CHAPTER 2

BASICS

2–1 Manipulating Rightmost Bits

Some of the formulas in this section find application in later chapters.

Use the following formula to turn off the rightmost 1-bit in a word, producing 0 if none (e.g., 01011000 ⇒ 01010000):

$$\boldsymbol{x} \mathbin{\&} (\boldsymbol{x} - \mathbf{1})$$

This may be used to determine if an unsigned integer is a power of 2; apply the formula followed by a 0-test on the result.

Similarly, the following formula can be used to test if an unsigned integer is of the form $2^n - 1$ (including 0 or all 1's):

$$\boldsymbol{x} \mathbin{\&} (\boldsymbol{x} + \mathbf{1})$$

Use the following formula to isolate the rightmost 1-bit, producing 0 if none (e.g., 01011000 ⇒ 00001000):

$$\boldsymbol{x} \mathbin{\&} (-\boldsymbol{x})$$

Use the following formula to isolate the rightmost 0-bit, producing 0 if none (e.g., 10100111 ⇒ 00001000):

$$\neg\boldsymbol{x} \mathbin{\&} (\boldsymbol{x} + \mathbf{1})$$

Use one of the following formulas to form a mask that identifies the trailing 0's, producing all 1's if $\boldsymbol{x} = \mathbf{0}$ (e.g., 01011000 ⇒ 00000111):

$$\begin{gathered}\neg\boldsymbol{x} \mathbin{\&} (\boldsymbol{x} - \mathbf{1}), \quad \text{or} \\ \neg(\boldsymbol{x} \mid -\boldsymbol{x}), \quad \text{or} \\ (\boldsymbol{x} \mathbin{\&} -\boldsymbol{x}) - \mathbf{1}\end{gathered}$$

The first formula has a little instruction-level parallelism.

Use the following formula to form a mask that identifies the rightmost 1-bit and the trailing 0's, producing all 1's if $\boldsymbol{x} = \mathbf{0}$ (e.g., 01011000 ⇒ 00001111):

$$\boldsymbol{x} \oplus (\boldsymbol{x} - \mathbf{1})$$

Use the following formula to right-propagate the rightmost 1-bit, producing all 1's if $\mathbf{x} = \mathbf{0}$ (e.g., 01011000 ⇒ 01011111):

$$\mathbf{x} \mid (\mathbf{x} - \mathbf{1})$$

Use the following formula to turn off the rightmost contiguous string of 1-bits (e.g., 01011000 ⇒ 01000000):

$$((\mathbf{x} \mid (\mathbf{x} - \mathbf{1})) + \mathbf{1})\ \&\ \mathbf{x}$$

This may be used to see if a nonnegative integer is of the form $2^j - 2^k$ for some $j \geq k \geq 0$; apply the formula followed by a 0-test of the result.

These formulas all have duals in the following sense. Read what the formula does, interchanging 1's and 0's in the description. Then, in the formula, replace $\mathbf{x} - \mathbf{1}$ with $\mathbf{x} + \mathbf{1}$, $\mathbf{x} + \mathbf{1}$ with $\mathbf{x} - \mathbf{1}$, $-\mathbf{x}$ with $\neg(\mathbf{x} + \mathbf{1})$, & with |, and | with &. Leave $\mathbf{x}$ and $\neg\mathbf{x}$ alone. Then the result is a valid description and formula. For example, the dual of the first formula in this section reads as follows:

Use the following formula to turn on the rightmost 0-bit in a word, producing all 1's if none (e.g., 10100111 ⇒ 10101111):

$$\mathbf{x} \mid (\mathbf{x} + \mathbf{1})$$

There is a simple test to determine whether or not a given function can be implemented with a sequence of *add*'s, *subtract*'s, *and*'s, *or*'s, and *not*'s [War]. We may, of course, expand the list with other instructions that can be composed from the basic list, such as *shift left* by a fixed amount (which is equivalent to a sequence of *add*'s), or *multiply.* However, we exclude instructions that cannot be composed from the list. The test is contained in the following theorem.

> THEOREM. *A function mapping words to words can be implemented with word-parallel add, subtract, and, or, and not instructions if and only if each bit of the result depends only on bits at and to the right of each input operand.*

That is, imagine trying to compute the rightmost bit of the result by looking only at the rightmost bit of each input operand. Then, try to compute the next bit to the left by looking only at the rightmost two bits of each input operand, and so forth. If you are successful in this, then the function can be computed with a sequence of *add*'s, *and*'s, and so on. If the function cannot be computed in this right-to-left manner, then it cannot be implemented with a sequence of such instructions.

The interesting part of this is the latter statement, and it is simply the contrapositive of the observation that the functions *add, subtract, and, or,* and *not* can all be computed in the right-to-left manner, so any combination of them must have this property.

To see the “if” part of the theorem, we need a construction that is a little awkward to explain. We illustrate it with a specific example. Suppose that a function of two variables *x* and *y* has the right-to-left computability property, and suppose that bit 2 of the result *r* is given by

$$r_2 = x_2 \mid (x_0 \,\&\, y_1). \tag{1}$$

We number bits from right to left, 0 to 31. Because bit 2 of the result is a function of bits at and to the right of bit 2 of the input operands, bit 2 of the result is “right-to-left computable.”

Arrange the computer words ***x***, ***x*** shifted left two, and ***y*** shifted left one, as shown below. Also, add a mask that isolates bit 2.

$$\begin{array}{ccccccc}
x_{31} & x_{30} & \ldots & x_3 & x_2 & x_1 & x_0 \\
x_{29} & x_{28} & \ldots & x_1 & x_0 & 0 & 0 \\
y_{30} & y_{29} & \ldots & y_2 & y_1 & y_0 & 0 \\
0 & 0 & \ldots & 0 & 1 & 0 & 0 \\
0 & 0 & \ldots & 0 & r_2 & 0 & 0
\end{array}$$

Now, form the word-parallel *and* of lines 2 and 3, *or* the result with row 1 (following Equation (1)), and *and* the result with the mask (row 4 above). The result is a word of all 0’s except for the desired result bit in position 2. Perform similar computations for the other bits of the result, *or* the 32 resulting words together, and the result is the desired function.

This construction does not yield an efficient program; rather, it merely shows that it can be done with instructions in the basic list.

Using the theorem, we immediately see that there is no sequence of such instructions that turns off the leftmost 1-bit in a word, because to see if a certain 1-bit should be turned off, we must look to the left to see if it is the leftmost one. Similarly, there can be no such sequence for performing a right shift, or a rotate shift, or a left shift by a variable amount, or for counting the number of trailing 0’s in a word (to count trailing 0’s, the rightmost bit of the result will be 1 if there are an odd number of trailing 0’s, and we must look to the left of the rightmost position to determine that).

A novel application of the sort of bit twiddling discussed above is the problem of finding the next higher number after a given number that has the same number of 1-bits. You are forgiven if you are asking, “Why on earth would anyone want to compute that?” It has application where bit strings are used to represent subsets. The possible members of a set are listed in a linear array, and a subset is represented by a word or sequence of words in which bit *i* is on if member *i* is in the subset. Set unions are computed by the logical *or* of the bit strings, intersections by *and*’s, and so on.

You might want to iterate through all the subsets of a given size. This is easily done if we have a function that maps a given subset to the next higher number (interpreting the subset string as an integer) with the same number of 1-bits.

A concise algorithm for this operation was devised by R. W. Gosper [HAK item 175].[1] Given a word $\boldsymbol{x}$ that represents a subset, the idea is to find the rightmost contiguous group of 1's in $\boldsymbol{x}$ and the following 0's, and "increment" that quantity to the next value that has the same number of 1's. For example, the string xxx0 1111 0000, where xxx represents arbitrary bits, becomes xxx1 0000 0111. The algorithm first identifies the "smallest" 1-bit in $\boldsymbol{x}$, with $\boldsymbol{s} = \boldsymbol{x} \,\&\, -\boldsymbol{x}$, giving 0000 0001 0000. This is added to $\boldsymbol{x}$, giving $\boldsymbol{r}$ = xxx1 0000 0000. The 1-bit here is one bit of the result. For the other bits, we need to produce a right-adjusted string of $n - 1$ 1's, where n is the size of the rightmost group of 1's in $\boldsymbol{x}$. This can be done by first forming the *exclusive or* of $\boldsymbol{r}$ and $\boldsymbol{x}$, which gives 0001 1111 0000 in our example.

This has two too many 1's, and needs to be right-adjusted. This can be accomplished by dividing it by $\boldsymbol{s}$, which right-adjusts it ($\boldsymbol{s}$ is a power of 2), and shifting it right two more positions to discard the two unwanted bits. The final result is the *or* of this and $\boldsymbol{r}$.

In computer algebra notation, the result is $\boldsymbol{y}$ in

$$\begin{aligned} \boldsymbol{s} &\leftarrow \boldsymbol{x} \,\&\, -\boldsymbol{x} \\ \boldsymbol{r} &\leftarrow \boldsymbol{s} + \boldsymbol{x} \\ \boldsymbol{y} &\leftarrow \boldsymbol{r} \mid (((\boldsymbol{x} \oplus \boldsymbol{r}) \overset{u}{\gg} 2) \overset{u}{\div} \boldsymbol{s}) \end{aligned} \tag{2}$$

A complete C procedure is given in Figure 2–1. It executes in seven basic RISC instructions, one of which is division. (Do not use this procedure with $\boldsymbol{x} = 0$; that causes division by 0.)

```
unsigned snoob(unsigned x) {
   unsigned smallest, ripple, ones;
                                // x = xxx0 1111 0000
   smallest = x & -x;           //     0000 0001 0000
   ripple = x + smallest;       //     xxx1 0000 0000
   ones = x ^ ripple;           //     0001 1111 0000
   ones = (ones >> 2)/smallest; //     0000 0000 0111
   return ripple | ones;        //     xxx1 0000 0111
}
```

FIGURE 2–1. Next higher number with same number of 1-bits.

[1] A variation of this algorithm appears in [H&S] section 7.6.7.

If division is slow but you have a fast way to compute the *number of trailing zeros* function ntz(x), the *number of leading zeros* function nlz(x), or *population count* (pop(x) is the number of 1-bits in x), then the last line of Equation (2) can be replaced with one of the following:

$$\begin{aligned}
y &\leftarrow r \mid ((x \oplus r) \overset{u}{\gg} (2 + \mathrm{ntz}(x)))\\
y &\leftarrow r \mid ((x \oplus r) \overset{u}{\gg} (33 - \mathrm{nlz}(s)))\\
y &\leftarrow r \mid ((\mathbf{1} \ll (\mathrm{pop}(x \oplus r) - 2)) - 1)
\end{aligned}$$

2–2 Addition Combined with Logical Operations

We assume the reader is familiar with the elementary identities of ordinary algebra and Boolean algebra. Below is a selection of similar identities involving addition and subtraction combined with logical operations:

$$\begin{aligned}
&\text{a.} & -x &= \neg x + \mathbf{1}\\
&\text{b.} & &= \neg(x - \mathbf{1})\\
&\text{c.} & \neg x &= -x - \mathbf{1}\\
&\text{d.} & -\neg x &= x + \mathbf{1}\\
&\text{e.} & \neg -x &= x - \mathbf{1}\\
&\text{f.} & x + y &= x - \neg y - \mathbf{1}\\
&\text{g.} & &= (x \oplus y) + 2(x \,\&\, y)\\
&\text{h.} & &= (x \mid y) + (x \,\&\, y)\\
&\text{i.} & &= 2(x \mid y) - (x \oplus y)\\
&\text{j.} & x - y &= x + \neg y + \mathbf{1}\\
&\text{k.} & &= (x \oplus y) - 2(\neg x \,\&\, y)\\
&\text{l.} & &= (x \,\&\, \neg y) - (\neg x \,\&\, y)\\
&\text{m.} & &= 2(x \,\&\, \neg y) - (x \oplus y)\\
&\text{n.} & x \oplus y &= (x \mid y) - (x \,\&\, y)\\
&\text{o.} & x \,\&\, \neg y &= (x \mid y) - y\\
&\text{p.} & &= x - (x \,\&\, y)\\
&\text{q.} & \neg(x - y) &= y - x - \mathbf{1}\\
&\text{r.} & &= \neg x + y\\
&\text{s.} & x \equiv y &= (x \,\&\, y) - (x \mid y) - \mathbf{1}\\
&\text{t.} & &= (x \,\&\, y) + \neg(x \mid y)\\
&\text{u.} & x \mid y &= (x \,\&\, \neg y) + y\\
&\text{v.} & x \,\&\, y &= (\neg x \mid y) - \neg x
\end{aligned}$$

Equation (d) may be applied to itself repeatedly, giving $-\neg-\neg\mathbf{x} = \mathbf{x} + \mathbf{2}$, and so on. Similarly, from (e) we have $\neg-\neg-\mathbf{x} = \mathbf{x} - \mathbf{2}$. So we can add or subtract any constant, using only the two forms of complementation.

Equation (f) is the dual of (j), where (j) is the well known relation that shows how to build a subtracter from an adder.

Equations (g) and (h) are from HAKMEM memo [HAK item 23]. Equation (g) forms a sum by first computing the sum with carries ignored $(\mathbf{x} \oplus \mathbf{y})$, and then adding in the carries. Equation (h) is simply modifying the addition operands so that the combination $0 + 1$ never occurs at any bit position; it is replaced with $1 + 0$.

It can be shown that in the ordinary addition of binary numbers with each bit independently equally likely to be 0 or 1, a carry occurs at each position with probability about 0.5. However, for an adder built by preconditioning the inputs using (g), the probability is about 0.25. This observation is probably not of value in building an adder, because for that purpose the important characteristic is the maximum number of logic circuits the carry must pass through, and using (g) reduces the number of stages the carry propagates through by only one.

Equations (k) and (l) are duals of (g) and (h), for subtraction. That is, (k) has the interpretation of first forming the difference ignoring the borrows $(\mathbf{x} \oplus \mathbf{y})$, and then subtracting the borrows. Similarly, Equation (l) is simply modifying the subtraction operands so that the combination $1 - 1$ never occurs at any bit position; it is replaced with $0 - 0$.

Equation (n) shows how to implement *exclusive or* in only three instructions on a basic RISC. Using only *and-or-not* logic requires four instructions $((\mathbf{x} \mid \mathbf{y}) \mathbin{\&} \neg(\mathbf{x} \mathbin{\&} \mathbf{y}))$. Similarly, (u) and (v) show how to implement *and* and *or* in three other elementary instructions, whereas using DeMorgan's laws requires four.

2–3 Inequalities among Logical and Arithmetic Expressions

Inequalities among binary logical expressions whose values are interpreted as unsigned integers are nearly trivial to derive. Here are two examples:

$$(\mathbf{x} \oplus \mathbf{y}) \overset{u}{\le} (\mathbf{x} \mid \mathbf{y}), \quad \text{and}$$

$$(\mathbf{x} \mathbin{\&} \mathbf{y}) \overset{u}{\le} (\mathbf{x} \equiv \mathbf{y}).$$

These can be derived from a list of all binary logical operations, shown in Table 2–1.

Let $f(x, y)$ and $g(x, y)$ represent two columns in Table 2–1. If for each row in which $f(x, y)$ is 1, $g(x, y)$ also is 1, then for all (x, y), $f(x, y) \overset{u}{\le} g(x, y)$. Clearly, this extends to word-parallel logical operations. One can easily read off such relations (most of which are trivial) as $(x \mathbin{\&} y) \overset{u}{\le} x \overset{u}{\le} (x \mid \neg y)$, and so on. Furthermore, if two columns have a row in which one entry is 0 and the other 1, and another row in which the entries are 1 and 0 respectively, then no inequality relation exists between the corresponding logical expressions. So the question of

TABLE 2–1. THE 16 BINARY LOGICAL OPERATIONS

x	y	0	$x \,\&\, y$	$x \,\&\, \neg y$	x	$\neg x \,\&\, y$	y	$x \oplus y$	$x \mid y$	$\neg(x \mid y)$	$x \equiv y$	$\neg y$	$x \mid \neg y$	$\neg x$	$\neg x \mid y$	$\neg(x \,\&\, y)$	1
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

whether or not $f(x, y) \overset{u}{\leq} g(x, y)$ is completely and easily solved for all binary logical functions f and g.

Use caution when manipulating these relations. For example, for ordinary arithmetic, if $x + y \leq a$ and $z \leq x$, then $z + y \leq a$. But this inference is not valid if "+" is replaced with *or*.

Inequalities involving mixed logical and arithmetic expressions are more interesting. Below is a small selection.

a. $(\boldsymbol{x} \mid \boldsymbol{y}) \overset{u}{\geq} \max(\boldsymbol{x}, \boldsymbol{y})$

b. $(\boldsymbol{x} \,\&\, \boldsymbol{y}) \overset{u}{\leq} \min(\boldsymbol{x}, \boldsymbol{y})$

c. $(\boldsymbol{x} \mid \boldsymbol{y}) \overset{u}{\leq} \boldsymbol{x} + \boldsymbol{y}$ if the addition does not overflow

d. $(\boldsymbol{x} \mid \boldsymbol{y}) \overset{u}{>} \boldsymbol{x} + \boldsymbol{y}$ if the addition overflows

e. $|\boldsymbol{x} - \boldsymbol{y}| \overset{u}{\leq} (\boldsymbol{x} \oplus \boldsymbol{y})$

The proofs of these are quite simple, except possibly for the relation $|\boldsymbol{x} - \boldsymbol{y}| \overset{u}{\leq} (\boldsymbol{x} \oplus \boldsymbol{y})$. By $|\boldsymbol{x} - \boldsymbol{y}|$ we mean the absolute value of $\boldsymbol{x} - \boldsymbol{y}$, which may be computed within the domain of unsigned numbers as $\max(\boldsymbol{x}, \boldsymbol{y}) - \min(\boldsymbol{x}, \boldsymbol{y})$. This relation may be proven by induction on the length of $\boldsymbol{x}$ and $\boldsymbol{y}$ (the proof is a little easier if you extend them on the left rather than on the right).

2–4 Absolute Value Function

If your machine does not have an instruction for computing the absolute value, this computation can usually be done in three or four branch-free instructions. First, compute $\boldsymbol{y} \leftarrow \boldsymbol{x} \overset{s}{\gg} 31$, and then one of the following:

abs	nabs
$(\boldsymbol{x} \oplus \boldsymbol{y}) - \boldsymbol{y}$	$\boldsymbol{y} - (\boldsymbol{x} \oplus \boldsymbol{y})$
$(\boldsymbol{x} + \boldsymbol{y}) \oplus \boldsymbol{y}$	$(\boldsymbol{y} - \boldsymbol{x}) \oplus \boldsymbol{y}$
$\boldsymbol{x} - (2\boldsymbol{x} \,\&\, \boldsymbol{y})$	$(2\boldsymbol{x} \,\&\, \boldsymbol{y}) - \boldsymbol{x}$

By "$2\boldsymbol{x}$" we mean, of course, $\boldsymbol{x} + \boldsymbol{x}$ or $\boldsymbol{x} \ll 1$.

If you have a fast multiply by a variable whose value is ±1, the following will do:

$$((\boldsymbol{x} \overset{s}{\gg} 30) \mid \mathbf{1}) * \boldsymbol{x}$$

2–5 Sign Extension

By “sign extension,” we mean to consider a certain bit position in a word to be the sign bit, and we wish to propagate that to the left, ignoring any other bits present. The standard way to do this is with *shift left logical* followed by *shift right signed*. However, if these instructions are slow or nonexistent on your machine, it may be done with one of the following, where we illustrate by propagating bit position 7 to the left:

$$((\boldsymbol{x} + \mathbf{0x00000080}) \,\&\, \mathbf{0x000000FF}) - \mathbf{0x00000080}$$
$$((\boldsymbol{x} \,\&\, \mathbf{0x000000FF}) \oplus \mathbf{0x00000080}) - \mathbf{0x00000080}$$

The “+” above can also be “–” or “⊕.” The second formula is particularly useful if you know that the unwanted high-order bits are all 0’s, because then the *and* can be omitted.

2–6 Shift Right Signed from Unsigned

If your machine does not have the *shift right signed* instruction, it may be computed using the formulas below. The first formula is from [GM], and the second is based on the same idea. Assuming the machine has mod 64 shifts, the first four formulas hold for $0 \le n \le 31$, and the last holds for $0 \le n \le 63$. The last formula holds for any *n* if by “holds” we mean “treats the shift amount to the same modulus as does the logical shift.”

When *n* is a variable, each formula requires five or six instructions on a basic RISC.

$$((\boldsymbol{x} + \mathbf{0x80000000}) \overset{u}{\gg} \boldsymbol{n}) - (\mathbf{0x80000000} \overset{u}{\gg} \boldsymbol{n})$$
$$\boldsymbol{t} \leftarrow \mathbf{0x80000000} \overset{u}{\gg} \boldsymbol{n}; \qquad ((\boldsymbol{x} \overset{u}{\gg} \boldsymbol{n}) \oplus \boldsymbol{t}) - \boldsymbol{t}$$
$$\boldsymbol{t} \leftarrow (\boldsymbol{x} \,\&\, \mathbf{0x80000000}) \overset{u}{\gg} \boldsymbol{n};\; (\boldsymbol{x} \overset{u}{\gg} \boldsymbol{n}) - (\boldsymbol{t} + \boldsymbol{t})$$
$$(\boldsymbol{x} \overset{u}{\gg} \boldsymbol{n}) \mid (-(\boldsymbol{x} \overset{u}{\gg} 31) \ll \mathbf{31} - \boldsymbol{n})$$
$$\boldsymbol{t} \leftarrow -(\boldsymbol{x} \overset{u}{\gg} 31); \qquad ((\boldsymbol{x} \oplus \boldsymbol{t}) \overset{u}{\gg} \boldsymbol{n}) \oplus \boldsymbol{t}$$

In the first two formulas, an alternative for the expression $\mathbf{0x80000000} \overset{u}{\gg} \boldsymbol{n}$ is $\mathbf{1} \ll \mathbf{31} - \boldsymbol{n}$.

If n is a constant, the first two formulas require only three instructions on many machines. If $n = 31$, the function can be done in two instructions with $-(\boldsymbol{x} \overset{u}{\gg} 31)$.

2–7 *Sign* Function

The *sign*, or *signum*, function is defined by

$$\operatorname{sign}(x) = \begin{cases} -1, & x < 0, \\ 0, & x = 0, \\ 1, & x > 0. \end{cases}$$

It may be calculated with four instructions on most machines [Hop]:

$$(\boldsymbol{x} \overset{s}{\gg} 31) \mid (-\boldsymbol{x} \overset{u}{\gg} 31)$$

If you don't have *shift right signed*, then use the substitute noted at the end of Section 2–6, giving the following nicely symmetric formula (five instructions):

$$-(\boldsymbol{x} \overset{u}{\gg} 31) \mid (-\boldsymbol{x} \overset{u}{\gg} 31)$$

Comparison predicate instructions permit a three-instruction solution, with either

$$\begin{aligned} &(\boldsymbol{x} > 0) - (\boldsymbol{x} < 0), \text{ or} \\ &(\boldsymbol{x} \geq 0) - (\boldsymbol{x} \leq 0). \end{aligned} \tag{3}$$

Finally, we note that the formula $(-\boldsymbol{x} \overset{u}{\gg} 31) - (\boldsymbol{x} \overset{u}{\gg} 31)$ almost works; it fails only for $\boldsymbol{x} = -\mathbf{2^{31}}$.

2–8 *Three-Valued Compare* Function

The *three-valued compare* function, a slight generalization of the *sign* function, is defined by

$$\operatorname{cmp}(x, y) = \begin{cases} -1, & x < y, \\ 0, & x = y, \\ 1, & x > y. \end{cases}$$

There are both signed and unsigned versions, and unless otherwise specified, this section applies to both.

Comparison predicate instructions permit a three-instruction solution, an obvious generalization of Equations (3):

$$\begin{aligned}&(\boldsymbol{x} > \boldsymbol{y}) - (\boldsymbol{x} < \boldsymbol{y}), \quad \text{or}\\ &(\boldsymbol{x} \ge \boldsymbol{y}) - (\boldsymbol{x} \le \boldsymbol{y}).\end{aligned}$$

A solution for unsigned integers on PowerPC is shown below [CWG]. On this machine, “carry” is “not borrow.”

```
subf  R5,Ry,Rx    # R5 <-- Rx - Ry.
subfc R6,Rx,Ry    # R6 <-- Ry - Rx, set carry.
subfe R7,Ry,Rx    # R7 <-- Rx - Ry + carry, set carry.
subfe R8,R7,R5    # R8 <-- R5 - R7 + carry, (set carry).
```

If limited to the instructions of the basic RISC, there does not seem to be any particularly good way to compute this function. The comparison predicates $\boldsymbol{x} < \boldsymbol{y}$, $\boldsymbol{x} \le \boldsymbol{y}$, and so on., require about five instructions (see Section 2–11), leading to a solution in about 12 instructions (using a small amount of commonality in computing $\boldsymbol{x} < \boldsymbol{y}$ and $\boldsymbol{x} > \boldsymbol{y}$). On the basic RISC it’s probably preferable to use compares and branches (six instructions executed worst case if compares can be commoned).

2–9 Transfer of Sign

The *transfer of sign* function, called ISIGN in Fortran, is defined by

$$\text{ISIGN}(x, y) = \begin{cases} \text{abs}(x), & y \ge 0, \\ -\text{abs}(x), & y < 0. \end{cases}$$

This function may be calculated (modulo 2^{32}) with four instructions on most machines:

$$\begin{aligned}&\boldsymbol{t} \leftarrow \boldsymbol{y} \overset{s}{\gg} 31;\\ &\text{ISIGN}(\boldsymbol{x}, \boldsymbol{y}) = (\text{abs}(\boldsymbol{x}) \oplus \boldsymbol{t}) - \boldsymbol{t}\\ &\qquad\qquad\qquad\; = (\text{abs}(\boldsymbol{x}) + \boldsymbol{t}) \oplus \boldsymbol{t}\end{aligned}$$

$$\begin{aligned}&\boldsymbol{t} \leftarrow (\boldsymbol{x} \oplus \boldsymbol{y}) \overset{s}{\gg} 31;\\ &\text{ISIGN}(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x} \oplus \boldsymbol{t}) - \boldsymbol{t}\\ &\qquad\qquad\qquad\; = (\boldsymbol{x} + \boldsymbol{t}) \oplus \boldsymbol{t}\end{aligned}$$

2–10 Decoding a “Zero Means 2***n*” Field

Sometimes a zero or negative value does not make much sense for a quantity, so it is encoded in an *n*-bit field with a zero value being understood to mean 2^n, and a nonzero value having its normal binary interpretation. An example is the length field of PowerPC’s *load string word immediate* (`lswi`) instruction, which occupies five bits. It is not useful to have an instruction that loads zero bytes, when the length is an immediate quantity, but it is definitely useful to be able to load 32 bytes. The length field could be encoded with values from 0 to 31 denoting lengths from 1 to 32, but the "zero means 32" convention results in simpler logic when the processor must also support a corresponding instruction with a variable (in-register) length that employs straight binary encoding (e.g., PowerPC's `lswx` instruction).

It is trivial to encode an integer in the range 1 to 2^n into the "zero means 2^n" encoding — simply mask the integer with $2^n - 1$. To do the decoding without a test-and-branch is not quite as simple, but below are some possibilities (no doubt overdone), illustrated for a 3-bit field. They all require three instructions, not counting possible loads of constants.

$$\begin{array}{lll}
((x - 1)\ \&\ 7) + 1 & ((x + 7)\ |\ -8) + 9 & 8 - (-x\ \&\ 7) \\
((x + 7)\ \&\ 7) + 1 & ((x + 7)\ |\ 8) - 7 & -(-x\ |\ -8) \\
((x - 1)\ |\ -8) + 9 & ((x - 1)\ \&\ 8) + x &
\end{array}$$

2–11 Comparison Predicates

A "comparison predicate" is a function that compares two quantities, producing a single bit result of 1 if the comparison is **true**, and 0 if the comparison is **false**. Below we show branch-free expressions to evaluate the result into the sign position. To produce the 1/0 value used by some languages (e.g., C), follow the code with a *shift right* of 31. To produce the $-1/0$ result used by some other languages (e.g., Basic), follow the code with a *shift right signed* of 31.

These formulas are, of course, not of interest on machines such as MIPS, the Compaq Alpha, and our model RISC, which have comparison instructions that compute many of these predicates directly, placing a 0/1-valued result in a general purpose register.

A machine instruction that computes the negative of the absolute value is handy here. We show this function as "nabs." Unlike absolute value, it is well defined in that it never overflows. Machines that do not have "nabs" but have the more usual "abs" may use $-\text{abs}(x)$ for $\text{nabs}(x)$. If x is the maximum negative number, this overflows twice, but the result is correct. (We assume that the absolute value and the negation of the maximum negative number is itself.) Because some machines have neither "abs" nor "nabs," we give an alternative that does not use them.

The "nlz" function is the number of leading 0's in its argument. The "doz" function (*difference or zero*) is described on page 37.

$$\begin{array}{lll}
x = y: & \mathrm{abs}(x - y) - \mathbf{1} & \\
 & \mathrm{abs}(x - y + \mathbf{0x8000\,0000}) & \\
 & \mathrm{nlz}(x - y) \ll 26 & \\
 & -(\mathrm{nlz}(x - y) \overset{u}{\gg} 5) & \\
 & \neg(x - y \mid y - x) & \\
x \ne y: & \mathrm{nabs}(x - y) & \\
 & \mathrm{nlz}(x - y) - \mathbf{32} & \\
 & x - y \mid y - x & \\
x < y: & (x - y) \oplus [(x \oplus y) \,\&\, ((x - y) \oplus x)] & \\
 & (x \,\&\, \neg y) \mid ((x \equiv y) \,\&\, (x - y)) & \\
 & \mathrm{nabs}(\mathrm{doz}(y, x)) & \text{[GSO]} \\
x \le y: & (x \mid \neg y) \,\&\, ((x \oplus y) \mid \neg(y - x)) & \\
 & ((x \equiv y) \overset{s}{\gg} 1) + (x \,\&\, \neg y) & \text{[GSO]} \\
x \overset{u}{<} y: & (\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, (x - y)) & \\
 & (\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, (x - y)) & \\
x \overset{u}{\le} y: & (\neg x \mid y) \,\&\, ((x \oplus y) \mid \neg(y - x)) &
\end{array}$$

For $x > y$, $x \ge y$, and so on, interchange x and y in the formulas for $x < y$, $x \le y$, and so on. The *add* of $\mathbf{0x8000\,0000}$ may be replaced with any instruction that inverts the high order bit (in x, y, or $x - y$).

Another class of formulas can be derived from the observation that the predicate $x < y$ is given by the sign of $x/2 - y/2$, and the subtraction in that expression cannot overflow. The result can be fixed up by subtracting 1 in the cases in which the shifts discard essential information, as follows:

$$\begin{array}{ll}
x < y: & (x \overset{s}{\gg} 1) - (y \overset{s}{\gg} 1) - (\neg x \,\&\, y \,\&\, \mathbf{1}) \\
x \overset{u}{<} y: & (x \overset{u}{\gg} 1) - (y \overset{u}{\gg} 1) - (\neg x \,\&\, y \,\&\, \mathbf{1})
\end{array}$$

These execute in seven instructions on most machines (six if it has *and not*), which is no better than what we have above (five to seven instructions depending upon the fullness of the set of logic instructions).

The formulas above involving “nlz” are due to [Shep], and his formula for the $x = y$ predicate is particularly useful because a minor variation of it gets the predicate evaluated to a 1/0-valued result with only three instructions:

$$\mathrm{nlz}(x - y) \overset{u}{\gg} 5.$$

Signed comparisons to 0 are frequent enough to deserve special mention. Below are some formulas for these, mostly derived directly from the above. Again, the result is in the sign position.

$x = \mathbf{0}$:	$\text{abs}(x) - \mathbf{1}$	
	$\text{abs}(x + \mathbf{0x80000000})$	
	$\text{nlz}(x) \ll 26$	
	$-(\text{nlz}(x) \overset{u}{\gg} 5)$	
	$\neg(x \mid -x)$	
	$\neg x \,\&\, (x - \mathbf{1})$	
$x \neq \mathbf{0}$:	$\text{nabs}(x)$	
	$\text{nlz}(x) - \mathbf{32}$	
	$x \mid -x$	
	$(x \overset{u}{\gg} 1) - x$	[CWG]
$x < \mathbf{0}$:	x	
$x \leq \mathbf{0}$:	$x \mid (x - \mathbf{1})$	
	$x \mid \neg -x$	
$x > \mathbf{0}$:	$x \oplus \text{nabs}(x)$	
	$-x \,\&\, \neg x$	
$x \geq \mathbf{0}$:	$\neg x$	

Signed comparisons can be obtained from their unsigned counterparts by biasing the signed operands upwards by 2^{31} and interpreting the results as unsigned integers. The reverse transformation also works. Thus we have

$$x < y \;=\; x + \mathbf{2^{31}} \overset{u}{<} y + \mathbf{2^{31}},$$

$$x \overset{u}{<} y \;=\; x - \mathbf{2^{31}} < y - \mathbf{2^{31}}.$$

Similar relations hold for $\leq$, $\overset{u}{\leq}$, and so on. Addition and subtraction of 2^{31} are equivalent, as they amount to inverting the sign bit.

Another way to get signed comparisons from unsigned is based on the fact that if x and y have the same sign, then $x < y \;=\; x \overset{u}{<} y$, whereas if they have opposite signs, then $x < y \;=\; x \overset{u}{>} y$ [Lamp]. Again, the reverse transformation also works, so we have

$$x < y \;=\; (x \overset{u}{<} y) \oplus x_{31} \oplus y_{31} \quad \text{and}$$

$$x \overset{u}{<} y \;=\; (x < y) \oplus x_{31} \oplus y_{31},$$

where x_{31} and y_{31} are the sign bits of x and y, respectively. Similar relations hold for $\le$, $\overset{u}{\le}$, and so on.

Using either of these devices enables computing all the usual comparison predicates other than = and ≠ in terms of any one of them, with at most three additional instructions on most machines. For example, let us take $x \overset{u}{\le} y$ as primitive, because it is one of the simplest to implement (it is the carry bit from $y - x$). Then the other predicates can be obtained as follows:

$$\begin{aligned}
x < y &= \neg(y + \mathbf{2^{31}} \overset{u}{\le} x + \mathbf{2^{31}}) \\
x \le y &= x + \mathbf{2^{31}} \overset{u}{\le} y + \mathbf{2^{31}} \\
x > y &= \neg(x + \mathbf{2^{31}} \overset{u}{\le} y + \mathbf{2^{31}}) \\
x \ge y &= y + \mathbf{2^{31}} \overset{u}{\le} x + \mathbf{2^{31}} \\
x \overset{u}{<} y &= \neg(y \overset{u}{\le} x) \\
x \overset{u}{>} y &= \neg(x \overset{u}{\le} y) \\
x \overset{u}{\ge} y &= y \overset{u}{\le} x
\end{aligned}$$

Comparison Predicates from the Carry Bit

If the machine can easily deliver the carry bit into a general purpose register, this may permit concise code for some of the comparison predicates. Below are listed several of these relations. The notation carry(*expression*) means the carry bit generated by the outermost operation in *expression*. We assume the carry bit for the subtraction $x - y$ is what comes out of the adder for $x + \bar{y} + 1$, which is the complement of “borrow.”

$x = y$:	carry($\mathbf{0} - (x - y)$), or carry($(x + \bar{y}) + \mathbf{1}$), or carry($(x - y - \mathbf{1}) + \mathbf{1}$)
$x \ne y$:	carry($(x - y) - \mathbf{1}$), i.e., carry($(x - y) + (-\mathbf{1})$)
$x < y$:	$\neg$carry($(x + \mathbf{2^{31}}) - (y + \mathbf{2^{31}})$)
$x \le y$:	carry($(y + \mathbf{2^{31}}) - (x + \mathbf{2^{31}})$)
$x \overset{u}{<} y$:	$\neg$carry($x - y$)
$x \overset{u}{\le} y$:	carry($y - x$)
$x = \mathbf{0}$:	carry($\mathbf{0} - x$), or carry($\bar{x} + \mathbf{1}$)
$x \ne \mathbf{0}$:	carry($x - \mathbf{1}$), i.e., carry($x + (-\mathbf{1})$)
$x < \mathbf{0}$:	carry($x + x$)
$x \le \mathbf{0}$:	carry($\mathbf{2^{31}} - (x + \mathbf{2^{31}})$)

For $x > y$, use the complement of the expression for $x \le y$, and similarly for other relations involving “greater than.”

The GNU Superoptimizer has been applied to the problem of computing predicate expressions on the IBM RS/6000 computer and its close relative PowerPC [GK]. The RS/6000 has instructions for abs($\boldsymbol{x}$), nabs($\boldsymbol{x}$), doz($\boldsymbol{x}, \boldsymbol{y}$), and a number of forms of *add* and *subtract* that use the carry bit. It was found that the RS/6000 can compute all the integer predicate expressions with three or fewer elementary (one-cycle) instructions, a result that surprised even the architects of the machine. “All” includes the six two-operand signed comparisons, the four two-operand unsigned comparisons, all these with the second operand being zero, and all in forms that produce a 1/0 result or a –1/0 result. PowerPC, which lacks abs($\boldsymbol{x}$), nabs($\boldsymbol{x}$), and doz($\boldsymbol{x}, \boldsymbol{y}$), can compute all the predicate expressions in four or fewer elementary instructions.

How the Computer Sets the Comparison Predicates

Most computers have a way of evaluating the integer comparison predicates to a 1-bit result. The result bit may be placed in a “condition register” or, for some machines (such as our RISC model), in a general purpose register. In either case, the facility is often implemented by subtracting the comparison operands and then performing a small amount of logic on the result bits to determine the 1-bit comparison result.

Below is the logic for these operations. It is assumed that the machine computes $\boldsymbol{x} - \boldsymbol{y}$ as $\boldsymbol{x} - \bar{\boldsymbol{y}} + 1$, and the following quantities are available in the result:

C_o, the carry out of the high-order position
C_i, the carry into the high-order position
N, the sign bit of the result
Z, which equals 1 if the result, exclusive of C_o, is all-zero, and is otherwise 0

Then we have the following in Boolean algebra notation (juxtaposition denotes *and*, + denotes *or*):

V:	$C_i \oplus C_o$	(signed overflow)
$\boldsymbol{x} = \boldsymbol{y}$:	Z	
$\boldsymbol{x} \neq \boldsymbol{y}$:	$\bar{Z}$	
$\boldsymbol{x} < \boldsymbol{y}$:	$N \oplus V$	
$\boldsymbol{x} \leq \boldsymbol{y}$:	$(N \oplus V) + Z$	
$\boldsymbol{x} > \boldsymbol{y}$:	$(N \equiv V)\bar{Z}$	
$\boldsymbol{x} \geq \boldsymbol{y}$:	$N \equiv V$	
$\boldsymbol{x} \overset{u}{<} \boldsymbol{y}$:	$\overline{C_o}$	
$\boldsymbol{x} \overset{u}{\leq} \boldsymbol{y}$:	$\overline{C_o} + Z$	
$\boldsymbol{x} \overset{u}{>} \boldsymbol{y}$:	$C_o\bar{Z}$	
$\boldsymbol{x} \overset{u}{\geq} \boldsymbol{y}$:	C_o	

2–12 Overflow Detection

“Overflow” means that the result of an arithmetic operation is too large or too small to be correctly represented in the target register. This section discusses methods that a programmer might use to detect when overflow has occurred, without using the machine’s “status bits” that are often expressly supplied for this purpose. This is important because some machines do not have such status bits (e.g., MIPS), and because even if the machine is so equipped, it is often difficult or impossible to access the bits from a high-level language.

Signed Add/Subtract

When overflow occurs on integer addition and subtraction, contemporary machines invariably discard the high-order bit of the result and store the low-order bits that the adder naturally produces. Signed integer overflow of addition occurs if and only if the operands have the same sign and the sum has sign opposite to that of the operands. Surprisingly, this same rule applies even if there is a carry into the adder—that is, if the calculation is $\boldsymbol{x} + \boldsymbol{y} + 1.$ This is important for the application of adding multiword signed integers, in which the last addition is a signed addition of two fullwords and a carry-in that may be 0 or +1.

To prove the rule for addition, let x and y denote the values of the one-word signed integers being added, let c (carry-in) be 0 or 1, and assume for simplicity a 4-bit machine. Then if the signs of x and y are different,

$$\begin{gathered} -8 \le x \le -1, \text{ and} \\ 0 \le y \le 7, \end{gathered}$$

or similar bounds apply if x is nonnegative and y is negative. In either case, by adding these inequalities and optionally adding in 1 for c,

$$-8 \le x + y + c \le 7.$$

This is representable as a 4-bit signed integer, and thus overflow does not occur when the operands have opposite signs.

Now suppose x and y have the same sign. There are two cases:

$$\begin{array}{cc} (a) & (b) \\ -8 \le x \le -1 & 0 \le x \le 7 \\ -8 \le y \le -1 & 0 \le y \le 7 \end{array}$$

Thus,

$$-16 \le x + y + c \le -1 \qquad 0 \le x + y + c \le 15.$$

Overflow occurs if the sum is not representable as a 4-bit signed integer—that is, if

$$-16 \le x + y + c \le -9 \qquad 8 \le x + y + c \le 15.$$

In case (a), this is equivalent to the high-order bit of the 4-bit sum being 0, which is opposite to the sign of x and y. In case (b), this is equivalent to the high-order bit of the 4-bit sum being 1, which again is opposite to the sign of x and y.

For subtraction of multiword integers, the computation of interest is $x - y - c$, where again c is $\mathbf{0}$ or $\mathbf{1}$, with a value of $\mathbf{1}$ representing a borrow-in. From an analysis similar to the above, it can be seen that overflow in the final value of $x - y - c$ occurs if and only if x and y have opposite signs and the sign of $x - y - c$ is opposite to that of x (or, equivalently, the same as that of y).

This leads to the following expressions for the overflow predicate, with the result being in the sign position. Following these with a *shift right* or *shift right signed* of 31 produces a 1/0- or a –1/0-valued result.

$$\begin{array}{ll}
\qquad\qquad x + y + c & \qquad\qquad x - y - c \\
(x \equiv y) \,\&\, ((x + y + c) \oplus x) & (x \oplus y) \,\&\, ((x - y - c) \oplus x) \\
((x + y + c) \oplus x) \,\&\, ((x + y + c) \oplus y) & ((x - y - c) \oplus x) \,\&\, ((x - y - c) \equiv y)
\end{array}$$

By choosing the second alternative in the first column, and the first alternative in the second column (avoiding the *equivalence* operation), our basic RISC can evaluate these tests with three instructions in addition to those required to compute $x + y + c$ or $x - y - c$. A fourth instruction (branch if negative) may be added to branch to code where the overflow condition is handled.

If executing with overflow interrupts enabled, the programmer may wish to test to see if a certain addition or subtraction will cause overflow, in a way that does not cause it. One branch-free way to do this is as follows:

$$\begin{array}{ll}
\qquad\qquad x + y + c & \qquad\qquad x - y - c \\
z \leftarrow (x \equiv y) \,\&\, \mathbf{0x80000000} & z \leftarrow (x \oplus y) \,\&\, \mathbf{0x80000000} \\
(x \equiv y) \,\&\, ((x \oplus z) + y + c) \equiv y & (x \oplus y) \,\&\, ((x \oplus z) - y - c) \oplus y
\end{array}$$

The assignment to z in the left column sets $z = \mathbf{0x80000000}$ if x and y have the same sign, and sets $z = \mathbf{0}$ if they differ. Then, the addition in the second expression is done with x and y having different signs, so it can't overflow. If x and y are nonnegative, the sign bit in the second expression will be 1 if and only if $(x - \mathbf{2^{31}}) + y + c \ge \mathbf{0}$—that is, iff $x + y + c \ge 2^{31}$, which is the condition for overflow in evaluating $x + y + c$. If x and y are negative, the sign bit in the second expression will be 1 iff $(x + \mathbf{2^{31}}) + y + c < \mathbf{0}$—that is, iff $x + y + c < -2^{31}$, which again is the condition for overflow. The term $x \equiv y$ ensures the correct result (0 in the sign position) if x and y have opposite signs. Similar remarks apply to the case of subtraction (right column). The code executes in nine instructions on the basic RISC.

It might seem that if the carry from addition is readily available, this might help in computing the signed overflow predicate. This does not seem to be the case. However, one method along these lines is as follows.

If $\boldsymbol{x}$ is a signed integer, then $\boldsymbol{x} + \mathbf{2^{31}}$ is correctly represented as an unsigned number, and is obtained by inverting the high-order bit of $\boldsymbol{x}$. Signed overflow in the positive direction occurs if $x + y \geq 2^{31}$—that is, if $(x + 2^{31}) + (y + 2^{31}) \geq 3 \cdot 2^{31}$. This latter condition is characterized by carry occurring in the unsigned add (which means that the sum is greater than or equal to 2^{32}) and the high-order bit of the sum being 1. Similarly, overflow in the negative direction occurs if the carry is 0 and the high-order bit of the sum is also 0.

This gives the following algorithm for detecting overflow for signed addition:

Compute $(\boldsymbol{x} \oplus \mathbf{2^{31}}) + (\boldsymbol{y} \oplus \mathbf{2^{31}})$, giving sum $\boldsymbol{s}$ and carry c.
Overflow occurred iff c equals the high-order bit of $\boldsymbol{s}$.

The sum is the correct sum for the signed addition, because inverting the high-order bits of both operands does not change their sum.

For subtraction, the algorithm is the same except in the first step a subtraction replaces the addition. We assume that the carry is that generated by computing $\boldsymbol{x} - \boldsymbol{y}$ as $\boldsymbol{x} + \bar{\boldsymbol{y}} + 1$. The subtraction is the correct difference for the signed subtraction.

These formulas are perhaps interesting, but on most machines they would not be quite as efficient as the formulas that do not even use the carry bit (e.g., overflow = $(\boldsymbol{x} \equiv \boldsymbol{y})\ \&\ (\boldsymbol{s} \oplus \boldsymbol{x})$ for addition, and $(\boldsymbol{x} \oplus \boldsymbol{y})\ \&\ (\boldsymbol{d} \oplus \boldsymbol{x})$ for subtraction, where $\boldsymbol{s}$ and $\boldsymbol{d}$ are the sum and difference, respectively, of $\boldsymbol{x}$ and $\boldsymbol{y}$).

How the Computer Sets Overflow for Signed Add/Subtract

Machines often set “overflow” for signed addition by means of the logic “the carry into the sign position is not equal to the carry out of the sign position.” Curiously, this logic gives the correct overflow indication for both addition and subtraction, assuming the subtraction $\boldsymbol{x} - \boldsymbol{y}$ is done by $\boldsymbol{x} + \bar{\boldsymbol{y}} + 1$. Furthermore, it is correct whether or not there is a carry- or borrow-in. This does not seem to lead to any particularly good methods for computing the signed overflow predicate in software, however, even though it is easy to compute the carry into the sign position. For addition and subtraction, the carry/borrow into the sign position is given by the sign bit after evaluating the following expressions (where $\boldsymbol{c}$ is $\mathbf{0}$ or $\mathbf{1}$):

carry	borrow
$(\boldsymbol{x} + \boldsymbol{y} + \boldsymbol{c}) \oplus \boldsymbol{x} \oplus \boldsymbol{y}$	$(\boldsymbol{x} - \boldsymbol{y} - \boldsymbol{c}) \oplus \boldsymbol{x} \oplus \boldsymbol{y}$

In fact, these expressions give, at each position i, the carry/borrow into position i.

Unsigned Add/Subtract

The following branch-free code may be used to compute the overflow predicate for unsigned add/subtract, with the result being in the sign position. The expressions involving a right shift are probably useful only when it is known that $c = \mathbf{0}$. The expressions in brackets compute the carry or borrow generated from the least significant position.

$$x + y + c\text{, unsigned}$$

$$(x \,\&\, y) \mid ((x \mid y) \,\&\, \neg(x + y + c))$$

$$(x \overset{u}{\gg} 1) + (y \overset{u}{\gg} 1) + [((x \,\&\, y) \mid ((x \mid y) \,\&\, c)) \,\&\, \mathbf{1}]$$

$$x - y - c\text{, unsigned}$$

$$(\neg x \,\&\, y) \mid ((x \equiv y) \,\&\, (x - y - c))$$

$$(\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, (x - y - c))$$

$$(x \overset{u}{\gg} 1) - (y \overset{u}{\gg} 1) - [((\neg x \,\&\, y) \mid ((\neg x \mid y) \,\&\, c)) \,\&\, \mathbf{1}]$$

For unsigned *add*’s and *subtract*’s, there are much simpler formulas in terms of comparisons [MIPS]. For unsigned addition, overflow (carry) occurs if the sum is less (by unsigned comparison) than either of the operands. This and similar formulas are given below. Unfortunately, there is no way in these formulas to allow for a variable c that represents the carry- or borrow-in. Instead, the program must test c, and use a different type of comparison depending upon whether c is $\mathbf{0}$ or $\mathbf{1}$.

$x + y$, unsigned	$x + y + \mathbf{1}$, unsigned	$x - y$, unsigned	$x - y - \mathbf{1}$, unsigned
$\neg x \overset{u}{<} y$	$\neg x \overset{u}{\le} y$	$x \overset{u}{<} y$	$x \overset{u}{\le} y$
$x + y \overset{u}{<} x$	$x + y + \mathbf{1} \overset{u}{\le} x$	$x - y \overset{u}{>} x$	$x - y - \mathbf{1} \overset{u}{\ge} x$

The first formula for each case above is evaluated before the add/subtract that may overflow, and it provides a way to do the test without causing overflow. The second formula for each case is evaluated after the add/subtract that may overflow.

There does not seem to be a similar simple device (using comparisons) for computing the signed overflow predicate.

Multiplication

For multiplication, overflow means that the result cannot be expressed in 32 bits (it can always be expressed in 64 bits, whether signed or unsigned). Checking for overflow is simple if you have access to the high-order 32 bits of the product. Let us denote the two halves of the 64-bit product by $\mathrm{hi}(\boldsymbol{x} \times \boldsymbol{y})$ and $\mathrm{lo}(\boldsymbol{x} \times \boldsymbol{y})$. Then the overflow predicates can be computed as follows [MIPS]:

$\boldsymbol{x} \times \boldsymbol{y}$, unsigned	$\boldsymbol{x} \times \boldsymbol{y}$, signed
$\mathrm{hi}(\boldsymbol{x} \times \boldsymbol{y}) \neq \mathbf{0}$	$\mathrm{hi}(\boldsymbol{x} \times \boldsymbol{y}) \neq (\mathrm{lo}(\boldsymbol{x} \times \boldsymbol{y}) \overset{s}{\gg} 31)$

One way to check for overflow of multiplication is to do the multiplication and then check the result by dividing. But care must be taken not to divide by 0, and there is a further complication for signed multiplication. Overflow occurs if the following expressions are **true**:

Unsigned	Signed
$\boldsymbol{z} \leftarrow \boldsymbol{x} * \boldsymbol{y}$	$\boldsymbol{z} \leftarrow \boldsymbol{x} * \boldsymbol{y}$
$\boldsymbol{y} \neq \mathbf{0} \overrightarrow{\&} \boldsymbol{z} \overset{u}{\div} \boldsymbol{y} \neq \boldsymbol{x}$	$(\boldsymbol{y} < \mathbf{0} \;\&\; \boldsymbol{x} = -\mathbf{2^{31}}) \mid (\boldsymbol{y} \neq \mathbf{0} \overrightarrow{\&} \boldsymbol{z} \div \boldsymbol{y} \neq \boldsymbol{x})$

The complication arises when $\boldsymbol{x} = -\mathbf{2^{31}}$ and $\boldsymbol{y} = -\mathbf{1}$. In this case the multiplication overflows, but the machine may very well give a result of $-\mathbf{2^{31}}$. This causes the division to overflow and thus any result is possible (for some machines). Thus this case has to be checked separately, which is done by the term $\boldsymbol{y} < \mathbf{0} \;\&\; \boldsymbol{x} = -\mathbf{2^{31}}$. The above expressions use the "conditional *and*" operator to prevent dividing by zero (in C, use the `&&` operator).

It is also possible to use division to check for overflow of multiplication without doing the multiplication (that is, without causing overflow). For unsigned integers, the product overflows iff $xy > 2^{32} - 1$, or $x > ((2^{32} - 1)/y)$, or, since x is an integer, $x > \lfloor(2^{32} - 1)/y\rfloor$. Expressed in computer arithmetic, this is

$$\boldsymbol{y} \neq \mathbf{0} \;\&\; \boldsymbol{x} \overset{u}{>} (\mathbf{0xFFFFFFFF} \overset{u}{\div} \boldsymbol{y}).$$

For signed integers, the determination of overflow of $\boldsymbol{x} * \boldsymbol{y}$ is not so simple. If $\boldsymbol{x}$ and $\boldsymbol{y}$ have the same sign, then overflow occurs iff $xy > 2^{31} - 1$. If they have opposite signs, then overflow occurs iff $xy < -2^{31}$. These conditions may be tested as indicated in Table 2–2, which employs signed division.

TABLE 2–2. OVERFLOW TEST FOR SIGNED MULTIPLICATION

	$\boldsymbol{y} > \mathbf{0}$	$\boldsymbol{y} \leq \mathbf{0}$
$\boldsymbol{x} > \mathbf{0}$	$\boldsymbol{x} > \mathbf{0x7FFFFFFF} \div \boldsymbol{y}$	$\boldsymbol{y} < \mathbf{0x80000000} \div \boldsymbol{x}$
$\boldsymbol{x} \leq \mathbf{0}$	$\boldsymbol{x} < \mathbf{0x80000000} \div \boldsymbol{y}$	$\boldsymbol{x} \neq \mathbf{0} \;\&\; \boldsymbol{y} < \mathbf{0x7FFFFFFF} \div \boldsymbol{x}$

This test is awkward to implement because of the four cases. It is difficult to unify the expressions very much because of problems with overflow and with not being able to represent the number $+2^{31}$.

The test can be simplified if unsigned division is available. One can use the absolute values of $\boldsymbol{x}$ and $\boldsymbol{y}$, which are correctly represented under unsigned integer interpretation. The complete test can then be computed as shown below. The variable $\boldsymbol{c} = \mathbf{2^{31} - 1}$ if $\boldsymbol{x}$ and $\boldsymbol{y}$ have the same sign, and $\boldsymbol{c} = \mathbf{2^{31}}$ otherwise.

$$\begin{aligned}
&\boldsymbol{c} \leftarrow ((\boldsymbol{x} \equiv \boldsymbol{y}) \overset{s}{\gg} 31) + \mathbf{2^{31}} \\
&\boldsymbol{x} \leftarrow \mathrm{abs}(\boldsymbol{x}) \\
&\boldsymbol{y} \leftarrow \mathrm{abs}(\boldsymbol{y}) \\
&\boldsymbol{y} \neq \mathbf{0} \;\&\; \boldsymbol{x} \overset{u}{>} (\boldsymbol{c} \overset{u}{\div} \boldsymbol{y})
\end{aligned}$$

The *number of leading zeros* instruction may be used to give an estimate of whether or not $\boldsymbol{x} * \boldsymbol{y}$ will overflow, and the estimate may be refined to give an accurate determination. First, consider the multiplication of unsigned numbers. It is easy to show that if $\boldsymbol{x}$ and $\boldsymbol{y}$, as 32-bit quantities, have m and n leading 0's, respectively, then the 64-bit product has either $m + n$ or $m + n + 1$ leading 0's (or 64, if either $\boldsymbol{x} = \mathbf{0}$ or $\boldsymbol{y} = \mathbf{0}$). Overflow occurs if the 64-bit product has fewer than 32 leading 0's. Hence,

$\mathrm{nlz}(\boldsymbol{x}) + \mathrm{nlz}(\boldsymbol{y}) \geq 32$: Multiplication definitely does not overflow.

$\mathrm{nlz}(\boldsymbol{x}) + \mathrm{nlz}(\boldsymbol{y}) \leq 30$: Multiplication definitely does overflow.

For $\mathrm{nlz}(\boldsymbol{x}) + \mathrm{nlz}(\boldsymbol{y}) = 31$, overflow may or may not occur. In this case, the overflow assessment may be made by evaluating $t = x\lfloor y/2 \rfloor$. This will not overflow. Since xy is $2t$ or, if y is odd, $2t + x$, the product xy overflows if $t \geq 2^{31}$. These considerations lead to a plan for computing xy but branching to “overflow” if the product overflows. This plan is shown in Figure 2–2.

```
unsigned x, y, z, m, n, t;

m = nlz(x);
n = nlz(y);
if (m + n <= 30) goto overflow;
t = x*(y >> 1);
if ((int)t < 0) goto overflow;
z = t*2;
if (y & 1) {
   z = z + x;
   if (z < x) goto overflow;
}
// z is the correct product of x and y.
```

FIGURE 2–2. Determination of overflow of unsigned multiplication.

For the multiplication of signed integers, we can make a partial determination of whether or not overflow occurs from the number of leading 0's of nonnegative arguments, and the number of leading 1's of negative arguments. Let

$$\begin{aligned} \boldsymbol{m} &= \mathrm{nlz}(\boldsymbol{x}) + \mathrm{nlz}(\bar{\boldsymbol{x}}), \text{ and} \\ \boldsymbol{n} &= \mathrm{nlz}(\boldsymbol{y}) + \mathrm{nlz}(\bar{\boldsymbol{y}}). \end{aligned}$$

Then we have

$$\boldsymbol{m} + \boldsymbol{n} \ge \mathbf{34}\text{: Multiplication definitely does not overflow.}$$
$$\boldsymbol{m} + \boldsymbol{n} \le \mathbf{31}\text{: Multiplication definitely does overflow.}$$

There are two ambiguous cases: 32 and 33. The case $m + n = 33$ overflows only when both arguments are negative and the true product is exactly 2^{31} (machine result is -2^{31}), so it can be recognized by a test that the product has the correct sign (that is, overflow occurred if $\boldsymbol{m} \oplus \boldsymbol{n} \oplus (\boldsymbol{m} * \boldsymbol{n}) < \mathbf{0}$). When $m + n = 32$, the distinction is not so easily made.

We will not dwell on this further, except to note that an overflow estimate for signed multiplication can also be made based on $\mathrm{nlz}(\mathrm{abs}(\boldsymbol{x})) + \mathrm{nlz}(\mathrm{abs}(\boldsymbol{y}))$, but again there are two ambiguous cases (a sum of 31 or 32).

Division

For the signed division $\boldsymbol{x} \div \boldsymbol{y}$, overflow occurs if the following expression is **true**:

$$\boldsymbol{y} = \mathbf{0} \ | \ (\boldsymbol{x} = \mathbf{0x80000000} \ \& \ \boldsymbol{y} = \mathbf{-1})$$

Most machines signal overflow (or trap) for the indeterminate form $\mathbf{0} \div \mathbf{0}$.

Straightforward code for evaluating this expression, including a final branch to the overflow handling code, consists of seven instructions, three of which are branches. There do not seem to be any particularly good tricks to improve on this, but below are a few possibilities:

$$[\mathrm{abs}(\boldsymbol{y} \oplus \mathbf{0x80000000}) \ | \ (\mathrm{abs}(\boldsymbol{x}) \ \& \ \mathrm{abs}(\boldsymbol{y} \equiv \mathbf{0x80000000}))] < \mathbf{0}$$

That is, evaluate the large expression in brackets, and branch if it is less than 0. This executes in about nine instructions, counting the load of the constant and the final branch, on a machine that has the indicated instructions and that gets the "compare to 0" for free.

Some other possibilities are to first compute $\boldsymbol{z}$ from

$$\boldsymbol{z} \leftarrow (\boldsymbol{x} \oplus \mathbf{0x80000000}) \ | \ (\boldsymbol{y} + \mathbf{1})$$

(three instructions on many machines), and then do the test and branch on $\boldsymbol{y} = \mathbf{0} \mid \boldsymbol{z} = \mathbf{0}$ in one of the following ways:

$$((\boldsymbol{y} \mid -\boldsymbol{y}) \,\&\, (\boldsymbol{z} \mid -\boldsymbol{z})) \geq \mathbf{0}$$
$$(\mathrm{nabs}(\boldsymbol{y}) \,\&\, \mathrm{nabs}(\boldsymbol{z})) \geq \mathbf{0}$$
$$((\mathrm{nlz}(\boldsymbol{y}) \mid \mathrm{nlz}(\boldsymbol{z})) \overset{u}{\gg} \mathbf{5}) \neq \mathbf{0}$$

These execute in nine, seven, and eight instructions, respectively, on a machine that has the indicated instructions. The last line represents a good method for PowerPC.

For the unsigned division $\boldsymbol{x} \overset{u}{\div} \boldsymbol{y}$, overflow occurs if and only if $\boldsymbol{y} = \mathbf{0}$.

2–13 Condition Code Result of *Add, Subtract*, and *Multiply*

Many machines provide a "condition code" that characterizes the result of integer arithmetic operations. Often there is only one *add* instruction, and the characterization reflects the result for both unsigned and signed interpretation of the operands and result (but not for mixed types). The characterization usually consists of the following:

- Whether or not carry occurred (unsigned overflow)
- Whether or not signed overflow occurred
- Whether the 32-bit result, interpreted as a signed two's-complement integer and ignoring carry and overflow, is negative, 0, or positive

Some older machines give an indication of whether the infinite precision result (that is, 33-bit result for *add*'s and *subtract*'s) is positive, negative, or 0. However, this indication is not easily used by compilers of high-level languages, and so has fallen out of favor.

For addition, only nine of the 12 combinations of these events are possible. The ones that cannot occur are "no carry, overflow, result > 0," "no carry, overflow, result = 0," and "carry, overflow, result < 0." Thus four bits are, just barely, needed for the condition code. Two of the combinations are unique in the sense that only one value of inputs produces them: Adding 0 to itself is the only way to get "no carry, no overflow, result = 0," and adding the maximum negative number to itself is the only way to get "carry, overflow, result = 0." These remarks remain true if there is a "carry in"—that is, if we are computing $\boldsymbol{x} + \boldsymbol{y} + 1$.

For subtraction, let us assume that to compute $\boldsymbol{x} - y$ the machine actually computes $\boldsymbol{x} + \bar{\boldsymbol{y}} + 1$, with the carry produced as for an *add* (in this scheme the meaning of "carry" is reversed for subtraction, in that carry = 1 signifies that the result fits in a single word, and carry = 0 signifies that the result does not fit in a single word). Then for subtraction only seven combinations of events are possible.

The ones that cannot occur are the three that cannot occur for addition, plus “no carry, no overflow, result = 0” and “carry, overflow, result = 0.”

If a machine’s multiplier can produce a doubleword result, then two *multiply* instructions are desirable: one for signed and one for unsigned operands. (On a 4-bit machine, in hexadecimal, $\mathbf{F} \times \mathbf{F} = \mathbf{01}$ signed, and $\mathbf{F} \times \mathbf{F} = \mathbf{E1}$ unsigned). For these instructions, neither carry nor overflow can occur, in the sense that the result will always fit in a doubleword.

For a multiplication instruction that produces a one-word result (the low-order word of the doubleword result), let us take “carry” to mean that the result does not fit in a word with the operands and result interpreted as unsigned integers, and let us take “overflow” to mean that the result does not fit in a word with the operands and result interpreted as signed two’s-complement integers. Then again there are nine possible combinations of results, with the missing ones being “no carry, overflow, result > 0,” “no carry, overflow, result = 0,” and “carry, no overflow, result = 0.” Thus considering addition, subtraction, and multiplication together, ten combinations can occur.

2–14 Rotate Shifts

These are rather trivial. Perhaps surprisingly, this code works for *n* ranging from 0 to 32 inclusive, even if the shifts are mod-32.

$$\text{Rotate left } \boldsymbol{n}\text{:} \quad \boldsymbol{y} \leftarrow (\boldsymbol{x} \ll \boldsymbol{n}) \mid (\boldsymbol{x} \overset{u}{\gg} (\mathbf{32} - \boldsymbol{n}))$$

$$\text{Rotate right } \boldsymbol{n}\text{:} \quad \boldsymbol{y} \leftarrow (\boldsymbol{x} \overset{u}{\gg} \boldsymbol{n}) \mid (\boldsymbol{x} \ll (\mathbf{32} - \boldsymbol{n}))$$

2–15 Double-Length Add/Subtract

Using one of the expressions shown on page 29 for overflow of unsigned addition and subtraction, we can easily implement double-length addition and subtraction without accessing the machine’s carry bit. To illustrate with double-length addition, let the operands be $(\boldsymbol{x}_1, \boldsymbol{x}_0)$ and $(\boldsymbol{y}_1, \boldsymbol{y}_0)$, and the result be $(\boldsymbol{z}_1, \boldsymbol{z}_0)$. Subscript 1 denotes the most significant half, and subscript 0 the least significant. We assume that all 32 bits of the registers are used. The less significant words are unsigned quantities.

$$\begin{aligned} &\boldsymbol{z}_0 \leftarrow \boldsymbol{x}_0 + \boldsymbol{y}_0 \\ &\boldsymbol{c} \leftarrow [(\boldsymbol{x}_0 \mathbin{\&} \boldsymbol{y}_0) \mid ((\boldsymbol{x}_0 \mid \boldsymbol{y}_0) \mathbin{\&} \neg \boldsymbol{z}_0)] \overset{u}{\gg} 31 \\ &\boldsymbol{z}_1 \leftarrow \boldsymbol{x}_1 + \boldsymbol{y}_1 + \boldsymbol{c} \end{aligned}$$

This executes in nine instructions. The second line can be: $\boldsymbol{c} \leftarrow (\boldsymbol{z}_0 \overset{u}{<} \boldsymbol{x}_0)$, permitting a four-instruction solution on machines that have this comparison operator in a form that gives the result as a **1** or **0** in a register, such as the “SLTU” (*Set on Less Than Unsigned*) instruction on MIPS [MIPS].

Similar code for double-length subtraction $(\boldsymbol{x}-\boldsymbol{y})$ is

$$\begin{aligned}
&\boldsymbol{z}_0 \leftarrow \boldsymbol{x}_0 - \boldsymbol{y}_0\\
&\boldsymbol{b} \leftarrow [(\neg \boldsymbol{x}_0 \mathbin{\&} \boldsymbol{y}_0) \mid ((\boldsymbol{x}_0 \equiv \boldsymbol{y}_0) \mathbin{\&} \boldsymbol{z}_0)] \overset{u}{\gg} 31\\
&\boldsymbol{z}_1 \leftarrow \boldsymbol{x}_1 - \boldsymbol{y}_1 - \boldsymbol{b}
\end{aligned}$$

This executes in eight instructions on a machine that has a full set of logical instructions. The second line can be $\boldsymbol{b} \leftarrow (\boldsymbol{x}_0 \overset{u}{<} \boldsymbol{y}_0)$, permitting a four-instruction solution on machines that have the “SLTU” instruction.

Double-length addition and subtraction can be done in five instructions on most machines by representing the multiple-length data using only 31 bits of the least significant words, with the high-order bit being 0 except momentarily when it contains a carry or borrow bit.

2–16 Double-Length Shifts

Let $(\boldsymbol{x}_1, \boldsymbol{x}_0)$ be a pair of 32-bit words to be shifted left or right as if they were a single 64-bit quantity, with $\boldsymbol{x}_1$ being the most significant half. Let $(\boldsymbol{y}_1, \boldsymbol{y}_0)$ be the result, interpreted similarly. Assume the shift amount *n* is a variable ranging from 0 to 63. Assume further that the machine’s shift instructions are modulo 64 or greater. That is, a shift amount in the range 32 to 63 or –32 to –1 results in an all-0 word, unless the shift is a signed right shift, in which case the result is 32 sign bits from the word shifted. (This code will not work on the Intel x86 machines, which have mod-32 shifts.)

Under these assumptions the *shift left double* operation may be accomplished as follows (eight instructions):

$$\begin{aligned}
&\boldsymbol{y}_1 \leftarrow \boldsymbol{x}_1 \ll \boldsymbol{n} \mid \boldsymbol{x}_0 \overset{u}{\gg} (\mathbf{32} - \boldsymbol{n}) \mid \boldsymbol{x}_0 \ll (\boldsymbol{n} - \mathbf{32})\\
&\boldsymbol{y}_0 \leftarrow \boldsymbol{x}_0 \ll \boldsymbol{n}
\end{aligned}$$

The main connective in the first assignment must be *or*, not *plus*, to give the correct result when $n = 32$. If it is known that $0 \le n \le 32$, the last term of the first assignment may be omitted, giving a six-instruction solution.

Similarly, a *shift right double unsigned* operation may be done with

$$\begin{aligned}
&\boldsymbol{y}_0 \leftarrow \boldsymbol{x}_0 \overset{u}{\gg} \boldsymbol{n} \mid \boldsymbol{x}_1 \ll (\mathbf{32} - \boldsymbol{n}) \mid \boldsymbol{x}_1 \overset{u}{\gg} (\boldsymbol{n} - \mathbf{32})\\
&\boldsymbol{y}_1 \leftarrow \boldsymbol{x}_1 \overset{u}{\gg} \boldsymbol{n}
\end{aligned}$$

Shift right double signed is more difficult, because of an unwanted sign propagation in one of the terms. Straightforward code follows:

$$
\begin{aligned}
&\text{if } \boldsymbol{n} < \mathbf{32} \text{ then } \boldsymbol{y}_0 \leftarrow \boldsymbol{x}_0 \overset{u}{\gg} \boldsymbol{n} \mid \boldsymbol{x}_1 \ll (\mathbf{32} - \boldsymbol{n})\\
&\text{else } \boldsymbol{y}_0 \leftarrow \boldsymbol{x}_1 \overset{s}{\gg} (\boldsymbol{n} - \mathbf{32})\\
&\boldsymbol{y}_1 \leftarrow \boldsymbol{x}_1 \overset{s}{\gg} \boldsymbol{n}
\end{aligned}
$$

If your machine has the *conditional move* instructions, it is a simple matter to express this in branch-free code, in which form it takes eight instructions. If the conditional move instructions are not available, the operation may be done in ten instructions by using the familiar device of constructing a mask with the *shift right signed 31* instruction to mask the unwanted sign propagating term:

$$
\begin{aligned}
&\boldsymbol{y}_0 \leftarrow \boldsymbol{x}_0 \overset{u}{\gg} \boldsymbol{n} \mid \boldsymbol{x}_1 \ll (\mathbf{32} - \boldsymbol{n}) \mid [(\boldsymbol{x}_1 \overset{s}{\gg} (\boldsymbol{n} - \mathbf{32}))\ \&\ ((\mathbf{32} - \boldsymbol{n}) \overset{s}{\gg} 31)]\\
&\boldsymbol{y}_1 \leftarrow \boldsymbol{x}_1 \overset{s}{\gg} \boldsymbol{n}
\end{aligned}
$$

2–17 Multibyte *Add, Subtract, Absolute Value*

Some applications deal with arrays of short integers (usually bytes or halfwords), and often execution is faster if they are operated on a word at a time. For definiteness, the examples here deal with the case of four 1-byte integers packed into a word, but the techniques are easily adapted to other packings, such as a word containing a 12-bit integer and two 10-bit integers, and so on. These techniques are of greater value on 64-bit machines, because more work is done in parallel.

Addition must be done in a way that blocks the carries from one byte into another. This can be accomplished by the following two-step method:

1. Mask out the high-order bit of each byte of each operand and *add* (there will then be no carries across byte boundaries).
2. Fix up the high-order bit of each byte with a 1-bit *add* of the two operands and the carry into that bit.

The carry into the high-order bit of each byte is of course given by the high-order bit of each byte of the sum computed in step 1. The subsequent similar method works for subtraction:

Addition

$$
\begin{aligned}
&\boldsymbol{s} \leftarrow (\boldsymbol{x}\ \&\ \mathbf{0x7F7F7F7F}) + (\boldsymbol{y}\ \&\ \mathbf{0x7F7F7F7F})\\
&(\boldsymbol{s} \leftarrow ((\boldsymbol{x} \oplus \boldsymbol{y})\ \&\ \mathbf{0x80808080}) \oplus \boldsymbol{s})
\end{aligned}
$$

Subtraction

$$
\begin{aligned}
&\boldsymbol{d} \leftarrow (\boldsymbol{x} \mid \mathbf{0x80808080}) - (\boldsymbol{y}\ \&\ \mathbf{0x7F7F7F7F})\\
&\boldsymbol{d} \leftarrow ((\boldsymbol{x} \oplus \boldsymbol{y}) \mid \mathbf{0x7F7F7F7F}) \equiv \boldsymbol{d}
\end{aligned}
$$

These execute in eight instructions, counting the load of **0x7F7F7F7F**, on a machine that has a full set of logical instructions. (Change the *and* and *or* of **0x80808080** to *and not* and *or not*, respectively, of **0x7F7F7F7F**.)

There is a different technique for the case in which the word is divided into only two fields. In this case, addition can be done by means of a 32-bit addition followed by subtracting out the unwanted carry. On page 28 we noted that the expression $(x + y) \oplus x \oplus y$ gives the carries into each position. Using this and similar observations about subtraction gives the following code for adding/subtracting two halfwords modulo 2^{16} (seven instructions):

Addition	Subtraction
$s \leftarrow x + y$	$d \leftarrow x - y$
$c \leftarrow (s \oplus x \oplus y)$ & **0x00010000**	$b \leftarrow (d \oplus x \oplus y)$ & **0x00010000**
$s \leftarrow s - c$	$d \leftarrow d + b$

Multibyte *absolute value* is easily done by complementing and adding 1 to each byte that contains a negative integer (that is, has its high-order bit on). The following code sets each byte of y equal to the absolute value of each byte of x (eight instructions):

```
$a \leftarrow x \,\&\, \mathbf{0x80808080}$      // Isolate signs.
$b \leftarrow a \overset{u}{\gg} 7$               // Integer 1 where x is negative.
$m \leftarrow (a - b) \mid a$                     // 0xFF where x is negative.
$y \leftarrow (x \oplus m) + b$                   // Complement and add 1 where negative.
```

The third line could as well be $m \leftarrow a + a - b$. The addition of b in the fourth line cannot carry across byte boundaries, because the quantity $x \oplus m$ has a high-order 0 in each byte.

2–18 Doz, Max, Min

The “doz” function is “difference or zero,” defined as follows, for signed arguments:

$$\text{doz}(x, y) = \begin{cases} x - y, & x \ge y, \\ 0, & x < y. \end{cases}$$

It has been called “first grade subtraction,” because the result is 0 if you try to take away too much. We will use it to implement max(x, y) and min(x, y). In this connection it is important to note that $\text{doz}(x, y)$ can be negative; it is negative if the subtraction overflows. The *difference or zero* function can be used directly to implement the Fortran IDIM function, although in Fortran, results are generally undefined if overflow occurs.

There seems to be no very good way to implement doz$(\boldsymbol{x}, \boldsymbol{y})$, max$(\boldsymbol{x}, \boldsymbol{y})$, and min$(\boldsymbol{x}, \boldsymbol{y})$ in a branch-free way that is applicable to most computers. About the best we can do is to compute doz$(\boldsymbol{x}, \boldsymbol{y})$ using one of the expressions given on page 22 for the $\boldsymbol{x} < \boldsymbol{y}$ predicate, and then compute max$(\boldsymbol{x}, \boldsymbol{y})$ and min$(\boldsymbol{x}, \boldsymbol{y})$ from it, as follows:

$$\begin{aligned}
&\boldsymbol{d} \leftarrow \boldsymbol{x} - \boldsymbol{y}\\
&\text{doz}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{d} \mathbin{\&} [(\boldsymbol{d} \equiv ((\boldsymbol{x} \oplus \boldsymbol{y}) \mathbin{\&} (\boldsymbol{d} \oplus \boldsymbol{x}))) \overset{s}{\gg} 31]\\
&\text{max}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{y} + \text{doz}(\boldsymbol{x}, \boldsymbol{y})\\
&\text{min}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} - \text{doz}(\boldsymbol{x}, \boldsymbol{y})
\end{aligned}$$

This computes doz$(\boldsymbol{x}, \boldsymbol{y})$ in seven instructions if the machine has *equivalence,* or eight if not, and it computes max$(\boldsymbol{x}, \boldsymbol{y})$ or min$(\boldsymbol{x}, \boldsymbol{y})$ in one more instruction.

The following are unsigned versions of these functions.

$$\begin{aligned}
&\boldsymbol{d} \leftarrow \boldsymbol{x} - \boldsymbol{y}\\
&\text{dozu}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{d} \mathbin{\&} \neg[((\neg\boldsymbol{x} \mathbin{\&} \boldsymbol{y}) \mid ((\boldsymbol{x} \equiv \boldsymbol{y}) \mathbin{\&} \boldsymbol{d})) \overset{s}{\gg} 31]\\
&\text{maxu}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{y} + \text{dozu}(\boldsymbol{x}, \boldsymbol{y})\\
&\text{minu}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x} - \text{dozu}(\boldsymbol{x}, \boldsymbol{y})
\end{aligned}$$

The IBM RS/6000 computer, and its predecessor the 801, has doz$(\boldsymbol{x}, \boldsymbol{y})$ provided as a single instruction. It permits computing the max$(\boldsymbol{x}, \boldsymbol{y})$ and min$(\boldsymbol{x}, \boldsymbol{y})$ of signed integers in two instructions, and is occasionally useful in itself. Implementing max$(\boldsymbol{x}, \boldsymbol{y})$ and min$(\boldsymbol{x}, \boldsymbol{y})$ directly is more costly because the machine would then need paths from the output ports of the register file back to an input port, bypassing the ALU.

Machines that have *conditional move* can get destructive[2] max$(\boldsymbol{x}, \boldsymbol{y})$ and min$(\boldsymbol{x}, \boldsymbol{y})$ in two instructions. For example, on our full RISC, $\boldsymbol{x} \leftarrow \text{max}(\boldsymbol{x}, \boldsymbol{y})$ can be calculated as follows (we write the target register first):

```
cmplt  z,x,y        Set z = 1 if x < y, else 0.
movne  x,z,y        If z is nonzero, set x = y.
```

2–19 Exchanging Registers

A very old trick is that of exchanging the contents of two registers without using a third [IBM]:

$$\begin{aligned}
\boldsymbol{x} &\leftarrow \boldsymbol{x} \oplus \boldsymbol{y}\\
\boldsymbol{y} &\leftarrow \boldsymbol{y} \oplus \boldsymbol{x}\\
\boldsymbol{x} &\leftarrow \boldsymbol{x} \oplus \boldsymbol{y}
\end{aligned}$$

[2] A destructive operation is one that overwrites one or more of its arguments.

This works well on a two-address machine. The trick also works if $\oplus$ is replaced by the $\equiv$ logical operation (complement of *exclusive or*), and can be made to work in various ways with *add*’s and *subtract*’s:

$$\begin{array}{lll}
x \leftarrow x + y & \qquad x \leftarrow x - y & \qquad x \leftarrow y - x \\
y \leftarrow x - y & \qquad y \leftarrow y + x & \qquad y \leftarrow y - x \\
x \leftarrow x - y & \qquad x \leftarrow y - x & \qquad x \leftarrow x + y
\end{array}$$

Unfortunately, each of these has an instruction that is unsuitable for a two-address machine, unless the machine has “reverse subtract.”

This little trick can actually be useful in the application of double buffering, in which two pointers are swapped. The first instruction can be factored out of the loop in which the swap is done (although this negates the advantage of saving a register):

$$\begin{array}{rl}
\text{Outside the loop:} & t \leftarrow x \oplus y \\
\text{Inside the loop:} & x \leftarrow x \oplus t \\
 & y \leftarrow y \oplus t
\end{array}$$

Exchanging Corresponding Fields of Registers

The problem here is to exchange the contents of two registers x and y wherever a mask bit $m_i = 1,$ and to leave x and y unaltered wherever $m_i = 0.$ By “corresponding” fields, we mean that no shifting is required. The 1-bits of m need not be contiguous. The straightforward method is as follows:

$$\begin{array}{l}
x' \leftarrow (x \mathbin{\&} \bar{m}) \mid (y \mathbin{\&} m) \\
y \leftarrow (y \mathbin{\&} \bar{m}) \mid (x \mathbin{\&} m) \\
x \leftarrow x'
\end{array}$$

By using “temporaries” for the four *and* expressions, this can be seen to require seven instructions, assuming that either m or $\bar{m}$ can be loaded with a single instruction and the machine has *and not* as a single instruction. If the machine is capable of executing the four (independent) *and* expressions in parallel, the execution time is only three cycles.

A method that is probably better (five instructions, but four cycles on a machine with unlimited instruction-level parallelism) is shown in column (a) below. It is suggested by the “three *exclusive or*” code for exchanging registers.

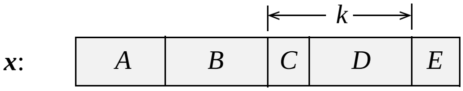
$$\begin{array}{lll}
\quad\text{(a)} & \qquad\quad\text{(b)} & \qquad\quad\text{(c)} \\
x \leftarrow x \oplus y & \qquad x \leftarrow x \equiv y & \qquad t \leftarrow (x \oplus y) \mathbin{\&} m \\
y \leftarrow y \oplus (x \mathbin{\&} m) & \qquad y \leftarrow y \equiv (x \mid \bar{m}) & \qquad x \leftarrow x \oplus t \\
x \leftarrow x \oplus y & \qquad x \leftarrow x \equiv y & \qquad y \leftarrow y \oplus t
\end{array}$$

The steps in column (b) do the same exchange as that of column (a), but column (b) is useful if $\boldsymbol{m}$ does not fit in an immediate field but $\bar{\boldsymbol{m}}$ does, and the machine has the *equivalence* instruction.

Still another method is shown in column (c) above [GLS1]. It also takes five instructions (again assuming one instruction must be used to load $\boldsymbol{m}$ into a register), but executes in only three cycles on a machine with sufficient instruction-level parallelism.

Exchanging Two Fields of the Same Register

Assume a register $\boldsymbol{x}$ has two fields (of the same length) that are to be swapped, without altering other bits in the register. That is, the object is to swap fields B and D, without altering fields A, C, and E, in the computer word illustrated below. The fields are separated by a shift distance k.



Straightforward code would shift D and B to their new positions, and combine the words with *and* and *or* operations, as follows:

$$\begin{aligned} \boldsymbol{t}_1 &= (\boldsymbol{x} \,\&\, \boldsymbol{m}) \ll k \\ \boldsymbol{t}_2 &= (\boldsymbol{x} \overset{u}{\gg} k) \,\&\, \boldsymbol{m} \\ \boldsymbol{x}' &= (\boldsymbol{x} \,\&\, \boldsymbol{m}') \mid \boldsymbol{t}_1 \mid \boldsymbol{t}_2 \end{aligned}$$

Here, $\boldsymbol{m}$ is a mask with 1’s in field D (and 0’s elsewhere), and $\boldsymbol{m}'$ is a mask with 1’s in fields A, C, and E. This code requires nine instructions and four cycles on a machine with unlimited instruction-level parallelism, allowing for two instructions to load the two masks.

A method that requires only seven instructions and executes in five cycles, under the same assumptions, is shown below [GLS1]. It is similar to the code in column (c) on page 39 for interchanging corresponding fields of two registers. Again, $\boldsymbol{m}$ is a mask that isolates field D.

$$\begin{aligned} \boldsymbol{t}_1 &= [\boldsymbol{x} \oplus (\boldsymbol{x} \overset{u}{\gg} k)] \,\&\, \boldsymbol{m} \\ \boldsymbol{t}_2 &= \boldsymbol{t}_1 \ll k \\ \boldsymbol{x}' &= \boldsymbol{x} \oplus \boldsymbol{t}_1 \oplus \boldsymbol{t}_2 \end{aligned}$$

The idea is that $\boldsymbol{t}_1$ contains $B \oplus D$ in position D (and 0’s elsewhere), and $\boldsymbol{t}_2$ contains $B \oplus D$ in position B. This code, and the straightforward code given earlier, work correctly if B and D are “split fields”—that is, if the 1-bits of mask $\boldsymbol{m}$ are not contiguous.

Conditional Exchange

The exchange methods of the preceding two sections that are based on *exclusive or* degenerate into no-operations if the mask $\boldsymbol{m}$ is 0. Hence they can perform an exchange of entire registers, or of corresponding fields of two registers, or of two fields of the same register, if $\boldsymbol{m}$ is set to all 1's if some condition c is true, and to all 0's if c is false. This gives branch-free code if $\boldsymbol{m}$ can be set up without branching.

2–20 Alternating among Two or More Values

Suppose a variable x can have only two possible values a and b, and you wish to assign to x the value other than its current one, and you wish your code to be independent of the values of a and b. For example, in a compiler x might be an opcode that is known to be either *branch true* or *branch false*, and whichever it is, you want to switch it to the other. The values of the opcodes *branch true* and *branch false* are arbitrary, probably defined by a C `#define` or `enum` declaration in a header file.

The straightforward code to do the switch is

```
if (x == a) x = b;
else x = a;
```

or, as often seen in C programs,

```
x = x == a ? b : a;
```

A far better (at least more efficient) way to code it is either

$$\boldsymbol{x} \leftarrow \boldsymbol{a} + \boldsymbol{b} - \boldsymbol{x}, \quad \text{or}$$
$$\boldsymbol{x} \leftarrow \boldsymbol{a} \oplus \boldsymbol{b} \oplus \boldsymbol{x}.$$

If $\boldsymbol{a}$ and $\boldsymbol{b}$ are constants, these require only one or two basic RISC instructions. Of course, overflow in calculating $\boldsymbol{a} + \boldsymbol{b}$ can be ignored.

This raises the question: Is there some particularly efficient way to cycle among three or more values? That is, given three arbitrary but distinct constants a, b, and c, we seek an easy-to-evaluate function f that satisfies

$$\begin{aligned} f(a) &= b, \\ f(b) &= c, \quad \text{and} \\ f(c) &= a. \end{aligned}$$

It is perhaps interesting to note that there is always a polynomial for such a function. For the case of three constants,

$$f(x) = \frac{(x-a)(x-b)}{(c-a)(c-b)}a + \frac{(x-b)(x-c)}{(a-b)(a-c)}b + \frac{(x-c)(x-a)}{(b-c)(b-a)}c. \tag{4}$$

(The idea is that if $x = a,$ the first and last terms vanish, and the middle term simplifies to b, and so on.) This requires 14 arithmetic operations to evaluate, and, for arbitrary a, b, and c, the intermediate results exceed the computer's word size. But it is just a quadratic; if written in the usual form for a polynomial and evaluated using Horner's rule,[3] it would require only five arithmetic operations (four for a quadratic with integer coefficients, plus one for a final division). Rearranging Equation (4) accordingly gives

$$\begin{aligned} f(x) = {} & \frac{1}{(a-b)(a-c)(b-c)}\{[(a-b)a + (b-c)b + (c-a)c]x^2 \\ & + [(a-b)b^2 + (b-c)c^2 + (c-a)a^2]x \\ & + [(a-b)a^2b + (b-c)b^2c + (c-a)ac^2]\}. \end{aligned}$$

This is getting too complicated to be interesting (or practical).

Another method, similar to Equation (4) in that just one of the three terms survives, is:

$$f(\boldsymbol{x}) = ((-(\boldsymbol{x} = \boldsymbol{c}))\ \&\ \boldsymbol{a}) + ((-(\boldsymbol{x} = \boldsymbol{a}))\ \&\ \boldsymbol{b}) + ((-(\boldsymbol{x} = \boldsymbol{b}))\ \&\ \boldsymbol{c}).$$

This takes 11 instructions if the machine has the *equal* predicate, not counting loads of constants. Because the two addition operations are combining two 0 values with a nonzero, they can be replaced with *or* or *exclusive or* operations.

The formula be simplified by precalculating $\boldsymbol{a} - \boldsymbol{c}$ and $\boldsymbol{b} - \boldsymbol{c},$ and then using [GLS1]:

$$\begin{aligned} f(\boldsymbol{x}) &= ((-(\boldsymbol{x} = \boldsymbol{c}))\ \&\ (\boldsymbol{a} - \boldsymbol{c})) + ((-(\boldsymbol{x} = \boldsymbol{a}))\ \&\ (\boldsymbol{b} - \boldsymbol{c})) + \boldsymbol{c}, \quad \text{or} \\ f(\boldsymbol{x}) &= ((-(\boldsymbol{x} = \boldsymbol{c}))\ \&\ (\boldsymbol{a} \oplus \boldsymbol{c})) \oplus ((-(\boldsymbol{x} = \boldsymbol{a}))\ \&\ (\boldsymbol{b} \oplus \boldsymbol{c})) \oplus \boldsymbol{c}, \end{aligned}$$

[3] Horner's rule simply factors out x. For example, it evaluates the fourth degree polynomial $ax^4 + bx^3 + cx^2 + dx + e$ as $x(x(x(ax + b) + c) + d) + e.$ For a polynomial of degree n it takes n multiplications and n additions, and it is very suitable for the *multiply-add* instruction.

Each of these operations takes eight instructions. But on most machines these are probably no better than the straightforward C code shown below, which executes in four to six instructions for small `a`, `b`, and `c`.

```
if (x == a) x = b;
else if (x == b) x = c;
else x = a;
```

Pursuing this matter, there is an ingenious branch-free method of cycling among three values on machines that do not have comparison predicate instructions [GLS1]. It executes in eight instructions on most machines.

Because $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are distinct, there are two bit positions n_1 and n_2 where the bits of $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are not all the same, and where the "odd one out" (the one whose bit differs in that position from the other two) is different in positions n_1 and n_2. This is illustrated below for the values 21, 31, and 20, shown in binary.

$$\begin{array}{cccccc}
1 & 0 & 1 & 0 & 1 & \quad \boldsymbol{c} \\
1 & 1 & 1 & 1 & 1 & \quad \boldsymbol{a} \\
1 & 0 & 1 & 0 & 0 & \quad \boldsymbol{b} \\
 & n_1 & & & n_2 &
\end{array}$$

Without loss of generality, rename $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ so that $\boldsymbol{a}$ has the odd one out in position n_1 and $\boldsymbol{b}$ has the odd one out in position n_2, as shown above. Then there are two possibilities for the values of the bits at position n_1, namely $(\boldsymbol{a}_{n_1}, \boldsymbol{b}_{n_1}, \boldsymbol{c}_{n_1}) = (0, 1, 1)$ or $(1, 0, 0)$. Similarly, there are two possibilities for the bits at position n_2, namely $(\boldsymbol{a}_{n_2}, \boldsymbol{b}_{n_2}, \boldsymbol{c}_{n_2}) = (0, 1, 0)$ or $(1, 0, 1)$. This makes four cases in all, and formulas for each of these cases are shown below.

Case 1, $(\boldsymbol{a}_{n_1}, \boldsymbol{b}_{n_1}, \boldsymbol{c}_{n_1}) = (0, 1, 1)$, $(\boldsymbol{a}_{n_2}, \boldsymbol{b}_{n_2}, \boldsymbol{c}_{n_2}) = (0, 1, 0)$:

$$f(\boldsymbol{x}) = \boldsymbol{x}_{n_1} * (\boldsymbol{a} - \boldsymbol{b}) + \boldsymbol{x}_{n_2} * (\boldsymbol{c} - \boldsymbol{a}) + \boldsymbol{b}.$$

Case 2, $(\boldsymbol{a}_{n_1}, \boldsymbol{b}_{n_1}, \boldsymbol{c}_{n_1}) = (0, 1, 1)$, $(\boldsymbol{a}_{n_2}, \boldsymbol{b}_{n_2}, \boldsymbol{c}_{n_2}) = (1, 0, 1)$:

$$f(\boldsymbol{x}) = \boldsymbol{x}_{n_1} * (\boldsymbol{a} - \boldsymbol{b}) + \boldsymbol{x}_{n_2} * (\boldsymbol{a} - \boldsymbol{c}) + (\boldsymbol{b} + \boldsymbol{c} - \boldsymbol{a}).$$

Case 3, $(\boldsymbol{a}_{n_1}, \boldsymbol{b}_{n_1}, \boldsymbol{c}_{n_1}) = (1, 0, 0)$, $(\boldsymbol{a}_{n_2}, \boldsymbol{b}_{n_2}, \boldsymbol{c}_{n_2}) = (0, 1, 0)$:

$$f(\boldsymbol{x}) = \boldsymbol{x}_{n_1} * (\boldsymbol{b} - \boldsymbol{a}) + \boldsymbol{x}_{n_2} * (\boldsymbol{c} - \boldsymbol{a}) + \boldsymbol{a}.$$

Case 4, $(\boldsymbol{a}_{n_1}, \boldsymbol{b}_{n_1}, \boldsymbol{c}_{n_1}) = (1, 0, 0)$, $(\boldsymbol{a}_{n_2}, \boldsymbol{b}_{n_2}, \boldsymbol{c}_{n_2}) = (1, 0, 1)$:

$$f(\boldsymbol{x}) = \boldsymbol{x}_{n_1} * (\boldsymbol{b} - \boldsymbol{a}) + \boldsymbol{x}_{n_2} * (\boldsymbol{a} - \boldsymbol{c}) + \boldsymbol{c}.$$

In these formulas, the left operand of each multiplication is a single bit. A multiplication by 0 or 1 may be converted into an *and* with a value of 0 or all 1's. Thus the formulas can be rewritten as illustrated below for the first formula.

$$f(\boldsymbol{x}) = ((\boldsymbol{x} \ll (31 - n_1)) \overset{s}{\gg} 31) \& (\boldsymbol{a} - \boldsymbol{b}) + ((\boldsymbol{x} \ll (31 - n_2)) \overset{s}{\gg} 31) \& (\boldsymbol{c} - \boldsymbol{a}) + \boldsymbol{b}$$

Because all variables except $\boldsymbol{x}$ are constants, this can be evaluated in eight instructions on the basic RISC. Here again the additions and subtractions can be replaced with *exclusive or.*

This idea can be extended to cycling among four or more constants. The essence of the idea is to find bit positions $n_1,\ n_2,\ \ldots,$ at which the bits uniquely identify the constants. For four constants, three bit positions always suffice. Then (for four constants) solve the following equation for *s*, *t*, *u*, and *v* (that is, solve the system of four linear equations in which $f(x)$ is *a*, *b*, *c*, or *d*, and the coefficients x_{n_i} are 0 or 1):

$$f(x) \ = \ x_{n_1}s + x_{n_2}t + x_{n_3}u + v$$

If the four constants are uniquely identified by only two bit positions, the equation to solve is

$$f(x) \ = \ x_{n_1}s + x_{n_2}t + x_{n_1}x_{n_2}u + v.$$