7.4.1. Preliminaries

Conditional Comparison Instructions

Comparison instructions perform an arithmetic operation for the purpose of guiding the conditional execution of a program. Table 1 lists the basic instructions associated with conditional control.

Table 1. Conditional Control Instructions

Instruction	Translation
`cmp R1, R2`	Compares R2 with R1 (i.e., evaluates R2 - R1)
`test R1, R2`	Computes R1 & R2

The `cmp` instruction compares the values of two registers, R2 and R1. Specifically, it subtracts R1 from R2. The `test` instruction performs bitwise AND. It is common to see an instruction like:

```
test %rax, %rax
```

In this example, the bitwise AND of `%rax` with itself is zero only when `%rax` contains zero. In other words, this is a test for a zero value and is equivalent to:

```
cmp $0, %rax
```

Unlike the arithmetic instructions covered thus far, `cmp` and `test` do not modify the destination register. Instead, both instructions modify a series of single-bit values known as **condition code flags**. For example, `cmp` will modify condition code flags based on whether the value R2 - R1 results in a positive (greater), negative (less), or zero (equal) value. Recall that condition code values encode information about an operation in the ALU. The condition code flags are part of the `FLAGS` register on x86 systems.

Table 2. Common Condition Code Flags.

Flag	Translation
ZF	Is equal to zero (1: yes; 0: no)
SF	Is negative (1: yes; 0: no)
OF	Overflow has occurred (1:yes; 0: no)

Flag	Translation
`CF`	Arithmetic carry has occurred (1: yes; 0:no)

Table 2 depicts the common flags used for condition code operations. Revisiting the `cmp R1, R2` instruction:

- The `ZF` flag is set to 1 if R1 and R2 are equal.
- The `SF` flag is set to 1 if R2 is *less* than R1 (R2 - R1 results in a negative value).
- The `OF` flag is set to 1 if the operation R2 - R1 results in an integer overflow (useful for signed comparisons).
- The `CF` flag is set to 1 if the operation R2 - R1 results in a carry operation (useful for unsigned comparisons).

The `SF` and `OF` flags are used for comparison operations on signed integers, whereas the `CF` flag is used for comparisons on unsigned integers. Although an in-depth discussion of condition code flags is beyond the scope of this book, the setting of these registers by `cmp` and `test` enables the next set of instructions we cover (the *jump* instructions) to operate correctly.

The Jump Instructions

A jump instruction enables a program's execution to "jump" to a new position in the code. In the assembly programs we have traced through thus far, `%rip` always points to the next instruction in program memory. The jump instructions enable `%rip` to be set to either a new instruction not yet seen (as in the case of an `if` statement) or to a previously executed instruction (as in the case of a loop).

Direct jump instructions

Table 3. Direct Jump Instructions

Instruction	Description
`jmp L`	Jump to location specified by L
`jmp *addr`	Jump to specified address

Table 3 lists the set of direct jump instructions; `L` refers to a **symbolic label**, which serves as an identifier in the program's object file. All labels consist of some letters and digits followed by a colon. Labels can be *local* or *global* to an object file's scope. Function labels tend to be *global* and usually consist of the function name and a colon. For example, `main:` (or `<main>:`) is used to label a user-defined

`main` function. In contrast, labels whose scope are *local* are preceded by a period. For example, `.L1:` is a local label one may encounter in the context of an `if` statement or loop.

All labels have an associated address. When the CPU executes a `jmp` instruction, it modifies `%rip` to reflect the program address specified by label `L`. A programmer writing assembly can also specify a particular address to jump to using the `jmp *` instruction. Sometimes, local labels are shown as an offset from the start of a function. Therefore, an instruction whose address is 28 bytes away from the start of `main` may be represented with the label `<main+28>`.

For example, the instruction `jmp 0x8048427 <main+28>` indicates a jump to address 0x8048427, which has the associated label `<main+28>`, representing that it is 28 bytes away from the starting address of the `main` function. Executing this instruction sets `%rip` to 0x8048427.

Conditional jump instructions

The behavior of conditional jump instructions depends on the condition code registers set by the `cmp` instruction. Table 4 lists the set of common conditional jump instructions. Each instruction starts with the letter `j` denoting that it is a jump instruction. The suffix of each instruction indicates the *condition* for the jump. The jump instruction suffixes also determine whether to interpret numerical comparisons as signed or unsigned.

Table 4. Conditional Jump Instructions; Synonyms Shown in Parentheses

Signed Comparison	Unsigned Comparison	Description
`je (jz)`		jump if equal (==) or jump if zero
`jne (jnz)`		jump if not equal (!=)
`js`		jump if negative
`jns`		jump if non-negative
`jg (jnle)`	`ja (jnbe)`	jump if greater (>)
`jge (jnl)`	`jae (jnb)`	jump if greater than or equal (>=)
`jl (jnge)`	`jb (jnae)`	jump if less (<)
`jle (jng)`	`jbe (jna)`	jump if less than or equal (<=)

Instead of memorizing these different conditional jump instructions, it is more helpful to sound out the instruction suffixes. Table 5 lists the letters commonly found in jump instructions and their word correspondence.

Table 5. Jump Instruction Suffixes.

Letter	Word
`j`	jump
n	not
e	equal
s	signed
g	greater (signed interpretation)
`l`	less (signed interpretation)
a	above (unsigned interpretation)
b	below (unsigned interpretation)

Sounding it out, we can see that `jg` corresponds to *jump greater* and that its signed synonym `jnl` stands for *jump not less*. Likewise, the unsigned version `ja` stands for *jump above*, whereas its synonym `jnbe` stands for *jump not below or equal*.

If you sound out the instructions, it helps to explain why certain synonyms correspond to particular instructions. The other thing to remember is that the terms *greater* and *less* instruct the CPU to interpret the numerical comparison as a signed value, whereas *above* and *below* indicate that the numerical comparison is unsigned.

The `goto` Statement

In the following subsections, we look at conditionals and loops in assembly and reverse engineer them back to C. When translating assembly code of conditionals and loops back into C, it is useful to understand the corresponding C language `goto` forms. The `goto` statement is a C primitive that forces program execution to switch to another line in the code. The assembly instruction associated with the `goto` statement is `jmp`.

The `goto` statement consists of the `goto` keyword followed by a **goto label**, a type of program label that indicates where execution should continue. So, `goto done` means that the program execution should jump to the line marked by label `done`. Other examples of program labels in C include the switch statement labels previously covered in Chapter 2.

Table 6. Comparison of a C function and its associated goto form.

Regular C version	Goto version

```c
int getSmallest(int x, int y) {
    int smallest;
    if ( x > y ) { //if (conditional)
        smallest = y; //then statement
    }
    else {
        smallest = x; //else statement
    }
    return smallest;
}
```

```c
int getSmallest(int x, int y) {
    int smallest;

    if (x <= y ) { //if (!conditional)
        goto else_statement;
    }
    smallest = y; //then statement
    goto done;

else_statement:
    smallest = x; //else statement

done:
    return smallest;
}
```

Table 6 depicts a function `getSmallest()` written in regular C code and its associated `goto` form in C. The `getSmallest()` function compares the value of two integers (`x` and `y`), and assigns the smaller value to variable `smallest`.

The `goto` form of this function may seem counterintuitive, but let's discuss what exactly is going on. The conditional checks to see whether variable `x` is less than or equal to `y`.

- If `x` is less than or equal to `y`, the program transfers control to the label marked by `else_statement`, which contains the single statement `smallest = x`. Since the program executes linearly, the program continues on to execute the code under the label `done`, which returns the value of `smallest (x)`.
- If `x` is greater than `y`, `smallest` is assigned the value `y`. The program then executes the statement `goto done`, which transfers control to the `done` label, which returns the value of `smallest (y)`.

While `goto` statements were commonly used in the early days of programming, the use of `goto` statements in modern code is considered bad practice, as it reduces the overall readability of code. In fact, computer scientist Edsger Dijkstra wrote a famous paper lambasting the use of `goto` statements called *Go To Statement Considered Harmful*[1].

In general, well-designed C programs do not use `goto` statements and programmers are discouraged from using them to avoid writing code that is difficult to read, debug, and maintain. However, the C `goto` statement is important to understand, as GCC typically changes C code with conditionals into a `goto` form prior to translating it to assembly, including code that contains `if` statements and loops.

The following subsections cover the assembly representation of `if` statements and loops in greater detail.

- If Statements
- Loops

References

1. Edsger Dijkstra. "Go To Statement Considered Harmful". *Communications of the ACM* 11(3) pp. 147—148. 1968.

Contents