ГЛАВА 10

ЦЕЛОЕ ДЕЛЕНИЕ НА КОНСТАНТЫ

На многих компьютерах операция деления выполняется довольно медленно, и по возможности ее использования стараются избежать. Время выполнения команды деления, как правило, превышает время выполнения элементарного *сложения* в 20 и более раз, причем обычно оно одинаково велико как для больших, так и для малых операндов. В этой главе приводится ряд методов, позволяющих избежать команды *деления* в случаях, когда делитель представляет собой константу.

10.1. Знаковое деление на известную степень 2

Похоже, многие ошибочно считают, что *знаковый сдвиг вправо на k позиций* делит число на 2^k с использованием обычного отекающего деления [38]. Однако на самом деле не все так просто. Приведенный ниже код выполняет деление $\boldsymbol{q} = \boldsymbol{n} \div \mathbf{2}^k$, где $1 \le k \le 31$ [52].

```
shrsi  t,n,k-1        Формируем целое число
shri   t,t,32-k       2**k - 1, если n < 0; иначе 0
add    t,n,t          Добавляем его к n
shrsi  q,t,k          и выполняем знаковый сдвиг вправо
```

Этот код не содержит команд ветвления. Он может быть упрощен до трех команд при делении на 2 ($k = 1$). Однако данный код имеет смысл только на компьютерах, которые способны выполнить сдвиг на большое значение за малое время. Случай $k = 31$ не имеет особого смысла, поскольку число 2^{31} не представимо на компьютере. Тем не менее приведенный код дает правильный результат и в этом случае ($q = -1$, если $n = -2^{31}$, и $q = 0$ для прочих значений n).

Для деления на -2^k после приведенного выше кода должна следовать команда изменения знака. Лучшего варианта для данного деления пока не придумано.

Вот еще один, более очевидный и простой код для деления на 2^k.

```
        bge    n,label      Ветвление при n >= 0
        addi   n,n,2**k-1   Прибавление 2**k - 1 к n
label   shrsi  n,n,k        и знаковый сдвиг вправо
```

Этот код предпочтительнее использовать на машине с медленным сдвигом и быстрыми командами ветвления.

Компьютер PowerPC имеет необычное устройство для ускорения деления на степень 2 [34]. Команда *знакового сдвига вправо* устанавливает бит переноса, если сдвигаемое число отрицательно и был сдвиг одного или нескольких единичных битов. Кроме того, данный компьютер имеет команду `addze` для сложения бита переноса с регистром. Все это позволяет реализовать деление на любую (положительную) степень 2 с помощью двух команд.

```
shrsi  q,n,k
addze  q,q
```

Единственная команда сдвига `shrsi` на k позиций выполняет знаковое деление на 2^k, которое совпадает как с модульным делением, так и с делением с округлением к меньшему значению. Это наводит на мысль о том, что такое деление может быть предпочтительнее для использования в языках высокого уровня, чем деление с отсечением, так как позволяет компилятору транслировать выражение $n/2$ в единственную команду `shrsi`. Кроме того, команда `shrsi` с последующей за ней командой `neg` выполняет модульное деление на -2^k, что, в свою очередь, указывает на преимущества использования модульного деления (впрочем, в основном это вопрос эстетического восприятия, в силу того что задача деления на отрицательную константу встречается достаточно редко).

10.2. Знаковый остаток от деления на известную степень 2

Если нам требуется вычислить и частное, и остаток от деления $\boldsymbol{n} \div \boldsymbol{2^k}$, простейший путь получения значения остатка — вычислить его по формуле $\boldsymbol{r} = \boldsymbol{n} - \boldsymbol{q} * \boldsymbol{2^k}$, для чего после вычисления частного требуется выполнение двух команд.

```
shli  r,q,k
sub   r,n,n
```

Для вычисления одного лишь остатка, похоже, необходимо выполнить около четырех или пяти команд. Один из способов вычисления состоит в использовании рассмотренной ранее последовательности из четырех команд для знакового деления на 2^k, за которой следуют две команды вычисления остатка. При этом две последовательные команды *сдвига* можно заменить командой *и*, что дает нам решение из пяти команд (четырех при $k = 1$).

```
shrsi  t,n,k-1        Формируем целое число
shri   t,t,32-k       2**k - 1, если n < 0; иначе 0
add    t,n,t          Добавляем его к n,
andi   t,t,-2**k      сбрасываем k правых битов
sub    r,n,t          и вычитаем его из n
```

Еще один метод основан на том, что

$$\operatorname{rem}\left(n, 2^k\right) = \begin{cases} n \,\&\, \left(2^k - 1\right), & n \geq 0, \\ -\left(\left(-n\right) \,\&\, \left(2^k - 1\right)\right), & n < 0. \end{cases}$$

Для того чтобы воспользоваться этой формулой, сначала вычисляем $\boldsymbol{t} \leftarrow \boldsymbol{n} \overset{s}{\gg} 31$, а затем

$$\boldsymbol{r} \leftarrow \left(\left(\operatorname{abs}\left(\boldsymbol{n}\right) \,\&\, \left(\boldsymbol{2^k - 1}\right)\right) \oplus \boldsymbol{t}\right) - \boldsymbol{t}$$

(требуется пять команд) или в случае $k = 1$, так как $\left(-\boldsymbol{n}\right) \,\&\, 1 = \boldsymbol{n} \,\&\, 1$, вычисляем

$$\boldsymbol{r} \leftarrow \left(\left(\boldsymbol{n} \,\&\, \boldsymbol{1}\right) \oplus \boldsymbol{t}\right) - \boldsymbol{t}$$

(требуется четыре команды). Этот метод не очень хорош при $k > 1$, если компьютер не оснащен командой вычисления *абсолютного значения* (в таком случае вычисление остатка будет требовать выполнения семи команд).

Еще один метод основан на том, что

$$\operatorname{rem}\left(n, 2^k\right) = \begin{cases} n \,\&\, \left(2^k - 1\right), & n \geq 0, \\ \left(\left(n + 2^k - 1\right) \,\&\, \left(2^k - 1\right)\right) - \left(2^k - 1\right), & n < 0. \end{cases}$$

Эта формула приводит к вычислениям

$$\boldsymbol{t} \leftarrow \left(\boldsymbol{n} \overset{s}{\gg} k - 1\right) \overset{u}{\gg} 32 - k,$$

$$\boldsymbol{r} \leftarrow \left(\left(\boldsymbol{n} + \boldsymbol{t}\right) \,\&\, \left(\boldsymbol{2}^k - \boldsymbol{1}\right)\right) - \boldsymbol{t}$$

(при $k > 1$ требуется выполнение пяти команд, при $k = 1$ — четырех).

Все перечисленные методы работают при $1 \leq k \leq 31$.

Кстати, если в компьютере нет команды *знакового сдвига вправо*, то значение, которое равно $2^k - 1$ для $n < 0$ и 0 для $n \geq 0$, может быть построено следующим образом.

$$\boldsymbol{t}_1 \leftarrow \boldsymbol{n} \overset{u}{\gg} 31$$
$$\boldsymbol{r} \leftarrow \left(\boldsymbol{t}_1 \ll k\right) - \boldsymbol{t}_1$$

Это приводит к добавлению только одной команды.

10.3. Знаковое деление и вычисление остатка для других случаев

Основной прием в этом случае состоит в умножении на некоторое соответствующее делителю *d* число, примерно равное $2^{32}/d$, с последующим выделением 32 левых битов произведения. Однако реальные вычисления для ряда делителей, в частности для 7, оказываются существенно сложнее.

Рассмотрим сначала несколько конкретных примеров, которые пояснят код, генерируемый обобщенным методом. Обозначим регистры следующим образом.

`n` — входное целое число (числитель)
`M` — в него загружается “магическое число”
`t` — временный регистр
`q` — в нем будет размещено частное
`r` — в нем будет размещен остаток

Деление на 3

```
li    M,0x55555556  Загрузка "магического числа" (2**32+2)/3
mulhs q,M,n         q = floor(M*n/2**32)
shri  t,n,31        Прибавляем 1 к q, если
add   q,q,t         n отрицательно
```

```
muli  t,q,3          Вычисляем остаток как
sub   r,n,t          r = n - q*3
```

Доказательство. Операция *старшее слово знакового умножения* (`mulhs`) не может вызвать переполнения, поскольку произведение двух 32-битовых чисел всегда может быть представлено 64-битовым числом, а команда `mulhs` возвращает старшие 32 бит 64-битового произведения. Это действие эквивалентно делению 64-битового произведения на 2^{32} и вычислению функции floor() от полученного результата, причем это замечание справедливо независимо от того, отрицательно рассматриваемое произведение или положительно. Таким образом, при $n \geq 0$ приведенный выше код вычисляет

$$q = \left\lfloor \frac{2^{32}+2}{3} \frac{n}{2^{32}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{2n}{3 \cdot 2^{32}} \right\rfloor .$$

Далее, $n < 2^{31}$, поскольку $2^{31}-1$ является наибольшим представимым целым числом. Следовательно, член-ошибка $2n/\left(3 \cdot 2^{32}\right)$ меньше $1/3$ (и это значение неотрицательно), так что в соответствии с теоремой D4 (с. 210) получаем $q = \lfloor n/3 \rfloor$, что и требовалось получить (формула (1) на с. 209).

В случае $n < 0$ к частному добавляется 1. Следовательно, приведенный выше код вычисляет в этом случае величину

$$q = \left\lfloor \frac{2^{32}+2}{3} \frac{n}{2^{32}} \right\rfloor + 1 = \left\lfloor \frac{2^{32}n + 2n + 3 \cdot 2^{32}}{3 \cdot 2^{32}} \right\rfloor = \left\lceil \frac{2^{32}n + 2n + 1}{3 \cdot 2^{32}} \right\rceil .$$

Здесь была использована теорема D2. Следовательно,

$$q = \left\lceil \frac{n}{3} + \frac{2n+1}{3 \cdot 2^{32}} \right\rceil .$$

При $-2^{31} \leq n \leq -1$ получаем

$$-\frac{1}{3} + \frac{1}{3 \cdot 2^{32}} \leq \frac{2n+1}{3 \cdot 2^{32}} \leq -\frac{1}{3 \cdot 2^{32}} .$$

Таким образом, ошибка отрицательна и больше, чем $-1/3$, а значит, в соответствии с теоремой D4 $q = \lceil n/3 \rceil$, что и является искомым результатом (формула (1) на с. 209).

Итак, установлено, что вычисленное значение частного корректно. Тогда корректно и значение остатка, поскольку остаток должен удовлетворять соотношению

$$n = qd + r .$$

Умножение на 3 не может вызвать переполнения (так как $-2^{31}/3 \leq q \leq (2^{31}-1)/3$), а *вычитание* не может привести к переполнению в связи с тем, что результат должен находиться в диапазоне от -2 до $+2$.

Команда *умножения на непосредственно заданное значение* может быть выполнена с помощью двух *сложений* или *сдвига* и *сложения* в том случае, если такая замена дает выигрыш во времени вычислений.

На многих современных RISC-компьютерах частное может быть вычислено так, как показано выше, за девять-десять тактов, в то время как команда *деления* может потребовать 20 тактов или около того.

Деление на 5

Для деления на 5 хотелось бы использовать код того же типа, что и для деления на 3, но, понятно, с множителем $(2^{32}+4)/5$. К сожалению, при этом ошибка оказывается слишком большой и результат отличается на единицу от точного примерно для пятой части значений $|n| \geq 2^{30}$. Но оказывается, можно использовать множитель $(2^{33}+3)/5$ и добавить к коду команду *знакового сдвига вправо*. В результате получается следующий код.

```
li    M,0x66666667  Загрузка "магического числа" (2**33+3)/5
mulhs q,M,n         q = floor(M*n/2**32)
shrsi q,q,1
shri  t,n,31        Прибавляем 1 к q, если
add   q,q,t         n отрицательно

muli  t,q,5         Вычисляем остаток как
sub   r,n,t         r = n - q*5
```

Доказательство. Команда `mulhs` дает 32 старших бита 64-битового произведения, после чего код знаково сдвигает полученное значение вправо на одну позицию. Это действие эквивалентно делению 64-битового произведения на 2^{33} и вычислению функции floor() от полученного результата. Таким образом, для $n \geq 0$ приведенный код вычисляет

$$q = \left\lfloor \frac{2^{33}+3}{5}\frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{5} + \frac{3n}{5\cdot 2^{33}} \right\rfloor.$$

Для $0 \leq n < 2^{31}$ ошибка составляет $3n/\left(5\cdot 2^{33}\right)$; это значение неотрицательно и меньше, чем $1/5$, так что в соответствии с теоремой D4 $q = \lfloor n/5 \rfloor$.

При $n < 0$ приведенный выше код вычисляет значение

$$q = \left\lfloor \frac{2^{33}+3}{5}\frac{n}{2^{33}} \right\rfloor + 1 = \left\lceil \frac{n}{5} + \frac{3n+1}{5\cdot 2^{33}} \right\rceil.$$

Здесь член-ошибка отрицателен и превышает $-1/5$, так что $q = \lceil n/5 \rceil$. Корректность вычисленного остатка доказывается так же, как и в случае деления на 3. Как и ранее, команда *умножения на непосредственно заданное значение* может быть заменена — в данном случае *сдвигом влево* на две позиции и *сложением*.

Деление на 7

При делении на 7 возникают новые проблемы. Множители $(2^{32}+3)/7$ и $(2^{33}+6)/7$ дают слишком большие значения ошибки. Множитель $(2^{34}+5)/7$ подходит, но он

слишком велик для размещения в 32-битовом знаковом слове. Однако умножение на такое большое число можно выполнить путем умножения на отрицательное число $(2^{34}+5)/7-2^{32}$ с последующей коррекцией произведения путем сложения. В результате получаем следующий код для деления на 7.

```
li     M,0x92492493  "Магическое число" (2**34+5)/7 - 2**32
mulhs  q,M,n         q = floor(M*n/2**32).
add    q,q,n         q = floor(M*n/2**32) + n
shrsi  q,q,2         q = floor(q/4)
shri   t,n,31        Прибавляем 1 к q, если
add    q,q,t         n отрицательно

muli   t,q,7         Вычисляем остаток как
sub    r,n,t         r = n - q*7
```

Доказательство. Важно обратить внимание на то, что команда "`add q,q,n`" не может вызвать переполнения. Дело в том, что q и n имеют противоположные знаки; это связано с умножением на отрицательное число. Таким образом, это "компьютерное сложение" выполняется так же, как и обычное арифметическое сложение, и для $n \ge 0$ приведенный выше код вычисляет

$$q = \left\lfloor \left(\left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) \Big/ 4 \right\rfloor \right) \Big/ 4 \right\rfloor = \left\lfloor \left\lfloor \frac{2^{34}n + 5n - 7\cdot 2^{32}n + 7\cdot 2^{32}n}{7\cdot 2^{32}} \right\rfloor \Big/ 4 \right\rfloor = \left\lfloor \frac{n}{7} + \frac{5n}{7\cdot 2^{34}} \right\rfloor$$

(здесь при преобразованиях использовано следствие из теоремы D3).

Для $0 \le n < 2^{31}$ ошибка, составляющая $5n/\left(7\cdot 2^{34}\right)$, неотрицательна и меньше $1/7$, так что $q = \lfloor n/7 \rfloor$.

Для $n < 0$ приведенный выше код вычисляет значение

$$q = \left\lfloor \left(\left\lfloor \left(\left\lfloor \left(\frac{2^{34}+5}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) \Big/ 4 \right\rfloor \right) \Big/ 4 \right\rfloor + 1 = \left\lceil \frac{n}{7} + \frac{5n+1}{7\cdot 2^{34}} \right\rceil .$$

В этом случае ошибка не положительна и больше $-1/7$, так что $q = \lceil n/7 \rceil$.

Команда *умножения на непосредственно заданное значение* может быть заменена *сдвигом влево* на три позиции и *вычитанием*.

10.4. Знаковое деление на делитель, не меньший 2

После рассмотренного материала вы можете заинтересоваться, не возникают ли новые проблемы при работе с другими делителями. В этом разделе вы узнаете, что не возникают: все проблемы при делителе $d \ge 2$ исчерпываются тремя рассмотренными случаями.

Некоторые из приводимых доказательств весьма сложны, так что читайте дальнейший материал внимательно и не забывайте о том, что работаете со словом размером W.

Итак, пусть даны размер слова $W \ge 3$ и делитель $2 \le d < 2^{W-1}$ и требуется найти наименьшее целое $0 \le m < 2^W$ и целое $p \ge W$, такие, что

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \text{ для } 0 \le n < 2^{W-1} \text{ и} \tag{1,а}$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \text{ для } -2^{W-1} \le n \le -1. \tag{1,б}$$

Найти *наименьшее* целое *m* необходимо потому, что меньший множитель может потребовать меньшую величину сдвига (возможно, 0) либо привести к коду наподобие кода из примера "деление на 5", но не "деление на 7". Требование $m \le 2^W - 1$ обеспечивает не большее количество команд, чем в примере деления на 7 (т.е. с множителем в диапазоне от 2^{W-1} до $2^W - 1$ можно справиться с помощью дополнительной команды `add`, как это было сделано в примере деления на 7, но предпочтительнее обойтись меньшими множителями). Требование $p \ge W$ нужно постольку, поскольку генерируемый код выделяет левую половину произведения *mn*, что эквивалентно сдвигу вправо на *W* позиций. Таким образом, общий сдвиг вправо составляет не менее *W* позиций.

Есть определенное различие между множителем *m* и "магическим числом", обозначаемым как *M*. Магическое число — это значение, которое используется в команде *умножения* и задается следующим образом:

$$M = \begin{cases} m, & \text{если } 0 \le m < 2^{W-1}, \\ m - 2^W, & \text{если } 2^{W-1} \le m < 2^W. \end{cases}$$

В силу того, что соотношение (1,б) должно выполняться при $n = -d$, т.е. $\left\lfloor -md/2^p \right\rfloor + 1 = -1$, получаем

$$\frac{md}{2^p} > 1. \tag{2}$$

Пусть n_c — наибольшее (положительное) значение *n*, такое, что $\mathrm{rem}(n_c, d) = d - 1$. Оно существует, поскольку имеется как минимум одно возможное значение $n_c = d - 1$. Его можно вычислить как $n_c = \left\lfloor 2^{W-1}/d \right\rfloor d - 1 = 2^{W-1} - \mathrm{rem}\left(2^{W-1}, d\right) - 1$. Поскольку n_c представляет собой одно из *d* наибольших допустимых значений *n*, получаем

$$2^{W-1} - d \le n_c \le 2^{W-1} - 1 \tag{3,а}$$

и, очевидно,

$$n_c \ge d - 1. \tag{3,б}$$

Поскольку (1,а) должно выполняться при $n = n_c$,

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d}$$

или

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}.$$

Объединяя полученный результат с (2), получим

$$\frac{2^p}{d} < m < \frac{2^p}{d}\frac{n_c + 1}{n_c}. \tag{4}$$

Поскольку m должно быть наименьшим целым, удовлетворяющим (4), оно представляет собой целое число, следующее за $2^p/d$, т.е.

$$\boxed{m = \frac{2^p + d - \mathrm{rem}\left(2^p, d\right)}{d}}. \tag{5}$$

Комбинируя полученную формулу с правой частью (4), получим

$$\boxed{2^p > n_c\left(d - \mathrm{rem}\left(2^p, d\right)\right)}. \tag{6}$$

Алгоритм

Таким образом, алгоритм поиска магического числа M и величины сдвига s для данного делителя d начинается с вычисления n_c, а затем неравенство (6) решается путем подстановки последовательных возрастающих значений. Если $p < W$, то устанавливаем $p = W$ (теорема ниже показывает, что данное значение также удовлетворяет неравенству (6)). Когда найдено наименьшее $p \geq W$, удовлетворяющее неравенству (6), из (5) вычисляется значение m. Это наименьшее возможное значение m, поскольку нами найдено наименьшее приемлемое p, а из (4) понятно, что, чем меньше значение p, тем меньше и значение m. И наконец $s = p - W$ и M просто являются интерпретацией m как знакового целого числа (как оно рассматривается командой `mulhs`).

То, что при $p < W$ мы устанавливаем $p = W$, обосновывается следующей теоремой.

ТЕОРЕМА DC1. *Если для некоторого значения p справедливо неравенство (6), то оно справедливо и для больших значений p.*

Доказательство. Предположим, что неравенство (6) справедливо для $p = p_0$. Умножение (6) на 2 дает

$$2^{p_0+1} > n_c\left(2d - 2\,\mathrm{rem}\left(2^{p_0}, d\right)\right).$$

Из теоремы D5 $\mathrm{rem}\left(2^{p_0+1}, d\right) \geq 2\,\mathrm{rem}\left(2^{p_0}, d\right) - d$. Объединяя эти выражения, получим

$$2^{p_0+1} > n_c\left(2d - \left(\mathrm{rem}\left(2^{p_0+1}, d\right) + d\right)\right), \text{ или } 2^{p_0+1} > n_c\left(d - \mathrm{rem}\left(2^{p_0+1}, d\right)\right).$$

Следовательно, неравенство (6) справедливо для $p = p_0 + 1$, а потому и для всех бо́льших значений.

Таким образом, можно решить (6) путем бинарного поиска, хотя, по-видимому, предпочтительнее простой линейный поиск, начинающийся со значения $p = W$, поскольку d обычно мало, а малые значения d приводят к малым значениям p.

Доказательство пригодности алгоритма

Покажем, что (6) всегда имеет решение и что $0 \le m < 2^W$ (нет необходимости показывать, что $p \ge W$, так как это условие выполняется принудительно).

Покажем, что (6) всегда имеет решение, получив верхнюю границу p. Кроме того, в познавательных целях получим также нижнюю границу в предположении, что p не обязано быть равным как минимум W. Для получения указанных границ p заметим, что для любого положительного целого x существует степень 2, большая x и не превосходящая $2x$. Следовательно, из (6)

$$n_c\left(d - \mathrm{rem}\left(2^p, d\right)\right) < 2^p \le 2n_c\left(d - \mathrm{rem}\left(2^p, d\right)\right).$$

Поскольку $0 \le \mathrm{rem}(2^p, d) \le d - 1$,

$$n_c + 1 \le 2^p \le 2n_c d. \tag{7}$$

Из (3,а) и (3,б) получаем $n_c \ge \max(2^{W-1} - d, d - 1)$. Графики функций $f_1(d) = 2^{W-1} - d$ и $f_2(d) = d - 1$ пересекаются в точке $d = (2^{W-1} + 1)/2$. Следовательно, $n_c \ge (2^{W-1} - 1)/2$. Поскольку n_c — целое число, $n_c \ge 2^{W-2}$. С учетом того, что $n_c, d \le 2^{W-1} - 1$, (7) превращается в

$$2^{W-2} + 1 \le 2^p \le 2\left(2^{W-1} - 1\right)^2$$

или

$$W - 1 \le p \le 2W - 2. \tag{8}$$

Нижняя граница $p = W - 1$ вполне может быть достигнута (например, для $W = 32$, $d = 3$), но в этом случае мы устанавливаем $p = W$.

Если не делать p равным W принудительно, то из (4) и (7) получаем

$$\frac{n_c + 1}{d} < m < \frac{2n_c d}{d}\frac{n_c + 1}{n_c}.$$

Применение (3,б) дает

$$\frac{d - 1 + 1}{d} < m < 2\left(n_c + 1\right).$$

Так как $n_c \le 2^{W-1} - 1$ (3,а),

$$2 \le m \le 2^W - 1.$$

Если p принудительно делается равным W, то из (4) получаем

$$\frac{2^W}{d} < m < \frac{2^W}{d}\frac{n_c + 1}{n_c}.$$

Поскольку $2 \le d \le 2^{W-1}-1$ и $n_c \ge 2^{W-2}$,

$$\frac{2^W}{2^{W-1}-1} < m < \frac{2^W}{2}\frac{2^{W-2}+1}{2^{W-2}} \text{ или}$$

$$3 \le m \le 2^{W-1}+1 .$$

Следовательно, в любом случае m находится в пределах границ для схемы, проиллюстрированной примером деления на 7.

Доказательство корректности произведения

Покажем, что если p и m вычисляются из (6) и (5), то выполняются уравнения (1,а) и (1,б).

Уравнение (5) и неравенство (6), как легко видеть, вытекают из (4). (В случае, когда p принудительно приравнивается к W, как показывает теорема DC1, неравенство (6) остается справедливым.) Далее рассмотрим отдельно пять диапазонов значений n:

$$0 \le n \le n_c ,$$
$$n_c + 1 \le n \le n_c + d - 1 ,$$
$$-n_c \le n \le -1 ,$$
$$-n_c - d + 1 \le n \le -n_c - 1 \text{ и}$$
$$n = -n_c - d .$$

Из (4), поскольку m — целое число, следует

$$\frac{2^p}{d} < m \le \frac{2^p (n_c+1)-1}{dn_c} .$$

При умножении на $n/2^p$ для $n \ge 0$ это соотношение превращается в

$$\frac{n}{d} \le \frac{mn}{2^p} \le \frac{2^p n(n_c+1)-n}{2^p dn_c} , \text{ так что}$$

$$\left\lfloor \frac{n}{d} \right\rfloor \le \left\lfloor \frac{mn}{2^p} \right\rfloor \le \left\lfloor \frac{n}{d} + \frac{(2^p-1)n}{2^p dn_c} \right\rfloor .$$

Если $0 \le n \le n_c$, то $0 \le (2^p-1)n/(2^p dn_c) < 1/d$, так что по теореме D4

$$\left\lfloor \frac{n}{d} + \frac{(2^p-1)n}{2^p dn_c} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor .$$

Следовательно, в случае $0 \le n \le n_c$ уравнение (1,а) выполняется.

Для $n > n_c$ значение n ограничено диапазоном

$$n_c + 1 \le n \le n_c + d - 1 , \qquad (9)$$

так как $n \ge n_c + d$ противоречит выбору n_c как наибольшего значения n, такого, что $\mathrm{rem}(n_c, d) = d - 1$ (кроме того, из (3,а) видно, что $n \ge n_c + d$ влечет за собой $n \ge 2^{W-1}$). Из (4) для $n \ge 0$ следует

$$\frac{n}{d} < \frac{mn}{2^p} < \frac{n}{d}\frac{n_c + 1}{n_c}.$$

Путем элементарных алгебраических преобразований это неравенство может быть записано следующим образом:

$$\frac{n}{d} < \frac{mn}{2^p} < \frac{n_c + 1}{d} + \frac{(n - n_c)(n_c + 1)}{dn_c}. \tag{10}$$

Из (9) $1 \le n - n_c \le d - 1$, так что

$$0 < \frac{(n - n_c)(n_c + 1)}{dn_c} \le \frac{d - 1}{d}\frac{n_c + 1}{n_c}.$$

Поскольку (из (3,б)) $n_c \ge d - 1$ и $(n_c + 1)/n_c$ максимально при минимальном n_c,

$$0 < \frac{(n - n_c)(n_c + 1)}{dn_c} \le \frac{d - 1}{d}\frac{d - 1 + 1}{d - 1} = 1.$$

В (10) член $(n_c + 1)/d$ является целым числом. Член $(n - n_c)(n_c + 1)/dn_c$ меньше или равен 1. Следовательно, (10) превращается в

$$\left\lfloor \frac{n}{d} \right\rfloor \le \left\lfloor \frac{mn}{2^p} \right\rfloor \le \frac{n_c + 1}{d}.$$

Для всех n из диапазона (9) $\lfloor n/d \rfloor = (n_c + 1)/d$. Следовательно, (1,а) выполняется и в этом случае ($n_c + 1 \le n \le n_c + d - 1$).

Для $n < 0$ из (4), так как m — целое число, имеем

$$\frac{2^p + 1}{d} \le m < \frac{2^p}{d}\frac{n_c + 1}{n_c}.$$

При умножении на $n/2^p$ с учетом того, что $n < 0$, это выражение преобразуется:

$$\frac{n}{d}\frac{n_c + 1}{n_c} < \frac{mn}{2^p} \le \frac{n}{d}\frac{2^p + 1}{2^p}$$

или

$$\left\lfloor \frac{n}{d}\frac{n_c + 1}{n_c} \right\rfloor + 1 \le \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \le \left\lfloor \frac{n}{d}\frac{2^p + 1}{2^p} \right\rfloor + 1.$$

Применив теорему D2, получим

$$\left\lceil \frac{n_c\left(n_c+1\right)-dn_c+1}{dn_c} \right\rceil +1 \le \left\lfloor \frac{mn}{2^p} \right\rfloor +1 \le \left\lceil \frac{n\left(2^p+1\right)-2^p d+1}{2^p d} \right\rceil +1,$$

$$\left\lceil \frac{n\left(n_c+1\right)+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor +1 \le \left\lceil \frac{n\left(2^p+1\right)+1}{2^p d} \right\rceil .$$

Поскольку $n+1 \le 0$, правое неравенство может быть ослаблено, что дает нам

$$\left\lceil \frac{n}{d}+\frac{n+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor +1 \le \left\lceil \frac{n}{d} \right\rceil . \tag{11}$$

Для $-n_c \le n \le -1$

$$\frac{-n_c+1}{dn_c} \le \frac{n+1}{dn_c} \le 0 \text{ или}$$

$$-\frac{1}{d} < \frac{n+1}{dn_c} \le 0 .$$

Следовательно, в силу теоремы D4

$$\left\lceil \frac{n}{d}+\frac{n+1}{dn_c} \right\rceil = \left\lceil \frac{n}{d} \right\rceil ,$$

так что в данном случае ($-n_c \le n \le -1$) уравнение (1,б) выполняется.

При $n < -n_c$ величина n ограничена диапазоном

$$-n_c - d \le n \le -n_c - 1 . \tag{12}$$

(Исходя из (3,а), из $n < -n_c - d$ вытекает $n < -2^{W-1}$, что невозможно.) Выполняя элементарные алгебраические преобразования в левой части (11), получаем

$$\left\lceil \frac{-n_c-1}{d}+\frac{\left(n+n_c\right)\left(n_c+1\right)+1}{dn_c} \right\rceil \le \left\lfloor \frac{mn}{2^p} \right\rfloor +1 \le \left\lceil \frac{n}{d} \right\rceil . \tag{13}$$

Для $-n_c - d + 1 \le n \le -n_c - 1$

$$\frac{\left(-d+1\right)\left(n_c+1\right)}{dn_c}+\frac{1}{dn_c} \le \frac{\left(n+n_c\right)\left(n_c+1\right)+1}{dn_c} \le \frac{-\left(n_c+1\right)+1}{dn_c} = -\frac{1}{d} .$$

Отношение $\left(n_c+1\right)/n_c$ максимально при минимальном значении n_c ; таким образом, $n_c = d-1$. Следовательно,

$$\frac{(-d+1)(d-1+1)}{d(d-1)}+\frac{1}{dn_c}\le\frac{(n+n_c)(n_c+1)+1}{dn_c}<0, \quad \text{или}$$

$$-1<\frac{(n+n_c)(n_c+1)+1}{dn_c}<0.$$

Из (13), поскольку $(-n_c-1)/d$ — целое число, а прибавляемая величина находится между 0 и -1, получим

$$\frac{-n_c-1}{d}\le\left\lfloor\frac{mn}{2^p}\right\rfloor+1\le\left\lceil\frac{n}{d}\right\rceil.$$

Для n из диапазона $-n_c-d+1\le n\le -n_c-1$

$$\left\lceil\frac{n}{d}\right\rceil=\frac{-n_c-1}{d}.$$

Следовательно, $\lfloor mn/2^p\rfloor+1=\lceil n/d\rceil$, т.е. выполняется (1,б).

Последний случай, $n=-n_c-d$, может осуществиться только для некоторых значений d. Из (3,а) вытекает, что $-n_c-d\le -2^{W-1}$, поэтому если n принимает указанное значение, то $n=-n_c-d=-2^{W-1}$ и, следовательно, $n_c=2^{W-1}-d$. Таким образом, $\mathrm{rem}(2^{W-1},d)=\mathrm{rem}(n_c+d,d)=d-1$ (т.е. d является делителем $2^{W-1}+1$).

Для рассматриваемого случая $n=-n_c-d$ формула (6) имеет решение $p=W-1$ (наименьшее возможное значение p), поскольку при этом

$$n_c\left(d-\mathrm{rem}\left(2^p,d\right)\right)=\left(2^{W-1}-d\right)\left(d-\mathrm{rem}\left(2^{W-1},d\right)\right)=$$

$$=\left(2^{W-1}-d\right)\left(d-(d-1)\right)=2^{W-1}-d<2^{W-1}=2^p.$$

Тогда из (5)

$$m=\frac{2^{W-1}+d-\mathrm{rem}\left(2^{W-1},d\right)}{d}=\frac{2^{W-1}+d-(d-1)}{d}=\frac{2^{W-1}+1}{d}.$$

Таким образом,

$$\left\lfloor\frac{mn}{2^p}\right\rfloor+1=\left\lfloor\frac{2^{W-1}+1}{d}\,\frac{-2^{W-1}}{2^{W-1}}\right\rfloor+1=\left\lfloor\frac{-2^{W-1}-1}{d}\right\rfloor+1=$$

$$=\left\lceil\frac{-2^{W-1}-d}{d}\right\rceil+1=\left\lceil\frac{-2^{W-1}}{d}\right\rceil=\left\lceil\frac{n}{d}\right\rceil,$$

так что уравнение (1,б) справедливо.

Этим завершается доказательство того, что если m и p вычислены по формулам (5) и (6), то уравнения (1,а) и (1,б) выполняются для всех возможных значений n.

10.5. Знаковое деление на делитель, не превышающий –2

Поскольку знаковое целое деление удовлетворяет соотношению $n\div(-d)=-(n\div d)$, вполне приемлемой реализацией будет генерация кода для деления $n\div|d|$ с последую-

щей командой обращения знака частного. (В этом случае мы не получим верный результат при делении на $d = -2^{W-1}$, но для этой и других отрицательных степеней 2 можно воспользоваться кодом из раздела 10.1, "Знаковое деление на известную степень 2", на с. 231, после которого просто изменить знак полученного результата.) Менять знак делимого при этом не следует из-за того, что делимое может оказаться максимальным по абсолютному значению отрицательным числом.

Однако можно избежать команды обращения знака. Схема такого вычисления имеет следующий вид.

$$q = \left\lfloor \frac{mn}{2^p} \right\rfloor \quad \text{при} \quad n \le 0$$

$$q = \left\lfloor \frac{mn}{2^p} \right\rfloor + 1 \quad \text{при} \quad n > 0$$

Однако добавление 1 при положительных значениях n неудобно (так как невозможно просто использовать знаковый бит n), так что вместо этого будет выполняться прибавление 1, если значение q отрицательно. Эти действия эквивалентны, поскольку сомножитель m отрицателен (как будет показано далее).

Генерируемый для случая $W = 32$, $d = -7$ код показан ниже.

```
li    M,0x6DB6DB6D  Магическое число -(2**34+5)/7 + 2**32
mulhs q,M,n         q = floor(M*n/2**32)
sub   q,q,n         q = floor(M*n/2**32) - n
shrsi q,q,2         q = floor(q/4)
shri  t,q,31        Прибавляем 1 к q, если
add   q,q,t         q отрицательно (n положительно)

muli  t,q,-7        Вычисляем остаток по формуле
sub   r,n,t         r = n - q*(-7).
```

Этот код очень напоминает код для деления на +7 с тем отличием, что он использует множитель для деления на +7, но с обратным знаком, команду `sub` вместо команды `add` после умножения, а кроме того, команда сдвига `shri` на 31 позицию использует q, а не n (о чем говорилось ранее). (Заметим, что в случае деления на +7 также можно использовать в качестве операнда q, но при этом код будет иметь меньшую степень параллелизма.) Команда *вычитания* не может привести к переполнению, поскольку операнды имеют один и тот же знак. Тем не менее эта схема работает не всегда! Хотя приведенный код для $W = 32$, $d = -7$ вполне корректен, аналогичные изменения кода деления на 3 для получения кода для деления на -3 приведут к неверному результату при $W = 32$, $n = -2^{31}$.

Рассмотрим ситуацию подробнее.

Пусть даны размер слова $W \ge 3$ и делитель $-2^{W-1} \le d \le -2$ и требуется найти наименьшее (по абсолютному значению) целое $-2^W \le m \le 0$ и целое $p \ge W$, такие, что

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \quad \text{для } -2^{W-1} \le n \le 0 \text{ и} \qquad (14,a)$$

$$\left\lfloor \frac{mn}{2^p} \right\rfloor + 1 = \left\lceil \frac{n}{d} \right\rceil \quad \text{для } 1 \le n < 2^{W-1}. \tag{14,б}$$

Поступим так же, как и в случае деления на положительный делитель. Пусть n_c — наибольшее по абсолютной величине отрицательное значение n, такое, что $n_c = kd + 1$ для некоторого целого k. Такая величина n_c существует в силу того, что имеется по крайней мере одно значение $n_c = d + 1$. Это значение можно вычислить, исходя из того, что $n_c = \left\lfloor \left(-2^{W-1} - 1\right) / d \right\rfloor d + 1 = -2^{W-1} + \mathrm{rem}\left(2^{W-1} + 1, d\right)$. Величина n_c представляет собой одно из $|d|$ наименьших допустимых значений n, так что

$$-2^{W-1} \le n_c \le -2^{W-1} - d - 1 \tag{15,а}$$

и, очевидно,

$$n_c \le d + 1. \tag{15,б}$$

Поскольку (14,б) должно выполняться при $n = -d$, а (14,а) — для $n = n_c$, то аналогично (4) получим

$$\frac{2^p}{d} \frac{n_c - 1}{n_c} < m < \frac{2^p}{d}. \tag{16}$$

Так как m является наибольшим целым числом, удовлетворяющим (16), оно представляет собой ближайшее целое, меньшее $2^p / d$, т.е.

$$\boxed{m = \frac{2^p - d - \mathrm{rem}\left(2^p, d\right)}{d}}. \tag{17}$$

Объединяя это выражение с левой частью (16) и упрощая, получим

$$\boxed{2^p > n_c \left(d + \mathrm{rem}\left(2^p, d\right)\right)}. \tag{18}$$

Доказательство пригодности предложенного в (17) и (18) алгоритма и корректности произведения выполняется аналогично доказательству для положительного делителя и здесь не приводится. Трудности возникают только при попытках доказать, что $-2^W \le m \le 0$. Для доказательства рассмотрите по отдельности случаи, когда d представляет собой отрицательное число, абсолютное значение которого является степенью двойки, и прочие числа. Для $d = -2^k$ легко показать, что $n_c = -2^{W-1} + 1$, $p = W + k - 1$ и $m = -2^{W-1} - 1$ (которое находится в изначально определяемом диапазоне). Для тех d, которые имеют отличный от -2^k вид, достаточно просто изменить доказательство, приведенное ранее, при рассмотрении положительного делителя.

Для каких делителей $m(-d) \neq -m(d)$?

Под $m(d)$ подразумевается множитель, соответствующий делителю d. Если $m(-d) = -m(d)$, код для деления на отрицательный делитель может быть сгенерирован

путем вычисления множителя для $|d|$, изменения его знака и генерации кода, аналогичного коду из примера "деления на –7", проиллюстрированному ранее.

Сравнивая (18) с (6), а (17) с (5), можно увидеть, что если значение n_c для $-d$ представляет собой значение n_c для d, но с обратным знаком, то $m(-d) = -m(d)$. Следовательно, $m(-d) \neq -m(d)$ может быть только тогда, когда значение n_c, вычисленное для отрицательного делителя, представляет собой наибольшее по абсолютному значению отрицательное число -2^{W-1}. Такие делители представляют собой отрицательные сомножители $2^{W-1}+1$. Эти числа встречаются очень редко, что иллюстрируется приведенными ниже разложениями.

$$2^{15}+1 = 3^2 \cdot 11 \cdot 331$$
$$2^{31}+1 = 3 \cdot 715827883$$
$$2^{63}+1 = 3^3 \cdot 19 \cdot 43 \cdot 5419 \cdot 77158673929$$

Для *всех* этих множителей $m(-d) \neq -m(d)$. Вот набросок доказательства. Для $d > 0$ мы имеем $n_c = 2^{W-1} - d$. Поскольку $\text{rem}(2^{W-1}, d) = d - 1$, значение $p = W - 1$ удовлетворяет (6), а следовательно, этому неравенству удовлетворяет и $p = W$. Однако для $d < 0$ мы имеем $n_c = -2^{W-1}$ и $\text{rem}(2^{W-1}, d) = |d| - 1$. Следовательно, ни $p = W - 1$, ни $p = W$ не удовлетворяют (18), так что $p > W$.

10.6. Встраивание в компилятор

Для того чтобы компилятор мог заменить деление на константу произведением, он должен вычислить для данного делителя *d* магическое число *M* и величину сдвига *s*. Простейший способ состоит в вычислении (6) или (18) для $p = W, W+1, \ldots$ до тех пор, пока условие будет выполняться. Затем из (5) или (17) вычисляется значение *m*. Магическое число *M* представляет собой не что иное, как интерпретацию *m* как знакового целого числа, а $s = p - W$.

Описанная ниже схема обрабатывает положительные и отрицательные *d* с помощью небольшого дополнительного кода и позволяет избежать арифметических выражений с двойными словами.

Вспомним, что n_c задается следующим образом.

$$n_c = \begin{cases} 2^{W-1} - \text{rem}\left(2^{W-1}, d\right) - 1 & \text{при} \quad d > 0 \\ -2^{W-1} + \text{rem}\left(2^{W-1}+1, d\right) & \text{при} \quad d < 0 \end{cases}$$

Следовательно, $|n_c|$ можно вычислить так.

$$t = 2^{W-1} + \begin{cases} 0, & d > 0 \\ 1, & d < 0 \end{cases}$$
$$|n_c| = t - 1 - \text{rem}\left(t, |d|\right)$$

Остаток должен вычисляться с использованием беззнакового деления, в соответствии со значениями аргументов. Здесь используется запись $\text{rem}\left(t, |d|\right)$, а не эквивалентная ей

$\mathrm{rem}(t,d)$ именно для того, чтобы подчеркнуть, что программа должна работать с двумя положительными (и беззнаковыми) аргументами.

Исходя из (6) и (18), значение p может быть вычислено из соотношения

$$2^p > |n_c|\left(|d| - \mathrm{rem}\left(2^p, |d|\right)\right), \tag{19}$$

после чего $|m|$ можно вычислить следующим образом (используя (5) и (17)).

$$|m| = \frac{2^p + |d| - \mathrm{rem}\left(2^p, |d|\right)}{|d|} \tag{20}$$

Непосредственное вычисление $\mathrm{rem}(2^p, |d|)$ в (19) требует применения "длинного деления" (деление $2W$-битового делимого на W-битовый делитель и получение W-битовых частного и остатка), причем это длинное деление должно быть *беззнаковым*. Однако имеется путь решения (19), который позволяет избежать использования длинного деления и может быть легко реализован в обычных языках программирования высокого уровня с использованием только W-битовой арифметики. Тем не менее нам потребуются беззнаковое деление и беззнаковое сравнение.

Вычислить $\mathrm{rem}(2^p, |d|)$ можно инкрементно, инициализируя переменные q и r значениями частного и остатка от деления 2^p на $|d|$ при $p = W - 1$, а затем обновляя значения q и r по мере роста p.

В процессе поиска при увеличении значения p на 1 значения q и r изменяются следующим образом (см. теорему D5 (первая формула)).

```
q = 2*q;
r = 2*r:
if (r >= abs(d))
{
    q = q + 1;
    r = r - abs(d);
}
```

Из левой половины неравенства (4) и правой половины (16) вместе с доказанными границами для m следует, что $q = \left\lfloor 2^p / |d| \right\rfloor < 2^W$, так что q можно представить W-битовым беззнаковым числом. Кроме того, $0 \le r < |d|$, так что r можно представить W-битовым знаковым или беззнаковым числом. (*Внимание*: промежуточный результат $2r$ может превысить $2^{W-1} - 1$, поэтому r должно быть беззнаковым числом, и выполняющиеся выше сравнения также должны быть беззнаковыми.)

Далее вычисляется $\delta = |d| - r$. Оба члена разности представимы в виде W-битовых беззнаковых целых чисел (так же, как и разность $1 \le \delta \le |d|$), так что данное вычисление не представляет никаких трудностей.

Для того чтобы избежать длинного умножения в (19), перепишем его.

$$\frac{2^p}{|n_c|} > \delta$$

Величина $2^p/|n_c|$ представима в виде W-битового беззнакового целого числа (аналогично (7) из (19) может быть показано, что $2^p \le 2|n_c|\cdot|d|$, и для $d = -2^{W-1}$, $n_c = -2^{W-1}+1$ и $p = 2W-2$, так что $2^p/|n_c| = 2^{2W-2}/(2^{W-1}-1) < 2^W$ при $W \ge 3$). Эти вычисления так же, как и в случае $\mathrm{rem}(2^p,|d|)$, могут быть инкрементными (при увеличении p). Сравнение для случая $2^p/|n_c| \ge 2^{W-1}$ (что может произойти при больших d) должно быть беззнаковым.

Для вычисления m не требуется непосредственное вычисление (20), при котором может понадобиться длинное деление. Заметим, что

$$\frac{2^p + |d| - \mathrm{rem}\left(2^p,|d|\right)}{|d|} = \left\lfloor \frac{2^p}{|d|} \right\rfloor + 1 = q + 1 .$$

Проверка завершения цикла $2^p/|n_c| > \delta$ вычисляется достаточно сложно. Величина $2^p/|n_c|$ доступна только в виде частного q_1 и остатка r_1. Величина $2^p/|n_c|$ может быть (а может и не быть) целой (эта величина является целой только для $d = 2^{W-2}+1$ и некоторых отрицательных значений d). Проверка $2^p/|n_c| \le \delta$ может быть закодирована следующим образом.

$$\boldsymbol{q}_1 < \boldsymbol{\delta} \,|\, \left(\boldsymbol{q}_1 = \boldsymbol{\delta} \,\&\, \boldsymbol{r}_1 = \boldsymbol{0}\right)$$

Полностью процедура для вычисления `M` и `s` для данного `d` показана в листинге 10.1, в котором представлена программа на языке C для $W = 32$. Здесь есть несколько мест, где может произойти переполнение, однако если его игнорировать, то полученный результат оказывается корректным.

Листинг 10.1. Вычисление магического числа для знакового деления

```
struct ms
{
    int M;    // Магическое число
    int s;    // Величина сдвига
};

struct ms magic(int d)
{   // d должно удовлетворять условию
    // 2 <= d <= 2**31-1 или -2**31 <= d <= -2
    int p;
    unsigned ad, anc, delta, q1, r1, q2, r2, t;
    const unsigned two31 = 0x80000000;  // 2**31
    struct ms mag;
    ad = abs(d);
    t = two31 + ((unsigned)d >> 31);
    anc = t - 1 - t%ad; // Абсолютное значение nc
    p = 31;             // Инициализация p
    q1 = two31/anc;     // Инициализация q1 = 2**p/|nc|
    r1 = two31 - q1*anc;// Инициализация r1 = rem(2**p,|nc|)
    q2 = two31/ad;      // Инициализация q2 = 2**p/|d|
    r2 = two31 - q2*ad; // Инициализация r2 = rem(2**p,|d|)
    do {
        p = p + 1;
        q1 = 2*q1;      // Обновление q1 = 2**p/|nc|.
        r1 = 2*r1;      // Обновление r1 = rem(2**p,|nc|)
```

```
        if (r1 >= anc)  // Здесь требуется беззнаковое
        {               // сравнение
            q1 = q1 + 1;
            r1 = r1 - anc;
        }
        q2 = 2*q2;      // Обновление q2 = 2**p/|d|
        r2 = 2*r2;      // Обновление r2 = rem(2**p,|d|)
        if (r2 >= ad)   // Здесь требуется беззнаковое
        {               // сравнение
            q2 = q2 + 1;
            r2 = r2 - ad;
        }
        delta = ad - r2;
    } while(q1 < delta || (q1 == delta && r1 == 0));
    mag.M = q2 + 1;
    if (d < 0)
        mag.M = -mag.M; // Магическое число и
    mag.s = p - 32;     // величина сдвига
    return mag;
}
```

Для использования результата этой программы компилятор должен сгенерировать команды `li` и `mulhs`, команду `add` при `d` > 0 и `M` < 0 или `sub` при `d` < 0 и `M` > 0, а также команду `shrsi` при `s` > 0. После этого генерируются команды `shri` и `add`.

В случае $W = 32$ можно избежать обработки отрицательного делителя, если использовать предвычисленный результат для $d = 3$ и $d = 715\,827\,883$, а для остальных отрицательных делителей использовать формулу $m(-d) = -m(d)$. Однако при этом программа из листинга 10.1 не станет значительно короче (если вообще сократится).

10.7. Дополнительные вопросы

ТЕОРЕМА DC2. *Наименьший множитель m нечетен, если p не равно W в принудительном порядке.*

Доказательство. Предположим, что уравнения (1,а) и (1,б) выполняются при наименьшем (не установленном принудительно) значении p и четном m. Очевидно, что тогда m можно разделить на 2, а p уменьшить на 1 и уравнения останутся удовлетворены. Но это противоречит предположению о том, что p — наименьшее значение, при котором выполняются данные уравнения.

Единственность

Магическое число для данного делителя может быть единственным (как, например, в случае $W = 32$, $d = 7$), но зачастую это не так. Более того, эксперименты показывают, что как раз обычно имеется несколько магических чисел для одного делителя. Например, для $W = 32$, $d = 6$ имеется четыре магических числа.

$$M = \quad 715\,827\,833 \qquad \left(\left(2^{32}+2\right)/6\right), \qquad s = 0$$

$$M = \quad 1\,431\,655\,766 \qquad \left(\left(2^{32}+2\right)/3\right), \qquad s = 1$$

$$M = \quad -1431655765 \qquad \left(\left(2^{33}+1\right)/3-2^{32}\right), \qquad s=2$$

$$M = \quad -1431655764 \qquad \left(\left(2^{33}+4\right)/3-2^{32}\right), \qquad s=2$$

Тем не менее имеется следующее свойство единственности.

ТЕОРЕМА DC3. *Для данного делителя d существует только один множитель m, имеющий минимальное значение p, если p не приравнивается к W в принудительном порядке.*

Доказательство. Рассмотрим сначала случай $d > 0$. Разность между верхней и нижней границами в неравенстве (4) составляет $2^p/dn_c$. Мы уже доказали (7), т.е. что если p минимально, то $2^p/dn_c \le 2$. Таким образом, может быть не более двух значений m, которые удовлетворяют неравенству (4). Пусть m равно минимальному из этих значений, задаваемому соотношением (5); тогда второе допустимое значение — $m+1$.

Пусть p_0 — наименьшее значение p, для которого $m+1$ удовлетворяет правой части неравенства (4) (значение p_0 не приравнено к W в принудительном порядке). Тогда

$$\frac{2^{p_0}+d-\operatorname{rem}\left(2^{p_0},d\right)}{d}+1<\frac{2^{p_0}}{d}\,\frac{n_c+1}{n_c}.$$

Это выражение упрощается до

$$2^{p_0} > n_c\left(2d-\operatorname{rem}\left(2^{p_0},d\right)\right).$$

Деление на 2 дает

$$2^{p_0-1} > n_c\left(d-\frac{1}{2}\operatorname{rem}\left(2^{p_0},d\right)\right).$$

Поскольку в соответствии с теоремой D5 на с. 210 $\operatorname{rem}\left(2^{p_0},d\right) \le 2\operatorname{rem}\left(2^{p_0-1},d\right)$, получаем

$$2^{p_0-1} > n_c\left(d-\operatorname{rem}\left(2^{p_0-1},d\right)\right),$$

что противоречит предположению о минимальности p_0.

Доказательство для случая $d < 0$ аналогично и здесь не приводится.

Делители с лучшими программами

Программа для $d = 3$, $W = 32$ оказывается короче программы для общего случая, поскольку в ней не требуется выполнение команд `add` и `shrsi` после команды `mulhs`. Возникает вопрос: для каких делителей программа также оказывается укороченной?

Рассмотрим только положительные делители. Итак, требуется найти целые числа m и p, которые удовлетворяют уравнениям (1,а) и (1,б) и для которых выполняются соотношения $p = W$ и $0 \le m \le 2^{W-1}$. Поскольку любые целые числа m и p, которые удовлетворяют уравнениям (1,а) и (1,б), должны также удовлетворять неравенству (4), достаточно найти те делители d, для которых (4) имеет решение при $p = W$ и $0 \le m \le 2^{W-1}$. Все решения (4) при $p = W$ задаются уравнением

$$m = \frac{2^W + kd - \mathrm{rem}\left(2^W, d\right)}{d}, \quad k = 1, 2, 3\ldots$$

Объединяя его с правой частью (4) и упрощая, получим

$$\mathrm{rem}\left(2^W, d\right) > kd - \frac{2^W}{n_c}. \tag{21}$$

Наименьшим ограничением на $\mathrm{rem}(2^W, d)$ является ограничение при $k = 1$ и n_c, равном своему минимальному значению 2^{W-2}, так что

$$\mathrm{rem}\left(2^W, d\right) > d - 4,$$

т.е. d является делителем $2^W + 1$, $2^W + 2$ или $2^W + 3$.

Посмотрим теперь, какие из данных делителей в действительности имеют оптимальные программы.

Если d является делителем $2^W + 1$, то $\mathrm{rem}(2^W, d) = d - 1$. Тогда решением (6) является $p = W$, так как неравенство превращается в очевидное:

$$2^W > n_c\left(d - \left(d - 1\right)\right) = n_c,$$

поскольку $n_c < 2^{W-1}$. При вычислении m имеем

$$m = \frac{2^W + d - \left(d - 1\right)}{d} = \frac{2^W + 1}{d},$$

и при $d \geq 3$ это значение оказывается меньше, чем 2^{W-1} (d не может быть равно 2, поскольку является делителем $2^W + 1$). Следовательно, все делители $2^W + 1$ имеют оптимальные программы.

Аналогично, если d является делителем $2^W + 2$, то $\mathrm{rem}(2^W, d) = d - 2$. Здесь также решением (6) является $p = W$, поскольку неравенство также превращается в

$$2^W > n_c\left(d - \left(d - 2\right)\right) = 2n_c,$$

которое, очевидно, является истинным. Вычисление m дает значение

$$m = \frac{2^W + d - \left(d - 2\right)}{d} = \frac{2^W + 2}{d},$$

превышающее $2^{W-1} - 1$ для $d = 2$, но не превышающее $2^{W-1} - 1$ при $W \geq 3$, $d \geq 3$ (случай $W = 3$ и $d = 3$ невозможен, поскольку 3 не является делителем $2^3 + 2 = 10$). Следовательно, все делители $2^W + 2$, за исключением числа 2 и дополнительного к нему (дополнительным к 2 делителем является $(2^W + 2)/2$, т.е. число, которое нельзя представить как W-битовое знаковое целое).

Если d является делителем $2^W + 3$, то показать, что оптимальной программы для него нет, можно следующим образом. Поскольку $\mathrm{rem}(2^W, d) = d - 3$, из неравенства (21) имеем

$$n_c < \frac{2^W}{kd - d + 3}\,(2^W + 2)$$

для некоторого $k = 1, 2, 3, \ldots$. Самое слабое ограничение — при $k = 1$, так что должно выполняться $n_c < 2^W/3$.

Из (3,а) $n_c \geq 2^{W-1} - d$ или $d \geq 2^{W-1} - n_c$, следовательно, получаем

$$d > 2^{W-1} - \frac{2^W}{3} = \frac{2^W}{6}.$$

Кроме того, поскольку 2, 3 и 4 не могут быть делителями $2^W + 3$, наименьшим возможным делителем $2^W + 3$ является 5 и, следовательно, наибольший возможный делитель равен $(2^W + 3)/5$. Таким образом, если *d* является делителем $2^W + 3$ и *d* имеет оптимальную программу, то

$$\frac{2^W}{6} < d \leq \frac{2^W + 3}{5}.$$

Преобразуя данное неравенство, для дополнительного к *d* делителя $(2^W + 3)/d$ получаем следующие границы:

$$5 \leq \frac{2^W + 3}{d} < \frac{\left(2^W + 3\right)\cdot 6}{2^W} = 6 + \frac{18}{2^W}.$$

Эти границы указывают, что при $W \geq 5$ единственными возможными дополнительными делителями являются 5 и 6. Легко убедиться, что при $W < 5$ делителей $2^W + 3$ не существует. Поскольку 6 не может быть делителем $2^W + 3$, единственным возможным делителем этого числа может быть 5. Таким образом, единственным возможным делителем $2^W + 3$, который может иметь оптимальную программу, является $(2^W + 3)/5$.

Для $d = (2^W + 3)/5$

$$n_c = \left\lfloor \frac{2^{W-1}}{\left(2^W + 3\right)/5} \right\rfloor \left(\frac{2^W + 3}{5} \right) - 1,$$

а так как для $W \geq 4$

$$2 < \frac{2^{W-1}}{\left(2^W + 3\right)/5} < 2.5,$$

получаем

$$n_c = 2\left(\frac{2^W + 3}{5} \right) - 1.$$

Эта величина превышает $2^W/3$, так что $d=(2^W+3)/5$ оптимальной программы не имеет. Поскольку для $W<4$ величина 2^W+3 делителей не имеет, можно сделать вывод о том, что делителей 2^W+3 с оптимальной программой не существует.

Резюмируем: все делители 2^W+1 и 2^W+2, за исключением 2 и $(2^W+2)/2$, имеют оптимальные программы, и никакие другие числа оптимальных программ не имеют. Более того, приведенное выше доказательство показывает, что алгоритм из листинга 10.1 всегда дает оптимальную программу, если таковая существует.

Рассмотрим частные случаи значений *W*, равные 16, 32 и 64. Соответствующие разложения на множители показаны ниже.

$$2^{16}+1=65537\ \text{(простое)}$$
$$2^{16}+2=2\cdot 3^2\cdot 11\cdot 331$$
$$2^{32}+1=641\cdot 6\,700\,417$$
$$2^{32}+2=2\cdot 3\cdot 715\,827\,883$$
$$2^{64}+1=247\,177\cdot 67\,280\,421\,310\,721$$
$$2^{64}+2=2\cdot 3^3\cdot 19\cdot 43\cdot 5419\cdot 77\,158\,673\,929$$

Следовательно, для $W=16$ имеется 20 делителей, для которых существует оптимальная программа. Из них меньше 100 делители 3, 6, 9, 11, 18, 22, 33, 66 и 99.

Для $W=32$ есть шесть таких делителей: 3, 6, 641, 6 700 417, 715 827 883, 1 431 655 766.

Для $W=64$ есть 126 таких делителей. Из них меньше 100 делители 3, 6, 9, 18, 19, 27, 38, 43, 54, 57 и 86.

10.8. Беззнаковое деление

Беззнаковое деление на величину, представляющую собой степень двойки, само собой, реализуется с помощью единственной команды *сдвига вправо*, а остаток — с помощью команды *и с непосредственно задаваемым операндом*.

Может показаться, что обработка других делителей должна быть несложной: нужно просто использовать результаты для знакового деления с $d>0$, опуская две команды, которые добавляют 1 при отрицательном частном. Однако вы увидите, что некоторые детали беззнакового деления в действительности несколько сложнее.

Беззнаковое деление на 3

Начнем рассмотрение беззнакового деления на другие числа с беззнакового деления на 3 на 32-разрядной машине. Поскольку делимое *n* теперь может достигать по величине $2^{32}-1$, множитель $(2^{32}+2)/3$ оказывается неадекватен (член-ошибка $2n/(3\cdot 2^{32})$ может превысить $1/3$). Однако можно использовать множитель $(2^{33}+1)/3$. Соответствующий код имеет следующий вид.

```
li    M,0xAAAAAAAB  Загрузка магического числа (2**33+1)/3
mulhu q,M,n         q = floor(M*n/2**32)
shri  q,q,1
```

```
muli  t,q,3          Вычисление остатка по формуле
sub   r,n,t          r = n - q*3
```

В этом случае нам требуется команда `mulhu`, которая возвращает старшие 32 бита беззнакового 64-битового произведения.

Чтобы убедиться в корректности приведенного кода, заметим, что он вычисляет следующее значение.

$$q = \left\lfloor \frac{2^{33}+1}{3}\frac{n}{2^{33}} \right\rfloor = \left\lfloor \frac{n}{3} + \frac{n}{3\cdot 2^{33}} \right\rfloor$$

При $0 \le n < 2^{32}$ выполняется неравенство $0 \le n/(3\cdot 2^{33}) < 1/3$, так что в соответствии с теоремой D4 $q = \lfloor n/3 \rfloor$.

При вычислении остатка может произойти переполнение при выполнении команды *умножения на непосредственно заданное значение*, если операнды рассматриваются как знаковые целые числа, но если считать операнды и результат беззнаковыми, то переполнение при выполнении данной команды невозможно. Переполнение невозможно и при выполнении команды *вычитания*, так как результат находится в диапазоне от 0 до 2, поэтому получаемый таким образом остаток корректен.

Беззнаковое деление на 7

При беззнаковом делении на 7 на 32-разрядном компьютере множители $(2^{32}+3)/7$, $(2^{33}+6)/7$ и $(2^{34}+5)/7$ оказываются неадекватными, поскольку дают слишком большой член-ошибку. Можно использовать множитель $(2^{35}+3)/7$, но он слишком велик для представления 32-битовым беззнаковым числом. Умножение на это число можно выполнить посредством умножения на $(2^{35}+3)/7-2^{32}$ с последующей коррекцией путем сложения. Соответствующий код имеет следующий вид.

```
li    M,0x24924925  Магическое число (2**35+3)/7 - 2**32
mulhu q,M,n         q = floor(M*n/2**32)
add   q,q,n         Может вызвать переполнение (перенос)
shrxi q,q,3         Сдвиг вправо с битом переноса

muli  t,q,7         Вычисление остатка по формуле
sub   r,n,t         r = n - q*7
```

Здесь возникает небольшая проблема: команда `add` может вызвать переполнение. Для того чтобы обойти эту ситуацию, была придумана новая команда *расширенного сдвига вправо на непосредственно заданную величину* (`shrxi`), которая рассматривает бит переноса команды `add` и 32 бита регистра *q* как единую 33-битовую величину и выполняет ее сдвиг вправо с заполнением освободившихся разрядов нулевыми битами. В семействе процессоров Motorola 68000 эти действия можно реализовать с помощью двух команд: *расширенного циклического сдвига* вправо на один бит с последующим *беззнаковым сдвигом вправо* на 3 бита (команда `rorx` на самом деле использует X-бит, но команда `add` присваивает ему то же значение, что и биту переноса). На большинстве

компьютеров для выполнения этих действий требуется большее количество команд. Например, на PowerPC требуется выполнение трех команд: сброс правых трех битов q, прибавление бита переноса к q и циклический сдвиг вправо на три позиции.

При той или иной реализации команды `shrxi` приведенный выше код вычисляет

$$q = \left\lfloor \left(\left\lfloor \left(\frac{2^{35}+3}{7} - 2^{32} \right) \frac{n}{2^{32}} \right\rfloor + n \right) \Big/ 2^3 \right\rfloor = \left\lfloor \frac{n}{3} + \frac{3n}{7 \cdot 2^{35}} \right\rfloor .$$

При $0 \le n < 2^{32}$ выполняется неравенство $0 \le 3n/(7 \cdot 2^{35}) < 1/7$, так что в соответствии с теоремой D4 $q = \lfloor n/7 \rfloor$.

Гранлунд (Granlund) и Монтгомери (Montgomery) [39] предложили остроумную схему, позволяющую избежать применения команды `shrxi`. Она требует того же количества команд, что и рассмотренная выше схема, но использует только элементарные команды, которые есть практически у каждого компьютера; переполнения при этом оказываются просто невозможны. Схема использует следующее тождество.

$$\left\lfloor \frac{q+n}{2^p} \right\rfloor = \left\lfloor \left(\left\lfloor \frac{n-q}{2} \right\rfloor + q \right) \Big/ 2^{p-1} \right\rfloor, \quad p \ge 1$$

Применим данное тождество к нашей задаче, полагая $q = \left\lfloor Mn/2^{32} \right\rfloor$, где $0 \le M < 2^{32}$. Вычитание не может вызвать переполнения, поскольку

$$0 \le q = \left\lfloor \frac{Mn}{2^{32}} \right\rfloor \le n ,$$

поэтому очевидно, что $0 \le n - q < 2^{32}$. Сложение также не может вызвать переполнения, поскольку

$$\left\lfloor \frac{n-q}{2} \right\rfloor + q = \left\lfloor \frac{n-q}{2} + q \right\rfloor = \left\lfloor \frac{n+q}{2} \right\rfloor$$

и $0 \le n, q < 2^{32}$.

Использование предложенных идей приводит к следующему коду для беззнакового деления на 7.

```
li    M,0x24924925  Магическое число (2**35+3)/7 - 2**32
mulhu q,M,n         q = floor(M*n/2**32)
sub   t,n,q         t = n - q
shri  t,t,1         t = (n - q)/2
add   t,t,q         t = (n - q)/2 + q = (n + q)/2
shri  q,t,2         q = (n+Mn/2**32)/8 = floor(n/7)

muli  t,q,7         Вычисление остатка по формуле
sub   r,n,t         r = n - q*7
```

Чтобы эта схема работала, величина сдвига в гипотетической команде `shrxi` должна быть больше 0. Можно показать, что если $d > 1$ и множитель $m \ge 2^{32}$ (т.е. необходима команда `shrxi`), то величина сдвига больше 0.

10.9. Беззнаковое деление на делитель, не меньший 1

Для данного размера слова $W \ge 1$ и делителя $1 \le d < 2^W$ требуется найти наименьшее целое m и целое p, такие, что

$$\left\lfloor \frac{mn}{2^p} \right\rfloor = \left\lfloor \frac{n}{d} \right\rfloor \text{ для } 0 \le n < 2^W, \tag{22}$$

при этом $0 \le m < 2^{W+1}$ и $p \ge W$.

В случае беззнакового деления магическое число M определяется как

$$M = \begin{cases} m, & 0 \le m < 2^W, \\ m - 2^W, & 2^W \le m < 2^{W+1}. \end{cases}$$

Поскольку (22) должно выполняться для $n = d$, $\lfloor md/2^p \rfloor = 1$ или

$$\frac{md}{2^p} \ge 1. \tag{23}$$

Пусть, как и в случае знакового деления, n_c представляет собой наибольшее значение n, такое, что $\mathrm{rem}(n_c, d) = d - 1$. Его можно вычислить как $n_c = \lfloor 2^W/d \rfloor d - 1 = 2^W - \mathrm{rem}(2^W, d) - 1$. Тогда

$$2^W - d \le n_c \le 2^W - 1 \tag{24,а}$$

и

$$n_c \ge d - 1. \tag{24,б}$$

Это означает, что $n_c \ge 2^{W-1}$.

Поскольку (22) должно выполняться при $n = n_c$,

$$\left\lfloor \frac{mn_c}{2^p} \right\rfloor = \left\lfloor \frac{n_c}{d} \right\rfloor = \frac{n_c - (d-1)}{d}$$

или

$$\frac{mn_c}{2^p} < \frac{n_c + 1}{d}.$$

Объединение этого неравенства с (23) дает

$$\frac{2^p}{d} \le m < \frac{2^p}{d} \frac{n_c + 1}{n_c}. \tag{25}$$

Поскольку m представляет собой наименьшее целое, удовлетворяющее неравенству (25), оно должно быть ближайшим целым числом, большим или равным $2^p/d$, т.е.

$$\boxed{m = \frac{2^p + d - 1 - \mathrm{rem}(2^p - 1, d)}{d}}. \tag{26}$$

Комбинируя это уравнение с правой частью (25) и упрощая, получим

$$\boxed{2^p > n_c\left(d-1-\operatorname{rem}\left(2^p-1,d\right)\right)}. \tag{27}$$

Беззнаковый алгоритм

Таким образом, алгоритм состоит в поиске методом проб и ошибок минимального $p \ge W$, удовлетворяющего неравенству (27), после чего из уравнения (26) вычисляется m. Это наименьшее возможное значение m, удовлетворяющее (22) при $p \ge W$. Как и в случае знакового деления, если неравенство (27) выполняется для некоторого значения p, то оно выполняется и для всех бо́льших значений p. Доказательство этого факта, по сути, ничем не отличается от доказательства теоремы DC1, только вместо первой формулы из теоремы D5 в доказательстве используется вторая.

Доказательство пригодности беззнакового алгоритма

Покажем, что (27) всегда имеет решение и что $0 \le m < 2^{W+1}$.

Поскольку для любого неотрицательного числа x имеется степень 2, бо́льшая x, но не превышающая $2x+1$, из (27) получаем

$$n_c\left(d-1-\operatorname{rem}\left(2^p-1,d\right)\right) < 2^p \le 2n_c\left(d-1-\operatorname{rem}\left(2^p-1,d\right)\right)+1.$$

Так как $0 \le \operatorname{rem}\left(2^p-1,d\right) \le d-1$,

$$1 \le 2^p \le 2n_c\left(d-1\right)+1. \tag{28}$$

С учетом того, что $n_c, d \le 2^W-1$, это неравенство превращается в

$$1 \le 2^p \le 2\left(2^W-1\right)\left(2^W-2\right)+1$$

или

$$0 \le p \le 2W. \tag{29}$$

Таким образом, неравенство (27) всегда имеет решение.

Если p не приравнивается принудительно к W, то из (25) и (28) следует

$$\frac{1}{d} \le m < \frac{2n_c\left(d-1\right)+1}{d}\,\frac{n_c+1}{n_c},$$

$$1 \le m < \frac{2d-2+1/n_c}{d}\left(n_c+1\right),$$

$$1 \le m < 2\left(n_c+1\right) \le 2^{W+1}.$$

Если же p в принудительном порядке приравнено к W, то из (25) вытекает

$$\frac{2^W}{d} \le m < \frac{2^W}{d}\,\frac{n_c+1}{n_c}.$$

В силу того, что $1 \le d \le 2^W - 1$ и $n_c \ge 2^{W-1}$,

$$\frac{2^W}{2^W - 1} \le m < \frac{2^W}{1} \frac{2^{W-1} + 1}{2^{W-1}},$$

$$2 \le m \le 2^W + 1.$$

Следовательно, в любом случае m находится в пределах схемы, проиллюстрированной примером беззнакового деления на 7.

Доказательство корректности произведения в случае беззнакового деления

Покажем, что если p и m вычислены на основании формул (27) и (26), то выполняется уравнение (22).

Легко показать, что уравнение (26) и неравенство (27) приводят к неравенству (25), которое практически тождественно неравенству (4); остальная часть доказательства, по сути, идентична доказательству для знакового деления при $n \ge 0$.

10.10. Встраивание в компилятор при беззнаковом делении

В реализации алгоритма, основанного на непосредственных вычислениях, использованных в доказательстве, имеются трудности. Хотя, как доказано выше, $p \le 2W$, случай $p = 2W$ не исключается (например, для $d = 2^W - 2,\ W \ge 4$). Если $p = 2W$, вычислить m становится сложно в силу того, что делимое в (26) не размещается в $2W$-битовом слове.

Однако реализация этих действий возможна при использовании методики “инкрементного деления и взятия остатка”, уже рассматривавшейся нами ранее и примененной для данного случая в алгоритме, который приведен в листинге 10.2 (для случая $W = 32$). В этом алгоритме, помимо магического числа и величины сдвига, возвращается индикатор необходимости генерации команды `add` (в случае знакового деления необходимость добавления этой команды распознавалась по тому, что `M` и `d` имели разные знаки).

Листинг 10.2. Вычисление магического числа для случая беззнакового деления

```
struct mu
{
    unsigned M;  // Магическое число
    int a;       // Индикатор "add"
    int s;       // Величина сдвига
};

struct mu magicu(unsigned d)
{
    // d должно быть в границах 1 <= d <= 2**32-1
    int p;
    unsigned nc, delta, q1, r1, q2, r2;
    struct mu magu;
    magu.a = 0;       // Инициализация индикатора "add"
    nc = -1 - (-d)%d; // Используется беззнаковая арифметика
    p = 31;                  // Инициализация p.
    q1 = 0x80000000/nc;      // Инициализация q1=2**p/nc
    r1 = 0x80000000 - q1*nc;// Инициализация r1=rem(2**p,nc)
```

```
    q2 = 0x7FFFFFFF/d;      // Инициализация q2=(2**p-1)/d
    r2 = 0x7FFFFFFF - q2*d; // Инициализация r2=rem(2**p-1,d)
    do {
        p = p + 1;
        if (r1 >= nc - r1)
        {
            q1 = 2*q1 + 1;   // Коррекция q1
            r1 = 2*r1 - nc;  // Коррекция r1
        }
        else
        {
            q1 = 2*q1;
            r1 = 2*r1;
        }
        if (r2 + 1 >= d - r2)
        {
            if (q2 >= 0x7FFFFFFF) magu.a = 1;
            q2 = 2*q2 + 1;     // Коррекция q2
            r2 = 2*r2 + 1 - d; // Коррекция r2
        }
        else
        {
            if (q2 >= 0x80000000) magu.a = 1;
            q2 = 2*q2;
            r2 = 2*r2 + 1;
        }
        delta = d - 1 - r2;
    } while (p < 64 &&
             (q1 < delta ||
              (q1 == delta && r1 == 0)));
    magu.M = q2 + 1;  // Возвращаемые магическое число
    magu.s = p - 32;  // и величина сдвига
    return magu;      // (magu.a установлено ранее)
}
```

Приведем ряд ключевых моментов в понимании этого алгоритма.

- В ряде мест алгоритма может произойти беззнаковое переполнение, которое должно быть проигнорировано.
- $n_c = 2^W - \mathrm{rem}\left(2^W, d\right) - 1 = \left(2^W - 1\right) - \mathrm{rem}\left(2^W - d, d\right)$.
- Частное и остаток от деления 2^p на n_c не могут быть подкорректированы так же, как и в алгоритме из листинга 10.1, поскольку здесь величина `2*r1` может вызвать переполнение. Следовательно, алгоритм должен выполнить проверку "`if (r1>=nc-r1)`" вместо более естественной проверки "`if (2*r1>=nc)`". Такое же замечание применимо и к вычислению частного и остатка от деления $2^p - 1$ на d.
- $0 \le \delta \le d - 1$, так что δ представимо в виде 32-битового беззнакового целого числа.
- $m = \left(2^p + d - 1 - \mathrm{rem}\left(2^p - 1, d\right)\right) \Big/ d = \left\lfloor \left(2^p - 1\right) \Big/ d \right\rfloor + 1 = q_2 + 1$.
- В программе не выполняется явное вычитание 2^W для случая, когда магическое число M превышает $2^W - 1$; это вычитание происходит в случае, когда вычисление `q2` вызывает переполнение.

- Индикатор `magu.a` необходимости команды `add` из-за переполнения не может быть установлен путем непосредственного сравнения `M` с 2^{32} или `q2` с $2^{32}-1$. Вместо этого программа проверяет `q2` до того, как может произойти переполнение. Если `q2` достигнет величины $2^{32}-1$, так что `M` станет не менее 2^{32}, то индикатору `magu.a` будет присвоено значение 1; в противном случае это значение останется равным 0.
- Неравенство (27) эквивалентно неравенству $2^p/n_c > \delta$.
- Проверка `p < 64` в условии цикла необходима постольку, поскольку без нее переполнение `q1` может привести к тому, что будет выполнено слишком большое количество итераций и будет получен неверный результат.

Чтобы воспользоваться результатами этой программы, компилятор должен генерировать команды `li` и `mulhu` и, в случае, если индикатор `a` применения команды `add` равен 0, генерировать команду `shri` на `s` бит (если `s > 0`), как проиллюстрировано в примере из раздела “Беззнаковое деление на 3” на с. 253. Если `a = 1` и машина оснащена командой `shrxi`, то компилятор должен генерировать `add` и `shri` на `s` бит, как проиллюстрировано в примере из раздела “Беззнаковое деление на 7” на с. 254. Если `a = 1`, но машина не имеет команды `shrxi`, используется пример, приведенный на с. 255: генерируется `sub`, `shri` на 1 бит, `add` и наконец `shri` на `s-1` бит (если `s-1 > 0`; в этом случае `s` не может быть равным 0, за исключением тривиального случая деления на 1, который, как мы предполагаем, компилятор просто удалит).

10.11. Дополнительные вопросы (беззнаковое деление)

ТЕОРЕМА DC2U. *Наименьший множитель m нечетен, если p принудительно не присваивается значение W.*

ТЕОРЕМА DC3U. *Для данного делителя d существует только один множитель m, имеющий минимальное значение p, если p не приравнивается к W в принудительном порядке.*

Доказательства этих теорем практически идентичны доказательствам соответствующих теорем для знакового деления.

Делители с лучшими программами (беззнаковое деление)

Для того чтобы найти в случае беззнакового деления делители (если таковые имеются) с оптимальными программами поиска частного, состоящими из двух команд (`li`, `mulhu`), можно выполнить почти такой же анализ, как и для случая знакового деления (см. выше раздел “Делители с лучшими программами” на с. 244). В результате анализа выясняется, что такими делителями являются делители чисел 2^W и 2^W+1 (за исключением $d=1$). Для распространенных размеров слов таких нетривиальных делителей оказывается очень мало. При использовании 16-битовых слов таких делителей нет вовсе;

для 32-битовых слов их всего два — 641 и 6700417. Два таких делителя (274177 и 67280421310721) есть и при работе с 64-битовыми словами.

Случай $d = 2^k$, $k = 1, 2, \ldots$ заслуживает отдельного упоминания. В этом случае рассмотренный алгоритм *magicu* из листинга 10.2 дает нам $p = W$ (принудительное присвоение) и $m = 2^{32-k}$. Это минимальное значение m, но не минимальное значение M. Наилучший код получается при использовании $p = W + k$. Тогда $m = 2^W$, M равно 0, a — 1, s — k. Сгенерированный код включает умножение на 0 и может быть упрощен до одной команды *сдвига вправо на величину k*. На практике делители, представляющие собой степень 2, стоит рассматривать как особый случай и не привлекать для их обработки программу `magicu`. (Этого не происходит при знаковом делении, поскольку в этом случае m не может быть степенью 2. *Доказательство*: при $d > 0$ неравенство (4), будучи скомбинированным с неравенством (3,б), дает $d - 1 < 2^p/m < d$. Следовательно, $2^p/m$ не может быть целым числом. При $d < 0$ аналогичный результат получается при комбинировании неравенств (16) и (15,б).)

При беззнаковом делении, когда компьютер не имеет команды `shrxi`, код для случая $m \geq 2^W$ оказывается значительно хуже, чем код для $m < 2^W$. Поэтому возникает вопрос, как часто приходится иметь дело с большими множителями. Для размера слова 32 бита среди целых чисел, не превышающих 100, имеется 31 "плохой" делитель: 1, 7, 14, 19, 21, 27, 28, 31, 35, 37, 38, 39, 42, 45, 53, 54, 55, 56, 57, 62, 63, 70, 73, 74, 76, 78, 84, 90, 91, 95 и 97.

Использование знакового деления вместо беззнакового и наоборот

Если ваш компьютер не оснащен командой `mulhu`, но имеет команду `mulhs` (или команду знакового длинного умножения), то можно воспользоваться приемом, о котором говорилось в разделе "Преобразование знакового и беззнакового произведений одно в другое" на с. 200.

В этом разделе была приведена последовательность из семи команд, которая позволяет получить результат выполнения команды `mulhu` из команды `mulhs`. Однако в нашей ситуации этот прием упрощается, поскольку магическое число M известно заранее, так что компилятор может протестировать старший бит заранее и сгенерировать в соответствии с его значением ту или иную последовательность действий после выполнения команды "`mulhu q,M,n`". Здесь `t` обозначает временный регистр.

```
     M31 = 0                  M31 = 1
mulhs  q,M,n            mulhs  q,M,n
shrsi  t,n,31           shrsi  t,n,31
and    t,t,M            and    t,t,M
add    q,q,t            add    t,t,n
                        add    q,q,t
```

С учетом других команд, используемых вместе с `mulhu`, всего для получения частного и остатка от беззнакового деления на константу на компьютере, не оснащенном беззнаковым умножением, требуется от шести до восьми команд.

Этот прием можно обратить для получения команды `mulhs` из команды `mulhu`. При этом получается практически такой же код, только команда `mulhs` заменяется командой `mulhu`, а последние команды `add` в обоих столбцах заменяются командами `sub`.

Более простой беззнаковый алгоритм

Отказ от требования минимальности магического числа приводит к более простому алгоритму. Вместо (27) можно использовать

$$2^p \geq 2^W \left(d - 1 - \mathrm{rem}\left(2^p - 1, d\right)\right), \tag{30}$$

а затем вычислить m, как и ранее, по формуле (26).

Должно быть очевидно, что формально этот алгоритм корректен (т.е. вычисленное значение m удовлетворяет уравнению (22)), поскольку единственное его отличие от предыдущего алгоритма состоит в том, что он вычисляет значение p, которое для некоторых значений d не обязательно будет большим. Можно доказать, что значение m, вычисленное по (30) и (26), меньше 2^{W+1}. Здесь доказательство опускается и просто приводится конечный результат — алгоритм, представленный в листинге 10.3.

Листинг 10.3. Упрощенный алгоритм вычисления магического числа для беззнакового деления

```
struct mu
{
    unsigned M;  // Магическое число
    int a;       // Индикатор "add"
    int s;       // Величина сдвига
};

struct mu magicu2(unsigned d)
{
    // d должно быть в границах  1 <= d <= 2**32-1
    int p;
    unsigned p32, q, r, delta;
    struct mu magu;
    magu.a = 0;           // Инициализация индикатора "add"
    p = 31;               // Инициализация p
    q = 0x7FFFFFFF/d;     // Инициализация q = (2**p-1)/d
    r = 0x7FFFFFFF - q*d; // Инициализация r = rem(2**p-1,d)
    do {
        p = p + 1;
        if (p == 32) p32 = 1; // p32 = 2**(p-32).
        else p32 = 2*p32;
        if (r + 1 >= d - r)
        {
            if (q >= 0x7FFFFFFF) magu.a = 1;
            q = 2*q + 1;     // Коррекция q
            r = 2*r + 1 - d; // Коррекция r
        }
        else
        {
            if (q >= 0x80000000) magu.a = 1;
            q = 2*q;
```

```
            r = 2*r + 1;
        }
        delta = d - 1 - r;
    } while (p < 64 && p32 < delta);
    magu.M = q + 1;  // Возвращаемые магическое число
    magu.s = p - 32; // и величина сдвига
    return magu;     // (magu.a установлено ранее)
}
```

Алверсон (Alverson) в [4] предложил гораздо более простой алгоритм, который рассматривается в следующем разделе, но он иногда дает очень большие значения *m*. Главное в алгоритме *magicu2* в том, что он почти всегда дает минимальное значение *m* при $d \le 2^{W-1}$. Если размер слова равен 32 битам, наименьший делитель, для которого *magicu2* не дает минимального множителя, равен 102 807 (в этом случае *magicu* дает значение *m*, равное 2 737 896 999, а *magicu2* — значение 5 475 793 997).

Существует аналог алгоритма *magicu2* и для знакового деления на положительные делители, но он плохо работает для знакового деления на произвольные делители.

10.12. Применение к модульному делению и делению с округлением к меньшему значению

Может показаться, что преобразовать в умножение модульное деление на константу или деление с округлением к меньшему значению — задача еще более простая, чем при делении с отсечением, и для этого достаточно всего лишь убрать шаг “добавить 1, если делимое отрицательно”. Однако это не так. Предложенные выше методы не применимы к другим типам деления путем простых и очевидных преобразований. Вероятно, разработанное преобразование такого типа будет включать изменение множителя *m* в зависимости от знака делимого.

10.13. Другие похожие методы

Вместо кодирования алгоритма *magic* можно воспользоваться таблицей с магическими числами и величинами сдвигов для некоторых небольших делителей. Делители, равные табличным значениям, умноженным на степень 2, легко обрабатываются следующим образом.

1. Подсчитывается количество завершающих нулевых битов в *d*, которое далее обозначается как *k*.
2. В качестве аргумента поиска в таблице используется значение $d/2^k$ (сдвиг вправо на *k* позиций).
3. Используем магическое число, найденное в таблице.
4. Используем найденную в таблице величину сдвига, увеличенную на *k*.

Таким образом, если в таблице имеются делители 3, 5, 25, то могут быть обработаны делители 6, 10, 100 и т.д.

Такая процедура обычно дает наименьшее магическое число, хотя и не всегда. Если размер слова равен 32 бита, то наименьший положительный делитель, когда процедура дает наименьшее магическое число, равен 334 972. Данный метод при этом дает значения $m = 3\,361\,176\,179$ и $s = 18$; минимальное же магическое число для данного делителя — 840 294 045, а величина сдвига — 16. Процедура также дает неминимальный результат при $d = -6$. В обоих приведенных случаях снижается качество генерируемого кода деления.

Алверсон [4] является первым известным автором, который указал на корректность работы описываемого далее метода для всех делителей. Используя наши обозначения, можно сказать, что его метод для беззнакового целого деления на d состоит в использовании величины сдвига $p = W + \lceil \log_2 d \rceil$ и множителя $m = \lceil 2^p / d \rceil$ с последующим выполнением деления $n \div d = \lfloor mn / 2^p \rfloor$ (т.е. путем *умножения* и *сдвига вправо*). Он доказал, что множитель m меньше 2^{W+1} и что этот метод дает точное значение частного для всех n, выражаемых с помощью W-битового числа.

Метод Алверсона представляет собой более простой вариант нашего, в котором для определения p не используется поиск методом проб и ошибок, а следовательно, он в большей степени подходит для аппаратной реализации, в чем и состоит его главное назначение. Однако его множитель m всегда оказывается не меньшим, чем 2^W, а потому при программной реализации всегда требуется код, проиллюстрированный в примере деления на 7 (т.е. всегда требуются либо команды `add` и `shrxi`, либо четыре альтернативные команды). Поскольку большинство небольших делителей могут быть обработаны с помощью множителей, меньших 2^W, представляется разумным рассмотреть эти случаи.

В случае знакового деления Алверсон предложил искать множитель для $|d|$ и длины слова $W-1$ (тогда $2^{W-1} \le m < 2^W$), умножать на него делимое и, если операнды имеют противоположные знаки, изменять знак полученного результата. (Множитель должен быть таким, чтобы давать корректный результат для делимого 2^{W-1}, которое представляет собой абсолютное значение максимального отрицательного числа). Похоже, что такое предложение может дать код, лучший по сравнению с кодом для множителя $m \ge 2^W$. Применяя данный прием к знаковому делению на 7, получим следующий код (для того чтобы избежать ветвления, в нем использовано соотношение $-\boldsymbol{x} = \bar{\boldsymbol{x}} + \mathbf{1}$).

```
abs   an,n
li    M,0x92492493  Магическое число (2**34+5)/7
mulhu q,M,an        q = floor(M*an/2**32)
shri  q,q,2
shrsi t,n,31        Эти три команды
xor   q,q,t         изменяют знак q,
sub   q,q,t         если n отрицательно
```

Этот код не столь хорош, как тот, который был предложен для знакового деления на 7 ранее (7 и 6 команд соответственно), но он может оказаться полезным на компьютере, на котором есть команды `abs` и `mulhu` и нет команды `mulhs`.

10.14. Некоторые магические числа

Таблица 10.1. Некоторые магические числа для $W = 32$

	Знаковое число		Беззнаковое число		
d	**M (шестнадцатеричное)**	**s**	**M (шестнадцатеричное)**	**a**	**s**
-5	99999999	1			
-3	55555555	1			
-2^k	7FFFFFFF	k-1			
1	-	-	0	1	0
2^k	80000001	k-1	2^{32-k}	0	0
3	55555556	0	AAAAAAAB	0	1
5	66666667	1	CCCCCCCD	0	2
6	2AAAAAAB	0	AAAAAAAB	0	2
7	92492493	2	24924925	1	3
9	38E38E39	1	38E38E39	0	1
10	66666667	2	CCCCCCCD	0	3
11	2E8BA2E9	1	BA2E8BA3	0	3
12	2AAAAAAB	1	AAAAAAAB	0	3
25	51EB851F	3	51EB851F	0	3
125	10624DD3	3	10624DD3	0	3
625	68DB8BAD	8	D1B71759	0	9

Таблица 10.2. Некоторые магические числа для $W = 64$

	Знаковое число		Беззнаковое число		
d	**M (шестнадцатеричное)**	**s**	**M (шестнадцатеричное)**	**a**	**s**
-5	99999999 99999999	1			
-3	55555555 55555555	1			
-2^k	7FFFFFFF FFFFFFFF	k-1			
1	-	-	0	1	0
2^k	80000000 00000001	k-1	2^{64-k}	0	0
3	55555555 55555556	0	AAAAAAAA AAAAAAAB	0	1
5	66666666 66666667	1	CCCCCCCC CCCCCCCD	0	2
6	2AAAAAAA AAAAAAAB	0	AAAAAAAA AAAAAAAB	0	2
7	49249249 24924925	1	24924924 92492493	1	3
9	1C71C71C 71C71C72	0	E38E38E3 8E38E38F	0	3
10	66666666 66666667	2	CCCCCCCC CCCCCCCD	0	3
11	2E8BA2E8 BA2E8BA3	1	2E8BA2E8 BA2E8BA3	0	1
12	2AAAAAAA AAAAAAAB	1	AAAAAAAA AAAAAAAB	0	3
25	A3D70A3D 70A3D70B	4	47AE147A E147AE15	1	5
125	20C49BA5 E353F7CF	4	0624DD2F 1A9FBE77	1	7
625	346DC5D6 3886594B	7	346DC5D6 3886594B	0	7

10.15. Простой код на языке программирования Python

Вычисление магического числа значительно упрощается, если вычисления не ограничены тем же размером слова, что и среда, в которой вычисленное магическое число будет использоваться. Например, в случае беззнакового деления в Python достаточно просто вычислить n_c, а затем (27) и (26), как описано в разделе 10.9. Соответствующая функция показана в листинге 10.4.

Листинг 10.4. Код на языке программирования Python вычисления магического числа для беззнакового деления

```
def magicgu(nmax, d):
    nc = (nmax//d)*d - 1
    nbits = int(log(nmax, 2)) + 1
    for p in range(0, 2*nbits + 1):
        if 2**p > nc*(d - 1 - (2**p - 1)%d):
        m = (2**p + d - 1 - (2**p - 1)%d)//d
        return (m, p)
    print "Не могу найти p, что-то не так".
    sys.exit(1)
```

Эта функция получает максимальное значение делимого `nmax` и делителя `d` и возвращает пару целых чисел: магическое число `m` и величину сдвига `p`. Для деления делимого `x` на `d` следует умножить `x` на `m`, а затем сдвинуть (полную длину) произведения вправо на `p` бит.

Эта программа является более обобщенной, чем прочие программы из данной главы, по двум направлениям: (1) в ней определено максимальное значение делимого (`nmax`), а не количество битов, требуемых для делимого, и (2) программа может использоваться для произвольно больших делимого и делителя. Преимущество указания максимального значения делимого заключается в возможности получить меньшее магическое число, чем при использовании в качестве этого максимального значения очередной степени двойки, уменьшенной на 1. Предположим, например, что максимальное значение делимого равно 90, а делитель равен 7. Тогда функция `magicgu` вернет пару $(37,8)$, означающую, что магическое число равно 37 (6-битовое число), а величина сдвига составляет 8 бит. Но если запросить, какое магическое число обеспечивает деление до делимого, равного 127, то результатом будет пара $(147,10)$, а 147 представляет собой 8-битовое число.

10.16. Точное деление на константу

Под “точным делением” подразумевается деление, о котором заранее известно, что его остаток равен 0. Хотя такая ситуация и не очень распространена, она имеет место, например, при вычитании двух указателей в языке C. В нем результат вычитания $p-q$, где p и q — указатели, определен и переносим, только если p и q указывают на объекты в одном и том же массиве [57, раздел 7.6.2]. Если размер элемента массива равен s, то код для вычисления разности двух указателей реально вычисляет $(p-q)/s$.

Материал этого раздела основан на [39, раздел 9].

Рассматриваемый здесь метод применим как к знаковому, так и к беззнаковому точному делению и базируется на следующей теореме.

ТЕОРЕМА MI. *Если a и m — взаимно простые целые числа, то существует целое число $1 \le \overline{a} < m$, такое, что $a\overline{a} \equiv 1 \pmod{m}$.*

Таким образом, $\overline{a}$ представляет собой обратный мультипликативный по модулю m элемент по отношению к a. Есть несколько способов доказательства этой теоремы, три из них описаны в [90, с. 52]. Доказательство, приведенное далее, требует только знания основ теории сравнений.

Доказательство. Докажем несколько более общее по сравнению с данной теоремой утверждение. Если a и m взаимно просты (и следовательно, имеют ненулевые значения), то для множества x, состоящего из m различных по модулю m значений, значения ax также будут различны по модулю m. Например, при $a = 3$ и $m = 8$ и значениях x из диапазона от 0 до 7 получим, что значения ax представляют собой множество 0, 3, 6, 9, 12, 15, 18, 21, или, по модулю 8, числа 0, 3, 6, 1, 4, 7, 2, 5. Заметим, что в полученной последовательности вновь представлены все числа от 0 до 7.

Чтобы доказать это в общем случае, воспользуемся методом доказательства от противного. Предположим, что существуют два различных по модулю m целых числа, которые отображаются в одно и то же число по модулю m при умножении на a, т.е. существуют такие x и y ($x \not\equiv y \pmod{m}$), что

$$ax \equiv ay \pmod{m}.$$

Но в таком случае существует целое число k, такое, что

$$ax - ay = km \quad \text{или}$$
$$a(x - y) = km.$$

Поскольку a не имеет общих множителей с m, разность $x - y$ должна быть кратной m, т.е.

$$x \equiv y \pmod{m}.$$

Однако этот вывод противоречит исходному предположению.

Теперь, поскольку значения ax принимают все m возможных различных значений по модулю m, среди значений x найдется такое, для которого ax по модулю m равно 1.

Приведенное доказательство показывает также, что имеется только одно значение x (по модулю m), такое, что $ax \equiv 1 \pmod{m}$, т.е. что мультипликативное обращение однозначно. Можно также показать, что имеется единственное (по модулю m) целое число x, такое, что $ax \equiv b \pmod{m}$, где b — любое целое число.

В качестве примера рассмотрим случай $m = 16$. Тогда $\overline{3} = 11$, так как $3 \cdot 11 = 33 \equiv \equiv 1 (\mathrm{mod} 16)$. Можно также считать, что $\overline{3} = -5$, поскольку $3 \cdot (-5) = -15 \equiv 1 \pmod{16}$. Аналогично $\overline{-3} = 5$, поскольку $(-3) \cdot 5 = -15 \equiv 1 \pmod{16}$.

Важность этих наблюдений заключается в том, что они показывают применимость рассматриваемых концепций как к знаковым, так и к беззнаковым числам. Если вы бу-

дете работать с беззнаковыми числами на 4-битовой машине, то получите $\overline{3} = 11$; если же будете иметь дело со знаковыми числами, то получите $\overline{3} = -5$. Однако и 11, и -5 имеют одно и то же представление в дополнительном коде (поскольку их разность равна 16), так что в обоих случаях в качестве мультипликативного обратного элемента используется одно и то же содержимое машинного слова.

Рассмотренная теорема непосредственно применима к задаче деления (знакового и беззнакового) на нечетное число d на W-битовом компьютере. Поскольку любое нечетное число является взаимно простым с 2^W, теорема гласит, что если d нечетно, то существует целое $\overline{d}$ (единственное в диапазоне от 0 до $2^W - 1$ или от -2^{W-1} до $2^{W-1} - 1$), такое, что

$$d\overline{d} \equiv 1 \left(\mathrm{mod}\ 2^W\right).$$

Следовательно, для любого n, кратного d, справедливо соотношение

$$\frac{n}{d} \equiv \frac{n}{d}\left(d\overline{d}\right) \equiv n\overline{d} \left(\mathrm{mod}\ 2^W\right).$$

Другими словами, n/d можно вычислить путем умножения n на $\overline{d}$, после чего взять правые W бит полученного произведения.

Если делитель d представляет собой четное число, то пусть $d = d_0 \cdot 2^k$, где d_0 нечетно, а $k \geq 1$. Тогда просто выполняется сдвиг числа n вправо на k позиций (устраняя нулевые биты), после чего выполняется умножение на $\overline{d}_0$ (сдвиг можно выполнить и после операции умножения).

Ниже приведен код для деления на 7 числа n, кратного 7. Этот код дает корректный результат как при знаковом, так и при беззнаковом делении.

```
li  M,0xB6DB6DB7  Мультипликативное обратное, (5*2**32+1)/7
mul q,M,n         q = n/7
```

Вычисление мультипликативного обратного по алгоритму Евклида

Каким же образом можно вычислить мультипликативное обратное число? Стандартный метод — использование расширенного алгоритма Евклида. Этот метод вкратце рассматривается далее, в приложении к нашей основной задаче, а заинтересовавшимся читателям можно посоветовать обратиться к [90, с. 13] и [67, раздел 4.5.2].

Пусть задан нечетный делитель d и нам требуется найти число x, такое, что

$$dx \equiv 1 \left(\mathrm{mod}\ m\right),$$

причем в нашей задаче $m = 2^W$ (где W — размер машинного слова). Это может быть выполнено, если уравнение

$$dx + my = 1$$

будет решено в целых числах x и y (положительных, отрицательных, равных 0).

Начнем с того, что сделаем d положительным, добавляя к нему достаточное количество раз число m (d и $d+km$ имеют одно и то же мультипликативное обратное). Затем запишем следующие уравнения (в которых $d, m > 0$):

$$d(-1)+m(1)=m-d\ , \tag{i}$$

$$d(1)+m(0)=d\ . \tag{ii}$$

Если $d=1$, поставленная задача решена, так как (ii) показывает, что $x=1$. В противном случае вычислим

$$q=\left\lfloor \frac{m-d}{d} \right\rfloor .$$

Далее умножим уравнение (ii) на q и вычтем его из (i). Это даст нам уравнение

$$d(-1-q)+m(1)=m-d-qd=\operatorname{rem}(m-d,d)\ .$$

Это уравнение справедливо, поскольку мы просто умножили одно из уравнений на константу и вычли его из другого. Если $\operatorname{rem}(m-d,d)=1$, то задача решена (последнее уравнение и является решением) и $x=-1-q$.

Повторим описанный процесс с последними двумя уравнениями, получая новое уравнение. Будем продолжать эти действия до тех пор, пока в правой части уравнения не будет получена 1. Тогда множитель при *d*, приведенный по модулю *m*, и будет представлять собой искомое мультипликативное обратное к *d*.

Кстати, если $m-d<d$, так что первое частное равно 0, то третья строка будет точной копией первой, так что второе частное будет ненулевым. Кроме того, в разных источниках зачастую в качестве первой строки используется следующая:

$$d(0)+m(1)=m\ ,$$

но в нашем приложении $m=2^W$ невозможно представить в компьютере.

Лучше всего проиллюстрировать описываемый процесс на конкретном примере. Пусть $m=256$ и $d=7$. Тогда процесс вычислений выглядит как показано ниже (для получения третьей строки используется $q=\lfloor 249/7 \rfloor = 35$).

```
 7(-1) + 256( 1) = 249
 7( 1) + 256( 0) = 7
7(-36) + 256( 1) = 4
7( 37) + 256(-1) = 3
7(-73) + 256( 2) = 1
```

Таким образом, мультипликативным обратным по модулю 256 числа 7 является –73 (или, используя числа из диапазона от 0 до 255, число 183). Проверяем: $7\cdot 183 = 1281 \equiv 1 \pmod{256}$.

Начиная с третьей строки числа в правом столбце представляют собой остатки от деления чисел, расположенных выше, так что это строго убывающая последовательность

неотрицательных чисел, которая, таким образом, должна завершиться 0 (в примере выше для этого требуется еще один шаг). Кроме того, число перед 0 должно быть 1 по следующей причине. Предположим, что последовательность остатков завершается некоторым числом *b*, не равным 1, после которого идет число 0. Тогда целое, предшествующее *b*, должно быть кратным *b* (скажем, числом $k_1 b$) с тем, чтобы следующим остатком в последовательности было число 0. Такие же рассуждения приводят к выводу, что для того, чтобы получить остаток *b*, предшествующим $k_1 b$ числом должно быть $k_1 k_2 b + b$. Продолжая эту последовательность, вы увидите, что каждое из этих чисел должно быть кратно *b*, включая числа из первых двух строк, которые равны $m - d$ и *d* и являются взаимно простыми.

Приведенное выше рассуждение является неформальным доказательством того, что рассматриваемый процесс завершается, причем наличием 1 в правом столбце, а следовательно, находит мультипликативное обратное *d* число.

Для проведения данных вычислений на компьютере заметим, что если $d < 0$, то к нему необходимо прибавить 2^W . Однако в случае арифметики, использующей дополнительный к 2 код, это делать необязательно; достаточно просто рассматривать *d* как беззнаковое число, независимо от того, как именно его рассматривает приложение.

Вычисление *q* должно использовать беззнаковое деление.

Заметим, что все вычисления можно выполнять по модулю *m*, так как это никак не влияет на правый столбец (его значения все равно остаются в диапазоне от 0 до $m-1$). Это важно, так как позволяет нам выполнять вычисления с “одинарной точностью”, используя беззнаковую компьютерную арифметику по модулю 2^W .

Большинство величин в рассматриваемой таблице не обязательно должны быть представлены при вычислениях, например столбец множителей при 256, поскольку при решении уравнения $dx + my = 1$ нас не интересует значение *y*. Нет необходимости и в представлении *d* в первом столбце. Оставляя в таблице только необходимое, получим ее сокращенный вид.

```
255   249
  1     7
220     4
 37     3
183     1
```

Программа на языке C для проведения этих вычислений представлена в листинге 10.5.

Листинг 10.5. Поиск мультипликативного обратного по модулю 2^{32} с помощью алгоритма Евклида

```
unsigned mulinv(unsigned d)  // d должно быть нечетно
{
    unsigned x1, v1, x2, v2, x3, v3, q;
    x1 = 0xFFFFFFFF;   v1 = -d;
    x2 = 1;            v2 = d;
    while (v2 > 1)
    {
        q = v1/v2;
```

```
        x3 = x1 - q*x2;    v3 = v1 - q*v2;
        x1 = x2;           v1 = v2;
        x2 = x3;           v2 = v3;
    }
    return x2;
}
```

Причина использования условия цикла `v2 > 1` вместо более естественного `v2 != 1` заключается в том, что если при использовании последнего условия функции будет ошибочно передано четное значение, то цикл может никогда не завершиться (если аргумент `d` четен, `v2` никогда не получит значение 1, но в конечном итоге примет значение 0).

Что же вычисляет данная программа, если передать ей четный аргумент? Как и следует из формул, она вычисляет число x, такое, что $dx \equiv 0 (\mathrm{mod}\ 2^{32})$, которое вряд ли может оказаться полезным. Однако минимальные изменения в программе (изменение условия цикла на `v2 != 0` и возврат значения `x1` вместо `x2`) приводят к вычислению ею числа x, такого, что $dx \equiv g (\mathrm{mod}\ 2^{32})$, где g — наибольший общий делитель d и 2^{32}, т.е. наибольшей степени 2, являющейся делителем d. Такая модифицированная программа, как и ранее, вычисляет для нечетного d мультипликативное обратное число, хотя и требует для этого на одну итерацию больше.

Что касается количества итераций, выполняемых данной программой, то для нечетного d, не превышающего 20, требуется максимум три итерации; среднее их число равно 1.7. Для d порядка 1000 требуется максимум 11, а в среднем — шесть итераций.

Вычисление мультипликативного обратного по методу Ньютона

Хорошо известно, что при работе с действительными числами значение $1/d$, где $d \neq 0$, может быть вычислено с возрастающей точностью с помощью итеративного вычисления

$$x_{n+1} = x_n \left(2 - dx_n\right) \tag{31}$$

при начальном приближении x_0, достаточно близком к $1/d$. Количество точных цифр в результате при каждой итерации примерно удваивается.

Однако гораздо менее известно, что данная формула может применяться для поиска мультипликативного обратного числа по модулю любой степени 2! Например, для поиска мультипликативного обратного к 3 по модулю 256 начнем с $x_0 = 1$ (можно использовать любое нечетное число). Тогда

$$\begin{aligned}
x_1 &= 1\left(2 - 3 \cdot 1\right) = -1, \\
x_2 &= -1\left(2 - 3\left(-1\right)\right) = -5, \\
x_3 &= -5\left(2 - 3\left(-5\right)\right) = -85, \\
x_4 &= -85\left(2 - 3\left(-85\right)\right) = -21845 \equiv -85 \left(\mathrm{mod}\ 256\right).
\end{aligned}$$

Итерации достигли неподвижной точки по модулю 256, так что -85, или 171, и есть мультипликативное обратное числа 3 по модулю 256. Все приведенные вычисления выполнялись по модулю 256.

Почему работает этот метод? Поскольку, если x_n удовлетворяет условию

$$dx_n \equiv 1 \pmod{m},$$

а x_{n+1} определяется формулой (31), то

$$dx_{n+1} \equiv 1 \pmod{m^2}.$$

Чтобы увидеть это, положим $dx_n = 1 + km$. Тогда

$$\begin{aligned} dx_{n+1} &= dx_n(2 - dx_n) \\ &= (1+km)(2-(1+km)) \\ &= (1+km)(1-km) \\ &= 1 - k^2m^2 \\ &= 1 \pmod{m^2}. \end{aligned}$$

В нашем приложении *m* является степенью 2, скажем, 2^N. В этом случае, если

$$dx_n \equiv 1 \pmod{2^N}, \text{ то}$$
$$dx_{n+1} \equiv 1 \pmod{2^{2N}}.$$

В этом смысле, если x_n рассматривается как некоторое приближение $\overline{d}$, то каждая итерация (31) удваивает количество битов "точности" приближения.

Так случилось, что по модулю 8 мультипликативным обратным любого нечетного числа *d* является само это число. Таким образом, в качестве начального приближения можно взять $x_0 = d$. Тогда по формуле (31) получим значения $x_1, x_2, \ldots$, такие, что

$$dx_1 \equiv 1 \pmod{2^6},$$
$$dx_2 \equiv 1 \pmod{2^{12}},$$
$$dx_3 \equiv 1 \pmod{2^{24}},$$
$$dx_4 \equiv 1 \pmod{2^{48}} \text{ и т.д.}$$

Таким образом, четырех итераций достаточно, чтобы найти мультипликативное обратное по модулю 2^{32} (если $x \equiv 1 \pmod{2^{48}}$, то $x \equiv 1 \pmod{2^n}$ для $n \le 48$). Это приводит нас к программе на языке C (листинг 10.6), в которой все вычисления выполняются по модулю 2^{32}.

Листинг 10.6. Вычисление мультипликативного обратного по модулю 2^{32} методом Ньютона

```
unsigned mulinv(unsigned d)  // d должно быть нечетно
{
    unsigned xn, t;

    xn = d;
loop:
    t = d*xn;
    if (t == 1) return xn;
    xn = xn*(2 - t);
    goto loop;
}
```

Примерно для половины значений *d* этой программе требуется 4.5 итерации, т.е. девять умножений. Для другой половины (у которой "верны четыре бита" начального значения `xn`, т.е. $d^2 \equiv 1 \pmod{16}$) требуется, как правило, семь (или меньше) умножений. Таким образом, можно считать, что этот метод требует в среднем восьми умножений.

Один из вариантов этой программы состоит в простом выполнении цикла четыре раза независимо от значения *d*, что позволяет обойтись восемью умножениями и без команд управления циклом. Другой вариант предусматривает выбор в качестве начального приближения мультипликативного обратного "с точностью 4 бита", т.е. перед началом вычислений нужно найти x_0, которое удовлетворяет условию $dx_0 \equiv 1 \pmod{16}$. Тогда для вычислений потребуется выполнить только три итерации цикла. Найти подходящее начальное приближение можно, например, следующими способами.

$$\boldsymbol{x_0 \leftarrow d + 2\left(\left(d + 1\right) \& 4\right)}$$
$$\boldsymbol{x_0 \leftarrow d^2 + d - 1}$$

Здесь умножение на 2 выполняется с помощью сдвига влево, а все вычисления производятся по модулю 2^{32} (игнорируя переполнения). Поскольку во второй формуле используется умножение, ее применение позволяет сэкономить только одну такую команду.

Такое пристальное рассмотрение вопроса времени вычислений, конечно, не имеет смысла, если метод применяется в компиляторе. В этом случае данная программа используется настолько редко, что при ее кодировании, в первую очередь, должен интересовать ее размер. Тем не менее могут существовать приложения, в которых поиск мультипликативного обратного должен выполняться максимально быстро.

Описанный здесь метод Ньютона применяется, только когда (1) модуль представляет собой целую степень некоторого числа *a* и (2) известно мультипликативное обратное *d* по модулю *a*. В частности, метод хорошо работает при $a = 2$, поскольку тогда тут же известно мультипликативное обратное любого нечетного числа *d* по модулю 2 — это просто 1.

Некоторые мультипликативные обратные

В завершение раздела перечислим в табл. 10.3 некоторые мультипликативные обратные числа.

Таблица 10.3. Некоторые мультипликативные обратные числа

d	$\overline{d}$		
	mod 16	**mod 2^{32}**	**mod 2^{64}**
(десятичное)	**(десятичное)**	**(шестнадцатеричное)**	**(шестнадцатеричное)**
-7	-7	49249249	92492492 49249249
-5	3	33333333	33333333 33333333
-3	5	55555555	55555555 55555555
-1	-1	FFFFFFFF	FFFFFFFF FFFFFFFF
1	1	1	1
3	11	AAAAAAAB	AAAAAAAA AAAAAAAB
5	13	CCCCCCCD	CCCCCCCC CCCCCCCD
7	7	B6DB6DB7	6DB6DB6D B6DB6DB7
9	9	38E38E39	8E38E38E 38E38E39
11	3	BA2E8BA3	2E8BA2E8 BA2E8BA3
13	5	C4EC4EC5	4EC4EC4E C4EC4EC5
15	15	EEEEEEEF	EEEEEEEE EEEEEEEF
25		C28F5C29	8F5C28F5 C28F5C29
125		26E978D5	1CAC0831 26E978D5
625		3AFB7E91	D288CE70 3AFB7E91

Вы можете заметить, что в ряде случаев ($d = 3,5,9,11$) мультипликативное обратное к d совпадает с магическим числом для беззнакового деления на d (см. раздел 10.14, “Некоторые магические числа” на с. 265). Это в большей или меньшей мере случайность. Подобное происходит для тех чисел, для которых магическое число M равно множителю m, и эти величины имеют вид $(2^p + 1)/d$ при $p \geq 32$. В этом случае

$$Md = \left(\frac{2^p + 1}{d}\right) d \equiv 1 \left(\mathrm{mod}\ 2^{32}\right),$$

так что $M \equiv \overline{d} \left(\mathrm{mod}\ 2^{32}\right)$.

10.17. Проверка нулевого остатка при делении на константу

Мультипликативное обратное делителя d может использоваться для проверки равенства 0 остатка после деления на d [39].

Беззнаковое деление

Сначала рассмотрим беззнаковое деление на нечетный делитель *d*. Обозначим через $\overline{d}$ мультипликативное обратное к *d* число. Тогда, поскольку $d\overline{d} \equiv 1 (\mathrm{mod}\ 2^W)$, где W — размер машинного слова в битах, $\overline{d}$ также нечетно. Следовательно, $\overline{d}$ является взаимно простым с 2^W и, как показано в доказательстве теоремы MI в предыдущем разделе, если *n* представляет собой множество всех 2^W различных чисел по модулю 2^W, то $n\overline{d}$ также представляет собой множество всех 2^W различных чисел по модулю 2^W.

В предыдущем разделе было показано, что если *n* кратно *d*, то

$$\frac{n}{d} \equiv n\overline{d} \left(\mathrm{mod}\ 2^W\right).$$

Следовательно, $n\overline{d} \equiv 0,1,2,\ldots,\left\lfloor \left(2^W-1\right)/d \right\rfloor \left(\mathrm{mod}\ 2^W\right)$ при $n = 0, d, 2d, \ldots, \left\lfloor \left(2^W-1\right)/d \right\rfloor d$. Таким образом, для *n*, *не* являющегося кратным *d*, значение $n\overline{d}$, приведенное по модулю 2^W к диапазону от 0 до 2^W-1, должно превышать величину $\left\lfloor \left(2^W-1\right)/d \right\rfloor$.

Этот вывод можно использовать для проверки на равенство нулю остатка от деления. Например, чтобы проверить, что число *n* кратно 25, умножим *n* на $\overline{25}$ и сравним правые *W* бит со значением $\left\lfloor \left(2^W-1\right)/25 \right\rfloor$. Использование базового набора RISC-команд дает следующий код.

```
li     M,0xC28F5C29  Загружаем мультипликативное обратное 25
mul    q,M,n         q = правая половина M*n
li     c,0x0A3D70A3  c = floor((2**32-1)/25)
cmpleu t,q,c         Сравниваем q и c; переход,
bt     t,is_mult     если n кратно 25
```

Для того чтобы распространить этот метод на четные делители, представим делитель в следующем виде: $d = d_0 \cdot 2^k$, где d_0 нечетное, а $k \geq 1$. Тогда, поскольку целое число делится на *d* тогда и только тогда, когда оно делится на d_0 и 2^k, и поскольку *n* и $n\overline{d}_0$ имеют одинаковое количество завершающих нулевых битов ($\overline{d}_0$ нечетно), проверка того, что *n* кратно *d*, состоит в следующем:

установить $q = \mathrm{mod}\left(n\overline{d}_0, 2^W\right)$;

$q \leq \left\lfloor \left(2^W-1\right)/d_0 \right\rfloor$ и q завершается не менее чем k нулевыми битами,

где под функцией “mod” подразумевается приведение $n\overline{d}_0$ к интервалу $\left[0, 2^W-1\right]$.

Непосредственная реализация описанного метода требует двух проверок и условных переходов, однако если компьютер имеет команду *циклического сдвига*, то метод можно реализовать с помощью одного *сравнения с ветвлением*. Это следует из приведенной далее теоремы, в которой запись $\boldsymbol{a} \overset{rot}{\gg} k$ означает компьютерное слово $\boldsymbol{a}$, циклически сдвинутое вправо на *k* бит ($0 \leq k \leq 32$).

ТЕОРЕМА ZRU. $\boldsymbol{x} \overset{u}{\leq} \boldsymbol{a}$ *и* $\boldsymbol{x}$ *заканчивается* k *нулевыми битами тогда и только тогда, когда* $\boldsymbol{x} \overset{rot}{\gg} k \overset{u}{\leq} \lfloor \boldsymbol{a}/2^k \rfloor$.

Доказательство. (Будем считать, что имеем дело с 32-разрядным компьютером.) Предположим, что $\boldsymbol{x} \overset{u}{\leq} \boldsymbol{a}$ и $\boldsymbol{x}$ заканчивается k нулевыми битами. Тогда, так как $\boldsymbol{x} \overset{u}{\leq} \boldsymbol{a}$, то $\lfloor \boldsymbol{x}/2^k \rfloor \overset{u}{\leq} \lfloor \boldsymbol{a}/2^k \rfloor$. Однако $\lfloor \boldsymbol{x}/2^k \rfloor = \boldsymbol{x} \overset{rot}{\gg} k$, и, таким образом, $\boldsymbol{x} \overset{rot}{\gg} k \overset{u}{\leq} \lfloor \boldsymbol{a}/2^k \rfloor$. Если $\boldsymbol{x}$ не заканчивается k нулевыми битами, то $\boldsymbol{x} \overset{rot}{\gg} k$ не начинается с k нулевых битов, в то время как $\lfloor \boldsymbol{a}/2^k \rfloor$ начинается с них, так что $\boldsymbol{x} \overset{rot}{\gg} k \overset{u}{>} \lfloor \boldsymbol{a}/2^k \rfloor$. И наконец, если $\boldsymbol{x} \overset{u}{>} \boldsymbol{a}$ и $\boldsymbol{x}$ заканчиваются k нулевыми битами, то целое число, образованное первыми $32-k$ битами $\boldsymbol{x}$ должно превышать число, образованное первыми $32-k$ битами $\boldsymbol{a}$, так что $\lfloor \boldsymbol{x}/2^k \rfloor \overset{u}{>} \lfloor \boldsymbol{a}/2^k \rfloor$.

При использовании этой теоремы проверка того, что n кратно d, где n и d — ненулевые беззнаковые целые числа, причем $d = d_0 \cdot 2^k$, где d_0 — нечетно, выполняется следующим образом.

$$\boxed{\begin{array}{l} \boldsymbol{q} \leftarrow \bmod\left(\boldsymbol{n}\overline{d}_0, \boldsymbol{2^W}\right) \\ \boldsymbol{q} \overset{rot}{\gg} k \overset{u}{\leq} \left\lfloor \left(\boldsymbol{2^W}-\boldsymbol{1}\right)/\boldsymbol{d} \right\rfloor \end{array}}$$

Здесь мы воспользовались тем, что

$$\left\lfloor \left\lfloor \left(2^W-1\right)/d_0 \right\rfloor / 2^k \right\rfloor = \left\lfloor \left(2^W-1\right)/\left(d_0 \cdot 2^k\right) \right\rfloor = \left\lfloor \left(2^W-1\right)/d \right\rfloor.$$

Далее в качестве примера приведен код, проверяющий, является ли беззнаковое целое число n кратным 100.

```
li     M,0xC28F5C29  Мультипликативное обратное 25
mul    q,M,n         q = правая половина M*n
shrri  q,q,2         Циклический сдвиг вправо на два бита
li     c,0x028F5C28  c = floor((2**32-1)/100)
cmpleu t,q,c         Сравнение q и c и переход, если
bt     t,is_mult     n кратно 100
```

Знаковое деление, делитель ≥2

В случае знакового деления, как было показано в предыдущем разделе, если n кратно d и d нечетно, получаем

$$\frac{n}{d} \equiv n\overline{d} \left(\bmod 2^W\right).$$

Таким образом, для $n = \left\lceil -2^{W-1}/d \right\rceil \cdot d, \ldots -d, 0, d, \cdots \left\lfloor \left(2^{W-1}-1\right)/d \right\rfloor \cdot d$ имеем $n\overline{d} \equiv$ $\equiv \left\lceil -2^{W-1}/d \right\rceil, \ldots \ -1, 0, 1, \cdots \left\lfloor \left(2^{W-1}-1\right)/d \right\rfloor \left(\bmod 2^W\right)$. Кроме того, поскольку $\overline{d}$ взаимно про-

стое с 2^W, если n принимает 2^W различных значений по модулю 2^W, то $n\bar{d}$ также принимает 2^W различных значений по модулю 2^W. Следовательно, n кратно d тогда и только тогда, когда

$$\left\lceil -2^{W-1}/d \right\rceil \le \mathrm{mod}\left(n\bar{d}, 2^W\right) \le \left\lfloor \left(2^{W-1}-1\right)/d \right\rfloor,$$

где под функцией "mod" подразумевается приведение $n\bar{d}$ к интервалу $\left[-2^{W-1}, 2^{W-1}-1\right]$.

Этот способ можно немного упростить, заметив, что, так как d нечетно, а из наших начальных предположений оно положительно и не равно единице, d не является делителем 2^{W-1}. Следовательно,

$$\left\lceil -2^{W-1}/d \right\rceil = \left\lceil \left(-2^{W-1}+1\right)/d \right\rceil = -\left\lfloor \left(2^{W-1}-1\right)/d \right\rfloor.$$

Для знаковых чисел проверка того, что n является кратным d, где $d = d_0 \cdot 2^k$, а d_0 нечетно, выполняется следующим образом:

установить $q = \mathrm{mod}\left(n\bar{d}_0, 2^W\right)$;

$-\left\lfloor \left(2^{W-1}-1\right)/d_0 \right\rfloor \le q \le \left\lfloor \left(2^W-1\right)/d_0 \right\rfloor$ и q завершается не менее чем k нулевыми битами.

На первый взгляд, при таком методе требуется три проверки и перехода. Однако, как и в случае беззнакового деления, все можно свести к одной команде *сравнения с ветвлением*, если воспользоваться следующей теоремой.

ТЕОРЕМА ZRS. *Если* $\boldsymbol{a} \ge \boldsymbol{0}$, *то следующие утверждения эквивалентны:*

(1) $-\boldsymbol{a} \le \boldsymbol{x} \le \boldsymbol{a}$ *и* $\boldsymbol{x}$ *заканчивается* k *или большим количеством нулевых битов,*

(2) $\mathrm{abs}(\boldsymbol{x}) \overset{rot}{\gg} k \overset{u}{\le} \left\lfloor \boldsymbol{a}/2^k \right\rfloor$,

(3) $\boldsymbol{x} + \boldsymbol{a}' \overset{rot}{\gg} k \overset{u}{\le} \left\lfloor 2\boldsymbol{a}'/2^k \right\rfloor$,

где $\boldsymbol{a}'$ *представляет собой* $\boldsymbol{a}$ *с установленными равными 0 правыми* k *битами (т.е.* $\boldsymbol{a}' = \boldsymbol{a} \,\&\, -\boldsymbol{2}^k$*).*

Доказательство. (Будем считать, что имеем дело с 32-разрядным компьютером.) Чтобы убедиться, что утверждение (1) эквивалентно утверждению (2), достаточно заметить, что выражения $-\boldsymbol{a} \le \boldsymbol{x} \le \boldsymbol{a}$ и $\mathrm{abs}(\boldsymbol{x}) \le \boldsymbol{a}$ эквивалентны. После этого эквивалентность утверждений (1) и (2) следует из теоремы ZRU.

Чтобы убедиться в эквивалентности утверждений (1) и (3), заметим, что утверждение (1) справедливо и при замене $\boldsymbol{a}$ на $\boldsymbol{a}'$. Тогда, в соответствии с "теоремой границ" на с. 90, оно, в свою очередь, эквивалентно

$$\boldsymbol{x} + \boldsymbol{a}' \overset{u}{\le} 2\boldsymbol{a}'.$$

Поскольку $\boldsymbol{x}+\boldsymbol{a}'$ заканчивается k нулевыми битами тогда и только тогда, когда k нулевыми битами заканчивается $\boldsymbol{x}$, можно применить теорему ZRU, которая и дает требующийся нам результат.

Используя утверждение (3) рассмотренной теоремы, проверку кратности n числу d, где n и d — знаковые целые числа, не меньшие 2, и $d = d_0 \cdot 2^k$, где d_0 нечетно, можно провести следующим образом.

$$\boxed{\begin{array}{l} \boldsymbol{q} \leftarrow \mathrm{mod}\left(\boldsymbol{n}\overline{\boldsymbol{d}}_0, \boldsymbol{2}^{\boldsymbol{W}}\right) \\ \boldsymbol{a}' \leftarrow \left\lfloor \left(\boldsymbol{2}^{\boldsymbol{W}-1} - \boldsymbol{1}\right) / \boldsymbol{d}_0 \right\rfloor \,\&\, -\boldsymbol{2}^{\boldsymbol{k}} \\ \boldsymbol{q} + \boldsymbol{a}' \overset{rot}{\gg} k \overset{u}{\leq} \left\lfloor \left(2\boldsymbol{a}'\right) / 2^k \right\rfloor \end{array}}$$

(Поскольку d — константа, $\boldsymbol{a}'$ можно вычислить во время компиляции.)

Далее в качестве примера приведен код, проверяющий, является ли знаковое целое число n кратным 100. Обратите внимание, что константа $\left\lfloor 2\boldsymbol{a}'/2^k \right\rfloor$ в любой момент может быть получена из константы $\boldsymbol{a}'$ путем сдвига на $k-1$ позиций, что позволяет сохранить одну команду либо загрузку из памяти для получения операнда сравнения.

```
li     M,0xC28F5C29  Загрузка мультипликативного обратного 25
mul    q,M,n         q = правая половина M*n
li     c,0x051EB850  c = floor((2**31 - 1)/25) & -4
add    q,q,c         Прибавляем c
shrri  q,q,2         Циклический сдвиг вправо на 2 позиции
shri   c,c,1         Вычисление константы для сравнения
cmpleu t,q,c         Сравнение q и c и переход, если
bt     t,is_mult     n кратно 100
```

10.18. Методы, не использующие команды умножения со старшим словом

В этом разделе мы рассмотрим некоторые методы деления на константы, в которых не используется команда *умножения со старшим словом* (*старшего слова умножения*), т.е. умножения, дающего в качестве результата двойное слово. Мы покажем, как заменить деление на константу последовательностью команд *сдвига* и *сложения* или *сдвига*, *сложения* и *умножения* для получения более компактного кода.

Беззнаковое деление

В этом случае беззнаковое деление оказывается проще знакового, так что мы начнем с него. Один из методов заключается в применении методов на основе “двухсловного” умножения, но с заменой последнего кодом, показанным в листинге 8.2 на с. 200. В листинге 10.7 показано, как такая технология работает для случая (беззнакового) деления на 3. Это комбинация кода, приведенного на с. 253, и кода из листинга 8.2 с заменой `int` на `unsigned`. Код выполняется с помощью 15 команд, включая четыре умножения. Умножения на большие константы при преобразовании к *сдвигам* и *умножениям* приводят к достаточно небольшому количеству команд. Очень похожий код может быть раз-

работан и в случае знакового деления. В целом этот метод не так уж хорош, и далее нами не рассматривается.

Листинг 10.7. Беззнаковое деление на 3 с использованием имитации команды *старшего слова беззнакового умножения*

```
unsigned divu3(unsigned n)
{
    unsigned n0, n1, w0, w1, w2, t, q;

    n0 = n & 0xFFFF;
    n1 = n >> 16;
    w0 = n0*0xAAAB;
    t  = n1*0xAAAB + (w0 >> 16);
    w1 = t & 0xFFFF;
    w2 = t >> 16;
    w1 = n0*0xAAAA + w1;
    q  = n1*0xAAAA + w2 + (w1 >> 16);
    return q >> 1;
}
```

Другой метод [37] заключается в вычислении обратного к делителю и умножении на него делимого с помощью ряда *сдвигов вправо* и *сложений*. Так мы получим приближенное значение частного. Это просто приближение, поскольку обратное к делителю (который, как предполагается, не является точной степенью двойки) не может быть точно выражен с помощью 32 бит, а также потому что каждый *сдвиг вправо* отбрасывает биты делимого. Затем вычисляется остаток по отношению к приближенному частному и делится на делитель для получения поправки, которая добавляется к приближенному частному и дает его точное значение. Остаток обычно мал по сравнению с делителем (в несколько раз меньше), так что зачастую имеется простой способ вычисления поправки без применения команды *деления*.

Для иллюстрации этого метода рассмотрим деление на 3, т.е. вычисление $\lfloor n/3 \rfloor$, где $0 \le n < 2^{32}$. Значение, обратное к 3, в бинарном представлении приближенно равно

$$0.0101\ 0101\ 0101\ 0101\ 0101\ 0101\ 0101\ 0101.$$

Для вычисления приближенного произведения этого числа и n можно воспользоваться формулой

$$\boldsymbol{q} \leftarrow \left(\boldsymbol{n} \overset{u}{\gg} 2\right) + \left(\boldsymbol{n} \overset{u}{\gg} 4\right) + \left(\boldsymbol{n} \overset{u}{\gg} 6\right) + \ldots + \left(\boldsymbol{n} \overset{u}{\gg} 30\right) \tag{32}$$

(всего 29 команд; последняя 1 в обратном значении игнорируется, поскольку соответствует добавлению члена $\boldsymbol{n} \overset{u}{\gg} 32$, который, очевидно, равен 0). Однако простое повторение нулей и единиц в обратном значении позволяет разработать метод, оказывающийся и более быстрым (девять команд), и более точным.

$$\begin{aligned}
\boldsymbol{q} &\leftarrow \left(\boldsymbol{n} \overset{u}{\gg} 2\right) + \left(\boldsymbol{n} \overset{u}{\gg} 4\right) \\
\boldsymbol{q} &\leftarrow \boldsymbol{q} + \left(\boldsymbol{q} \overset{u}{\gg} 4\right) \\
\boldsymbol{q} &\leftarrow \boldsymbol{q} + \left(\boldsymbol{q} \overset{u}{\gg} 8\right) \\
\boldsymbol{q} &\leftarrow \boldsymbol{q} + \left(\boldsymbol{q} \overset{u}{\gg} 16\right)
\end{aligned} \tag{33}$$

Для сравнения точности этих методов рассмотрим биты, удаляемые сдвигом каждого из членов (32), когда $\boldsymbol{n}$ состоит из одних единичных битов. Первый член убирает два единичных бита, следующий — четыре и т.д. Каждый сдвиг вносит ошибку, почти равную 1 младшего разряда. Поскольку всего таких членов 16 (учитывая игнорируемый член), сдвиги дают ошибку, почти равную 16. Это ошибка, дополнительная к возникающей из-за ограниченного 32-битового представления обратного к делителю значения; оказывается, максимальная общая ошибка равна 16.

В случае процедуры (33) каждый правый сдвиг также вносит ошибку, почти равную единице в младшем разряде. Однако здесь имеется только 5 операций сдвига. Они вносят ошибку, близкую к 5; имеется также ошибка, связанная с обрезкой обратного значения 32 битами. Как оказывается, максимальная общая ошибка равна 5.

После вычисления оценочного значения частного $\boldsymbol{q}$ остаток $\boldsymbol{r}$ вычисляется как

$$\boldsymbol{r} \leftarrow \boldsymbol{n} - \boldsymbol{q} * 3 .$$

Остаток не может быть отрицательным, поскольку значение $\boldsymbol{q}$ никогда не превышает точное частное. Чтобы разработать наиболее простой метод вычисления $\boldsymbol{r} \overset{u}{\div} 3$, нам надо знать, насколько больш*и*м может быть значение $\boldsymbol{r}$. В общем случае для делителя d и оценочного значения частного q, меньше точного значения частного на k, остаток находится в диапазоне от $k*d$ до $k*d+d-1$ (верхняя граница консервативна и в действительности может не быть достигнута). Таким образом, используя (33), где $\boldsymbol{q}$ может оказаться меньше, чем следует, не более чем на 5, получаем, что остаток не превышает величину $5*3+2=17$. Эксперименты показывают, что в действительности он не превышает 15. Таким образом, поправка, которую мы должны вычислить, составляет (в точности)

$$\boldsymbol{r} \overset{u}{\div} 3 \text{ при } 0 \le \boldsymbol{r} \le 15 .$$

Поскольку $\boldsymbol{r}$ мал*о* по сравнению с наибольшим значением, которое может храниться в регистре, это вычисление может быть получено приближенно путем умножения $\boldsymbol{r}$ на некоторое приближение $1/3$ вида a/b, где b представляет собой точную степень 2. Такое умножение легко вычислить, поскольку деление выполняется путем простого сдвига. Значение a/b должно быть немного больше $1/3$, чтобы после сдвига результат соответствовал делению с отсечением. Последовательность таких приближений имеет вид

$$1/2,\ 2/4,\ 3/8,\ 6/16,\ 11/32,\ 22/64,\ 43/128,\ 86/256,\ 171/512,\ 342/1024,\ \ldots .$$

Обычно чем меньше числитель и знаменатель дроби в последовательности, тем легче ее вычислить, так что выберем наименьшую подходящую. В нашем случае это дробь 11/32 . Таким образом, окончательное точное значение частного вычисляется как

$$\boldsymbol{q} \leftarrow \boldsymbol{q} + \left(11 * \boldsymbol{r} \overset{u}{\gg} 5 \right).$$

Решение включает два умножения на небольшие числа (3 и 11), которые можно заменить командами *сдвига* и *сложения*.

В листинге 10.8 показано окончательное решение на языке программирования C. Как видно из листинга, оно включает 14 команд, в том числе два умножения. Если эти умножения заменить командами *сдвига* и *сложения*, то получится 18 элементарных команд. Однако если желательно избежать умножений, то любой из указанных альтернативных операторов `return` дает решение из 17 элементарных команд. Вторая альтернатива дает несколько меньшую степень параллельности, но, по правде говоря, очень небольшой степенью параллельности обладает сам метод.

Листинг 10.8. Беззнаковое деление на 3

```
unsigned divu3(unsigned n)
{
    unsigned q, r;

    q = (n >> 2) + (n >> 4);     // q = n*0.0101 (приближенно)
    q = q + (q >> 4);            // q = n*0.01010101.
    q = q + (q >> 8);
    q = q + (q >> 16);
    r = n - q*3;                 // 0 <= r <= 15.
    return q + (11*r >> 5);      // Возврат q + r/3.
//  return q + (5*(r + 1) >> 4);           // Альтернатива 1.
//  return q + ((r + 5 + (r << 2)) >> 4);  // Альтернатива 2.
}
```

Более точная оценка получается при замене первой выполнимой строки строкой

```
q = (n >> 1) + (n >> 3);
```

(это делает `q` слишком больш*и*м (в два раза), но при этом добавляется один бит точности) и вставке непосредственно перед присваиванием `r` строки

```
q = q >> 1;
```

В этом варианте остаток не превышает 9. Однако не похоже, чтобы ограниченность $\boldsymbol{r}$ величиной 9 давала лучший код вычисления $\boldsymbol{r} \overset{u}{\div} 3$ по сравнению с границей 15 (четыре элементарные команды в любом случае). Таким образом, стоимость этой идеи — одна команда. Рассмотренная возможность упомянута здесь, поскольку она *улучшает* код для большинства делителей.

В листинге 10.9 показаны два варианта этого метода для деления на 5. Бинарное представление обратного к 5 имеет вид

$$0.0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011.$$

Как и в случае деления на 3 простое повторение шаблона из единиц и нулей позволяет достаточно эффективно и точно вычислить оценку частного. Оценка частного, вычисленная кодом слева, может отличаться от точного значения не более чем на 5, а остаток — не более чем на 25. Код справа сохраняет два бита точности при оценке частного, которая в этом случае отличается от точного значения не более чем на 2, а остаток — не более чем на 10. В этом случае наименьшим допустимым приближением $1/5$ является $7/32$, а не $13/64$, что приводит к несколько более эффективной программе при замене умножения *сдвигами* и *сложениями*. Количество команд в коде слева составляет 14 команд, включая два умножения, или 18 элементарных команд; в коде справа — 15 команд, включая 2 умножения, или 17 элементарных команд. Альтернативный код в операторе `return` используется только в том случае, когда машина оснащена командами предикатов. Он не уменьшает количество команд, а просто обладает небольшой степенью параллельности на уровне команд.

Листинг 10.9. Беззнаковое деление на 5

```
unsigned divu5a(unsigned n)
{
    unsigned q, r;

    q = (n >> 3) + (n >> 4);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    r = n - q*5;
    return q + (13*r >> 6);
}
```

```
unsigned divu5b(unsigned n)
{
    unsigned q, r;

    q = (n >> 1) + (n >> 2);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    q = q >> 2;
    r = n - q*5;
    return q + (7*r >> 5);
 // return q + (r>4) + (r>9);
}
```

Деление на 6 можно реализовать с помощью деления на 3 с последующим *сдвигом вправо* на 1. Однако дополнительную команду можно сэкономить при непосредственном вычислении, используя бинарное приближение

$$4/6 \approx 0.1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010.$$

Соответствующий код показан в листинге 10.10. Версия слева выполняет умножение на приближение $1/6$, а затем выполняет коррекцию путем умножения на $11/64$. Версия справа использует преимущество того факта, что при умножении на приближение $4/6$ оценка частного отличается от точного значения не более чем на 1. Это позволяет получить более простой код коррекции — простое прибавление 1 к `q` при `r` ≥ 6. Код второго оператора `return` предназначен для машин, оснащенных командами предикатов. Функция `divu6b` включает, как показано, 15 команд, в том числе одно *умножение*, или 17 элементарных команд, если умножение на 6 заменено *сдвигами* и *сложениями*.

Листинг 10.10. Беззнаковое деление на 6

```
unsigned divu6a(unsigned n)
{
    unsigned q, r;

    q = (n >> 3) + (n >> 5);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    r = n - q*6;
    return q + (11*r >> 6);
}
```

```
unsigned divu6b(unsigned n)
{
    unsigned q, r;

    q = (n >> 1) + (n >> 3);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    q = q >> 2;
    r = n - q*6;
    return q + ((r + 2) >> 3);
//  return q + (r > 5);
}
```

Похоже, что для бо́льших делителей наилучшим оказывается использование приближения к $1/d$, сдвинутого влево так, чтобы старший бит был равен 1. Обычно при этом оценка частного не отличается от точного значения более чем на 1 (возможно, это верно всегда — этого автор не знает), так что шаг коррекции реализуется весьма эффективно. В листинге 10.11 показан код для деления на 7 и 9, использующий бинарные приближения

$$4/7 \approx 0.1001\ 0010\ 0100\ 1001\ 0010\ 0100\ 1001\ 0010 \quad \text{и}$$
$$8/9 \approx 0.1110\ 0011\ 1000\ 1110\ 0011\ 1000\ 1110\ 0011.$$

При замене умножений на 7 и 9 *сдвигами* и *сложениями* эти функции состоят из 16 и 15 элементарных команд соответственно.

Листинг 10.11. Беззнаковое деление на 7 и 9

```
unsigned divu7(unsigned n)
{
    unsigned q, r;

    q = (n >> 1) + (n >> 4);
    q = q + (q >> 6);
    q = q + (q>>12) + (q>>24);
    q = q >> 2;
    r = n - q*7;
    return q + ((r + 1) >> 3);
//  return q + (r > 6);
}
```

```
unsigned divu9(unsigned n)
{
    unsigned q, r;

    q = n - (n >> 3);
    q = q + (q >> 6);
    q = q + (q>>12) + (q>>24);
    q = q >> 3;
    r = n - q*9;
    return q + ((r + 7) >> 4);
//  return q + (r > 8);
}
```

В листингах 10.12 и 10.13 показан код для деления на 10, 11, 12 и 13. Приведенные функции основаны на следующих бинарных приближениях.

$$8/10 \approx 0.1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100$$
$$8/11 \approx 0.1011\ 1010\ 0010\ 1110\ 1000\ 1011\ 1010\ 0010$$
$$8/12 \approx 0.1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010\ 1010$$
$$8/13 \approx 0.1001\ 1101\ 1000\ 1001\ 1101\ 1000\ 1001\ 1101$$

При замене умножений *сдвигами* и *сложениями* эти функции состоят из 17, 20, 17 и 20 элементарных команд соответственно.

Листинг 10.12. Беззнаковое деление на 10 и 11

```
unsigned divu10(unsigned n)
{
    unsigned q, r;

    q = (n >> 1) + (n >> 2);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    q = q >> 3;
    r = n - q*10;
    return q + ((r + 6) >> 4);
//  return q + (r > 9);
}
```

```
unsigned divu11(unsigned n)
{
    unsigned q, r;

    q = (n >> 1) + (n >> 2) -
        (n >> 5) + (n >> 7);
    q = q + (q >> 10);
    q = q + (q >> 20);
    q = q >> 3;
    r = n - q*11;
    return q + ((r + 5) >> 4);
//  return q + (r > 10);
}
```

Листинг 10.13. Беззнаковое деление на 12 и 13

```
unsigned divu12(unsigned n)
{
    unsigned q, r;

    q = (n >> 1) + (n >> 3);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    q = q >> 3;
    r = n - q*12;
    return q + ((r + 4) >> 4);
//  return q + (r > 11);
}
```

```
unsigned divu13(unsigned n)
{
    unsigned q, r;

    q = (n>>1) + (n>>4);
    q = q + (q>>4) + (q>>5);
    q = q + (q>>12) + (q>>24);
    q = q >> 3;
    r = n - q*13;
    return q + ((r + 3) >> 4);
//  return q + (r > 12);
}
```

Случай деления на 13 весьма поучителен, поскольку показывает, как нужно искать повторяющиеся строки в бинарном представлении обратного к делителю. Первое присваивание устанавливает `q` равным `n*0.1001`. Второе присваивание `q` добавляет `n*0.00001001` и `n*0.000001001`. В этот момент `q` (приближенно) равно `n*0.100111011`. Третье присваивание `q` добавляет повторения этого шаблона. Иногда помогает применение вычитания, как в функции `divu9` ранее. Однако при использовании вычитания вы должны быть особенно осторожны, так как оно может привести к слишком большой оценке частного, так что остаток окажется отрицательным, и рассматриваемый метод будет неработоспособен. Достаточно сложно получить оптимальный код, и у нас нет какого-то универсального метода, который можно было бы вставить в компилятор для обработки любого делителя.

Приведенные выше примеры обеспечивали экономное использование команд благодаря простому повторяющемуся шаблону бинарного представления обратного к делителю, а также тому, что умножение при вычислении остатка `r` выполнялось на небольшую константу, так что его можно было заменить только несколькими командами *сдвига* и *сложения*. Может вызвать интерес вопрос о том, насколько эффективен рассматриваемый метод для бо́льших делителей. Для получения грубого представления по этому во-

просу в листингах 10.14 и 10.15 показан код для деления на (десятичные) 100 и 1000. Соответствующие обратные значения имеют следующий вид.

$$64/100 \approx 0.1010\ 0011\ 1101\ 0111\ 0000\ 1010\ 0011\ 1101$$
$$512/1000 \approx 0.1000\ 0011\ 0001\ 0010\ 0110\ 1110\ 1001\ 0111$$

Если умножения заменяются *сдвигами* и *сложениями*, то эти функции выполняются с помощью 25 и 23 элементарных команд соответственно.

Листинг 10.14. Беззнаковое деление на 100

```
unsigned divu100(unsigned n)
{
    unsigned q, r;

    q = (n >> 1) + (n >> 3) + (n >> 6) - (n >> 10) +
        (n >> 12) + (n >> 13) - (n >> 16);
    q = q + (q >> 20);
    q = q >> 6;
    r = n - q*100;
    return q + ((r + 28) >> 7);
//  return q + (r > 99);
}
```

Листинг 10.15. Беззнаковое деление на 1000

```
unsigned divu1000(unsigned n)
{
    unsigned q, r, t;

    t = (n >> 7) + (n >> 8) + (n >> 12);
    q = (n >> 1) + t + (n >> 15) + (t >> 11) + (t >> 14);
    q = q >> 9;
    r = n - q*1000;
    return q + ((r + 24) >> 10);
//  return q + (r > 999);
}
```

В случае деления на 1000 младшие восемь битов оценки обратного значения, по сути, игнорируются. Код в листинге 10.15 заменяет биты 10010111 битами 01000000, но оценка частного при этом остается в пределах единицы от истинного частного. Таким образом, похоже, что, хотя бинарное представление обратных больших делителей может иметь мало повторений, как минимум некоторые биты можно игнорировать, чтобы снизить количество необходимых *сдвигов* и *сложений* при вычислении оценки частного.

В этом разделе было показано, пусть и не совсем строгим способом, как беззнаковое деление на константу можно свести к последовательности, как правило, состоящей из около 20 элементарных команд. Получение алгоритма, генерирующего такие последовательности, годного для встраивания в компилятор, — задача нетривиальная из-за трех сложностей в получении оптимального кода.

1. Требуется выполнить поиск повторяющегося шаблона в битовой строке оценки обратного к делителю.

2. Иногда могут использоваться отрицательные члены (как в `divu10` и `divu100`), но соответствующий анализ для определения таких ситуаций весьма сложен.
3. Иногда можно игнорировать некоторые из младших битов оценки обратного к делителю (сколько именно?).

Еще одной сложностью для некоторых целевых машин является то, что имеется много вариантов примеров кода, в которых оказывается больше команд, но которые могут выполняться быстрее на машинах с несколькими модулями суммирования или сдвига.

Коды, представленные в листингах 10.7–10.15, протестированы для всех 2^{32} значений делителей.

Знаковое деление

Методы, рассмотренные выше, можно сделать применимыми для знакового деления. Команда *сдвига вправо* при вычислении оценки частного становится командой *знакового сдвига вправо*, которая вычисляет результат деления с округлением к меньшему значению на степени 2. Таким образом, оценка частного оказывается слишком малой (алгебраически), так что остаток оказывается неотрицательным, как и в случае беззнакового деления.

Код наиболее естественным образом вычисляет результат деления с округлением к меньшему значению, так что нам нужна поправка, которая приведет его к обычному результату с отсечением по направлению к 0. Это можно сделать с помощью трех вычислительных команд, добавляя $d-1$ к делимому, если оно отрицательно. Например, если делитель равен 6, то код начинается с

```
n = n + (n >> 31 & 5);
```

(здесь сдвиг является *знаковым сдвигом*).

В остальном код очень похож на код в случае беззнакового деления. Количество требуемых элементарных операций обычно на три больше, чем в соответствующей функции беззнакового деления. Несколько примеров приведены в листингах 10.16–10.22. Все коды прошли исчерпывающее тестирование.

Листинг 10.16. Знаковое деление на 3

```
int divs3(int n)
{
    int q, r;

    n = n + (n>>31 & 2);          // Прибавить 2, если n < 0
    q = (n >> 2) + (n >> 4);      // q = n*0.0101 (приближенно)
    q = q + (q >> 4);             // q = n*0.01010101.
    q = q + (q >> 8);
    q = q + (q >> 16);
    r = n - q*3;                  // 0 <= r <= 14
    return q + (11*r >> 5);       // Возврат q + r/3
//  return q + (5*(r + 1) >> 4);         // Альтернатива 1
//  return q + ((r + 5 + (r << 2)) >> 4);// Альтернатива 2
}
```

Листинг 10.17. Знаковое деление на 5 и 6

```
int divs5(int n)
{
    int q, r;

    n = n + (n>>31 & 4);
    q = (n >> 1) + (n >> 2);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    q = q >> 2;
    r = n - q*5;
    return q + (7*r >> 5);
//  return q + (r>4) + (r>9);
}
```

```
int divs6(int n)
{
    int q, r;

    n = n + (n>>31 & 5);
    q = (n >> 1) + (n >> 3);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    q = q >> 2;
    r = n - q*6;
    return q + ((r + 2) >> 3);
//  return q + (r > 5);
}
```

Листинг 10.18. Знаковое деление на 7 и 9

```
int divs7(int n)
{
    int q, r;

    n = n + (n>>31 & 6);
    q = (n >> 1) + (n >> 4);
    q = q + (q >> 6);
    q = q + (q>>12) + (q>>24);
    q = q >> 2;
    r = n - q*7;
    return q + ((r + 1) >> 3);
//  return q + (r > 6);
}
```

```
int divs9(int n)
{
    int q, r;

    n = n + (n>>31 & 8);
    q = (n >> 1) + (n >> 2) +
        (n >> 3);
    q = q + (q >> 6);
    q = q + (q>>12) + (q>>24);
    q = q >> 3;
    r = n - q*9;
    return q + ((r + 7) >> 4);
//  return q + (r > 8);
}
```

Листинг 10.19. Знаковое деление на 10 и 11

```
int divs10(int n)
{
    int q, r;

    n = n + (n>>31 & 9);
    q = (n >> 1) + (n >> 2);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    q = q >> 3;
    r = n - q*10;
    return q + ((r + 6) >> 4);
//  return q + (r > 9);
}
```

```
int divs11(int n)
{
    int q, r;

    n = n + (n>>31 & 10);
    q = (n >> 1) + (n >> 2) -
    (n >> 5) + (n >> 7);
    q = q + (q >> 10);
    q = q + (q >> 20);
    q = q >> 3;
    r = n - q*11;
    return q + ((r + 5) >> 4);
//  return q + (r > 10);
}
```

Листинг 10.20. Знаковое деление на 12 и 13

```
int divs12(int n)
{
    int q, r;

    n = n + (n>>31 & 11);
    q = (n >> 1) + (n >> 3);
    q = q + (q >> 4);
    q = q + (q >> 8);
    q = q + (q >> 16);
    q = q >> 3;
    r = n - q*12;
    return q + ((r + 4) >> 4);
//  return q + (r > 11);
}
```

```
int divs13(int n)
{
    int q, r;

    n = n + (n>>31 & 12);
    q = (n>>1) + (n>>4);
    q = q + (q>>4) + (q>>5);
    q = q + (q>>12) + (q>>24);
    q = q >> 3;
    r = n - q*13;
    return q + ((r + 3) >> 4);
//  return q + (r > 12);
}
```

Листинг 10.21. Знаковое деление на 100

```
int divs100(int n)
{
    int q, r;

    n = n + (n>>31 & 99);
    q = (n >> 1) + (n >> 3) + (n >> 6) - (n >> 10) +
        (n >> 12) + (n >> 13) - (n >> 16);
    q = q + (q >> 20);
    q = q >> 6;
    r = n - q*100;
    return q + ((r + 28) >> 7);
//  return q + (r > 99);
}
```

Листинг 10.22. Знаковое деление на 1000

```
int divs1000(int n)
{
    int q, r, t;

    n = n + (n>>31 & 999);
    t = (n >> 7) + (n >> 8) + (n >> 12);
    q = (n >> 1) + t + (n >> 15) + (t >> 11) + (t >> 14) +
    (n >> 26) + (t >> 21);
    q = q >> 9;
    r = n - q*1000;
    return q + ((r + 24) >> 10);
//  return q + (r > 999);
}
```

10.19. Получение остатка суммированием цифр

В этом разделе рассматривается задача вычисления остатка от деления на константу без вычисления частного. Методы из этого раздела применимы только для делителей вида $2^k \pm 1$, где k — целое число, не меньшее 2, и в большинстве случаев код прибегает к получению значения из таблицы (команда индексированной *загрузки*) после достаточно короткого вычисления.

Мы будем часто использовать следующее элементарное свойство сравнимости по модулю.

ТЕОРЕМА С. *Если* $a \equiv b \pmod m$ *и* $c \equiv d \pmod m$, *то*

$$a + c \equiv b + d \pmod m \quad \text{и}$$
$$ac \equiv bd \pmod m.$$

Беззнаковый случай проще и рассматривается первым.

Беззнаковый остаток

При делении на 3 многократное умножение тривиального сравнения $1 \equiv 1 \pmod 3$ на сравнение $2 \equiv -1 \pmod 3$ в соответствии с теоремой С дает

$$2^k \equiv \begin{cases} 1 \pmod 3, & k \text{ четно}, \\ -1 \pmod 3, & k \text{ нечетно}. \end{cases}$$

Следовательно, число n, записываемое в бинарном виде как $\ldots b_3 b_2 b_1 b_0$, удовлетворяет соотношению

$$n = \ldots + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2 + b_0 \equiv \ldots - b_3 + b_2 - b_1 + b_0 \pmod 3,$$

которое получается при многократном применении теоремы С. Таким образом, можно чередовать сложение и вычитание битов в бинарном представлении числа, чтобы получить меньшее число, которое имеет тот же остаток при делении на 3. Если сумма отрицательна, можно добавить кратное трем, чтобы сделать ее неотрицательной. Затем этот процесс может повторяться до тех пор, пока результат не окажется в диапазоне от 0 до 2.

Тот же метод работает и при поиске остатка десятичного числа на 11.

Таком образом, если машина оснащена командой подсчета *количества единичных битов*, то функция, которая вычисляет остаток по модулю 3 беззнакового целого числа `n`, может начинаться с

```
n = pop(n & 0x55555555) - pop(n & 0xAAAAAAAA);
```

Это выражение можно упростить, используя следующее удивительное тождество, открытое Паоло Бонцини (Paolo Bonzini) [12].

$$\operatorname{pop}(\boldsymbol{x} \,\&\, \overline{\boldsymbol{m}}) - \operatorname{pop}(\boldsymbol{x} \,\&\, \boldsymbol{m}) = \operatorname{pop}(\boldsymbol{x} \oplus \boldsymbol{m}) - \operatorname{pop}(\boldsymbol{m}) \tag{34}$$

Доказательство

$$\begin{aligned}
&\operatorname{pop}(\boldsymbol{x} \,\&\, \overline{\boldsymbol{m}}) - \operatorname{pop}(\boldsymbol{x} \,\&\, \boldsymbol{m}) = \\
&\quad = \operatorname{pop}(\boldsymbol{x} \,\&\, \overline{\boldsymbol{m}}) - \left(32 - \operatorname{pop}\left(\overline{\boldsymbol{x} \,\&\, \boldsymbol{m}}\right)\right) = && \operatorname{pop}(\boldsymbol{a}) = 32 - \operatorname{pop}(\overline{\boldsymbol{a}}) \\
&\quad = \operatorname{pop}(\boldsymbol{x} \,\&\, \overline{\boldsymbol{m}}) + \operatorname{pop}(\overline{\boldsymbol{x}} \mid \overline{\boldsymbol{m}}) - 32 = && \text{Законы де Моргана} \\
&\quad = \operatorname{pop}(\boldsymbol{x} \,\&\, \overline{\boldsymbol{m}}) + \operatorname{pop}(\overline{\boldsymbol{x}} \,\&\, \boldsymbol{m}) + \operatorname{pop}(\overline{\boldsymbol{m}}) - 32 = && \operatorname{pop}(\boldsymbol{a} \mid \boldsymbol{b}) = \operatorname{pop}(\boldsymbol{a} \,\&\, \overline{\boldsymbol{b}}) + \operatorname{pop}(\boldsymbol{b}) \\
&\quad = \operatorname{pop}\left((\boldsymbol{x} \,\&\, \overline{\boldsymbol{m}}) \mid (\overline{\boldsymbol{x}} \,\&\, \boldsymbol{m})\right) + \operatorname{pop}(\overline{\boldsymbol{m}}) - 32 = && \text{Непересекающиеся множества} \\
&\quad = \operatorname{pop}(\boldsymbol{x} \oplus \boldsymbol{m}) - \operatorname{pop}(\boldsymbol{m})
\end{aligned}$$

Поскольку ссылки на размер слова 32 сокращаются, результат справедлив для любого размера слова. Другой способ доказательства (34) состоит в том, чтобы заметить, что оно справедливо при $\boldsymbol{x} = 0$, и если нулевой бит в $\boldsymbol{x}$ заменяется единичным там, где бит $\boldsymbol{m}$ равен 1, то обе стороны (34) уменьшаются на 1, а если нулевой бит $\boldsymbol{x}$ заменяется единичным там, где бит $\boldsymbol{m}$ равен 0, то обе части (34) увеличиваются на 1.

Применение (34) к приведенной выше строке кода на языке программирования C дает следующее.

```
n = pop(n ^ 0xAAAAAAAA) - 16;
```

Мы хотим применять это преобразование и далее, пока `n` не окажется в диапазоне от 0 до 2, если это возможно. Лучше избежать получения отрицательного значения `n`, чтобы не столкнуться с некорректной трактовкой знакового бита на следующей итерации. Избежать отрицательного значения можно, добавляя достаточно большое кратное 3 к `n`. Код Бонцини, показанный в листинге 10.23, увеличивает константу на 39. Это более чем необходимо, чтобы сделать `n` неотрицательным, но при этом после второго этапа приведения `n` приводится к диапазону от -3 до 2 (вместо диапазона от -3 до 3). Это упрощает код оператора `return`, который прибавляет 3, если `n` отрицательно. Эта функция требует для выполнения 11 команд, с учетом двух команд для загрузки большой константы.

Листинг 10.23. Остаток при беззнаковом делении на 3 с использованием команды *подсчета количества единичных битов*

```
int remu3(unsigned n)
{
    n = pop(n ^ 0xAAAAAAAA) + 23; // Теперь 23 <= n <= 55
    n = pop(n ^ 0x2A) - 3;        // Теперь -3 <= n <= 2
    return n + (((int)n >> 31) & 3);
}
```

В листинге 10.24 показана версия кода, выполняющаяся за 4 команды, плюс простая команда поиска в таблице (например, команда индексированной *загрузки байта*).

Листинг 10.24. Остаток при беззнаковом делении на 3 с использованием команды *подсчета количества единичных битов* и поиска в таблице

```
int remu3(unsigned n)
{
    static char table[33] = {2, 0,1,2, 0,1,2, 0,1,2,
           0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
           0,1,2, 0,1};

    n = pop(n ^ 0xAAAAAAAA);
    return table[n];
}
```

Чтобы избежать команды *подсчета количества единичных битов*, заметим, что поскольку $4 \equiv 1 \pmod 3$, постольку $4^k \equiv 1 \pmod 3$. Бинарное число можно рассматривать как число в системе счисления 4, беря его биты парами и рассматривая биты от 00 до 11

как цифры в системе счисления с основанием 4 в диапазоне от 0 до 3. Пары битов можно суммировать с использованием кода из листинга 5.1 на с. 104, пропуская первую выполнимую строку (при сложении переполнение не возникает). Конечная сумма находится в диапазоне от 0 до 48, и для приведения ее к диапазону от 0 до 2 можно использовать поиск в таблице. Результирующая функция выполняется за 16 элементарных команд, плюс индексированная *загрузка*.

Имеется подобный, но несколько лучший путь. В качестве первого шага `n` можно привести к меньшему значению, находящемуся в том же классе сравнимости по модулю 3 с помощью

```
n = (n >> 16) + (n & 0xFFFF);
```

Этот код разбивает число на две 16-битовые части и суммирует их. Вклад левых 16 бит по модулю 3 не изменяется при сдвиге на 16 разрядов, поскольку сдвинутое число, умноженное на 2^{16}, представляет собой исходное число, а $2^{16} \equiv 1 \pmod 3$. В общем случае $2^k \equiv 1 \pmod 3$, если k четно. Это свойство неоднократно (пять раз) использовано в коде в листинге 10.25. Код состоит из 19 команд. Это количество может быть уменьшено, если прервать суммирование раньше и прибегнуть к поиску в таблице, как показано в листинге 10.26 (9 команд плюс индексированная *загрузка*). Количество команд можно уменьшить до шести (плюс индексированная *загрузка*), если использовать таблицу размером $0x2FE = 766$ байт.

Листинг 10.25. Остаток при беззнаковом делении на 3 с использованием суммирования цифр и поиска в регистре

```
int remu3(unsigned n)
{
    n = (n >> 16) + (n & 0xFFFF);  // Максимум 0x1FFFE
    n = (n >> 8)  + (n & 0x00FF);  // Максимум 0x2FD
    n = (n >> 4)  + (n & 0x000F);  // Максимум 0x3D
    n = (n >> 2)  + (n & 0x0003);  // Максимум 0x11
    n = (n >> 2)  + (n & 0x0003);  // Максимум 0x6
    return (0x0924 >> (n << 1)) & 3;
}
```

Листинг 10.26. Остаток при беззнаковом делении на 3 с использованием суммирования цифр и поиска в памяти

```
int remu3(unsigned n)
{
    static char table[62] = {0,1,2, 0,1,2, 0,1,2, 0,1,2,
        0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
        0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
        0,1,2, 0,1,2, 0,1};
    n = (n >> 16) + (n & 0xFFFF);  // Максимум 0x1FFFE
    n = (n >> 8)  + (n & 0x00FF);  // Максимум 0x2FD
    n = (n >> 4)  + (n & 0x000F);  // Максимум 0x3D
    return table[n];
}
```

Для вычисления остатка от беззнакового деления на 5 код в листинге 10.27 использует соотношения $16^k \equiv 1 \pmod 5$ и $4 \equiv -1 \pmod 5$. Он состоит из 21 элементарной команды в предположении, что умножение на 3 выполняется с помощью *сдвига* и *сложения*.

Листинг 10.27. Остаток при беззнаковом делении на 5 с использованием суммирования цифр

```
int remu5(unsigned n)
{
    n = (n >> 16) + (n & 0xFFFF);          // Максимум 0x1FFFE
    n = (n >> 8)  + (n & 0x00FF);          // Максимум 0x2FD
    n = (n >> 4)  + (n & 0x000F);          // Максимум 0x3D
    n = (n>>4) - ((n>>2) & 3) + (n & 3);   // От -3 до 6
    return (01043210432                    // Восьмеричная константа
            >> 3*(n + 3)) & 7;
}
```

Количество команд можно уменьшить, прибегнув к таблице, подобной использованной в листинге 10.26. Код оказывается идентичным, за исключением таблицы.

```
static char table[62] = {0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
   0,1,2,3,4, 0,1,2,3,4, 0,1};
```

Для поиска остатка от беззнакового деления на 7 код в листинге 10.28 использует соотношение $8^k \equiv 1 \pmod 7$ (9 элементарных команд плюс индексированная *загрузка*).

В качестве последнего примера код из листинга 10.29 вычисляет остаток от беззнакового деления на 9. Он основан на соотношении $8 \equiv -1 \pmod 9$. Как видно из листинга, для вычисления требуются 9 элементарных команд плюс индексированная *загрузка*. Количество элементарных команд может быть уменьшено до шести, если использовать таблицу из 831 элемента.

Листинг 10.28. Остаток при беззнаковом делении на 7 с использованием суммирования цифр

```
int remu7(unsigned n)
{
    static char table[75] = {0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
              0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
              0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
              0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4};

    n = (n >> 15) + (n & 0x7FFF); // Максимум 0x27FFE
    n = (n >> 9) + (n & 0x001FF); // Максимум 0x33D
    n = (n >> 6) + (n & 0x0003F); // Максимум 0x4A
    return table[n];
}
```

Листинг 10.29. Остаток при беззнаковом делении на 9 с использованием суммирования цифр

```
int remu9(unsigned n)
{
    int r;
    static char table[75] = {0,1,2,3,4,5,6,7,8,
          0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
          0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
          0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
          0,1,2,3,4,5,6,7,8, 0,1,2};

    r = (n & 0x7FFF) - (n >> 15); // От FFFE0001 до 7FFF
    r = (r & 0x01FF) - (r >> 9);  // От FFFFFFC1 до 2FF
    r = (r & 0x003F) + (r >> 6);  // От 0 до 4A
    return table[r];
}
```

Знаковый остаток

Метод суммирования цифр можно адаптировать для вычисления остатка, получающегося при знаковом делении. Похоже, лучшие всего это делать путем добавления нескольких шагов для коррекции результата, полученного методом, применяемым для беззнакового деления. Необходимы две коррекции: (1) поправка на разную интерпретацию знакового бита и (2) добавление или вычитание кратного делителю d для получения результата в диапазоне от 0 до $-(d-1)$.

Для деления на 3 код получения остатка в беззнаковом случае рассматривает знаковый бит делимого ***n*** как вклад 2 в остаток (поскольку $2^{31} \bmod 3 = 2$). В случае знакового деления знаковый бит дает вклад, равный 1 (поскольку $(-2^{31}) \bmod 3 = 1$). Следовательно, можно использовать код для беззнакового остатка и исправить полученный результат, вычитая 1. Далее, результат должен находиться в диапазоне от 0 до -2, т.е. результат вычисления беззнакового остатка должен быть отображен следующим образом.

$$(0,1,2) \Rightarrow (-1,0,1) \Rightarrow (-1,0,-2)$$

Достаточно эффективно этого можно добиться путем вычитания 1 из беззнакового остатка, если он равен 0 или 1, и 4, если остаток равен 2 (когда делимое отрицательно). Код не должен менять значение делимого `n`, поскольку оно требуется на последнем шаге.

Эту процедуру легко применить к любой из функций для вычисления остатка от беззнакового деления на 3. Например, применение к коду из листинга 10.26 на с. 291 дает функцию, показанную в листинге 10.30. Она состоит из 13 элементарных команд, плюс индексированная *загрузка*. Количество команд может быть уменьшено ценой увеличения таблицы.

Листинг 10.30. Остаток при знаковом делении на 3 с использованием суммирования цифр

```
int rems3(int n)
{
    unsigned r;
```

```
    static char table[62] = {0,1,2, 0,1,2, 0,1,2, 0,1,2,
        0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
        0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2, 0,1,2,
        0,1,2, 0,1,2, 0,1};

    r = n;
    r = (r >> 16) + (r & 0xFFFF); // Максимум 0x1FFFE
    r = (r >> 8) + (r & 0x00FF);  // Максимум 0x2FD
    r = (r >> 4) + (r & 0x000F);  // Максимум 0x3D
    r = table[r];
    return r - (((unsigned)n >> 31) << (r & 2));
}
```

В листингах 10.31–10.33 показан аналогичный код для вычисления остатка от знакового деления на 5, 7 и 9. Все эти функции состоят из 15 элементарных операций, плюс индексированная *загрузка*. Они используют знаковые сдвиги вправо, а последняя коррекция состоит в вычитании модуля, если делимое отрицательно, а остаток ненулевой. Количество команд может быть уменьшено ценой увеличения таблицы.

Листинг 10.31. Остаток при знаковом делении на 5 с использованием суммирования цифр

```
int rems5(int n)
{
    int r;
    static char table[62] = {2,3,4, 0,1,2,3,4, 0,1,2,3,4,
              0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
              0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4, 0,1,2,3,4,
              0,1,2,3,4, 0,1,2,3};

    r = (n >> 16) + (n & 0xFFFF); // От FFFF8000 до 17FFE
    r = (r >> 8) + (r & 0x00FF);  // От FFFFFF80 до 27D
    r = (r >> 4) + (r & 0x000F);  // От -8 до 53 (десятичных)
    r = table[r + 8];
    return r - (((int)(n & -r) >> 31) & 5);
}
```

Листинг 10.32. Остаток при знаковом делении на 7 с использованием суммирования цифр

```
int rems7(int n)
{
    int r;
    static char table[75] =   {5,6, 0,1,2,3,4,5,6,
      0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
      0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
      0,1,2,3,4,5,6, 0,1,2,3,4,5,6, 0,1,2,3,4,5,6,
      0,1,2};

    r = (n >> 15) + (n & 0x7FFF); // От FFFF0000 до 17FFE
    r = (r >> 9) + (r & 0x001FF); // От FFFFFF80 до 2BD
    r = (r >> 6) + (r & 0x0003F); // От -2 до 72 (десятичных)
    r = table[r + 2];
    return r - (((int)(n & -r) >> 31) & 7);
}
```

Листинг 10.33. Остаток при знаковом делении на 9 с использованием суммирования цифр

```
int rems9(int n)
{
    int r;
    static char table[75] = {7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0,1,2,3,4,5,6,7,8,
              0,1,2,3,4,5,6,7,8, 0};

    r = (n & 0x7FFF) - (n >> 15); // От FFFF7001 до 17FFF
    r = (r & 0x01FF) - (r >> 9);  // От FFFFFF41 до 0x27F
    r = (r & 0x003F) + (r >> 6);  // От -2 до 72 (десятичных)
    r = table[r + 2];
    return r - (((int)(n & -r) >> 31) & 9);
}
```

10.20. Получение остатка путем умножения и сдвига вправо

Описанный в этом разделе метод применим, в принципе, ко всем целым делителям, большим 2, но с точки зрения практического применения он годится только для малых делителей и делителей вида $2^k - 1$. Как и в предыдущем разделе, в большинстве случаев код выполняет поиск в таблице после достаточно короткого вычисления.

Беззнаковый остаток

В этом разделе используется математическое (а не компьютерно-алгебраическое) обозначение $a \bmod b$, где a и b — целые числа и $b > 0$, и которое используется для указания числа $0 \le x < b$, удовлетворяющего условию $x \equiv a \pmod b$.

Чтобы вычислить $x \bmod 3$, заметим, что

$$n \bmod 3 = \left\lfloor \frac{4}{3} n \right\rfloor \bmod 4 . \tag{35}$$

Доказательство. Пусть $n = 3k + \delta$, где δ и k — целые числа и $0 \le \delta \le 2$. Тогда

$$\left\lfloor \frac{4}{3}(3k + \delta) \right\rfloor \bmod 4 = \left\lfloor 4k + \frac{4\delta}{3} \right\rfloor \bmod 4 = \left\lfloor \frac{4\delta}{3} \right\rfloor \bmod 4 .$$

Ясно, что значение последнего выражения равно 0, 1 или 2 при $\delta = 0$, 1 или 2 соответственно. Это позволяет заменить задачу вычисления остатка по модулю 3 задачей вычисления остатка по модулю 4, что, конечно, на бинарном компьютере выполнить проще.

Соотношения наподобие (35) выполняются не для всех модулей, но аналогичные соотношения имеются для модулей вида $2^k - 1$ при целом k, большем 1. Например, легко показать, что

$$n \bmod 7 = \left\lfloor \frac{8}{7} n \right\rfloor \bmod 8 .$$

Для чисел вида, отличного от $2^k - 1$, таких простых соотношений нет, но имеется определенное свойство единственности, которое может быть использовано для вычисления

остатка для других делителей. Например, если делитель представляет собой десятичное число 10, рассмотрим выражение

$$\left\lfloor \frac{16}{10} n \right\rfloor \bmod 16 . \tag{36}$$

Пусть $n = 10k + \delta$, где $0 \le \delta \le 9$. Тогда

$$\left\lfloor \frac{16}{10} n \right\rfloor \bmod 16 = \left\lfloor \frac{16}{10} (10k + \delta) \right\rfloor \bmod 16 = \left\lfloor \frac{16\delta}{10} \right\rfloor \bmod 16 .$$

Для $\delta = 0$, 1, 2, 3, 4, 5, 6, 7, 8 и 9 последнее выражение принимает значения 0, 1, 3, 4, 6, 8, 9, 11, 12 и 14 соответственно. Все числа в последнем множестве различны. Следовательно, если имеется достаточно простой способ вычисления (36), можно преобразовать 0 в 0, 1 в 1, 3 в 2, 4 в 3 и так далее, получая таким образом остатки от деления на 10. В общем случае этот метод требует применения таблицы преобразования с размером, равным ближайшей степени 2, большей делителя, так что этот метод практичен только для достаточно малых делителей (и для делителей вида $2^k - 1$, для которых поиск в таблице не требуется).

Приведенный далее код получен при скромном применении описанной выше теории ценой огромного количества проб и ошибок.

Рассмотрим остаток от беззнакового деления на 3. Следуя (35), мы хотим вычислить два крайних справа бита целой части $4n/3$. Это можно сделать приближенно, выполняя умножение на $\lfloor 2^{32}/3 \rfloor$, а затем деление на 2^{30} с помощью команды *сдвига вправо*. После того как умножение на $\lfloor 2^{32}/3 \rfloor$ выполнено (с помощью команды *умножения*, которая дает младшие 32 бита результата), старшие биты оказываются потерянными. Но это не имеет значения и в действительности даже полезно, поскольку нас интересует результат по модулю 4. Следовательно, так как $\lfloor 2^{32}/3 \rfloor = 0\text{x}55555555$, возможный план заключается в вычислении

$$\boldsymbol{r} \leftarrow (\mathbf{0x55555555} * \boldsymbol{n}) \overset{u}{\gg} 30 .$$

Эксперимент показывает, что этот способ работает для $\boldsymbol{n}$ из диапазона от 0 до $2^{30} + 2$. Я бы сказал, что он почти работает, — если $\boldsymbol{n}$ не равно 0 и кратно 3, в результате мы получаем 3. Следовательно, необходим шаг, преобразующий $(0,1,2,3)$ соответственно в $(0,1,2,0)$.

Чтобы расширить диапазон применимости, умножение должно выполняться более точно. Двух битов точности оказывается достаточно (т.е. умножения на число **0x55555555.4**). Приведенное далее вычисление с последующим шагом преобразования работает для всех $\boldsymbol{n}$, представимых с помощью беззнакового 32-битового числа.

$$\boldsymbol{r} \leftarrow \left(\mathbf{0x55555555} * \boldsymbol{n} + \left(\boldsymbol{n} \overset{u}{\gg} 2 \right) \right) \overset{u}{\gg} 30$$

Конечно, можно дать формальное доказательство этого факта, но соответствующие выкладки очень длинны, и в них легко ошибиться.

Шаг преобразования результата можно выполнить с помощью трех или четырех команд на большинстве машин, но есть способ избежать его ценой двух команд. Приведенное выше выражение для вычисления ***r*** дает заниженную оценку. Если ее немного повысить, результат всегда оказывается равным 0, 1 или 2. Это приводит нас к функции на языке программирования C, показанной в листинге 10.34 (восемь команд, включая *умножение*).

Листинг 10.34. Беззнаковый остаток по модулю 3, метод умножения

```
int remu3(unsigned n)
{
    return (0x55555555*n + (n >> 1) - (n >> 3)) >> 30;
}
```

Умножение можно заменить *сдвигами* и *сложениями*, так что получится код, приведенный в листинге 10.35 и использующий только 13 таких команд.

Листинг 10.35. Беззнаковый остаток по модулю 3, метод умножения на основе сдвигов и сложений

```
int remu3(unsigned n)
{
    unsigned r;

    r = n + (n << 2);
    r = r + (r << 4);
    r = r + (r << 8);
    r = r + (r << 16);
    r = r + (n >> 1);
    r = r - (n >> 3);
    return r >> 30;
}
```

Остаток от беззнакового деления на 5 можно вычислить очень схожим с вычисление остатка от деления на 3 способом. Пусть $n = 5k + r$, где $0 \le r \le 4$. Тогда $(8/5)n \bmod 8 = (8/5)(5k + r) \bmod 8 = (8/5)r \bmod 8$. При $r = 0$, 1, 2, 3 и 4 это дает нам значения 0, 1, 3, 4 и 6 соответственно. Поскольку $\lfloor 2^{32}/5 \rfloor = 0x33333333$, это приводит к функции, показанной в листинге 10.36 (11 команд, включая *умножение*). Последний шаг (код оператора `return`) отображает $(0,1,3,4,6,7)$ на $(0,1,2,3,4,0)$ соответственно, используя метод работы с регистром вместо индексированной *загрузки* из памяти. Отображение, кроме того, 2 на 2 и 5 на 4 снижает требуемую точность при умножении на $2^{32}/5$, что позволяет обойтись одним лишь членом `n>>3` для приближения отсутствующей части множителя (шестнадцатеричное значение 0.333…). Если опустить корректирующий член `n>>3`, код останется работоспособным для значений `n` от 0 до 0x60000004.

Листинг 10.36. Беззнаковый остаток по модулю 5, метод умножения

```
int remu5(unsigned n)
{
    n = (0x33333333*n + (n >> 3)) >> 29;
    return (0x04432210 >> (n << 2)) & 7;
}
```

Код для вычисления остатка от деления на 7 похож на приведенный выше, но в нем проще шаг отображения — требуется только преобразование 7 в 0. Один из способов решения этой задачи показан в листинге 10.37 (11 команд, включая *умножение*). Если опустить корректирующий член `n>>4`, код останется работоспособным для значений `n` до 0x40000006. При удалении обоих корректирующих членов код работает для значений `n` до 0x08000006.

Листинг 10.37. Беззнаковый остаток по модулю 7, метод умножения

```
int remu7(unsigned n)
{
    n = (0x24924924*n + (n >> 1) + (n >> 4)) >> 29;
    return n & ((int)(n - 7) >> 31);
}
```

Код для вычисления остатка от деления на 9 приведен в листинге 10.38. Он состоит из шести команд, включая *умножение*, плюс индексированная *загрузка*. Если опустить корректирующий член `n>>1`, а множитель заменить на 0x1C71C71D, то функция работает для значений `n` до 0x1999999E.

Листинг 10.38. Беззнаковый остаток по модулю 9, метод умножения

```
int remu9(unsigned n)
{
    static char table[16] = {0, 1, 1, 2, 2, 3, 3, 4,
                             5, 5, 6, 6, 7, 7, 8, 8};

    n = (0x1C71C71C*n + (n >> 1)) >> 28;
    return table[n];
}
```

В листинге 10.39 показан способ вычисления беззнакового остатка по модулю 10. Он состоит из восьми команд, включая *умножение*, плюс индексированная *загрузка*. Если опустить корректирующий член `n>>3`, то код работает для значений `n` до 0x40000004. При удалении обоих корректирующих членов код работает для значений `n` до 0x0AAAAAAD.

Листинг 10.39. Беззнаковый остаток по модулю 10, метод умножения

```
int remu10(unsigned n)
{
    static char table[16] = {0, 1, 2, 2, 3, 3, 4, 5,
                             5, 6, 7, 7, 8, 8, 9, 0};

    n = (0x19999999*n + (n >> 1) + (n >> 3)) >> 28;
    return table[n];
}
```

В качестве последнего примера рассмотрим вычисление остатка по модулю 63. Эта функция используется в программе вычисления степени заполнения на с. 106. Джо Кин (Joe Keane) [64] предложил совершенно удивительный код, показанный в листинге 10.40. Он состоит из 12 элементарных команд из базового набора RISC.

Листинг 10.40. Беззнаковый остаток по модулю 63, метод Кина

```
int remu63(unsigned n)
{
    unsigned t;

    t = (((n >> 12) + n) >> 10) + (n << 2);
    t = ((t >> 6) + t + 3) & 0xFF;
    return (t - (t >> 6)) >> 2;
}
```

Метод умножения и сдвига вправо приводит к коду, показанному в листинге 10.41. Здесь выполняется 11 элементарных команд из базового набора RISC, одна из которых — *умножение*. Этот метод не так быстр, как метод Кина, если только машина не выполняет умножение очень быстро, а загрузку константы 0x04104104 можно вынести из цикла.

Листинг 10.41. Беззнаковый остаток по модулю 63, метод умножения

```
int remu63(unsigned n)
{
    n = (0x04104104*n + (n >> 4) + (n >> 10)) >> 26;
    return n & ((n - 63) >> 6); // Замена 63 нулевым
}                               // значением
```

На некоторых машинах можно ускорить код, заменив умножение сдвигами и сложениями следующим образом (15 элементарных команд для всей функции).

```
r = (n << 2) + (n << 8); // r = 0x104*n
r = r + (r << 12);       // r = 0x104104*n
r = r + (n << 26);       // r = 0x04104104*n
```

Знаковый остаток

Как и в случае метода суммирования цифр, метод умножения и сдвига вправо можно адаптировать для вычисления остатка от знакового деления. Похоже, что и в этом случае нет более подходящего пути, чем добавление нескольких корректирующих шагов к методу, применяющемуся для беззнакового деления. Например, в листинге 10.42 показан код, полученный из листинга 10.34 на с. 297 (12 команд, включая *умножение*).

Листинг 10.42. Знаковый остаток по модулю 3, метод умножения

```
int rems3(int n)
{
    unsigned r;

    r = n;
    r = (0x55555555*r + (r >> 1) - (r >> 3)) >> 30;
    return r - (((unsigned)n >> 31) << (r & 2));
}
```

Некоторые правдоподобные способы вычисления остатка от знакового деления на 5, 7, 9 и 10 показаны в листингах 10.43–10.46. Код для делителя 7 использует несколько большее количество команд (всего 19, включая *умножение*); может оказаться предпочтительнее использовать таблицу, подобную показанной для случаев 5, 9 и 10. В последних случаях использованы удвоенные в размере таблицы для беззнакового деления; при вычислении индекса для загрузки из таблицы используется знаковый бит. Элементы таблицы, показанные как `u`, в действительности не используются.

Листинг 10.43. Знаковый остаток по модулю 5, метод умножения

```
int rems5(int n)
{
    unsigned r;
    static signed char table[16] = {0, 1, 2, 2, 3, u, 4, 0,
                                    u, 0,-4, u,-3,-2,-2,-1};

    r = n;
    r = ((0x33333333*r) + (r >> 3)) >> 29;
    return table[r + (((unsigned)n >> 31) << 3)];
}
```

Листинг 10.44. Знаковый остаток по модулю 7, метод умножения

```
int rems7(int n)
{
    unsigned r;

    r = n - (((unsigned)n >> 31) << 2);   // Учет знака
    r = ((0x24924924*r) + (r >> 1) + (r >> 4)) >> 29;
    r = r & ((int)(r - 7) >> 31);         // Замена 7 на 0
    return r - (((int)(n&-r) >> 31) & 7); // Случай n<0
}
```

Листинг 10.45. Знаковый остаток по модулю 9, метод умножения

```
int rems9(int n)
{
    unsigned r;
    static signed char table[32] = {0, 1, 1, 2, u, 3, u, 4,
                                    5, 5, 6, 6, 7, u, 8, u,
                                   -4, u,-3, u,-2,-1,-1, 0,
                                    u,-8, u,-7,-6,-6,-5,-5};

    r = n;
    r = (0x1C71C71C*r + (r >> 1)) >> 28;
    return table[r + (((unsigned)n >> 31) << 4)];
}
```

Листинг 10.46. Знаковый остаток по модулю 10, метод умножения

```
int rems10(int n)
{
    unsigned r;
    static signed char table[32] = {0, 1, u, 2, 3, u, 4, 5,
                                    5, 6, u, 7, 8, u, 9, u,
                                   -6,-5, u,-4,-3,-3,-2, u,
                                   -1, 0, u,-9, u,-8,-7, u};
```

```
    r = n;
    r = (0x19999999*r + (r >> 1) + (r >> 3)) >> 28;
    return table[r + (((unsigned)n >> 31) << 4)];
}
```

10.21. Преобразование в точное деление

Поскольку остаток оказывается возможным вычислить без вычисления частного, можно попробовать вычислять частное $q = \lfloor n/d \rfloor$, сначала вычисляя остаток, вычитая его из делимого *n*, а затем деля разность на делитель *d*. Последнее деление является точным, так что его можно выполнять путем умножения на мультипликативное обратное к *d* (см. раздел 10.16,"Точное деление на константу", на с. 266). Этот метод оказывается особенно привлекательным, когда требуется найти и частное, и остаток.

Давайте попробуем применить наши рассуждения к случаю беззнакового деления на 3. Вычисление остатка методом умножения (листинг 10.34 на с. 297) приводит к функции, показанной в листинге 10.47.

Листинг 10.47. Беззнаковые остаток и частное для делителя 3 с использованием точного деления

```
unsigned divu3(unsigned n)
{
    unsigned r;

    r = (0x55555555*n + (n >> 1) - (n >> 3)) >> 30;
    return (n - r)*0xAAAAAAAB;
}
```

Эта функция включает 11 команд, в том числе два умножения на большие числа (константа 0x55555555 получается из константы 0xAAAAAAAB сдвигом вправо на одну позицию). Более прямолинейный метод вычисления частного `q` с использованием, например, кода из листинга 10.8 на с. 281 требует 14 команд, включая два умножения на небольшие числа, или 17 команд при замене умножений *сдвигами* и *сложениями*. Если требуется и значение остатка, которое вычисляется как `r=n-q*3`, прямолинейный метод требует выполнения 16 команд, включая три умножения на небольшие числа или 20 команд при замене умножений *сдвигами* и *сложениями*.

Код из листинга 10.47 при замене умножений *сдвигами* и *сложениями* привлекательнее не становится: результат состоит из 24 элементарных команд. Таким образом, метод точного деления может быть применим, в первую очередь, на машинах, на которых нет команды вычисления *старшего слова умножения*, но есть быстрое *умножение* по модулю 2^{32} и медленное *деление*, в особенности если они легко работают с большими константами.

Для знакового деления на 3 метод точного деления может быть закодирован так, как показано в листинге 10.48. Этот код состоит из 15 команд, включая два умножения на большие константы.

Листинг 10.48. Знаковые остаток и частное для делителя 3 с использованием точного деления

```
int divs3(int n)
{
    unsigned r;

    r = n;
    r = (0x55555555*r + (r >> 1) - (r >> 3)) >> 30;
    r = r - (((unsigned)n >> 31) << (r & 2));
    return (n - r)*0xAAAAAAAB;
}
```

В качестве последнего примера в листинге 10.49 приведен код для вычисления частного и остатка от беззнакового деления на 10. В нем 12 команд, включая два умножения на большие константы, плюс команда индексированной *загрузки*.

Листинг 10.49. Беззнаковые остаток и частное для делителя 10 с использованием точного деления

```
unsigned divu10(unsigned n)
{
    unsigned r;
    static char table[16] = {0, 1, 2, 2, 3, 3, 4, 5,
                             5, 6, 7, 7, 8, 8, 9, 0};

    r = (0x19999999*n + (n >> 1) + (n >> 3)) >> 28;
    r = table[r];
    return ((n - r) >> 1)*0xCCCCCCCD;
}
```

10.22. Проверка времени выполнения

На многих машинах имеется команда *умножения* $32\times32\Rightarrow64$, так что можно ожидать, что при делении на константу, такую, как 3, код, показанный на с. 253, окажется самым быстрым. Если такой команды *умножения* на машине нет, но есть быстрая команда *умножения* $32\times32\Rightarrow32$, то лучшим для машин с быстрым умножением и медленным делением может оказаться метод точного деления. Чтобы проверить это утверждение, для сравнения четырех методов деления на 3 была написана программа на ассемблере, результаты работы которой подытожены в табл. 10.4. Использовалась машина Pentium III 667 МГц (ок. 2000 г.); следует ожидать сходных результатов и на других машинах.

Таблица 10.4. Беззнаковое деление на 3 на Pentium III

Метод деления	Количество тактов
Использование машинной команды деления (`divl`)	41.08
Использование *умножения* 32×32⇒64 (код на с. 253)	4.28
Все элементарные команды (листинг 10.8 на с. 281)	14.10
Приведение к точному делению (листинг 10.47 на с. 301)	6.68

Первая строка дает время в тактах для двух команд (`xorl` для сброса левой половины 64-битового регистра источника и для команды `divl`), которые требуют для выполнения 40 тактов. Вторая строка также дает время выполнения двух команд: *умножения* и *сдвига вправо* на 1 бит (`mull` и `shrl`). Третья строка указывает время выполнения 21 элементарной команды. Это код из листинга 10.8 на с. 281, использующий вторую альтернативу, и с умножением на 3, осуществляемым одной командой (`leal`). Требуются также несколько команд *перемещения* в силу того, что машина (в основном) двухадресная. Последняя строка содержит время выполнения последовательности из 10 команд, включая два умножения (`imull`). Эти две команды `imull` используют для больших констант четырехбайтовые непосредственные поля. (Использована команда знакового *умножения* `imull`, а не ее беззнаковый двойник `mull`, так как они дают один и тот же результат в младших 32 битах и при этом команда `imull` имеет большее количество режимов адресации).

Метод точного деления даже предпочтительнее второго и третьего методов, если требуются как частное, так и остаток, поскольку в таком случае к этим методам следует добавить код вычисления $\boldsymbol{r} \leftarrow \boldsymbol{n} - \boldsymbol{q} * 3$ (команда `divl` дает как частное, так и остаток).

10.23. Аппаратная схема для деления на 3

Имеется простая схема деления на 3 — примерно того же уровня сложности, что и сумматор. Она может быть построена методом, очень схожим с тем, который применяется для построения n-битового сумматора из n 1-битовых “полных сумматоров”. Однако в делителе сигналы идут от старшего бита к младшему.

Рассмотрим деление на 3 школьным методом “в столбик”, но в бинарной системе счисления. Для получения каждого бита частного мы делим на 3 очередной бит, которому предшествует остаток 0, 1 или 2 от предыдущего шага. Соответствующая логика показана в табл. 10.5. Здесь остаток представлен двумя битами, r_i и s_i, где r_i представляет собой старший бит. Остаток никогда не равен 3, так что последние две строки таблицы представляют собой случаи “не имеет значения”.

Таблица 10.5. Логика деления на 3

r_{i+1}	s_{i+1}	x_i	y_i	r_i	s_i
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	–	–	–
1	1	1	–	–	–

Схема для 32-битового деления на 3 показана на рис. 10.1. Частное представляет собой слово, состоящее из битов с y_{31} по y_0, а остаток равен $2r_0 + s_0$.

Еще один способ аппаратной реализации операции деления на 3 — использование умножителя для того, чтобы умножить делимое на обратное к 3 (бинарное 0.010101…) с соответствующим округлением и масштабированием. Эта методика показана на с. 233 и 253.

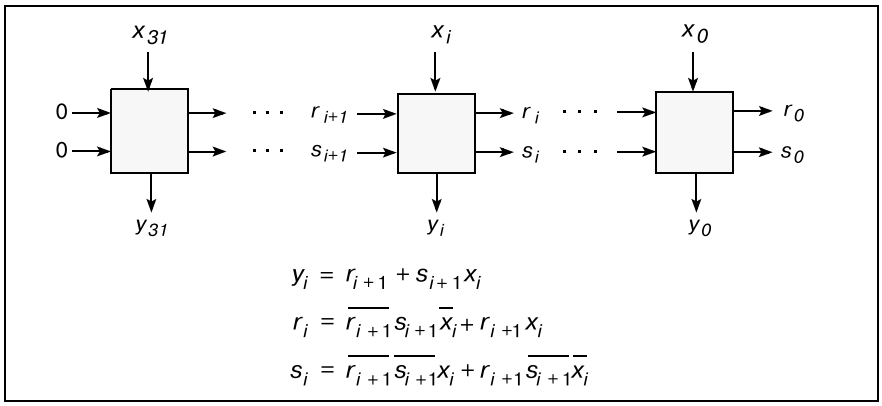

Рис. 10.1. Логическая схема для деления на 3

Упражнения

1. Покажите, что в случае беззнакового деления на четное число команды `shrxi` (или эквивалентного кода) можно избежать путем (а) сброса младшего бита делимого (операцией *и*) [14] или (б) деления делимого на 2 (команда *сдвига вправо на 1 бит*) с последующим делением на половину делителя.
2. Запишите на языке программирования Python код функции, аналогичной функции из листинга 10.4 на с. 266, но для вычисления магического числа для знакового деления. Рассматривайте только положительные делители.
3. Покажите, как использовать метод Ньютона для вычисления мультипликативного обратного целого числа *d* по модулю 81. Покажите ход вычислений для $d = 146$.