
CHAPTER 3

MACHINE-LEVEL REPRESENTATION OF PROGRAMS

CHAPTER 2 RECAP

In chapter 2 we saw:

- **Information is generally organized as sequences of bytes**
- **How different models of computers are used for integers, real numbers, and characters**
- **Different models of computers use different conventions for encoding numbers and for ordering bytes — little/big endian**
- **Discussed how the C language is designed to accommodate a wide range of different implementations in terms of word size and numeric encoding**

CHAPTER 2 RECAP

In chapter 2 we also saw:

- **How most machines use 2's complements, and why they do not use 1's complement nor signed magnitude**
- **How computers use IEEE 754 floating point encoding**
- **The possible effects of casting unsigned to signed, visa versa**
- **The effects overflow has on data**
- **So much more**

SHIFTING FROM DATA REPRESENTATION TO MACHINE REPRESENTATION OF PROGRAMS

➤ **We will now make a huge shift and look at some assembly language.**

INTRO

- **In the earlier days of computing programmers wrote code in assembly. They were usually very skilled programmers and wrote very efficiently optimized code. In today's time we write code in higher level languages, designed to be much easier to understand and readable. This is possible because we have compilers that, for the most part, make our code just as efficient as programs written using assembly.**
- **Although compilers do most of the work in generating assembly code, being able to read and understand assembly is an important skill to have as a programmer.**
- **Basically, the course book says, *"the need for programmers to learn machine level code has shifted over the years from one of being able to write programs directly in assembly code to one of being able to read and understand the code generated by compilers."***
- **The type of assembly used in our book is based on is x86-64 rather than ARM which is what was previously taught in this course**

3.1 A HISTORICAL PERSPECTIVE

- **The Intel processor line (x86), has a long evolutionary line**
 - **8086 (1978, 29K Transistors) - one of the first single-chip, 16-bit microprocessors.**
 - **80286 - (1982 134K Transistors) formed the basis for early PC's — original platform for MS Windows**
 - **i386 (1985, 275K transistors) — expanded to the 32 bit architecture — first model to fully support UNIX OS**
 - **i486 (1989, 1.2M transistors) — improved performance and integrated the floating-point unit onto the processor chip but did not significantly change the instruction set.**

3.1 A HISTORICAL PERSPECTIVE CONTINUED

- **Pentium - (1993, 3.1 M transistors)** Improved performance but only added minor extensions to the instruction set
- **Pentium Pro - (1995, 5.5 M transistors)** - Introduced a radical new processor design. Added a class of “conditional move” instructions to the instruction set.
- **Pentium/MMX - (1997, 4.5 M transistors)** — Added new class of instructions to the Pentium processor for manipulating vectors of integers.
- **Pentium II - (1997, 7M transistors)**
- **Pentium III - (1999, 8.2M transistors)** Introduces SSE, a class of instructions for manipulating vectors of integer or floating-point data. Later versions of this chip went to 24M transistors. And incorporated level 2 cache memory
- **Pentium 4 - (2000, 42 M transistors)** added new data type including double precision floating point, along with 144 new instructions to help with the new floating point
- **Pentium 4E - (2004, 125 M transistors)** Added hyper threading, a method to run two programs simultaneous on a single processor, as well as EM64T, Intel’s implementation of 64 bit extension to IA32 developed by Advanced Micro Devices (AMD), AKA x85-64

3.1 A HISTORICAL PERSPECTIVE CONTINUED

- **Core 2 (2006 291 M transistors) - first multi-core Intel microprocessor, where multiple processors are implemented on a single chip. Did not support hyper-threading.**
- **Core i7 - Nehalem (2008, 781 M transistors) Incorporated both hyper-threading and multi-core, with the initial version supporting two executing programs on each core and up to four cores on each chip**
- **Core i7 - Sandy Bridge (2011, 1.17 G transistors) Introduced AVX, an extension of the SSE to support data packed into 256-bit vectors**
- **Core i7 - Haswell (2013, 1.4 G transistors) Extended AVX to AVX2, adding more instructions and instruction formats.**

TRANSISTORS

- **What is the significance of transistors?**
 - **Much of the success of computers in the past 60+ years has been because of transistors. Developed in the 1940s to replace the vacuum tubes used in TVs, radios, and other electronics.**
 - **They are small, tough, and use low power**
 - **They are used in computers, digital cameras, cell phones, etc.**
 - **Often used as switches. They can turn currents on and off billions of times per second. Used in computers as a basic mechanism for storing and moving data.**

3.2 PROGRAM ENCODINGS

- **We already know what happens when we compile a set of code. We learned the steps of this in Chapter 1. You should review this.**
- **You should also know how to compile a file: `gcc prograName.c -o executableName -save-temps` (and any other flag you want to add) such as `-Wall`**
- **One flag that you may not be familiar with `-Og`**
 - **`-Og` tells the compiler to apply a level of optimization that yields machine code that follows the overall structure of the original C Code**
 - **Enables all optimization that does not conflict with debugging. It can be used with `(-g)` flag for enabling debugging symbols**

3.2 PROGRAM ENCODINGS

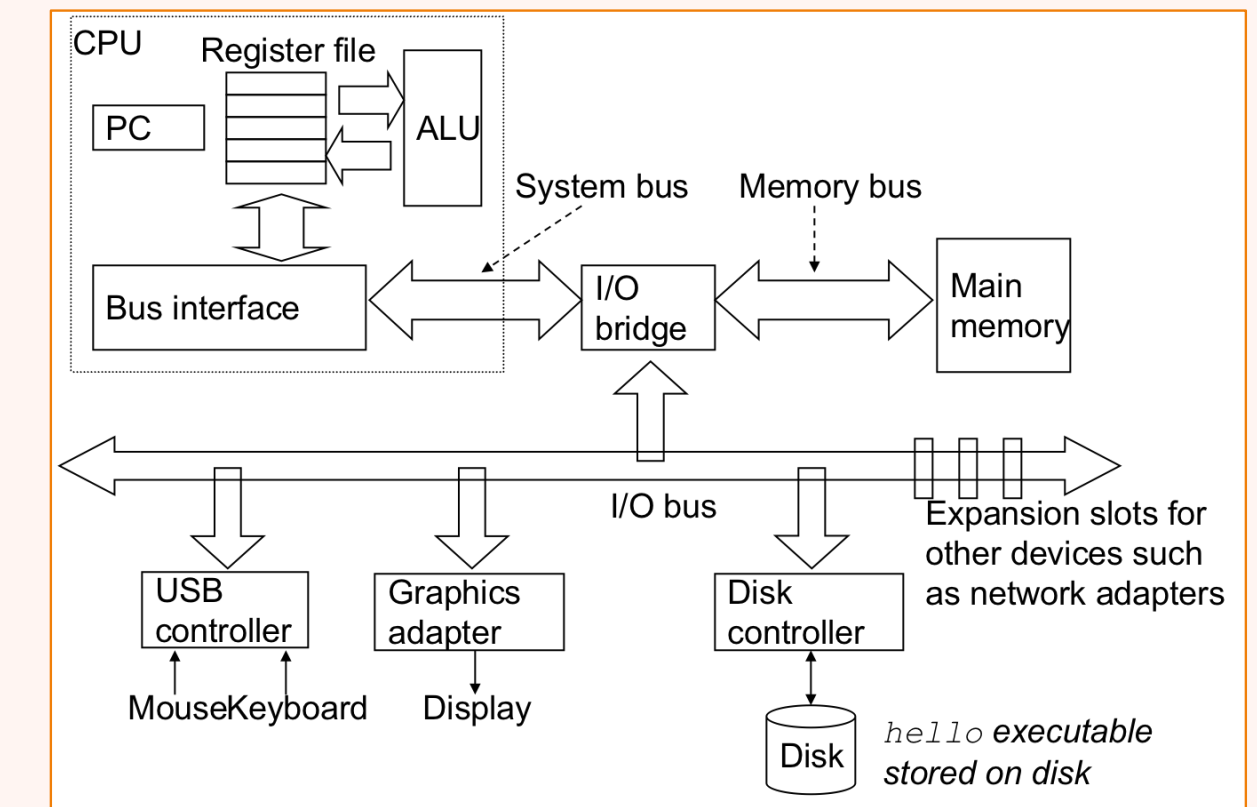
- **Basically this will optimize debugging experience. -Og should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience. It is a better choice than -O0 for producing debuggable code because some compiler disabled the debug information when compiled with zero optimization -O0**
- **Invoking high levels of optimization can generate code that is very difficult to understand**

3.2.1 MACHINE-LEVEL CODE

- **We go from “C” code to “Assembly” code to “Machine” code all during the compile process.**
- **From our point of view, machine code and assembly code is vastly different, but they are really not. It is basically the same, we just can not read machine code but assembly is much more readable. (Well Sorta)**
- **We can understand assembly (with a little practice) but machine code looks like gibberish to us.**

3.2.1 MACHINE-LEVEL CODE

- **Things that are visible to machine code but hidden from the programmer:**
 - **Program counter (PC) - called %rip (instruction pointer register) in x86-64 - this indicates the address in memory of the next instruction to be executed**
 - **Integer register file - contains 16 named locations storing 64-bit values**
 - **They hold register addresses used for C pointers or integer data**
 - **Keep track of program state**
 - **Used to hold temporary data (function arguments, local variables, and return values)**
 - **Condition code registers hold status information about the most recently executed arithmetic or logical instructions. Ex. if and while statements**
 - **A set of vector registers - each hold one or more integer or floating-point values.**



3.2.1 MACHINE-LEVEL CODE

- **A single machine instruction performs only a very elementary operation**
 - **Such as:**
 - **Add 2 numbers stored in registers**
 - **Transfer data between memory and a register, vice versa**
 - **Conditional branch to a new instruction address**
- **The compilers job is to generate sequences of these type instructions to implement program constructs, such as, evaluating a mathematical expressions, loops, or procedure calls and returns.**

3.2.2 CODE EXAMPLE

➤ Suppose we run the following code using `gcc -Og -S mstore.c`

```
long mult2(long, long);  
void multstore(long x, long y, long* text){  
    long t = mult2(x,y);  
    *dest = t;  
}
```

`gcc` will run the compiler, generating only the assembly file `mstore.s` then it will stop the compile process.

`gcc -Og -S mstore.c`

Below is two text document of this ran on Linux and MacOS . You will see they are different. This is due to different compilers.

Code: `mstore_s_linux.txt` and `mstore_s_mac.txt`

DISASSEMBLER EXAMPLE

- **If we use -c command line option, gcc will both compile and assembly the code**
- **This will produce the .o file but not the executable. You will need to use a hex editor or some other tool to see the content of the .o file in hexadecimal. Under normal circumstances it is not readable.**
- **I ran this on my MacOS (gcc -Og -c mstore.c)**
 - **output: mstore.o**
- **We can use a disassembler to inspect the machine-code that is similar to assembly. Linux and the Mac have a program called objdump that will do this for us.**

DISASSEMBLER EXAMPLE

- **Example:**
 - **objdump -d mstore.o**
 - **Or we could use (otool -tvj msore.o) - Otool does not work on SoC servers**
- **Text file of code from the SoC: mstoreDump_linux.txt**
- **Text file of code from my MAC: mstoreDump_mac.txt**

DISASSEMBLER EXAMPLE

➤ **Things to note:**

- **x86-64 instructions can range from 1 - 5 bytes. If an instruction does not need all 5 bytes it need not use them**
- **Some instructions are always associated with a unique byte in the machine instruction ex. 53 is always associated with push %rbx**
- **The disassembler determines the assembly code based purely on the byte sequence in the machine-code file. It does not require access to the source or assembly-code versions of the program.**

Disassembled code from my Mac:
The hex numbers in the 2nd column represent the byte values in groups of 1 to 5 bytes, each of which represent one assembly language instruction shown on the right.

Hex

0: 55	pushq	%rbp
1: 48 89 E5	movq	%rsp, %rbp
4: 53	pushq	%rbx
5: 50	pushq	%rax
6: 48 89 D3	movq	%rdx, %rbx
9: E8 00 00 00 00	callq	0 <_multstore+0xe>
E: 48 89 03	movq	%rax, (%rbx)
11: 48 83 C4 08	addq	\$8, %rsp
15: 5B	popq	%rbx
16: 5D	popq	%rbp
17: C3	retq	

DISASSEMBLED CODE CONTINUED

- **We can also run the disassembler on fully compiled code. This ensures the linking phase is done.**
- **Compile mstoreMain.c mstore.c -o main using -Og and then run the disassembler on the executable (main) objdump -d main **CODE:** mainDump.txt**
- **You may often see one or more NOP in disassembled code — this means no operation. These are here to fill space to better enable placement of the next block of code. This is a memory system performance step.**

3.3 DATA FORMATS

- **In the early days the chip architecture used 16 bits, later expanding to 32 bit**
- **Intel began referring to a 16 bit data type as a “word”, when the switch was made to 32 rather than confuse things by calling a 32 bit data type “word” they dubbed it “double word”. As you can image when the switch was made to 64 bit data type, they are referred to as “quad word”.**

3.3 DATA FORMATS

- Most of the examples in our book will be of type “long” so they are referred to as Quad Word.
- Floating point data come in 2 formats:
 - Single precision (4 bytes)
 - Double precision (8 bytes)
- Each data type has a single letter suffix that indicates the size of the data.
Example: ‘l’ represents a double word, 4 byte int, as well as a data type double which is double precision 8 bytes

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Quad word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

Figure 3.1 Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.

3.4 ACCESSING INFORMATION

- The current x86-64 has 16 general purpose registers, each storing 64 bit values
- The 8086 had eight 16 bit registers (remember from the History section) (%ax - %sp)
- In later versions, these 8 registers were expanded to 32-bit registers (%eax - %esp)
- The x86-64 expanded the original registers to 64 bits and added 8 more registers %r8 - %r15
- As the image indicates, each of the registers can operate on data of different sizes smaller data using the low order bits all the way up to 64 bits.



Figure 3.2 Integer registers. The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

MORE ABOUT REGISTERS

- **Registers are the fastest kind of memory.**
- **There are also some special-purpose registers**
- **Registers have no specific address - not like a pointer**
- **“%rbp points to the bottom of the current function’s stack frame, and local variables are often accessed relative to its value. However, when optimization is on, the compiler may determine that all local variables can be stored in registers. This frees up %rbp for use as another general - purpose register”**

General and special purpose registers

Register Name	32 bit	16 bit	8 bit	Use in calling convention
%rsp	%esp	%sp	%spl	Stack Pointer
%rbp	%ebp	%bp	%bpl	Base Pointer
%rip	%eip	%ip	-	Instruction Pointer (program counter)
%rflags	%eflags	%flags	-	Flags and condition codes

Special-purpose registers

CALLING CONVENTION

- **A calling convention governs how functions on a particular architecture and operating system interact. They ensure functions compiled by different compilers can interoperate, and they ensure that operating systems can run code from different programming languages and compilers.**
- **Including:**
 - **How function arguments are placed**
 - **Where return values go**
 - **What registers functions may use**
 - **How they may allocate local variables**
- **Calling conventions constrain both callers and callee**
 - **A caller is a function that calls another function**
 - **A callee is a function that was called**

ARGUMENT PASSING AND STACK FRAMES

- **On x86-64 Linux, the first six function arguments are passed in registers**
 - **%rdi - (param 1), %rsi - (param 2), %rdx - (param 3), %rcx - (param 4), %r8 - (param 5), %r9 - (param 6)**
 - **The seventh and beyond are passed to the stack (stored on the stack)**
 - **The return register is %rax**

WHAT IF A FUNCTION PARAMETER IS A STRUCTURE?

- **The ABI (application binary interface) - Calling Convention determines how this is handles**
- **Two most used formats:**
 - **64-bit Microsoft ABI - all structures passed by value are pushed to the stack**
 - **X86-64 System V ABI - used by linux/MacOS/BSD - an algorithm is used to determine if the struct can not be stored in registers it will be pushed to the stack**

RETURN ADDRESS AND ENTRY AND EXIT SEQUENCE

The steps required to call a function are sometimes called the *entry sequence* and the steps required to return are called the *exit sequence*. Both caller and callee have responsibilities in each sequence.

To prepare for a function call, the caller performs the following tasks in its entry sequence.

- The caller stores the first six arguments in the corresponding registers.
- If the callee(function being called) takes more than six arguments, or if some of its arguments are large, the caller must store the surplus arguments on its stack frame. It stores these in increasing order so that the 7th argument has a smaller address than the 8th argument, and so forth. The 7th argument must be stored at (%rsp) (that is the top of the stack) when the caller executes its callq instruction.
- The caller saves any caller-saved registers.
- The caller executes callq FUNCTION.

0x7FF7B8CD96D8	Arg12
0x7FF7B8CD96D0	Arg 11
0x7FF7B8CD96C8	Arg 10
0x7FF7B8CD96C0	Arg 9
0x7FF7B8CD96B8	Arg 8
0x77FF7B8CD96B0	Arg 7

%rsp is the top of the stack

RETURN ADDRESS AND ENTRY AND EXIT SEQUENCE

To return from a function:

- **The callee places its return value (an address) in %rax**
- **If necessary, the callee restores the stack pointer to its value at entry (“entry %rsp”)**
- **The callee executes the retq instruction. This has an effect like pop %rip (instruction pointer), which removes the return address from the stack and jumps to that address**
- **The caller then cleans up any space it prepared for arguments and restores caller-saved registers if necessary.**

Particularly simple callees don't need to do much more than return, but most callees will perform more tasks, such as allocating space for local variables and calling functions themselves.

FORMAT EXAMPLE

- The example below simply gives a little explanation of the x86-64 code for the multstore function

void multstore(long x, long y, long* dest)
x is stored in %rdi, y in %rsi, dest in %rdx

```
void multstore(long x, long y, long *dest){  
    long t = mult2(x,y);  
    *dest = t;  
}
```

The code to the right is the format the book often uses for the assembly code along with an explanation. The line numbers are for explanation only. The data in red lines are basically comments.

1	multstore		
2	pushq	%rbx	Save %rbx
3	movq	%rdx, %rbx	Copy dest to %rbx
4	callq	mult2	Call mult2(x,y)
5	movq	%rax, (%rbx)	Store result at*dest
6	popq	%rbx	Restore %rbx
7	ret		Return

3.4.1 OPERAND SPECIFIERS

Many Assembly instructions have one or more operands specifying the source values to use in performing an operation and destination location into which to place the result. Basically these operations are calculating a place in memory. Source values come in the form of constants or can be read from registers or memory. Results can be noted in either registers or memory.

3.4.1 OPERAND SPECIFIERS

There are three type of operand possibilities.

1.Immediate - constant values written using \$ followed by an integer or other value that evaluates to an int (\$-577 or \$0x1F)

2.Register - denotes the content of the register r_a - denotes an arbitrary register “a” $R[r_a]$ indicates its value.

3.Memory - access some memory according to a computed address, a.k.a., effective address

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

Most common form Immediate(base, index, scaler)



If Given

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

Fill in the table showing the values for the indicated operands

Calculate

Register
Memory
Immediate

Operand	Value
%rax	
0x104	
\$0x108	
(%rax)	
4(%rax)	
9(%rax,%rdx)	
260(%rcx,%rdx)	
0xFC(,%rcx,4)	
(%rax,%rdx,4)	

Remember for the memory operands we are accessing memory according to the computed address.

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Figure 3.3 Operand forms. Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

3.4.2 DATA MOVEMENT INSTRUCTIONS

- **Move is one of the most used instructions.**
- **It basically copies data from one location to another.**
- **In this section we will look at several different data movement instructions differing in their source and destination types, what conversions they perform, and other side-effects they may have.**
- **Figure 3.4 depicts the simplest form of a data movement instructions. Which simply copies data from a source location to a destination location without any transformation.**

S = source D = destination

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
movb			Move byte
movw			Move word
movl			Move double word
movq			Move quad word
movabsq	I, R	$R \leftarrow I$	Move absolute quad word

Figure 3.4 Simple data movement instructions.

3.4.2 DATA MOVEMENT INSTRUCTIONS

- **The source (S) operand designates a value that is an immediate, stored in a register, or stored in memory.**
- **The destination (D) operand designates a location that is either a register or a memory address.**
- **X86-64 does not allow both move instructions to refer to memory locations. Copying a value from one memory location to another memory location requires two instructions**
 - **Load the source value into a register**
 - **Then, write the register value to the destination.**

3.4.2 DATA MOVEMENT INSTRUCTIONS

- **The following move instruction examples show the five possible combinations of source and destination types. Recall that the source operand comes first and the destination second.**
- **Can not have memory - memory**

1	<code>movl \$0x4050, %eax</code>	Immediate - Register	4 bytes
2	<code>movw %bp, %sp</code>	Register - Register	2 bytes
3	<code>movb (%rdi, %rcx), %al</code>	Memory - Register	1 byte
4	<code>movb \$-17, (%r10)</code>	Immediate - Memory	1 byte
5	<code>movq %rax, -12(%rbp)</code>	Register - Memory	8 bytes

➤ **Assembly from testCompileOption.s**

➤ **movl \$0, -4(%rbp)**

➤ **movl -8(%rbp), %ecx**

➤ **movl %ecx, -16(%rbp)**

3.4.2 DATA MOVEMENT INSTRUCTIONS

- The movabsq in figure 3.4 is used to deal with 64-bit immediate data.
- The move instruction can only have immediate source operands that can be represented as 32-bit 2's complement. The value is then signed extended to produce a 64-bit number value for the destination.
- The movabsq instruction can have an arbitrary 64-bit immediate value as its source operand and can only have a register as a destination.

S = source D = destination

Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
movb			Move byte
movw			Move word
movl			Move double word
movq			Move quad word
movabsq	I, R	$R \leftarrow I$	Move absolute quad word

Figure 3.4 Simple data movement instructions.

Practice:

For each of the following lines of assembly language, determine the appropriate instruction suffix based on the operands (for example, mov can be rewritten as movb, movw, movl, or movq) use the the registers in the image to help you make this decision. As we have seen the disassemblers do not always show the suffix. Being able to switch between these two forms is an important skill to learn. One important feature is that memory references in x86-64 are always given with quad word registers, such as %rax, even if the operand is a byte, single word, or double word.

	Move	Instruction	Which combination
	movl	%eax , (%rsp)	register -- memory
1.	mov__	(%rax), %dx	
2.	mov__	\$0xFF, %bl	
3.	mov__	(%rsp, %rdx, 4), %dl	
4.	mov__	(%rdx), %rax	
5.	mov__	%dx, (%rax)	

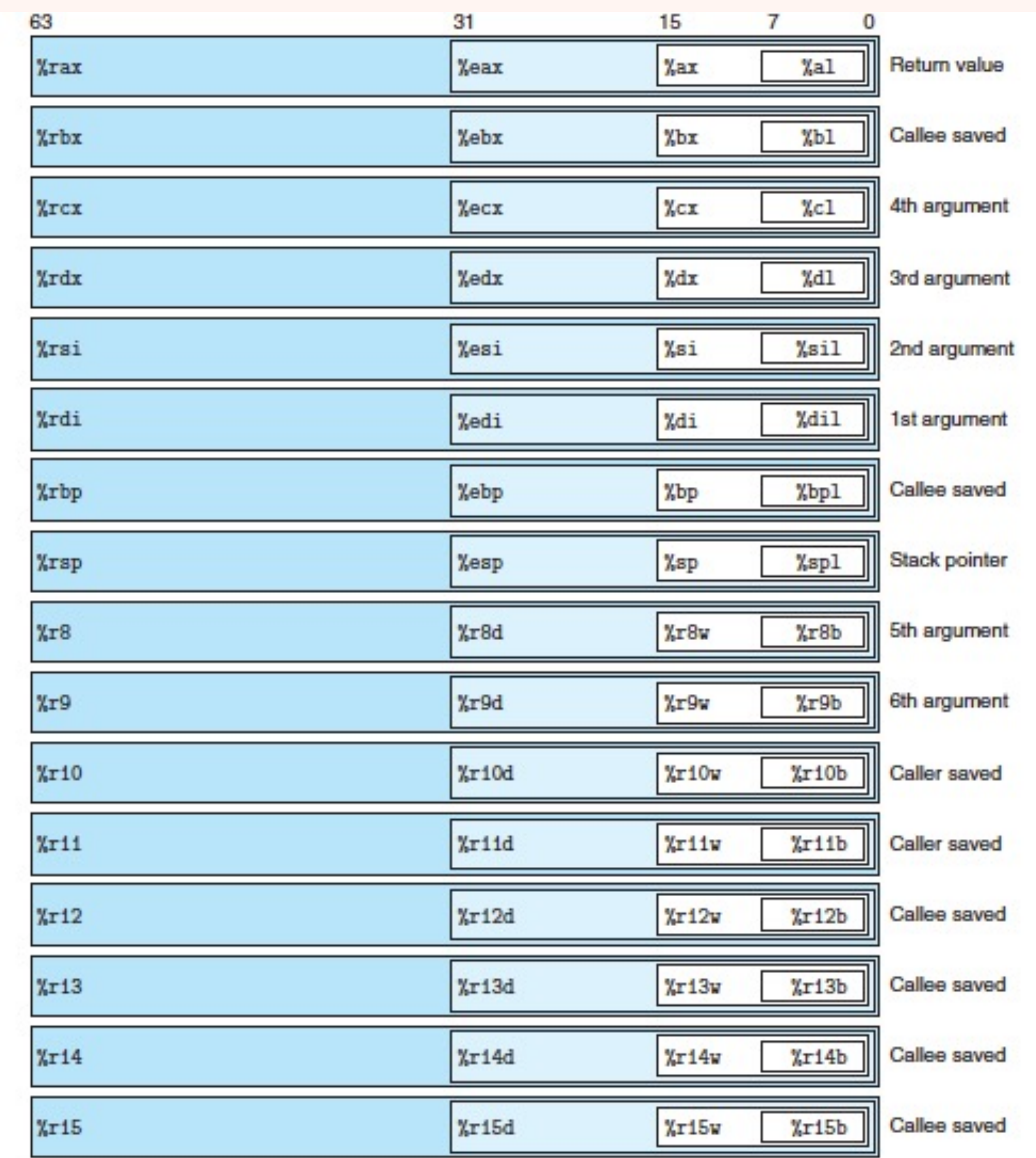


Figure 3.2 Integer registers. The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

PRACTICE PROBLEM

Each of the following lines of assembly code would generate an error message when the assembler is invoked. Explain what is wrong with each line.

Code	Problem
movb \$0xF, (%ebx)	
movl %rax, (%rsp)	
movw (%rax), 4(%rsp)	
movb %al, %sl	
movq %rax, \$0x123	
movl %eax %dx	
movb %si, 8(%rbp)	



Figure 3.2 Integer registers. The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

3.4.3 DATA MOVEMENT EXAMPLE

Let's take a look at the code to the right

- During execution `xp` & `y` are stored in registers `%rdi` and `%rsi` respectively.
- Line 2 reads `xp` from memory (`xp`) and stores the value in register `%rax`
- Line 3 writes `y` to memory as designated by `xp`
- This example illustrates how move instruction can be used to read from memory storing to a register and to write from a register to memory
- `dataMovementEx.s`

(a) C code

```
long exchange(long *xp, long y)
{
    long x = *xp;
    *xp = y;
    return x;
}
```

(b) Assembly code

```
long exchange(long *xp, long y)
xp in %rdi, y in %rsi
1  exchange:
2      movq    (%rdi), %rax    Get x at xp. Set as return value.
3      movq    %rsi, (%rdi)    Store y at xp.
4      ret                    Return.
```

Figure 3.7 C and assembly code for exchange routine. Registers `%rdi` and `%rsi` hold parameters `xp` and `y`, respectively.

`%rax` is used to hold a functions return value

ANOTHER PRACTICE

Suppose you are given the following function prototype. Assume the actual code was compiled into the following assembly code.

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

```
    xp = %rdi, yp = %rsi, zp in %rdx
```

Decode1:

```
    movq (%rdi), %r8  
    movq (%rsi), %rcx  
    movq (%rdx), %rax  
    movq %r8, (%rsi)  
    movq %rcx, (%rdx)  
    movq %rax, (%rdi)  
    ret
```

What do you think the C code for decode1 would look like based the above assembly code?

Decode1.pdf

3.4.2 DATA MOVEMENT INSTRUCTIONS

- **In chapter 2 we talked about what happens when you go from a smaller data type to a larger data type. Anyone remember????**
- **When copying data in x86-64, we have to do an extension which can be zero or sign extension.**

3.4.2 MOVZ

- **movz__ = move zero-extended**
- **The last 2 letters indicate source size to destination size**

Instruction	Effect	Description
MOVZ <i>S, R</i>	$R \leftarrow \text{ZeroExtend}(S)$	Move with zero extension
movzbw		Move zero-extended byte to word
movzbl		Move zero-extended byte to double word
movzwl		Move zero-extended word to double word
movzbq		Move zero-extended byte to quad word
movzwq		Move zero-extended word to quad word

Figure 3.5 Zero-extending data movement instructions. These instructions have a register or memory location as the source and a register as the destination.

3.4.2 MOVs

- **movs __ __ = move sign extension**
- **The last 2 letters indicate source size to destination size**
- **Notice the cltq - this instruction has no operands - it always uses registers %eax (32 bits long) as the source and %rax (64 bits quad) as the destination for the signed-extend result.**
- **But, wait, don't we already have movslq (move signed double to quad)? They do the exact same thing but the encoding of cltq is more compact encoding. (efficiency)**

Instruction	Effect	Description
MOVS <i>S, R</i>	$R \leftarrow \text{SignExtend}(S)$	Move with sign extension
movsbw		Move sign-extended byte to word
movsbl		Move sign-extended byte to double word
movswl		Move sign-extended word to double word
movsbq		Move sign-extended byte to quad word
movswq		Move sign-extended word to quad word
movslq		Move sign-extended double word to quad word
cltq	$\%rax \leftarrow \text{SignExtend}(\%eax)$	Sign-extend %eax to %rax

Figure 3.6 Sign-extending data movement instructions. The movs instructions have a register or memory location as the source and a register as the destination. The cltq instruction is specific to registers %eax and %rax.

MEMORY TO MEMORY EXAMPLE

Assume: `scr_t *sp; dest_t *dp;` where `scr_t` and `dest_t` are typedef

We wish to use the appropriate pair of data movement instructions to implement the operation:

`*dp = (dest_t) *sp;`

`sp` and `dp` are stored in `%rdi` and `%rsi`, respectively.

For each entry in the table on the following slide, show the two instructions that implement the specified data movement.

The first instruction in the sequence reads from memory, does the appropriate conversion, and sets the appropriate portion of register `%rax`.

The second instruction writes the appropriate portion of `%rax` to memory. In both cases, the portions may be `%rax`, `%eax`, `%ax`, or `%al`, and they may differ from one another.

Recall when performing a cast that involves both a size change and a change in “signedness” ; the operation should change the size first. **YOU MUST CONSIDER WHAT CHANGES ARE BEING MADE**

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
char	Byte	b	1
short	Word	w	2
int	Double word	l	4
long	Quad word	q	8
char *	Quad word	q	8
float	Single precision	s	4
double	Double precision	l	8

Figure 3.1 Sizes of C data types in x86-64. With a 64-bit machine, pointers are 8 bytes long.

Quad Word	Double Word	Word	Byte
%rax	%eax	%ax	%al

scr_t *sp stored in %rdi
dest_t *dp stored in %rsi

dest_t = scr_t	instruction	comments
long = long	movq (%rdi), %rax	Read 8 bytes
	movq rax, (%rsi)	store 8 bytes
int = char	movsbl(%rdi), %eax	convert char - int
	movl %eax, (%rsi)	Store 4 bytes
unsigned int = char	movsbl (%rdi), %eax	convert char to int
	movl %eax, (%rsi)	Store 4 bytes
long = unsigned char	movzbq(%rdi), %rax	read byte - z extend
	movq %rax, (%rsi)	Store 8 bytes
char = int	movl (%rdi), %eax	Read 4 bytes
	movb %al, (%rsi)	Store Low-order byte
unsigned char = unsigned int	movl (%rdi), %eax	Read 4 bytes
	movb %al, (%rsi)	store low-order byte
short = char	movsbw (%rdi), %ax	Read byte and s -extend
	movw %ax, (%rsi)	Store 2 bytes

3.4.4 PUSHING AND POPPING STACK DATA

- **A stack is basically a data structure where items are added and deleted using “lifo” last-in, first-out**
- **Push and pop are the assembly operations that are used to add to (push) and remove from (pop) the program stack.**
- **While not called mov, these are considered to be movement operations.**
- **The stack grows downward, such that the top element has the lowest address**
- **The stack pointer %rsp holds the address of the top of the stack element. (Which is the most recent item pushed on the stack)**

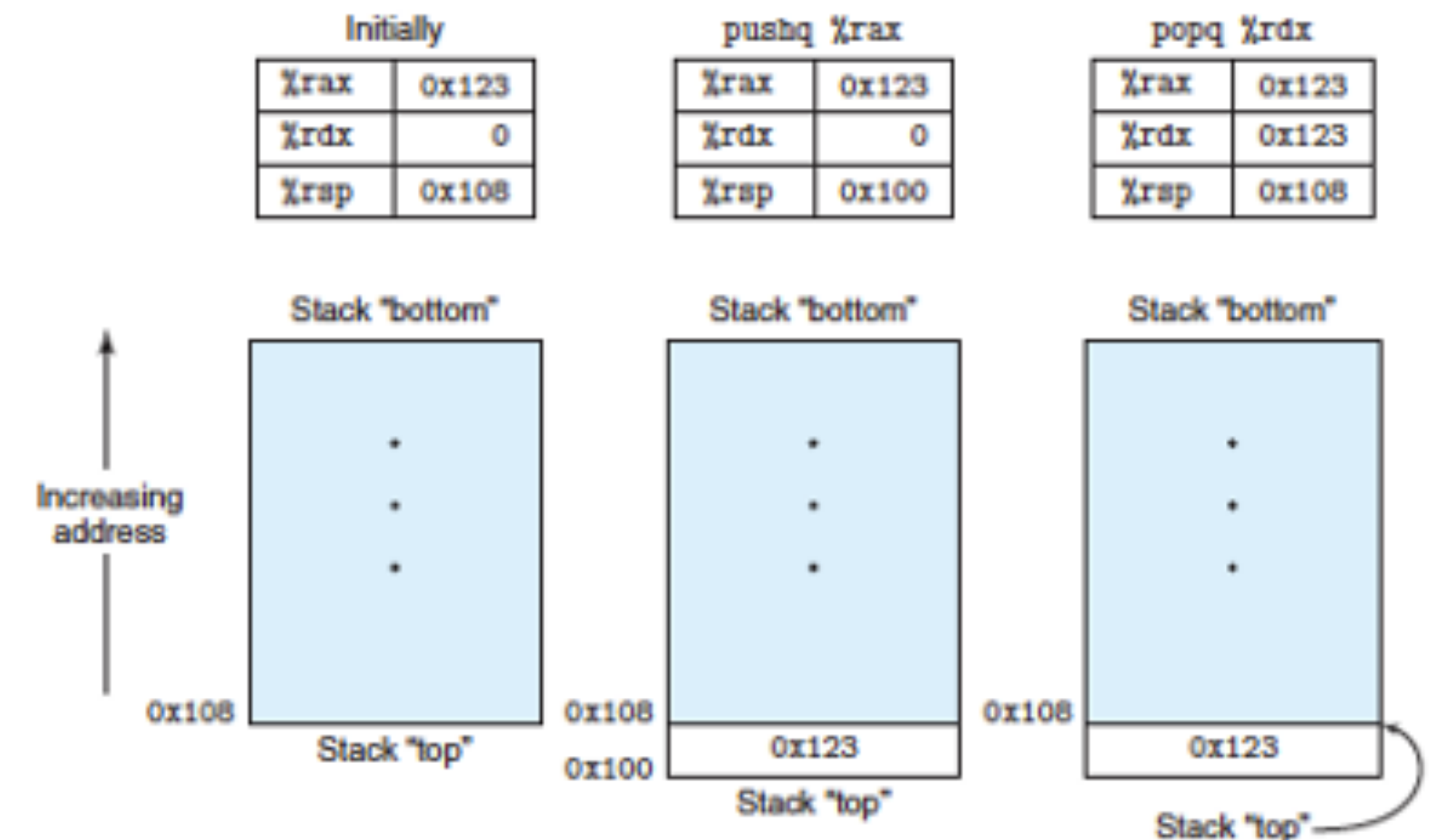
3.4.4 PUSHING STACK DATA

- Pushing a quad word value onto the stack involves first decrementing the stack pointer by 8 and then writing the value at the new top-of-stack address
- `pushq %rax` is the equivalent to the following pair of instructions.
 - `subq $8, %rsp` - decrement the stack pointer
 - `movq %rax, (%rsp)` - store the value of `%rax` on stack
- The first two columns illustrate this:

`rsp` is decremented by 8, $0x108 - 8 = 0x100$ is new top of stack, the value of `rax` (`0x123`) is stored in `rsp` (top of the stack)

Instruction	Effect	Description
<code>pushq S</code>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	Push quad word
<code>popq D</code>	$D \leftarrow M[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	Pop quad word

Figure 3.8 Push and pop Instructions.

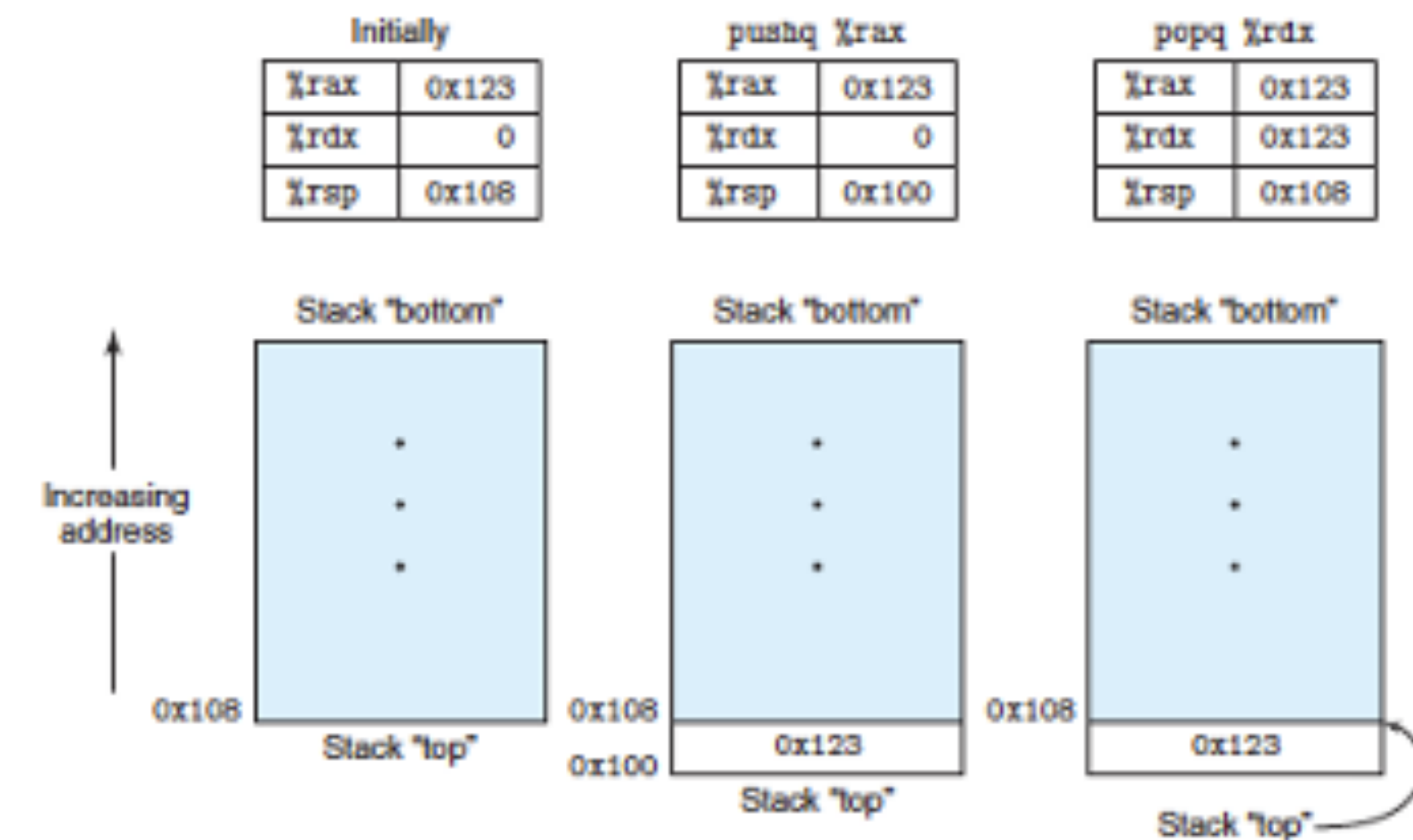


3.4.4 POPPING STACK DATA

- Popping a quad word involves reading from the top-of-stack location and incrementing the stack pointer by 8.
- **popq %rdx** is equivalent to:
 - **movq(%rsp), %rdx** - read from %rsp and store in %rdx
 - **addq \$8, %rsp** - increment stack pointer

Instruction	Effect	Description
pushq <i>S</i>	$R[\%rsp] \leftarrow R[\%rsp] - 8;$ $M[R[\%rsp]] \leftarrow S$	Push quad word
popq <i>D</i>	$D \leftarrow M[R[\%rsp]];$ $R[\%rsp] \leftarrow R[\%rsp] + 8$	Pop quad word

Figure 3.8 Push and pop instructions.



3.5 ARITHMETIC AND LOGICAL OPERATIONS

- The figure to the right list some of the instructions for integer and logic operations.
- Most of these are given as an instruction class but they are used with various size data. Ex. Add is addb, addw, addl, addq. leaq is the exception -why do you think that is?
- These operations are divided into four groups
 - Load effective address
 - Unary
 - Binary
 - Shifts

Instruction		Effect	Description
leaq	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D+1$	Increment
DEC	D	$D \leftarrow D-1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \wedge S$	Exclusive-or
OR	S, D	$D \leftarrow D S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D \ll k$	Left shift
SHL	k, D	$D \leftarrow D \ll k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D \gg_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D \gg_L k$	Logical right shift

Unary

Binary

Shifts

Figure 3.10 Integer arithmetic operations. The load effective address (leaq) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $\gg_A$ and $\gg_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

3.5.1 LOAD EFFECTIVE ADDRESS (LEAQ)

- The load effective address instruction is actually a variant of the movq instruction
- Reads from memory to a register
- Does not reference memory
- The instruction appears to be a memory address but it is reading from the designated location, and copying the effective address to the destination
- Can be used to generate pointers for later memory reference
- Can be used to describe common arithmetic operations EX.
 - If %rdx contains value x , instruction : `leaq 7(%rdx, %rdx, 4), %rax` will set %rax to $5x + 7$ (how did we get $5x + 7$ here)
- While the above line may not really relate to effective address, the compiler often will do this type of computation using leaq.

3.5.1 LOAD EFFECTIVE ADDRESS (LEAQ)

As an illustration of the use of `leaq` in compiled code, consider the following C program:

```
long scale(long x, long y, long z) {  
    long t = x + 4 * y + 12 * z;  
    return t;  
}
```

When compiled, the arithmetic operations of the function are implemented by a sequence of three `leaq` functions, as is documented by the comments on the right-hand side:

```
long scale(long x, long y, long z)  
x in %rdi, y in %rsi, z in %rdx  
scale:  
leaq    (%rdi,%rsi,4), %rax    x + 4*y  
leaq    (%rdx,%rdx,2), %rdx    z + 2*z = 3*z  
leaq    (%rax,%rdx,4), %rax    (x+4*y) + 4*(3*z) = x + 4*y + 12*z  
ret
```

The ability of the `leaq` instruction to perform addition and limited forms of multiplication proves useful when compiling simple arithmetic expressions such as this example.

PRACTICE

`%rax = x` `%rcx = y`

- Suppose register `%rax` holds value `x` and `%rcx` holds value `y`.
- Fill in the table below with formulas indicating the value that will be stored in register `%rdx` for each of the given assembly-code instruction:

	INSTRUCTION	RESULT
1.	<code>leaq 6(%rax), %rdx</code>	<code>6 + x</code>
2.	<code>leaq (%rax, %rcx), %rdx</code>	<code>x+y</code>
3.	<code>leaq (%rax, %rcx, 4), %rdx</code>	<code>x + 4y</code>
4.	<code>leaq 7(%rax, %rax, 8), %rdx</code>	<code>7 + x +8x = 9x + 7</code>
5.	<code>leaq 0xA(, %rcx, 4), %rdx</code>	<code>10 + 4y</code>
6.	<code>leaq 9(%rax, %rcx, 2), %rdx</code>	<code>9 + x + 2y</code>

3.5.2 UNARY AND BINARY OPERATIONS

- **Unary operations have a single operand serving as both source and destination**
- **It can be either a register or memory location**
- **These are inc, dec, neg, and not, with each being used with all sizes. Ex. incq (%rsp) - this will cause the 8 byte element to be incremented**
- **Binary operations, (add, sub, imul, xor, or, and) where the 2nd operand is used as both source and destination.**
 - **Ex. sub %rax, %rdx (subtract %rax from %rdx) %rax can be - immediate, register, or memory, %rdx can be a register or memory - however both %rax and %rdx cannot both be memory.**
- **When the 2nd operand is a memory location, the processor must read the value from memory, perform the operation, then write the result back to memory.**

Relative snippet from figure 3.10

INC	<i>D</i>	$D \leftarrow D + 1$	Increment	← Unary
DEC	<i>D</i>	$D \leftarrow D - 1$	Decrement	
NEG	<i>D</i>	$D \leftarrow -D$	Negate	
NOT	<i>D</i>	$D \leftarrow \sim D$	Complement	
ADD	<i>S, D</i>	$D \leftarrow D + S$	Add	← Binary
SUB	<i>S, D</i>	$D \leftarrow D - S$	Subtract	
IMUL	<i>S, D</i>	$D \leftarrow D * S$	Multiply	
XOR	<i>S, D</i>	$D \leftarrow D \sim S$	Exclusive-or	
OR	<i>S, D</i>	$D \leftarrow D S$	Or	
AND	<i>S, D</i>	$D \leftarrow D \& S$	And	

3.5.3 SHIFT OPERATIONS

- Both arithmetic and logical right shifts are possible
- The shift amount can be an immediate or with the single-byte register %cl
- Immediate - you specify the shift amount. $x \ll= 4$
- %cl - the shift is determined by the variable being used. $x \gg= n$ if $n = \text{byte}$ the instruction would be sarb, word - sarw, double word - sarl, quad - sarq

SAL	k, D	$D \leftarrow D \ll k$	Left shift		Shifts
SHL	k, D	$D \leftarrow D \ll k$	Left shift (same as SAL)		
SAR	k, D	$D \leftarrow D \gg_A k$	Arithmetic right shift		
SHR	k, D	$D \leftarrow D \gg_L k$	Logical right shift		

Why do you think the shift only uses %cl? 1 byte

EXAMPLE

In general the compiler generates code that uses individual register for multiple program values and moves program values among the registers.

Instruction	Effect	Description
leaq <i>S, D</i>	$D \leftarrow \&S$	Load effective address
INC <i>D</i>	$D \leftarrow D+1$	Increment
DEC <i>D</i>	$D \leftarrow D-1$	Decrement
NEG <i>D</i>	$D \leftarrow -D$	Negate
NOT <i>D</i>	$D \leftarrow \sim D$	Complement
ADD <i>S, D</i>	$D \leftarrow D + S$	Add
SUB <i>S, D</i>	$D \leftarrow D - S$	Subtract
IMUL <i>S, D</i>	$D \leftarrow D * S$	Multiply
XOR <i>S, D</i>	$D \leftarrow D \wedge S$	Exclusive-or
OR <i>S, D</i>	$D \leftarrow D S$	Or
AND <i>S, D</i>	$D \leftarrow D \& S$	And
SAL <i>k, D</i>	$D \leftarrow D \ll k$	Left shift
SHL <i>k, D</i>	$D \leftarrow D \ll k$	Left shift (same as SAL)
SAR <i>k, D</i>	$D \leftarrow D \gg_A k$	Arithmetic right shift
SHR <i>k, D</i>	$D \leftarrow D \gg_L k$	Logical right shift

Figure 3.10 Integer arithmetic operations. The load effective address (leaq) instruction is commonly used to perform simple arithmetic. The remaining ones are more standard unary or binary operations. We use the notation $\gg_A$ and $\gg_L$ to denote arithmetic and logical right shift, respectively. Note the nonintuitive ordering of the operands with ATT-format assembly code.

(a) C code

```
long arith(long x, long y, long z)
{
    long t1 = x ^ y;
    long t2 = z * 48;
    long t3 = t1 & 0x0F0F0F0F;    (252645135)
    long t4 = t2 - t3;
    return t4;
}
```

(b) Assembly code

```
long arith(long x, long y, long z)
x in %rdi, y in %rsi, z in %rdx
1  arith:
2      xorq    %rsi, %rdi          t1 = x ^ y
3      leaq    (%rdx,%rdx,2), %rax  3*z
4      salq    $4, %rax            t2 = 16 * (3*z) = 48*z
5      andl    $252645135, %edi    t3 = t1 & 0x0F0F0F0F
6      subq    %rdi, %rax          Return t2 - t3
7      ret
```

What does left shift do?

Why only edi?

Figure 3.11 C and assembly code for arithmetic function.

3.5.5 SPECIAL ARITHMETIC OPERATIONS

- We learned in chapter 2 that if we multiply a piece of data of size 1 byte the result is 2 bytes. What about 8 bytes (64 bits)? How is that handled?
- x86-64 calls a 128 bit an oct word.
- The high order 64 bits are stored in %rdx and the low order 64 bits are stored in %rax
- For unsigned data mulq is used
- For signed imulq is used

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
cqto		$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	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 3.12 Special arithmetic operations. These operations provide full 128-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.

3.5.5 SPECIAL ARITHMETIC OPERATIONS

- While figure 3.10 does not list division and mod as operators, they are still provided by the single operand divide instruction similar to the single operand multiply instruction
- The signed division instruction “idivq” takes as its dividend the 128-bit quantity in registers %rdx (high order bits) and %rax(low-order 64 bits)
- The divisor is given as the instruction operand
- The instructions stores the quotient in register %rax and the remainder in %rdx
- Unsigned division uses divq and %rdx is set to 0.

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
cqto		$R[\%rdx]:R[\%rax] \leftarrow \text{SignExtend}(R[\%rax])$	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 3.12 Special arithmetic operations. These operations provide full 128-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.

3.6 CONTROL

- **In C we have conditionals, loops, and switches that require conditional execution**
- **These conditionals are different than the single line instructions we have seen thus far**
- **With conditional operations we have a sequence of operations that get performed depending on the outcome of some test that is applied to data**
- **There are two basic low-level mechanisms for implementing conditional behavior**
 - **Test data values**
 - **Alter either the control flow or the data flow based on the result of the test**
- **For this sections, first we will examine, data-dependent control flow for implementing conditional behavior.**
- **We will look briefly at Jump instructions**
- **We will also discuss loops and possibly switch statements.**

3.6.1 CONDITION CODES

- Along with the 16 general purpose registers we have seen, there are also a set of single-bit condition code registers
- They describe attributes of arithmetic or logical operations
- They can be used to perform conditional branches
 - **CF: Carry flag** - detects overflow or hold carry of most significant bit
 - **ZF: Zero flag** - operation that yields a zero
 - **SF: Sign Flag** - most recent operation yielded a negative value
 - **OF: overflow Flag** - operation caused a 2's complement overflow; either - or +; this flag is also used with shifts
- All instructions listed in fig. 3.10, except leaq, cause one or more of the condition codes to be set.

Flag	Expression	Result
CF	(unsigned)t < (unsigned)a	Unsigned overflow
ZF	(t == 0)	Zero
SF	(t < 0)	Negative
OF	(A<0 == b<0) && (t<0 != a<0)	Signed overflow

3.6.1 CONDITION CODES

- In addition to the instruction codes from 3.10 there are 2 instruction classes with 8, 16, 32, and 64 bit form that set condition codes without altering other registers
 - **CMP**
 - **TEST**
- **CMP** sets condition codes based on the differences of there two operands - acts like the SUB instruction but does not update the register (notice the order of the source when doing the subtraction)
- **TEST** instruction work the same as AND but does not update the register.
 - **Ex: testq %rax, %rax** to see whether %rax is negative, zero, or positive.

Instruction	Based on	Description
CMP	S_1, S_2	$S_2 - S_1$
cmpb		Compare byte
cmpw		Compare word
cmpl		Compare double word
cmpq		Compare quad word
TEST	S_1, S_2	$S_1 \& S_2$
testb		Test byte
testw		Test word
testl		Test double word
testq		Test quad word

Figure 3.13 Comparison and test Instructions. These instructions set the condition codes without updating any other registers.

3.6.2 ACCESSING THE CONDITION CODES

- **There are three common ways to use condition codes**
 - **Set a single byte to 0 or 1 depending on some combination of the condition code**
 - **Conditionally jump to some other part of the program**
 - **Conditionally transfer data**

3.6.2 ACCESSING THE CONDITION CODES

➤ For the first case we have SET instructions

l is not long

b is not byte

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

Figure 3.14 The SET instructions. Each instruction sets a single byte to 0 or 1 based on some combination of the condition codes. Some instructions have "synonyms," that is, alternate names for the same machine instruction.

Signed

Unsigned

Destination is a low order single byte register or memory location.

CONSIDER THIS CODE

int comp(data_t a, data_t b)

a in %rdi, b in %rsi

1 comp:

2 cmpq %rsi, %rdi Compare a:b (This will set 1 or more of the conditional flags)

3 setl %al Set low-order byte of %eax to 0 or 1 (based on SF^OF)

4 movzbl %al, %eax Clears remaining bits of %eax (basically %rax)

Practice13_14

3.6.3 JUMP INSTRUCTION

- Normal execution of a C program is one line after the other. This is not always the case in assembly.
- A jump instruction can cause the execution to switch to a completely new position in the program.
- The jump destination is usually indicated by labels (you saw these in the assembly lab)

Ex.

movq \$0, %rax	set %rax to 0
jmp .L1	Goto .L1
movq (%rax), %rdx	Null pointer dereference (skipped)

.L1:

popq %rdx	Jump target
------------------	--------------------

jmp .L1 will cause the program to skip over the movq instruction and instead resume execution with the popq instruction. The addresses of the labels are determined by the assembler during the compile process.

3.6.3 JUMP INSTRUCTIONS

- These are the different jump instructions
 - The first set -`jmp` is an unconditional jump. It has 2 types
 - Direct - jump is encoded to specific target using label
 - Indirect - jump target reads from memory - written using `*`
- Ex**
- `jmp *%rax` - value in the register acts as the label
- `jmp *(%rax)` - reads the target from memory dereferenced and goes to that address.
- The second set - has conditions (shown in the figure to the right) if the condition holds then the jump is executed

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

Figure 3.15 The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have “synonyms,” alternate names for the same machine instruction.

CONDITIONAL BRANCHES WITH CONDITIONAL MOVES

cmovExample

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

Figure 3.18 The conditional move instructions. These instructions copy the source value *S* to its destination *R* when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

3.6.7 LOOPS (DO-WHILE)

- **C provides constructs - do-while, while, and for**
- **There are no instruction sets for these. They are handled using a combination of conditional test and jumps.**
- **Figure 3.19 is an example of the do-while**
- **As you can see from line 7 there is a conditional jump (jg - if the compare on line 6 is true the conditional jump will continue to execute the loop.**

DoWhilePractice

(a) C code

```
long fact_do(long n)
{
    long result = 1;
    do {
        result *= n;
        n = n-1;
    } while (n > 1);
    return result;
}
```

(b) Equivalent goto version

```
long fact_do_goto(long n)
{
    long result = 1;
loop:
    result *= n;
    n = n-1;
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

```
long fact_do(long n)
n in %rdi
1 fact_do:
2     movl    $1, %eax        Set result = 1
3     .L2:                                loop:
4     imulq   %rdi, %rax       Compute result *= n    %rax = %rdi*%rax
5     subq    $1, %rdi        Decrement n
6     cmpq    $1, %rdi        Compare n:1
7     jg      .L2             If >, goto loop
8     rep; ret                Return
```

Figure 3.19 Code for do-while version of factorial program. A conditional jump causes the program to loop.

3.6.7 (WHILE)

- The while loop is slightly different than do-while
- The loop is potentially terminated before the first execution of the body
- There are multiple ways to implement a while loop
- Again there is a conditional jump (jg) that is based on a cmpq instruction
- Notice this jump goes to .L6 if condition is met. The table .L6 does the multiplication and decrement

Note: The book shows other versions of while loop for x-86-64 assembly when various command line compile optimization codes are used. You should take a look at these and the explanations given .

Loop_while Practice

(a) C code

```
long fact_while(long n)
{
    long result = 1;
    while (n > 1) {
        result *= n;
        n = n-1;
    }
    return result;
}
```

(b) Equivalent goto version

```
long fact_while_jm_goto(long n)
{
    long result = 1;
    goto test;
loop:
    result *= n;
    n = n-1;
test:
    if (n > 1)
        goto loop;
    return result;
}
```

(c) Corresponding assembly-language code

```
long fact_while(long n)
n in %rdi
fact_while:
    movl    $1, %eax        Set result = 1
    jmp     .L5              Goto test
.L6:                          loop:
    imulq   %rdi, %rax        Compute result *= n
    subq    $1, %rdi          Decrement n
.L5:                          test:
    cmpq    $1, %rdi          Compare n:1
    jg      .L6               If >, goto loop
    rep; ret                  Return
```

Figure 3.20 C and assembly code for while version of factorial using jump-to-middle translation. The C function fact_while_jm_goto illustrates the operation of the assembly-code version.

3.6.7 (FOR)

- The C language standard states for the most part, the behavior of a “for” loop is identical to the behavior of a while loop.
- If the optimization code -Og is used the assembly is very similar to a while loop. Basically the compiler will make a few changes to the for loop transforming it into code very similar to the while loop
- Consider the following code:

```
long fact_for(long n){  
    long i;  
    long result = 1;  
    for(i = 2; i <= n; i++){ result *= i;}  
    return result;  
}
```

```
long fact_for(long n)
n in %rdi
fact_for:
    movl    $1, %eax        Set result = 1
    movl    $2, %edx        Set i = 2
    jmp     .L8             Goto test
.L9:                        loop:
    imulq   %rdx, %rax       Compute result *= i
    addq    $1, %rdx         Increment i
.L8:                        test:
    cmpq    %rdi, %rdx       Compare i:n
    jle     .L9              If <=, goto loop
    rep; ret                 Return
```

This is the assembly code produced by gcc using the -Og optimization code.

3.6.8 SWITCH

- **A switch statement provides multi-way branching based on the value of an index**
 - **They are useful when dealing with test, where there can be a large number of possible outcomes**
 - **Allows for efficient implementation using a data structure called a jump table**
 - **An array where each entry is the address of a code segment implementing the action the program should take when the switch index = i**
 - **The code performs an array reference into the jump table using the switch index to determine the target for a jump instruction.**
 - **Based on the # of cases, during compile time the code may be changed to a more efficient set of code.**
 - **We will look at an example.**
-

- **Couple things to point out.**
- **What happens when you do not have a break? What if x is 102?**
- **Line 5 of (b) creates a jump table**
- **Line 6 using the jump table to go directly to the appropriate place**
- **Line 13 says go directly to the default if the index is greater than 6**
- **Line 27 combines case 104 and 106 since there is nothing in 104**
- **Now lets look at the assembly**

(a) Switch statement	(b) Translation into extended C
<code>void switch_eg(long x, long n,</code>	<code>1 void switch_eg_impl(long x, long n,</code>
<code>long *dest)</code>	<code>2 long *dest)</code>
<code>{</code>	<code>3 {</code>
<code>long val = x;</code>	<code>4 /* Table of code pointers */</code>
<code>switch (n) {</code>	<code>5 static void *jt[7] = {</code>
<code>case 100:</code>	<code>6 &&loc_A, &&loc_def, &&loc_B,</code>
<code>val += 13;</code>	<code>7 &&loc_C, &&loc_D, &&loc_def,</code>
<code>break;</code>	<code>8 &&loc_D</code>
<code>case 102:</code>	<code>9 };</code>
<code>val += 10;</code>	<code>10 unsigned long index = n - 100;</code>
<code>/* Fall through */</code>	<code>11 long val;</code>
<code>case 103:</code>	<code>12</code>
<code>val += 11;</code>	<code>13 if (index > 6)</code>
<code>break;</code>	<code>14 goto loc_def;</code>
<code>case 104:</code>	<code>15 /* Multiway branch */</code>
<code>case 106:</code>	<code>16 goto *jt[index];</code>
<code>val += val;</code>	<code>17</code>
<code>break;</code>	<code>18 loc_A: /* Case 100 */</code>
<code>default:</code>	<code>19 val = x * 13;</code>
<code>val = 0;</code>	<code>20 goto done;</code>
<code>}</code>	<code>21 loc_B: /* Case 102 */</code>
<code>*dest = val;</code>	<code>22 x = x + 10;</code>
<code>}</code>	<code>23 /* Fall through */</code>
	<code>24 loc_C: /* Case 103 */</code>
	<code>25 val = x + 11;</code>
	<code>26 goto done;</code>
	<code>27 loc_D: /* Cases 104, 106 */</code>
	<code>28 val = x * x;</code>
	<code>29 goto done;</code>
	<code>30 loc_def: /* Default case */</code>
	<code>31 val = 0;</code>
	<code>32 done:</code>
	<code>33 *dest = val;</code>
	<code>34 }</code>

Figure 3.22 Example switch statement and its translation into extended C. The translation shows the structure of jump table `jt` and how it is accessed. Such tables are supported by gcc as an extension to the C language.

3.6.8 SWITCH

```
void switch_eg(long x, long n, long *dest)
x in %rdi, n in %rsi, dest in %rdx

1  switch_eg:
2      subq    $100, %rsi          Compute index = n-100
3      cmpq    $6, %rsi           Compare index:6
4      ja      .L8                If >, goto loc_def
5      jmp     *.L4(,%rsi,8)       Goto *jg[index]
6  .L3:                                loc_A:
7      leaq    (%rdi,%rdi,2), %rax  3*x
8      leaq    (%rdi,%rax,4), %rdi  val = 13*x
9      jmp     .L2                Goto done
10 .L5:                                loc_B:
11      addq    $10, %rdi          x = x + 10
12 .L6:                                loc_C:
13      addq    $11, %rdi          val = x + 11
14      jmp     .L2                Goto done
15 .L7:                                loc_D:
16      imulq   %rdi, %rdi         val = x * x
17      jmp     .L2                Goto done
18 .L8:                                loc_def:
19      movl    $0, %edi           val = 0
20 .L2:                                done:
21      movq    %rdi, (%rdx)       *dest = val
22      ret                          Return
```

Figure 3.23 Assembly code for switch statement example in Figure 3.22.

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

```
1      .section      .rodata
2      .align 8      Align address to multiple of 8
3  .L4:
4      .quad    .L3      Case 100: loc_A
5      .quad    .L8      Case 101: loc_def
6      .quad    .L5      Case 102: loc_B
7      .quad    .L6      Case 103: loc_C
8      .quad    .L7      Case 104: loc_D
9      .quad    .L8      Case 105: loc_def
10     .quad    .L7      Case 106: loc_D
```

```
/* Table of code pointers */
static void *jt[7] = {
    &loc_A, &loc_def, &loc_B,
    &loc_C, &loc_D, &loc_def,
    &loc_D
};
```

(a) Switch statement

```
void switch_eg(long x, long n,
               long *dest)
{
    long val = x;

    switch (n) {

        case 100:
            val += 13;
            break;

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

        case 103:
            val += 11;
            break;

        case 104:
        case 106:
            val += val;
            break;

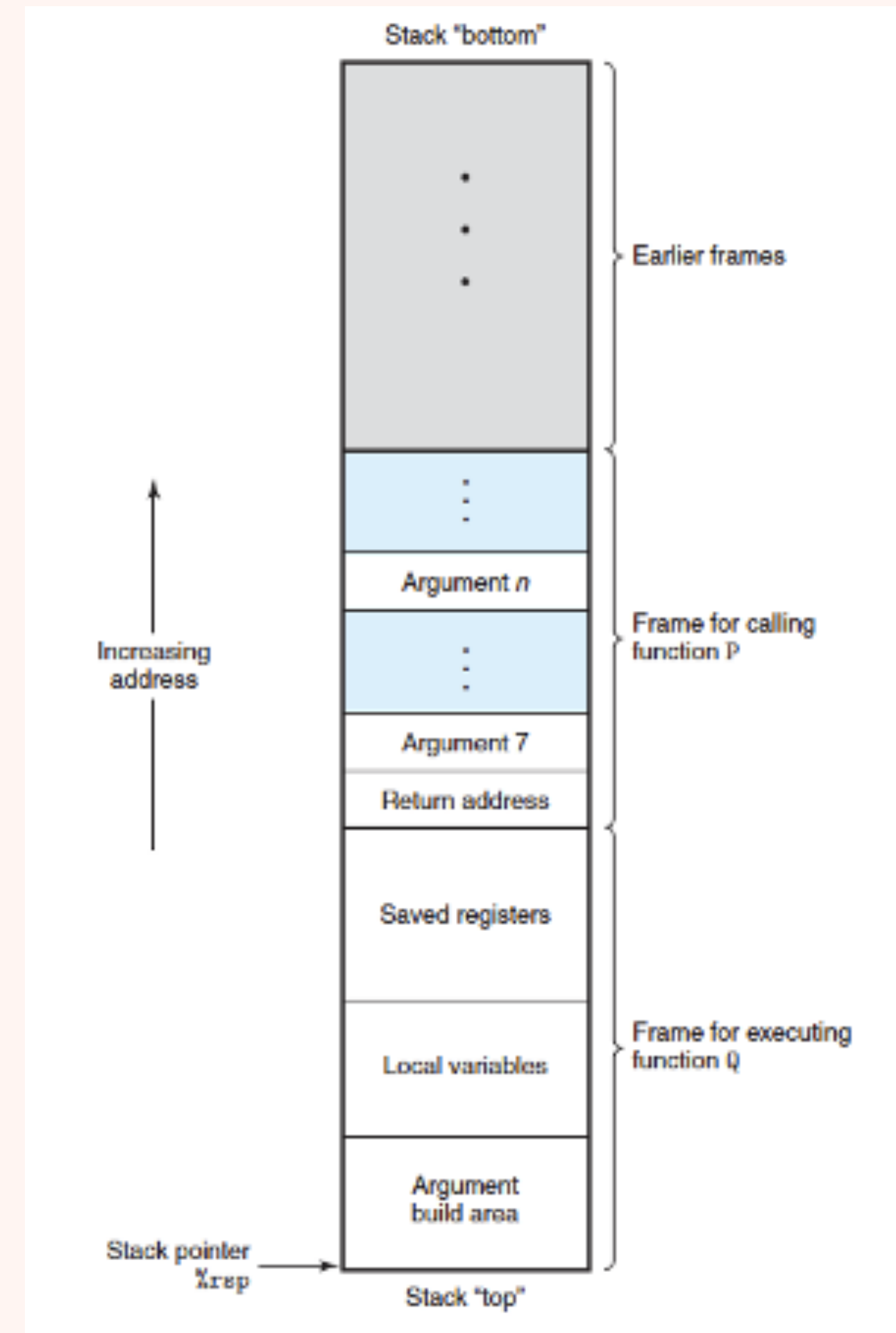
        default:
            val = 0;
    }
    *dest = val;
}
```

3.7 PROCEDURES

- **Machine-level support for procedures must take care to handle the many attributes they have. We will consider procedure Q and P. Suppose P calls Q, and Q then executes and returns to the calling procedure P.**
- **These actions involve one or more of the following mechanisms:**
 - **Passing control - The PC(program counter) must be set to the starting address of the code for Q upon entry and then set to the instruction in P following the call to Q upon return.**
 - **Passing Data - P must be able to provide one or more parameters to Q, and Q must be able to return a value back to P.**
 - **Allocating and deallocating memory - Q may need to allocate space for local variables when it begins and then free that storage before it returns.**
- **x86-64 implementation of procedures involves a combination of special instructions and a set of conventions on how to use the machine resources, such as the registers and the program memory.**

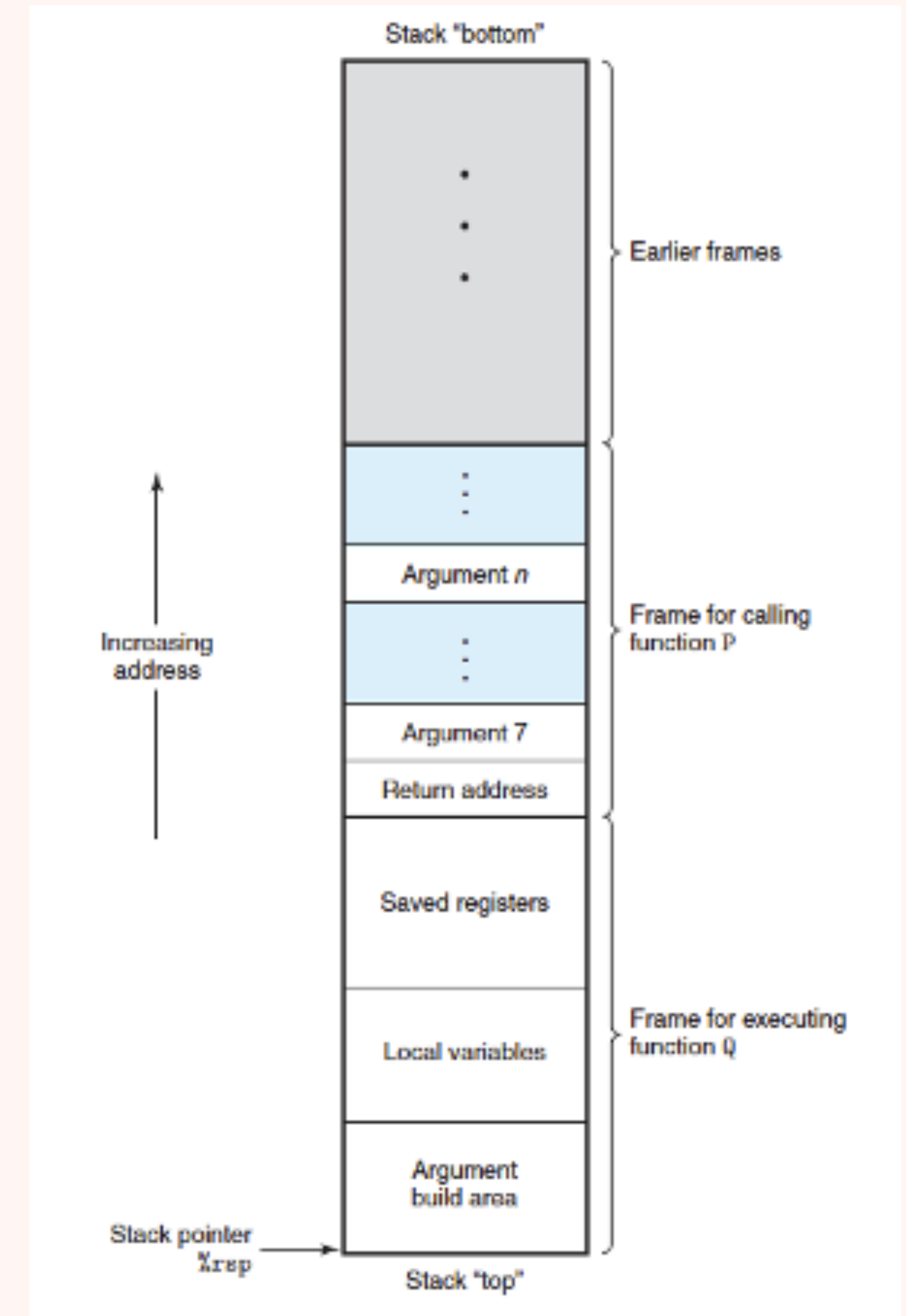
3.7.1 RUN-TIME STACK

- We have already discussed stack and how it uses the LIFO mechanism
- Considering the P and Q example:
 - While Q is running it is the only procedure that needs to allocate new storage on the stack for its variables. When Q is done, it returns all the storage for the local variables back to the stack.
- Also, when an x86-64 requires storage beyond what it can hold in registers, it allocates space on the stack. This region is referred to the procedures run-time stack frame.



3.7.1 RUN-TIME STACK

- Remember P called Q
- Q did what it needed to do
- Q returned to the address P gave for the return.
- P was pushed on the stack first along with any arguments it needs to pass to Q
- The last thing pushed on is the address of where Q is supposed to return to.
- For efficiency, procedures only use the stack frame if needed. If registers can be used and the functions are small with limited arguments the stack frame may not be needed.



3.7.2 CONTROL TRANSFER

- **Continuing with the P and Q example**
- **When P passes control to Q the PC (program counter) is set to Q**
- **So what happens when Q returns? How does it know where to return?**
- **Q is called using a 'call' instruction - this instruction first pushes an address on the stack. This address is where Q returns when it is finished. This is done using the 'ret' instruction.**
- **The instruction ret pops the address off the stack and sets the PC to that address.**

3.7.3 DATA TRANSFER

- **In addition to passing control to a procedure when called, and then back again when the procedure returns, procedure calls may involve passing data as arguments, and returning from a procedure may also involve returning a value. With x86-64 most of this passing of data to and from procedures is done using registers.**
- **Back to P and Q - when P calls Q the code for P must copy the arguments into the proper registers**
- **When Q returns back to P, the code for P can access the returned value in register `%rax`**

3.7.3 DATA TRANSFER

- **x86-64 allows up to 6 arguments to be handled through registers (Now it makes sense as to why my professors always told me to try to limit the # of parameters in my functions. Use of registers over storing in memory is more efficient.)**
- **Arguments past 6 are passed to the stack rather than storing them in registers**

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

Figure 3.28 Registers for passing function arguments. The registers are used in a specified order and named according to the argument sizes.

3.7 EXAMPLE

(a) C code

```
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;
}
```

(b) Generated assembly code

```
void proc(a1, a1p, a2, a2p, a3, a3p, a4, a4p)
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 %rsp+8    ( 8 bits)
a4p at %rsp+16  (64 bits)

1 proc:
2 movq 16(%rsp), %rax    Fetch a4p (64 bits)
3 addq %rdi, (%rsi)      *a1p += a1 (64 bits)
4 addl %edx, (%rcx)      *a2p += a2 (32 bits)
5 addw %r8w, (%r9)       *a3p += a3 (16 bits)
6 movl 8(%rsp), %edx      Fetch a4 ( 8 bits)
7 addb %dl, (%rax)        *a4p += a4 ( 8 bits)
8 ret                    Return
```

First six stored in registers

These 2 are Stored on the stack

Figure 3.30 Stack frame structure for function proc. Arguments a4 and a4p are passed on the stack.

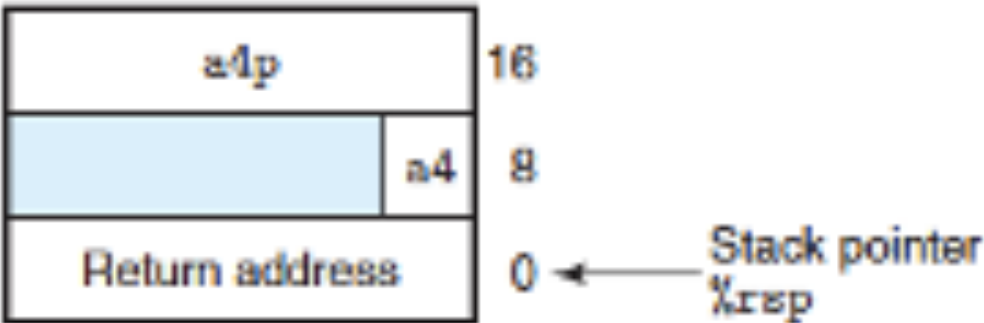


Figure 3.29 Example of function with multiple arguments of different types. Arguments 1–6 are passed in registers, while arguments 7–8 are passed on the stack.

3.7.4 LOCAL STORAGE ON THE STACK

- **Some time local data must be stored in memory (on the stack). Such as when:**
 - **There is not enough registers to hold all of the local data**
 - **The address operator & is applied to a local variable, and hence we must be able to generate an address for it.**
 - **Some of the local variables are arrays or structures and hence must be accessed by a reference**
- **These are stored in the section of the stack called “local variable”.**
- **Let’s look at an example.**

(a) Code for swap_add and calling function

```
long swap_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}

long caller()
{
    long arg1 = 534;
    long arg2 = 1057;
    long sum = swap_add(&arg1, &arg2);
    long diff = arg1 - arg2;
    return sum * diff;
}
```

Line 1 indicates this is a call to the function caller(). Lines 2-6 sets up the memory for args 1 and 2 and passes them to the swap_add function on line 7. Swap_add does it's stuff then returns and completes the remainder of the caller program.

The code for caller() is placed on a stack frame because there are addresses used in the function. We can also infer this because of the offset on line 5.

(b) Generated assembly code for calling function

```
long caller()
1 caller:
2     subq    $16, %rsp           Allocate 16 bytes for stack frame
3     movq    $534, (%rsp)        Store 534 in arg1
4     movq    $1057, 8(%rsp)      Store 1057 in arg2
5     leaq    8(%rsp), %rsi       Compute &arg2 as second argument
6     movq    %rsp, %rdi          Compute &arg1 as first argument
7     call    swap_add            Call swap_add(&arg1, &arg2)
8     movq    (%rsp), %rdx        Get arg1
9     subq    8(%rsp), %rdx       Compute diff = arg1 - arg2
10    imulq   %rdx, %rax           Compute sum * diff
11    addq    $16, %rsp           Deallocate stack frame
12    ret                                Return
```

Figure 3.31 Example of procedure definition and call. The calling code must allocate a stack frame due to the presence of address operators.

3.7.5 LOCAL STORAGE ON THE REGISTERS

- **Registers are basically a shared resource - shared by all procedures**
- **Although, only one procedure can be active at a time, care must be taken to ensure that the value of arguments or other registers being used by P are not changed**
- **By convention registers %rbx, %rbp, and %r12 - %r15 are classified as “callee saved registers”.**
- **Ex. If P calls Q, Q must preserve the values of these registers, ensuring that they have the same values when Q returns to P as they had when P called Q.**
- **Q can preserve the register values by either not changing it at all or by pushing the original value on the stack, altering it, and then popping the old value from the stack before returning. This has the effect of pushing value on a portion of the stack labeled “Saved Register”**
- **With this convention P’s code can be safely stored on the stack in callee saved registers with no worries as to what Q does with the values**

3.7.5 LOCAL STORAGE ON THE REGISTERS

- **With the exception of the stack pointer `%rsp`, all other registers are reserved for “caller-saved” registers.**
- **This means they can be modified by any function.**
- **The name “caller saved” can be understood in the context of a procedure `P` having some local data in such a register and calling procedure `Q`. Since `Q` is free to alter this register it is up to `P` (the caller) to first save the data before it makes the call.**
- **Lets see an example**

3.7.5 LOCAL STORAGE ON THE REGISTERS

(a) Calling function

```
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

(b) Generated assembly code for the calling function

```
long P(long x, long y)
x in %rdi, y in %rsi
1  P:
2  pushq   %rbp           Save %rbp   Callee saved registers
3  pushq   %rbx           Save %rbx
4  subq    $8, %rsp       Align stack frame
5  movq    %rdi, %rbp     Save x       Copies x in %rdi it is a parameter
6  movq    %rsi, %rdi     Move y to first argument
7  call    Q              Call Q(y)
8  movq    %rax, %rbx     Save result  Copies the result of the call to return
9  movq    %rbp, %rdi     Move x to first argument
10 call    Q              Call Q(x)
11 addq    %rbx, %rax     Add saved Q(y) to Q(x)
12 addq    $8, %rsp       Deallocate last part of stack
13 popq    %rbx           Restore %rbx
14 popq    %rbp           Restore %rbp
15 ret
```

Figure 3.34 Code demonstrating use of callee-saved registers. Value *x* must be preserved during the first call, and value *Q(y)* must be preserved during the second.

2 calls to Q

The first call must retain the value of *x*

The second call must retain the value computed and returned from *Q(y)*

13 - 14 restores the values of the two callee-saved registers by popping

off the stack (popped in reverse order of them being pushed — LIFO)