ГЛАВА 2

ОСНОВЫ

2.1. Манипуляции с младшими битами

Многие из приведенных в этом разделе формул используются в следующих главах.

Для того чтобы обнулить крайний справа единичный бит (например, 01011000 $\Rightarrow$ 01010000), используется формула

$$x \,\&\, (x-1)\,.$$

Эту формулу можно применять для проверки того, является ли беззнаковое целое степенью 2 (путем проверки результата вычислений на равенство 0).

Для проверки того, имеет ли беззнаковое целое число вид 2^n–1, можно воспользоваться следующей формулой:

$$x \,\&\, (x+1)\,.$$

Чтобы выделить в слове крайний справа единичный бит (например, 01011000 $\Rightarrow$ 00001000, если такого бита нет, возвращает 0), используется формула

$$x \,\&\, (-x)\,.$$

Чтобы выделить в слове крайний справа нулевой бит (например, 10100111 $\Rightarrow$ 00001000, если такого бита нет, возвращает 0), используется формула

$$\neg x \,\&\, (x+1)\,.$$

Для создания маски, идентифицирующей завершающие нулевые биты, можно воспользоваться одной из приведенных ниже формул. (Например, 01011000 $\Rightarrow$ 00000111, если исходное слово равно 0, все биты маски будут равны 1.)

$$\neg x \,\&\, (x-1)\text{, или}$$

$$\neg(x \mid -x)\text{, или}$$

$$(x \,\&\, -x)-1$$

Первая из этих формул позволяет распараллелить вычисления на уровне команд.

Приведенная далее формула используется для создания маски, идентифицирующей крайний справа единичный бит и все завершающие нулевые биты (например, 01011000 $\Rightarrow$ 00001111, если исходное слово равно 0, все биты маски будут равны 1).

$$x \oplus (x-1)$$

Очередная формула распространяет вправо крайний правый единичный бит (например, 01011000 $\Rightarrow$ 01011111, если исходное слово равно 0, все биты результата будут равны 1).

$$x \mid (x-1)$$

Для обнуления крайней справа непрерывной подстроки единичных битов (например, 01011000 $\Rightarrow$ 01000000) можно использовать формулу

$$\big((x \mid (x-1))+1\big) \,\&\, x\,.$$

Эта формула позволяет проверить, имеет ли положительное целое число вид $2^j–2^k$, где $j \ge k \ge 0$ (в этом случае формула возвращает нулевой результат).

Для всех приведенных выше формул выполняется принцип дуализма: если в описаниях формул единицы заменить нулями (и нули единицами), а в самих формулах заменить $\boldsymbol{x}-1$ на $\boldsymbol{x}+1$ и обратно, $-\boldsymbol{x}$ на $\neg(\boldsymbol{x}+1)$, & на | и | на &, оставив без изменений только $\boldsymbol{x}$ и $\neg\boldsymbol{x}$, то в результате получатся правильные выражения и описания. Например, после замен и преобразований в первой формуле получим дуальную ей формулу, которая будет выглядеть следующим образом.

Для того чтобы установить крайний справа нулевой бит (например, 10100111 ⇒ 10101111, если исходное слово равно 0, возвращает слово, состоящее из одних единиц), используется следующая формула:

$$\boldsymbol{x} \mid (\boldsymbol{x}+1)\ .$$

Имеется простая проверка того, можно ли реализовать заданную функцию в виде последовательности команд *сложения*, *вычитания*, *и*, *или* и *отрицания* [59]. К перечисленным выше командам можно добавить другие команды из основного множества, которые, в свою очередь, являются комбинацией команд *сложения*, *вычитания*, *и*, *или* и *не*, например можно добавить команду *сдвиг влево* на определенную величину (которая эквивалентна последовательности команд *сложения*) или команду *умножения*. Команды, которые не могут быть выражены через команды *сложения, вычитания* и другие из основного списка, исключаются из рассмотрения. Критерий проверки следует из приведенной теоремы.

Теорема. *Функция, отображающая слова в слова, может быть реализована посредством операций побитового сложения, вычитания, и, или, отрицания тогда и только тогда, когда каждый бит результата зависит только от битов исходных операндов в той же позиции и правее нее.*

Иными словами, представьте ситуацию, когда вы вычисляете младший бит результирующего значения, исходя только из младших битов операндов. Затем вычисляется второй справа бит результата, для чего используются только два младших бита операндов, и т.д. Если таким образом можно получить результирующее значение, то данная функция может быть вычислена при помощи последовательности команд *сложения*, *и* и т.п. Если же таким образом (справа налево) вычислить функцию невозможно, то ее невозможно реализовать в виде последовательности указанных команд.

Наибольший интерес представляет утверждение, что любая из функций *сложения*, *вычитания*, *и*, *или* и *не* может быть вычислена справа налево, так что любая комбинация этих функций также обладает этим свойством, т.е. может быть вычислена справа налево.

Доказательство второй части теоремы достаточно громоздко, так что просто приведем конкретный пример. Предположим, что функция двух переменных x и y может быть вычислена справа налево, и пусть второй бит результата r вычисляется следующим образом:

$$\boldsymbol{r}_2 = \boldsymbol{x}_2 \mid (\boldsymbol{x}_0 \,\&\, \boldsymbol{y}_1)\ . \qquad (1)$$

(Биты пронумерованы справа налево от 0 до 31.) Так как второй бит результата является функцией только крайних справа битов исходных операндов (бита номер 2 и битов младше его), он также вычисляется справа налево.

Запишем машинные слова $\boldsymbol{x}$, $\boldsymbol{x}$, сдвинутое на два бита влево, и $\boldsymbol{y}$, сдвинутое на один бит влево, как показано ниже. Добавим к записи маску, которая выделяет второй бит.

$$\begin{array}{cccccc} \boldsymbol{x}_{31} & \boldsymbol{x}_{30} & \dots & \boldsymbol{x}_3 & \boldsymbol{x}_2 & \boldsymbol{x}_1 & \boldsymbol{x}_0 \\ \boldsymbol{x}_{29} & \boldsymbol{x}_{28} & \dots & \boldsymbol{x}_1 & \boldsymbol{x}_0 & 0 & 0 \\ \boldsymbol{y}_{30} & \boldsymbol{y}_{29} & \dots & \boldsymbol{y}_2 & \boldsymbol{y}_1 & \boldsymbol{y}_0 & 0 \\ 0 & 0 & \dots & 0 & 1 & 0 & 0 \\ 0 & 0 & \dots & 0 & \boldsymbol{r}_2 & 0 & 0 \end{array}$$

К строкам 2 и 3 применим операцию *побитового и*, затем, согласно формуле (1), к получившемуся слову применим операцию *побитового или* со строкой 1, после чего к полученному результату и маске (строка 4) применим операцию *побитового и*. В полученном таким образом слове все биты, кроме второго, будут нулевыми. Выполним подобные вычисления для получения остальных битов результата и объединим все 32 слова с помощью операции *или*. В итоге получим искомую функцию.

Такое построение не дает эффективной программы вычисления значений функции, всего лишь показывая, что искомая функция может быть выражена с использованием команд из базового списка.

Используя эту теорему, можно сразу же увидеть, что не существует такой последовательности команд, которая обнуляла бы в слове крайний слева единичный бит, так как для этого необходимо просмотреть биты, находящиеся слева от него (чтобы точно знать, что это действительно крайний слева единичный бит). Аналогично, не существует такой последовательности операций, которая бы давала сдвиг вправо, циклический сдвиг или сдвиг влево на переменную величину, а также могла подсчитать количество завершающих нулевых битов в слове (при подсчете завершающих нулевых битов младший бит результата должен быть равен 1, если это количество нечетно, и 0, если четно; так что необходимо просмотреть биты слева от младшего, чтобы выяснить его значение).

Одним из применений рассмотренной сортировки битов является задача поиска следующего числа, которое больше заданного, но имеет такое же количество единичных битов. Зачем это может понадобиться? Эта задача возникает при использовании битовых строк для представления подмножеств. Возможные члены множества перечислены в одномерном массиве, а подмножество представляет собой слово или последовательность слов, в которых бит i установлен равным 1, если i-й элемент принадлежит этому подмножеству. Тогда объединение множеств вычисляется с помощью *побитовой операции или* над битовыми строками, пересечение — с помощью операции *и* и т.д.

Иногда требуется выполнить определенные действия над всеми подмножествами заданной длины, что легко сделать с помощью функции, отображающей представленное числом заданное подмножество в следующее большее число, содержащее такое же количество единичных битов, что и исходное (строка подмножества интерпретируется как целое число).

Компактный алгоритм для этой задачи составлен Р.У. Госпером (R.W. Gosper) [25, item 175][1]. Пусть имеется некоторое слово $\boldsymbol{x}$, представляющее собой подмножество. Идея в том, чтобы сначала найти в $\boldsymbol{x}$ крайнюю справа группу смежных единичных битов, следующую за нулевыми битами, а затем увеличить представленную этой группой величину таким образом, чтобы количество единичных битов осталось неизменным. Например, строка xxx011110000, где xxx — произвольные биты, преобразуется в строку xxx100000111. Алгоритм работает следующим образом. Сначала в $\boldsymbol{x}$ определяется самый младший единичный бит, т.е. вычисляется $s = \boldsymbol{x} \,\&\, -\boldsymbol{x}$ (в результате получаем 000000010000). Затем эта строка складывается с $\boldsymbol{x}$, давая $\boldsymbol{r}$=xxx100000000, в которой

[1] Вариант этого алгоритма имеется в [32], разд.7.6.7.

получен первый единичный бит результата. Чтобы получить остальные биты, установим $n-1$ младших битов слова равными 1, где n — размер крайней правой группы единичных битов в исходном слове $\boldsymbol{x}$. Для этого над $\boldsymbol{r}$ и $\boldsymbol{x}$ выполняется операция *исключающего или*, что в нашем примере дает строку 0001 1111 0000.

В полученной строке содержится больше единичных битов, чем в исходной, поэтому выполним еще одно преобразование. Поделим искомое число на $\boldsymbol{s}$, что эквивалентно его сдвигу вправо ($\boldsymbol{s}$ — степень 2), сдвинем его вправо еще на два разряда, что отбросит еще два нежелательных бита, и выполним операцию побитового *или* над полученным числом и $\boldsymbol{r}$.

В записи компьютерной алгебры результат $\boldsymbol{y}$ получается следующим образом.

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

В листинге 2.1 представлена законченная процедура на языке С, выполняющая семь базовых RISC-команд, одной из которых является команда деления. (Данная процедура неприменима для $\boldsymbol{x}=0$, так как в этом случае получается деление на 0.)

Листинг 2.1. Вычисление следующего большего числа с тем же количеством единичных битов

```
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
}
```

Если деление выполняется медленно, но у вас есть быстрый способ вычисления *количества завершающих нулевых битов* $\text{ntz}(\boldsymbol{x})$, *количества ведущих нулевых битов* $\text{nlz}(\boldsymbol{x})$ или функции *степени заполнения* $\text{pop}(\boldsymbol{x})$ (количество единичных битов в $\boldsymbol{x}$), то последнюю строку в (2) можно заменить одной из приведенных ниже.

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

2.2. Сложение и логические операции

Предполагается, что читатель знаком с элементарными тождествами обычной и булевой алгебры. Ниже приводится набор аналогичных тождеств для операций сложения и вычитания в комбинации с логическими операциями.

а) $-\boldsymbol{x} = \neg\boldsymbol{x} + 1$

б) $\phantom{-\boldsymbol{x}} = \neg(\boldsymbol{x} - 1)$

в) $\neg\boldsymbol{x} = -\boldsymbol{x} - 1$

$$
\begin{array}{lrcl}
\text{г)} & -\neg x & = & x+1 \\
\text{д)} & \neg -x & = & x-1 \\
\text{е)} & x+y & = & x-\neg y-1 \\
\text{ж)} & & = & (x \oplus y)+2(x \,\&\, y) \\
\text{з)} & & = & (x \mid y)+(x \,\&\, y) \\
\text{и)} & & = & 2(x \mid y)-(x \oplus y) \\
\text{к)} & x-y & = & x+\neg y+1 \\
\text{л)} & & = & (x \oplus y)-2(\neg x \,\&\, y) \\
\text{м)} & & = & (x \,\&\, \neg y)-(\neg x \,\&\, y) \\
\text{н)} & & = & 2(x \,\&\, \neg y)-(x \oplus y) \\
\text{о)} & x \oplus y & = & (x \mid y)-(x \,\&\, y) \\
\text{п)} & x \,\&\, \neg y & = & (x \mid y)-y \\
\text{р)} & & = & x-(x \,\&\, y) \\
\text{с)} & \neg(x-y) & = & y-x-1 \\
\text{т)} & & = & \neg x+y \\
\text{у)} & x \equiv y & = & (x \,\&\, y)-(x \mid y)-1 \\
\text{ф)} & & = & (x \,\&\, y)+\neg(x \mid y) \\
\text{х)} & x \mid y & = & (x \,\&\, \neg y)+y \\
\text{ц)} & x \,\&\, y & = & (\neg x \mid y)-\neg x
\end{array}
$$

Равенство (г) можно повторно применять к самому себе, например: $-\neg-\neg x = x+2$ и т.д. Аналогично, после повторного использования равенства (д) получаем: $\neg-\neg- x = x-2$. Таким образом, чтобы добавить к x или вычесть из него некоторую константу, можно воспользоваться только этими выражениями дополнения.

Равенство (е) дуально равенству (к), которое представляет собой известное отношение, позволяющее получить вычитающее устройство из сумматора.

Равенства (ж) и (з) взяты из [25, item 23]. В равенстве (ж) сумма x и y вычисляется следующим образом: сначала x и y суммируются без учета переносов $(x \oplus y)$, а затем к полученному результату добавляются переносы. Равенство (з) изменяет операнды перед сложением так, что в любом разряде все комбинации типа $0+1$ заменяются на $1+0$.

Можно показать, что при обычном сложении двоичных чисел с независимыми битами вероятность возникновения переноса в любом разряде равна 0.5. Однако, если при создании сумматора используется предварительная подготовка входных данных, как при сложении по формуле (ж), вероятность появления переноса становится равной 0.25. При разработке сумматора вероятность переноса значения не имеет; самой важной характеристикой при этом является максимальное количество логических схем, через которые может потребоваться пройти переносу, а использование равенства (ж) позволяет свести это количество всего лишь к одной схеме.

Равенства (л) и (м) дуальны равенствам (ж) и (з) для вычитания. В (л) сначала выполняется вычитание без заемов $(x \oplus y)$, а затем из полученного результата вычитаются заемы. Аналогично, равенство (м) изменяет операнды перед вычитанием таким образом, что все комбинации типа $1-1$ заменяются на $0-0$.

Равенство (о) показывает, как реализовать *исключающее или*, используя всего три базовых RISC-команды. Использование логики *и–или–не* потребует выполнения четырех команд $\left((\boldsymbol{x} \mid \boldsymbol{y}) \& \neg(\boldsymbol{x} \,\&\, \boldsymbol{y})\right)$. Аналогично, равенства (х) и (ц) реализуют операции *и* и *или* с помощью трех элементарных команд (хотя по законам де Моргана (DeMorgan) их требуется четыре.

2.3. Неравенства с логическими и арифметическими выражениями

Неравенства с двоичными логическими выражениями, значения которых интерпретируются как целые беззнаковые величины, выводятся практически тривиально. Вот пара примеров:

$$(\boldsymbol{x} \oplus \boldsymbol{y}) \overset{u}{\le} (\boldsymbol{x} \mid \boldsymbol{y}) \text{ и}$$

$$(\boldsymbol{x} \,\&\, \boldsymbol{y}) \overset{u}{\le} (\boldsymbol{x} \equiv \boldsymbol{y}).$$

Их легко получить из табл. 2.1, в которой представлены все логические операции.

Таблица 2.1. Результаты работы 16 логических операций

x	y	0	$x\&y$	$x\&\neg y$	x	$\neg x\&x$	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

Пусть функции $f(x,y)$ и $g(x,y)$ представлены двумя столбцами табл. 2.1. Если в каждой строке, где $f(x,y)$ равна 1, $g(x,y)$ также равна 1, то для любых значений (x,y) выполняется соотношение $f(x,y) \overset{u}{\le} g(x,y)$. Очевидно, что это неравенство справедливо и для побитовых логических операций. Легко вывести и более сложные соотношения (многие из которых тривиальны), например: $(x \,\&\, y) \overset{u}{\le} x \overset{u}{\le} (x / \neg y)$ и т.п. Если же значения функций $f(x,y)$ и $g(x,y)$ в одной строке таблицы равны 0 и 1, а в другой — 1 и 0 соответственно, то между этими логическими функциями не существует отношения неравенства. Таким образом, всегда можно выяснить, выполняется ли для каких-то функций f и g соотношение $f(x,y) \overset{u}{\le} g(x,y)$ или нет.

При работе с неравенствами требуется внимание. Например, в обычной арифметике, если $x + y \le a$ и $z \le x$, то $z + y \le a$. Но этот вывод становится неверным, если “+” заменить оператором *или*.

Неравенства, включающие как логические, так и арифметические выражения, более интересны. Ниже приведены некоторые из них.

а) $(x \mid y) \overset{u}{\geq} \max(x, y)$

б) $(x \,\&\, y) \overset{u}{\leq} \min(x, y)$

в) $(x \mid y) \overset{u}{\leq} x + y$ Если сложение не вызывает переполнения

г) $(x \mid y) \overset{u}{>} x + y$ Если сложение вызывает переполнение

д) $|x - y| \overset{u}{\leq} (x \oplus y)$

Доказать все эти неравенства достаточно просто, за исключением, возможно, только неравенства (д). Под $|x - y|$ подразумевается абсолютное значение $x - y$, которое может быть вычислено в области беззнаковых чисел как $\max(x, y) - \min(x, y)$. Это неравенство можно доказать по индукции по длинам x и y (доказать неравенство будет проще, если длины x и y расширять справа налево, а не наоборот).

2.4. Абсолютное значение

Если нет отдельной команды вычисления абсолютного значения, ее можно реализовать тремя или четырьмя командами (без ветвления). Сначала вычисляется значение $y \leftarrow x \overset{s}{\gg} 31$, а затем одно из перечисленных ниже выражений (разумеется, $2x$ означает $x + x$, или $x \ll 1$).

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

Если у вас есть возможность быстрого умножения на переменную, значение которой равно ±1, можно поступить следующим образом:

$$\left(\left(x \overset{s}{\gg} 30\right) \mid \mathbf{1}\right) * x\,.$$

2.5. Распространение знака

Под “распространением знака” (*sign extension*) подразумевается наличие бита в определенной позиции слова, выступающего в роли бита знака, и распространение этого бита влево при игнорировании всех остальных битов слова. Стандартный способ решения этой задачи состоит в *логическом сдвиге влево*, за которым следует *знаковый сдвиг вправо*. Однако, если эти команды работают медленно или их вообще нет на вашем компьютере, можно поступить иначе. Ниже приведены примеры распространения влево бита в седьмой позиции.

$$\left((x + \mathbf{0x00000080}) \,\&\, \mathbf{0x000000FF}\right) - \mathbf{0x00000080}$$

$$\left((x \,\&\, \mathbf{0x000000FF}) \oplus \mathbf{0x00000080}\right) - \mathbf{0x00000080}$$

Вместо “+” можно использовать “–” или “⊕”. Если известно, что все ненужные старшие биты нулевые, вторая формула оказывается более практичной, так как в этом случае можно не выполнять команду *и*.

2.6. Знаковый сдвиг вправо на основе беззнакового сдвига

Если на машине нет команды *знакового сдвига вправо* (*shift right signed*), ее можно заменить другими командами. Первая формула взята из [21], вторая основана на той же идее, что и первая. На 64-разрядной машине первые четыре формулы применимы для $0 \le n \le 31$, а последняя — для $0 \le n \le 63$. В принципе она применима для любых n, где под “применима” подразумевается “рассматривает величину сдвига по тому же модулю, что и логический сдвиг”.

Для переменной n реализация каждой формулы требует пяти или шести команд из базового множества RISC-команд.

$$\left((x + \mathbf{0x80000000}) \overset{u}{\gg} n\right) - \left(\mathbf{0x80000000} \overset{u}{\gg} n\right)$$

$$t \leftarrow \mathbf{0x80000000} \overset{u}{\gg} n; \quad \left(\left(x \overset{u}{\gg} n\right) \oplus t\right) - t$$

$$t \leftarrow (x \,\&\, \mathbf{0x80000000}) \overset{u}{\gg} n; \quad \left(x \overset{u}{\gg} n\right) - \left(t + t\right)$$

$$\left(x \overset{u}{\gg} n\right) \mid \left(-\left(x \overset{u}{\gg} 31\right) \ll \mathbf{31} - n\right)$$

$$t \leftarrow -\left(x \overset{u}{\gg} 31\right); \quad \left(\left(x \oplus t\right) \overset{u}{\gg} n\right) \oplus t$$

В первых двух формулах выражение $\mathbf{0x80000000} \overset{u}{\gg} n$ можно заменить на $\mathbf{1} \ll \mathbf{31} - n$.

Если n — константа, то для реализации первых двух формул на большинстве машин требуется всего лишь три команды. Если $n = 31$, функция может быть вычислена с помощью двух команд: $-\left(x \overset{u}{\gg} 31\right)$.

2.7. Функция *sign*

Функция sign, или *signum*, определяется как

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

На большинстве машин ее можно вычислить с помощью четырех команд [29].

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

Если на машине нет команды *знакового сдвига вправо*, вместо нее можно использовать последнее выражение из раздела 2.6. В результате получается элегантная симметричная формула (вычисляемая за пять команд).

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

Наличие команд сравнения допускает решения из трех инструкций.

$$\begin{aligned}(x > 0) - (x < 0) \\ (x \ge 0) - (x \le 0)\end{aligned} \qquad (3)$$

И последнее замечание: почти всегда работает формула $\left(-x \overset{u}{\gg} 31\right) - \left(x \overset{u}{\gg} 31\right)$; ошибка возникает только при $x = -2^{31}$.

2.8. Трехзначная функция сравнения

Трехзначная функция сравнения представляет собой определенное обобщение функции *sign* и определяется следующим образом:

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

Данное определение справедливо для любых переменных — и знаковых и беззнаковых. Все, что говорится в этом разделе, если явно не оговорено иное, применимо к обоим случаям.

Использование команд сравнения позволяет реализовать эту функцию за три команды, как очевидное обобщение выражений (3).

$$\begin{aligned}(x > y) - (x < y) \\ (x \ge y) - (x \le y)\end{aligned}$$

Решение для беззнаковых целых чисел на компьютере PowerPC приведено ниже [10]. На этой машине “перенос” означает “не заем”.

```
subf  R5, Ry, Rx # R5 <-- Rx-Ry
subfc R6, Rx, Ry # R6 <-- Ry- Rx, установлен перенос
subfe R7, Ry, Rx # R7 <-- Rx-Ry+перенос, установлен перенос
subfe R8, R7, R5 # R8 <-- R5-R7+перенос, (установлен перенос)
```

Если ограничиться командами из базового набора RISC-команд, то эффективного метода вычисления этой функции нет. Для вычисления $x < y$, $x \le y$ и других команд сравнения потребуется выполнить пять команд (см. раздел 2.11), что приводит к решению, содержащему 12 команд (при использовании определенной общности вычислений $x < y$ и $y < x$). На RISC-машинах с базовым набором команд лучшим путем решения будет использование команд сравнения и ветвления (в худшем случае потребуется выполнить шесть команд).

2.9. Перенос знака

Функция *переноса знака* (известная в Fortran как `ISIGN`), определяется следующим образом:

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

На большинстве машин эта функция может быть вычислена (по модулю 32) за четыре команды.

$$\begin{array}{ll} t \leftarrow y \overset{s}{\gg} 31; & t \leftarrow (x \oplus y) \overset{s}{\gg} 31; \\ \mathrm{ISIGN}(x, y) = (\mathrm{abs}(x) \oplus t) - t = & \mathrm{ISIGN}(x, y) = (x \oplus t) - t = \\ \quad = (\mathrm{abs}(x) + t) \oplus t & \quad = (x + t) \oplus t \end{array}$$

2.10. Декодирование поля “0 означает 2**n”

Иногда 0 или отрицательное значение для некоторой величины не имеет смысла. Поэтому она кодируется нулевым значением в n-м поле, которое интерпретируется как 2^n, а ненулевое значение имеет обычный двоичный смысл. Например, у PowerPC длина поля в команде загрузки непосредственно заданной строки слов (`lswi`) занимает 5 битов. Нелогично использовать эту команду для загрузки нуля байтов, поскольку длина представляет собой непосредственно заданную величину, но было бы очень полезно иметь возможность загрузить 32 байта. Длина поля кодируется значениями от 0 до 31, которые могут, например, означать длины от 1 до 32, но соглашение “нуль означает 32” приводит к более простой логике, в особенности когда процессор должен также поддерживать соответствующую команду с переменной (задаваемой в регистре) длиной, которая выполняет простое бинарное кодирование (например, команду `lswx` на PowerPC).

Кодирование целых чисел от 1 до 2^n в поле, ноль в котором означает 2^n, тривиально: просто маскировать биты целого числа с помощью маски 2^n–1. Выполнить же декодирование без проверок и переходов не так-то просто. Ниже приведено несколько возможных вариантов для работы с третьим битом. Все эти варианты требуют выполнения трех команд, не считая возможных загрузок констант.

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

2.11. Предикаты сравнения

Предикаты сравнения представляют собой функции, которые сравнивают две величины и возвращают однобитовый результат, равный 1, если проверяемое отношение истинно, и 0, если ложно. В этом разделе приводится несколько методов вычисления результата сравнения с размещением его в бите знака (эти методы не используют команд ветвления). Чтобы получить значение 1 или 0, используемое в некоторых языках (например, в C), после вычисления следует использовать команду *сдвига вправо* на 31 бит. Для получения результата –1/0, используемого в некоторых других языках (например, Basic), выполняется команда *знакового сдвига вправо* на 31 бит.

На таких машинах, как MIPS, Compaq Alpha, и нашей модели RISC есть команды сравнения, которые непосредственно вычисляют большинство отношений и помещают однобитовый результат 0/1 в регистр общего назначения, а потому для таких машин приводимые ниже формулы не представляют особого интереса.

При вычислении функций сравнения двух операндов весьма полезной оказывается машинная команда “nabs”, вычисляющая отрицательное абсолютное значение. В отличие от функции абсолютного значения, она никогда не приводит к переполнению. Если среди ко-

манд машины нет функции “nabs”, но есть более привычная команда “abs”, то вместо $\text{nabs}(\boldsymbol{x})$ можно использовать $-\text{abs}(\boldsymbol{x})$. Если $\boldsymbol{x}$ — максимальное отрицательное число, переполнение возникает дважды, но итоговый результат получается правильный (предполагается, что абсолютное значение и отрицание максимального отрицательного числа дают одинаковый результат). Поскольку на многих машинах нет ни команды “abs”, ни команды “nabs”, приведем и альтернативные формулы, в которых эти функции не применяются.

Используемая ниже функция “nlz” возвращает количество ведущих нулевых битов аргумента. Описание функции doz (*разность или ноль*) приводится в разделе 2.18.

$$
\begin{array}{lll}
\boldsymbol{x} = \boldsymbol{y}: & \text{abs}(\boldsymbol{x}-\boldsymbol{y})-\mathbf{1} & \\
 & \text{abs}(\boldsymbol{x}-\boldsymbol{y}+\mathbf{0x80000000}) & \\
 & \text{nlz}(\boldsymbol{x}-\boldsymbol{y}) \ll 26 & \\
 & -\left(\text{nlz}(\boldsymbol{x}-\boldsymbol{y}) \overset{u}{\gg} 5\right) & \\
 & \neg(\boldsymbol{x}-\boldsymbol{y} \mid \boldsymbol{y}-\boldsymbol{x}) & \\
\boldsymbol{x} \neq \boldsymbol{y}: & \text{nabs}(\boldsymbol{x}-\boldsymbol{y}) & \\
 & \text{nlz}(\boldsymbol{x}-\boldsymbol{y})-\mathbf{32} & \\
 & \boldsymbol{x}-\boldsymbol{y} \mid \boldsymbol{y}-\boldsymbol{x} & \\
\boldsymbol{x} < \boldsymbol{y}: & (\boldsymbol{x}-\boldsymbol{y}) \oplus \left[(\boldsymbol{x} \oplus \boldsymbol{y}) \,\&\, ((\boldsymbol{x}-\boldsymbol{y}) \oplus \boldsymbol{x})\right] & \\
 & (\boldsymbol{x} \,\&\, \neg\boldsymbol{y}) \mid ((\boldsymbol{x} \equiv \boldsymbol{y}) \,\&\, (\boldsymbol{x}-\boldsymbol{y})) & \\
 & \text{nabs}(\text{doz}(\boldsymbol{y},\boldsymbol{x})) & [24] \\
\boldsymbol{x} \le \boldsymbol{y}: & (\boldsymbol{x} \mid \neg\boldsymbol{y}) \,\&\, ((\boldsymbol{x} \oplus \boldsymbol{y}) \mid \neg(\boldsymbol{y}-\boldsymbol{x})) & \\
 & \left((\boldsymbol{x} \equiv \boldsymbol{y}) \overset{s}{\gg} 1\right) + \left(\boldsymbol{x} \,\&\, \neg\boldsymbol{y}\right) & [24] \\
\boldsymbol{x} \overset{u}{<} \boldsymbol{y}: & (\neg\boldsymbol{x} \,\&\, \boldsymbol{y}) \mid ((\boldsymbol{x} \equiv \boldsymbol{y}) \,\&\, (\boldsymbol{x}-\boldsymbol{y})) & \\
 & (\neg\boldsymbol{x} \,\&\, \boldsymbol{y}) \mid ((\neg\boldsymbol{x} \mid \boldsymbol{y}) \,\&\, (\boldsymbol{x}-\boldsymbol{y})) & \\
\boldsymbol{x} \overset{u}{\le} \boldsymbol{y}: & (\neg\boldsymbol{x} \mid \boldsymbol{y}) \,\&\, ((\boldsymbol{x} \oplus \boldsymbol{y}) \mid \neg(\boldsymbol{y}-\boldsymbol{x})) &
\end{array}
$$

Для $\boldsymbol{x} > \boldsymbol{y}$, $\boldsymbol{x} \ge \boldsymbol{y}$ и прочих отношений необходимо в соответствующих формулах поменять местами $\boldsymbol{x}$ и $\boldsymbol{y}$. Команду *сложения* с $\mathbf{0x80000000}$ можно заменить любой другой командой, которая инвертирует значение старшего бита.

Другой класс формул основан на том, что предикат $\boldsymbol{x} < \boldsymbol{y}$ вычисляется по знаку величины $\boldsymbol{x}/2-\boldsymbol{y}/2$, а вычисление данной разности не может привести к переполнению. В тех случаях, когда сдвиги приводят к потере информации, результат можно уточнить путем вычитания единицы.

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

Большинство машин вычисляют эти отношения при помощи семи команд (или шести при наличии команды *и-не*), что ничем не лучше результата, достигнутого в предыдущих формулах (от пяти до семи команд в зависимости от полноты используемого множества команд).

Формулы, в которых используется функция "nlz", взяты из [56]. Ее использование позволяет вычислить результат отношения $\boldsymbol{x} = \boldsymbol{y}$ в виде 1 или 0 всего за три команды.

$$\operatorname{nlz}(\boldsymbol{x} - \boldsymbol{y}) \overset{u}{\gg} 5$$

Команды знакового сравнения с 0 встречаются достаточно часто, что заслуживает отдельного рассмотрения. Ниже приводится несколько формул, которые в основном непосредственно выведены из предыдущих выражений. Так же, как и ранее, результат сравнения отображается в позиции бита знака.

$$
\begin{array}{lll}
\boldsymbol{x} = 0: & \operatorname{abs}(\boldsymbol{x}) - 1 & \\
 & \operatorname{abs}(\boldsymbol{x} + \mathbf{0x80000000}) & \\
 & \operatorname{nlz}(\boldsymbol{x}) \ll 26 & \\
 & -\left(\operatorname{nlz}(\boldsymbol{x}) \overset{u}{\gg} 5\right) & \\
 & \neg(\boldsymbol{x} \mid -\boldsymbol{x}) & \\
 & \neg\boldsymbol{x} \,\&\, (\boldsymbol{x} - 1) & \\
\boldsymbol{x} \neq 0: & \operatorname{nabs}(\boldsymbol{x}) & \\
 & \operatorname{nlz}(\boldsymbol{x}) - \mathbf{32} & \\
 & \boldsymbol{x} \mid -\boldsymbol{x} & \\
 & \left(\boldsymbol{x} \overset{u}{\gg} 1\right) - \boldsymbol{x} & [10] \\
\boldsymbol{x} < 0: & \boldsymbol{x} & \\
\boldsymbol{x} \le 0: & \boldsymbol{x} \mid (\boldsymbol{x} - 1) & \\
 & \boldsymbol{x} \mid \neg -\boldsymbol{x} & \\
\boldsymbol{x} > 0: & \boldsymbol{x} \oplus \operatorname{nabs}(\boldsymbol{x}) & \\
 & \left(\boldsymbol{x} \overset{s}{\gg} 1\right) - \boldsymbol{x} & \\
 & -\boldsymbol{x} \,\&\, \neg\boldsymbol{x} & \\
\boldsymbol{x} \ge 0: & \neg\boldsymbol{x} &
\end{array}
$$

Знаковое сравнение можно получить из беззнакового, увеличивая сравниваемые величины на 2^{31}. Обратное преобразование также справедливо. Таким образом получаем

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

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

Аналогичные соотношения выполняются для $\le$, $\overset{u}{\le}$ и т.д. В данном случае прибавление и вычитание числа 2^{31} — эквивалентные действия, так как обе операции инвертируют значение знакового разряда.

Еще один путь получения знакового сравнения из беззнакового основан на том, что если $\boldsymbol{x}$ и $\boldsymbol{y}$ имеют одинаковые знаки, то $x < y = x \overset{u}{<} y$, а если их знаки различны, то $x < y = x \overset{u}{>} y$ [42]. В этом случае также справедливы обратные преобразования; таким образом, получаем

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

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

где x_{31} и y_{31} представляют собой знаковые биты $\boldsymbol{x}$ и $\boldsymbol{y}$ соответственно. Аналогичные выражения получаются для отношений $\le$, $\overset{u}{\le}$ и других.

Применение описанных способов позволяет вычислить все стандартные команды сравнения (кроме = и ≠) через другие команды сравнения с использованием не более трех дополнительных команд на большинстве машин. Например, возьмем в качестве исходной команды сравнения отношение $x \overset{u}{\le} y$, поскольку реализовать его проще других команд (как бит переноса $y - x$). Тогда остальные команды сравнения можно реализовать следующим образом:

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

Команды сравнения и бит переноса

Если процессор может легко поместить бит переноса в регистр общего назначения, то для некоторых операций отношения это позволяет получить очень лаконичный код. Ниже приводятся выражения такого рода для некоторых отношений. Запись carry(*выражение*) означает, что разряд переноса генерируется при выполнении команды *выражение*. Предполагается, что бит переноса для разности $x - y$ тот же, что и на выходе сумматора для выражения $x + \bar{y} + 1$, и является дополнением "заема".

$$\begin{aligned}
x = y: \quad & \text{carry}(0 - (x - y)), \text{ или} \\
& \text{carry}((x + y) + 1), \text{ или} \\
& \text{carry}((x - y - 1) + 1) \\
x \ne y: \quad & \text{carry}((x - y) - 1) = \text{carry}((x - y) + (-1)) \\
x < y: \quad & \neg\text{carry}\left(\left(x + 2^{31}\right) - \left(y + 2^{31}\right)\right)
\end{aligned}$$

$x \le y$: $\quad \text{carry}\left(\left(y + 2^{31}\right) - \left(x + 2^{31}\right)\right)$

$x \overset{u}{<} y$: $\quad \neg\text{carry}(x - y)$

$x \overset{u}{\le} y$: $\quad \text{carry}(y - x)$

$x = 0$: $\quad \text{carry}(0 - x)$, или $\text{carry}\left(\bar{x} + 1\right)$

$x \ne 0$: $\quad \text{carry}(x - 1) = \text{carry}(x + (-1))$

$x < 0$: $\quad \text{carry}(x + x)$

$x \le 0$: $\quad \text{carry}\left(2^{31} - \left(x + 2^{31}\right)\right)$

Для $x > y$ используется дополнение выражения для $x \le y$; то же справедливо и для других типов отношения “больше”.

При вычислении условных выражений на IBM RS/6000 и близком к нему PowerPC была использована программа GNU Superoptimizer [17]. У RS/6000 имеются команды $\text{abs}(x)$, $\text{nabs}(x)$, $\text{doz}(x, y)$ и ряд различных команд сложения и вычитания с использованием переноса. Как выяснилось, RS/6000 может вычислить все целочисленные условные выражения с помощью не более чем трех элементарных (выполняющихся за один такт) команд — результат, удививший даже самих создателей машины. В состав “всех условных выражений” входят шесть команд знакового сравнения и четыре команды беззнакового сравнения, а также их версии, в которых второй операнд равен нулю (при этом каждая из команд имеет два варианта — возвращающий результат 1/0 и возвращающий результат –1/0). Power PC, где нет функций $\text{abs}(x)$, $\text{nabs}(x)$ и $\text{doz}(x, y)$, вычисляет все перечисленные условные выражения не более чем за четыре элементарные команды.

Вычисление отношений

Большинство компьютеров возвращают однобитовый результат операции сравнения, который обычно размещается в “регистре условий”, а на некоторых машинах (как, например, в нашей RISC-модели) — в регистре общего назначения. В любом случае вычисление отношений зачастую реализуется вычитанием операндов и некоторыми действиями над битами полученного результата, чтобы получить окончательный однобитовый результат работы команды сравнения.

Рассмотрим логику описанных выше действий. Пусть вместо разности $x - y$ компьютер вычисляет сумму $x + \bar{y} + 1$, и в результате этих вычислений получаются следующие величины:

C_0 – перенос из старшего разряда в разряд переноса;
C_i – перенос в старший разряд;
N – бит знака результата;
Z – величина, равная 1, если полученный результат (за исключением C_0) нулевой, и 0 в противоположном случае.

Таким образом, в системе обозначений булевой алгебры для различных отношений получим приведенные ниже выражения (расположение рядом двух величин подразумевает операцию *и*, а оператор “+” означает операцию *или*).

$$
\begin{array}{rl}
V\text{:} & C_i \oplus C_0 \text{ (знаковое переполнение)} \\
\boldsymbol{x} = \boldsymbol{y}: & Z \\
\boldsymbol{x} \neq \boldsymbol{y}: & \overline{Z} \\
\boldsymbol{x} < \boldsymbol{y}: & N \oplus V \\
\boldsymbol{x} \leq \boldsymbol{y}: & (N \oplus V) + Z \\
\boldsymbol{x} > \boldsymbol{y}: & (N \equiv V)\overline{Z} \\
\boldsymbol{x} \geq \boldsymbol{y}: & N \equiv V \\
\boldsymbol{x} \overset{u}{<} \boldsymbol{y}: & \overline{C_0} \\
\boldsymbol{x} \overset{u}{\leq} \boldsymbol{y}: & \overline{C_0} + Z \\
\boldsymbol{x} \overset{u}{>} \boldsymbol{y}: & C_0\overline{Z} \\
\boldsymbol{x} \overset{u}{\geq} \boldsymbol{y}: & C_0
\end{array}
$$

2.12. Обнаружение переполнения

“Переполнение” означает, что результат арифметической операции либо слишком велик, либо слишком мал, чтобы его можно было корректно представить в выходном регистре. В этом разделе обсуждаются методы, которые позволяют определить, когда возникнет переполнение, не используя при этом биты состояния, предназначенные для этой цели. Эти методы имеют особое значение на тех машинах (например, MIPS), где биты состояния отсутствуют (даже при наличии таких битов обращение к ним из языков высокого уровня, как правило, затруднено или даже невозможно).

Знаковое сложение и вычитание

Если в результате сложения и вычитания целых чисел произошло переполнение, то, как правило, старший бит результата отбрасывается и сумматор выдает только младшие биты. Переполнение при сложении целых знаковых чисел возникает тогда и только тогда, когда операнды имеют одинаковые знаки, а знак суммы противоположен знаку операндов. Удивительно, но это правило справедливо даже тогда, когда в сумматоре был выполнен перенос, т.е. при вычислении суммы $\boldsymbol{x} + \boldsymbol{y} + 1$. Данное правило играет важную роль при сложении знаковых целых чисел, состоящих из нескольких слов, так как в этом случае последнее сложение представляет собой знаковое сложение двух полных слов и бита переноса, значение которого может быть равно 0 или +1.

Для доказательства этого правила предположим, что слагаемые $\boldsymbol{x}$ и $\boldsymbol{y}$ представляют собой целые знаковые значения, состоящие из одного слова, а бит переноса c равен 0 или 1. Предположим также для простоты, что сложение выполняется на 4-разрядной машине. Если $\boldsymbol{x}$ и $\boldsymbol{y}$ имеют разные знаки, то $-8 \leq \boldsymbol{x} \leq -1$ и $0 \leq \boldsymbol{y} \leq 7$ (аналогичные границы получаются и при отрицательном $\boldsymbol{y}$ и неотрицательном $\boldsymbol{x}$). В любом случае после сложения этих неравенств и необязательного добавления c получим $-8 \leq \boldsymbol{x} + \boldsymbol{y} + \boldsymbol{c} \leq 7$.

Как видите, сумма может быть представлена как 4-разрядное целое число со знаком; следовательно, при сложении операндов с разными знаками переполнения не будет.

Теперь предположим, что $\boldsymbol{x}$ и $\boldsymbol{y}$ имеют одинаковые знаки. Здесь возможны два случая.

(а)	(б)
$-8 \leq \boldsymbol{x} \leq -1$	$0 \leq \boldsymbol{x} \leq 7$

$$-8 \le y \le -1 \qquad\qquad 0 \le y \le 7$$

Таким образом:

$$\begin{array}{cc} \text{(а)} & \text{(б)} \\ -16 \le x + y + c \le -1 & 0 \le x + y + c \le 15 \end{array}$$

Переполнение возникает тогда, когда сумма не может быть представлена в виде 4-битового целого числа со знаком, т.е. если

$$\begin{array}{cc} \text{(а)} & \text{(б)} \\ -16 \le x + y + c \le -9 & 8 \le x + y + c \le 15 \end{array}$$

В случае (а) старший разряд 4-битовой суммы равен 0, т.е. противоположен знакам x и y. В случае (б) старший бит 4-битовой суммы равен 1, т.е. также противоположен знакам x и y.

При вычитании целых чисел, состоящих из нескольких слов, нас интересует разность $x - y - c$, где c равно **0** или **1**, причем **1** означает, что был выполнен заем. Проводя анализ, аналогичный только что рассмотренному, можно увидеть, что при вычислении выражения $x - y - c$ переполнение будет тогда и только тогда, когда x и y имеют разные знаки и знак разности противоположен знаку x (или, что то же самое, имеет знак, одинаковый со знаком y).

Это приводит нас к следующему выражению предиката наличия переполнения (помещающему результат в бит знака). Беззнаковый или знаковый сдвиг вправо на 31 бит дает нам результат в виде чисел 1/0 или –1/0.

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

Если взять вторую формулу для суммы и первую формулу для разности (не содержащие операции *эквивалентности*), то при использовании базового множества RISC-команд для проверки на переполнение потребуется выполнить три команды в дополнение к тем, которые вычисляют саму сумму (или разность). Четвертой может стать команда перехода к коду обработки переполнения.

Если при возникновении переполнения генерируется прерывание, то может потребоваться проверка, не вызовет ли сложение или вычитание переполнения (причем сама проверка ни в коем случае не должна вызвать переполнения). Для этого можно воспользоваться приведенными ниже формулами (в которых нет команд ветвления и которые заведомо не генерируют переполнения).

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

Присваивание в левом столбце устанавливает значение z равным **0x80000000**, если знаки x и y одинаковы, и 0 в противном случае. Таким образом, во втором выражении слагаемые имеют противоположные знаки и переполнение возникнуть не может. Если и x и y неотрицательны, значение знакового бита в результате во втором выражении будет равным 1 тогда и только тогда, когда $(x - \mathbf{2^{31}}) + y + c \ge \mathbf{0}$, т.е. если $x + y + c \ge 2^{31}$, что и представляет собой условие возникновения переполнения при вычислении $x + y + c$. Если и x и y отрицательны, то значение бита знака во втором выражении будет равным 1 тогда и только тогда, когда $(x + \mathbf{2^{31}}) + y + c < \mathbf{0}$, т.е. если $x + y + c < -2^{31}$, что также представляет собой условие

переполнения при вычислении $\boldsymbol{x} + \boldsymbol{y} + \boldsymbol{c}$. Член $\boldsymbol{x} \equiv \boldsymbol{y}$ гарантирует корректность результата (равный 0 знаковый бит), если $\boldsymbol{x}$ и $\boldsymbol{y}$ имеют разные знаки. Условия переполнения для разности (приведенные в правом столбце) получаются аналогично. Для реализации этих формул понадобится выполнить девять команд из базового множества RISC.

Может показаться, что использование значения разряда переноса дает эффективный алгоритм проверки на переполнение в случае знакового сложения, но на самом деле это не так. Однако есть еще один метод.

Если $\boldsymbol{x}$ — целое знаковое число, то $\boldsymbol{x} + \mathbf{2}^{31}$ представляет собой корректное беззнаковое число, которое получается в результате инвертирования старшего бита $\boldsymbol{x}$. При сложении положительных знаковых чисел переполнение возникает тогда, когда $\boldsymbol{x} + \boldsymbol{y} \geq 2^{31}$, т.е. если $\left(\boldsymbol{x} + 2^{31}\right) + \left(\boldsymbol{y} + 2^{31}\right) \geq 3 \cdot 2^{31}$. Последнее выражение означает, что при сложении беззнаковых операндов был перенос (так как сумма не меньше 2^{32}) и старший бит суммы равен 1. Аналогично, при сложении отрицательных чисел переполнение возникает тогда, когда значение разряда переноса и значение старшего разряда суммы равны 0.

Это дает нам следующий алгоритм обнаружения переполнения при знаковом сложении.

Вычисляем $\left(\boldsymbol{x} \oplus \mathbf{2}^{31}\right) + \left(\boldsymbol{y} \oplus \mathbf{2}^{31}\right)$ и получаем сумму $\boldsymbol{s}$ и значение бита переноса $\boldsymbol{c}$.

Переполнение возникает тогда и только тогда, когда значение разряда переноса $\boldsymbol{c}$ равно значению старшего разряда суммы $\boldsymbol{s}$.

Полученная сумма представляет собой корректное значение знакового сложения, поскольку инвертирование старшего бита обоих операндов не изменяет их суммы.

При вычитании используется такой же алгоритм, с тем отличием, что на первом шаге вместо суммы вычисляется разность. Мы считаем, что перенос в данном случае генерируется при вычислении $\boldsymbol{x} - \boldsymbol{y}$ как $\boldsymbol{x} + \overline{\boldsymbol{y}} + 1$. Полученная разность представляет собой корректный результат знакового вычитания.

Несмотря на то что рассмотренные методы крайне интересны, на многих компьютерах они работают не так эффективно, как методы без использования переноса (например, вычисление переполнения как $(\boldsymbol{x} \equiv \boldsymbol{y}) \,\&\, (\boldsymbol{s} \oplus \boldsymbol{x})$ для сложения и $(\boldsymbol{x} \oplus \boldsymbol{y}) \,\&\, (\boldsymbol{d} \oplus \boldsymbol{x})$ для вычитания, где $\boldsymbol{s}$ и $\boldsymbol{d}$ — соответственно сумма и разность $\boldsymbol{x}$ и $\boldsymbol{y}$).

Установка переполнения при знаковом сложении и вычитании

Зачастую переполнение при сложении знаковых операндов устанавливается посредством правила: “перенос в знаковый разряд отличается от переноса из знакового разряда”. Как ни удивительно, но эта логика позволяет корректно определять переполнение как при сложении, так и при вычитании, считая, что разность $\boldsymbol{x} - \boldsymbol{y}$ вычисляется как $\boldsymbol{x} + \overline{\boldsymbol{y}} + 1$. Более того, это утверждение корректно независимо от того, был ли перенос или заем либо нет. Конечно, то, что перенос в знаковый разряд легко вычислить, вовсе не означает, что существуют эффективные методы определения знакового переполнения. В случае сложения и вычитания перенос/заем формируется в знаковом разряде после вычисления приведенных ниже выражений (здесь $\boldsymbol{c}$ равно **0** или **1**).

Перенос:	Заем:
$(\boldsymbol{x} + \boldsymbol{y} + \boldsymbol{c}) \oplus \boldsymbol{x} \oplus \boldsymbol{y}$	$(\boldsymbol{x} - \boldsymbol{y} - \boldsymbol{c}) \oplus \boldsymbol{x} \oplus \boldsymbol{y}$

В действительности для каждого разряда i эти выражения дают перенос/заем в разряд i.

Беззнаковое сложение/вычитание

При беззнаковом сложении/вычитании для вычисления предикатов переполнения можно использовать следующий метод без переходов, результат работы которого помещается в знаковый разряд. Методы, использующие команды сдвига вправо, эффективно работают только тогда, когда известно, что $\boldsymbol{c}=0$. Выражения в скобках вычисляют перенос или заем, который появляется из младшего разряда.

Беззнаковое вычисление $\boldsymbol{x}+\boldsymbol{y}+\boldsymbol{c}$:

$$(\boldsymbol{x} \,\&\, \boldsymbol{y}) \mid \left((\boldsymbol{x} \mid \boldsymbol{y}) \,\&\, \neg(\boldsymbol{x}+\boldsymbol{y}+\boldsymbol{c})\right)$$

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

Беззнаковое вычисление $\boldsymbol{x}-\boldsymbol{y}-\boldsymbol{c}$:

$$(\neg\boldsymbol{x} \,\&\, \boldsymbol{y}) \mid \left((\boldsymbol{x} \equiv \boldsymbol{y}) \,\&\, (\boldsymbol{x}-\boldsymbol{y}-\boldsymbol{c})\right)$$

$$(\neg\boldsymbol{x} \,\&\, \boldsymbol{y}) \mid \left((\neg\boldsymbol{x} \mid \boldsymbol{y}) \,\&\, (\boldsymbol{x}-\boldsymbol{y}-\boldsymbol{c})\right)$$

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

В [46] для беззнаковых *сложения* и *вычитания* есть существенно более простые формулы, использующие команды сравнения. При беззнаковом сложении переполнение (перенос) возникает, если полученная сумма меньше (при беззнаковом сравнении) любого из операндов. Эта и подобные формулы приведены ниже. К сожалению, нет возможности обобщить эти формулы для использования в них переменной $\boldsymbol{c}$, которая представляет собой перенос или заем. Вместо этого приходится сначала проверять значение $\boldsymbol{c}$, а затем, в зависимости от того, равно оно **0** или **1**, использовать одно из следующих типов сравнений (напомним — вычисления сумм и разностей беззнаковые).

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

В каждом из приведенных случаев перед выполнением сложения/вычитания вычисляется первая формула, которая проверяет, возможно ли переполнение, не вызывая при этом самого переполнения. Вторая формула вычисляется после выполнения сложения/вычитания, при которых возможно переполнение.

Для случая знакового сложения/вычитания аналогичного (с использованием сравнений) простого метода проверки, похоже, не существует.

Умножение

При умножении переполнение означает, что результат не помещается в 32 разрядах (результат как знакового, так и беззнакового умножения 32-битовых чисел всегда может быть представлен в виде 64-битового числа). Если доступны старшие 32 разряда произведения, то определить, имеет ли место переполнение, очень просто. Пусть $\mathrm{hi}(\boldsymbol{x}\times\boldsymbol{y})$ и $\mathrm{lo}(\boldsymbol{x}\times\boldsymbol{y})$ — старшая и младшая половины 64-битового произведения. Тогда условия переполнения можно записать следующим образом [46].

Беззнаковое произведение $x \times y$:	Знаковое произведение $x \times y$:
$\mathrm{hi}(x \times y) \neq 0$	$\mathrm{hi}(x \times y) \neq \left(\mathrm{lo}(x \times y) \overset{s}{\gg} 31\right)$

Один из способов проверки наличия переполнения состоит в контрольном делении. Однако при этом необходимо следить, чтобы не произошло деления на 0, а кроме того, при знаковом умножении возникают дополнительные сложности.

Переполнение при умножении имеет место, если приведенные ниже выражения истинны.

Беззнаковое умножение:	Знаковое умножение:
$z \leftarrow x * y$	$z \leftarrow x * y$
$y \neq 0 \mathbin{\vec{\&}} z \overset{u}{\div} y \neq x$	$(y < 0 \mathbin{\&} x = -2^{31}) \mid (y \neq 0 \mathbin{\vec{\&}} z \div y \neq x)$

Затруднения возникают при $x = -2^{31}$ и при $y = -1$. В этом случае при умножении будет переполнение, но результат может получиться равным -2^{31}. Такой результат приводит к переполнению при делении, и таким образом оказывается возможным любой результат (на некоторых машинах). Поэтому данный частный случай должен проверяться отдельно, что и делается добавлением члена $(y < 0 \mathbin{\&} x = -2^{31})$. Чтобы в приведенных выше выражениях предотвратить деление на 0, используется оператор "условного *и*" (в языке C — оператор `&&`).

Можно также использовать проверку делением без выполнения самого умножения (т.е. без генерации переполнения). При беззнаковом умножении переполнение возникает тогда и только тогда, когда $xy > 2^{32} - 1$, или $x > ((2^{32} - 1)/y)$, или, поскольку x — целое число, $x > \lfloor (2^{32} - 1)/y \rfloor$. В обозначениях компьютерной арифметики это выглядит следующим образом: $y \neq 0 \mathbin{\vec{\&}} x \overset{u}{>} \left(\mathbf{0xFFFFFFFF} \overset{u}{\div} y\right)$.

Для целых знаковых чисел переполнение в результате умножения x и y определить сложнее. Если операнды имеют одинаковые знаки, то переполнение будет тогда и только тогда, когда $xy > 2^{31} - 1$. Если x и y имеют противоположные знаки, то переполнение будет тогда и только тогда, когда $xy < -2^{31}$. Эти условия могут быть проверены с помощью знакового деления (табл. 2.2).

Таблица 2.2. Проверка на переполнение произведения знаковых операндов

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

Как видно из таблицы, при проверке необходимо учесть четыре возможных случая. Вследствие проблем переполнения и представления числа $+2^{31}$ с объединением этих выражений могут возникнуть трудности.

В случае доступности беззнакового деления тест можно упростить. Если использовать абсолютные значения x и y, которые корректно представимы в виде целых чисел без знака, то весь тест можно записать так, как показано ниже. Переменная c имеет значение $2^{31} - 1$, если x и y имеют одинаковые знаки, и 2^{31} в противном случае.

$$c \leftarrow \left((x \equiv y) \overset{s}{\gg} 31\right) + 2^{31}$$

$$x \leftarrow \mathrm{abs}(x)$$

$$\boldsymbol{y} \leftarrow \mathrm{abs}(\boldsymbol{y})$$

$$\boldsymbol{y} \neq \mathbf{0} \,\vec{\&}\, \boldsymbol{x} \overset{u}{>} \left(\boldsymbol{x} \overset{u}{\div} \boldsymbol{y}\right)$$

Чтобы выяснить, будет ли переполнение при вычислении произведения $\boldsymbol{x} * \boldsymbol{y}$, можно использовать команду, вычисляющую *количество ведущих нулей* у операндов. Способ, основанный на использовании этой функции, позволяет более точно определить условие переполнения. Сначала рассмотрим умножение беззнаковых чисел. Легко показать, что если $\boldsymbol{x}$ и $\boldsymbol{y}$ — 32-разрядные величины, содержащие m и n ведущих нулей соответственно, то в 64-разрядном произведении будет $m+n$ или $m+n+1$ ведущих нулей (или 64, если $\boldsymbol{x} = \mathbf{0}$ или $\boldsymbol{y} = 0$). Переполнение возникает тогда, когда в 64-разрядном произведении количество ведущих нулей оказывается меньше 32. Из этого следует:

если $\mathrm{nlz}(\boldsymbol{x}) + \mathrm{nlz}(\boldsymbol{y}) \geq 32$, переполнения точно не будет;
если $\mathrm{nlz}(\boldsymbol{x}) + \mathrm{nlz}(\boldsymbol{y}) \leq 30$, переполнение точно будет.

Если же $\mathrm{nlz}(\boldsymbol{x}) + \mathrm{nlz}(\boldsymbol{y}) = 31$, то переполнение может быть, а может и не быть. Чтобы узнать, будет ли переполнение в этом частном случае, вычисляется значение $t = x\lfloor y/2 \rfloor$ (переполнения при этом вычислении не возникает). Поскольку xy равно $2t$ (или $2t + x$, если y нечетно), переполнение при вычислении xy будет в том случае, когда $t \geq 2^{31}$. Все это приводит к коду, показанному в листинге 2.2, который вычисляет значение произведения xy с проверкой переполнения (в этом случае выполняется переход к метке `overflow`).

Листинг 2.2. Беззнаковое умножение с проверкой переполнения

```
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;
}
// В z содержится произведение x и y.
```

При знаковом умножении условие переполнения можно определить через количество ведущих нулей положительных аргументов и количество ведущих единиц отрицательных аргументов. Пусть $\boldsymbol{m} = \mathrm{nlz}(\boldsymbol{x}) + \mathrm{nlz}(\overline{\boldsymbol{x}})$ и $\boldsymbol{n} = \mathrm{nlz}(\boldsymbol{y}) + \mathrm{nlz}(\overline{\boldsymbol{y}})$, тогда:

если $\boldsymbol{m} + \boldsymbol{n} \geq \mathbf{34}$, переполнения точно не будет;
если $\boldsymbol{m} + \boldsymbol{n} \leq \mathbf{31}$, переполнение точно будет.

Остались два неоднозначных случая — 32 и 33. В случае $\boldsymbol{m} + \boldsymbol{n} = 33$ переполнение возникает только тогда, когда оба аргумента отрицательны и произведение равно в точности 2^{31} (машинный результат равен -2^{31}), так что распознать эту ситуацию можно при помощи проверки корректности знака произведения (т.е. переполнение будет, если $\boldsymbol{m} \oplus \boldsymbol{n} \oplus (\boldsymbol{m} * \boldsymbol{n}) < 0$). Для случая $\boldsymbol{m} + \boldsymbol{n} = 32$ определить наличие переполнения не так просто.

Не будем больше задерживаться на этой теме; заметим только, что определение наличия переполнения может основываться на значении $\operatorname{nlz}(\operatorname{abs}(\boldsymbol{x}))+\operatorname{nlz}(\operatorname{abs}(\boldsymbol{y}))$, но и при этом возможны неоднозначные случаи, когда значение данного выражения равно 31 или 32.

Деление

При знаковом делении $\boldsymbol{x} \div \boldsymbol{y}$ переполнение возникает, если истинно выражение

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

Большинство компьютеров сообщают о переполнении (или генерируют прерывание) при возникновении неопределенности типа $\mathbf{0} \div \mathbf{0}$.

Простой код для вычисления этого выражения, включая завершающую команду перехода к коду обработки переполнения, состоит из семи команд (три из которых являются командами перехода). Создается впечатление, что этот код улучшить нельзя, однако рассмотрим некоторые возможности для этого.

$$\left[\operatorname{abs}(\boldsymbol{y} \oplus \mathbf{0x80000000}) \mid (\operatorname{abs}(\boldsymbol{x}) \;\&\; \operatorname{abs}(\boldsymbol{y} \equiv \mathbf{0x80000000}))\right] < \mathbf{0}$$

Здесь сначала вычисляется выражение в скобках, а затем выполняется команда перехода, если полученный результат меньше 0. Это выражение на машине, имеющей все необходимые команды (включая сравнение с 0), вычисляется примерно за девять команд с учетом загрузки константы и завершающего перехода.

Еще один метод предполагает первоначальное вычисление значения

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

(что на большинстве машин выполняется за три команды), а затем выполнение проверки и ветвления по условию $\boldsymbol{y}=\mathbf{0} \mid z=\mathbf{0}$ одним из перечисленных способов.

$$((\boldsymbol{y} \mid -\boldsymbol{y}) \;\&\; (z \mid -z)) \geq \mathbf{0}$$

$$(\operatorname{nabs}(\boldsymbol{y}) \;\&\; \operatorname{nabs}(z)) \geq \mathbf{0}$$

$$\left((\operatorname{nlz}(\boldsymbol{y}) \mid \operatorname{nlz}(z)) \overset{u}{\gg} \mathbf{5}\right) \neq \mathbf{0}$$

Вычисление этих выражений может быть реализовано за девять, семь и восемь команд соответственно (при наличии всех необходимых для вычисления команд). Для PowerPC наиболее эффективным оказывается последний из перечисленных методов.

При беззнаковом делении переполнение возникает тогда и только тогда, когда $\boldsymbol{y}=\mathbf{0}$.

2.13. Флаги условий после сложения, вычитания и умножения

Во многих процессорах в результате выполнения арифметических действий устанавливаются так называемые флаги условий, в совокупности представляющие слово состояния процессора. Зачастую имеется только одна команда *сложения*, после которой выполняется оценка результата для беззнаковых или знаковых операндов (но не для смешанных типов). Как правило, на флаги условий оказывают влияние следующие события:

- был ли перенос или нет (беззнаковое переполнение);
- было или нет знаковое переполнение;
- является ли полученный 32-битовый результат, интерпретируемый как знаковое целое число в дополнительном коде без учета переноса и переполнения, отрицательным, положительным или нулевым.

На некоторых старых моделях машин в случае переполнения обычно выдается сообщение о том, какой именно результат (в случае *сложения* и *вычитания* — 33-битовый) получен: положительный, отрицательный или нулевой. Однако использовать такое сообщение в компиляторах языков высокого уровня — непростая задача, так что данная возможность нами не рассматривается.

При сложении возможны только 9 из 12 комбинаций этих трех событий. Невозможны следующие три комбинации: “нет переноса, переполнение, результат > 0”, “нет переноса, переполнение, результат = 0” и “перенос, переполнение, результат < 0”. Таким образом, для флагов условий требуется четыре бита. Две из комбинаций флагов уникальны в том смысле, что могут произойти только при единственном значении исходных аргументов: комбинация “нет переноса, нет переполнения, результат = 0” возникает только при сложении 0 с 0; комбинация “перенос, переполнение, результат = 0” возникает только при сложении двух максимальных отрицательных чисел. Эти замечания справедливы и при наличии “входного переноса”, т.е. при вычислении $\boldsymbol{x}+\boldsymbol{y}+1$.

В случае вычитания полагаем, что для вычисления разности $\boldsymbol{x}-\boldsymbol{y}$ компьютер вычисляет сумму $\boldsymbol{x}+\overline{\boldsymbol{y}}+1$ с переносом, который образуется так же, как и при *сложении* (понятие “перенос” при вычитании трактуется наоборот: перенос, равный 1, означает, что результат помещается в одно слово, а перенос, равный 0, означает, что результат в одно слово не помещается). Следовательно, при вычитании могут произойти только семь из всех возможных комбинаций событий. Невозможны те же три комбинации, что и при сложении, а кроме того, еще две комбинации: “нет переноса, нет переполнения, результат = 0” и “перенос, переполнение, результат = 0”.

Если умножение на компьютере может давать произведение двойной длины, то желательно иметь две команды умножения: одну для знаковых операндов, а другую — для беззнаковых. (На 4-разрядной машине в результате умножения знаковых операндов произведение **F×F = 01**, а в случае перемножения беззнаковых операндов — **F×F = E1** (в шестнадцатеричной записи)). Ни переноса, ни переполнения при выполнении этих команд умножения не возникает, так как результат умножения всегда может разместиться в двойном слове.

Если же произведение дает в результате одно (младшее) слово, то в случае, когда операнды и результат интерпретируются как целые беззнаковые числа, перенос означает, что результат не помещается в одно слово. Если же операнды и результат интерпретируются как знаковые целые числа в дополнительном коде, то, если результат не помещается в слово, устанавливается флаг переполнения. Таким образом, при умножении возможны только девять комбинаций событий. Невозможны комбинации “нет переноса, переполнение, результат > 0”, “нет переноса, переполнение, результат = 0” и “перенос, нет переполнения, результат = 0”. Таким образом, при рассмотрении сложения, вычитания и умножения получаем, что реально могут встретиться только 10 комбинаций флагов условий из 12 возможных.

2.14. Циклический сдвиг

Здесь все тривиально. Приведенные ниже выражения справедливы для любых *n* от 0 до 32 включительно, даже если сдвиги выполняются по модулю 32.

Циклический сдвиг влево на ***n*** разрядов: $\boldsymbol{y} \leftarrow \left(\boldsymbol{x} \ll \boldsymbol{n}\right) \mid \left(\boldsymbol{x} \overset{u}{\gg} (\mathbf{32}-\boldsymbol{n})\right)$

Циклический сдвиг вправо на ***n*** разрядов: $\boldsymbol{y} \leftarrow \left(\boldsymbol{x} \overset{u}{\gg} \boldsymbol{n}\right) \mid \left(\boldsymbol{x} \ll (\mathbf{32}-\boldsymbol{n})\right)$

2.15. Сложение/вычитание двойных слов

Взяв за основу любое из приведенных в подразделе “Беззнаковое сложение/вычитание” раздела 2.12 выражение для переполнения при беззнаковом сложении и вычитании, можно легко реализовать сложение и вычитание двойных слов, не используя при этом значение бита переноса. Пусть (z_1, z_0) — результат сложения операндов $(\boldsymbol{x}_1, \boldsymbol{x}_0)$ и $(\boldsymbol{y}_1, \boldsymbol{y}_0)$. Индекс 1 обозначает старшее полуслово, а индекс 0 — младшее. Считаем также, что при сложении используются все 32 бита регистров. Младшие слова интерпретируются как беззнаковые величины. Тогда

$$\begin{aligned} z_0 &\leftarrow \boldsymbol{x}_0 + \boldsymbol{y}_0 \\ \boldsymbol{c} &\leftarrow \left[(\boldsymbol{x}_0 \mathbin{\&} \boldsymbol{y}_0) \mid \left((\boldsymbol{x}_0 \mid \boldsymbol{y}_0) \mathbin{\&} \neg z_0 \right) \right] \overset{u}{\gg} 31 \\ z_1 &\leftarrow \boldsymbol{x}_1 + \boldsymbol{y}_1 + \boldsymbol{c}. \end{aligned}$$

Всего для вычисления потребуется девять команд. Во второй строке можно использовать выражение $\boldsymbol{c} \leftarrow \left(z_0 \overset{u}{<} \boldsymbol{x}_0 \right)$, допускающее получение конечного результата за четыре команды на машинах, в наборе команд которых имеется оператор сравнения в форме, помещающей результат (0 или 1) в регистр, как, например, команда `SLTU` (*Set on Less Then Unsigned — беззнаковая проверка “меньше, чем”*) в MIPS [46].

Аналогичный код используется для вычисления разности двойных слов.

$$\begin{aligned} z_0 &\leftarrow \boldsymbol{x}_0 - \boldsymbol{y}_0 \\ \boldsymbol{b} &\leftarrow \left[(\neg \boldsymbol{x}_0 \mathbin{\&} \boldsymbol{y}_0) \mid \left((\boldsymbol{x}_0 \equiv \boldsymbol{y}_0) \mathbin{\&} z_0 \right) \right] \overset{u}{\gg} 31 \\ z_1 &\leftarrow \boldsymbol{x}_1 - \boldsymbol{y}_1 - \boldsymbol{b} \end{aligned}$$

На машинах с полным набором логических команд для вычисления потребуется выполнить восемь команд. Если вторую строку записать как $\boldsymbol{b} \leftarrow \left(\boldsymbol{x}_0 \overset{u}{<} \boldsymbol{y}_0 \right)$, то на машинах, имеющих команду `SLTU`, для вычисления разности понадобится всего четыре команды.

На большинстве машин сложение и вычитание двойных слов реализуется за пять команд, посредством представления данных в младшем слове с помощью 31 бита (старший бит всегда равен 0, за исключением временного хранения в нем бита переноса или заема).

2.16. Сдвиг двойного слова

Пусть $(\boldsymbol{x}_1, \boldsymbol{x}_0)$ — два 32-разрядных слова, которые требуется сдвинуть вправо или влево как единое 64-разрядное слово, в котором $\boldsymbol{x}_1$ представляет старшее полуслово, и пусть $(\boldsymbol{y}_1, \boldsymbol{y}_0)$ представляет собой аналогично интерпретируемый результат сдвига. Будем считать, что величина сдвига n может принимать любые значения от 0 до 63. Предположим также, что машина может выполнять команду сдвига по модулю 64 или большему. Это означает, что если величина сдвига лежит в диапазоне от 32 до 63 или от –32 до –1, то в результате сдвига получится слово, состоящее из одних нулевых битов. Исключением является команда знакового сдвига вправо, так как в этом случае 32 разряда результата будут заполнены значением знакового разряда слова. (Рассматриваемый алгоритм не работает на процессорах Intel x86, сдвиг в которых выполняется по модулю 32.)

При этих предположениях команду *сдвига двойного слова влево* можно реализовать, как показано ниже (для вычисления требуется восемь команд).

$$y_1 \leftarrow x_1 \ll n \mid x_0 \overset{u}{\gg} (32 - n) \mid x_0 \ll (n - 32)$$
$$y_0 \leftarrow x_0 \ll n$$

Чтобы при $n = 32$ получился корректный результат, в первом присваивании должна обязательно использоваться команда *или*, а не *плюс*. Если известно, что $0 \le n \le 32$, то последний член в первом выражении можно опустить, тем самым реализуя сдвиг двойного слова за шесть команд.

Аналогично, *беззнаковый сдвиг двойного слова вправо* можно вычислить так:

$$y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32 - n) \mid x_1 \overset{u}{\gg} (n - 32)$$
$$y_1 \leftarrow x_1 \overset{u}{\gg} n$$

Знаковый сдвиг двойного слова вправо реализовать сложнее из-за нежелательного распространения знакового разряда в одном из членов. Вот как выглядит простейший код для вычисления этой операции:

$$\texttt{if}\ n < 32$$
$$\quad \texttt{then}\ y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll (32 - n)$$
$$\quad \texttt{else}\ y_0 \leftarrow x_1 \overset{s}{\gg} (n - 32)$$
$$y_1 \leftarrow x_1 \overset{s}{\gg} n$$

Если в компьютере есть команда *условной пересылки* (*conditional move*), то код реализуется за восемь команд без переходов. Если же команды условной пересылки нет, то понадобится выполнить десять команд, с созданием маски для устранения нежелательного распространения знака в одном из членов.

$$y_0 \leftarrow x_0 \overset{u}{\gg} n \mid x_1 \ll \left(32 - n\right) \mid \left[\left(x_1 \overset{s}{\gg} (n - 32)\right) \& \left((32 - n) \overset{s}{\gg} 31\right)\right]$$
$$y_1 \leftarrow x_1 \overset{s}{\gg} n$$

2.17. Сложение, вычитание и абсолютное значение многобайтовых величин

Ряд приложений работает с массивами коротких целых чисел (обычно байтов или полуслов), и часто работу программы можно ускорить, если начать выполнять действия не над короткими числами, а над словами. Пусть для определенности слово содержит четыре однобайтовых целых числа (хотя все методы, которые рассматриваются в этом разделе, легко приспособить к другим типам упаковки, например к словам, содержащим одно 12-битовое и два 10-битовых целых числа и т.д.). Описываемые методы играют бóльшую роль при работе на 64-битовых машинах, поскольку в этом случае повышается степень возможной параллельности вычислений.

Сложение должно быть организовано таким образом, чтобы не возникало переносов из одного байта в другой; для этого можно воспользоваться приведенным ниже двухшаговым алгоритмом.

1. Замаскировать старший бит в каждом байте операнда и выполнить сложение (избежав таким образом переносов через границы байтов).
2. Учесть старшие биты каждого байта, т.е. выполнить сложение старших битов операндов (и при необходимости перенос в этот бит).

Перенос в старший бит каждого байта обусловлен суммированием на первом шаге алгоритма. Аналогичный метод используется и при вычитании.

Сложение:

$$s \leftarrow (x \,\&\, \mathbf{0x7F7F7F7F}) + (y \,\&\, \mathbf{0x7F7F7F7F})$$
$$s \leftarrow ((x \oplus y) \,\&\, \mathbf{0x80808080}) \oplus s$$

Вычитание:

$$d \leftarrow (x \mid \mathbf{0x80808080}) - (y \,\&\, \mathbf{0x7F7F7F7F})$$
$$d \leftarrow ((x \oplus y) \mid \mathbf{0x7F7F7F7F}) \equiv d$$

На компьютере, оснащенном полным набором логических команд, реализация описанных выше вычислений потребует выполнения восьми команд (с учетом загрузки числа **0x7F7F7F7F**). (В приведенных формулах команды *и* и *или* с числом **0x80808080** можно заменить соответственно командами *и-не* и *или-не* с числом **0x7F7F7F7F**.)

Существует и другая методика для случая, когда слово состоит из двух полей. Сложение при этом можно реализовать посредством прибавления 32-битовых чисел с последующим вычитанием нежелательного переноса из полученной суммы. Ранее уже отмечалось, что выражение $(x + y) \oplus x \oplus y$ дает переносы в каждом разряде. Использование этой и аналогичной формулы для вычитания дает нам приведенный ниже код для сложения/вычитания двух полуслов по модулю 2^{16} (реализуется семью командами).

Сложение:	Вычитание:
$s \leftarrow x + y$	$d \leftarrow x - y$
$c \leftarrow (s \oplus x \oplus y) \,\&\, \mathbf{0x00010000}$	$b \leftarrow (d \oplus x \oplus y) \,\&\, \mathbf{0x00010000}$
$s \leftarrow s - c$	$d \leftarrow d + b$

Абсолютное значение многобайтовой величины можно вычислить путем дополнения этого числа и прибавления 1 к каждому байту, содержащему отрицательное целое число (т.е. прибавление значения его старшего бита). Приведем код, который делает каждый байт y равным абсолютному значению соответствующего байта x (реализуется восемью командами).

$a \leftarrow x \,\&\, \mathbf{0x80808080}$	Выделяем биты знаков
$b \leftarrow a \overset{u}{\gg} 7$	Целое число, равное 1, если x отрицательно
$m \leftarrow (a - b) \mid a$	**0xFF** для отрицательного x
$y \leftarrow (x \oplus m) + b$	Дополнение и прибавление 1 для отрицательных чисел

Вместо третьей формулы можно использовать $m \leftarrow a + a - b$. Прибавление b в четвертой строке не может вызвать переноса через границу байта, так как в каждом байте значение старшего бита $x \oplus m$ равно 0.

2.18. Функции Doz, Max, Min

Функция doz для знаковых аргументов определяется следующим образом:

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

Функцию doz называют также “вычитанием в первом классе”, так как, если вычитаемое больше уменьшаемого, результат равен 0. Эту функцию удобно использовать при вычислении двух других функций — $\max(x, y)$ и $\min(x, y)$. В связи с этим следует отметить, что функция $\mathrm{doz}(x, y)$ может быть и отрицательной, если при вычислении разности произошло переполнение. В Fortran эта функция используется в реализации функции `IDIM`, хотя в этом языке программирования при возникновении переполнения результат считается неопределенным.

Похоже, что очень хорошей реализации функций doz, max и min без использования команд перехода, применимой на большинстве компьютеров, просто не существует. Пожалуй, лучшее, что можно сделать, — это вычислить $\mathrm{doz}(x, y)$ с использованием одного из выражений для рассматривавшегося в этой главе предиката $x < y$, а затем вычислить значения функций max и min.

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

Вычислить функцию $\mathrm{doz}(x, y)$ можно за семь команд, если в наборе имеется команда *эквивалентности*, или за восемь команд в противном случае. Для вычисления функций max и min необходимо выполнить еще одну команду.

В случае беззнаковых аргументов используем беззнаковые версии указанных функций.

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

В компьютере IBM RISC/6000 (и его предшественнике 801) есть отдельная команда для вычисления функции $\mathrm{doz}(x, y)$, так что на этих машинах функции $\max(x, y)$ и $\min(x, y)$ с целыми знаковыми аргументами вычисляются за две команды, что весьма эффективно (непосредственная реализация этих функций требует бóльших затрат).

На машинах, имеющих команду *условной пересылки*, возможна двухкомандная реализация деструктивных[2] функций max и min. Например, при использовании нашего RISC-компьютера с расширенным набором команд выражение $x \leftarrow \max(x, y)$ можно вычислить следующим образом:

```
cmplt z, x, y    ; Установить z = 1, если x < y, иначе z = 0
movne x, z, y    ; Если z не равен 0, то присвоить x = y
```

[2] Деструктивные операции представляют собой операции, которые перезаписывают один или несколько своих аргументов.

2.19. Обмен содержимого регистров

Существует старый, хорошо известный способ обмена содержимым двух регистров без использования третьего [34].

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

Этот метод хорошо работает на двухадресной машине. Он остается работоспособным и при замене оператора $\oplus$ дополнительным к нему оператором эквивалентности $\equiv$. Возможно также использование различных наборов операций сложения и вычитания.

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

К сожалению, в каждом из вариантов обмена есть команда, неприемлемая для двухадресной машины (если, конечно, среди команд машины нет "обратного вычитания").

Вот еще одна маленькая хитрость, которая может оказаться полезной в приложениях с двойной буферизацией, где меняются местами два указателя. Первая команда выносится за пределы цикла, в котором выполняется обмен (хотя при этом и теряется преимущество сохранения регистра).

$$\begin{array}{rl} \text{Вне цикла:} & t \leftarrow x \oplus y \\ \text{Внутри цикла:} & x \leftarrow x \oplus t \\ & y \leftarrow y \oplus t \end{array}$$

Обмен соответствующих полей регистров

Зачастую требуется обменять биты двух регистров x и y, если i-й бит маски $m_i=1$, и оставить их неизменными, если $m_i=0$. Под обменом "соответствующими" полями подразумевается обмен без сдвигов. В маске m единичные биты не обязательно должны быть смежными. Приведем простейший метод решения поставленной задачи.

$$\begin{aligned} x' &\leftarrow (x \,\&\, \bar{m}) \mid (y \,\&\, m) \\ y &\leftarrow (y \,\&\, \bar{m}) \mid (x \,\&\, m) \\ x &\leftarrow x' \end{aligned}$$

Использование промежуточных временных данных в четырех выражениях с командой *и* требует выполнения семи команд при условии, что загрузка $\bar{m}$ или m занимает одну команду и имеется команда *и-не*. Если машина в состоянии вычислить четыре независимых команды *и* параллельно, то время выполнения равно всего трем тактам.

Более эффективный метод (пять команд, требующих для выполнения четыре такта на машине с неограниченными возможностями распараллеливания вычислений на уровне команд) показан ниже, в столбце (а). Этот алгоритм использует три команды *исключающего или*.

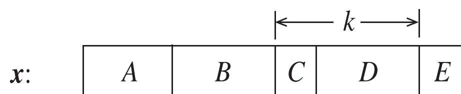
$$\begin{array}{lll} \text{(а)} & \text{(б)} & \text{(в)} \\ x \leftarrow x \oplus y & x \leftarrow x \equiv y & t \leftarrow (x \oplus y) \,\&\, m \\ y \leftarrow y \oplus (x \,\&\, m) & y \leftarrow y \equiv (x \mid \bar{m}) & x \leftarrow x \oplus t \\ x \leftarrow x \oplus y & x \leftarrow x \equiv y & y \leftarrow y \oplus t \end{array}$$

Метод, предложенный в столбце (б), лучше использовать в тех случаях, когда m не помещается в поле непосредственно задаваемого значения, но в него можно поместить $\bar{m}$, а в наборе команд имеется команда *эквивалентности*.

Метод из столбца (в) предложен в [19]. Он тоже требует выполнения пяти команд (в предположении, что для загрузки $\boldsymbol{m}$ в регистр используется одна команда), но на машинах с достаточными возможностями распараллеливания вычислений на уровне команд выполнение этого кода занимает всего три такта.

Обмен двух полей одного регистра

Предположим, что в регистре $\boldsymbol{x}$ необходимо поменять два поля (одинаковой длины), не изменяя при этом прочие биты регистра — т.е. нам требуется поменять местами поля B и D в пределах одного машинного слова, не изменяя при этом содержимое полей A, C и E. Поля расположены друг от друга на расстоянии k бит.



Следующий код сдвигает поля D и B на новые позиции и затем комбинирует полученные слова с помощью команд *и* и *или*.

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

Здесь маска $\boldsymbol{m}$ содержит единичные биты в поле D (и нулевые биты в других полях), а маска $\boldsymbol{m}'$ содержит единичные биты в полях A, C и E. На машине с неограниченными возможностями распараллеливания вычислений на уровне команд требуется выполнение девяти команд за четыре такта, так как для загрузки двух масок необходимо выполнить две команды.

Ниже описан метод обмена полями [19], который в тех же предположениях требует выполнения семи команд и выполняется за пять тактов. Он аналогичен алгоритму обмена соответствующими полями двух регистров, описанному ранее в подразделе "Обмен соответствующих полей регистров" в столбце (в). Здесь, как и ранее, $\boldsymbol{m}$ — маска, выделяющая поле D.

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

Идея состоит в том, что $\boldsymbol{t}_1$ в позициях поля D содержит биты $\boldsymbol{B} \oplus \boldsymbol{D}$ (и нули в остальных полях), а $\boldsymbol{t}_2$ содержит биты $\boldsymbol{B} \oplus \boldsymbol{D}$ в поле B. Этот код, как и приведенный ранее, работает корректно, даже если B и D представляют собой поля, разбитые на части, т.е. даже если в маске $\boldsymbol{m}$ единичные биты не являются смежными.

Условный обмен

Методы обмена из предыдущих двух разделов, использующие команду *исключающего или*, вырождаются в отсутствие каких-либо действий, если все биты маски $\boldsymbol{m}$ нулевые. Следовательно, эти методы могут выполнять условный обмен содержимым регистров, соответствующих полей регистров или полей в одном регистре, если некоторое условие $\boldsymbol{c}$ истинно, и не выполнять его при ложности условия. Для этого достаточно, чтобы все биты маски $\boldsymbol{m}$ были нулевыми, если условие ложно, и име-

лись корректно установленные единичные биты при выполнении условия $\boldsymbol{c}$. Если формирование маски $\boldsymbol{m}$ выполняется без команд ветвления, код условного обмена также не будет содержать этих команд.

2.20. Выбор среди двух или большего количества значений

Предположим, что переменная $\boldsymbol{x}$ может принимать только два возможных значения — a и b. Пусть требуется присвоить переменной значение, отличающееся от текущего, причем алгоритм не должен зависеть от конкретных значений a и b. Например, в компиляторе $\boldsymbol{x}$ может быть кодом одной из операций — *переход при выполнении условия* или *переход при невыполнении условия*, и вам требуется изменить текущий код на противоположный.

Вот простейший код переключения:

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

А это его эквивалент в языке С:

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

Однако существенно лучше (по крайней мере эффективнее) использовать другой способ: $\boldsymbol{x} \leftarrow \boldsymbol{a} + \boldsymbol{b} - \boldsymbol{x}$ или $\boldsymbol{x} \leftarrow \boldsymbol{a} \oplus \boldsymbol{b} \oplus \boldsymbol{x}$. Если $\boldsymbol{a}$ и $\boldsymbol{b}$ — константы, то вычисления требуют всего одной или двух команд из базового множества RISC-команд. Естественно, переполнение при вычислении $\boldsymbol{a} + \boldsymbol{b}$ можно игнорировать.

Возникает закономерный вопрос: существует ли эффективный метод циклического перебора трех и более значений? Иными словами, пусть заданы три произвольные, не равные друг другу константы a, b и c. Требуется найти легко вычисляемую функцию f, которая удовлетворяет условиям

$$f(a) = b; \quad f(b) = c; \quad f(c) = a.$$

Любопытно отметить, что всегда существует полиномиальная функция, удовлетворяющая указанным условиям. В случае трех констант это функция

$$f(x) = \frac{(x-a)(x-b)}{(c-a)(c-b)}a + \frac{(x-b)(x-c)}{(a-b)(a-c)}b + \frac{(x-c)(x-a)}{(b-c)(b-a)}c. \qquad (2)$$

Идея заключается в том, что при $\boldsymbol{x} = \boldsymbol{a}$ первый и последний члены функции обращаются в нуль и остается только средний член при коэффициенте $\boldsymbol{b}$ и т.д. Для вычисления этой функции необходимо выполнить 14 арифметических операций, при этом в общем случае промежуточные результаты превышают размер машинного слова, хотя мы имеем дело всего лишь с квадратичной функцией. Однако если для вычисления полинома применить схему Горнера[3], то понадобится выполнить только пять арифметических операций (четыре для вычисления квадратов с целыми коэффициентами и заключительное деление). После преобразований (2) получим следующий вид функции:

---

[3] Правило Горнера состоит в вынесении **x** за скобки. Например, полином четвертой степени $\boldsymbol{ax^4 + bx^3 + cx^2 + dx + e}$ вычисляется как $\boldsymbol{x(x(x(ax+b)+c)+d)+e}$. Для вычисления полинома степени **n** требуется **n** умножений и **n** сложений; при этом наличие команды умножения со сложением существенно повышает эффективность вычислений.

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

Полученное выражение слишком громоздко и непригодно для практического использования.

Еще один метод аналогичен формуле (2) в том смысле, что при вычислении также остается только один из трех членов функции:

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

Для вычисления этого выражения, при условии наличия в наборе команд предиката равенства, требуется выполнить 11 команд, не считая команды загрузки констант. Так как две команды сложения суммируют два нулевых значения с одним ненулевым, их можно заменить командами *и* или *исключающего или*.

Данную формулу можно упростить, если предварительно вычислить значения $\boldsymbol{a}-\boldsymbol{c}$ и $\boldsymbol{b}-\boldsymbol{c}$ [19]:

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

Каждое из этих выражений вычисляется восемью командами. Но на большинстве компьютеров это, вероятно, ничуть не эффективнее простого кода на языке C, для выполнения которого требуется от четырех до шести команд для малых значений `a`, `b` и `c`.

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

В [19] предлагается еще один оригинальный метод без переходов с циклом выборки из трех значений, работающий даже там, где нет команд сравнения. На большинстве машин его можно выполнить за восемь команд.

Пусть $\boldsymbol{a}$, $\boldsymbol{b}$ и $\boldsymbol{c}$ не равны друг другу. Следовательно, имеются два разряда n_1 и n_2 такие, что биты чисел $\boldsymbol{a}$, $\boldsymbol{b}$ и $\boldsymbol{c}$ в этих разрядах различны, и два числа, оба бита которых отличаются от битов другого числа. Проиллюстрируем это на примере чисел 21, 31 и 20.

1	0	1	0	1	$\boldsymbol{c}$
1	1	1	1	1	$\boldsymbol{a}$
1	0	1	0	0	$\boldsymbol{b}$
	$\boldsymbol{n}_1$			$\boldsymbol{n}_2$	

Без потери общности переименуем $\boldsymbol{a}$, $\boldsymbol{b}$ и $\boldsymbol{c}$ так, чтобы биты $\boldsymbol{a}$ и $\boldsymbol{b}$ были различны в обоих разрядах, как и показано выше. Тогда существует два возможных набора битов в позиции n_1, а именно $(\boldsymbol{a}_{n_1},\boldsymbol{b}_{n_1},\boldsymbol{c}_{n_1})=(0,0,1)$ или $(1,0,0)$. Аналогично, для битов в позиции n_2 также возможны два варианта: $(\boldsymbol{a}_{n_2},\boldsymbol{b}_{n_2},\boldsymbol{c}_{n_2})=(0,1,0)$ или $(1,0,1)$. Это приводит нас к рассмотрению четырех возможных случаев, формулы для которых приведены ниже.

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

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

Случай 2. $\left(a_{n_1}, b_{n_1}, c_{n_1}\right) = (0,1,1), \quad \left(a_{n_2}, b_{n_2}, c_{n_2}\right) = (1,0,1):$

$$f(x) = x_{n_1} * (a - b) + x_{n_2} * (a - c) + (b + c - a)$$

Случай 3. $\left(a_{n_1}, b_{n_1}, c_{n_1}\right) = (1,0,0), \quad \left(a_{n_2}, b_{n_2}, c_{n_2}\right) = (0,1,0):$

$$f(x) = x_{n_1} * (b - a) + x_{n_2} * (c - a) + a$$

Случай 4. $\left(a_{n_1}, b_{n_1}, c_{n_1}\right) = (1,0,0), \quad \left(a_{n_2}, b_{n_2}, c_{n_2}\right) = (1,0,1):$

$$f(x) = x_{n_1} * (b - a) + x_{n_2} * (a - c) + c$$

В этих формулах левый операнд каждого умножения представляет собой отдельный бит. Умножение на 0 или 1 можно заменить операцией *и* с 0 или с числом, все биты которого равны 1. Таким образом, эти формулы можно переписать иначе, например для первого случая получим:

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

Поскольку все переменные, кроме x, являются константами, функцию можно вычислить восемью базовыми RISC-командами. Как и ранее, команды сложения и вычитания можно заменить командами *исключающего или*.

Данная идея может быть распространена на случай циклической выборки из четырех и более констант. Главное — найти такие номера битов n_1, n_2, …, которые бы идентифицировали константы единственным образом. Для четырех констант всегда достаточно трех позиций битов. Затем (в случае четырех констант) решаются уравнения для s, t, u и v (т.е. решается система из четырех линейных уравнений, в которой $f(x)$ принимает значения a, b, c и d, а коэффициенты x_{n_i} равны 0 или 1):

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

Если все четыре константы определяются двумя битами единственным образом, то уравнение принимает следующий вид:

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