3.6.5 Implementing Conditional Branches with Conditional Control

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

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

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

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

If (test-expr)

Then-statement

Else

Else-statement

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

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

t = test-expr;

if (!t)

goto false;

then-statement

goto done;

false:

else-statement

done;

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

3.6.6 Implementing Conditional Branches with Conditional Moves

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

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

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

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

As an example, we ran timings of the absdiff function on an Intel Haswell processor using both methods of implementing the conditional operation. In a typical application, the outcome of the test x < y is highly unpredictable, and so even the most sophisticated branch prediction hardware will guess correctly only around 50% of the time. In addition, the computations performed in each of the two code sequences require only a single clock cycle. As a consequence, the branch misprediction penalty dominates the performance of this function. For X86-64 code with conditional jumps, we found that the function requires around 8 clock cycles per call when the branching pattern is easily predictable, and around 17.50 clock cycles per call when the branching pattern is random. From this, we can infer that the branch misprediction penalty is around 19 clock cycles. That means time required by the function ranges between around 8 and 27 cycles, depending on whether or not the branch is predicted correctly.

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

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

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

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

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

Figure 3.18 The conditional move instructions. These instructions copy the source value S to its destination R when the move condition holds. Some instructions have “synonyms”, alternate names for the same machine instruction.

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

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