3.6.5 Implementing Conditional Branches with Conditional Control

The most general way to translate conditional expressions and statements from C into machine code is to use combinations of combinations of conditional and unconditional jumps. (As an alternative, we will see in Section 3.6.6 that some conditionals can be implemented by conditional transfers of data rather than control.) For example, Figure 3.16(a) shows the C code for a function that computes the absolute value of the difference of two numbers. The Function also has a side effect of incrementing one of two counters, encoded as global variables lt\_cnt and ge\_cnt. Gcc generates the assembly code shown as Figure 3.16(c). Our rendition of the machine code into C is shown as the function gotodiff\_se (Figure 3.16(b)). It uses the goto statement in C, which is similar to the unconditional jump of assembly code. Using goto statements is generally considered a bad programming style, since their use can make code very difficult to read and debug. We use them in our presentation as a way to construct C programs that describe the control flow of machine code. We call this style of programming “goto code”

In the goto code (Figure 3.16(b)), the statement goto x\_ge\_y on line 5 causes a jump to the label x\_ge\_y(since it occurs when x >= y) on line9. Continuing the execution from this point, it completes the computations specified by the else portion of function absdiff\_se and returns. On the other hand, if the test x >= y fails, the program procedure will carry out the steps specified by the if portion of absdiff\_se and return.

The assembly-code implementation first compares the two operands, setting the condition codes. If the comparison result indicates that x is greater than or equal to y, it then jumps to a block of code starting at line 8 that increments global variable ge\_cnt, computes x-y as the return value, and returns. Otherwise, it continues with the execution of code beginning at line 4 that increments global variable lt\_cnt, computes y-x as the return value, and returns. We can see, then, that the control flow of the assembly code generated for absdiff\_se closely follows the goto code of gotodiff\_se.

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

If (test-expr)

Then-statement

Else

Else-statement

Where test-expr is an integer expression that evaluates either to zero (interpreted as meaning “false”) or to a nonzero value (interpreted as meaning “true”). Only one of the two branch statements is executed.

For this general form, the assembly implementation typically adheres to the following form, where we use C syntax to describe the control flow:

t = test-expr;

if (!t)

goto false;

then-statement

goto done;

false:

else-statement

done;

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

3.6.6 Implementing Conditional Branches with Conditional Moves

The conventional way to implement conditional operations is through a conditional transfer of control, where the program follows one execution path when a condition holds and another when it does not. This mechanism is simple and general, but it can be very inefficient on modern processors.

An alternate strategy is through a conditional transfer of data. This approach computes both outcomes of a conditional operation and then selects one based on whether or not the condition holds. This strategy makes sense only in restricted cases, but it can then be implemented by a simple conditional move instruction that is better matched to the performance characteristics of modern processors. Here, we examine this strategy and its implementation with x86-64.

Figure 3.17(a) shows an example of code that can be compiled using a conditional move. The function computes the absolute value of its arguments x and y, as did our earlier example. Whereas the earlier example had side effects in the branches, modifying the value of either lt\_cnt or ge\_cnt, this version simple computes the value to be returned by the function.

To understand why code based on conditional data transfers can outperform code based on conditional control transfers (as in Figure3.16), we must understand something about how modern processors operate. As we will see in Chapters 4 and 5, processors achieve high performance through pipelining, where an instruction is processed via a sequence of stages, each performing one small portion of the required operations (e.g., fetching the instruction from memory, determining the instruction type, reading from memory, performing an arithmetic operation, writing to memory, and updating the program counter). This approach achieves high performance by overlapping the steps of the successive instructions, such as fetching one instruction while performing the arithmetic operations for a previous instruction. To do this requires being able to determine the sequence of instructions to be executed. When the machine encounters a conditional jump(referred to as a “branch”), it cannot determine which way the branch will go until it has evaluated the branch condition. Processors employ sophisticated branch prediction logic to try to guess whether or not each jump instruction will be followed. As long as it can guess reliably(modern microprocessor designs try to achieve success rates on the order of 90%), the instruction pipline will be kept full of instructions. Mispredicting a jump, on the other hand, requires that the processor discard much of the work it has already done on future instructions and then begin filling the pipeline with instructions starting at the correct location. As we will see, such a misprediction can incur a serious penalty, say, 15-30 clock cycles of wasted effort, causing a serious degradation of program performance.

As an example, we ran timings of the absdiff function on an Intel Haswell processor using both methods of implementing the conditional operation. 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. For X86-64 code with conditional jumps, we found that the function requires around 8 clock cycles per call when the branching pattern is easily predictable, and around 17.50 clock cycles per call when the branching pattern is random. From this, we can infer that the branch misprediction penalty is around 19 clock cycles. That means time required by the function ranges between around 8 and 27 cycles, depending on whether or not the branch is predicted correctly.

On the other hand, the code compiled using conditional moves requires around 8 clock cycles regardless of the data being tested. The flow of control does not depend on data, and this makes it easier for the processor to keep its pipeline full.

Figure3.18 illustrates some of the conditional move instructions available with x86-64. Each of these instructions has two operands: a source register or memory location S, and a destination register R. As with the different SET and jump instructions, the outcome of these instructions depends on the values of the condition codes. The source value is read from either memory or the source register, but it copied to the destination only if the specified condition holds.

The source and destination values can be 16, 32, or 64 bits long. Single condition move is not supported. Unlike the unconditional instructions, where the operand length is explicitly encoded in the instruction name, the assembler can infer the operand length of a conditional move instruction from the name of the destination register, and so the same instruction name can be used for all operand lengths.

**Unlike conditional jumps, the processor can execute conditional move instructions without having to predict the outcome of the test.** The processor simply reads the source value(possibly from memory), checks the condition code, and then either updates the destination register or keeps it the same.

To understand how conditional operations can be implemented via conditional data transfers, consider the following general form of conditional expression and assignment:

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.

V = test-expr ? then-expr : else-expr;

The standard way to compile this expression using conditional control transfer would have the following form:

if (!test-expr)

goto false;

v = then – expr;

goto done;

false:

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.

For the code based on a conditional move, 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:

v = then-expr;

ve = else-expr;

t = test-expr;

if (!t) v = ve;

The final statement in this sequence is implemented with a conditional move—value ve is copied to v only if test condition t does not hold.

Not all conditional expressions can be compiled using conditional moves. Most significantly, the abstract code we have shown 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. Such is the case for our earlier example (Figure 3.16). Indeed, **we put the side effects into this example specifically to force GCC to implement this function using conditional transfers.**

As a second illustration, consider the following C function:

long created(long \*xp) {

return (xp ? \*xp :0);

}

At first, this seems like a good candidate to compile using conditional move to set the result to zero when the pointer is null, as shown in the following assembly code:

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

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. Compilers must take into account the relative performance of wasted computation versus the potential for performance penalty due to branch misprediction. In truth, they do not really know how well the branches will follow predictable patterns. Our experiments with GCC indicate that it only uses conditional moves when the two expressions can be computed very easily, for example, with single add instructions. In our experience, GCC uses conditional control transfers even in many cases where the cost of branch misprediction would exceed even more complex computations.

Overall, then, we see that conditional data transfer offer an alternative strategy to conditional control transfers for implementing conditional operations. They can only be used in restricted cases, but these cases are fairly common and provide a much better match to the operation of modern processors.

3.6.7 Loops

C provides several looping constructs-namely, do-while, while and for. No corresponding instructions exist in machine code. Instead, combinations of conditional tests and jumps are used to implement the effect of loops. Gcc and other compilers generate loop code based on the two basic loop patterns. We will study the translation of loops as a progression, starting with do-while and then working toward ones with more complex implementations, covering both patterns.

Do-While Loops

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

do

body-statement

while (test-expr);

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

This general form can be translated into conditionals and goto statements as follows:

loop:

body-statement

t = test-expr;

if (t)

goto loop;

That is, on each iteration the program evaluates the body statement and then the test expression. If the test succeeds, the program goes back for another iteration.

Aside Reverse engineering loops

A key to understanding how the generated assembly code relates to the original source code is to find a mapping between program values and registers. This task was simple enough for the loop of Figure3.19, but it can be much more challenging for more complex programs. The C compiler will often rearrange the computations, so that some variables in the C code have no counterpart in the machine code, and new values are introduced into the machine code that do not exist in the source code. Moreover, it will often try to minimize register usage by mapping multiple program values onto a single register.

(The process we described for fact\_do works as a general strategy for reverse engineering loops. Look at how registers are initialized before the loop, updated and tested within the loop, and used after the loop. Each of these provides a clue that can be combined to solve a puzzle. Be prepared for surprising transformations, some of which are clearly cases where the compiler was able to optimize the code, and others where it is hard to explain why the compiler chose that particular strategy.)

The conditional jump instruction jg is the key instruction in implementing a loop. It determines whether to continue iterating or to exit the loop.

Reverse engineering assembly code, such as that of Figure 3.19(c), requires determining which registers are used for which program values. In this case, the mapping is fairly simple to determine: We know that n will be passed to the function in register %rdi. We can see register %rax getting initialized to 1. We can see that this register is also updated by multiplication on line4. Furthermore, since %rax is used to return the function value, it is often chosen to hold program values that are returned. We therefore conclude that %rax corresponds to program value result.

While Loops

The general form of a while statement is as follows:

While (test-expr)

Body-statement

It differs from do-while in that test-expr is evaluated and the loop is potentially terminated before the first execution of body-statement. There are a number of ways to translate a while loop into machine code, two of which are used in code generated by GCC. Both use the same loop structure as we saw for do-while loops but differ in how to implement the initial test.

The first translation method, which we refer to as jump to middle, performs the initial test by performing an unconditional jump to the test at the end of the loop. It can be expressed by the following template for translating from the general while loop from to goto code:

goto test:

loop:

body-statement

test:

t = test-expr;

if (t)

goto loop;

The second translation method, which we refer to as guarded do, first transforms the code into a do-while loop by using a conditional branch to skip over the loop if the initial test fails. **GCC follows this strategy when compiling with higher levels of optimization**, for example, with command-line option -O1. This method can be expressed by the following template for translating from the general while loop form to a do-while loop:

t = test-expr

if (!t)

goto done;

do

body-statement

done:

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

t = test-expr;

if(!t)

goto done;

loop:

body-statement

t = test-expr;

if (t)

goto loop;

done:

Using this implementation strategy, the compiler can often optimize the initial test, for example, determining that the test condition will always hold.

3.6.8 Switch Statements

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

Figure 3.23 shows the assembly code generated when compiling switch\_eg. The behavior of this code is shown in C as the procedure switch\_eg\_impl in Figure 3.22(b). This code makes use of support provided by GCC for jump tables, as an extension to the C language. The array jt contains seven entries, each of which is the address of a block of code. These locations are defined by labels in the code and indicated in the entries in jt by code pointers, consisting of the labels prefixed by &&. (Recall that the operator ‘&’ creates a pointer for a data value. In making this extension, the authors of GCC created a new operator && to create a pointer for a code location.) we recommend that you study the C procedure switch\_eg\_impl and how it relates to the assembly-code version.

Our original C code has cases for values 100, 102-104, and 106, but the switch variable n can be an arbitrary integer. The compiler first shifts the range to between 0 and 6 by subtracting 100 from n, creating a new program variable that we call index as an unsigned value, making use of the fact that negative numbers in a two’s-complement representation map to large positive numbers in an unsigned representation. It can therefore test whether index is outside of the rang 0-6 by testing whether it is greater than 6. In the C and assembly code, there are five distinct locations to jump to, based on the value of index. There are loc\_A(.L3), loc\_B, loc\_C, loc\_D, loc\_def, where the latter is the destination for the default case. Each of these labels identifies a block of code implementing one of the case branches. In both the C and assembly code, the program compare index to 6 and jumps to the code for the default case if it is greater.

The key step in executing a switch statement is to access a code location through the jump table. This occurs in line 16 in the C code, with a goto statement that references the jump table jt. This computed goto is supported by GCC as an extension to the C language. In our assembly-code version, a similar operation occurs on line 5, where the jmp instruction’s operand is prefixed with ‘\*’, indicating an indirect jump, and the operand specifies a memory location indexed by register %eax, which holds the value of index. (We will see in Section 3.8 how array references are translated into machine code.)

Our C code declares the jump table as an array of seven elements, each of which is a pointer to a code location. These elements span values 0-6 of index, corresponding to values 100-106 of n. Observe that the jump table handles duplicate cases by simply having the same code label(loc\_D) for entries4 and 6, and it handles missing cases by using the label case loc\_def as entries 1 and 5.

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

.section .rodata

.align 8

.L4:

.quad .L3

.quad .L8

.quad .L5

.quad .L6

.quad .L7

.quad .L8

.quad .L7

These declarations state that within the segment of the object-code file called .rodata(for “read-only data”), there should be a sequence of seven “quad” (8-byte) words, where the value of each word is given by the instruction address associated with the indicated assembly-code labels. Label .L4 marks the start of this allocation. The address associated with this label server as the base for the indirect jump.

The different code blocks implement the different branches of the switch statement. Most of them simply compute a value for val and then go to the end of the function. Similarly, the assembly-code blocks compute a value for register %rdi and jump to the position indicated by label .L2 at the end of the function. Only the code for case label 102 does not follow this pattern, to account for the way the code for this case falls through to the block with label 103 in the original C code. This is handled in the assembly-code block starting with label .L5, by omitting the jump instruction at the end of the block, so that the code continues execution of the next block. Similarly, the C version switch\_eg\_impl has no goto statement at the end of the block starting with label loc\_B.

Examining all of this code requires careful study, but the key point is to see that the use of a jump table allows a very efficient way to implement a multi-way branch. In our case, the program could branch to five distinct locations with a single jump table reference. Even if we had a switch statement with hundreds of cases, they could be handles by a single jump table access.

3.7 Procedures

Procedures are a key abstraction in software. They provide a way to package code that implements some functionality with a designated set of arguments and an optional return value. This function can then be invoked from different points in a program. Well-designed software uses procedures as an abstraction mechanism, hiding the detailed implementation of some action while providing a clear and concise interface definition of what values will be computed and what effects the procedure will have on the program state. Procedures come in many guises in different programming languages--functions, methods, subroutines, handlers, and so on -- but they all share a general set of features.

There are many different attributes that must be handled when **providing machine-level support for procedures.** For discussion purposes, suppose procedure P calls procedure Q, and Q then executes and returns back to P. These actions involve one or more of the following mechanisms:

**Passing control.** The 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 to P.

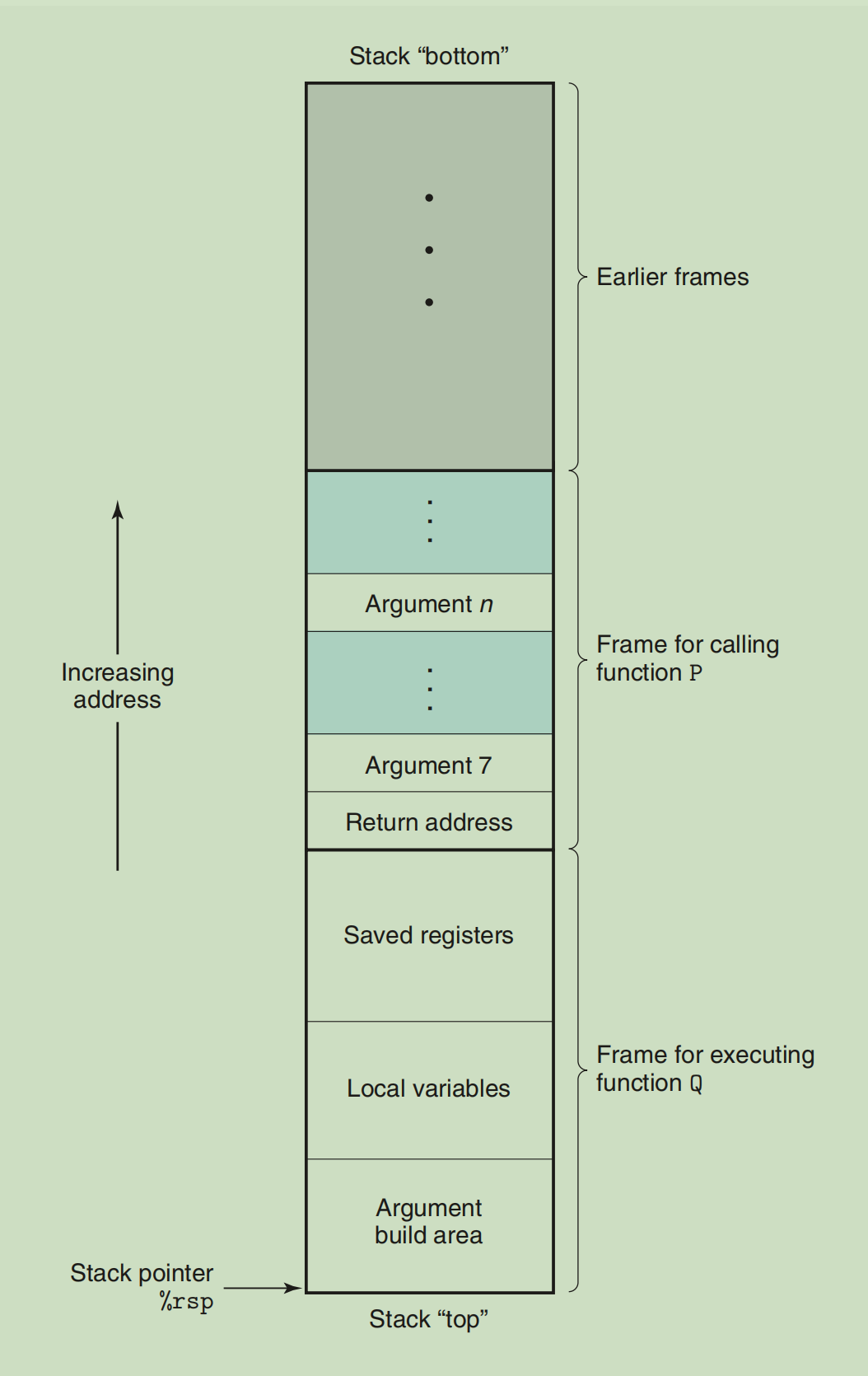
**Allocating and deallocating memory.** Q may need to allocate space for local variables when it begins and then free that storage before it returns

The 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. Great effort has been made to minimize the overhead involved in invoking a procedure. As a consequence, it follows what can be seen as a minimalist strategy, implementing only as much of the above set of mechanisms as is required for each particular procedure. In our presentation, we build up the different mechanisms step by step, first **describing control**, then **data passing**, and, finally, **memory management**.

3.7.1 The Run-Time Stack

A key feature of the procedure-calling mechanism of C, and of most other languages, is that it can make use of the last-in, first-out memory management discipline provided by a stack data structure. Using our example of procedure P calling procedure Q, we can see that while Q is executing, P, along with any of the procedures in the chain of calls up to P, is temporarily suspended. While Q is running, only it will need the ability to allocate new storage for its local variables or to set up a call to another procedure. On the other hand, when Q returns, any local storage it has allocated can be freed. Therefore, a program can manage the storage required by its procedure using a stack, where the stack and the program registers store the information required for passing control and data, and for allocation memory. As P calls Q, control and data information are added to the end of the stack. This information gets deallocated when P returns.

As described in Section3.4.4, **the X86-64 stack grows toward lower address** and **the stack pointer %rsp points to the top element of the stack**. Data can be stored on and retrieved from the stack using the pushq and popq instructions. Space for data with no specified initial value can be allocated on the stack by simply decrementing the stack pointer by an appropriate amount. Similarly, space can be deallocated be incrementing the stack pointer.



When an x86-64 procedure requires storage beyond what it can hold in registers, it allocates space on the stack. This region is referred to as the procedure’s space for local variables, and set up arguments for the procedures it calls. The stack frames for most procedures are of fixed size, allocated at the beginning of the procedure. Some procedures, however, require variable-size frames. This issue is discussed in Section 3.10.5. **Procedure P can pass up to six integral values**(i.e., pointers and integers) on the stack, but if Q requires more arguments, t**hese can be stored by P within its stack frame prior to the call.**

In the interest of space and time efficiency, x86-64 procedures allocate only the portions of stack frames they require. For example, many procedures have six or fewer arguments, and so all of their parameters can be passed in registers. Thus, parts of the stack frame diagrammed in Figure 3.25 may be omitted. Indeed, many functions do not even require a stack frame. **This occurs when 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). For example, **none of the functions we have examined thus far required stack frames.**

3.7.2 Control Transfer

Passing control from function P to function Q involves simply setting the program counter(PC) to the starting address of the code for Q. However, when it later comes time for Q to return, the processor must have some record of the code location where it should resume the execution of P. This information is recorded in X86-64 machines by invoking procedure Q with the instruction **call Q**. This instruction pushes an address A onto the stack and sets the PC to the beginning of Q. The pushed address A is referred to as the return address and is computed as the address of the instruction immediately following the call instruction. The counterpart instruction ret pops an address A off the stack and sets the PC to A.

The general forms of the call and ret instructions are described as follows:

call Label //Procedure call

call \*Operand //Procedure call

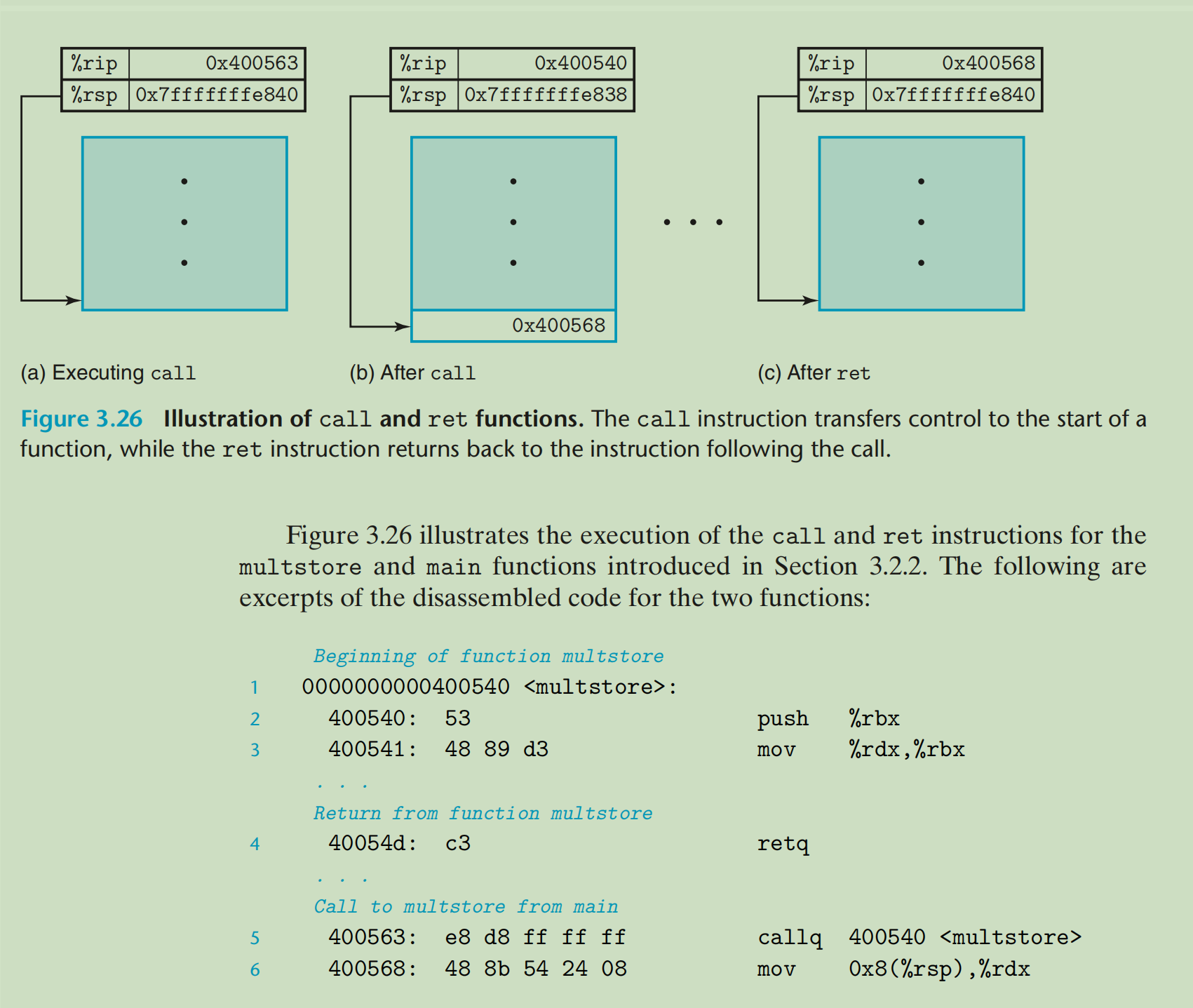
Ret //Return from call

(These instructions are referred to as callq and retq in the disassembly outputs generated by the program OBJUMP. The added suffix ‘q’ simply emphasizes that these are x86-64 versions of call an return instructions, not IA32.)

The call instruction has a target indication the address of the instruction where the called procedure starts. Like jumps, a call can be either direct or indirect. In assembly code, the target of a direct call is given as a label, while the target of an indirect call is given by ‘\*’ followed by an operand specifier using one of the formats described in Figure3.3.

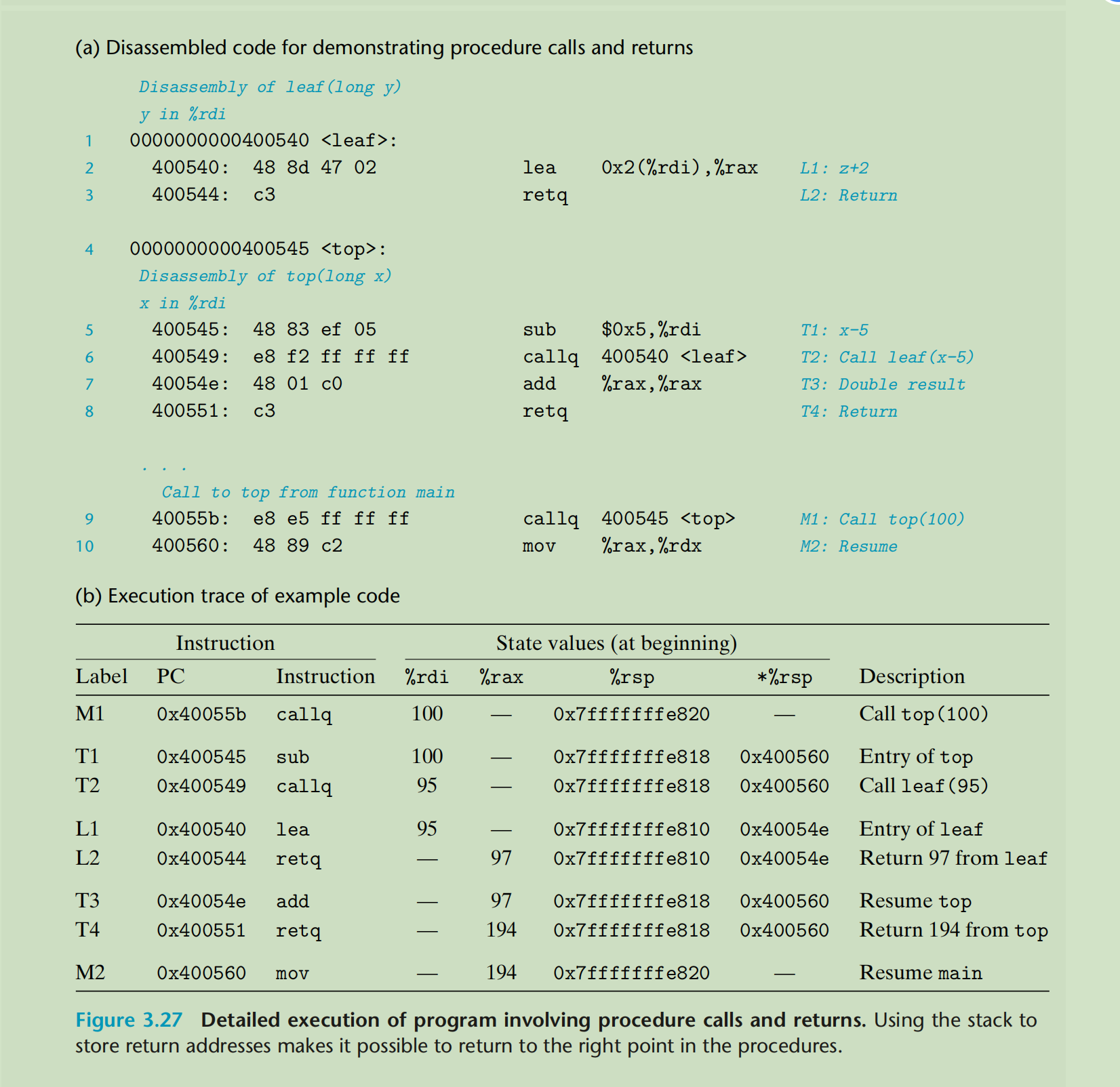
%rip will always point the current running instruction address

%rsp will always point to the stack top(the lowest addr)



In this code, we can see that the call instruction with address 0x400563 in main calls function multstore. This status is shown in Figure 3.26(a), with the indicated values for the stack pointer %rsp and the program counter %rip. The effect of the call is to push the return address 0x400568 onto the stack and to jump to the first instruction in function multstore, at address 0x400540. The execution of function multstore continues until it hits the ret instruction at address 0x40054d. This instruction pops the value 0x400568 from the stack and jumps to this address, resuming the execution of main just after the call instruction.

//note that the stack address use 16, so for the pic(a) and pic(b) 相差8个字节，正好用来存0x0000000400568这个64地址。内存中每个地址都对应一个字节



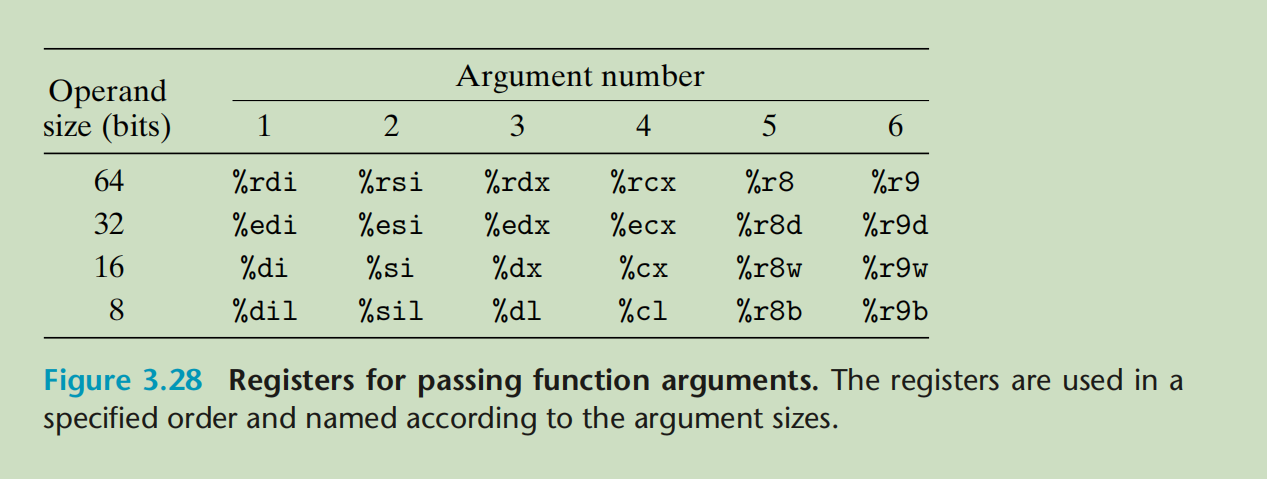
As a more detailed example of passing control to and from procedures, Figure3.27(a) shows the disassembled code for two functions, top and leaf, as well as the portion of code in function main where top gets called. Each instruction is identified by labels L1-L2(in leaf), T1-T4(in top), and M1-M2 in main. Part(b) of the figure shows a detailed trace of the code execution, in which main calls top(100), causing top to call leaf(95). Function leaf returns 97 to top, which then returns 194 to main. The first three columns describe the instruction being executed, including the instruction label, the address and the instruction type. The next four columns show the state of the program **before** the instruction is executed, including the contents of registers %rdi, %rax, and %rsp, as well as the value at the top of the stack. The contents of this table should be studied carefully, as they demonstrate the important role of the run-time stack in managing the storage needed to support procedure calls and returns.

We can see that this simple mechanism of pushing the return address onto the stack makes it possible for the function to later return to the proper point in the program. The standard call/return mechanism of C(and of most programming languages) conveniently matches the last-in, first-out memory management discipline provided by a stack.

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 data as arguments, and returning from a procedure may also involve returning a value. With x86-64, most of the data passing to and from procedures take place via registers. For example, we have already seen numerous examples of functions where arguments are passed in registers %rdi, %rsi, and others, and where values are returned in register %rax. When procedure P calls procedure Q, the code for P must first copy the arguments into the proper registers. Similarly, when Q returns back to P, the code for P can access the returned value in register %rax. In this section, we explore these conventions in grated detail.

With x86-64, up to six integral(i.e., integer and pointer) arguments can be passed via registers. The registers are used in a specified order, with the name used for a register depending on the size of the data type being passed. These are shown in Figure 3.28. Arguments are allocated to these registers according to their ordering in argument 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.



When a function has more than six integral arguments, the other ones are passed on the stack. Assume that procedure P calls procedure Q with n integral arguments, such that n > 6. Then the code for P must allocate a stack frame with enough storage for arguments 7 through n, as illustrated in Figure 3.25. **It copies arguments 1-6 into the appropriate registers, and it puts arguments 7 through n onto the stack**, with argument 7 at the top of the stack. When passing parameters on the stack, all data sizes are rounded up to be multiples of eight. With the arguments in place, the program can then execute a call instruction to transfer control to procedure Q. Procedure Q can access its arguments via registers and possibly from the stack. If Q, in turn , calls some function that has more than six arguments, it can allocate space withi**n its stack frame** for these.

As an example of argument passing, consider the C function proc shown in Figure 3.29(a). This function has eight arguments, including integers with different numbers of bytes(8,4,2,and 1), as well as different types of pointers, each of which is 8 bytes.

The assembly code generated for proc is shwon in Figure 3.29(b). The first six arguments are passed in registers.