

Bits, Flags, Branches, and Tables

Easing Into Mainstream Assembly Coding

As you've seen by now, my general method for explaining things starts with the "view from a height" and then moves down toward the details. That's how I do things because that's how people learn: by plugging individual facts into a larger framework that makes it clear how those facts relate to one another. It's possible (barely) to move from details to the big picture, but across 60-odd years of banging my head against various subjects in the pursuit of knowledge, it's become very clear that having the overall framework in place first makes it a *lot* easier to establish all those connections between facts. It's like carefully placing stones into a neat pile before shoveling them into a box. If the goal is to get the stones into a box, it's much better to have the box in place before starting to pick up the stones.

And so it is here. The big picture is mostly in place. From now on in this book, we'll be looking at the details of assembly code and seeing how they fit into that larger view.

Bits Is Bits (and Bytes Is Bits)

Assembly language is big on bits. Bits, after all, are what bytes are made of, and one essential assembly language skill is building bytes and taking them apart again. A technique called bit mapping is widely used in assembly language. *Bit mapping* assigns special meanings to individual bits within a byte to save space and squeeze the last little bit of utility out of a given amount of memory.

There is a family of instructions in the x64 instruction set that allows you to manipulate the bits within the bytes by applying Boolean logical operations between bytes on a bit-by-bit basis. These are the *bitwise logical instructions*: AND, OR, XOR, and NOT. Another family of instructions allows you to slide bits back and forth within a single byte or word. These are the most-used shift/rotate instructions: ROL, ROR, RCL, RCR, SHL, and SHR. (There are a few others that I will not be discussing in this book.)

Bit Numbering

Dealing with bits requires that we have a way of specifying which bits we’re dealing with. By convention, bits in assembly language are numbered, starting from 0, at the *least-significant bit* in the byte, word, double word, or other item we’re using as a bit map. The *least-significant bit* is the one with the least value in the binary number system. It’s also the bit on the far right, if you write the value down as a binary number in the conventional manner.

I’ve shown this in Figure 9.1, for a 16-bit word. Bit numbering works exactly the same way no matter how many bits you’re dealing with: bytes, words, double words, or quadwords. Bit 0 is *always* on the right-hand end, and the bit numbers increase toward the left.

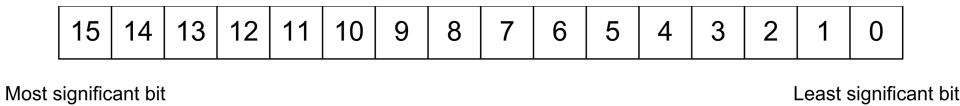


Figure 9.1: Bit numbering

When you count bits, start with the bit on the right-hand end, and number them leftward from 0.

“It’s the Logical Thing to Do, Jim. . .”

The term *Boolean logic* sounds arcane and forbidding, but remarkably, it reflects the realities of ordinary thought and action. The Boolean operator AND, for instance, pops up in many of the decisions you make every day of your life. For example, to write a check that doesn’t bounce, you must have money in your checking account AND checks in your checkbook. Neither alone will do the job. You can’t write a check that you don’t have, and a check without money behind it will bounce. People who live out of their checkbooks use the AND operator frequently.

When mathematicians speak of Boolean logic, they manipulate abstract values called True and False. The AND operator works like this: Condition1

AND Condition2 will be considered True if *both* Condition1 and Condition2 are True. If either condition is False, the result will be False.

There are in fact four different combinations of the two input values, so logical operations between two values are usually summarized in a form called a *truth table*. The truth table for the logical operator AND (*not* the AND instruction yet; we'll get to that shortly) is shown in Table 9.1.

Table 9.1: The AND Truth Table for Formal Logic

CONDITION1	OPERATOR	CONDITION2	RESULT
False	AND	False =	False
False	AND	True =	False
True	AND	False =	False
True	AND	True =	True

There's nothing mysterious about the truth table. It's just a summary of all possibilities of the AND operator as applied to two input conditions. The important thing to remember about AND is that *only* when both input values are True will the result also be True.

That's the way mathematicians see AND. In assembly language terms, the AND instruction looks at two bits and yields a third bit based on the values of the first two bits. By convention, we consider a 1 bit to be True and a 0 bit to be False. The *logic* is identical; we're just using different symbols to represent True and False. Keeping that in mind, we can rewrite AND's truth table to make it more meaningful for assembly language work. See Table 9.2.

Table 9.2: The AND Truth Table for Assembly Language

BIT 1	OPERATOR	BIT 2	RESULT BIT
0	AND	0 =	0
0	AND	1 =	0
1	AND	0 =	0
1	AND	1 =	1

The AND Instruction

The AND instruction embodies this concept in the x64 instruction set. The AND instruction performs the AND logical operation on two like-sized operands and replaces the destination operand with the result of the operation as a whole.

(Remember that the destination operand is the operand closest to the mnemonic.) In other words, consider this instruction:

```
and al,bl
```

What will happen here is that the CPU will perform a gang of eight bitwise AND operations on the eight bits in AL and BL. Bit 0 of AL is **ANDed** with bit 0 of BL, bit 1 of AL is **ANDed** with bit 1 of BL, and so on. Each AND operation generates a result bit, and that bit is placed in the destination operand (here, AL) *after* all eight AND operations occur. This is a common thread among machine instructions that perform some operation on two operands and produce a result: The result replaces the first operand (the destination operand) and *not* the second!

Masking Out Bits

A major use of the **AND** instruction is to isolate one or more bits out of a byte, word, dword, or qword value. *Isolate* here simply means to set all *unwanted* bits to a reliable 0 value. As an example, suppose we are interested in testing bits 4 and 5 of a value to see what those bits are. To do that, we have to be able to ignore the other bits (bits 0 through 3 and 6 through 7), and the only way to safely ignore bits is to set them to 0.

AND is the way to go. We set up a *bit mask* in which the bit numbers that we want to inspect and test are set to 1, and the bits we wish to ignore are set to 0. To mask out all bits but bits 4 and 5, we must set up a mask in which bits 4 and 5 are set to 1, with all other bits at 0. This mask in binary is 00110000B, or 30H. (To verify it, count the bits from the right-hand end of the binary number, starting with 0.) This bit mask is then **ANDed** against the value in question. Figure 9.2 shows this operation in action, with the 30H bit mask just described and an initial value of 9DH.

The three binary values involved are shown laid out vertically, with the least-significant bit (that is, the right-hand end) of each value at the top. You should be able to trace each AND operation and verify it by looking at Table 9.2.

The end result is that all bits except bits 4 and 5 are *guaranteed* to be 0 and can thus be safely ignored. Bits 4 and 5 could be either 0 or 1. (That's why we need to test them; we don't *know* what they are.) With the initial value of 9DH, bit 4 turns out to be a 1, and bit 5 turns out to be a 0. If the initial value were something else, bits 4 and 5 could both be 0, both be 1, or some combination of the two.

Don't forget: The result of the **AND** instruction replaces the destination operand after the operation is complete.

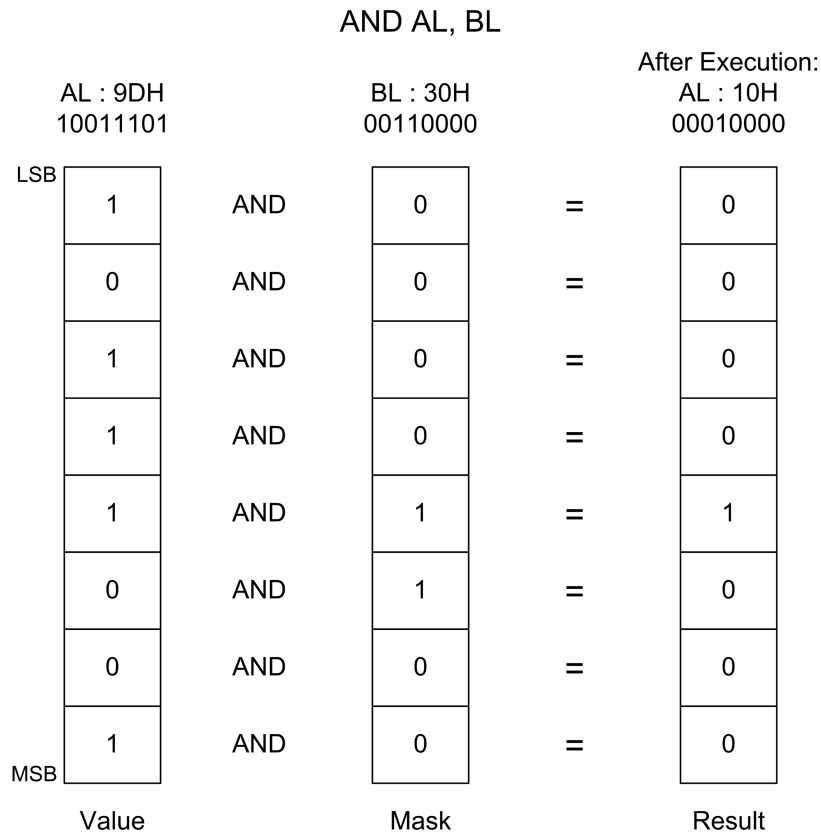


Figure 9.2: The anatomy of an AND instruction

The OR Instruction

Closely related to the AND logical operation is OR, which, like the AND logical operation, has an embodiment with the same name in the x86/x64 instruction set. Structurally, the OR instruction works identically to AND. Only its truth table is different: While AND requires that both its operands be 1 for the result to be 1, OR is satisfied that at least *one* operand has a 1 value. The truth table for OR is shown in Table 9.3.

Table 9.3: The OR Truth Table for Assembly Language

BIT 1	OPERATOR	BIT 2	RESULT BIT
0	OR	0 =	0
0	OR	1 =	1
1	OR	0 =	1
1	OR	1 =	1

Because it's unsuitable for isolating bits, the OR instruction is used much more rarely than AND.

The XOR Instruction

In a class by itself is the exclusive OR operation, embodied in the XOR instruction. XOR, again, does in broad terms what AND and OR do: It performs a bit-by-bit logical operation on its two operands, and the result replaces the destination operand. The logical operation, however, is *exclusive or*, meaning that the result is 1 only if the two operands are *different* (that is, 1 and 0 or 0 and 1). The truth table for XOR (see Table 9.4) should make this slightly slippery notion a little clearer.

Table 9.4: The XOR Truth Table for Assembly Language

BIT 1	OPERATOR	BIT 2	RESULT BIT
0	XOR	0 =	0
0	XOR	1 =	1
1	XOR	0 =	1
1	XOR	1 =	0

Look Table 9.4 over carefully! In the first and last cases, where the two operands are the *same*, the result is 0. In the middle two cases, where the two operands are *different*, the result is 1.

Some interesting things can be done with the XOR instruction, but most of them are a little arcane for a beginners' book like this. One nonobvious but handy use of XOR is this: XORing any value against *itself* yields 0. In other words, if you execute the XOR instruction with both operands as the same register, that register will be cleared to 0:

```
xor rax,rax ; Zero out the rax register
```

In the old days, this was faster than loading a 0 into a register from immediate data using MOV. Although that's no longer the case, it's an interesting trick to know. How it works should be obvious from reading the truth table, but to drive it home I've laid it out in Figure 9.3.

Follow each of the individual exclusive OR operations across the figure to its result value. Because each bit in AL is XORed against itself, in every case the XOR operations happen between two operands that are identical. Sometimes both are 1, sometimes both are 0, but in every case the two are the same. With the XOR operation, when the two operands are the same, the result is always 0. *Voilà!* Zero in a register.

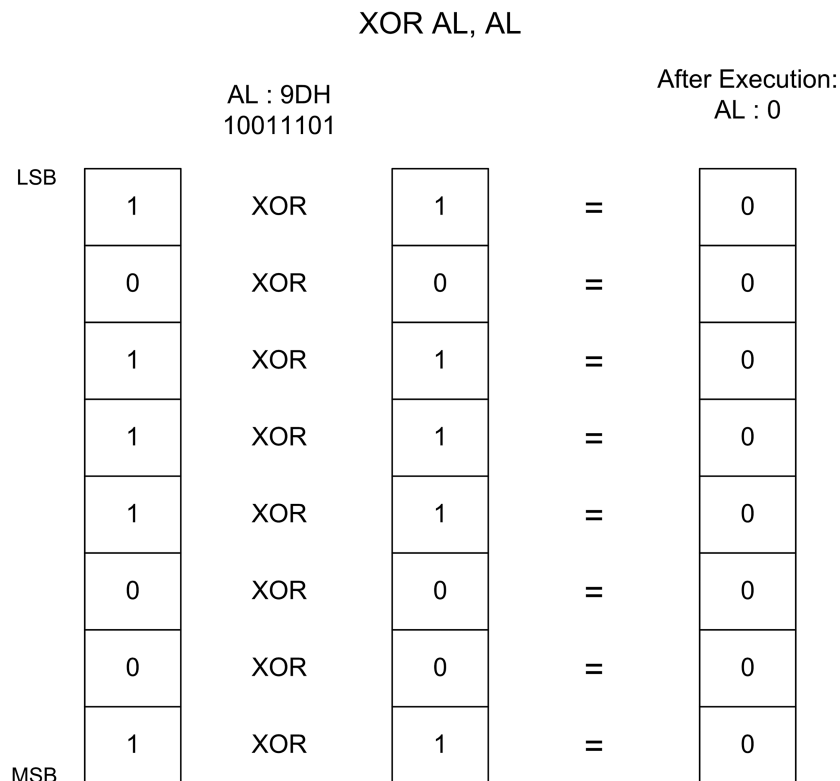


Figure 9.3: Using XOR to zero a register

The NOT Instruction

Easiest to understand of all the bitwise logical instructions is `NOT`. The truth table for `NOT` is simpler than the others we’ve looked at because `NOT` takes only one operand. And what it does is simple as well: `NOT` takes the state of each bit in its single operand and changes that bit to its opposite state. What was 1 becomes 0, and what was 0 becomes 1. I show this in Table 9.5.

Table 9.5: The NOT Truth Table for Assembly Language

BIT	OPERATOR	RESULT BIT
0	NOT	1
1	NOT	0

Segment Registers Don't Respond to Logic!

You won't be directly accessing the segment registers until you get into the depths of operating-system programming. The segment registers now belong to the OS for its own use, and user-space programs cannot change them in any way.

But even when you begin working at the operating-system level, the segment registers come with significant limitations. One such limitation is that they cannot be used with any of the bitwise logic instructions. If you try, the assembler will hand you an "Illegal use of segment register" error. If you need to perform a logical operation on a segment register, you must first copy the segment register's value into one of the general-purpose registers, perform the logical operation on the GP register, and then copy the result in the GP register back into the segment register.

The general-purpose registers are called "general purpose" for a reason, and the segment registers are not in any way general-purpose. They are specialists in memory addressing, and if you ever have to work on segment values, the general approach is to do the work in a general-purpose register and then copy the modified value back into the segment register in question.

Shifting Bits

The other way of manipulating bits within a byte is a little more straightforward: You *shift* them toward one side or the other. There are a few wrinkles to the process, but the simplest shift instructions are pretty obvious: `SHL` SHifts its operand Left, whereas `SHR` SHifts its operand Right.

All of the shift instructions (including the slightly more complex ones I'll describe a little later) have the same general form, illustrated here by the `SHL` instruction:

```
shl <register/memory>,<count>
```

The first operand is the target of the shift operation, that is, the value that you're going to be shifting. It can be register data or memory data, but not immediate data. The second operand specifies the number of bits by which to shift.

Shift by What?

This `<count>` operand has a peculiar history. On the ancient 8086 and 8088, it could be one of two things: the immediate digit 1, or else the register CL. (*Not* CX!) If you specified the count as 1, then the shift would be by one bit. If you wanted to shift by more than one bit at a time, you had to first load the shift count into register CL. In the days before the x86 general-purpose registers became truly general-purpose, counting things used to be CX's (and hence CL's)

“hidden agenda.” It would count shifts, passes through loops, string elements, and a few other things. That’s why it’s sometimes called the *count register* and can be remembered by the C in *count*. Starting with the 286 and for all more recent x86/x64 CPUs, the <count> operand may be any immediate value from 0 to 255. The shift count may also be passed in CL if you prefer. Note that you cannot specify RCX for the count, even though it “contains” CL. Even in x64, the shift instructions really do require either an immediate value from 0–255 or CL. Any other register specified for the count value will trigger an assembler error.

Obviously, shifting by 0 bits is pointless, but it’s possible and not considered an error. *Watch your typing.*

Now, there’s an important asterisk to the previous paragraph: *You can’t shift more positions than the destination register has.* In 64-bit long mode, you can’t shift (or rotate; see the next section) more than 63 counts. Attempting to do so won’t trigger an error. It just won’t work. It won’t work because before the instruction is executed, the CPU masks the count value to the six lowest bits. Those low six bits can count only to 63. It takes seven bits to express 64.

A literal value of 146 won’t cause an error, but you’ll only shift the destination operand by 18 positions.

In 32-bit protected mode, the CPU masks the count value to the five lowest bits, because five bits can count to 31.

How Bit Shifting Works

Understanding the shift instructions requires that you think of the numbers being shifted as *binary* numbers, and not hexadecimal or decimal numbers. (If you’re fuzzy on binary notation, again, take another focused pass through Chapter 2.) A simple example would start with register AX containing a value of 0B76FH. (I’m using AX for the example here to keep the binary numbers short and graspable, but the shift instructions may be used on any size register.) Expressed as a binary number (and hence as a bit pattern), 0B76FH is as follows:

```
1011011101101111
```

Keep in mind that each digit in a binary number is one bit. If you execute an `SHL AX, 1` instruction, what you’d find in AX after the shift is the following:

```
0110111011011110
```

A 0 has been inserted at the right-hand end of the number, and the whole shebang has been bumped toward the left by one digit. Notice that a 1 bit has been bumped off the left end of the number into cosmic nothingness.

You can even use the shift instructions on CL, with CL containing the count. This is legal, even if it looks peculiar, and may not be the best idea:

```
mov cl,1
shl cl,cl
```

What happens in this example is that the count value in CL is shifted left by the value CL contains. Here the 1 bit in CL is shifted to become a 2 bit. If this still seems strange, put it in a sandbox and watch the registers.

Bumping Bits into the Carry Flag

Shifting a bit off the left end of a binary value doesn't *exactly* send that bit into cosmic nothingness. A bit shifted out of the left end of a binary value is bumped into a temporary bucket for bits called the Carry flag (CF). The Carry flag is one of those informational bits gathered together as the RFlags register, which I described in Chapter 7. You can test the state of the Carry flag with a branching instruction, as I'll explain a little later in this chapter.

However, keep in mind when using shift instructions that a *lot* of different instructions use the Carry flag—not only the shift instructions. If you bump a bit into the Carry flag with the intent of testing that bit later to see what it is, test it *before* you execute another instruction that affects the Carry flag. That list includes all the arithmetic instructions, all the bitwise logical instructions, a few other miscellaneous instructions—and, of course, all the other shift instructions.

If you shift a bit into the Carry flag and then immediately execute another shift instruction, the bit bumped into the Carry flag earlier *will* be bumped off the end of the world into cosmic nothingness.

The Rotate Instructions

That said, if a bit's destiny is *not* to be lost in cosmic nothingness, you need to use the rotate instructions `RCL`, `RCR`, `ROL`, and `ROR` instead. The rotate instructions are almost identical to the shift instructions, but with a crucial difference: A bit bumped off one end of the operand reappears at the opposite end of the operand. As you rotate an operand by more than one bit, the bits march steadily in one direction, falling off the end and immediately reappearing at the opposite end. The bits thus “rotate” through the operand as the rotate instruction is executed.

Like so many things, this shows better than it tells. Take a look at Figure 9.4. The example shown here is the `ROL` (Rotate Left) instruction, but the `ROR` instruction works the same way, with the bits moving in the opposite direction. An initial binary value of 10110010 (0B2h) is placed in AL. When an `ROL AL,1` instruction is executed, all the bits in AL march toward the left by one position. The 1-bit in bit 7 exits AL stage left but runs around and reappears immediately from stage right.

Again, `ROR` works exactly the same way, but the movement of bits is from left to right instead of (as with `ROL`) right to left. The number of bits by which an operand is rotated can be either an immediate value or a value in CL.

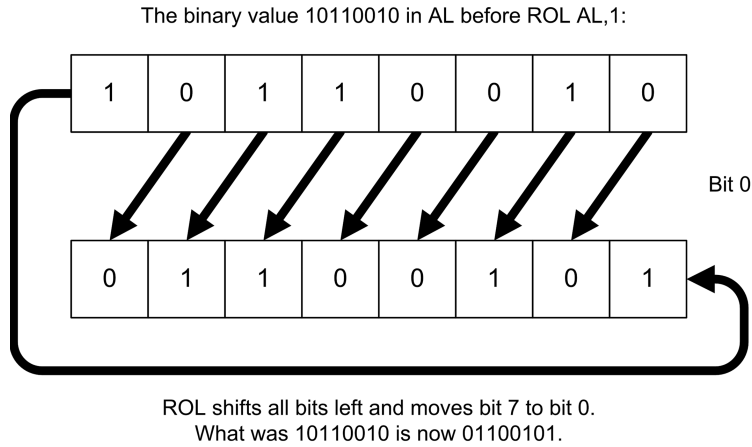


Figure 9.4: How the rotate instructions work

Rotating Bits Through the Carry Flag

There is a second pair of rotate instructions in the x86/x64 instruction set: `RCR` (Rotate Carry Right) and `RCL` (Rotate Carry Left). These operate as `ROL` and `ROR` do, but with a twist: The bits that are shifted out the end of an operand and re-enter the operand at the beginning travel by way of the Carry flag. The path that any single bit takes in a rotate through CF is thus one bit longer than it would be in `ROL` and `ROR`. I've shown this graphically in Figure 9.5.

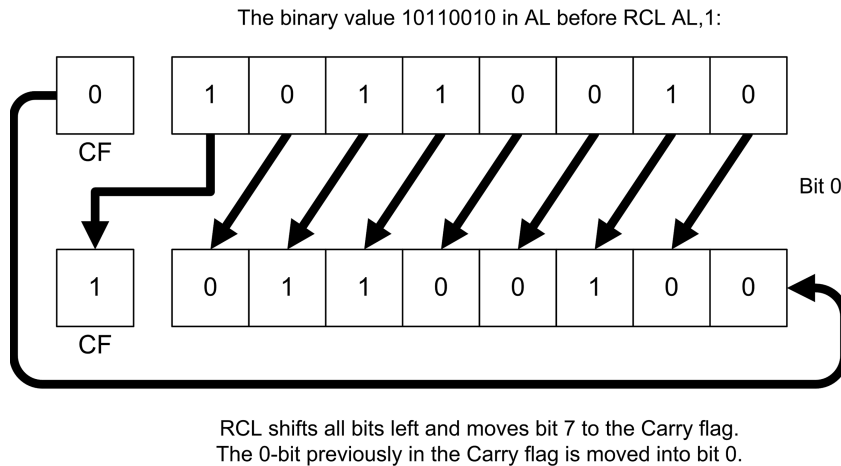


Figure 9.5: How the rotate through carry instructions work

Setting a Known Value into the Carry Flag

It's also useful to remember that previous instructions can leave values in CF, and those values will be rotated into an operand during an `RCL` or `RCR` instruction. Some people have the mistaken understanding that CF is forced to 0 before a shift or rotate instruction, and that's just not true. If another instruction leaves a 1-bit in CF immediately before an `RCR` or `RCL` instruction, that 1-bit will obediently enter the destination operand, whether you want it to or not.

If starting out a rotate with a known value in CF is important, there is a pair of x86 instructions that will do the job for you: `CLC` and `STC`. `CLC` clears the Carry flag to 0. `STC` sets the Carry flag to 1. Neither instruction takes an operand and neither has any other effects.

Bit-Bashing in Action

As we saw in earlier chapters, Linux has a fairly convenient method for displaying text to your screen. The problem is that it displays only *text*—if you want to display a numeric value from a register as a pair of hex digits, Linux won't help. You first have to convert the numeric value into its string representation and then display the string representation by calling the `sys_write` kernel service via `syscall`.

Converting hexadecimal numbers to hexadecimal digits isn't difficult, and the code that does the job demonstrates several of the new concepts we're exploring in this chapter. The code in Listing 9.1 is the bare-bones core of a hex-dump utility. When you redirect its input from a file of any kind, it will read that file 16 bytes at a time and display those 16 bytes in a line, as 16 hexadecimal values separated by spaces. The code contains a number of new techniques that are worth discussing. Then, in Chapter 10, we'll expand it to include the ASCII equivalent column to the right of the `hexdump` column.

Listing 9.1: `hexdump1gcc.asm`

```
; Executable name : hexdump1gcc
; Version          : 2.0
; Created date     : 5/9/2022
; Last update      : 5/8/2023
; Author           : Jeff Duntemann
; Description      : A simple program in assembly for Linux, using
;                   NASM 2.15 under the SASM IDE, demonstrating the
;                   conversion of binary values to hexadecimal
;                   strings. It acts as a very simple hex dump utility
;                   for files, without the ASCII equivalent column.
;
; Run it this way:
;   hexdump1gcc < (input file)
```

```

;
; Build using SASM's default build setup for x64
;
SECTION .bss                ; Section containing uninitialized data

    BUFFLEN equ 16          ; We read the file 16 bytes at a time
    Buff:    resb BUFFLEN   ; Text buffer itself, reserve 16 bytes

SECTION .data               ; Section containing initialised data

    HexStr: db " 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00",10
    HEXLEN equ $-HexStr

    Digits: db "0123456789ABCDEF"

SECTION .text               ; Section containing code

global main                 ; Linker needs this to find the entry point!

main:
    mov rbp,rsbp            ; SASM Needs this for debugging

; Read a buffer full of text from stdin:
Read:
    mov rax,0               ; Specify sys_read call 0
    mov rdi,0               ; Specify File Descriptor 0: Standard Input
    mov rsi,Buff            ; Pass offset of the buffer to read to
    mov rdx,BUFFLEN         ; Pass number of bytes to read at one pass
    syscall                 ; Call sys_read to fill the buffer
    mov r15,rax             ; Save # of bytes read from file for later
    cmp rax,0               ; If rax=0, sys_read reached EOF on stdin
    je Done                 ; Jump If Equal (to 0, from compare)

; Set up the registers for the process buffer step:parm
    mov rsi,Buff            ; Place address of file buffer into esi
    mov rdi,HexStr          ; Place address of line string into edi
    xor rcx,rcx             ; Clear line string pointer to 0

; Go through the buffer and convert binary values to hex digits:
Scan:
    xor rax,rax             ; Clear rax to 0

; Here we calculate the offset into the line string, which is rcx X 3
    mov rdx,rcx             ; Copy the pointer into line string into rdx
;    shl rdx,1               ; Multiply pointer by 2 using left shift
;    add rdx,rcx             ; Complete the multiplication X3
    lea rdx,[rdx*2+rdx]      ; This does what the above 2 lines do!
;    See discussion of LEA later in Ch. 9

```

```

; Get a character from the buffer and put it in both rax and rbx:
    mov al,byte [rsi+rcx] ; Put a byte from the input buffer into al
    mov rbx,rax          ; Duplicate byte in bl for second nybble

; Look up low nybble character and insert it into the string:
    and al,0Fh           ; Mask out all but the low nybble
    mov al,byte [Digits+rax] ; Look up the char equivalent of nybble
    mov byte [HexStr+rdx+2],al ; Write the char equivalent to
                                ; the line string

; Look up high nybble character and insert it into the string:
    shr bl,4             ; Shift high 4 bits of char into low 4 bits
    mov bl,byte [Digits+rbx] ; Look up char equivalent of nybble
    mov byte [HexStr+rdx+1],bl ; Write the char equivalent to
                                ; the line string

; Bump the buffer pointer to the next character and see if we're done:
    inc rcx              ; Increment line string pointer
    cmp rcx,r15          ; Compare to the number of characters in the buffer
    jna Scan             ; Loop back if rcx is <= number of chars in buffer

; Write the line of hexadecimal values to stdout:
    mov rax,1            ; Specify syscall call 1: sys_write
    mov rdi,1            ; Specify File Descriptor 1: Standard output
    mov rsi,HexStr       ; Pass address of line string in rsi
    mov rdx,HEXLEN       ; Pass size of the line string in rdx
    syscall              ; Make kernel call to display line string
    jmp Read             ; Loop back and load file buffer again

; All done! Let's end this party:
Done:
    ret                  ; Return to the glibc shutdown code

```

The `hexdump1` program is at its heart a filter program and has the same general filter machinery I used in the `uppercaser2` program from Chapter 8. The important parts of the program for this discussion are the parts that read 16 bytes from the input buffer and convert them to a string of characters for display to the Linux console. This is the code between the `Scan` label and the `RET` instruction. I'll be referring to that block of code in the discussion that follows.

If you read Listing 9.1, you'll see that two of the lines of code are commented out. This was not a mistake, and I'll come back to it.

Splitting a Byte into Two Nybbles

Remember that the values read by Linux from a file are read into memory as binary values. Hexadecimal is a way of displaying binary values, and for you to display binary values as displayable ASCII hexadecimal digits, you have to do some converting.

Displaying a single 8-bit binary value requires two hexadecimal digits. The bottom four bits in a byte are represented by one digit (the least-significant, or rightmost, digit), and the top four bits in the byte are represented by another digit (the most significant, or leftmost, digit). The binary value 11100110, for example, is the equivalent of E6 in hex. (I went over all this in detail in Chapter 2.) Converting an 8-bit value into two 4-bit digits must be done one digit at a time, which means that we have to separate the single byte into two 4-bit quantities, which are often called *nybbles*, especially in assembly work.

In the `hexdump1` program, a byte is read from `Buff` and is placed in two registers, RAX and RBX. This is done because separating the high from the low nybble in a byte is destructive, in that we basically zero out the nybble that we don't want.

To isolate the low nybble in a byte, we need to *mask out* the unwanted high nybble. This is done with an `AND` instruction:

```
and al,0Fh
```

The immediate constant 0Fh expressed in binary is 00001111. If you follow the operation through the AND truth table (Table 9.2) you'll see that any bit ANDed against 0 is 0. We AND the high nybble of register AL with 0000, which zeros out anything that might be there. ANDing the low nybble against 1111 leaves the bits of the low nybble precisely as they were.

When we're done, we have the low nybble of the byte read from `Buff` in AL.

Shifting the High Nybble into the Low Nybble

Masking out the high nybble from the input byte in AL destroys it. We need that high nybble, but we have a second copy in RBX, and that's the copy from which we'll extract the high nybble. As with the low nybble, we'll actually work with the least significant eight bits of RBX, as BL. Remember that BL is just a different way of referring to the lowest eight bits of RBX. It's not a different register. If a value is loaded into RBX, its least-significant eight bits are in BL.

We could mask out the low nybble in BL with an `AND` instruction, leaving behind the high nybble, but there's a catch: Masking out the low four bits of a byte does not make the high four bits a nybble. We have to somehow move the high four bits of the input byte into the low four bits.

The fastest way to do this is simply to shift BL to the right by four bits. This is what the `SHR BL, 4` instruction does. The low nybble is simply shifted off the edge of BL, into the Carry flag, and then out into cosmic nothingness. After the shift, what was the high nybble in BL is now the low nybble.

At this point, we have the low nybble of the input byte in AL, and the high nybble of the input byte in BL. The next challenge is converting the four-bit binary number in a nybble (for example, 1110) into its displayable ASCII hex digit; in this example, that's the "E" character.

Using a Lookup Table

In the `.data` section of the program is the definition of a very simple *lookup table*. The `Digits` table has this definition:

```
Digits db '0123456789ABCDEF'
```

The important thing to note about the `Digits` table is that each digit occupies a position in the string whose offset from the start of the string is the value it represents. In other words, the ASCII character 0 is at the very start of the string, offset zero bytes from the string's beginning. The character 7 lies seven bytes from the start of the string, and so on.

We “look up” a character in the `Digits` table using a memory reference:

```
mov al,byte [Digits+rax]
```

As with most of assembly language, everything here depends on memory addressing. The first hex digit character in the lookup table is at the address in `Digits`. To get at the desired digit, we must *index* into the lookup table. We do this by adding an offset into the table to the address inside the brackets. This offset is the nybble in AL.

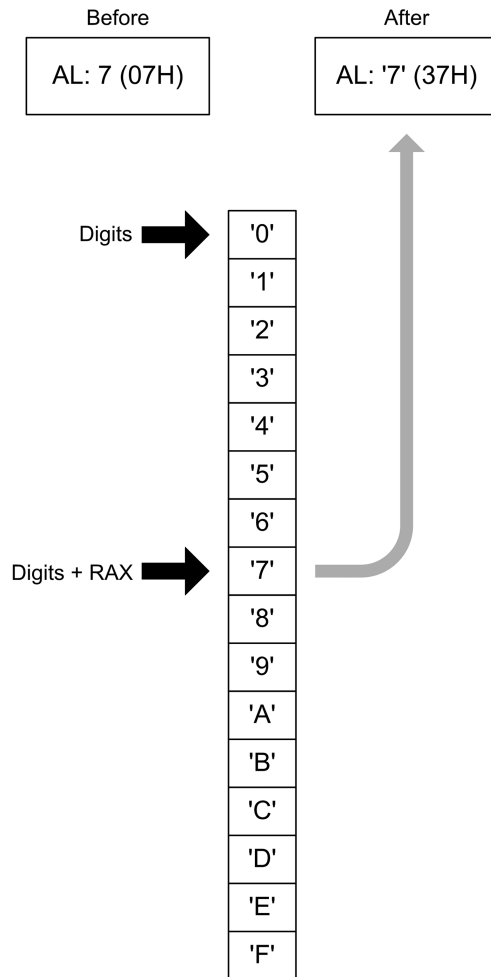
Adding the offset in AL to the address of `Digits` (using RAX) takes us right to the character that is the ASCII equivalent of the value in AL. I've drawn this out graphically in Figure 9.6.

There are two possibly confusing things about the `MOV` instruction that fetches a digit from `Digits` and places it in AL:

- We must use RAX in the memory reference rather than AL, because *AL cannot take part in effective address calculations*. Don't forget that AL is “inside” RAX! (More on effective address calculations a little later in this chapter.)
- We are replacing the nybble in AL with its character equivalent. The instruction first fetches the character equivalent of the nybble from the table and then stores the character equivalent back into AL. The nybble that had been in AL is overwritten and thus gone.

So far, we've read a character from the lookup table into AL. The conversion of that nybble is done. The next task sounds simple but is actually surprisingly tricky: Writing the ASCII hex digit character now stored in AL into the display string at `HexStr`.

MOV AL, BYTE [Digits+RAX]



Note: Here, 'Digits' is the address of a 16-byte table in memory, containing ASCII hex digits from 0-F.

Figure 9.6: Using a lookup table

Multiplying by Shifting and Adding

The `hexdump1` program reads bytes from a file and displays them in lines, with 16 bytes represented in hex in each line. A sample of the output from `hexdump1` is shown here:

```
3B 20 20 45 78 65 63 75 74 61 62 6C 65 20 6E 61
6D 65 20 3A 20 45 40 54 53 59 53 43 40 4C 4C 0D
0A 3B 20 20 56 65 72 73 69 6F 6E 20 20 20 20 20
```

```

20 20 20 20 3A 20 30 2E 30 0D 0A 3B 20 20 43 72
65 60 74 65 64 20 64 60 74 65 20 20 20 20 3A 20
30 2F 37 2F 32 30 30 39 0D 0A 3B 20 20 4C 60 73
74 20 75 70 64 60 74 65 20 20 20 20 20 3A 20 32
2F 30 38 2F 32 30 30 39 0D 0A 3B 20 20 40 75 74
68 6F 72 20 20 20 20 20 20 20 20 20 20 3A 20 4A
    
```

Each of these lines is a display of the same data item: `HexStr`, a string of 48 characters with an EOL value (0ah) on the end. Each time `hexdump1` reads a block of 16 bytes from the input file, it formats them as ASCII hex digits and inserts them into `HexStr`. In a sense, this is another type of table manipulation, except that instead of looking up something in a table, we’re writing values *into* a table based on an index.

One way to think about `HexStr` is as a table of 16 entries, each entry three characters long. (See Figure 9.7.) In each entry, the first character is a space, and the second and third characters are the hex digits themselves. The space characters are already there, as part of the original definition of `HexStr` in the `.data` section. The original “empty” `HexStr` has 0 characters in all hex digit positions. To “fill” `HexStr` with “real” data for each line’s display, we have to scan through `HexStr` in an assembly language loop, writing the low nybble character and the high nybble character into `HexStr` separately.

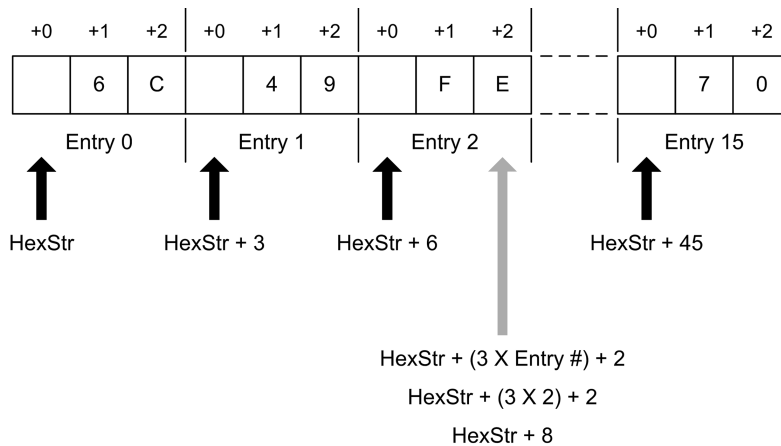


Figure 9.7: A table of 16 three-byte entries

The tricky business here is that for each pass through the loop, we have to “bump” the index into `HexStr` by three instead of just by one. The offset of one of those 3-byte entries in `HexStr` is the index of the entry multiplied by three. I’ve already described the `MUL` instructions, which handle arbitrary unsigned multiplication in the x86/x64 instruction set. `MUL`, however, is slow as instructions go.

It has other limitations as well, especially the ways it requires specific registers for its implicit operands.

Fortunately, with a little cleverness, there are other, faster ways to multiply in assembly. These ways are based on the fact that it's very easy and very fast to multiply by powers of two, using the `SHL` (Shift Left) instruction. It may not be immediately obvious to you, but each time you shift a quantity one bit to the left, you're multiplying that quantity by two. Shift a quantity two bits to the left, and you're multiplying it by four. Shift it three bits to the left, and you're multiplying by eight, and so on.

You can take my word for it, or you can actually watch it happen in a sandbox. Set up a fresh sandbox in SASM and enter the following instructions:

```
mov al,3
shl al,1
shl al,1
shl al,2
```

Build the sandbox and go into debug mode. Then step through the instructions, watching the value of `RAX` change in the Registers view for each step.

The first instruction loads the value 3 into `AL`. The next instruction shifts `AL` to the left by one bit. The value in `AL` becomes 6. The second `SHL` instruction shifts `AL` left by one bit again, and the 6 becomes 12. The third `SHL` instruction shifts `AL` by two bits, and the 12 becomes 48. I've shown this graphically in Figure 9.8.

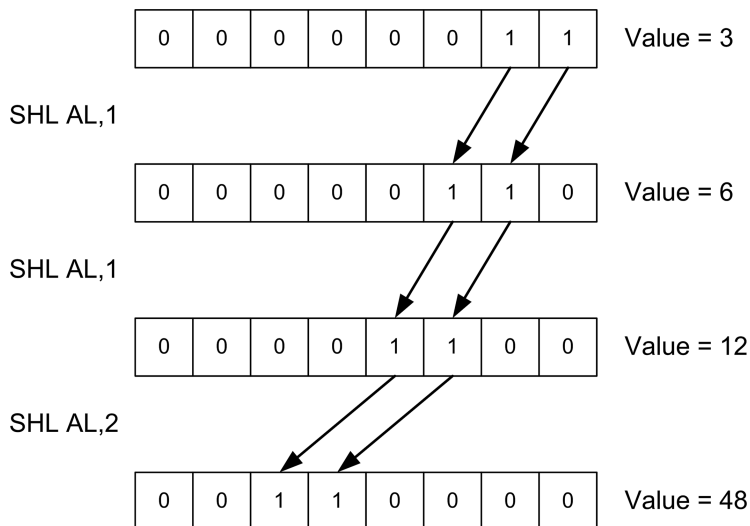


Figure 9.8: Multiplying by shifting

But what if you want to multiply by three? Easy: You multiply by 2 and then add one more copy of the multiplicand to the product. In the `hexdump1` program, it's done this way:

```
mov rdx,rcx    ; Copy the character counter into edx
shl rdx,1      ; Multiply pointer by 2 using left shift
add rdx,rcx    ; Complete the multiplication X3
```

Here, the multiplicand is loaded from the loop counter `RCX` into `RDX`. `RDX` is then shifted left by one bit to multiply it by 2. Finally, `RCX` is added once to the product `RDX` to make it multiplication by 3.

Multiplication by other numbers that are not powers of two may be done by combining an `SHL` and one or more `ADDS`. To multiply a value in `RCX` by seven, you would do this:

```
mov rdx,rcx    ; Keep a copy of the multiplicand in rcx
shl rdx,2      ; Multiply rdx by 4
add rdx,rcx    ; Makes it X 5
add rdx,rcx    ; Makes it X 6
add rdx,rcx    ; Makes it X 7
```

This may look clumsy, but remarkably enough, it's still faster than using `MUL`! (And there's an even faster way to multiply by three that I'll show you a little later in this chapter.)

Once you understand how the string table `HexStr` is set up, writing the hex digits into it is straightforward. The least significant hex digit is in `AL`, and the most significant hex digit is in `BL`. Writing both hex digits into `HexString` is done with a three-part effective memory address:

```
mov byte [HexStr+rdx+2],al ; Write LSB char digit to line string
mov byte [HexStr+rdx+1],bl ; Write MSB char digit to line string
```

Refer back to Figure 9.7 to work this out for yourself: You begin with the address of `HexStr` as a whole. `RDX` contains the offset of the first character in a given entry. To obtain the address of the entry in question, you add `HexStr` and `RDX`. However, that address is of the first character in the entry, which in `HexStr` is always a space character. The position of the LSB digit in an entry is the entry's offset +2, and the position of the MSB digit in an entry is the entry's offset +1. The address of the LSB digit is therefore `HexStr + the offset of the entry + 2`. The address of the MSB digit is therefore `HexStr + the offset of the entry + 1`.

Flags, Tests, and Branches

From a height, the idea of conditional jump instructions is simple, and without it, you won't get much done in assembly. I've been using conditional jumps informally in the last few example programs without saying much about them,

because the sense of the jumps was pretty obvious from context, and they were necessary to demonstrate other things. But underneath the simplicity of the idea of assembly language jumps lies a great deal of complexity. It's time to get down and cover that in detail.

Unconditional Jumps

A *jump* is just that: an abrupt change in the flow of instruction execution. Ordinarily, instructions are executed one after the other, in order, moving from low memory toward high memory. *Jump instructions* alter the address of the next instruction to be executed. Execute a jump instruction, and *zap!* All of a sudden you're somewhere else. A jump instruction can move execution forward in memory or backward. It can bend execution back into a loop (and it can tie your program logic in knots).

There are two kinds of jumps: conditional and unconditional. An *unconditional* jump is a jump that *always* happens. It takes this form:

```
jmp <label>
```

When this instruction executes, the sequence of execution moves to the instruction located at the label specified by <label>. It's just that simple.

Conditional Jumps

A *conditional* jump instruction is one of those fabled tests I introduced in Chapter 1. When executed, a *conditional jump* tests something, usually one, occasionally two, or far more rarely three of the flags in the RFlags register. If the flag or flags being tested happen to be in a particular state, execution will jump to a label somewhere else; otherwise, it simply falls through to the next instruction in sequence.

This two-way nature is important. Either a conditional jump instruction jumps or it falls through. Jump or no jump. It can't jump to one of two places, or three. Whether it jumps or not depends on the current value of a very small set of bits within the CPU.

As I mentioned earlier in this book while discussing the RFlags register as a whole, there is a flag that is set to 1 by certain instructions when the result of that instruction is zero: the Zero flag ZF. The DEC (DECrement) instruction is a good example. DEC subtracts 1 from its operand. If by that subtraction the operand becomes zero, ZF is set to 1. One of the conditional jump instructions, JZ (Jump if Zero) tests ZF. If ZF is found set to 1, a jump occurs, and execution transfers to the label after the JZ mnemonic. If ZF is found to be 0, execution falls through to the next instruction in sequence. This may be the most common conditional jump in the entire x86/x64 instruction set. It's often used when you're counting a register down to zero while executing a loop, and when the register counting

passes through the loop goes to zero by virtue of the `DEC` instruction, the loop ends, and execution picks up again at the instruction right after the loop.

Here's a simple (if nonoptimal) example, using instructions you should already understand:

```

mov [RunningSum],0 ; Clear the running total
mov rcx,17         ; We're going to do this 17 times

WorkLoop:
  add [RunningSum],3 ; Add three to the running total
  dec rcx            ; Subtract 1 from the loop counter
  jz SomewhereElse   ; If the counter is 0, we're done!
  jmp WorkLoop

```

The variable `RunningSum` was defined earlier with the `DQ` specifier, making it 64 bits in size. Before the loop begins, we set up a value in `RCX`, which acts as the count register and contains the number of times we're going to run through the loop. The body of the loop is where something gets done on each pass through the loop. In this example it's a single `ADD` instruction, but the body could be dozens or hundreds of instructions long.

After the work of the loop is accomplished, the count register is decremented by 1 with a `DEC` instruction. Immediately afterward, the `JZ` instruction tests the Zero flag. Decrementing `RCX` from 17 to 16, or from 4 to 3, does not set `ZF`, and the `JZ` instruction simply falls through. The instruction after `JZ` is an unconditional jump instruction, which obediently and consistently takes execution back to the `WorkLoop` label every time.

Now, decrementing `RCX` from 1 to 0 *does* set `ZF`. . .and that's when the loop ends. `JZ` finally takes us out of the loop by jumping to `SomewhereElse` (a label in the larger program that is not shown here), and execution leaves the loop.

You may be sharp enough (or experienced enough) to think that this is a lousy way to set up a loop, and you're right. (That doesn't mean it's never been done, nor that you yourself may not do it in a late-night moment of impatience.) What we're really looking for each time through the loop is when a condition—the Zero flag—*isn't* set, and there's an instruction for that too.

Jumping on the Absence of a Condition

There are quite a few conditional jump instructions, of which I'll discuss several but not all in this book. Their number is increased by the fact that almost every conditional jump instruction has an alter ego: a jump when the specified condition is *not* set to 1.

The `JZ` instruction provides a good example of jumping on a condition. `JZ` jumps to a new location in the code segment if the Zero flag (`ZF`) is set to 1.

JZ's alter ego is JNZ (Jump if Not Zero). JNZ jumps to a label if ZF is 0 and falls through if ZF is 1.

This may be confusing at first, because JNZ jumps when ZF is equal to 0. Keep in mind that the name of the instruction applies to the *condition* being tested and not necessarily the binary bit value of the flag. In the previous code example, JZ jumped when the DEC instruction decremented a counter to zero. The condition being tested is something connected with an earlier instruction, *not* simply the state of ZF.

Think of it this way: A condition raises a flag. "Raising a flag" means setting the flag to 1. When one of numerous instructions forces an operand to a value of zero (which is the condition), the Zero flag is raised. The logic of the instruction refers to the condition, *not* to the flag.

As an example, let's improve the little loop shown earlier by changing the loop logic to use JNZ:

```
mov word [RunningSum],0 ; Clear the running total
mov ecx,17               ; We're going to do this 17 times

WorkLoop:
add word [RunningSum],3 ; Add 3 to the running total
dec ecx                 ; Subtract 1 from the loop counter
jnz WorkLoop            ; If the counter is 0, we're done!
```

The JZ instruction has been replaced with a JNZ instruction. That makes much more sense, since to close the loop we have to jump, and we close the loop only while the counter is greater than 0. The jump back to label `WorkLoop` will happen only while the counter is greater than 0.

Once the counter decrements to 0, the loop is considered finished. JNZ "falls through," and the code that follows the loop (which I don't show here) executes. The point is that if you can position the program's next task immediately after the JNZ instruction, you don't need to use the unconditional JMP instruction *at all*. Instruction execution will just flow naturally into the next task that needs performing. The program will have a more natural and less-tangled top-to-bottom flow and will be easier to read and understand.

Flags

In Chapter 7, I explained the RFlags register and briefly described the purposes of all the flags it contains. RFlags is sparse; more than half of it is reserved for future use and thus undefined. Most of the flags that are defined are not terribly useful, especially when you're first starting out as an assembly programmer. The Carry flag (CF) and the Zero flag (ZF) will be 90 percent of your involvement in flags as a beginner, with the Direction flag (DF), Sign flag (SF), and

Overflow flag (OF) together making up an additional 9.998 percent. It might be a good idea to reread that part of Chapter 7 now, just in case your grasp of flag etiquette has gotten a little rusty.

As I explained earlier, `JZ` jumps when ZF is 1, whereas `JNZ` jumps when ZF is 0. Most instructions that perform some operation on an operand (such as `AND`, `OR`, `XOR`, `INC`, `DEC`, and all arithmetic instructions) set ZF according to the results of the operation. On the other hand, instructions that simply move data around (such as `MOV`, `XCHG`, `PUSH`, and `POP`) do not affect ZF nor any of the other flags at all. (Obviously, `POPF` affects the flags by popping the top-of-stack value into them.) One irritating exception is the `NOT` instruction, which performs a logical operation on its operand but does *not* set any flags—even when it causes its operand to become 0. Before you write code that depends on flags, *check your instruction reference* to make sure that you have the flag etiquette down correctly for that particular instruction. The x86/x64 instruction set is nothing if not quirky.

Comparisons with CMP

One major use of flags is in controlling loops. Another is in comparisons between two values. Your programs will often need to know whether a value in a register or memory is equal to some other value. Further, you may want to know if a value is greater than a value or less than a value if it is not equal to that value. There is a jump instruction to satisfy every need, but something has to set the flags for the benefit of the jump instruction. The `CMP` (CoMPare) instruction is what sets the flags for comparison tasks.

`CMP`'s use is straightforward and intuitive. The second operand is compared with the first, and several flags are set accordingly:

```
cmp <op1>,<op2>      ; Sets OF, SF, ZF, AF, PF, and CF
```

The sense of the comparison can be remembered if you simply recast the comparison in arithmetic terms:

```
Result = <op1> - <op2>
```

`CMP` is very much a subtraction operation where the result of the subtraction is thrown away, and only the flags are affected. The second operand is subtracted from the first. Based on the results of the subtraction, the flags it affects are set to appropriate values.

After a `CMP` instruction, you can jump based on several arithmetic conditions. People who have a reasonable grounding in math, and FORTRAN or Pascal programmers, will recognize the conditions: *Equal*, *Not equal*, *Greater than*, *Less than*, *Greater than or equal to*, and *Less than or equal to*. The sense of these operators follows from their names and is exactly like the sense of the equivalent operators in most high-level languages.

A Jungle of Jump Instructions

There is a bewildering array of jump instructions, but those dealing with arithmetic relationships sort out well into just six categories, one category for each of the six conditions I just listed. Complication arises out of the fact that there are *two* mnemonics for each machine instruction, for example, `JLE` (Jump if Less than or Equal) and `JNG` (Jump if Not Greater than). These two mnemonics are *synonyms* in that the assembler generates the identical binary opcode when it encounters either mnemonic. The synonyms are a convenience to you the programmer in that they provide two alternate ways to think about a given jump instruction. In the preceding example, *Jump if Less than or Equal to* is logically identical to *Jump if Not Greater than*. (Think about it!) If the importance of the preceding compare was to see if one value is less than or equal to another, you'd use the `JLE` mnemonic. On the other hand, if you were testing to be sure one quantity was not greater than another, you'd use `JNG`. The choice is yours.

Another complication is that there is a separate set of instructions for signed and unsigned arithmetic comparisons. I haven't spoken much about assembly language math in this book and thus haven't said much about the difference between signed and unsigned quantities. A *signed* quantity is one in which the high bit of the quantity is considered a built-in flag indicating whether the quantity is negative. If that bit is 1, the quantity is considered negative. If that bit is 0, the quantity is considered positive.

Signed arithmetic in assembly language is complex and subtle and not as useful as you might immediately think. I won't be covering it in detail in this book, though most assembly language books treat it to some extent. All you need to know to get a high-level understanding of signed arithmetic is that in signed arithmetic, negative quantities are legal and the most significant bit of a value is treated as the sign bit. (If the sign bit is set to 1, the value is considered negative.) Unsigned arithmetic, on the other hand, does not recognize negative numbers, and the most significant bit is just one more bit in the binary number expressing the value being tested.

"Greater Than" Versus "Above"

To tell the signed jumps apart from the unsigned jumps, the mnemonics use two different expressions for the relationship between two values:

- *Signed values* are thought of as being *greater than* or *less than*. For example, to test whether one signed operand is greater than another, you would use the `JG` (Jump if Greater) mnemonic after a `CMP` instruction.
- *Unsigned values* are thought of as being *above* or *below*. For example, to tell whether one unsigned operand is greater than (above) another, you would use the `JA` (Jump if Above) mnemonic after a `CMP` instruction.

Table 9.6 summarizes the arithmetic jump mnemonics and their synonyms. Any mnemonics containing the words *above* or *below* are for unsigned values, whereas any mnemonics containing the words *greater* or *less* are for signed values. Compare the mnemonics with their synonyms and see how the two represent opposite viewpoints from which to look at identical instructions.

Table 9.6: Jump Instruction Mnemonics and Their Synonyms

MNEMONICS		SYNONYMS	
JA	Jump if Above	JNBE	Jump if Not Below or Equal
JAE	Jump if Above or Equal	JNB	Jump if Not Below
JB	Jump if Below	JNAE	Jump if Not Above or Equal
JBE	Jump if Below or Equal	JNA	Jump if Not Above
JE	Jump if Equal	JZ	Jump if result is Zero
JNE	Jump if Not Equal	JNZ	Jump if result is Not Zero
JG	Jump if Greater	JNLE	Jump if Not Less than or Equal
JGE	Jump if Greater or Equal	JNL	Jump if Not Less
JL	Jump if Less	JNGE	Jump if Not Greater or Equal
JLE	Jump if Less or Equal	JNG	Jump if Not Greater

Table 9.6 simply serves to expand the mnemonics into a more comprehensible form and associate a mnemonic with its synonym. Table 9.7, on the other hand, sorts the mnemonics out by logical condition and according to their use with signed and unsigned values. Also listed in Table 9.7 are the flags whose values are tested by each jump instruction. Notice that some of the jump instructions require one of two possible flag values to take the jump while others require *both* of two flag values.

Several of the signed jumps compare two of the flags against one another. `JG`, for example, will jump when either `ZF` is 0 or when the Sign flag (`SF`) is equal to the Overflow flag (`OF`). (See Table 9.7.) I won't spend any further time explaining the nature of the Sign flag or Overflow flag. As long as you have the sense of each instruction under your belt, understanding exactly how the instructions test the flags can wait until you've gained some programming experience.

Table 9.7: Arithmetic Tests Useful After a CMP Instruction

CONDITION	PASCAL OPERATOR	UNSIGNED VALUES	JUMPS WHEN	SIGNED VALUES	JUMPS WHEN
Equal	=	JE	ZF=1	JE	ZF=1
Not Equal	<>	JNE	ZF=0	JNE	ZF=0
Greater Than	>	JA	CF=0 and ZF=0	JG	ZF=0 or SF=OF
Not Less than or Equal to		JNBE	CF=0 and ZF=0	JNLE	ZF=0 or SF=OF
Less than	<	JB	CF=1	JL	SF<>OF
Not Greater than or equal to		JNAE	CF=1	JNGE	SF<>OF
Greater than or equal to	>=	JAE	CF=0	JGE	SF=OF
Not Less than		JNB	CD=0	JNL	SF=OF
Less than or Equal to	<=	JBE	CF=1 or ZF=1	JLE	ZF=1 or SF<>OF
Not Greater Than		JNA	CF=1 or ZF=1	JNG	ZF=1 or SF<>OF

Some people have trouble understanding how it is that the `JE` and `JZ` mnemonics are synonyms, as are `JNE` and `JNZ`. Think again of the way a comparison is done within the CPU: The second operand is subtracted from the first, and if the result is 0 (indicating that the two operands were in fact equal), the Zero flag `ZF` is set to 1. That's why `JE` and `JZ` are synonyms: Both are simply testing the state of the Zero flag.

Looking for 1-Bits with TEST

The x86/x64 instruction set recognizes that bit testing is done a lot in assembly language, and it provides what amounts to a `CMP` instruction for bits: `TEST`. `TEST` performs an `AND` logical operation between two operands and then sets the flags as the `AND` instruction would, *without* altering the destination operand, as `AND` would. Here's the `TEST` instruction syntax:

```
test <operand>, <bit mask>
```

The bit mask operand should contain a 1 bit in each position where a 1 bit is to be sought in the operand, and 0 bits in all the other bits.

What `TEST` does is perform the `AND` logical operation between the instruction's destination operand and the bit mask and then set the flags as the `AND` instruction would do. The result of the `AND` operation is discarded, and the destination operand doesn't change. For example, if you want to determine if bit 3 of `RAX` is set to 1, you could use this instruction:

```
test rax,08h ; Bit 3 in binary is 00001000B, or 08h
```

Bit 3, of course, does not have the numeric value 3—you have to look at the bit pattern of the mask and express it as a binary or hexadecimal value. (Bit 3 represents the value 8 in binary.) Using binary for literal constants is perfectly legal in `NASM` and is often the clearest expression of what you're doing when you're working with bit masks:

```
test rax,00001000B ; Bit 3 in binary is 00001000B, or 08h
```

Destination operand `RAX` doesn't change as a result of the operation, but the `AND` truth table is asserted between `RAX` and the binary pattern `00001000`. If bit 3 in `RAX` is a 1 bit, then the Zero flag is cleared to 0. If bit 3 in `RAX` is a 0 bit, then the Zero flag is set to 1. Why? If you `AND` 1 (in the bit mask) with 0 (in `RAX`), you get 0. (Look it up in the `AND` truth table, which I showed in Table 9.2.) And if all eight bitwise `AND` operations come up 0, the result is 0, and the Zero flag is raised to 1, indicating that the result is 0.

Key to understanding `TEST` is thinking of `TEST` as a sort of Phantom of the Opcode, where the Opcode is `AND`. `TEST` puts on a mask (as it were) and *pretends* to be `AND` but then doesn't follow through with the results of the operation. It simply sets the flags *as though* an `AND` operation had occurred.

The `CMP` instruction we spoke of earlier is another Phantom of the Opcode and bears the same relation to `SUB` as `TEST` bears to `AND`. `CMP` subtracts its second operand from its first but doesn't follow through and store the result in the destination operand. It just sets the flags *as though* a subtraction had occurred. As we've already seen, this can be mighty useful when combined with conditional jump instructions.

Here's something important to keep in mind: *TEST is only useful for finding 1 bits*. If you need to identify 0 bits, you must first flip each bit to its opposite state with the `NOT` instruction. `NOT` changes all 1 bits to 0 bits and changes all 0 bits to 1 bits. Once all 0 bits are flipped to 1 bits, you can test for a 1 bit where you need to find a 0 bit. (Sometimes it helps to draw it out on paper to keep it all straight in your head.)

Finally, `TEST` will *not* reliably test for two or more 1 bits in the operand *at the same time*. `TEST` doesn't check for the presence of a bit pattern; *it checks for the presence of a single 1 bit*. In other words, if you need to check to make sure that both bits 4 and 5 are set to 1, `TEST` won't hack it.

Looking for 0-Bits with BT

As I explained, `TEST` has its limits: It's not cut out for determining when a bit is set to 0. `TEST` has been with us since the very earliest X86 CPUs, but the 386 and newer processors have an instruction that allows you to test for either 0 bits or 1 bits. `BT` (Bit Test) performs a very simple task: It copies the specified bit from the first operand into the Carry flag `CF`. In other words, if the selected bit was a 1 bit, the Carry flag becomes set. If the selected bit was a 0 bit, the Carry flag is cleared. You can then use any of the conditional jump instructions that examine and act on the state of `CF`.

`BT` is easy to use. It takes two operands: The destination operand is the value containing the bit in question. The source operand is the ordinal number of the bit that you want to test, *counting from 0*:

```
bt <value containing bit>,<bit number>
```

Once you execute a `BT` instruction, you should immediately test the value in the Carry flag and branch based on its value. Here's an example:

```
bt rax,4    ; Test bit 4 of RAX
jnc quit    ; We're all done if bit 4 = 0
```

Something to be careful of, especially if you're used to using `TEST`, is that *you are not creating a bit mask*. With `BT`'s source operand you are specifying the ordinal number of a bit. The literal constant 4 shown in the previous code is the bit's *number*, not the bit's *value*, and that's a crucial difference.

Also note in the previous code that we're branching if `CF` is *not* set; that's what `JNC` (Jump if Not Carry) does.

X64 Long Mode Memory Addressing in Detail

In so many ways, life is *better* now. And I'm not just talking about modern dentistry, plug-and-play networking, and eight-core CPUs. I used to program in assembly for the real-mode 8088 CPUs in the original IBM PC. And I remember real-mode memory addressing.

Like dentistry in the 1950s, 8088-based real-mode memory addressing was just. . .painful. It was a hideous ratbag of restrictions and gotchas and limits and Band-Aids, all of which veritably screamed out that the CPU was *desperately* hungry for more transistors on the die. Addressing memory, for example, was limited to `BX` and `BP` in most instructions, which meant a lot of fancy footwork when several separate items had to be addressed in memory all at the same time. And thinking about segment management still makes me shudder.

Well, over the past 40 years our Intel-family CPUs got pretty much all the transistors they wanted, and the bulk of those infuriating 16-bit memory addressing limitations have simply gone away. You can address memory with *any* of the general-purpose registers. You can even address memory directly with the stack pointer RSP, something that its 16-bit ancestor SP could not do. (You shouldn't *change* the value in RSP without considerable care, but RSP can now take part in addressing modes from which the stack pointer was excluded in 16-bit real-mode land.)

32-bit protected mode on the 386 CPU family introduced a general-purpose memory-addressing scheme in which all the GP registers could participate equally. Memory addressing in x64 long mode implements the same scheme with very few changes. I've sketched it out in Figure 9.9, which may well be the single most important figure in this entire book. *Memory addressing is the key skill in assembly language work.* If you don't understand how the CPU addresses memory, nothing else matters.

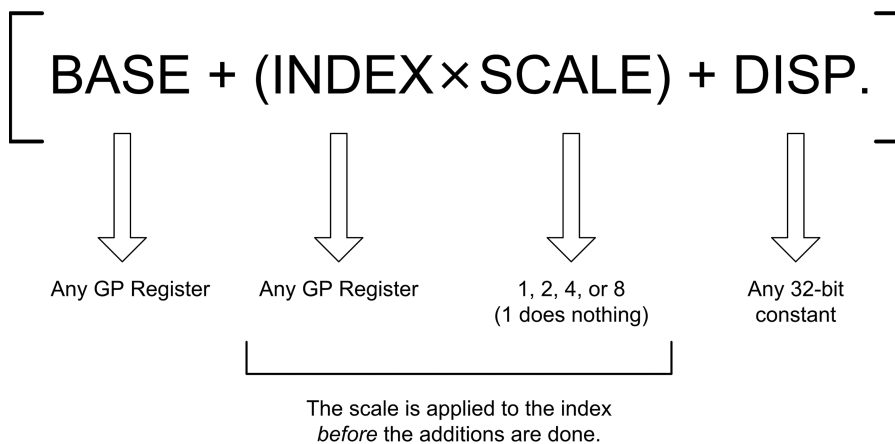


Figure 9.9: x64 long mode memory addressing

When I first studied and understood this scheme, wounds still bleeding from 16-bit 8088 segmented memory addressing, it looked too good to be true. But true it is! Here are the rules:

- The base and index registers may be any of the 64-bit general-purpose registers, including RSP.
- The displacement may be any 32-bit constant, either a literal value or a named value. Obviously, 0, while legal, isn't useful.
- The scale must be one of the values 1, 2, 4, or 8. That's it! The value 1 is legal, but given that the scale is used to multiply another value, 1 doesn't do anything useful.

- The index register is multiplied by the scale before the additions are done. In other words, it's not $(\text{base} + \text{index}) \times \text{scale}$. Only the index register may be multiplied by the scale.
- All of the elements are optional and may be used in almost any combination.
- Both 32-bit and 64-bit registers may be used, but *you may not mix register sizes in a single address*. That is, the registers in a single memory addressing operation must be all 32-bit or all 64-bit.
- 16-bit and 8-bit registers may *not* be used in memory addressing.

Within those rules, there are several different ways you can address memory, by gathering the address components shown in Figure 9.9 in different combinations. Examples are shown in Table 9.8.

Table 9.8: 64-Bit Long Mode Memory-Addressing Schemes

SCHEME	EXAMPLE	DESCRIPTION
[BASE]	[rdx]	Base only
[DISPLACEMENT]	[0F3h] or [<variable>]	Displacement, either literal or named constant
[BASE + DISPLACEMENT]	[rcx + 033h]	Base plus displacement
[BASE + INDEX]	[rax + ecx]	Base plus index
[INDEX × SCALE]	[rbx × 4]	Index times scale
[INDEX × SCALE + DISPLACEMENT]	[rax × 8 + 65]	Index times scale plus displacement
[BASE + INDEX × SCALE]	[rsp + rdi × 2]	Base plus index times scale
[BASE + INDEX × SCALE + DISP.]	[rsi + rbp × 4 + 9]	Base plus index times scale plus displacement

Effective Address Calculations

Each of the lines in Table 9.8 summarizes a method of expressing a memory address in 64-bit long mode. All but the first two involve a little arithmetic among two or more terms within the brackets that signify an address. This arithmetic is called *effective address calculation*, and the result of the calculation is the *effective address*. This term means the address that will ultimately be used to read or write memory, irrespective of how it is expressed. Effective address calculation is done by the instruction, when the instruction is executed.

The effective address in the Base scheme is simply the 64-bit value stored in the GP register between the brackets. No calculation is involved, but what we see in the source code is not a literal or symbolic address. So although the

instruction is coded with a register name between the brackets, the address that will be sent out to the memory system when the code executes is stored inside the register.

In most cases when you're dealing with an effective address, there's some arithmetic going on. In the Base + Index scheme, for example, the contents of the two GP registers between the brackets are added when the instruction is executed to form the effective address.

Displacements

Among the several components of a legal x64 long mode address, the displacement term is actually the slipperiest to understand. As I indicated in the previous paragraph, the displacement term can be a literal address, but in all my years of protected-mode assembly programming I've never done it nor seen anyone else do it. The reason? *You almost never know the literal address of something at assembly time.* There's another reason not to use literal addresses, which I'll come to shortly.

When the displacement term stands alone, it is virtually always a symbolic address. By that I mean a named data item that you've defined in your `.data` or `.bss` sections, like the `HexStr` variable from the `hexdump1` program in Listing 9.1:

```
mov rax, HexStr
```

What is placed in RAX here is the address given to the variable `HexStr` when the program is loaded into memory by the OS. Like all addresses, it's just a number, but it's determined when the program is loaded rather than at assembly time, as a literal constant numeric address would be. Also note that the previous bit of source code loads an address into RAX, *not* the value in memory at that address. For that you need brackets:

```
mov rax, [HexStr]
```

A lot of beginners get confused when they see what looks like two displacement terms in a single address. The confusion stems from the fact that if NASM sees two (or more) constant values in a memory reference, it will combine them at assembly time into a single displacement value, which is placed in RAX by the `MOV` instruction. That's what's done here:

```
mov rax, HexStr+3
```

Note the lack of brackets. The address referred to symbolically by the variable named `HexStr` is simply added to the literal constant 3 to form a single displacement value.

The key characteristic of a displacement term is that it is not stored in a register.

The x64 Displacement Size Problem

Now, there's an x64-specific gotcha with respect to displacements: A displacement value must not be more than 32 bits in size. Why? As I sometimes have to say, . . . it's complicated. And it's got nothing to do with the number of address bits supported in the silicon of a given x64 CPU. Put as simply as I can manage, limiting displacements to 32 bits was a design decision at AMD at the dawn of x64 time that "stuck." It may be fixed someday—or it may not. But, hey, never say "never."

In the meantime, we just have to live with it.

Base Addressing

When you exclude Displacement addressing, all x64 memory addressing is based on registers. The Base addressing scheme simply uses a single register into which an address has been loaded. It's called *Base* because all the more complex addressing schemes start with Base and extend it. Here's an example of Base addressing:

```
mov qword rax, [rcx]
```

This instruction takes whatever 64-bit value is stored in memory at the address contained in register RCX and loads it into register RAX.

Base + Displacement Addressing

A simple and common addressing scheme is Base + Displacement, and I demonstrated it in the `hexdump1` program in Listing 9.1. The instruction that inserts an ASCII character into the output line looks like this:

```
mov byte [HexStr+rdx+2], al
```

What happens here is that an 8-bit character value stored in register AL is written to the byte in memory addressed as `HexStr+RDX+2`. This is a perfect example of a case where there are two displacement terms that NASM combines into one. The variable name `HexStr` resolves to a number (the address of `HexStr`), and it is easily added to the literal constant 2. So, there is in truth only one base term (RDX) and one displacement term.

It's also a good example of how even 8-bit registers still have their uses, especially when you're dealing with 8-bit values like ASCII characters. Note too that the order of the terms in an address does not matter. The effective address could as well have been `RDX+HexStr+2`.

Base + Index Addressing

Perhaps the most common single addressing scheme is Base + Index, in which the effective address is calculated by adding the contents of two GP registers within the brackets. I demonstrated this addressing scheme in Chapter 8, in the `uppercaser2` program in Listing 8.2. Converting a character in the input buffer from lowercase to uppercase is done by subtracting 20h from it:

```
sub byte [r13+rbx],20h
```

The address of the buffer was earlier placed in RBP, and the number in RCX is the offset from the buffer start of the character being processed during any given pass through the loop. Adding the address of the buffer with an offset into the buffer yields the effective address of the character acted upon by the `SUB` instruction.

But wait. . .why not use Base + Displacement addressing? This instruction would be legal:

```
sub byte [Buff+rbx],20h
```

However, if you remember from the program (and it would be worth looking back and reading the associated text), we had to decrement the address of `Buff` by 1 before beginning the loop. But wait some more. . .could we have NASM do that little tweak by adding a second displacement term of -1? Indeed we could, and it would work. The central loop of the `uppercaser2` program would then look like this:

```
; Set up the registers for the process buffer step:

mov rbx, rax          ; Place the number of bytes read into rbx
mov r13, Buff         ; Place address of buffer into r13
; dec r13             We don't need this instruction anymore!

; Go through the buffer and convert lowercase to uppercase characters:

Scan:
cmp byte [r13-1+rbx], 61h ; Test input char against lowercase 'a'
jb Next                 ; If below 'a' in ASCII, not lowercase
cmp byte [r13-1+rbx], 7Ah ; Test input char against lowercase 'z'
ja Next                 ; If above 'z' in ASCII, not lowercase

                        ; Now we have a lowercase char
sub byte [r13-1+rbx], 20h ; Subtract 20h to give uppercase...

Next:
dec rbx                ; Decrement counter
jnz Scan               ; If characters remain, loop back
```

The `DEC R13` instruction in the first block is no longer necessary, and in the previous code that line is commented out. NASM does the math, and the address of `Buff` is decremented by 1 within the effective address expression when the program loads. This is actually the correct way to code this particular loop, and I thought long and hard about whether to show it back in Chapter 8 or wait until I could explain memory addressing schemes in detail.

Some people find the name “Base + Displacement” confusing, because in most cases, the Displacement term contains an address, and the Base term is a register containing an offset into a data item at that address. The word *displacement* resembles the word *offset* in most people’s experience, which can lead to confusion. This is one reason I don’t emphasize the names of the various memory addressing schemes in this book and certainly don’t recommend memorizing the names. *Understand how effective address calculation works*, and ignore the names of the schemes.

Index X Scale + Displacement Addressing

Base + Index addressing is what you’ll typically use to scan through a buffer in memory byte by byte. But what if you need to access a data item in a buffer or table where each data item is not a single byte, but a word or double word? This requires slightly more powerful memory addressing machinery.

As a side note here, the word *array* is the general term for what I’ve been calling a buffer or a table. Other writers may call a table an array, especially when the context of the discussion is a high-level language. But all three terms cook down to the same definition: a sequence of data items in memory, all of the same size and same internal definition. In the programs I’ve shown you so far, we’ve spoken only of very simple tables and buffers consisting of a sequence of 1-byte values all in a row. The `Digits` table in the `hexdump1` program in Listing 9.1 is such a table:

```
Digits: db "0123456789ABCDEF"
```

It consists of 16 single-byte ASCII characters in a row in memory, starting at the address represented by `Digits`. You can access the “C” character within `Digits` this way, using Base + Displacement addressing:

```
mov rcx,12
mov rdx,[Digits+rcx]
```

But what if you have a table containing 64-bit numeric values? Such a table is easy enough to define:

```
Sums: dq "15,12,6,0,21,14,4,0,0,19"
```

The `DQ` qualifier tells NASM that each item in the table `Sums` is a 64-bit quad word quantity. The literal constants plug a numeric value into each element of the table. The address of the first element (here, 15) in `Sums` is just the address of the table as a whole. So what is the address of the second element, 12? And how do you access it from assembly code?

Keep in mind that memory is addressed byte-by-byte, and not double word-by-double word or quad word-by-quad word. The second entry in the table is at an offset of 8 bytes into the table. If you tried to reference the second entry in the table using an address `[Sums+1]`, you would get one of the bytes inside the first table element's quad word, and that would *not* be useful.

This is where the concept of *scaling* comes in. An address may include a Scale term, which is a multiplier and may be any of the literal constants 2, 4, or 8. (The literal constant 1 is technically legal, but since the scale is a multiplier, 1 is not a useful scale value.) The product of the index and the scale terms is added to the displacement to give the effective address. This is known as the Index \times Scale + Displacement addressing scheme. Keep in mind that the Scale term can only be used with the Index term.

Typically, the scale term is the size of the individual elements in the table. If your table consists of 2-byte word values, the scale would be 2. If your table consists of 4-byte double word values, the scale would be 4. If your table consists of 8-byte quad word values, the scale would be 8.

The best way to explain this is with a diagram. In Figure 9.10, we're confronted with the address `[DQTable+ECX*8]`. `DQTable` is a table of quad word (64-bit) values. `DQTable`'s address is the displacement. The `RCX` register is the index, and for this example it contains 2, which is the number of the table element that you want to access. Because it's a table of 8-byte quad words, the scale value is 8. Note also that the multiplication symbol is not an "x" but an asterisk. The multiplication symbol "x" is not part of the ASCII character set, so like most high-level languages, assembly uses the asterisk as the multiplication operator symbol.

Because each table element is 8 bytes in size, the offset of element #2 from the start of the table is 16. The effective address of the element is calculated by first multiplying the index by the scale and then adding the product to the address of `DQTable`. There it is!

[DQTable + RCX * 8]

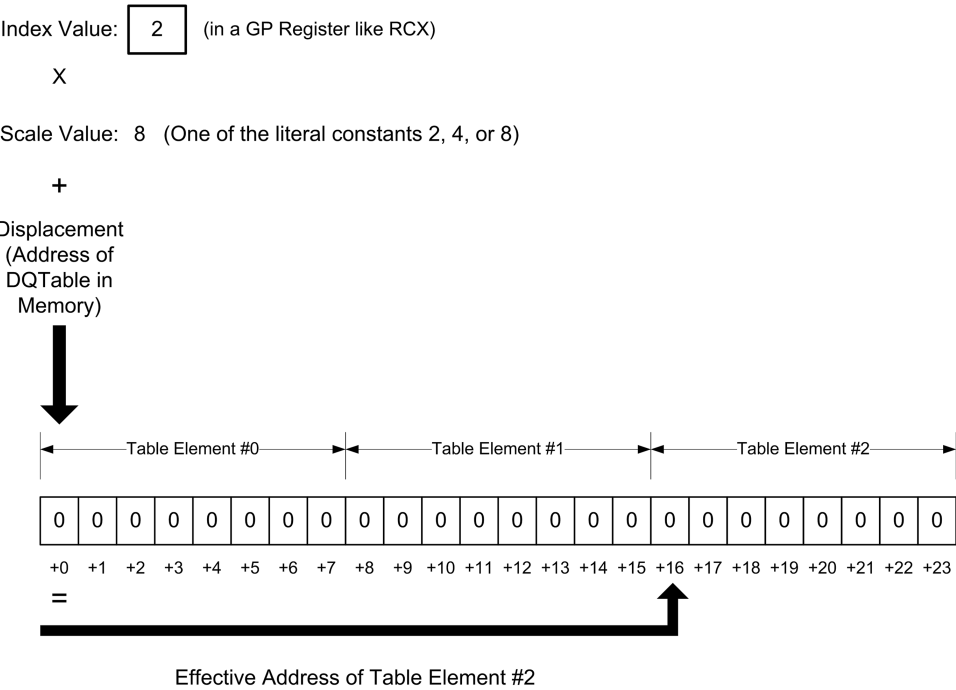


Figure 9.10: How address scaling works

Other Addressing Schemes

Any addressing scheme that includes scaling works just this way. The differences lie in what other terms are figured into the effective address. The Base + Index × Scale scheme adds a scaled index to a base value in register rather than a displacement:

```
mov rcx,2           ; Index is in rcx
mov rbp,DDTable     ; Table address is in rbp
mov rdx,[rbp+rcx*8] ; Put the selected element into rdx
```

You won't always be working with the address of a predefined variable like DDTable. Sometimes the address of the table will come from somewhere else, most often a two-dimensional table consisting of a number of subtables in memory,

each subtable containing some number of elements. Such tables are accessed in two steps: First you derive the address of the inner table in the outer table, and then you derive the address of the desired element within the inner table.

The most familiar example of this sort of two-dimensional table is something I presented in earlier editions of this book, written for DOS. The 25-line × 80-character text video memory buffer under DOS was a two-dimensional table. Each of the 25 lines was a table of 80 characters, and each character was represented by a 2-byte word. (One byte was the ASCII value, and the other byte specified attributes such as color, underlining, and so on.) So, the buffer as a whole was an overall table of 25 smaller tables, each containing 80 2-byte word values.

That sort of video access system died with DOS; Linux does not allow you direct access to PC video memory. It was done a lot in the DOS era, however, and is a good example of a two-dimensional table.

Scaling will serve you well for tables with 2-byte, 4-byte, or 8-byte elements. So what if your table consists of 3-byte elements? Or 5-byte elements? Or 17-byte elements? Alas, in such cases you will have to do some additional calculations to zero in on one particular element. Effective address calculation won't do the whole job itself. I've already given you an example of such a table in Listing 9.1. The line display string is a table of 3-byte elements. Each element contains a space character followed by the two hex digit characters. Because the elements are each three characters long, scaling cannot be done within the instruction and must be handled separately.

It's not difficult. Scaling for the 3-byte elements in the `HexStr` table in the `hexdump1` program is done like this:

```
mov rdx,rcx      ; Copy the character counter into rdx
shl rdx,1        ; Multiply counter by 2 using left shift
add rdx,rcx      ; Complete the multiplication X3
```

The calculation to multiply a value in `RDX` by 3 is done with a combination of an `SHL` instruction to multiply by 2, followed by an `ADD` instruction that adds a third copy of the index value to the shifted index value, effectively multiplying the original count value by 3.

Scaling for other index values can be done the same way. Scaling by 5 would be done by shifting the index value left by 2 bits, thus multiplying it by 4, followed by adding another copy of the index value to complete the multiplication by 5. In general terms, to scale an index value by *X*:

1. Find the largest power of 2 less than *X*.
2. Shift the index value left by that power of 2.
3. Add a copy of the original index value to the shifted copy as many times as it takes to complete the multiplication by *X*.

For example, if *X* is 11, the scale calculation would be done this way:

```
mov rdx,rcx ; Copy the index into rdx
shl rdx,3   ; Multiply index X 8 by shifting index left 3X
add rdx,rcx ; Add first of 3 additional copies of index
add rdx,rcx ; Add second of 3 additional copies of index
add rdx,rcx ; Add third of 3 additional copies of index
```

This works best for relatively small-scale values; once you get past 20, there will be a lot of `ADD` instructions. At that point, the answer is not to calculate the scale but look up the scale in a table specially defined for a given scale value. For example, suppose your table elements are each 25 bytes long. You could define a table with multiples of 25:

```
ScaleValues: dd 0,25,50,75,100,125,150,175,200,225,250,275
```

To scale an index value of 6 for an entry size of 25, you would look up the product of 6×25 in the table this way:

```
mov rcx,6
mov rax,[ScaleValues+rcx*4]
```

The value in `RAX` now contains the effective address of the first byte of element 6, counting elements (as usual) from 0.

LEA: The Top-Secret Math Machine

But wait, there's *more*. One of the oddest and in some respects the most wonderful instructions in the Intel architecture is `LEA`, Load Effective Address. On the surface, what it does is simple: It calculates an effective address using the terms between the brackets of its source operand and loads that address into any 64-bit general-purpose register given as its destination operand.

Look back to the code shown just before this section begins. The `MOV` instruction looks up the element with index 6 in the table `ScaleValues`. To look up the item at index 6, it has to first calculate the effective address of the item at index 6. This address is then used to access memory.

But what if you'd like to save that address in a register to use it later without having to calculate it all over again? That's what `LEA` does. Here's `LEA` in action:

```
lea rbx,[ScaleValues+rcx*4]
```

What happens here is that the CPU calculates the effective address given inside the brackets and loads that address into the `RBX` register. Keep in mind that the individual entries in a table do not have labels and thus cannot be referenced directly. `LEA` gives you the ability to calculate the effective address of any element in a table (or any calculable address at all!) and drop that address in a register.

In itself this is very useful. However, `LEA` has an “off-label” purpose: doing fast math without shifts, adds, or `MUL`. If you remember, there is a calculation in the `hexdump1gcc` program that multiplies by 3 using a shift and an add:

```
mov rdx,rcx    ; Copy the character counter into rdx
shl rdx,1      ; Multiply pointer by 2 using left shift
add rdx,rcx    ; Complete the multiplication X3
```

This works. But look at what we can use that does exactly the same thing:

```
mov rdx,rcx    ; Copy the character counter into rdx
lea rdx,[rdx*2+rdx] ; Multiply rdx X 3
```

Not only is this virtually always faster than shifts combined with adds, it’s also clearer from your source code what sort of calculation is actually being done. The fact that what ends up in `RDX` may not in fact be the legal address of anything is unimportant. *LEA does not try to reference the address it calculates.* It does the math on the stuff inside the brackets and drops the result in the destination operand. Job over. Memory is not touched, and the flags are not affected.

Of course, you’re limited to what calculations can be done that yield effective addresses. But right off the top, you can multiply any GP register by 2, 3, 4, 5, 8, and 9. It’s not arbitrary math, but multiplying by 2, 3, 4, 5, 8, and 9 comes up regularly in assembly work, and you can combine `LEA` with shifts and adds to do more complex math and “fill in the holes.” You can also use multiple `LEA` instructions in a row. Two consecutive `LEA` instructions can multiply a value by 10, which is useful indeed:

```
lea rbx,[rbx*2]    ; Multiply rbx X 2, put product in RBX
lea rbx,[rbx*4+rbx] ; Multiply rbx X 5 for a total of X 10
```

Some people consider this use of `LEA` a scurvy trick, but in all the years I’ve worked in x86/x64 assembly I’ve never seen a downside. Before throwing five or six instructions into the pot to cook up a particular multiplication, see if two or three `LEAs` can do it instead. `LEA` does its work in one machine cycle, and CPU math doesn’t get any faster than that!

Character Table Translation

There is a type of table lookup that is (or perhaps was) so common that Intel’s engineers baked a whole instruction into the x86 architecture to do nothing but. The type of table lookup is what I was alluding to toward the end of Chapter 8: character conversion. In the early 1980s I needed to convert character sets in various ways, the simplest of which was forcing all lowercase characters to uppercase. And so in Chapter 8 we built a simple program that went through a

file a buffer at a time, bringing in characters, converting all lowercase characters to uppercase, and then writing them all back out again to a new file.

The conversion itself was simple: By relying on the ASCII chart for the relationship between all uppercase characters and their associated lowercase, we could convert a lowercase character to uppercase by simply subtracting 20h (32) from the character. That's reliable, but is very much a special case. It just so happens that ASCII lowercase characters are always 32 higher on the chart than their equivalent uppercase characters. What do you do if you need to convert all "vertical bar" (ASCII 124) characters to exclamation points? (I had to do this once, because one of the doddering old mainframes couldn't digest vertical bars.) You can write special code for each individual case that you have to deal with. . .

. . .or you can use a translation table.

Translation Tables

A translation table is a special type of table, and it works the following way: You set up a table of values, with one entry for every possible value that must be translated. A number (or a character, treated as a numeric value) is used as an index into the table. At the index position in the table is a value that is used to replace the original value that was used as the index. In short, the original value indexes into the table and finds a new value that replaces the original value, thus translating the old value to a new one.

We've done this once before, in the `hexdump1gcc` program in Listing 9.1. Recall the `Digits` table:

```
Digits: db "0123456789ABCDEF"
```

This is a translation table, though I didn't call it that at the time. The idea, if you recall, was to separate the two 4-bit halves of an 8-bit byte and convert those 4-bit values into ASCII characters representing hexadecimal digits. The focus at the time was separating the bytes into two nybbles via bitwise logical operations, but there was translation going on there as well.

The translation was accomplished by these three instructions:

```
mov al,byte [rsi+rcx]    ; Put a byte from the input buffer
                        ; into al
and al,0Fh              ; Mask out all but the low nybble
mov al,byte [Digits+rax] ; Look up the char equivalent of nybble
```

The first instruction loads a byte from the input buffer into the 8-bit AL register. The second instruction masks out all but the low nybble of AL. The third instruction does a memory fetch: It uses the value in AL to index into the `Digits` table and brings back whatever value was in the ALth entry in the table. (This has to be done using RAX between the brackets, because AL cannot take part

in effective address calculations. Just remember that AL is the lowest-order byte in the RAX register.) If AL held 0, the effective address calculation added 0 to the address of `Digits`, bringing back the 0th table entry, which is the ASCII character for 0. If AL held 5, effective address calculation added 5 to the address of `Digits`, bringing back the 5th table entry, which is the ASCII character for 5. And so it would go for all 16 possible values that may be expressed in a 4-bit nybble. Basically, the code is used to translate a number to the ASCII character equivalent of that number.

There are only 16 possible hexadecimal digits, so the conversion table in `hexdump1gcc` only needs to be 16 bytes long. A byte contains enough bits to represent 256 different values, so if we're going to translate byte-sized values, we're going to need a table with 256 entries. Technically, the ASCII character set uses only the first 128 values, but as I described earlier in this book, the "high" 128 values have often been assigned to special characters like non-English letters, "box-draw" characters, mathematical symbols, and so on. One common use of character translation is to convert any characters with values higher than 128 to something lower than 128 to avoid havoc in older systems that can't deal with extended ASCII values.

Such a table is easy enough to define in an assembly language program:

`UpCase:`

```
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,09h,0Ah,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
db 60h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,7Bh,7Ch,7Dh,7Eh,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
```

The `UpCase` table is defined in 16 lines of 16 separate hexadecimal values. The fact that it's split across 16 lines in the code listing is purely for readability on the screen or printed page and does not affect the binary table that NASM generates in the output `.o` file. Once it's in binary, it's 256 8-bit values in a row.

A quick syntactic note here: When defining tables (or any data structure containing multiple predefined values), commas are used to separate values

within a single definition. There is no need for commas at the ends of the lines of the DB definitions in the previous table. Each DB definition is separate and independent, but because they are adjacent in memory, we can treat the 16 DB definitions as a single 256-byte table.

Any translation table can be thought of as expressing one or more “rules” governing what happens during the translation process. The `UpCase` table shown earlier expresses these translation rules:

- All lowercase ASCII characters are translated to uppercase.
- All printable ASCII characters less than 127 that are *not* lowercase are translated to themselves. (They’re not precisely “left alone” but are still translated, just to the same characters.)
- All “high” character values from 127 through 255 are translated to the ASCII space character (32, or 20h).
- All nonprintable ASCII characters (basically, values 0–31, plus 127) are translated to spaces *except* for values 9 and 10.
- Character values 9 and 10 (tab and EOL) are translated as themselves.

Not bad for a single data item, eh? (Just imagine how much work it would be to do all that fussing purely with machine instructions!)

Translating with MOV or with XLAT

So, how do we use the `UpCase` table? The obvious way would be this:

- Load the character to be translated into AL.
- Create a memory reference using AL as the base term and `UpCase` as the displacement term, and `MOV` the byte at the memory reference into AL, replacing the original value used as the base term.

The hypothetical `MOV` instruction would look like this:

```
mov al, byte [UpCase+al]
```

There’s only one problem: *NASM won’t let you do this*. In 32-bit protected mode and x64 long mode, the AL register can’t take part in effective address calculations, nor can any other of the 8-bit registers. Enter `XLAT`.

The `XLAT` instruction is hard-coded to use certain registers in certain ways. Its two operands are both implicit:

- The address of the translation table must be in RBX.
- The character to be translated must be in AL.
- The translated character will be returned in AL, replacing the character originally placed in AL.

With the registers set up, the `XLAT` instruction has no operands and is used all by its lonesome:

```
xlat
```

I'll be honest here: `XLAT` is less of a win than it used to be. In x64 long mode, the same thing can be done with the following instruction:

```
mov al, byte [UpCase+rax]
```

The 64-bit register `RAX` can stand in for little 8-bit `AL` when calculating an effective address of the character used to translate the character in `AL`. There's only one catch: You must clear out any "leftover" values in the high 56 bits of `RAX`, or you could accidentally index far beyond the bounds of the translation table. The problem doesn't arise with `XLAT` since the `XLAT` instruction uses *only* `AL` for the index, ignoring whatever else might be in the higher bits of `RAX`.

Clearing `RAX` before loading the value to be translated into `AL` is done in either of these two common ways:

```
xor rax,rax
mov rax,0
```

In truth, given `XLAT`'s requirement that it use `AL` and `RBX`, it's a wash, but the larger topic of character translation via tables is really what I'm trying to present here. Listing 9.2 puts it all into action. The program as shown does exactly what the `uppercaser2` program in Listing 8.2 does: It forces all lowercase characters in an input file to uppercase and writes them to an output file. I didn't call it "uppercaser3" because it is a general-purpose character translator. In this particular example, with the `UpCase` table, it translates lowercase characters to uppercase; however, that's simply one of the rules that the `UpCase` table expresses. Change the table, and you change the rules. You can translate any or all of the 256 different values in a byte to any 8-bit value or values.

I've added a second table to the program for you to experiment with. The `Custom` table expresses these rules:

- All printable ASCII characters less than 127 are translated to themselves. (They're not precisely "left alone" but are still translated, just to the same characters.)
- All "high" character values from 127 through 255 are translated to the ASCII space character (32, or 20h.)
- All non-printable ASCII characters (basically, values 0–31, plus 127) are translated to spaces *except* for values 9 and 10.
- Character values 9 and 10 (tab and EOL) are translated as themselves.

Basically, it leaves all printable characters (plus tab and EOL) alone and converts all other character values to 20h, the space character. You can substitute

the label `Custom` for `UpCase` in the program, make changes to the `Custom` table, and try it out. Convert that pesky vertical bar to an exclamation point. Change all “Z” characters to “Q.” Changing the rules is done by changing the table. The code does not change at all!

As with earlier programs, `xlat1gcc` reads from standard input and writes to standard output. Copy some text to the clipboard and drop it into SASM’s Input window. Then run the program, and see what it writes to the Output window.

Listing 9.2: `xlat1gcc.asm`

```
; Executable name : xlat1gcc
; Version          : 2.0
; Created date     : 8/21/2022
; Last update      : 7/17/2023
; Author           : Jeff Duntemann
; Description       : A simple program in assembly for Linux,
;                   : using NASM 2.15, demonstrating the XLAT
;                   : instruction to translate characters using
;                   : translation tables.
;
; Run it either in SASM or using this command in the Linux terminal:
;
;     xlat1gcc < input file > output file
;
;     If an output file is not specified, output goes to stdout
;
; Build using SASM's default build setup for x64
; To test from a terminal, save out the executable to disk.

SECTION .data          ; Section containing initialised data

    StatMsg: db "Processing...",10
    StatLen: equ $-StatMsg
    DoneMsg: db "...done!",10
    DoneLen: equ $-DoneMsg

; The following translation table translates all lowercase characters
; to uppercase. It also translates all non-printable characters to
; spaces, except for LF and HT. This is the table used by default in
; this program.
UpCase:
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,09h,0Ah,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
db 60h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,7Bh,7Ch,7Dh,7Eh,20h
```

```

db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h

```

```

; The following translation table is "stock" in that it translates all
; printable characters as themselves, and converts all non-printable
; characters to spaces except for LF and HT. You can modify this to
; translate anything you want to any character you want. To use it,
; replace the default table name (UpCase) with Custom in the code below.

```

```
Custom:
```

```

db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,09h,0Ah,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,21h,22h,23h,24h,25h,26h,27h,28h,29h,2Ah,2Bh,2Ch,2Dh,2Eh,2Fh
db 30h,31h,32h,33h,34h,35h,36h,37h,38h,39h,3Ah,3Bh,3Ch,3Dh,3Eh,3Fh
db 40h,41h,42h,43h,44h,45h,46h,47h,48h,49h,4Ah,4Bh,4Ch,4Dh,4Eh,4Fh
db 50h,51h,52h,53h,54h,55h,56h,57h,58h,59h,5Ah,5Bh,5Ch,5Dh,5Eh,5Fh
db 60h,61h,62h,63h,64h,65h,66h,67h,68h,69h,6Ah,6Bh,6Ch,6Dh,6Eh,6Fh
db 70h,71h,72h,73h,74h,75h,76h,77h,78h,79h,7Ah,7Bh,7Ch,7Dh,7Eh,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h
db 20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h,20h

```

```
SECTION .bss ; Section containing uninitialized data
```

```

READLEN equ 1024 ; Length of buffer
ReadBuffer: resb READLEN ; Text buffer itself

```

```
SECTION .text ; Section containing code
```

```
global main
```

```
main:
```

```
mov rbp, rsp ; This keeps gdb happy...
```

```
; Display the "I'm working..." message via stderr:
```

```

mov rax,1 ; Specify sys_write call
mov rdi,2 ; Specify File Descriptor 2: Standard error
mov rsi,StatMsg ; Pass address of the message

```

```

    mov rdx,StatLen    ; Pass the length of the message
    syscall            ; Make kernel call

; Read a buffer full of text from stdin:
read:
    mov rax,0          ; Specify sys_read call
    mov rdi,0          ; Specify File Descriptor 0: Standard Input
    mov rsi,ReadBuffer ; Pass address of the buffer to read to
    mov rdx,READLEN    ; Pass number of bytes to read at one pass
    syscall
    mov rbp,rax        ; Copy sys_read return value for safekeeping
    cmp rax,0          ; If rax=0, sys_read reached EOF
    je done            ; Jump If Equal (to 0, from compare)

; Set up the registers for the translate step:
    mov rbx,UpCase      ; Place the address of the table into rbx
    mov rdx,ReadBuffer  ; Place the address of the buffer into rdx
    mov rcx,rbp         ; Place number of bytes in the buffer into rcx

; Use the xlat instruction to translate the data in the buffer:
translate:
    xor rax,rax         ; Clear rax
    mov al,byte [rdx-1+rcx] ; Load character into AL for translation
    xlat                ; Translate character in AL via table
    mov byte [rdx-1+rcx],al ; Put the xlated character back in buffer
    dec rcx             ; Decrement character count
    jnz translate       ; If there are more chars in the buffer, repeat

; Write the buffer full of translated text to stdout:
write:
    mov rax,1          ; Specify sys_write call
    mov rdi,1          ; Specify File Descriptor 1: Standard output
    mov rsi,ReadBuffer ; Pass address of the buffer
    mov rdx,rbp        ; Pass the # of bytes of data in the buffer
    syscall            ; Make kernel call
    jmp read           ; Loop back and load another buffer full

; Display the "I'm done" message via stderr:
done:
    mov rax,1          ; Specify sys_write call
    mov rdi,2          ; Specify File Descriptor 2: Standard error
    mov rsi,DoneMsg    ; Pass address of the message
    mov rdx,DoneLen    ; Pass the length of the message
    syscall            ; Make kernel call

; All done! Let's end this party:
    ret                ; Return to the glibc shutdown code

```

Tables Instead of Calculations

Standardization among computer systems has made character translation a lot less common than it used to be, but translation tables can be extremely useful in other areas. One of them is to perform faster math. Consider the following table:

```
Squares: db 0,1,4,9,16,25,36,49,64,81,100,121,144,169,196,225
```

No mystery here: `Squares` is a table of the squares of the numbers from 0–15. If you needed the square of 14 in a calculation, you could use `MUL`, which is slower than most instructions and requires two GP registers. Or you could simply fetch down the result from the `Squares` table:

```
mov rcx,14
mov al,byte [Squares+rcx]
```

Voilà! `RAX` now contains the square of 14. You can do the same trick with `XLAT`, though it requires that you use certain registers. Also remember that `XLAT` is limited to 8-bit quantities. The `Squares` table shown here is as large a squares value table as `XLAT` can use, because the next square value (of 16) is 256, which cannot be expressed in 8 bits, and thus a lookup table containing it cannot be used by `XLAT`.

Making the entries of a squares value lookup table 16 bits in size will allow you to include the squares of all integers up to 255. And if you give each entry in the table 32 bits, you can include the squares of integers up to 65,535, but that would be a *very* substantial table!

I don't have the space in this book to go into floating-point math, but using tables to look up values for things such as square roots was once done very frequently. Modern CPUs with math systems like AVX make such techniques a lot less compelling. Still, when confronted with a math calculation challenge, you should always keep the possibility of using table lookups somewhere in the corner of your mind.