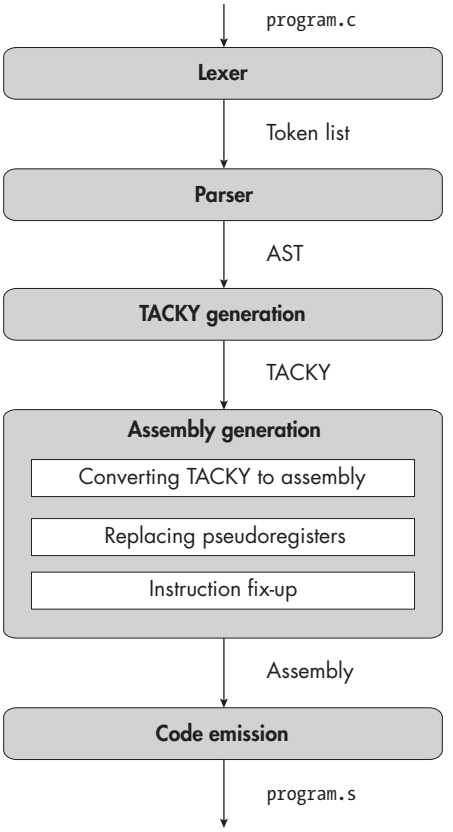
program.c
Lexer
Token list
Parser
AST
TACKY generation
TACKY
Assembly generation
Converting TACKY to assembly
Replacing pseudoregisters
Instruction fix-up
Assembly
Code emission
program.s

4

LOGICAL AND RELATIONAL OPERATORS

Now that you know how to compile binary operators, you're going to add a whole mess of them (plus one more unary operator). In this chapter, you'll add three logical operators: NOT (`!`), AND (`&&`), and OR (`||`). You'll also add the relational operators: `<`, `>`, `==`, and so on. Each of these operators tests some condition, resulting in a value of 1 if that condition is true and 0 if it's false.

The `&&` and `||` operators differ from the binary operators we've seen so far because they *short-circuit*: if you know the result after the first operand, you don't evaluate the second operand. To support short-circuiting logic, we'll add new instructions to TACKY that let us skip over blocks of code. We'll also introduce several new instructions in the assembly generation pass, including conditional assembly instructions that let us take specific actions only if some condition is met.

Let's start with a brief discussion of short-circuiting operators before moving on to the compiler passes.

Short-Circuiting Operators

The C standard guarantees that `&&` and `||` short-circuit when you don't need the second operand. For example, consider the expression `(1 - 1) && foo()`. Because the first operand's value is 0, the whole expression will evaluate to 0 regardless of what `foo` returns, so we won't call `foo` at all. Likewise, if the first operand of `||` is nonzero, we don't evaluate the second operand.

This isn't just a performance optimization; the second operand might not change the result of the expression, but evaluating it can have visible side effects. For example, the `foo` function might perform I/O or update global variables. If your compiler doesn't implement `&&` and `||` as short-circuiting operators, some compiled programs will behave incorrectly. (The standard defines this behavior in section 6.5.13, paragraph 4, for the `&&` operator and in section 6.5.14, paragraph 4, for the `||` operator.)

Now that we've clarified how these operators work, you're ready to continue coding.

The Lexer

In this chapter, you'll add nine new tokens:

- `!` An exclamation point, the logical NOT operator
- `&&` Two ampersands, the logical AND operator
- `||` Two vertical bars, the logical OR operator
- `==` Two equal signs, the "equal to" operator
- `!=` An exclamation point followed by an equal sign, the "not equal to" operator
- `<` The "less than" operator
- `>` The "greater than" operator
- `<=` The "less than or equal to" operator
- `>=` The "greater than or equal to" operator

Your lexer should handle these the same way as the other operators you've added so far. Remember that the lexer should always choose the longest possible match for the next token. For example, if your input is `<=something`, the next token the lexer emits should be `<=`, not `<`.

TEST THE LEXER

Test the lexer with the usual command:

```
$ ./test_compiler /path/to/your_compiler --chapter 4 --stage lex
```

None of this chapter's test programs contain invalid tokens, so your lexer should process all of them without throwing an error.

The Parser

Next, we'll add the new operations to the AST definition. Listing 4-1 shows the updated definition, with these additions bolded.

```
program = Program(function_definition)
function_definition = Function(identifier name, statement body)
statement = Return(exp)
exp = Constant(int)
    | Unary(unary_operator, exp)
    | Binary(binary_operator, exp, exp)
unary_operator = Complement | Negate | Not
binary_operator = Add | Subtract | Multiply | Divide | Remainder | And | Or
                | Equal | NotEqual | LessThan | LessOrEqual
                | GreaterThan | GreaterOrEqual
```

Listing 4-1: The abstract syntax tree with comparison and logical operators

We also need to make the corresponding changes to the grammar, as shown in Listing 4-2.

```
<program> ::= <function>
<function> ::= "int" <identifier> "(" "void" ")" "{" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <factor> | <exp> <binop> <exp>
<factor> ::= <int> | <unop> <factor> | "(" <exp> ")"
<unop> ::= "-" | "~" | "!"
<binop> ::= "-" | "+" | "*" | "/" | "%" | "&&" | "||"
          | "==" | "!=" | "<" | "<=" | ">" | ">="
<identifier> ::= ? An identifier token ?
<int> ::= ? A constant token ?
```

Listing 4-2: The grammar with comparison and logical operators

In Listings 4-1 and 4-2, we've added some new operators, but we haven't made any other changes. Now we're ready to update the parsing code. First,

update `parse_factor` to handle the new `!` operator. It should parse! the same way it parses the unary `~` and `-` operators.

Next, update `parse_exp` to handle the new binary operators. In Chapter 3, we associated every binary operator with a numeric precedence value. Now we'll give the new operators precedence values. These operators have lower precedence than the ones from Chapter 3, and they're all left-associative. Among the new operators, `<`, `<=`, `>`, and `>=` have the highest precedence, followed by the equality operators, `==` and `!=`. The `&&` operator has lower precedence than the equality operators, and `||` has the lowest precedence of all. The precedence values I've chosen are listed in Table 4-1, with new operators bolded.

Table 4-1: Precedence Values of Old and New Binary Operators

Operator	Precedence
*	50
/	50
%	50
+	45
-	45
<	**35**
<=	**35**
>	**35**
>=	**35**
==	**30**
!=	**30**
&&	**10**
||	**5**

These values are spaced far enough apart to leave room for the optional bitwise operators from Chapter 3. There's also room at the bottom of the scale for the `=` and `?:` operators we'll add in the next two chapters. You don't need to use the exact values in this table as long as all operators have the correct precedence relative to each other.

You'll also need to extend the code that converts tokens into `unary_operator` and `binary_operator` AST nodes. For example, the function that converts a `+` token into an `Add` node should also convert a `==` token into an `Equal` node. (The pseudocode in the last two chapters called separate functions, `parse_unop` and `parse_binop`, to handle these conversions.)

Once you've updated your parser's table of precedence values, `parse_binop`, and `parse_unop`, you're done! The precedence climbing algorithm we implemented in the last chapter can handle the new operators without further changes.

TEST THE PARSER

The parser should successfully parse every valid test case in *tests/chapter_4/valid* and raise an error on every invalid test case in *tests/chapter_4/invalid_parse*. To test your parser, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 4 --stage parse
```

TACKY Generation

Now that the lexer and parser are working properly, we can venture into less familiar territory: handling the new operators in TACKY. You can convert relational operators to TACKY in the same way as the binary operators you've already implemented. For example, given the expression `e1 < e2`, the resulting TACKY looks something like Listing 4-3.

```
<instructions for e1>
v1 = <result of e1>
<instructions for e2>
v2 = <result of e2>
Binary(LessThan, v1, v2, result)
```

Listing 4-3: Implementing the < operator in TACKY

You can't generate the `&&` and `||` operators this way, though, because they short-circuit. The code in Listing 4-3 always evaluates both `e1` and `e2`, but we need to generate code that sometimes skips `e2`. To support short-circuiting operators, we'll add an *unconditional jump* instruction, which lets us jump to a different point in the program. We'll also add two *conditional jump* instructions, which jump only when a particular condition is met.

Adding Jumps, Copies, and Comparisons to the TACKY IR

Listing 4-4 shows the latest TACKY IR, including the new jump instructions.

```
program = Program(function_definition)
function_definition = Function(identifier, instruction* body)
instruction = Return(val)
            | Unary(unary_operator, val src, val dst)
            | Binary(binary_operator, val src1, val src2, val dst)
            | Copy(val src, val dst)
            | Jump(identifier target)
            | JumpIfZero(val condition, identifier target)
            | JumpIfNotZero(val condition, identifier target)
            | Label(identifier)
```

```
val = Constant(int) | Var(identifier)
unary_operator = Complement | Negate | Not
binary_operator = Add | Subtract | Multiply | Divide | Remainder | Equal | NotEqual
                | LessThan | LessOrEqual | GreaterThan | GreaterOrEqual
```

Listing 4-4: Adding conditionals, jumps, and labels to TACKY

The `Jump` instruction works just like `goto` in C: it makes the program jump to the point labeled with some identifier, `target`. The `Label` instruction associates an identifier with a location in the program. The snippet of TACKY in Listing 4-5 shows how `Jump` and `Label` instructions work together.

```
Unary(Negate, Constant(1), Var("tmp"))
Jump("there")
❶ Unary(Negate, Constant(2), Var("tmp"))
Label("there")
Return(Var("tmp"))
```

Listing 4-5: A snippet of TACKY with a `Jump` instruction

This program stores `-1` in `tmp`, then executes the `Jump` instruction, which jumps to the `Label` instruction. Next, it executes the `Return` instruction, which returns `-1`. The second `Unary` instruction ❶ won't execute at all, because we jumped over it.

The first conditional jump in the TACKY IR, `JumpIfZero`, jumps to the instruction indicated by `target` if the value of `condition` is 0. If `condition` is anything other than 0, we don't jump to `target`; instead, we execute the next instruction as usual. The second conditional jump, `JumpIfNotZero`, does the opposite: we jump to `target` only if `condition` isn't 0. We don't really need both of these instructions, since any behavior you can express with one can be expressed with the other plus a `Not` instruction. But adding both lets us generate simpler TACKY for the `&&` and `||` operations, which will ultimately translate into simpler, shorter assembly.

The other new instruction is `Copy`. Since `&&` and `||` ultimately return 1 or 0, we use this instruction to copy 1 or 0 into the temporary variable that holds the result of the expression.

Besides these five additional instructions, the latest TACKY IR includes the new relational and logical binary operators and the unary `Not` operator.

Converting Short-Circuiting Operators to TACKY

Let's use the new TACKY instructions to implement the `&&` and `||` operators. The TACKY for the expression `e1 && e2` should look like Listing 4-6.

```
<instructions for e1>
v1 = <result of e1>
JumpIfZero(v1, false_label)
<instructions for e2>
v2 = <result of e2>
```

```
JumpIfZero(v2, false_label)
result = 1
Jump(end)
Label(false_label)
result = 0
Label(end)
```

Listing 4-6: Implementing the && operator in TACKY

We start by evaluating `e1`. If it's 0, we short-circuit and set `result` to `0`, without evaluating `e2`. We accomplish this with the `JumpIfZero` instruction; if `v1` is 0, we jump straight to `false_label`, then set `result` to `0` with the `Copy` instruction. (I've written this out as `result = 0` instead of `Copy(0, result)` to make it more readable. I'll take similar liberties with TACKY notation in later chapters.) If `v1` isn't 0, we still need to evaluate `e2`. We handle the case where `v2` is 0 exactly like the case where `v1` is 0, by jumping to `false_label` with `JumpIfZero`. We reach the second `Copy` instruction, `result = 1`, only if we didn't take either conditional jump. That means both `e1` and `e2` are non-zero, so we set `result` to `1`. Then, we jump over `result = 0` to the `end` label to avoid overwriting `result`.

I'll leave it to you to translate the `||` operation to TACKY on your own. The resulting TACKY will look similar to Listing 4-6, but it will use the `JumpIfNotZero` instruction instead of `JumpIfZero`. That leaves `!` and all the relational operations; you can convert them to TACKY in the same way as the unary and binary operations you added in the previous chapters.

Generating Labels

Labels, like temporary variables, must be globally unique: an instruction like `Jump("foo")` is useless if the label `foo` shows up in multiple places. You can make sure they're unique by incorporating a global counter into labels, like you did with variable names in Chapter 2.

Unlike temporary variables, labels will appear in the final assembly program, so they must be identifiers that the assembler considers syntactically valid. They should contain only letters, digits, periods, and underscores. Choose descriptive labels to make your assembly programs easier to read and debug. For example, you could use the string `and_falseN` as `false_label` in Listing 4-6, where `N` is the current value of a global counter.

Although labels must not conflict with each other, it's okay for them to conflict with temporary variable names. It's also okay if the labels you generate here conflict with user-defined function names, even though both autogenerated labels and function names become labels in the final assembly program. We'll mangle our autogenerated labels during code emission so they don't conflict with user-defined identifiers.

TEST THE TACKY GENERATION STAGE

To test out TACKY generation, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 4 --stage tacky
```

Comparisons and Jumps in Assembly

Before starting on the assembly generation pass, let's talk through the new assembly instructions we'll need. First, we'll discuss the `cmp` instruction, which compares two values, and the *conditional set* instructions, which set a byte to 1 or 0 based on the result of a comparison. We'll use these to implement relational operators like <. Next, we'll talk about conditional and unconditional jump instructions.

Comparisons and Status Flags

The "condition" that all conditional instructions depend on is the state of the RFLAGS register. Unlike EAX, RSP, and the other registers we've seen so far, we usually can't directly set RFLAGS. Instead, the CPU updates RFLAGS automatically every time it issues an instruction. As the name suggests, each bit in this register is a flag that reports some fact about the last instruction or the status of the CPU. Different instructions update different flags: the `add`, `sub`, and `cmp` instructions update all the flags we'll talk about in this section, and the `mov` instruction doesn't update any of them. We can ignore the effects of other instructions for now. Whenever I refer to the "last instruction" or "last result" while discussing RFLAGS, I mean the last instruction that affects the particular flag I'm talking about.

Right now, we care about three of these flags:

Zero flag (ZF)

ZF is set to 1 if the result of the last instruction was 0. It's set to 0 if the result of the last instruction was nonzero.

Sign flag (SF)

SF is set to 1 if the most significant bit of the last result was 1. It's set to 0 if the most significant bit of that result was 0. Remember that in two's complement, the most significant bit of a negative number is always 1, and the most significant bit of a positive number is always 0. Therefore, the sign flag tells us whether the result of the last instruction was positive or negative. (If the last result should be interpreted as an unsigned integer, it can't be negative, so the sign flag is meaningless.)

Overflow flag (OF)

OF is set to 1 if the last instruction resulted in a signed integer overflow, and 0 otherwise. An *integer overflow* occurs when the result of a signed

integer operation can't be represented in the number of bits available. A positive result overflows when it's larger than the maximum value the type can hold. Suppose we're operating on 4-bit integers. The largest signed number we can represent is 7, or `0111` in binary. If we add one to it with the `add` instruction, the result is `1000`. If we interpret this as an unsigned integer, its value is 8, but its value is –8 if we interpret it as a two's complement signed integer. The result of the computation should be positive, but since it overflowed, it appears negative. This computation sets the overflow flag to 1.

We also encounter integer overflow in the opposite situation: when the result should be negative, but it's below the smallest possible value. For example, in ordinary math, –8 – 1 = –9. But if we use the `sub` instruction to subtract one from the 4-bit two's complement representation of –8, which is `1000`, we end up with `0111`, or 7. The overflow flag is set to 1 in this case too.

An unsigned result can also be too large or small for its type to represent, but I won't refer to this as integer overflow in this book. Instead, I say the result *wrapped around*, which is more consistent with the terminology for unsigned operations in the C standard and in most discussions of x64 assembly. I draw this distinction because unsigned wraparound follows different rules from signed integer overflow in the C standard, and the CPU detects it differently. You'll learn how to handle unsigned wraparound in Part II. Like SF, OF is meaningless if the result is unsigned.

Tables 4-2 and 4-3 summarize the cases where each kind of integer overflow is possible. Table 4-2 describes the results of addition.

Table 4-2: Integer Overflow and Underflow from Addition

`a + b`	`b > 0`	`b < 0`
`a > 0`	Overflow from positive to negative	Neither
`a < 0`	Neither	Overflow from negative to positive

Table 4-3 describes the results of subtraction; it's just Table 4-2 with the columns swapped, since `a - b` and `a + (- b)` are equivalent.

Table 4-3: Integer Overflow and Underflow from Subtraction

`a - b`	`b > 0`	`b < 0`
`a > 0`	Neither	Overflow from positive to negative
`a < 0`	Overflow from negative to positive	Neither

The instruction `cmp b, a` computes `a - b`, exactly like the `sub` instruction, and has the same impact on RFLAGS, but it discards the result instead of storing it in `a`. This is more convenient when you want to subtract two numbers only in order to compare them and don't want to overwrite `a`.

Let's figure out the values of ZF and SF after the instruction `cmp b, a`:

- If `a == b`, then `a - b` is 0, so ZF is 1 and SF is 0.
- If `a > b`, then `a - b` is a positive number, so both SF and ZF are 0.
- If `a < b`, then `a - b` is a negative number, so SF is 1 and ZF is 0.

By issuing a `cmp` instruction and then checking ZF and SF, you can handle every comparison we're implementing in this chapter. But wait! That's not quite true, because `a - b` could overflow, which would flip SF. Let's consider how that impacts each case:

- If `a == b`, then `a - b` can't overflow because it's 0.
- If `a > b`, then `a - b` could overflow when `a` is positive and `b` is negative. The correct result in this case is positive, but if it overflows, the result will be negative. In that case, SF will be 1, and OF will be too.
- If `a < b`, then `a - b` could overflow when `a` is negative and `b` is positive. In this case, the correct result is negative, but the actual result will be positive. That means SF will be 0, but OF will be 1.

Table 4-4 gives the values of these flags in every case we've considered.

Table 4-4: Impact of `cmp` Instruction on Status Flags

	ZF	OF	SF
`a == b`	1	0	0
`a > b`, *no overflow*	0	0	0
`a > b`, *overflow*	0	1	1
`a < b`, *no overflow*	0	0	1
`a < b`, *overflow*	0	1	0

You can tell whether `a` or `b` is larger by checking whether SF and OF are the same. If they are, we know that `a ≥ b`. Either both are 0, because we got a positive (or 0) result with no overflow, or both are 1, because we got a large positive result that overflowed until it became negative. If SF and OF are different, we know that `a < b`. Either we got a negative result with no overflow, or we got a negative result that overflowed and became positive.

UNDEFINED BEHAVIOR ALERT!

If the `add` and `sub` instructions can overflow, why didn't we account for that in Chapter 3? We didn't need to because integer overflow in C is *undefined behavior*, where the standard doesn't tell you what should happen. Compilers are permitted to handle undefined behavior however they want—or not handle it at all.

When an expression in C overflows, for example, the result *usually* wraps around like the examples we saw earlier. However, it's equally acceptable for the program to generate a result at random, raise a signal, or erase your hard drive. That last option may sound unlikely, but production compilers really do handle undefined behavior in surprising (and arguably undesirable) ways. Take the following program:

```
#include <stdio.h>

int main(void) {
    for (int i = 2147483646; i > 0; i = i + 1)
        printf("The number is %d\n", i);
    return 0;
}
```

The largest value an `int` can hold is 2,147,483,647, so the expression `i + 1` overflows the second time we execute it. When the `add` assembly instruction overflows, it produces a negative result, so we might expect this loop to execute twice, then stop because the condition `i > 0` no longer holds. That's exactly what happens if you compile this program without optimizations, at least with the versions of Clang and GCC that I tried:

```
$ clang overflow.c
$ ./a.out
The number is 2147483646
The number is 2147483647
```

But if you enable optimizations, the behavior might change completely:

```
$ clang -O overflow.c
$ ./a.out
The number is 2147483646
The number is 2147483647
The number is -2147483648
The number is -2147483647
The number is -2147483646
The number is -2147483645
--snip--
```

What happened? The compiler tried to optimize the program by removing conditional checks that always succeed. The initial value of `i` is positive, and it's updated only in the expression `i = i + 1`, so the compiler concluded that the condition `i > 0` is always true. That's correct, as long as `i` doesn't overflow. It's incorrect if `i` does overflow, of course, but that's undefined behavior, so the compiler didn't have to account for it. It therefore removed the condition entirely, resulting in a loop that never terminates.

I used Clang for this example because GCC produced completely different, even less intuitive behavior. You may well see different results if you

compile this program on your own machine. Try it out with a few different optimization levels, and see what happens.

Note that setting the overflow flag in assembly doesn't necessarily indicate overflow in the source program. For example, when we implement an expression like `a < 10` with `cmp`, that `cmp` instruction may set the overflow flag. But the result of `a < 10` is either 0 or 1—both of which are in the range of `int`—so the expression itself does not overflow. This expression won't produce undefined behavior, regardless of how we implement it in assembly.

C has a bunch of different kinds of undefined behavior; integer overflow is just one example. It's a particularly ugly example, though, because it's difficult to avoid and can have dire consequences, including security vulnerabilities. To address this long-standing problem, the next version of the C standard, C23, adds a few standard library functions that perform checked integer operations. If you use the new `ckd_add`, `ckd_sub`, and `ckd_mul` functions instead of the `+`, `-`, and `*` operators, you'll get an informative return code instead of undefined behavior when the result is out of bounds. To learn more about these new library functions, see Jens Gustedt's blog post titled "Checked Integer Arithmetic in the Prospect of C23" (*https://gustedt.wordpress.com/2022/12/18/checked-integer-arithmetic-in-the-prospect-of-c23*).

Undefined behavior is different from *unspecified behavior*. If some aspect of a program's behavior is unspecified, there are several possible ways it could behave, but it can't behave totally unpredictably. For example, in Chapter 3, we learned that the operands in a binary expression are unsequenced (or indeterminately sequenced, if either is a function call), so their evaluation order is unspecified. This doesn't mean the expression's behavior is undefined. When we evaluate the expression `printf("Hello, ") + printf("World!")`, the program can print either `"Hello, "` or `"World!"` first, but it can't go off and do something else entirely. Unsequenced operations *can* produce undefined behavior under certain circumstances—say, if you perform two unsequenced updates to the same variable—but performing unsequenced or indeterminately sequenced operations is not an undefined behavior in and of itself.

Unspecified behavior is a normal part of any C program. It's a problem only if your program relies on a particular behavior that the standard doesn't specify, like in the `Hello, World!` example. Undefined behavior, on the other hand, is always a problem; if it occurs anywhere in your program, you can't count on any part of the program to work correctly.

To learn more about undefined behavior, and the trail of chaos and destruction it leaves in its wake, explore the references in "Additional Resources" on page XX.

Now that you understand how to set ZF, OF, and SF, let's take a look at a few instructions that depend on those flags.

Conditional Set Instructions

To implement a relational operator, we first set some flags using the `cmp` instruction, then set the result of the expression based on those flags. We perform that second step with a *conditional set* instruction. Each conditional set instruction takes a single register or memory address as an operand, which it sets to 0 or 1 based on the state of RFLAGS. The conditional set instructions are all identical, except that they test for different conditions. Table 4-5 lists the conditional set instructions we need in this chapter.

Table 4-5: Conditional Set Instructions

Instruction	Meaning	Flags
`sete`	Set byte if a == b	ZF set
`setne`	Set byte if a != b	ZF not set
`setg`	Set byte if a > b	ZF not set and SF == OF
`setge`	Set byte if a ≥ b	SF == OF
`setl`	Set byte if a < b	SF != OF
`setle`	Set byte if a ≤ b	ZF set or SF != OF

Unlike the other instructions we've seen so far, conditional set instructions take only 1-byte operands. For example, `sete %eax` is not a valid instruction, because EAX is a 4-byte register. The instruction `sete %al`, however, is valid; this sets the AL register, the least significant byte of EAX. To conditionally set the whole EAX register to 0 or 1, you need to zero out EAX before you set AL, because the conditional set instruction won't clear its upper bytes. For example, if EAX is

```
11111111111111111111111111111011
```

and you run

```
movl    $2, %edx
cmpl    $1, %edx
sete    %al
```

then the new value in EAX is

```
11111111111111111111111100000000
```

which is, of course, not 0. The `sete` instruction zeroed out the last byte of EAX, but not the rest of it.

If its operand is a memory address, a conditional set instruction updates the single byte at that address. Note that a memory address can be a 1-byte, 4-byte, or 8-byte operand, depending on context. In `sete -4(%rbp)`,

the operand `-4(%rbp)` indicates a single byte of memory at RBP – 4; in `addl $1, -4(%rbp)`, it indicates the 4 bytes of memory starting at RBP – 4.

Jump Instructions

The `jmp` assembly instruction takes a label as an argument and performs an unconditional jump to that label. Jump assembly instructions manipulate another special-purpose register, RIP, which always holds the address of the next instruction to execute (IP stands for *instruction pointer*). To execute a sequence of instructions, the CPU carries out the *fetch-execute cycle*:

1. Fetch an instruction from the memory address in RIP and store it in a special-purpose *instruction register*. (This register doesn't have a name because you can't refer to it in assembly.)
2. Increment RIP to point to the next instruction. Instructions in x64 aren't all the same length, so the CPU has to check the length of the instruction it just fetched and increment RIP by that many bytes.
3. Run the instruction in the instruction register.
4. Repeat.

Normally, following these steps executes instructions in the order they appear in memory. But `jmp` puts a new value in RIP, which changes what instruction the CPU executes next. The assembler and linker convert the label in a jump instruction into a *relative offset* that tells you how much to increment or decrement RIP. Consider the snippet of assembly in Listing 4-7.

```
    addl    $1, %eax
    jmp     foo
    movl    $0, %eax
foo:
    ret
```

Listing 4-7: A snippet of assembly code with a `jmp` instruction

The machine instruction for `movl $0, %eax` is 5 bytes long. To jump over it and execute the `ret` instruction instead, `jmp` needs to increment RIP by an extra 5 bytes. The assembler and linker therefore convert `jmp foo` into the machine instruction for `jmp 5`. Then, when the CPU executes this instruction, it:

1. Fetches the instruction `jmp 5` and stores it in the instruction register.
2. Increments RIP to point to the next instruction, `movl $0, %eax`.
3. Executes `jmp 5`. This adds 5 bytes to RIP, so that it points to `ret`.
4. Fetches the instruction RIP points to, `ret`, and continues the fetch-execute cycle from there.

Note that labels aren't instructions: the CPU doesn't execute them, and they don't appear in the text section of the final executable (the section that contains machine instructions).

A *conditional jump* takes a label as an argument but jumps to that label only if the condition holds. Conditional jumps look a lot like conditional set instructions; they depend on the same conditions, using the same flags in RFLAGS. For example, the assembly in Listing 4-8 returns `3` if the EAX and EDX registers are equal, and `0` otherwise.

```
    cmpl    %eax, %edx
    je      return3
    movl    $0, %eax
    ret
return3:
    movl    $3, %eax
    ret
```

Listing 4-8: A snippet of assembly code with a conditional jump

If the values in EAX and EDX are equal, `cmpl` sets ZF to 1, so `je` jumps to `return3`. Then, the two instructions following `return3` execute, so the function returns `3`. If EAX and EDX aren't equal, `je` doesn't perform the jump, so the function returns `0`. Similarly, `jne` jumps only if ZF is 0. There are also jump instructions that check other conditions, but we don't need them in this chapter.

Assembly Generation

Now that you understand the new assembly instructions you'll need, let's extend the assembly AST and update each assembly generation pass. Listing 4-9 defines the latest assembly AST, with additions bolded.

```
program = Program(function_definition)
function_definition = Function(identifier name, instruction* instructions)
instruction = Mov(operand src, operand dst)
            | Unary(unary_operator, operand)
            | Binary(binary_operator, operand, operand)
            | Cmp(operand, operand)
            | Idiv(operand)
            | Cdq
            | Jmp(identifier)
            | JmpCC(cond_code, identifier)
            | SetCC(cond_code, operand)
            | Label(identifier)
            | AllocateStack(int)
            | Ret
unary_operator = Neg | Not
binary_operator = Add | Sub | Mult
```

```
operand = Imm(int) | Reg(reg) | Pseudo(identifier) | Stack(int)
cond_code = E | NE | G | GE | L | LE
reg = AX | DX | R10 | R11
```

Listing 4-9: The assembly AST with comparisons and conditional instructions

Since all conditional jump instructions have the same form, we represent them with a single `JmpCC` instruction and distinguish between them using different condition codes. We do the same with conditional set instructions. We also treat labels like instructions at this stage, even though `Label` isn't really an instruction since labels aren't executed by the CPU.

To implement the TACKY `JumpIfZero` and `JumpIfNotZero` instructions, we use the new `JmpCC` assembly instruction. We convert

```
JumpIfZero(val, target)
```

to:

```
Cmp(Imm(0), val)
JmpCC(E, target)
```

We implement `JumpIfNotZero` the same way, but with `NE` instead of `E` as the condition code.

Similarly, we implement all the relational operators using conditional set instructions. For example, the TACKY instruction

```
Binary(GreaterThan, src1, src2, dst)
```

becomes:

```
Cmp(src2, src1)
Mov(Imm(0), dst)
SetCC(G, dst)
```

For all the other relational operators, replace `G` with the appropriate condition code. Remember to zero out the destination before the conditional set instruction, since it sets only the lowest byte. It's safe to perform a `mov` right after the `cmp` instruction because `mov` doesn't change RFLAGS. One potential wrinkle is that `SetCC` needs a 1-byte operand, but `dst` is 4 bytes; luckily, we can account for this in the code emission pass. If `dst` is a location in memory, `SetCC` sets the first byte at that location, which is the behavior we want. (Because x64 processors are *little-endian*, the first byte is the least significant, so setting that byte to 1 sets the whole 32-bit value to 1.) If `dst` is a register, we'll use the corresponding 1-byte register name when we emit `SetCC` during code emission. Registers in the assembly AST are size-agnostic,

so for now we represent `dst` the same way whether we're using it as a 4-byte or 1-byte operand.

Because `!x` is equivalent to `x == 0`, we also implement the unary `!` operator with a conditional set instruction. We convert the TACKY instruction

```
Unary(Not, src, dst)
```

into:

```
Cmp(Imm(0), src)
Mov(Imm(0), dst)
SetCC(E, dst)
```

The remaining TACKY instructions—`Jump`, `Label`, and `Copy`—are easy. A TACKY `Jump` becomes an assembly `Jmp`, `Label` becomes `Label`, and `Copy` becomes `Mov`. Tables 4-6 and 4-7 summarize how to convert each new TACKY construct to assembly. Note that these tables include only new constructs, unlike the equivalent tables in Chapters 2 and 3.

Table 4-6 shows how to convert the new `Copy`, `Label`, and conditional and unconditional jump instructions to assembly, as well as `Unary` instructions with the new `Not` operator and `Binary` instructions with the new relational operators.

Table 4-6: Converting TACKY Instructions to Assembly

TACKY instruction	Assembly instructions

`Unary(Not, src, dst)`	`Cmp(Imm(0), src)`
	`Mov(Imm(0), dst)`
	`SetCC(E, dst)`

`Binary(relational_operator, src1, src2, dst)`	`Cmp(src2, src1)`
	`Mov(Imm(0), dst)`
	`SetCC(relational_operator, dst)`

`Jump(target)`	`Jmp(target)`
`JumpIfZero(condition, target)`	`Cmp(Imm(0), condition)` `JmpCC(E, target)`
`JumpIfNotZero(condition, target)`	`Cmp(Imm(0), condition)` `JmpCC(NE, target)`
`Copy(src, dst)`	`Mov(src, dst)`
`Label(identifier)`	`Label(identifier)`

Table 4-7 gives the corresponding condition code for each relational operator in TACKY.

Table 4-7: Converting TACKY Comparisons to Assembly

TACKY comparison	Assembly condition code
`Equal`	`E`
`NotEqual`	`NE`
`LessThan`	`L`
`LessOrEqual`	`LE`
`GreaterThan`	`G`
`GreaterOrEqual`	`GE`

From now on, the tables describing each chapter's conversion from TACKY to assembly will show only what's changed from the chapter before. Appendix B includes two sets of tables giving the complete conversion from TACKY to assembly: one shows the conversion at the end of Part I, and the other shows the conversion at the end of Part II.

Replacing Pseudoregisters

Update this pass to replace any pseudoregisters used by the new `Cmp` and `SetCC` instructions with stack addresses, just like you did for all the other instructions.

Fixing Up the cmp Instruction

The `cmp` instruction, much like `mov`, `add`, and `sub`, can't use memory addresses for both operands. We rewrite it in the usual way, turning

```
cmpl    -4(%rbp), -8(%rbp)
```

into:

```
movl    -4(%rbp), %r10d
cmpl    %r10d, -8(%rbp)
```

The second operand of a `cmp` instruction can't be a constant. This sort of makes sense if you remember that `cmp` follows the same form as `sub`; the second operand of a `sub`, `add`, or `imul` instruction can't be a constant either, since that operand holds the result. Even though `cmp` doesn't produce a result, the same rules apply. We rewrite

```
cmpl    %eax, $5
```

as:

```
movl    $5, %r11d
cmpl    %eax, %r11d
```

Following the convention from the previous chapter, we use R10 to fix a `cmp` instruction's first operand and R11 to fix its second operand.

TEST THE ASSEMBLY GENERATION STAGE

To test the assembly generation stage, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 4 --stage codegen
```

Code Emission

We've generated a valid assembly program, and we're ready to emit it. Code emission is slightly more complicated in this chapter, for two reasons. First, we're dealing with both 1-byte and 4-byte registers. We'll print out a different name for a register depending on whether it appears in a conditional set instruction, which takes 1-byte operands, or any of the other instructions we've encountered so far, which take 8-byte operands.

The second issue is emitting labels. Some assembly labels are autogenerated by the compiler, while others—function names—are user-defined identifiers. Right now, the only function name is `main`, but eventually we'll compile programs with arbitrary function names. Because labels must be unique, autogenerated labels must not conflict with any function names that could appear in a program.

We'll avoid conflicts by adding a special *local label* prefix to our autogenerated labels. The local label prefix is `.L` on Linux and `L` on macOS. On Linux, these labels won't conflict with user-defined identifiers because identifiers in C can't contain periods. On macOS, they won't conflict because we prefix all user-defined names with underscores (so that `main` becomes `_main`, for example).

Local labels are handy for another reason: they won't confuse GDB or LLDB when you need to debug this code. The assembler puts most labels in the object file's symbol table, but it leaves out any that start with the local label prefix. If your autogenerated labels were in the symbol table, GDB and LLDB would mistake them for function names, which would cause problems when you tried to disassemble a function or view a stack trace.

Aside from those two issues, code emission is pretty straightforward. Tables 4-8 through 4-10 summarize the changes to this pass. From this point forward, the code emission tables will show only what's changed from the previous chapter, much like the tables describing the conversion from TACKY to assembly. See Appendix B for a complete overview of the code emission pass; it includes three sets of tables showing how this pass will look at the end of Part I, Part II, and Part III.

Table 4-8 shows how to print out this chapter's new assembly instructions. It uses the `.L` prefix for local labels; if you're on macOS, use an `L` prefix without a period instead.

Table 4-8: Formatting Assembly Instructions

Assembly instruction	Output
`Cmp(operand, operand)`	`cmpl` *`<operand>, <operand>`*
`Jmp(label)`	`jmp .L`*`<label>`*
`JmpCC(cond_code, label)`	`j`*`<cond_code>`* `.L`*`<label>`*
`SetCC(cond_code, operand)`	`set`*`<cond_code>`* *`<operand>`*
`Label(label)`	`.L`*`<label>`*`:`

The `cmp` instruction gets an `l` suffix to indicate that it operates on 4-byte values. Conditional set instructions don't take a suffix to indicate the operand size, because they support only 1-byte operands. Jumps and labels also don't use operand size suffixes, since they don't take operands. However, conditional jump and set instructions do need suffixes to indicate what condition they test. Table 4-9 gives the corresponding suffix for each condition code.

Table 4-9: Instruction Suffixes for Condition Codes

Condition code	Instruction suffix
`E`	`e`
`NE`	`ne`
`L`	`l`
`LE`	`le`
`G`	`g`
`GE`	`ge`

Finally, Table 4-10 gives the 1-byte and 4-byte aliases for each register. The 4-byte aliases are the same as in the previous chapter; the new 1-byte aliases are bolded.

Table 4-10: Formatting Assembly Operands

Assembly operand		Output
`Reg(AX)`	4-byte **1-byte**	`%eax` **`%al`**
`Reg(DX)`	4-byte **1-byte**	`%edx` **`%dl`**
`Reg(R10)`	4-byte **1-byte**	`%r10d` **`%r10b`**
`Reg(R11)`	4-byte **1-byte**	`%r11d` **`%r11b`**

Emit the 1-byte names for registers when they appear in `SetCC` and the 4-byte names anywhere else.

TEST THE WHOLE COMPILER

To check that you're compiling every test program correctly, run:

```
$ ./test_compiler /path/to/your_compiler --chapter 4
```

Once all the tests pass, you're ready to move on to the next chapter.

Summary

Your compiler can now handle relational and logical operators. In this chapter, you added conditional jumps to TACKY to support short-circuiting operators, and you learned about several new assembly instructions. You also learned how the CPU keeps track of the current instruction and records the results of comparisons. The new TACKY and assembly instructions you introduced in this chapter will eventually help you implement complex control structures like `if` statements and loops. But first, you'll implement one of the most essential features of C: variables!

Additional Resources

For more in-depth discussions of undefined behavior, see these blog posts:

- "A Guide to Undefined Behavior in C and C++, Part 1" by John Regehr is a good overview of what undefined behavior means in the C standard and how it impacts compiler design (*https://blog.regehr.org/archives/213*).
- "With Undefined Behavior, Anything Is Possible" by Raph Levien explores some sources of undefined behavior in C and the history of how it got into the standard to begin with (*https://raphlinus.github.io/programming/rust/2018/08/17/undefined-behavior.html*).