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

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

```
    int comp(data_t a, data_t b)
    a in %rdi, b in %rsi
1   comp:
2     cmpq    %rsi, %rdi     Compare a:b
3     setl    %al            Set low-order byte of %eax to 0 or 1
4     movzbl  %al, %eax      Clear rest of %eax (and rest of %rax)
5     ret
```

Note the comparison order of the `cmpq` instruction (line 2). Although the arguments are listed in the order `%rsi` (`b`), then `%rdi` (`a`), the comparison is really between `a` and `b`. Recall also, as discussed in Section 3.4.2, that the `movzbl` instruction (line 4) clears not just the high-order 3 bytes of `%eax`, but the upper 4 bytes of the entire register, `%rax`, as well.

For some of the underlying machine instructions, there are multiple possible names, which we list as "synonyms." For example, both `setg` (for "set greater") and `setnle` (for "set not less or equal") refer to the same machine instruction. Compilers and disassemblers make arbitrary choices of which names to use.

Although all arithmetic and logical operations set the condition codes, the descriptions of the different SET instructions apply to the case where a comparison instruction has been executed, setting the condition codes according to the computation `t = a-b`. More specifically, let a, b, and t be the integers represented in two's-complement form by variables `a`, `b`, and `t`, respectively, and so $t = a -_w^t b$, where w depends on the sizes associated with `a` and `b`.

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

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

The instruction `jmp .L1` will cause the program to skip over the `movq` instruction and instead resume execution with the `popq` instruction. In generating the object-code file, the assembler determines the addresses of all labeled instructions and encodes the *jump targets* (the addresses of the destination instructions) as part of the jump instructions.

Figure 3.15 shows the different jump instructions. The `jmp` instruction jumps unconditionally. It can be either a *direct* jump, where the jump target is encoded as part of the instruction, or an *indirect* jump, where the jump target is read from a register or a memory location. Direct jumps are written in assembly code by giving a label as the jump target, for example, the label `.L1` in the code shown. Indirect jumps are written using '`*`' followed by an operand specifier using one of the memory operand formats described in Figure 3.3. As examples, the instruction

```
jmp *%rax
```

uses the value in register `%rax` as the jump target, and the instruction

```
jmp *(%rax)
```

reads the jump target from memory, using the value in `%rax` as the read address.

The remaining jump instructions in the table are *conditional*—they either jump or continue executing at the next instruction in the code sequence, depending on some combination of the condition codes. The names of these instructions

both statements in C and instructions in machine code are executed *sequentially*, in the order they appear in the program. The execution order of a set of machine-code instructions can be altered with a *jump* instruction, indicating that control should pass to some other part of the program, possibly contingent on the result of some test. The compiler must generate instruction sequences that build upon this low-level mechanism to implement the control constructs of C.

In our presentation, we first cover the two ways of implementing conditional operations. We then describe methods for presenting loops and `switch` statements.

3.6.1 Condition Codes

In addition to the integer registers, the CPU maintains a set of single-bit *condition code* registers describing attributes of the most recent arithmetic or logical operation. These registers can then be tested to perform conditional branches. These condition codes are the most useful:

`CF`: Carry flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.

`ZF`: Zero flag. The most recent operation yielded zero.

`SF`: Sign flag. The most recent operation yielded a negative value.

`OF`: Overflow flag. The most recent operation caused a two's-complement overflow—either negative or positive.

For example, suppose we used one of the ADD instructions to perform the equivalent of the C assignment `t = a+b`, where variables `a`, `b`, and `t` are integers. Then the condition codes would be set according to the following C expressions:

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

The `leaq` instruction does not alter any condition codes, since it is intended to be used in address computations. Otherwise, all of the instructions listed in Figure 3.10 cause the condition codes to be set. For the logical operations, such as XOR, the carry and overflow flags are set to zero. For the shift operations, the carry flag is set to the last bit shifted out, while the overflow flag is set to zero. For reasons that we will not delve into, the INC and DEC instructions set the overflow and zero flags, but they leave the carry flag unchanged.

In addition to the setting of condition codes by the instructions of Figure 3.10, there are two instruction classes (having 8-, 16-, 32-, and 64-bit forms) that set condition codes without altering any other registers; these are listed in Figure 3.13. The CMP instructions set the condition codes according to the differences of their two operands. They behave in the same way as the SUB instructions, except that they set the condition codes without updating their destinations. With ATT format,

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

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

the operands are listed in reverse order, making the code difficult to read. These instructions set the zero flag if the two operands are equal. The other flags can be used to determine ordering relations between the two operands. The TEST instructions behave in the same manner as the AND instructions, except that they set the condition codes without altering their destinations. Typically, the same operand is repeated (e.g., `testq %rax,%rax` to see whether `%rax` is negative, zero, or positive), or one of the operands is a mask indicating which bits should be tested.

3.6.2 Accessing the Condition Codes

Rather than reading the condition codes directly, there are three common ways of using the condition codes: (1) we can set a single byte to 0 or 1 depending on some combination of the condition codes, (2) we can conditionally jump to some other part of the program, or (3) we can conditionally transfer data. For the first case, the instructions described in Figure 3.14 set a single byte to 0 or to 1 depending on some combination of the condition codes. We refer to this entire class of instructions as the SET instructions; they differ from one another based on which combinations of condition codes they consider, as indicated by the different suffixes for the instruction names. It is important to recognize that the suffixes for these instructions denote different conditions and not different operand sizes. For example, instructions `setl` and `setb` denote "set less" and "set below," not "set long word" or "set byte."

A SET instruction has either one of the low-order single-byte register elements (Figure 3.2) or a single-byte memory location as its destination, setting this byte to either 0 or 1. To generate a 32-bit or 64-bit result, we must also clear the high-order bits. A typical instruction sequence to compute the C expression `a < b`, where `a` and `b` are both of type `long`, proceeds as follows: