfact(long int x)
{
    long int result;
    sfact_helper(x, &result);
    return result;
}
```

The x86-64 code for `sfact_helper` is shown below

```
    x86-64 implementation of sfact_helper
    Argument x in %rdi, resultp in %rsi
 1  sfact_helper:
 2    movq    %rbx, -16(%rsp)      Save %rbx (callee save)
 3    movq    %rbp, -8(%rsp)       Save %rbp (callee save)
 4    subq    $24, %rsp            Allocate 24 bytes on stack
 5    testq   %rdi, %rdi           Test x
 6    movq    %rdi, %rbx           Copy x to %rbx
 7    movq    %rsi, %rbp           Copy resultp to %rbp
 8    jle     .L17                 if x<=0 goto finish
 9    leaq    -1(%rdi), %rdi       xm1 = x-1  (1st argument)
10    movq    %rsp, %rsi           &nresult   (2nd argument)
11    call    sfact_helper         sfact_helper(xm1, &nresult)
12    imulq   (%rsp), %rbx         x*nresult
```

```
13   movq    %rbx, (%rbp)          *result = x*nresult
14   movq    8(%rsp), %rbx         Restore %rbx
15   movq    16(%rsp), %rbp        Restore %rbp
16   addq    $24, %rsp             Deallocate stack frame
17   ret                           Return
18 .L17:                          finish:
19   movq    $1, (%rsi)            *resultp = 1
20   movq    8(%rsp), %rbx         Restore %rbx
21   movq    16(%rsp), %rbp        Restore %rbp
22   addq    $24, %rsp             Deallocate stack frame
23   ret                           Return
```

Figure 9 illustrates how `sfact_helper` uses the stack to store the values of callee-saved registers and to hold the local variable `nresult`. This implementation has the interesting feature that the two callee-saved registers it uses (`%rbx` and `%rbp`) are saved on the stack (lines 2–3) *before* the stack pointer is decremented (line 4) to allocate the stack frame. As a consequence, the stack offset for `%rbx` shifts from -16 at the beginning to $+8$ at the end (line 14). Similarly, the offset for `%rbp` shifts from -8 to $+16$.

Being able to access memory beyond the stack pointer is an unusual feature of x86-64. It requires that the virtual memory management system allocate memory for that region. The x86-64 ABI [8] specifies that programs can use the 128 bytes beyond (i.e., at lower addresses than) the current stack pointer. The ABI refers to this area as the *red zone*. It must be kept available for reading and writing as the stack pointer moves.

Note also how the code for `sfact_helper` has two separate `ret` instructions: one for each branch of the conditional statement. This contrasts with IA32 code, where we always saw the different branches come together at the end of a function to a shared `ret` instruction. This is partly because IA32 requires a much more elaborate instruction sequence to exit a function, and it is worthwhile to avoid duplicating these instructions at multiple return points.

Practice Problem 6:

For the following C program

```
long int local_array(int i)
{
    long int a[4] = {2L, 3L, 5L, 7l};
    int idx = i & 3;
    return a[idx];
}
```

GCC generates the following code

```
    x86-64 implementation of local_array
    Argument i in %edi
1 local_array:
2   andl    $3, %edi
3   movq    $2, -40(%rsp)
4   movq    $3, -32(%rsp)
```

```
5   movq    $5, -24(%rsp)
6   movq    $7, -16(%rsp)
7   movq    -40(%rsp,%rdi,8), %rax
8   ret
```

A. Draw a diagram indicating the stack locations used by this function and their offsets relative to the stack pointer.

B. Annotate the assembly code to describe the effect of each instruction

C. What interesting feature does this example illustrate about the x86-64 stack discipline?

Practice Problem 7:

For the following recursive factorial program

```
long int rfact(long int x)
{
    if (x <= 0)
        return 1;
    else {
        long int xm1 = x-1;
        return x * rfact(xm1);
    }
}
```

GCC generates the following code

```
    x86-64 implementation of recursive factorial function rfact
    Argument x in %rdi
 1  rfact:
 2    testq   %rdi, %rdi
 3    pushq   %rbx
 4    movl    $1, %eax
 5    movq    %rdi, %rbx
 6    jle     .L9
 7    leaq    -1(%rdi), %rdi
 8    call    rfact
 9    imulq   %rbx, %rax
10  .L9:
11    popq    %rbx
12    ret
```

A. What value does the function store in `%rbx`?

B. What are the purposes of the `pushq` (line 3) and `popq` (line 11) instructions?

C. Annotate the assembly code to describe the effect of each instruction

D. What interesting feature does this example illustrate about the x86-64 stack discipline?

7 Data Structures

Data structures follow the same principles in x86-64 as they do in IA32: arrays are allocated as sequences of identically-sized blocks holding the array elements, structures are allocated as sequences of variably-sized blocks holding the structure elements, and unions are allocated as a single block big enough to hold the largest union element.

One difference is that x86-64 follows a more stringent set of alignment requirements. For any scalar data type requring K bytes, its starting address must be a multiple of K. Thus, data types `long`, `double`, and pointers must be aligned on 8-byte boundaries. In addition, data type `long double` uses a 16-byte alignment (and size allocation), even though the actual representation requires only 10 bytes. These alignment conditions are imposed to improve memory system performance—the memory interface is designed in most processors to read or write aligned blocks that are eight or sixteen bytes long.

Practice Problem 8:

For each of the following structure declarations, determine the offset of each field, the total size of the structure, and its alignment requirement under x86-64.

A. `struct P1 { int i; char c; long j; char d; };`
B. `struct P2 { long j; char c; char d; int i; };`
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 };`

8 Floating Point

Starting with the Pentium MMX in 1997, both Intel and AMD have introduced successive generations of *media* instructions to support graphics and image processing. The most recent version of these is named *SSE3*, for "Streaming SIMD Extensions, version 3." It is available on all x86-64 processors. The media instructions originally focused on allowing multiple operations to be performed in a parallel mode known as *single instruction, multiple data* or *SIMD* (pronounced SIM-DEE). In this mode the same operation is performed on a number of different data values in parallel.

The media instructions implement SIMD operations by having a set of registers that can hold multiple data values in *packed* format. For example, SSE3 provides sixteen *XMM* registers of 128 bits each, named `%xmm0`–`%xmm15`. Each one of these registers can hold a vector of K elements of N bits each, such that $K \times N = 128$. For integers, K can be 8, 16, 32, or 64 bits, while for floating-point numbers, K can be 32 or 64. For example, a single SSE3 instruction can add two byte vectors of eight elements each. The floating point formats match the IEEE standard formats for single and double-precision values. The major use of these media instructions are in library routines for graphics and image processing. These routines must be written with special tools and using assembly code, since the major compilers do not support the media extensions.

Instruction	Source	Destination	Description
`movss`	M_{32}/X	X	Move single precision
`movss`	X	M_{32}	Move single precision
`movsd`	M_{64}/X	X	Move double precision
`movsd`	X	M_{64}	Move double precision
`cvtss2sd`	M_{32}/X	X	Convert single to double precision
`cvtsd2ss`	M_{64}/X	X	Convert double to single precision
`cvtsi2ss`	M_{32}/R_{32}	X	Convert integer to single precision
`cvtsi2sd`	M_{32}/R_{32}	X	Convert integer to double precision
`cvtsi2ssq`	M_{64}/R_{64}	X	Convert quadword integer to single precision
`cvtsi2sdq`	M_{64}/R_{64}	X	Convert quadword integer to double precision
`cvttss2si`	X/M_{32}	R_{32}	Convert with truncation single precision to integer
`cvttsd2si`	X/M_{64}	R_{32}	Convert with truncation double precision to integer
`cvttss2siq`	X/M_{32}	R_{64}	Convert with truncation single precision to quadword integer
`cvttsd2siq`	X/M_{64}	R_{64}	Convert with truncation double precision to quadword integer

X: XMM register (e.g., `%xmm3`)
R_{32}: 32-bit general-purpose register (e.g., `%eax`)
R_{64}: 64-bit general-purpose register (e.g., `%rax`)
M_{32}: 32-bit memory range
M_{64}: 64-bit memory range

Figure 10: **Floating-point movement and conversion operations.** These operations transfer values between memory and registers, possibly converting between data types.

With the SSE3 came the opportunity to completely change the way floating point code is compiled for x86 processors. Historically, floating point is implemented with an arcane set of instructions dating back to the 8087, a floating-point coprocessor for the Intel 8086. These instructions, often referred to as "x87" operate on a shallow stack of registers, a difficult target for optimizing compilers (CS:APP Section 3.14). They also have many quirks due to a nonstandard 80-bit floating-point format (CS:APP Section 2.4.6).

SSE3 introduced a set of *scalar* floating-point operations that work on single values in the low-order 32 or 64 bits of XMM registers, The overall effect is to provide a set of registers and instructions that are more typical of the way other processors support floating point. GCC now implements floating point using mostly SSE3 instructions, reverting to x87 floating point for only a few special cases.

This section describes the x86-64 implementation of floating point based on SSE3. It supplants the presentation of floating point from CS:APP, Section 2.4.5.

8.1 Floating-Point Movement and Conversion Operations

Figure 10 shows a set of instructions for transfering data and for performing conversions between floating-point and integer data types. Floating-point data are held either in memory (indicated in the table as M_{32} and M_{64}) or in XMM registers (shown in the table as X). Integer data are held either in memory (indicated in the table as M_{32} or M_{64}) or in general-purpose registers (shown in the table as R_{32} and R_{64}). All 32-bit

memory data must satisfy a 4-byte alignment, while 64-bit data must satisfy an 8-byte alignment.

The floating-point move operations can transfer data from register to register, from memory to register and from register to memory. A single instruction cannot move data from memory to memory. The floating-point conversion operations have either memory or a register as source and a register as destination, where the registers are general-purpose registers for integer data and XMM registers for floating-point data. The instructions, such as `cvttss2si`, for converting floating-point values to integers use truncation, always rounding values toward zero, as is required by C and most other programming languages.

As an example of the different floating-point move and conversion operations, consider the following C function:

```
double fcvt(int i, float *fp, double *dp, long *lp)
{
    float f = *fp; double d = *dp; long l = *lp;
    *lp = (long) d;
    *fp = (float) i;
    *dp = (double) l;
    return (double) f;
}
```

and its associated x86-64 assembly code

```
     x86-64 implmentation of fcvt
     Arguments:
          i       %edi    int
          fp      %rsi    float *
          dp      %rdx    double *
          lp      %rcx    long *
 1  fcvt:
 2    movlpd  (%rdx), %xmm0             d = *dp
 3    movq    (%rcx), %r8               l = *lp
 4    movss   (%rsi), %xmm1             f = *fp
 5    cvttsd2siq      %xmm0, %rax       (long) d
 6    xorps   %xmm0, %xmm0              Set %xmm0 to 0
 7    cvtsi2ss        %edi, %xmm0       (float) i
 8    movq    %rax, (%rcx)              *lp = (long) d
 9    movss   %xmm0, (%rsi)             *fp = (float) i
10    cvtsi2sdq       %r8, %xmm0        (double) l
11    movsd   %xmm0, (%rdx)             *dp = (double) l
12    cvtss2sd        %xmm1, %xmm0      Return (double) f
13    ret
```

All of the arguments to `fcvt` are passed through the general-purpose registers, since they are either integers or pointers. The return value is returned in register `%xmm0`, the designated return register for `float` or `double` values. In this code, we see a number of the move and conversion instructions of Figure 10. In line 2, we see a variant of the `movsd` instruction called `movlpd`. This “move lower packed double-precision floating point” differs from `movsd` in its affect on the upper 64 bits of the destination XMM register. Since we are not making use of these bits, the two instructions can be used interchangeably.

Single	Double	Effect	Description
addss	addsd	$D \leftarrow D + S$	Floating-point add
subss	subsd	$D \leftarrow D - S$	Floating-point subtract
mulss	mulsd	$D \leftarrow D \times S$	Floating-point multiply
divss	divsd	$D \leftarrow D \,/\, S$	Floating-point divide
maxss	maxsd	$D \leftarrow \max(D,\ S)$	Floating-point maximum
minss	minsd	$D \leftarrow \min(D,\ S)$	Floating-point minimum
sqrtss	sqrtsd	$D \leftarrow \sqrt{S}$	Floating-point square root

Figure 11: **Floating-point arithmetic operations.** All have source S and destination D operands.

In line 6 we see the instruction `xorps`. In this context, the instruction is simply setting the destination register `%xmm0` to all zeros, much as we've seen the `xorl` used to set an integer register to zero (see CS:APP Practice Problem 3.6). It's not clear why GCC inserted this instruction, since the lower 32 bits of the register are set to the value of `(float) i` by the next instruction, and only those bits are required by the subsequent instructions (`movss` and `movsd`) having `%xmm0` as source.

Practice Problem 9:

The following C function converts an argument of type `src_t` to a return value of type `dst_t`, where these two types are defi ned using `typedef`.

```
dest_t cvt(src_t x)
{
    return (dest_t) x;
}
```

Assume argument `x` is either in `%xmm0` or in the appropriately named portion of register `%rdi` (i.e., `%rdi` or `%edi`), and that one of the conversion instructions is to be used to perform the type conversion and to copy the value to the appropriately named portion of register `%rax` (integer result) or `%xmm0` (fbating-point result). Fill in the following table indicating the instruction, the source register, and the destination register for the following combinations of source and destination type:

T_x	T_y	Instruction	S	D
long	double	cvtsi2sdq	%rdi	%xmm0
double	int			
float	double			
long	float			
float	long			

8.2 Floating-Point Arithmetic operations

Figure 11 documents a set of SSE3 floating-point instructions that perform arithmetic operations. Each has two operands: a source S, which can be either an XMM register or a memory location, and a destination D,

which must be an XMM register. Each operation has an instruction for single-precision and an instruction for double precision. The result is stored in the destination register.

As an example, consider the example function from CS:APP Section 3.14.5:

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

The x86-64 code is as follows:

```
    x86-64 implementation of funct
    Arguments:
         a       %xmm0   double
         x       %xmm1   float
         b       %xmm2   double
         i       %edi    int
1   funct:
2     cvtss2sd        %xmm1, %xmm1      %xmm1 = (double) x
3     mulsd   %xmm0, %xmm1              %xmm1 = a*x
4     cvtsi2sd        %edi, %xmm0       %xmm0 = (double) i
5     divsd   %xmm0, %xmm2              %xmm2 = b/i
6     movsd   %xmm1, %xmm0              %xmm0 = a*x
7     subsd   %xmm2, %xmm0              return a*x - b/i
8     ret
```

The three floating point arguments `a`, `x`, and `b` are passed in XMM registers `%xmm0`–`%xmm2`, while integer argument `i` is passed in register `%edi`. Conversion instructions are required to convert arguments `x` and `i` to double. The function value is returned in register `%xmm0`. Compared to the stack code shown for IA32 in CS:APP, floating-point x86-64 code more closely resembles integer code.

Unlike integer arithmetic operations, the floating-point operations cannot have immediate values as operands. Instead, the compiler must allocate and initialize storage for any constant values, and then generate code that reads the values from memory. This is illustrated by the following Celsius to Fahrenheit conversion function:

```
double cel2fahr(double temp)
{
    return 1.8 * temp + 32.0;
}
```

The relevant parts of the x86-64 assembly code are as follows:

```
    Constant declarations

1   .LC2:
2     .long   3435973837        Low order four bytes of 1.8
3     .long   1073532108        High order four bytes of 1.8
4   .LC4:
```

```
5    .long   0                   Low order four bytes of 32.0
6    .long   1077936128          High order four bytes of 32.0
     Code

     x86-64 implementation of cel2fahr
     Argument temp in %xmm0
7  cel2fahr:
8    mulsd   .LC2(%rip), %xmm0          Multiply by 1.8
9    addsd   .LC4(%rip), %xmm0          Add 32.0
10   ret
```

We see that the function reads the value 1.8 from the memory location labeled `.LC2`, and the value 32.0 from the memory location labeled `.LC4`. Looking at the values associated with these labels, we see that each is specified by a pair of `.long` declarations with the values given in decimal. How should these be interpreted as floating-point values? Looking at the declaration labeled `.LC2`, we see that the two values are 3435973837 (`0xcccccccd`) and 1073532108 (`0x3ffccccc`). Since the machine uses little-endian byte ordering, the first value gives the low-order 4 bytes, while the second gives the high-order 4 bytes. From the high-order bytes, we can extract an exponent field of `0x3ff` (1023), from which we subtract a bias of 1023 to get an exponent of 0. Concatenating the fraction bits of the two values, we get a fraction field of `0xcccccccccccccd`, which can be shown to be the fractional binary representation of 0.8, to which we add the implied leading one to get 1.8.

Practice Problem 10:

Show how the numbers declared at label `.LC4` encode the number 32.0.

8.3 Floating-Point Comparison operations

The SSE3 extenstions provide two instructions for comparing floating values:

Instruction	Based on	Description
`ucomiss` S_2, S_1	S_1 - S_2	Compare single precision
`ucomisd` S_2, S_1	S_1 - S_2	Compare double precision

These instructions are similar to the `cmpl` and `cmpq` instructions (see CS:APP Section 3.6.1), in that they compare operands S_1 and S_2 and set the condition codes to indicate their relative values. As with `cmpq`, they follow the GAS convention of listing the operands in reverse order. Argument S_2 must be in an XMM register, while S_1 can either be in an XMM register or in memory.

The floating-point comparison instructions set three condition codes: the zero flag ZF, the carry flag CF, and the parity flag PF. We did not document the parity flag in CS:APP, because it is not used in GCC-generated IA32 code. For integer operations, this flag is set when the most recent arithmetic operation yielded a value with odd parity (i.e., an odd number of 1's in the word). For floating-point comparisons, however, the flag is set when either operand is *NaN*. By convention, any comparison is considered to fail when one of the arguments is a *NaN*, and this flag is used to detect such a condition. For example, even the comparison `x == x` yields 0 when `x` is a *NaN*.

The condition codes are set as follows:

Ordering	CF	ZF	PF
Unordered	1	1	1
<	1	0	0
=	0	1	0
>	0	0	0

The *Unordered* case occurs when either of the operands is *NaN*. This can be detected from the parity flag. Commonly, the `jp` (for "jump on parity") instruction is used to conditionally jump when a floating-point comparison yields an unordered result. Except for this case, the values of the carry and zero flags are the same as those for an unsigned comparison: ZF is set when the two operands are equal, and CF is set when $S_1 < S_2$. Instructions such as `ja` and `jb` are used to conditionally jump on various combinations of these flags.

As an example of floating-point comparisons, the following C function classifies its argument `x` according to its relation to 0.0, returning an enumerated type as result.

```
typedef enum {NEG, ZERO, POS, OTHER} range_t;

range_t find_range(float x)
{
    int result;
    if (x < 0)
        result = NEG;
    else if (x == 0)
        result = ZERO;
    else if (x > 0)
        result = POS;
    else
        result = OTHER;
    return result;
}
```

Enumerated types in C are encoded as integers, and so the possible function values are: 0 (`NEG`), 1 (`ZERO`), 2 (`POS`), and 3 (`OTHER`). This final outcome occurs when the value of `x` is *NaN*.

GCC generates the following x86-64 code for `find_range`:

```
    x86-64 implementation of find_range
    Argument x in %xmm0 (single precision)
1 find_range:
2   xorps   %xmm1, %xmm1        %xmm1 = 0
3   xorl    %eax, %eax          result = 0
4   ucomiss %xmm0, %xmm1        Compare 0:x
5   ja      .L5                 If (> and not NaN) goto done
6   movb    $1, %al             result = 1
7   ucomiss %xmm1, %xmm0        Compare x:0
8   jp      .L13                If unordered goto next
```

```
 9   je      .L5                 If (= and not NaN) goto done
10 .L13:                       next:
11   xorl    %eax, %eax          result = 0
12   ucomiss %xmm1, %xmm0        compare x:0
13   setbe   %al                 if (<= or NaN) result = 1
14   addl    $2, %eax            result += 2
15 .L5:                         return
16   rep ; ret
```

The code is somewhat arcane—it compares `x` to 0.0 three times, even though the required information could be obtained with a single comparison. Let us trace the flow of the function for the four possible comparison results.

x < 0.0**:** The `ja` instruction on line 5 will be taken, jumping to the end with a return value of 0.

x $= 0.0$**:** The `ja` instruction on line 5 will not be taken, nor will be the `jp` instruction on line 8. The `je` instruction on line 9 will be taken, jumping to the end with a return value of 1 (set on line 6).

x > 0.0**:** Neither the `ja` instruction on line 5, the `jp` instruction on line 8, nor the `je` instruction on line 9 will be taken. In addition, the `setbe` instruction on line 13 will not change the return value, and so the return value will be incremented from 0 to 2 on line 14, and this value will be returned.

x $= NaN$**:** The `ja` instruction on line 5 will not be taken, but the `jp` instruction on line 8 will be. Then the `setbe` instruction on line line 13 will change the return value from 0 to 1, and this value is then incremented from 1 to 3 on line 14.

Once again, we see that x86-64 floating-point code more closely resembles integer code than it does the stack-based floating-point code of IA32.

8.4 Floating-Point Code in Procedures

We have already seen examples of several functions that have floating-point arguments and a floating-point result. The XMM registers can be used for these purposes. Specifically, the following conventions are observed:

- Up to eight floating point arguments can be passed in XMM registers `%xmm0`–`%xmm7`. These registers are used in the order the arguments are listed. Additional floating-point arguments can be passed on the stack.
- A function that returns a floating-point value does so in register `%xmm0`.
- All XMM registers are caller saved, The callee may overwrite any of these registers without first saving it.

8.5 Performing Common Floating-Point Operations in Uncommon Ways

At times, GCC makes surprising choices of instructions for performing common tasks. As examples, we've seen how the `leal` instruction is often used to perform integer arithmetic (CS:APP Section 3.5), and the `xorl` instruction is used to set registers to 0 (CS:APP Practice Problem 3.6).

There are far more instructions for performing floating-point operations than we've documented here or in CS:APP Section 3.14, and some of these appear in unexpected places. We document a few such cases here.

The following assembly code shows examples of instructions designed to operate on vectors of values in packed format:

```
movlpd  (%rax), %xmm0     Read double into %xmm0
xorps   %xmm2, %xmm2      Set %xmm2 to zero
movaps  %xmm0, %xmm1      Copy %xmm0 to %xmm1
```

As mentioned earlier, the `movlpd` is similar to the `movsd` instruction. In this example, it reads the eight bytes starting at the address specified by register `%rax` and stores it in the lower eight bytes of XMM register `%xmm0`.

The `xorps` instruction is an exclusive-or instruction for XMM data. Since both operands are the same in this example, the effect is to set `%xmm2` to 0.

The `movaps` instruction (for "move aligned packed single-precision") is used to copy entire XMM registers. In this example, it simply copies `%xmm0` to `%xmm1`.

In addition, there are times that GCC uses IA32 floating-point code in x86-64. Most commonly, it makes use of the ability of IA32 floating-point instructions to perform format conversions while loading a value from memory and while storing a value to memory. This is illustrated by the following code sequence

```
flds    16(%rsp)          Load single-precision into FP register
fstpl   8(%rsp)           Store FP register as double precision
```

Recall that IA32 floating point makes use of a shallow stack of registers, each holding an 80-bit floating-point value. In this example, the `flds` instruction reads a single-precision value from memory, converts it to the 80-bit format, and stores the result in the top stack register. The `fstpl` instruction pops the top value from the stack, converts it to double precision, and stores it in memory.

This combination of copying and conversion could readily be implemented using SSE instructions:

```
movss           16(%rsp), %xmm0   Read single-precision into %xmm0
cvtss2sd        %xmm0, %xmm0      Convert to double precision
movsd           %xmm0, 8(%rsp)    Store double precision
```

but, for some reason, GCC prefers the stack-based version.

References

[1] Advanced Micro Devices, Inc. *Software Optimization Guide for AMD Athlon 64 and AMD Opteron Processors*, 2004. Publication Number 25112.
Also available at `http://www.amd.com/us-en/Processors/TechnicalResources/`.

[2] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, 2005. Publication Number 24592.
Also available at `http://www.amd.com/us-en/Processors/TechnicalResources/`.

[3] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*, 2005. Publication Number 24594.
Also available at `http://www.amd.com/us-en/Processors/TechnicalResources/`.

[4] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*, 2005. Order Number 253665.
Also available at `http://developer.intel.com/`.

[5] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A–M*, 2005. Order Number 253666.
Also available at `http://developer.intel.com/`.

[6] Intel Corporation. *Intel Architecture Software Developer's Manual, Volume 2B: Instruction Set Reference, N–Z*, 2005. Order Number 253667.
Also available at `http://developer.intel.com/`.

[7] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Third Edition*. Morgan-Kaufmann, San Francisco, 2002.

[8] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V application binary interface AMD64 architecture processor supplement. Technical report, AMD64.org, 2005. Available at `http://www.x86-64.org/`.

Practice Problem Solutions

Problem 1 Solution: [Pg. 9]

This problem illustrates some of the subtleties of type conversion and the different move instructions. In some cases, we make property that the `movl` instruction will set the upper 32 bits of the destination register to 0's. Some of the problems have multiple solutions.

T_x	T_y	Instruction	S	D	Explanation
`long`	`long`	`movq`	`%rdi`	`%rax`	No conversion
`int`	`long`	`movslq`	`%edi`	`%rax`	Sign extend
`char`	`long`	`movsbq`	`%dil`	`%rax`	Sign extend
`unsigned int`	`unsigned long`	`movl`	`%edi`	`%eax`	Zero extends to 64 bits
`unsigned char`	`unsigned long`	`movzbq`	`%dil`	`%rax`	Zero extend to 64
`unsigned char`	`unsigned long`	`movzbl`	`%dil`	`%eax`	Zero extends to 64 bits
`long`	`int`	`movslq`	`%edi`	`%rax`	Sign extends to 64 bits
`long`	`int`	`movl`	`%edi`	`%eax`	Zero extends to 64 bits
`unsigned long`	`unsigned`	`movl`	`%edi`	`%eax`	Zero extends to 64 bits

We show that the `long` to `int` conversion can use either `movslq` or `movl`, even though one will sign extend the upper 32 bits, while the other will zero extend it. This is because the values of the upper 32 bits are ignored for any subsequent instruction `%eax` as an operand.

Problem 2 Solution: [Pg. 11]

We can step through the code for `arithprob` and determine the following:

1. The `movslq` instruction sign extends `d` to a long integer. This implies that `d` has type `int` and `c` has type `long`.
2. The `movsbl` instruction sign extends `b` to an integer. This means that `b` has type `char` and `a` has type `int`.
3. The sum is computed using a `leal` instruction, indicating that the return value has type `int`.

From this, we can determine that the unique prototype for `arithprob` is

```
int arithprob(int a, char b, long c, int d);
```

Problem 3 Solution: [Pg. 15]

This problem provides a chance to study the use of conditional moves.

A. The operator is '`/`'. We see this is an example of dividing by a power of two by right shifting (see CS:APP Section 2.3.7). Before shifting by $k = 2$, we must add a bias of $2^k - 1 = 3$ when the dividend is negative.

B. Here is an annotated version of the assembly code.

```
   x86-64 implementation of arith
   x in register %edi
1  arith:
2    leal    3(%rdi), %eax    x+3
3    cmpl    $-1, %edi        Compare x:-1
4    cmovle  %eax, %edi       If <= then x = x+3
5    sarl    $2, %edi         x >>= 2
6    movl    %edi, %eax       return value in %eax
7    ret
```

The `cmovle` instruction conditionally changes the number to be shifted from `x` to `x`$+3$ when `x` ≤ -1.

Problem 4 Solution: [Pg. 19]

Reverse engineering code is a good way to understand machine-level programming. This example gives us a chance to look at the new loop structure.

The following is an annotated version of the assembly code

```
   x86-64 implementation of puzzle
   a in register %edi, b in register %esi
1  puzzle:
      i is in register %esi, which is initialized to a
2    movslq  %esi,%rdx          result = (long) b
3    jmp     .L60               goto middle
4  .L61:                      loop:
5    movslq  %edi,%rax          (long) i
6    subl    %esi, %edi         i -= b
7    imulq   %rax, %rdx         result *= (long) i
8  .L60:                      middle:
9    testl   %edi, %edi         Test i
10   jg      .L61               If >0  goto loop
11   movq    %rdx, %rax         Return result
12   ret
```

A. Variable `result` is in register `%rax`.

B. Variable `i` is in register `%esi`. Since this is an argument register, `i` is initialized to `a`.

C. Here are the definitions of the five expressions

```
#define EXPR1 b
#define EXPR2 a
#define EXPR3 0
#define EXPR4 b
#define EXPR5 i
```

Problem 5 Solution: [Pg. 22]

We can step through the code for `incrprob` and determine the following:

1. The `addl` instruction fetches a 32-bit integer from the location given by the third argument register and adds it to the 32-bit version of the first argument register. From this, we can infer that `t` is the third argument and `x` is the first argument. In addition, `x` must be a signed or unsigned integer, and `t` must be a pointer to a signed or unsigned integer.

2. The `movslq` instruction sign extends the sum (a copy of `*t`) to a long integer. From this, we can infer that `t` must be a pointer to a signed integer.

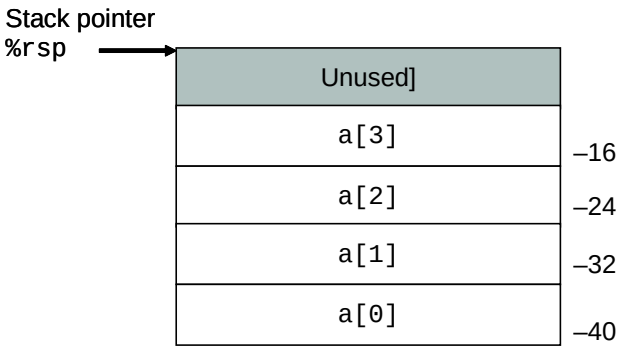
3. The `addq` instruction adds the sign-extended value of the previous sum to the location indicated by the second argument register. From this, we can infer that `q` is the second argument and that it is a pointer to a long integer.

There are two valid prototypes for `incrprob`, which we name `incrprob_s` and `incrprob_u`. They differ only in whether `x` is signed or unsigned.

```
void incrprob_s(int x, long *q, int *t);
void incrprob_u(unsigned x, long *q, int *t);
```

Problem 6 Solution: [Pg. 27]

This function is an example of a leaf function that requires local storage. It can use space beyond the stack pointer for its local storage, never altering the stack pointer.



A.

B.

```
    x86-64 implementation of local_array
    Argument i in %edi
1   local_array:
2     andl    $3, %edi                  idx = i & 3;
3     movq    $2, -40(%rsp)             a[0] = 2
4     movq    $3, -32(%rsp)             a[1] = 3
5     movq    $5, -24(%rsp)             a[2] = 5
6     movq    $7, -16(%rsp)             a[3] = 7
```

```
7   movq    -40(%rsp,%rdi,8), %rax    return a[idx]
8   ret
```

C. The function never changes the stack pointer. It stores all of its local values in the region beyond the stack pointer.

Problem 8 Solution: [Pg. 29]

This problem is similar to CS:APP Practice Problem 3.23, but updated for x86-64.

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

i	c	j	d	Total	Alignment
0	4	8	16	24	8

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

i	c	d	j	Total	Alignment
0	8	9	12	16	8

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	32	8

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

a	p	Total	Alignment
0	48	56	8

Problem 7 Solution: [Pg. 28]

A. Register `%rbx` is used to hold the parameter `x`.

B. Since `%rbx` is callee saved, it must be stored on the stack. Since this is the only use of the stack for this function, the code uses push and pop instructions to save and restore the register.

C.
```
    x86-64 implementation of recursive factorial function rfact
    Argument x in %rdi
1 rfact:
2   testq   %rdi, %rdi          Test x
3   pushq   %rbx                Save %rbx (callee save)
```

```
 4   movl    $1, %eax            result = 1
 5   movq    %rdi, %rbx          Copy x to %rbx
 6   jle     .L9                 if x<=0 goto done
 7   leaq    -1(%rdi), %rdi      xm1 = x-1
 8   call    rfact               rfact(xm1)
 9   imulq   %rbx, %rax          result = x*rfact(x)
10 .L9:                        done:
11   popq    %rbx                Restore %rbx
12   ret                         Return
```

D. Instead of explicitly decrementing and incrementing the stack pointer, the code can use `pushq` and `popq` to both modify the stack pointer and to save and restore register state.

Problem 9 Solution: [Pg. 32]

These cases can be handled by selecting the appropriate entry from the table in Figure 10.

T_x	T_y	Instruction	S	D
`long`	`double`	`cvtsi2sdq`	`%rdi`	`%xmm0`
`double`	`int`	`cvttsd2si`	`%xmm0`	`%eax`
`float`	`double`	`cvtss2sd`	`%xmm0`	`%xmm0`
`long`	`float`	`cvtsi2ssq`	`%rdi`	`%xmm0`
`float`	`long`	`cvtss2siq`	`%xmm0`	`%rax`

Problem 10 Solution: [Pg. 34]

This problem involves the same reasoning as was required to see that numbers declared at label `.LC2` encode 1.8, but with a simpler example.

We see that the two values are 0 and 1077936128 (`0x40400000`). From the high-order bytes, we can extract an exponent field of `0x404` (1028), from which we subtract a bias of 1023 to get an exponent of 5. Concatenating the fraction bits of the two values, we get a fraction field of 0, but with the implied leading value giving value 1.0. The constant is therefore $1.0 \times 2^5 = 32.0$.

Index