Содержание

Список иллюстраций

Список таблиц

«Каждой твари по паре !...»

Из инструкции по посадке в Ноев ковчег

1 Введение

Двоичная арифметика не особенно отличается от той арифметики, которую учат в начальных классах средней школы. В чем-то она даже проще – в школе надо было запоминать десять цифр и восемь таблиц умножения, в двоичной арифметике цифр всего две, а базовых таблиц всего три. В тоже время двоичная арифметика имеет свои особенности, так как является основой функционирования всех цифровых вычислительных устройств, и поэтому используется во всех микропроцессорах.

Рассмотрим, почему именно двоичная система счисления является основой всех вычислительных устройств, и каким образом она используется. И как положено в инженерной деятельности, вначале надо определиться с терминологией, которая будет использоваться в последующем тексте.

Первым определением является понятие **системы счисления**, которая является некоторой совокупностью способов и средств записи чисел. К ним обычно относятся алфавит представления чисел и способы записи определенных значений чисел с использованием этого алфавита.

В зависимости от способа записи системы счисления бывают **позиционными** и **непозиционными**. В непозиционных системах счисления числовое значение символа не зависит от его места в записи числа.

К числу таких систем относится античная римская система счисления, использующая определенный алфавит для базовых значений, из которых затем составляется число (I=1; V=5; X=10; L=50; C=100; D=500; M=1000.). Для того, чтобы составить число 24, нам нужно 2 знака числа 10, один знак числа пять и один знак единицы: XXIV. Обратим внимание на использование символа единицы с отрицательным значением в случае ее предшествования знаку алфавита с большим значением. Для числа 1024 запись соотвественно будет MXXIV. Численный эквивалент каждого символа не зависит от позиции символа в записи, например число 999 может быть записано как IM.

В позиционных системах счисления численное значение символа непосредственно зависит от позиции символа в записи числа. К числу таких систем относится традиционная десятичная система, которой мы все пользуемся в повседневной жизни. Алфавитом такой системы

являются совокупность десяти символов (цифр) от 0 до 9, причем в записи числа крайние левые символы имеют наибольшее численное значение.

Причем общее число символов в алфавите позиционной системы счисления является основанием системы счисления (radix)– максимальным значением, выраженным одним символом. **Таблица 1** иллюстрирует алфавиты и основания двоичной, восьмеричной и шестнадцатиречной систем счислений.

Таблица 1. Алфавиты и основания основных систем счисления.

Система счисления	**Алфавит**	**Основание radix**
Двоичная	0, 1	2
Восьмеричная	0, 1, 2, 3, 4, 5, 6, 7	8
Десятичная	0,1, 2, 3, 4, 5, 6, 7, 8, 9	10
Шестнадцатеричная	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (A=10, B=11, C=12, D=13, E=14, F=15)	16

Теперь рассмотрим особенности позиции символа в записи числа, что важно для позиционной системы в отличии от непозиционной древнеримской. Номер позиции в числе называется **разрядом** и нумеруются справа налево, начиная с нуля. При этом с каждой позицией ассоциируется **вес разряда** – число, равное основанию системы счисления, возведенное в степень номера разряда.

В цифровых вычислительных устройствах используются только позиционные системы счисления – двоичная (Binary) в качестве основной, которая реализована в аппаратуре, а также десятичная (Decimal) и шестнадцатеричная (Hexadecimal) в качестве вспомогательных, ранее также использовалась и восьмеричная (Octal) система счисления. Таблица 2 иллюстрирует веса разрядов для каждой системы счисления, при этом для удобства восприятия все значения выражены через алфавит десятичной системы счисления.

Таблица 2. Разряды и их вес в основных системах счисления.

	Разряд N	**. . . . . .**	**Разряд 2**	**Разряд 1**	**Разряд 0**
Двоичная (BIN)	2^N		2^2	2^1	2^0
Восьмеричная (OCT)	8^N		8^2	8^1	8^0
Десятичная (DEC)	10^N		10^2	10^1	10^0

Шестнадцатеричная (HEX)	16^N		16^2	16^1	16^0

Рассмотрим простые примеры представления чисел в каждой системе счисления с иллюстрацией значение каждого числа в десятичной системе, рассчитанное согласно веса каждого разряда:

$$1010_2 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 10_{10}$$

$$56_8 = 5 * 8^1 + 6 * 8^0 = 46_{10}$$

$$125_{10} = 1 * 10^2 + 2 * 10^1 + 5 * 10^0 = 125_{10}$$

$$56_{16} = 5 * 16^1 + 6 * 16^0 = 86_{10}$$

Теперь попробуем обосновать, почему именно двоичная система счисления стала основой работы всех цифровых устройств. В первую очередь, это связано с представлением значения на уровне электрического сигнала и простотой схем обработки таких сигналов. В случае двоичной системы мы имеем только два уровня сигнала для представления алфавита системы счисления (высокий уровень соотвествует единице, низкий уровень – нулю, либо наоборот в случае инверсной логики). Схемы, обрабатывающие такие сигналы очень просты и помехоустойчивы, так как должны различать только два уровня. Если уровней будет восемь или десять, то схемы значительно усложняются и возникает проблема правильной идентификации уровней электрических сигналов, которые могут изменять свои значения из-за нестабильности работы элементов схем.

Шестнадцатеричная и восьмеричная системы не использовались в повседневной жизни до появления и массового распространения компьютеров. Их применение, прежде всего связано возможностью компактного представления двоичных чисел и операций над ними, которую обеспечивают использование этих систем счисления. Для пользователя гораздо удобнее читать и анализировать сжатое представление двоичных чисел в восьмеричной или шестнадцатеричной форме по сравнению с многочисленными нулями и единицами в оригинальном представлении. В 50-е и 60-годы широко использовалась восьмеричная система для представления двоичных данных, в дальнейшем с распространением байта (с форматом в 8 бит) как базового элемента данных, шестнадцатеричное представление получило более широкое распространение, так как один байт мог кодироваться двумя шестнадцатеричными цифрами. Таблица 3 иллюстрирует сравнительное представление двоичных чисел в различных системах счисления.

Таблица 3. Сравнительное представление двоичных чисел в восьмеричной, десятичной и шестнадцатеричной системах счисления.

Двоичное представление (Binary) $_2$	Восьмеричное представле-ние (Octal) $_8$	Десятичное представление (Decimal) $_{10}$	Шестнадцатеричное представление (Hexadecimal) $_{16}$
0 000	0	0	0
0 001	1	1	1
0 010	2	2	2
0 011	3	3	3
0 100	4	4	4
0 101	5	5	5
0 110	6	6	6
0 111	7	7	7
1 000	10	8	8
1 001	11	9	9
1 010	12	10	A
1 011	13	11	B
1 100	14	12	C
1 101	15	13	D
1 110	16	14	E
1 111	17	15	F

Как видно из вышеприведенной таблицы, восьмеричная система позволяет кодировать одним символом все возможные 8 комбинаций только трех разрядов, в то время десятичная система практически не имеет преимуществ, обеспечивая кодирование одним символом только 10 комбинаций из 16 возможных для 4 разрядов. Шестнадцатеричная система покрывает все возможные комбинации из 4 разрядов, используя только один символ из афавита в 16 знаков. Для кодирования байта необходимо только два символа в отличии от трех в восьмеричной или десятичной системе.

В качестве полезного совета рекомендуем запомнить эту таблицу кодирования аналогично таблице умножения в начальной школе, автоматизм в преобразовании двоичного представления в ту или иную форму может сильно сэкономить время в профессиональной деятельности.

2 Двоичные числа со знаком и без знака

Для представления чисел в двоичной арифметике используется позиционная система счисления по основанию 2, т.е. для записи чисел используются только две цифры: 0 и 1. Положение разряда определяет его вес или степень для возведения основания, значение разряда – множитель для возведенного в степень основания, согласно примера, приведенного в предыдущем разделе:

$$1010_2 = 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 10_{10}.$$

Пристрастие к степеням двойки в кругах, связанных с вычислительной техникой, не останавливается на основании системы счисления: в качестве атомарной единицы адресации данных в памяти используют восьмиразрядный байт, а для представления целых чисел в современных процессорах при расчетах используется четыре или восемь байт, что соответствует 32 или 64 двоичным разрядам-битам. Используя, например, 64 бита для представления натуральных чисел, процессор может производить вычисления над числами в диапазоне от 0 до 2^{64}-1 или 18 квинтиллионов с хвостиком. Иллюстрация возможного кодирования числа в байте представлена в Таблица 4.

Таблица 4. Представление чисел и вес разрядов в байте.

Бит 7	Бит 6	Бит 5	Бит 4	Бит 3	Бит 2	Бит 1	Бит 0
2^7 =128	2^6 =64	2^5=32	2^4 =16	2^3 =8	2^2 =4	2^1 =2	2^0 =1
Диапазон представления числа в байте от 0 до 255: 1*128 + 1*64 + 1*32 + 1*16 + 1*8 +1*4 +1*2 +1*0=255							

В первых поколениях ЭВМ также существовало понятие машинного слова, которое имело различную разрядность в разных типах ЭВМ. В дальнейшем произошла некоторая стандартизация в соответствии с теми же степенями двойки.

Если с размерами, кратными степени двойки, и весами разрядом внутри одного байта, компьютерные инженеры пришли к единому стандарту, то с тем, с какой стороны считать веса разрядов в многобайтовом числе, противоречий не меньше, чем между тупоконечниками и остроконечниками из романа "Приключения Гулливера" Джонатана Свифта.

Первый подход берет свое начало с первых компьютеров корпорации IBM, и считает биты справа налево, но при этом нулевой значащий разряд и соотвественно стартовый адрес многобайтового числа отображается в адрес старшего или наиболее значащего байта (Рис.1а). Этой схемы придерживаются также RISC процессоры семейств Power, MIPS, SPARC и Alpha. В англоязычной литературе

такой способ адресации называют big-endian или MSB (Most Significant Byte first – наиболее значащий байт первым)

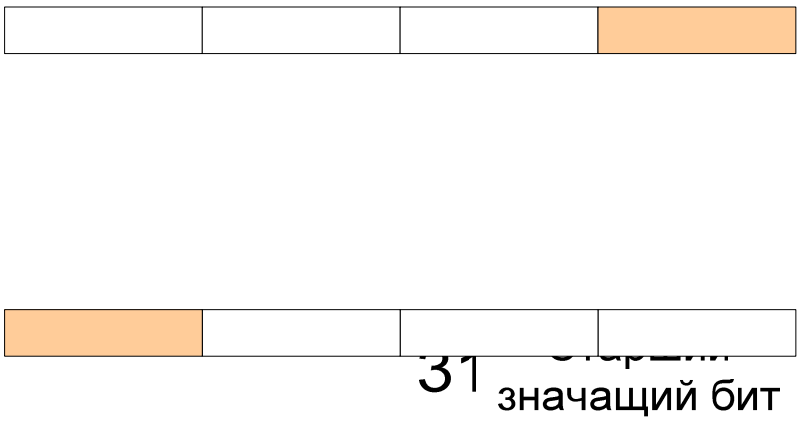


Рис.1. Способы адресации и расположение разрядов в многобайтовых словах.

Другой подход, в котором биты считают слева направо, используется в процессорах компании Intel. В этом случае нулевой бит находится в самом левом разряде, а адресом многобайтового числа считается адрес самого младшего байта (Рис.1б). В англоязычной литературе такой способ представления данных получил название little-endian или LSB (Least Significant Byte first – наименьший байт первым).

Дебаты о «правильности» представления многобайтовых слов ведутся давно, причем каждая сторона выдвигает свои аргументы: LSB позволяет загружать разные порции многобайтовых чисел с одного адреса и более прост при расширении многобайтовых последовательностей, а MSB позволяет оценить максимальное значение числа, не загружая всего числа и более понятен при записи, поскольку размещение байт совпадает с порядком записи чисел в обычной позиционной системе счисления, например десятичной, которая используется практически во всех языках мира. Кроме того принцип MSB используется в сетевом протоколе IP, что делает процессоры, использующие этот способ адресации более пригодными для обработки этого протокола. Многие современные процессоры имеют режимы поддержки обоих режимов представления данных в

многобайтовых словах, которые могут переключаться программным способом.

После рассмотрения альтернативных способов адресации многобайтовых чисел, рассмотрим двоичное представление чисел без знака, а также положительных и отрицательных целых чисел, что тоже имеет некоторые особенности.

Наиболее простым случаем являются числа без знака, можно подразумевать их положительными величинами в диапазоне от нуля до бесконечности. Иллюстрация на Рис. 2 ограничивает диапазон от нуля до 15_{10} для четырех-разрядного двоичного кода. Кроме того, на диаграмме представлены альтернативные обозначения чисел $10_{10}=A_{16}$, $11_{10}=B_{16}$, $12_{10}=C_{16}$, $13_{10}=D_{16}$, $14_{10}=E_{16}$ и $15_{10}=F_{16}$. Это дополнительные цифры, которые используются в шестнадцатеричной системе счисления, представленные в Таблица 3. Числа без знака являются типичными операндами команд обработки целых чисел.

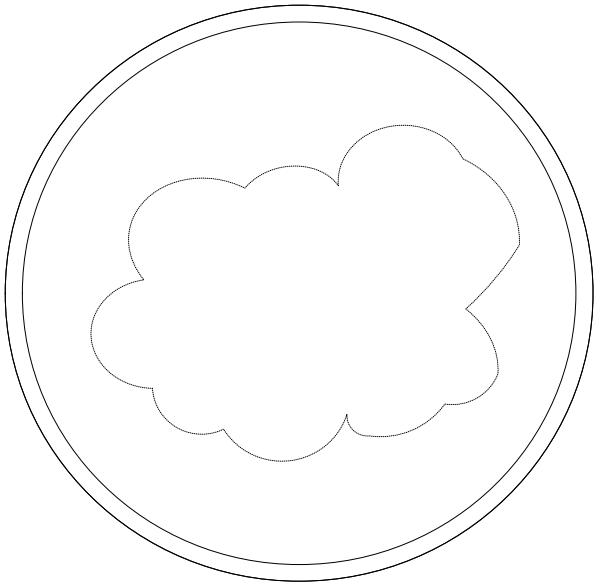


Рис. 2. Иллюстрация представления чисел без знака

В простейшем случае для представления отрицательных чисел необходимо ввести некоторый признак, который бы позволил отличать положительные величины от отрицательных. Таким признаком является дополнительный знаковый разряд, обычно старший – у положительных числе он равен "0", а у отрицательных – "1". Например, $10_{10} = 01010_2$, а $-10_{10} = 11010_2$. Такой способ представления чисел называется записью в прямом коде (sign-magnitude representation).

Интересной особенностью записи в прямом коде является наличие двух способов записи нуля: он может быть записан и как $1000\ldots0_2$, и как $0000\ldots0_2$., что проиллюстрировано на круговой диаграмме Рис. 3.

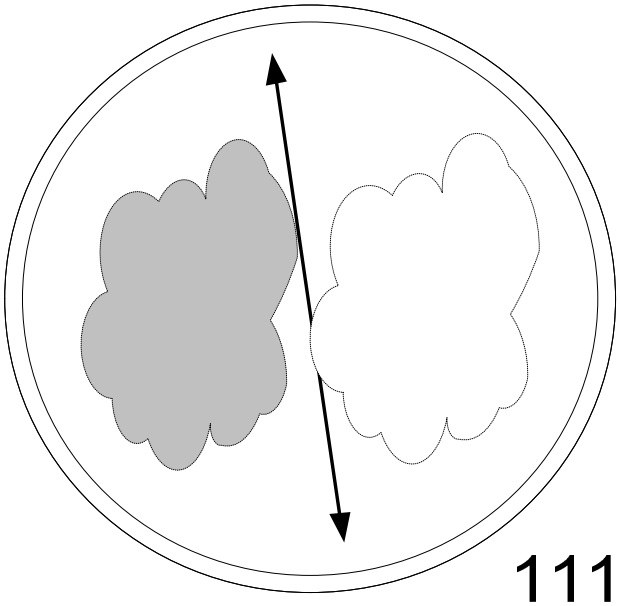


Рис. 3. Иллюстрация представления чисел в прямом коде

ПРИМЕР:

Рассмотрим примеры сложения и вычитания 4-разрядных чисел в прямом коде:

$3_{10} + 4_{10} = 7_{10}$ => $(0_+)011_2 + (0_+)100_2 = (0_+)111_2 = 0111_2$;

$6_{10} - 3_{10} = 3_{10}$ => $(0_+)110_2 + (1_-)011_2 = 110_2 - 011_2 = (0_+)011_2 = 0011_2$

Старший разряд в операциях сложения и вычитания содержит знак, который определяет арифметическое действие. Во втором примере мы просто складываем положительное и отрицательное число и знак второго операнда превращает операцию сложения в вычитание, как это происходит и в обычной десятичной арифметике.

Кроме записи в прямом коде, существует несколько других способов представления отрицательных величин: можно использовать дополнения до единицы или двойки (complement representation) либо можно использовать смещение в положительную область (biased representation).

При представлении отрицательных значений в виде дополнения до единицы (1's complement), что также называется записью в

обратном или инверсном коде, проиллюстрированном на круговой диаграмме Рис. 4. Здесь в каждом разряде, где при положительном значении числа находится единица, в отрицательном значении того же числа будет ноль и наоборот, например:

$1010 = \mathbf{0}\ 1010_2$

$-1010 = \mathbf{1}\ 0101_2$

В случае использования обратного кода сумма положительного N-разрядного числа с его отрицательным дополнением будет равна 2^{N-1}.

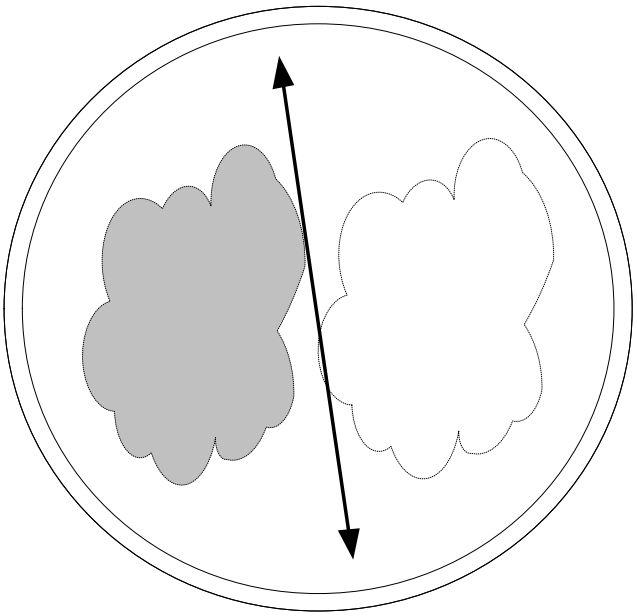


Рис. 4. Иллюстрация представления чисел в обратном коде

Представление отрицательных чисел в виде дополнения до двойки (2's complement) проиллюстрировано на круговой диаграмме Рис. 5. Его также называют записью в дополнительном коде, который позволяет легко производить операции вычитания с использованием того же сумматора, что и для операций сложения. При этом вычитаемое надо преобразовать в дополнительный код, что происходит в два этапа – на первом этапе производится перевод положительного значения в обратный код, после чего на втором этапе к полученному на первом этапе значению добавляется единица, например:

Прямой код $10_{10} = 01010_2$

Обратный код $-10_{10} = 10101_2$

Дополнительный код $-10_{10} = 10101_2 + 0001_2 = 10110_2$

Очевидно, что сумма положительного N-разрядного числа с его отрицательным отображением, представленном в дополнительном коде, будет равна 2^N.

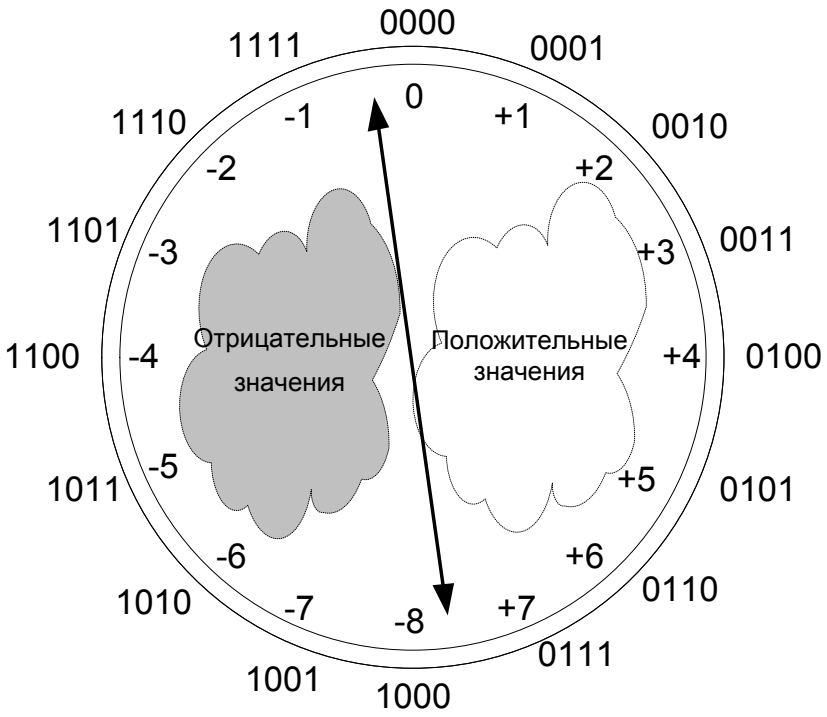


Рис. 5. Иллюстрация представления чисел в дополнительном коде

ПРИМЕР:

Рассмотрим пример вычитания 4-разрядных чисел в дополнительном коде, используя преобразованный выше код числа 10_{10}:

$15_{10} - 10_{10} = 0\ 1111_2 + 1\ 0110_2 = (1)\ 00101_2 = 5_{10}$

Вместо вычитания производится обычное сложение, при этом $10_{10} = 01010_2$ должно быть преобразовано в дополнительный код, как это было проиллюстрировано выше. При этом сложении возникает перенос из знакового разряда, который игнорируется.

Особенностью записи в дополнительном коде также является тот факт, что диапазон значений, которые можно представить, используя N разрядов, несимметричен относительно нуля: минимальное отрицательное число, которое можно записать, используя N разрядов, имеет значение -2^{N-1}, а максимальное положительное - $2^{N-1}-1$, что создает некоторые неудобства при обращении чисел. Несмотря на это, запись в дополнительном коде используется для представления чисел со знаком во всех современных компьютерных архитектурах – причины этого будут понятны после рассмотрения способов сложения

и вычитания двоичных чисел, реализованных в аппаратных блоках микропроцессоров.

В случае записи положительных значений в прямом, обратном или дополнительном коде, двоичное представление не отличается от записи этих значений в беззнаковой форме. Использование смещенного (biased representation) кода изменяет такой порядок – в этом коде для представления числа со знаком используется сумма числа с значением некоторого положительного смещения. Такой способ также называется беззнаковым представлением положительных и отрицательных чисел. На круговой диаграмме Рис. 6. проиллюстрировано представление числа в смещенном коде (смещение 8), когда к значению числа в прямом коде прибавляется значение смещения, например если к отрицательному числу -7 прибавить смещение 8, то результатом будет -7+8=1, имеющее двоичное представление "0001". Аналогично будут смещены и положительные числа, например положительный нуль "0000" станет восьмеркой (см. представление чисел без знака), положительная единица "0001" станет девяткой и так далее.

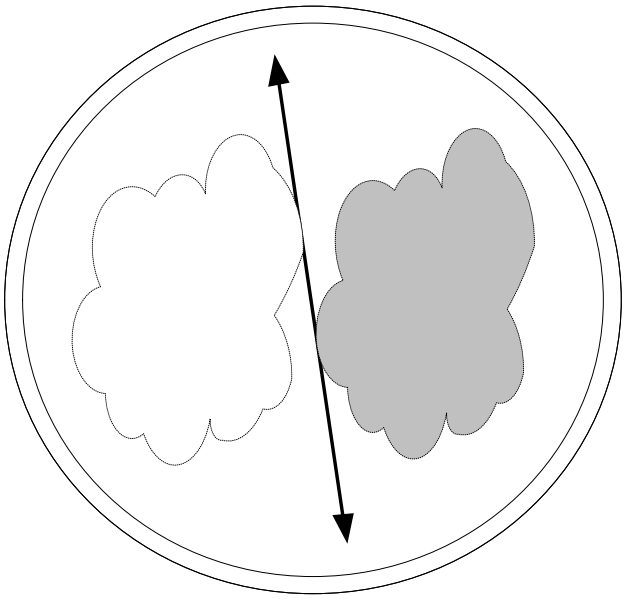


Рис. 6. Иллюстрация представления чисел в смещенном коде

Обычно при записи N-разрядных чисел используют смещение, равное 2^{N-1}, что позволяет представить симметричный относительно нуля диапазон значений (пример на диаграмме $2^{N-1}=8$). Кроме того, использование смещенного кода приводит к тому, что значения монотонно возрастают, что делает этот код удобным для использования в блоках АЦП и ЦАП, а также в представлении

компонент вещественных чисел, что позволяет также иметь монотонно возрастающую экспоненту или порядок.

Арифметика в смещенном коде обычно сводится к приведению операндов в обычный прямой код, что требует вычитания значения смещения. Затем проводится сама операция над прямым кодом (и дополнительным в случае вычитания) и к результату опять добавляется значение смещения. Так как, мы прибавляем и вычитаем одно и тоже значение смещения, и они могут взаимно уничтожаться, что позволяет упростить арифметическое устройство и избежать избыточных операций.

Существуют также другая форма представления чисел, когда в каждой позиции используется не только цифра со значением, но также и знак. Такое представление называется «знак-цифра» (sign-digit), причем основание такой системы счисления будет совершенно аналогично интерпретироваться, как и в примере чисел без знака, где есть только значения. Например, для десятичной системы с диапазоном [0, 9] существует аналог в виде «знак-цифра» [-4, 5]. Число элементов алфавита также равно 10 и с его помощью можно представить любое десятичное число. Например, число «3 -1 5» будет эквивалентно десятичному числу 255 = 300 - 10 + 5, или число «-3 1 5» будет эквивалентно десятичному числу -285= -300 + 10 +5.

Пример выше описывает неизбыточную систему «знак-цифра», когда число элементов алфавита равно основанию (radix=10). Если ввести даже один дополнительный элемент, то система кодирования становится избыточной (redundant sign-digit). Например, диапазон [-5, 5] является избыточным для основания 10 и одно и тоже значение может быть выражено несколькими варантами комбинаций «знак-цифра». Например тоже самое десятичное число 255_{10} может быть представлено как (300 -50 +5) = «3 -5 5» , либо как $(200 + 50 + 5)_{10}$ = «2 5 5». Если избыточность представления увеличить, например до диапазона [-7, 7], оставляя основание 10, то тогда десятичное число 295 можно представить в трех различных записях: 295_{10} = «3 -1 5» = «3 0 -5» = «1 -7 0 -5»

Если диапазон двоичной (radix-2) системы без знака [0,1], то диапазон представление двоичной системы «знак-цифра» будет [-1, 0, 1], что говорит о ее избыточности. Зачем нужно такое представление и как его использовать в арифметических устройствах, мы будем обсуждать в секции двоичного умножения.

Другой версией избыточного представления чисел является использование двух цифр, например, значения суммы и возможного переноса в следующий разряд, каждый из которых кодируется

отдельной цифрой. Использование такого кодирования будет рассмотрено в следующей секции.

3 Сложение и вычитание двоичных чисел. Двоичные сумматоры

Поскольку в двоичной арифметике используется позиционная система записи чисел, сложение может быть выполнено поразрядно. Сложение является простейшей двоичной операцией. Правила сложения также достаточно тривиальны:

0 + 0 = 0 ; 0 + 1 = 1 ; 1 + 0 = 1; 1 + 1 = 10 (возникает «1» переноса в старший разряд)

Сложение двух единиц порождает значение "10", эквивалентное десятичной 2. Это абсолютно также как происходит в обычной десятичной системе при сложении двух цифр если результат равен или превышает основание системы 10, то цифра слева должна быть увеличена на единицу.

5 + 5 = 10 ; 7 + 9 = 16

Это известно как эффект переноса в старший разряд в большинстве систем счисления, возникающий при превышении основания системы счисления. В двоичной системе все работает таким же образом :

1 1 1 1 1 (перенос)

0 1 1 0 1 (первый операнд)

\+ 1 0 1 1 1 (второй операнд)

= 1 0 0 1 0 0 (результат)

В этом примере два числа 01101_2 (13_{10}) and 10111_2 (23_{10}) складываются друг с другом. Верхняя строчка показывает использование переносов между разрядами, которые возникают при поразрядном сложении операндов. Младший разряд в самом правом столбце $1 + 1 = 10_2$. При этом 1 является переносом в старший (левый) разряд, 0 записывается в результат в правом столбце.

Производится сложение операндов в следующем разряде (столбце) $1 + 0 + 1 = 10_2$, при этом добавляется значение переноса из предыдущего разряда (столбца). Возникает опять 1 переноса и 0 записывается в этот разряд результата. В третьем разряде (столбце) $1 + 1 + 1 = 11_2$. В этот раз также возникает 1 переноса и единица записывается в третий разряд (столбец) результата. И так происходит разряд за разрядом, пока не получится конечный результат 100100_2.

Правила вычитания двоичных чисел также достаточно просты:

0 − 0 = 0 ; 0 − 1 = 1 (заём из старшего разряда) **; 1 − 0 = 1 ; 1 − 1 = 0**

Одно двоичное число может быть вычтено из другого похожим способом, только вместо переноса используется заём единицы из старшего разряда:

```
    *   * * *   (звездочки означают заём из старшего разряда)
  1 1 0 1 1 1 0
−     1 0 1 1 1
---------------
= 1 0 1 0 1 1 1
```

Следует отметить, что вычитание положительного числа эквивалентно сложению с отрицательным числом той же абсолютной величины. Преобразование вычитаемого в дополнительный код, как уже упоминалось выше, позволяет полностью исключить необходимость в отдельной операции вычитания в АЛУ микропроцессоров.

Теперь вкратце рассмотрим основные логические элементы и их функции, используемые для построения АЛУ микропроцессоров. В самом начале главы упоминалось о всего двух цифрах и трех базовых таблицах для двоичной системы счисления, теперь наступил момент, когда содержание этих таблиц нужно выучить для дальнейшего понимания излагаемого материала. На Рис. 7 представлены схематические обозначения и функции логических элементов «И», «ИЛИ», «Инвертор», «Исключающее ИЛИ» и логическая схема мультиплексора, часто используемого в цифровых структурах. Запомнить содержание трех простых таблиц логических функций не представляет особого труда, поэтому рекомендуем это сделать сразу и только после этого продолжить чтение. Крайне полезным может оказаться запоминание системы обозначений логических элементов – как метрической, так и американской, которые приведены для каждого элемента. Во многих документах и публикациях используются либо одна, либо другая система.

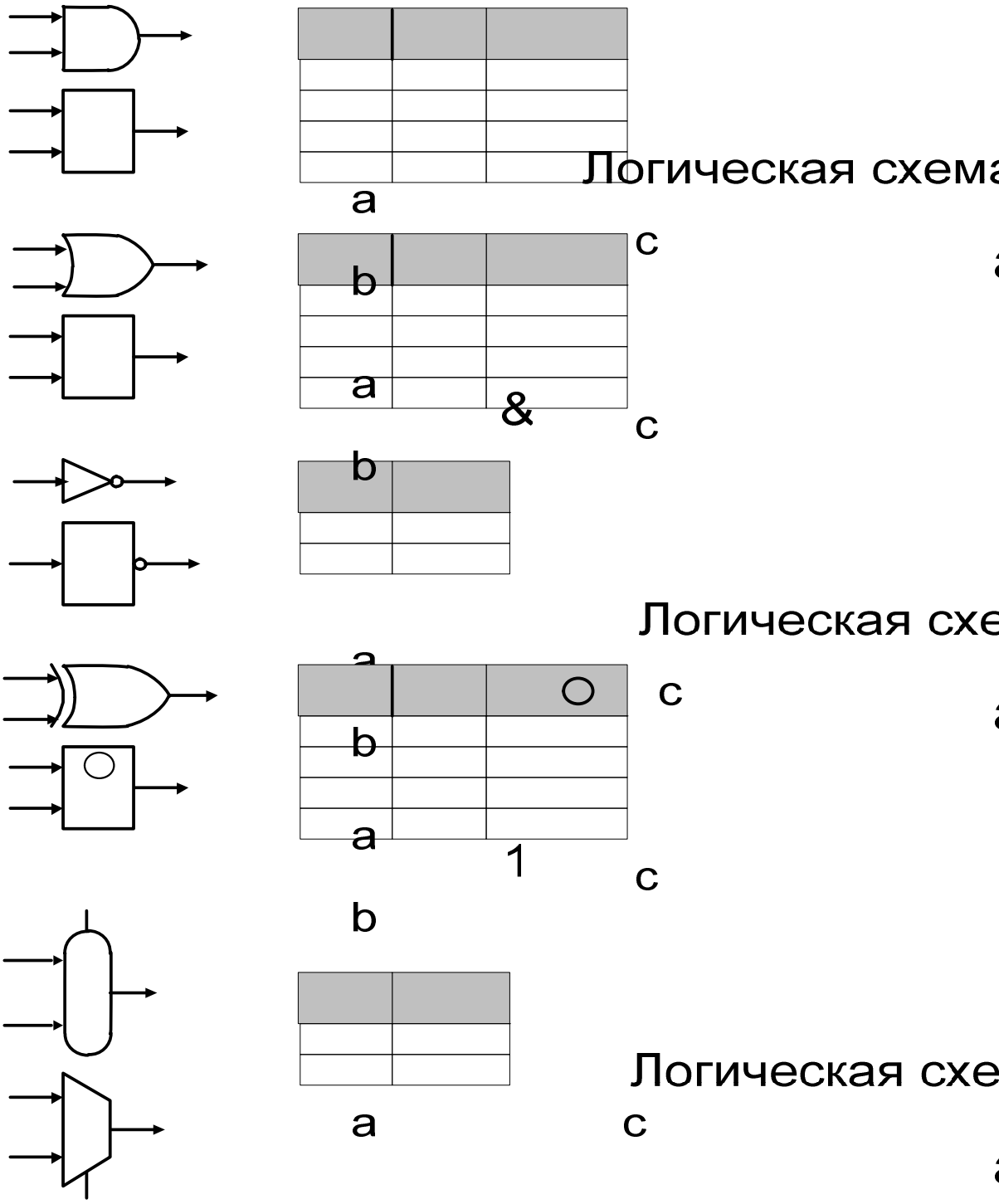
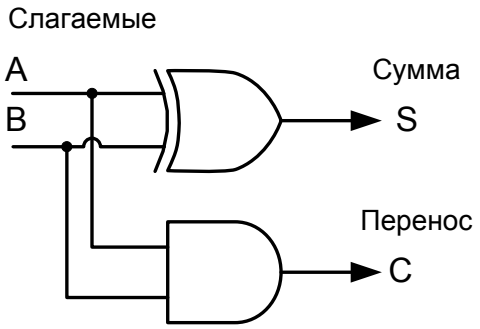


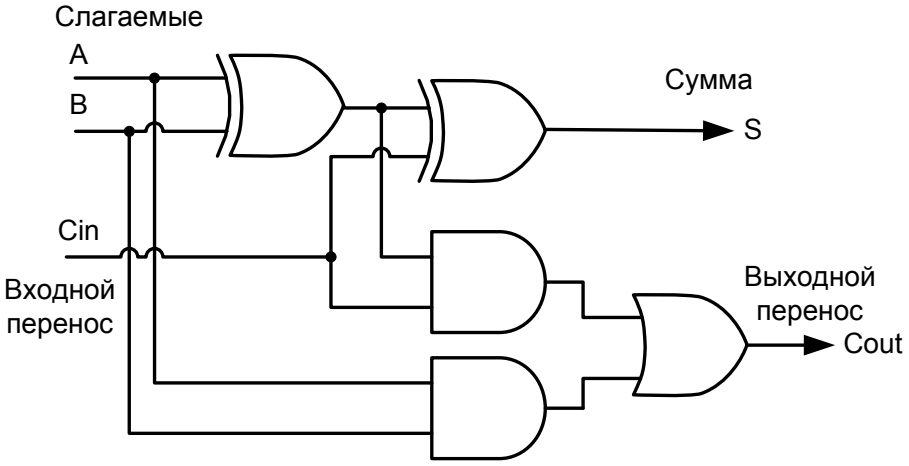
Рис. 7. Логические элементы и функции, используемые в построении цифровых устройств

В простейшем случае для сложения двух одноразрядных двоичных чисел достаточно одной логической схемы, выполняющей операцию «исключающее ИЛИ» (сложение по модулю 2). Такие схемы еще называют четверть-сумматором. Сумматор, который имеет на входе два одно-разрядных числа, а на выходе – одноразрядный

результат и значение переноса в следующий разряд называется полусумматором. Для получения полного одноразрядного сумматора, которым может быть использован как кирпичик при строительстве многоразрядных сумматоров, необходимо в процесс вычисления результата добавить значение переноса из предыдущего младшего разряда. Схемы полусумматора и полного сумматора представлены на Рис. 8.



A) Логическая схема одноразрядного полусумматора



B) Логическая схема полного сумматора

Рис. 8. Логическая схема одноразрядного полусумматора и полного сумматора.

Имея в своем распоряжении полный одноразрядный сумматор, можно достаточно просто построить последовательный многоразрядный сумматор – на вход каждого одноразрядного сумматора подается значение соответствующего разряда многоразрядного числа, а перенос из N-го разряда подается на вход переноса N+1-го сумматора (Рис. 9а). Рачительный читатель может заметить, что в младшем разряде многоразрядного сумматора может быть использован полусумматор, но не стоит торопиться с экономией

нескольких транзисторов – полный одноразрядный сумматор в младшем разряде может еще пригодиться.

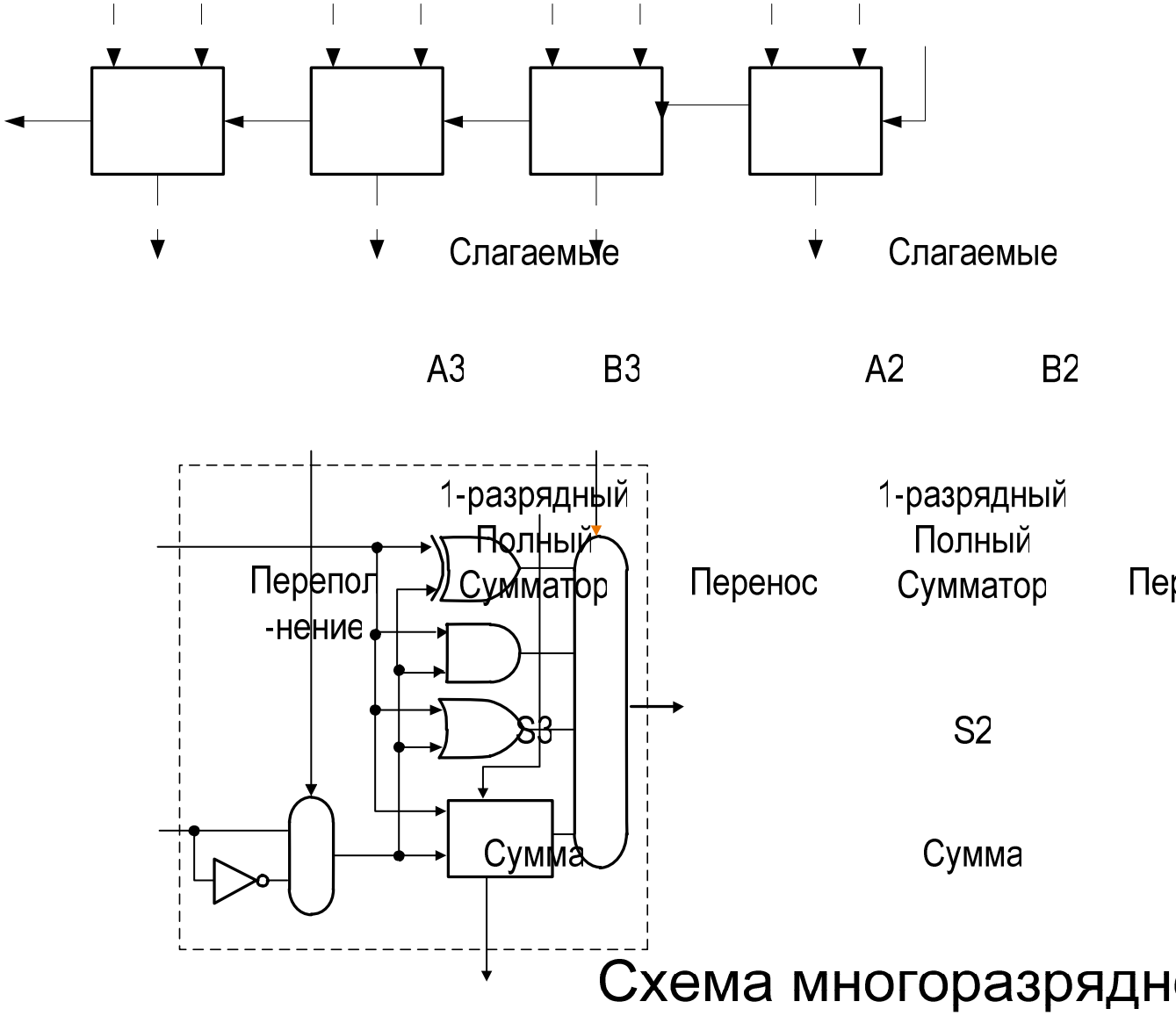


Рис. 9. Схемы многоразрядного сумматора и простого одноразрядного АЛУ.

Итак, мы построили последовательный многоразрядный сумматор (в англоязычной литературе такой тип сумматоров называют сумматорами с распространением переноса (ripple-carry adder). Что произойдет, если на вход такого сумматора подать положительное и отрицательное число, представленное в дополнительном коде? Если представить, что значение знакового разряда при записи в дополнительном коде используется как признак умножения этого разряда на –1, т.е.

$$A = (-1)*a_N*2^N + \Sigma a_i*2^i \quad (i=0, N-1)$$

Например, можно представить $-10_{10}= (1)\ 0110_2= (-1)*1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0 = -16_{10} + 6_{10} = -10_{10}$

тогда можно записать результат сложение A + B в следующем виде:

$$A + B = -1*a_N*2^N + \Sigma a_i*2^i + -1*b_N*2^N + \Sigma b_i*2^i = (a_N + b_N) * -1*2^N + \Sigma(a_i + b_i)*2^i \quad (i=0, N-1)$$

Как видно, в каждый разряд суммы вычисляется как результат сложения соответствующих разрядов слагаемых, при этом способ записи слагаемых не играет роли, т.е. мы можем использовать обычный сумматор для сложения чисел, представленных в дополнительном коде.

Более того, используя тот факт, что сумма числа A^+ и записанного в дополнительном коде противоположного A^- всегда равна 2^N, мы можем записать:

$$B – A^+ = B + (-A^+) = B + (2^N + A^-) = 2^N + B + A^-,$$

То есть, вычисление разности между двумя числами можно заменить вычислением суммы уменьшаемого и обратного значения вычитаемого. Несимметричность диапазона представления становится мелочью по сравнению с преимуществами, возникающими при использовании записи в дополнительном коде.

Вот только что случилось с 2^N? Куда оно исчезает из конечного результата? И что вообще происходит, если сложить два числа, сумма которых не может поместиться в N разрядов? В этом случае происходит переполнение разрядной сетки. Различают положительное и отрицательное переполнение: первое возникает когда сумма двух положительных N-разрядных чисел больше, чем 2^N-1; второе – когда сумма двух отрицательных чисел меньше, чем $–2^N$. Для обнаружения переполнения при при выполнении арифметических операций используется копия знакового разряда - если в результате операции значения первого и второго знакового разряда отличаются, то произошло переполнение. Например, складывая два четырехразрядных числа 0101_2 и 0011_2, мы имеем:

$$(0)0.101 + (0)0.011 = (0)1.000$$

или, в десятичном виде, 5 + 3 = -16 – очевидно неправильно, следовательно произошло положительное переполнение, что подтверждается фактом разницы в значениях знаковых разрядов. При сложении чисел 1101_2 и 1011_2

$$(1)1.101 + (1)1.011 = (1)1.000$$

отрицательного переполнения не возникает.

Аналогично многоразрядному сумматору, построены многоразрядные простые АЛУ, которые используют одноразрядную секцию, структура которой приведена на Рис. 9б. Секция АЛУ может выполнять четыре логические операции (И, ИЛИ, Исключающее ИЛИ, Инверсия), а также сложение и вычитание одноразрядных переменных. Помимо входов операндов и переноса и выхода результата с переносом имеются входы управления для инвертирования входного операнда и выбора выполняемой операции. Инвертирование может использоваться как логическая операция или как часть операции вычитания, когда вычитаемый операнд инвертируется перед сложением и в самый младший разряд АЛУ добавляется единица входного переноса. Как видно из схемы, АЛУ выполняет все операции одновременно, вход операции выбирает нужный результат, который попадает на вход мультиплексора. Только сложение и вычитание не могут выполняться одновременно, так как эти операции используют тот же самый сумматор. Следует отметить, что принцип выбора нужного результата часто используется при построении тракта данных микропроцессоров, так как обычно проще всего создать отдельный тракт данных для определенной группы операций и затем просто выбирать необходимый результат на выходном мультиплексоре. Требуемые комбинации для выполнения операций проиллюстрированы в Таблица 5.

Таблица 5. Управление одноразрядным простым АЛУ

A B	**Оп=00 Ин =X C0in=X EXOR**	**Оп =01 Ин =X C0in=X AND**	**Оп =10 Ин =X C0in=X OR**	**Оп =11 Ин =1 C0in=0 A=0 NOT**	**Оп =11 Ин =0 C0in=X Сумма A, B**	**Оп =11 Ин =1 C0in=1 Разность A, B**
0 0	0	0	0	1	Cin, Cout=0	Cin, Cout=0
0 1	1	0	1	0	1 + Cin, Cout	0 + Cin, Cout
1 0	1	0	1	-	1 + Cin, Cout	0 + Cin, Cout
1 1	0	1	1	-	1+1 + Cin, Cout	1+0 + Cin, Cout

Вернемся опять к нашему многоразрядному сумматору на Рис. 9а. Как мы видим, для вычисления значения i-го разряда суммы необходимо знать не только значения соответствующих разрядов каждого из слагаемых, но и значение переноса из предыдущего

разряда. Поскольку значение переноса вычисляется последовательно для каждого из разрядов, общее время вычисление суммы будет пропорционально числу разрядов слагаемых, что может привести к значительной задержке для современных 64-разрядных сумматоров. Для уменьшения времени суммирования существует несколько способов ускорения вычисления переноса.

Поскольку двоичный сумматор является логической схемой, то очевидно можно переписать правила вычисления для переноса из i-го разряда таким образом, чтобы вместо последовательного вычисления можно было вычислить перенос с использованием только двух логических уровней. Запишем правило для вычисления переноса в следующем виде:

$$C_i = G_{i-1} + P_{i-1}C_{i-1}, \quad \text{где } G_{i-1} = a_{i-1}*b_{i-1}, \text{ а } P_{i-1} = a_{i-1} + b_{i-1}.$$

Очевидно, что если оба слагаемых в i-1-м разряде имеют значение 1, то мы всегда будем иметь перенос в следующий разряд – это значение выражается через $\mathbf{G_i}$ (**generation**), в случае если одно из слагаемых имеет единицу в i-1 разряде, то перенос из предыдушего разряда будет «перемещен» в следующий разряд – это значение выражается через $\mathbf{P_i}$ (**propagation**). Подставив последовательно значения для переноса из предыдущих разрядов, можно подсчитать перенос в i-й разряд:

$$C_i = G_{i-1} + P_{i-1}*G_{i-2} + P_{i-1}*P_{i-2}*G_{i-3} + \ldots. + P_{i-1}*P_{i-2}\ldots..P_1*G_0 + P_{i-1}*G_{i-1}\ldots..P_1*C_0$$

Как видно из приведенной выше формулы можно вычислить перенос с использованием всего трех последовательных уровней логических схем: один для вычисления каждого из значений $\mathbf{P_i}$ и $\mathbf{G_i}$ и два для вычисления переноса (Рис. 10). Такой способ вычисления переноса можно назвать предсказанием и соответственно использовать в названии сумматора. В англоязычной литературе сумматоры такого типа называются carry-lookahead adder (CLA).

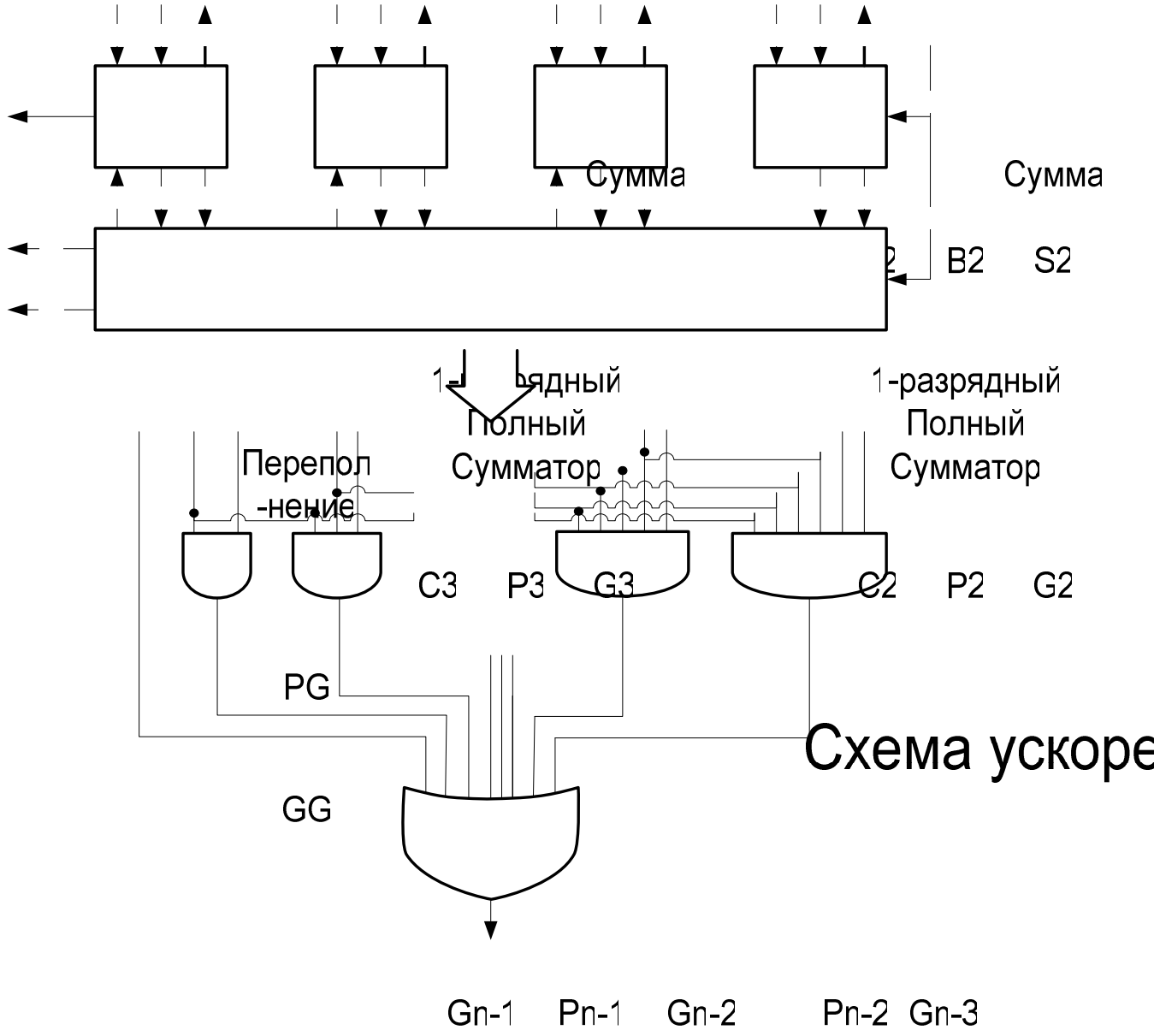


Рис. 10 . Схема сумматора с предсказанием переноса CLA

К сожалению, нерегулярность структуры, используемой для вычисления переноса, и необходимость разводки выходов схем первого уровня Gi и Pi на большое количество входов элементов Логическое-И второго уровня значительно снижает эффективность использования CLA в сумматорах большой разрядности. Нижняя часть Рис. 10, где показана реализация схемы ускоренного переноса для одного разряда, наглядно иллюстрирует данную проблему.

Можно использовать идею, предложенную в CLA, для того, чтобы построить сумматор, который уменьшает негативное влияние недостатков «чистого» CLA: если расширить понятия P и G с одного

разряда до блока из нескольких разрядов, Перенос из блока разрядов возникает либо в случае, когда он возникает внутри блока, либо распространяется через блок. Формулы для вычисления Pik и Gik можно записать в следующем виде:

$P_{ik} = P_{ij} * P_{j+1,k}$ (j=0, N-1; k=0, N-1)

$G_{ik} = G_{j+1,k} + P_{j+1,k} * G_{ij}$(j=0, N-1; k=0, N-1)

Используя эти формулы, можно построить регулярную древовидную структуру (Рис.11), которая позволяет вычислять перенос за log2n логических уровней или последовательных логических операций..

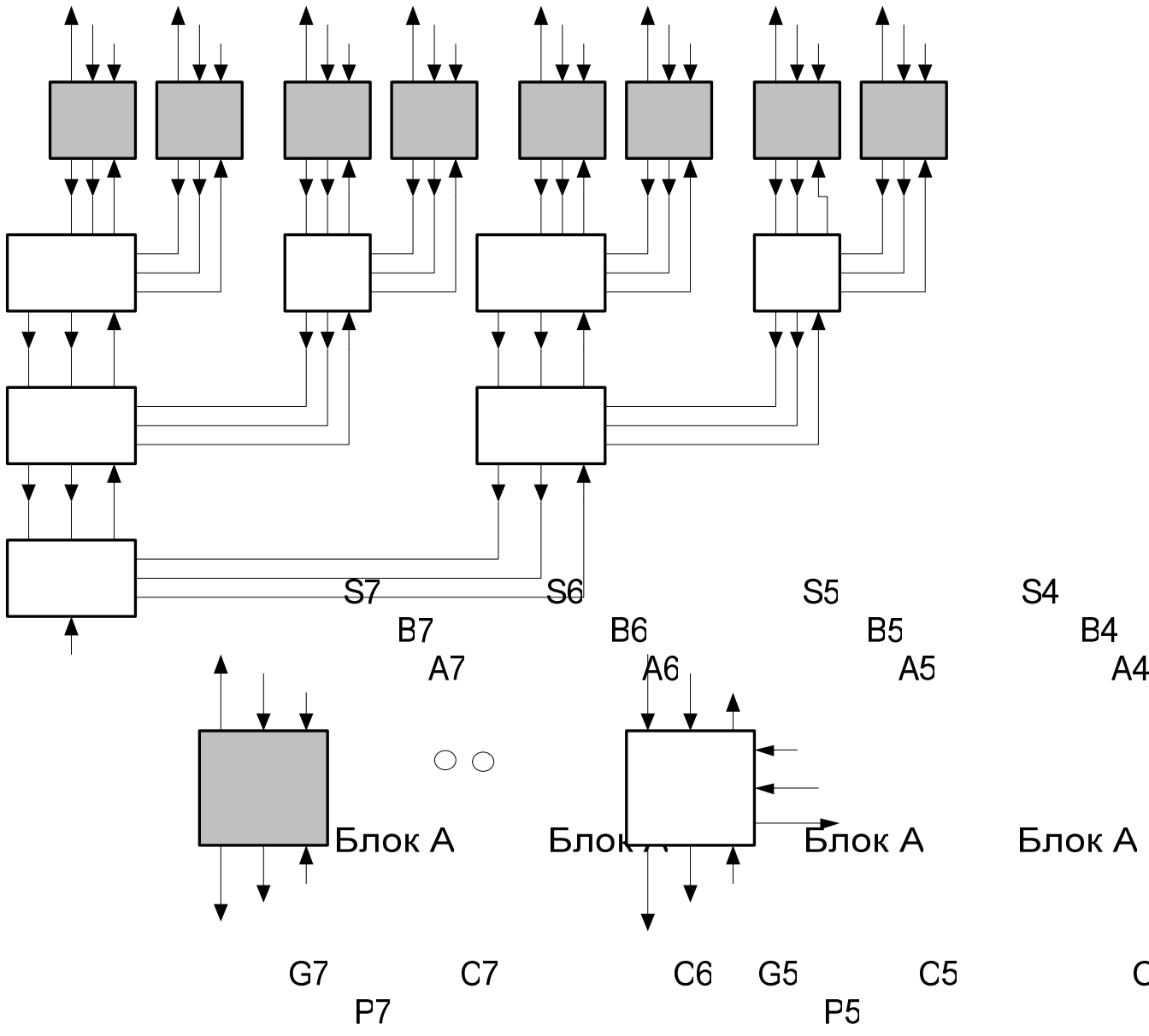


Рис.11 . Схема модифицированного сумматора с предсказанием переноса CLA

Развитие идей CLA на этом не заканчивается: не сложно заметить, что вычисление P_{ik} значительно проще, чем вычисление G_{ik}. Используя тот факт, что если блок из n разрядов порождает перенос (G_{ik} = 1), перенос не зависит от значения переноса из предыдущего блока. В этом случае подавая на вход блока 0 как значение переноса из предыдущего блока, мы гарантируем невозможность возникновения ложного переноса. Вычисляя значение P для каждого блока и подавая сумму P и переноса из младшего блока в блок, находящийся через один от соседнего, через схему «Логическое-И», можно значительно ускорить получение конечного результата. В этом случае P_{ik} используется как функция распространения переноса через блок и если оба P_{ik} и перенос имеют значение единицы, то нам не надо ждать, результата вычисления переноса из блока. Такой способ вычисления переноса получил название пропуска переноса или carry-skip adder (Рис.12).

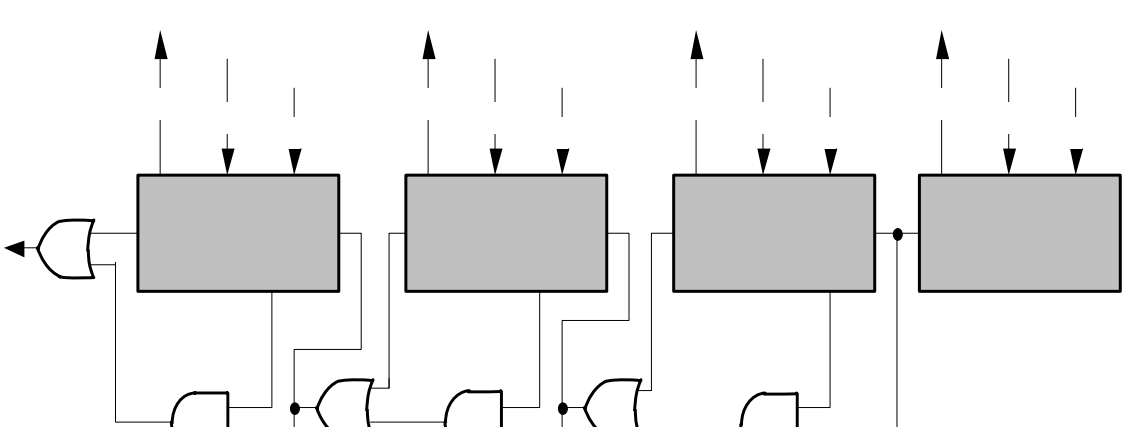
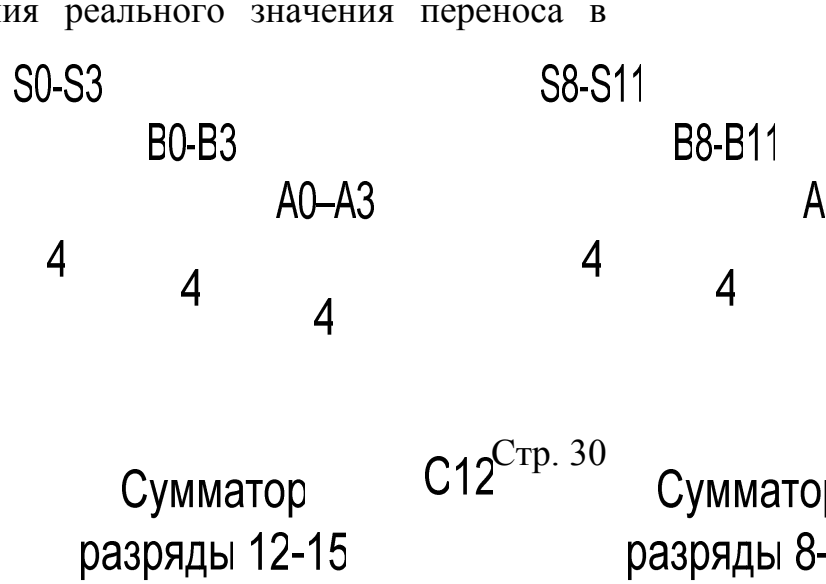


Рис.12. Схема сумматора с пропуском переноса carry-skip adder

Кроме блоков, описанных выше, используют также сумматоры, в которых параллельно вычисляются две суммы: одна для случая, когда перенос есть, а другая – когда переноса нет. В конце вычисления выбирается одна из двух сумм в зависимости от реального значения переноса (Рис.13а). При этом разряды сумматора разбиваются на группы для ускорения получения реального значения переноса в группе из четырех разрядов.



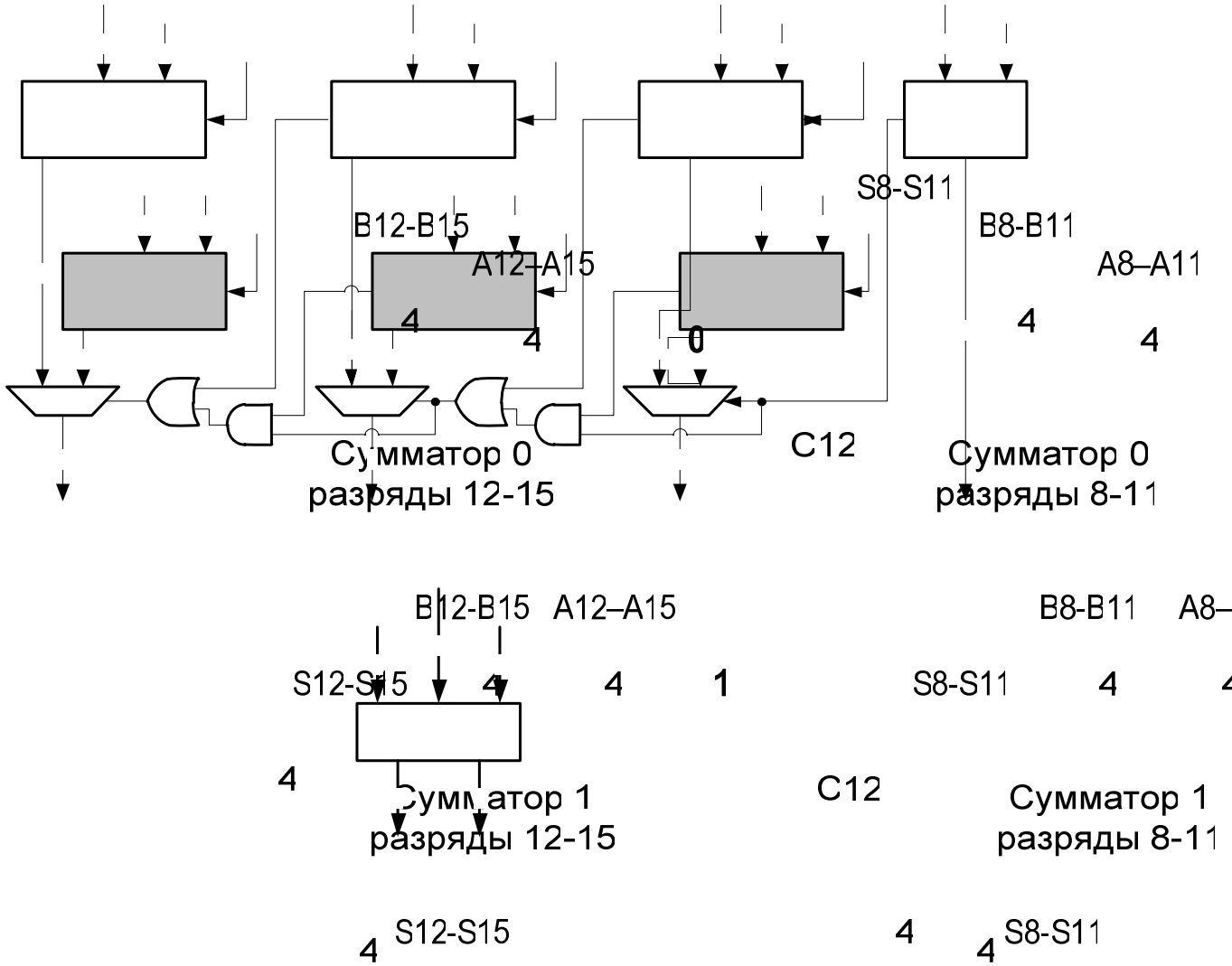


Рис.13. Схемы сумматора с выбором результата (carry-select adder) и сумматора с сохранением переноса (carry-save adder)

Другим способом решения проблемы переноса является использование избыточного кодирования результата сложения, когда в результате в каждой позиции мы можем получать значения, превышающие основание системы счисления. К примеру, попробуем сложить два десятичных числа без переноса из разряда в разряд:

	5	7	8	2	4	9
+	6	2	9	3	8	9
						Набор цифр [0,9]
	11	9	17	5	12	18
						Избыточный или расширенный набор цифр [0,18]
	1	1	1	0	1	1
						Десятичный перенос

CSA сумматор

1 2 0 7 6 3 8 Набор цифр [0,9]

Можно заметить, что в этом примере мы обошлись без переноса из младшего в старшие разряды, заменив это расширением набора значащих цифр в каждом разряде результата. А что произойдет, если мы произведем несколько последовательных сложений? Чтобы избежать переноса мы должны продолжать расширять набор значащих цифр, что нельзя делать бесконечно. В конце концов для того, чтобы вернуться к стандартному десятичному представлению, нам нужно будет выполнить все переносы, что проиллюстрировано в этом же примере.

В двоичной системе все обстоит таким же образом. Для избыточного кодирования используются две двоичные цифры, что соответсвует двум разрядам или битам: значение 0 представляется как (0,0), 1 представляется как (0,1) или (1,0), 2 представляется как (1,1). Нетрудно заметить, что избыточной цифрой для двоичного представления значением является двойка. На Рис.3.13б представлена схема сумматора с сохранением переноса (carry-save adder), которая производит операции в избыточной системе с цифрами [0,2]. На входы X Y подается число в двухбитовом представлении в диапазоне [0,2] и двоичное значение в диапазоне [0,1] на вход переноса. Разряд переноса из предыдущего разряда и сумма формируют результат в диапазоне [0,2] из предыдущего разряда. Перенос из сумматора формирует результат с суммой следующего разряда. Использование сумматоров с сохраненем переноса позволяет существенно ускорить последовательные сложения, которые, к примеру, необходимы при выполнении умножения. Более подробно использование дерева или матриц сумматоров с сохранением переноса будет рассмотрено в секции, описывающей реализацию двоичного умножения.

4 Выполнение сдвига двоичных чисел и логических операций

Хотя в большинстве случаев процессор обрабатывает данные на уровне слов, иногда бывает необходимо производить операции, которые изменяют не целое слово, а только несколько бит в слове, причем каждый бит в такой операции обрабатывается независимо от соседних битов. Все современные процессоры имеют команды для выполнения битовых операций И и ИЛИ. Некоторые процессоры имеют также команды для операций Исключающее-ИЛИ и НЕТ, реализация таких операций проиллюстрирована на схеме простого АЛУ Рис. 9б.

Кроме логических операций сушествует также класс операций, называемых сдвигом – эти операции перемещают все биты в слове на

несколько позиций вправо или влево, например: 001100 после сдвига на два разряда влево будет иметь значение 110000. При сдвиге влево новые разряды заполняются нулями. Сдвиг вправо более сложен – существует две разновидности этой операции: логический и арифметический сдвиг. В первом случае при сдвиге вправо новые разряды всегда заполняются нулями, а во втором – значение знакового разряда копируется в освободившийся разряд.

Сдвиг влево на один разряд

```
1 0 0 1 0 1 1 1 Исходное значение

0 0 1 0 1 1 1 0 ← Освободившиеся позиции справа
заполняются нулями
```

Сдвиг вправо на один разряд

```
1 0 0 1 0 1 1 1     Исходное значение

1 1 0 0 1 0 1 1 → Освободившиеся позиции слева
заполняются значением самого старшего (знакового)
разряда исходного значения
```

В дополнение к простым сдвигам, некоторые процессоры реализуют операцию циклического сдвига – в этом случае выдвигаемые разряды при сдвиге не сбрасываются, а перемещаются в другой конец слова. Например при выполнении циклического сдвига влево на три разряда 0001101 принимает значение 1010001.

Интересной особенностью операции сдвига является тот факт, что сдвиг целого числа на один разряд влево равнозначен умножению этого числа на 2, а арифметический сдвиг на один разряд вправо – делению на 2.

Сдвиг на один разряд за такт осуществляется регистром сдвига, сдвиг на несколько разрядов за такт осуществляется с помощью схемы сдвига представленной на Рис.14. Такие схемы называют матричным сдвигателем (barrel shifter), причем в зависимости от ширины сдвигаемого слова и соотвествующему этому максимального числа разрядов на которые может осуществляться сдвиг, такие схемы могут иметь несколько последовательных линеек мультиплексоров. Причем число позиций для требуемого сдвига в двоичном коде подается на соответствующие управляющие входы каждой линейки мультиплексоров. Для максимально возможного диапазона сдвига 8-разрядного слова требуется три линейки, для 16-разрядного - четыре и так далее. Если первая линейка обеспечивает сдвиг на одну позицию, то вторая на две, третья на четыре, четвертая на восемь и так далее.

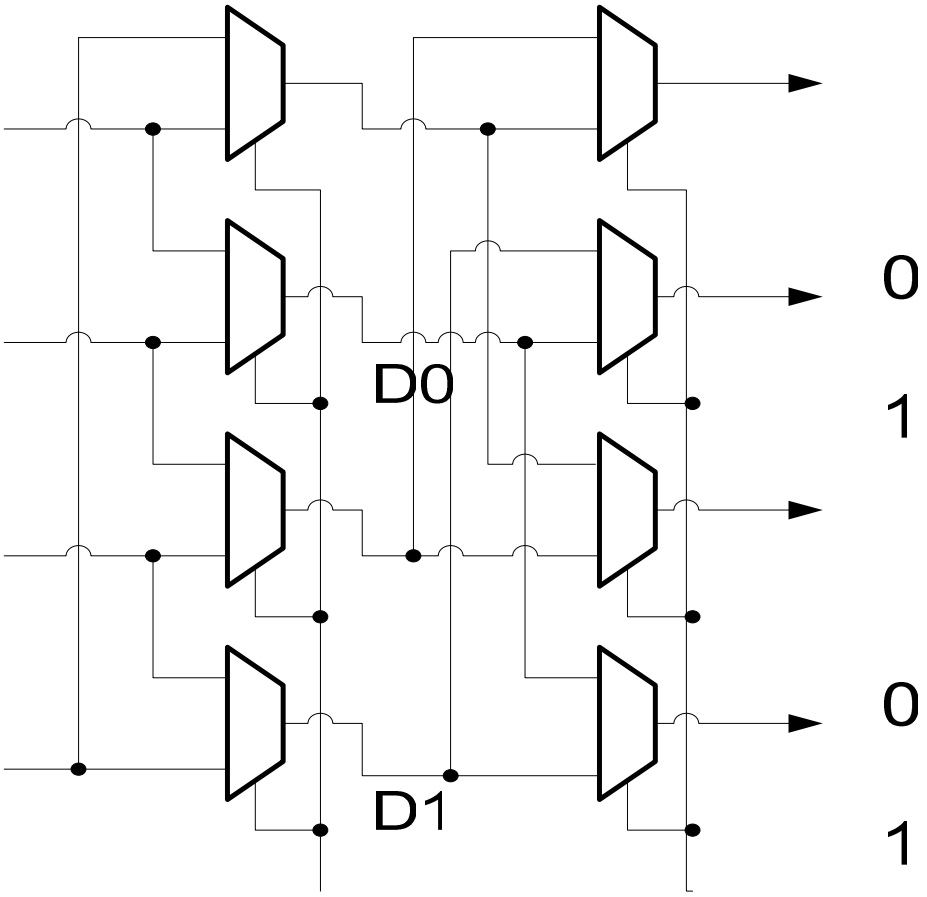


Рис.14. Схема сдвига на несколько разрядов

5 Умножение двоичных чисел. Двоичные умножители.

Умножение двоичных чисел снова возвращает нас к начальной школе – двоичное умножение многозначных чисел не отличается от умножения «в столбик». Например:

```
        0101 (5₁₀)
         *0011 (3₁₀)
         -----
        0101(«1» начальное значение, равное
множимому)
         0101 («1» Сдвиг влево и суммирование
множимого)
        0000  («0» только сдвиг)
        0000    («0» только сдвиг)
    -----------
```

00001111 (15_{10})(произведение)

Если внимательно посмотреть на пример выше, то видно, что все операции сложения производятся с единственным слагаемым – множимым, и в зависимости от значения каждого разряда множителя мы производим сложение значения множимого или нуля, при этом значение множимого сдвигается влево на каждом шаге последовательного сложения. Простейший алгоритм умножения работает именно на таком принципе (Рис.3.15).

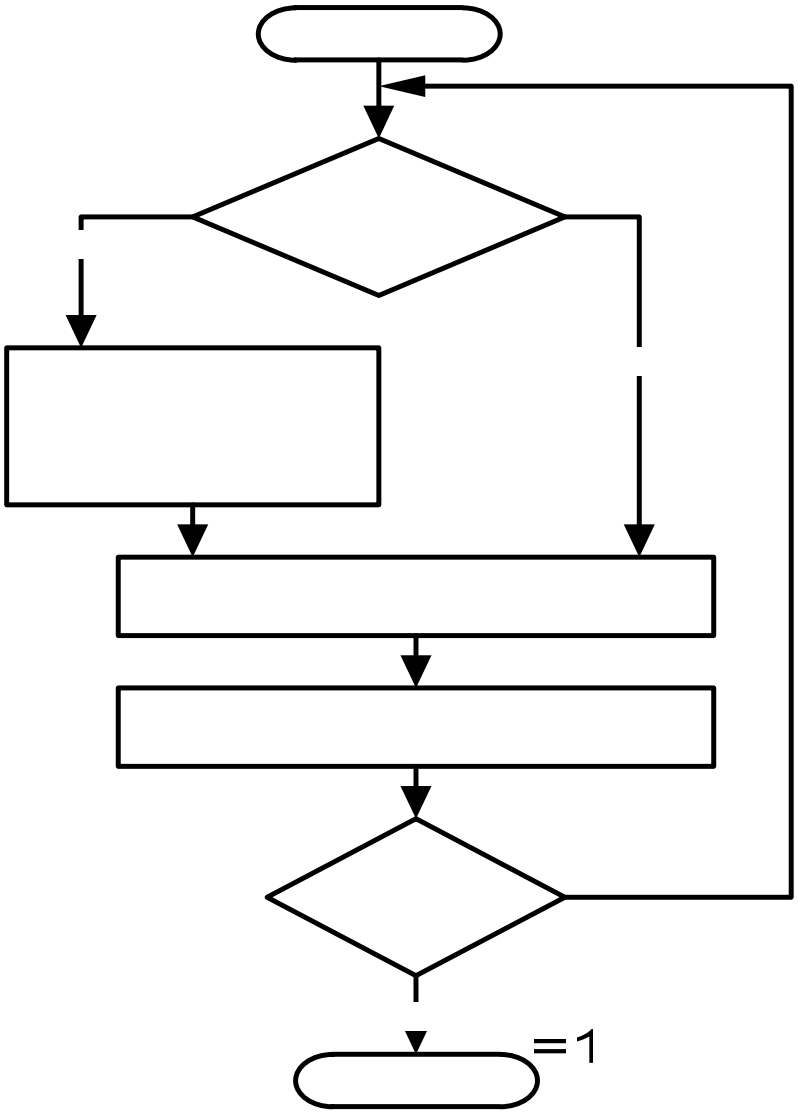


Рис.15. Алгоритм двоичного умножения.

Двоичное умножение строится на двух операциях – сдвига и сложения. Вначале произведение устанавливается в ноль, после чего в зависимости от значения соответствующего бита в множителе к частичному произведению добавляется значение множимого и после чего множимое сдвигается на один разряд влево. Множитель при этом сдвигается вправо. Весь алгоритм занимает столько шагов, сколько разрядов в множителе. А поскольку произведение двух N разрядных

чисел занимает 2N разрядов, такой способ умножения требует наличия двух регистров, способных вместить 2N-разрядное числа и сумматора такой же разрядности (Рис.16а).

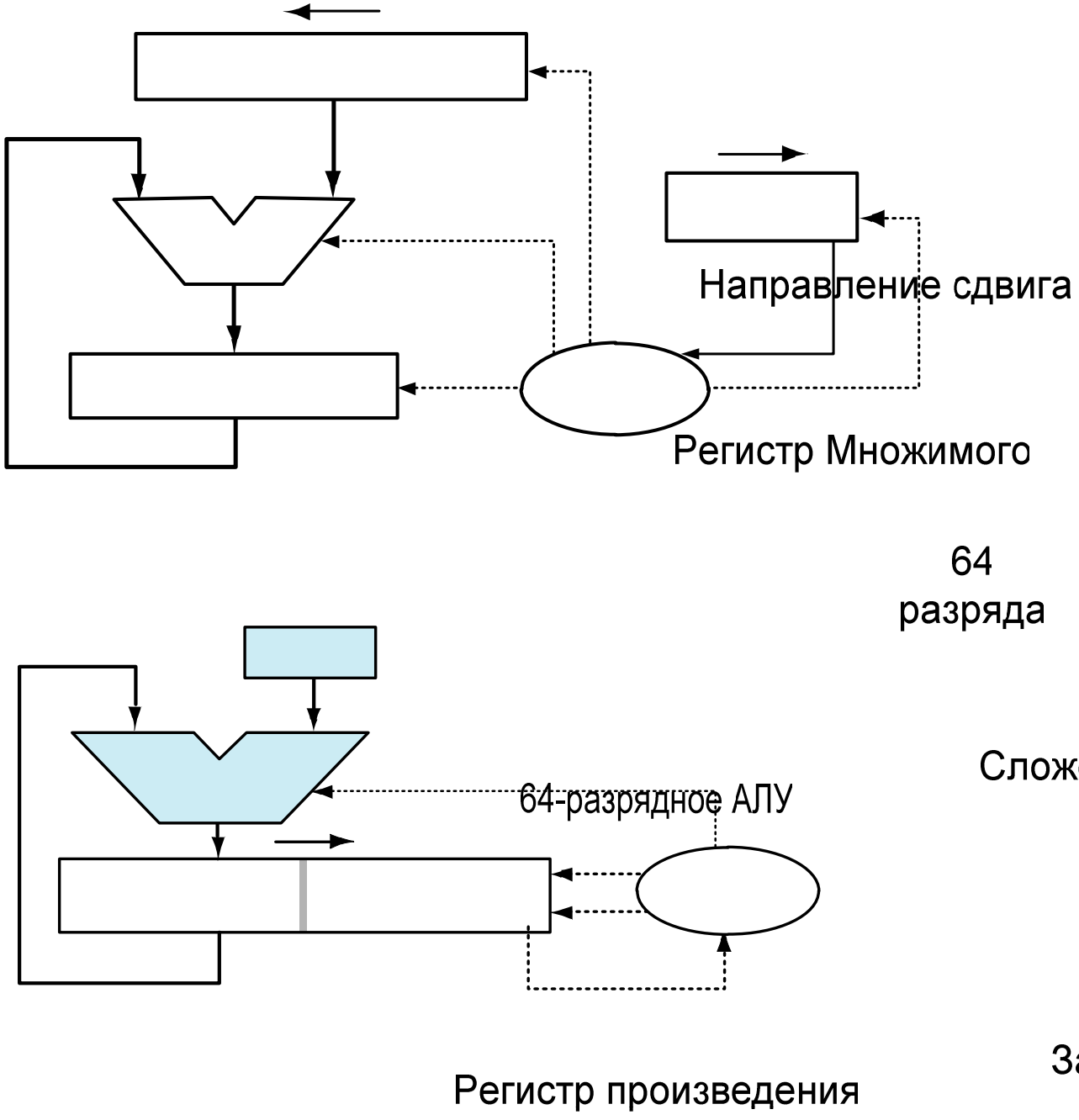


Рис.16. Схемы простого и оптимизированного умножителей.

Не сложно заметить, что на каждом шаге реально используются только половина разрядов сумматора. Если заменить сдвиг множимого влево на сдвиг произведения вправо, а результат суммирования помещать не в младшие, а в старшие разряды произведения, то можно использовать N-разрядный сумматор и N-разрядный регистр для множимого. Ну и поскольку на каждом шаге используются только

половина произведения, записывая значение частичного произведения в регистр множителя, можно отказаться от использования отдельного регистра для множителя (Рис.16б)

В примере, приведенном выше, оба операнда были положительными. А что произойдет, если нам необходимо умножить два числа с разным знаком? Очевидно, что можно перевести оба операнда к положительному виду, вычислить произведение и потом, в зависимости от знаков операндов, перевести произведение к отрицательному виду. Такой подход, хотя и прост, требует дополнительных затрат времени и ресурсов для приведения из одной формы в другую.

Если для представления чисел со знаком использовать дополнительный код, то в случае, если только один из множителей отрицательный, можно использовать тот же самый способ умножения, используя положительный множитель в качестве начального значения и сдвигая произведение с использованием арифметического, а не логического сдвига, как представлено на следующем примере:

1000 (-8_{10}) дополнительный код

* 0011 (3_{10}) прямой код

\-----

\+ 1000 («1» Положительное значение множимого в качестве начального значения)

>> 11000 (Арифметический сдвиг вправо с дублированием старшего разряда)

\+ 1000 («1» Прибавить значение множимого к частичному произведению)

>> 101000 (Арифметический сдвиг вправо частичного произведения)

>> 1101000 («0» только арифметический сдвиг вправо частичного произведения)

>> 11101000 (-24_{10})(«0» только арифметический сдвиг вправо)

Замечательно, но что же делать, если оба множителя отрицательные? Есть несколько способов вычисления произведения чисел со знаком. Наиболее распространенный способ использует кодирование Буфа (Booth) при умножении отрицательных чисел. Идея кодирования Буфа была впервые предложена для ускорения операций на машине, в которой сдвиг мог выполняться значительно быстрее, чем сложение. Поэтому Буф предложил использовать представление, в котором минимизируется число единиц в множителе, для чего множитель представлялся как сумма двух чисел, например $7_{10}= 0111_2$ представлялось как $–1_{10} + 8_{10}$ или $–0001_2 + 1000_2$, что в представлении

Буфа будет выглядеть как 1001^{-} . При умножении по правилам Буфа, если в множителе встречается 1-, то сложение заменяется на вычитание, например, умножая 5 на 7, мы имеем:

```
          0101 5₁₀
        * 0111 7₁₀ (1001¯) в представлении Буфа
          -----
       11111011 (множимое в дополнительном коде по разряду 1¯
)
          0000  (сдвиг по разряду 0)
         0000   (сдвиг по разряду 0)
        0101    (множимое в прямом коде по старшему биту 1)
      ---------
       00100011 (произведение)
```

Для случая с отрицательными числами, при использовании правила Буфа, число в дополнительном коде рассматривается как положительное число и для него вычисляется дополнение, например -4_{10} (1100_2 в дополнительном коде) рассматривается аналогично числу без знака 12 и далее вычисляется его представление в коде Буфа, что будет $12 = 16 - 4 = 101^{-}00$. Поскольку при умножении 4-х разрядных чисел единица в пятом разряде не будет принимать участие в формировании результата (умножение 4-х разрядных чисел использует только младшие 4 разряда), то результатом умножения будет противоположное значение множителя, сдвинутое на 2 разряда, что эквивалентно умножению на 4.

Множимое A		0	1	0	1					5
Множитель	x	1	1	0	0					-4
Y	1	0	1^{-}	0	0					Множитель в кодировании Буфа
Пошаговые операции										
Только сдвиг		0	0	0	0	0				0
Только сдвиг		0	0	0	0	0	0			0
Прибавить -A	+	1	0	1	1					Вычитание по 1^{-}
Результат		1	0	1	1	0	0			
Сдвиг		1	1	0	1	1	0	0		Сдвиг частичного произведения
Только сдвиг		1	1	1	0	1	1	0	0	-20

Другим способом вычисления произведения по алгоритму Буфа может быть использование двух соседних разрядов множителя. В зависимости от их значений производиться одна из операций:

«00» – нет арифметической операции

«01» – сложение множителя с частичным произведением

«10» – вычитание множителя из частичного произведения

«11» – нет арифметической операции.

Вне зависимости от значения разрядов на каждом шаге производится арифметический сдвиг вправо на один разряд.

Еще одним способом умножения чисел со знаком является модификация простого алгоритма умножения чисел без знака: множитель A, который используется для установки начального значения произведения, рассматривается как число без знака и выполняется n-1 шагов алгоритма умножения, на каждом шаге используется арифметический сдвиг вместо логического. На последнем шаге, если A было отрицательным, то из произведения вычитается значение множителя B и выполняется заключительный арифметический сдвиг, а если A было положительным, то выполняется только сдвиг.

Существует множество модификаций алгоритма Буфа и все современные высокоскоростные умножители используют ту или иную версию алгоритма Буфа в комбинации с деревом сумматоров.

Поскольку произведение занимает вдвое больше места, чем каждый из множителей, возникает интересная проблема с обработкой произведения: с одной стороны, в языках высокого уровня результат произведения двух целых чисел обычно присваивается другому целому числу – компилятор не использует старшие биты в произведении и можно было-бы их просто не вычислять, а с другой стороны, имея регистр двойной длины для записи результата можно ускорить вычисление произведения чисел разной длины.

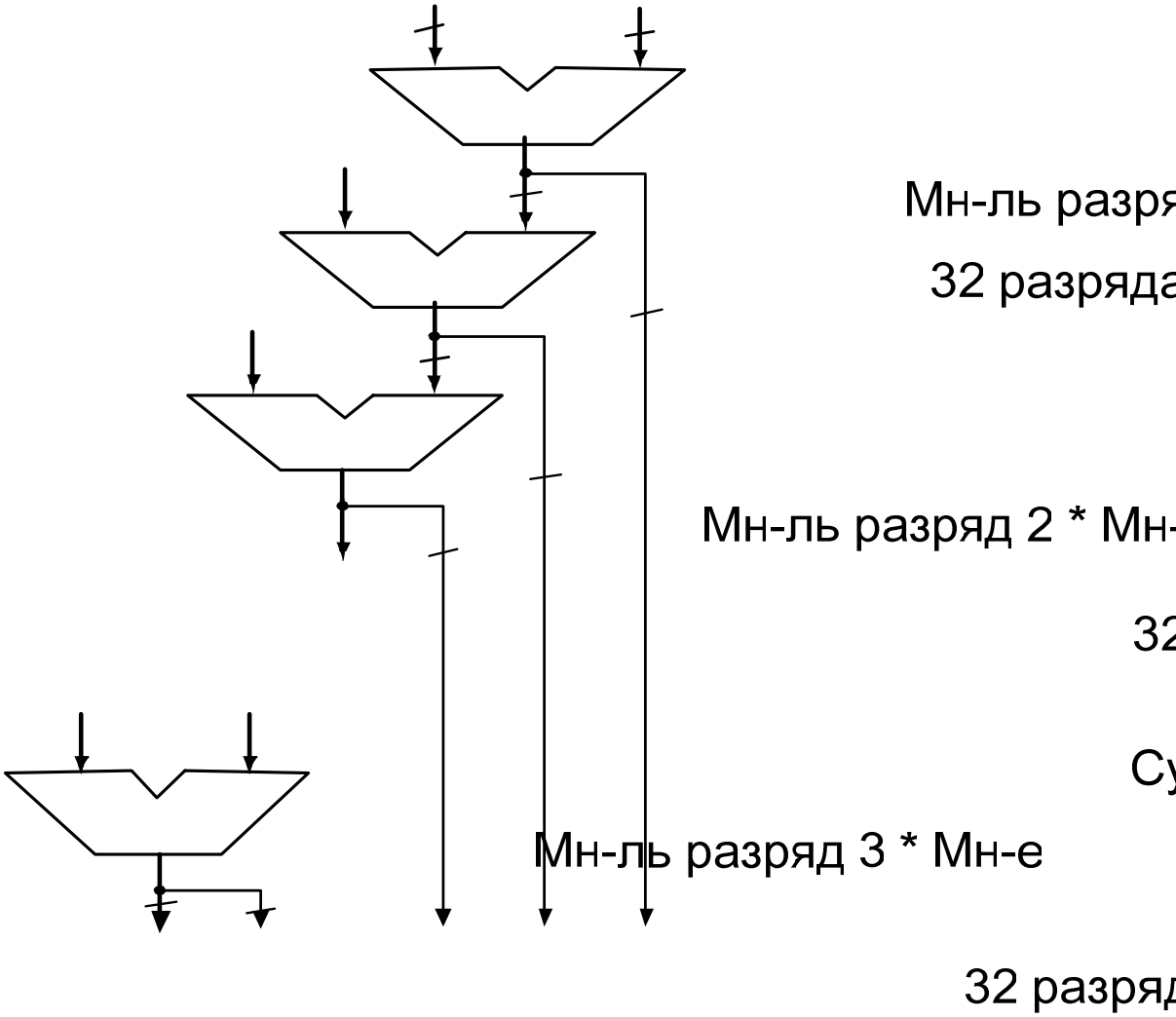


Рис.17. Упрощенная структура быстрого умножителя

Не сложно заметить, что вычисление произведения занимает довольно много времени – на каждом шаге необходимо произвести как минимум одно арифметическое действие и один сдвиг, в результате чего мы получаем значение только одного бита произведения. Существуют несколько способов ускорить это процесс.

Простейший способ – это не выполнять сложение, когда в очередном рязряде множителя A встречается нуль, а сразу производить сдвиг на один или несколько разрядов. Такой способ может ускорить вычисление произведения, но выигрыш по времени будет зависеть от распределения нулей в множителе A, что затрудняет использование этого метода в современных конвейерных процессорах, которые плохо работают с операциями с переменным временем исполнения или задержкой.

Другим способом ускорения вычисления произведения является использование специального сумматора. Как мы уже говорили в разделе о двоичном сложении, одной из проблем, возникающих при

вычислении суммы является вычисление значений переноса. К счастью, эта проблема может быть частично решена за счет регулярности данных в операции сложения – поскольку на каждом шаге мы складываем частичное произведение с одним и тем-же множителем, мы можем использовать результат вычисления переносов на предыдушем шаге для всех разрядов за исключеним последнего. Сохраняя значение переносов от шага к шагу, мы можем построить специальный многоуровневый сумматор (Рис.17), который состоит из линейки 32-разрядных сумматоров, младший разряд результата каждого сложения формирует значение разрядов результата. Безусловно, подобная структура будет очень медленно работать, если мы будем дожидаться полного распространения переноса на каждой ступени сложения. Здесь на выручку приходит возможность использования сумматоров с сохранением переноса, проиллюстрированная ранее на Рис.13б. Из таких сумматоров можно построить матричную структуру или дерево для быстрого выполнения всех промежуточных сложений, результаты которых представлены в избыточном кодировании. На Рис.18 проиллюстрировано использование сумматоров с сохранением переноса CSA для построения умножителя. Следует отметить, что на последнем уровне используется обычный сумматор с распространением переноса, чтобы получить результат в обычном, а не избыточном кодировании.

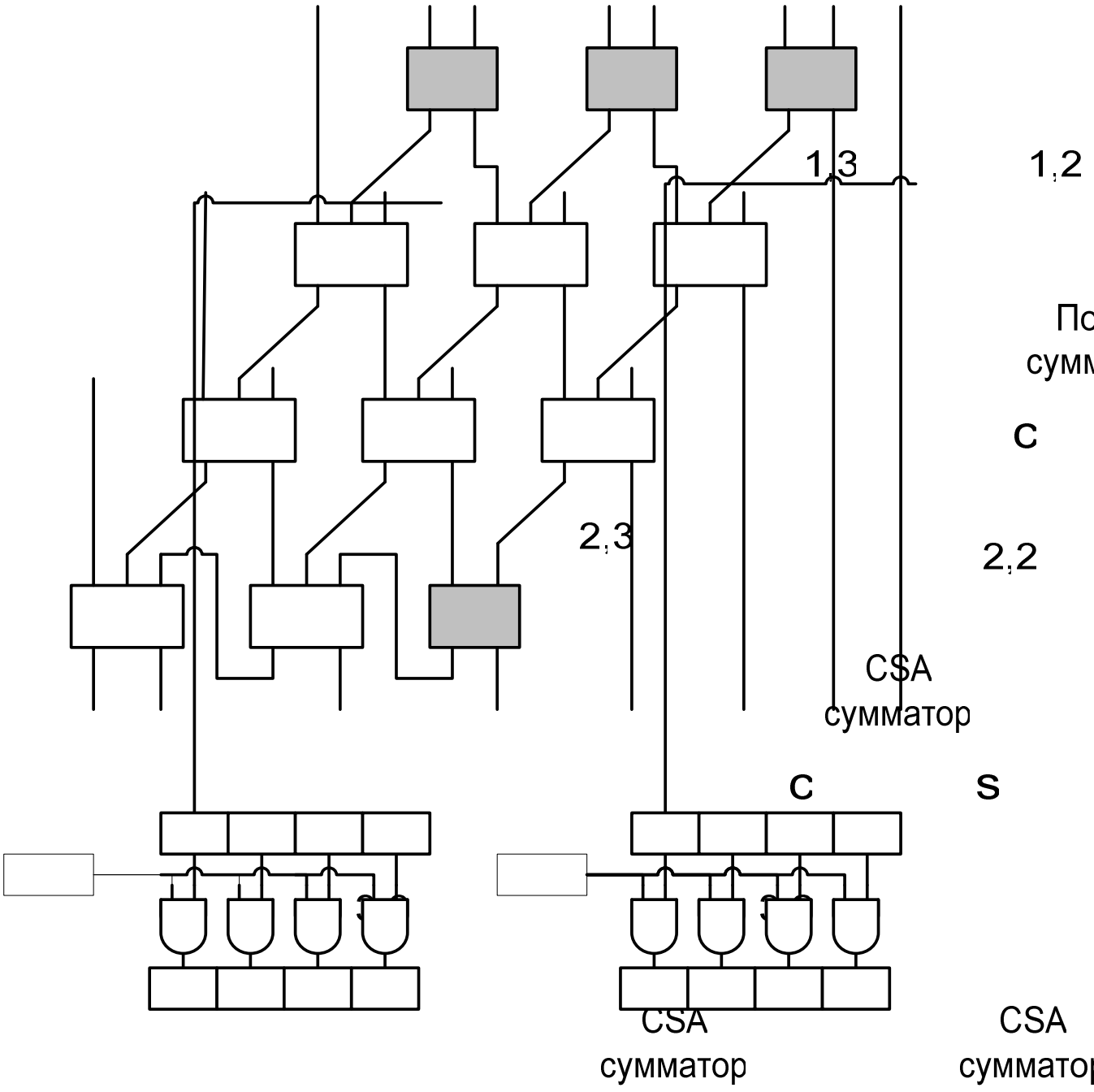


Рис.18. Схема использования сумматоров CSA для умножения .

Еще одним способом ускорения умножения является умножение на несколько разрядов. Например, при умножении на два разряда, если в этих разрядах находится значение 00, то можно просто выполнить сдвиг на два разряда, в случае если значение будет 01, то мы добавляем множитель к частичному произведению и сдвигаем, если значение 10, то мы можем сдвинуть множитель на один разряд влево перед тем как добавить его к произведению. А что происходит, если значение будет 11? Кажется, что в это случае надо будет вычислить значение 2b+b. К счастью, этого можно избежать, используя кодирование Бута с

основанием 2 (radix-2) – значение 11 записывается как 101^{-} , на текушем шаге вычитается значение множителя, а к значениям в следуюших двух разрядах добавляется единица.

Из примера видно, использование представления множителя в форме «знак-цифра» является одним из способов ускорения умножения, когда перекодирование множителя в формат «знак-цифра» Буфа позволяет ускорить процесс умножения.

Тот же подход может быть использован при умножении на три и большее число разрядов, если использовать кодирование Буфа с более широким основанием (radix-4, radix-8). Иллюстрация умножения методом Буфа двух 8-разрядных чисел представлена в Таблица 6.

Более детально использование алгоритма Буфа рассмотрено в секции, описывающей построение АЛУ с ПЗ SPARC64.

При условии наличия аппаратных ресурсов для нескольких сумматоров, можно еще больше ускорить процесс умножения, используя матрицу сумматоров, аналогичную представленной на Рис.18. Схема использования сумматоров CSA для умножения.

Таблица 6. Пример умножения методом Буфа двух восьмиразрядных двоичных чисел.

A множимое		0	1	0	1	0	1	0	1									85_{10}
X множитель	x	**0**	**0**	**0**	**0**	**1**	**0**	**1**	**1**									11_{10}
Y перекодировка		**0**	**0**	**0**	**1**	**1^{-}**	**1**	**0**	**1^{-}**									Код Буфа для множителя
Прибавить -A	+	1	0	1	0	1	0	1	1									**Вычитание A по 1^{-}**
Сдвиг		1	1	0	1	0	1	0	1	1								
Только сдвиг		1	1	1	0	1	0	1	0	1	1							**Сдвиг по 0**
Прибавить A	+	0	1	0	1	0	1	0	1									**Сложение по 1**
		0	0	1	1	1	1	1	1	1	1							**Результат**
Сдвиг		0	0	0	1	1	1	1	1	1	1	1						**Сдвиг частичного произведен**

																		ия
Прибавить -A	+	1	0	1	0	1	0	1	1									**Вычитание A по 1^-**
		1	1	0	0	1	0	1	0	1	1	1						**Результат**
Сдвиг		1	1	1	0	0	1	0	1	0	1	1	1					**Сдвиг произведения**
Прибавить A	+	0	1	0	1	0	1	0	1									**Сложение по 1**
		0	0	1	1	1	0	1	0	0	1	1	1					
Сдвиг		0	0	0	1	1	1	0	1	0	0	1	1	1				**Сдвиг произведения**
Только сдвиг		0	0	0	0	1	1	1	0	1	0	0	1	1	1			**Сдвиг по 0**
Только сдвиг		0	0	0	0	0	1	1	1	0	1	0	0	1	1	1		**Сдвиг по 0**
Только сдвиг		0	0	0	0	0	0	1	1	1	0	1	0	0	1	1	1	935_{10} **Сдвиг по 0**

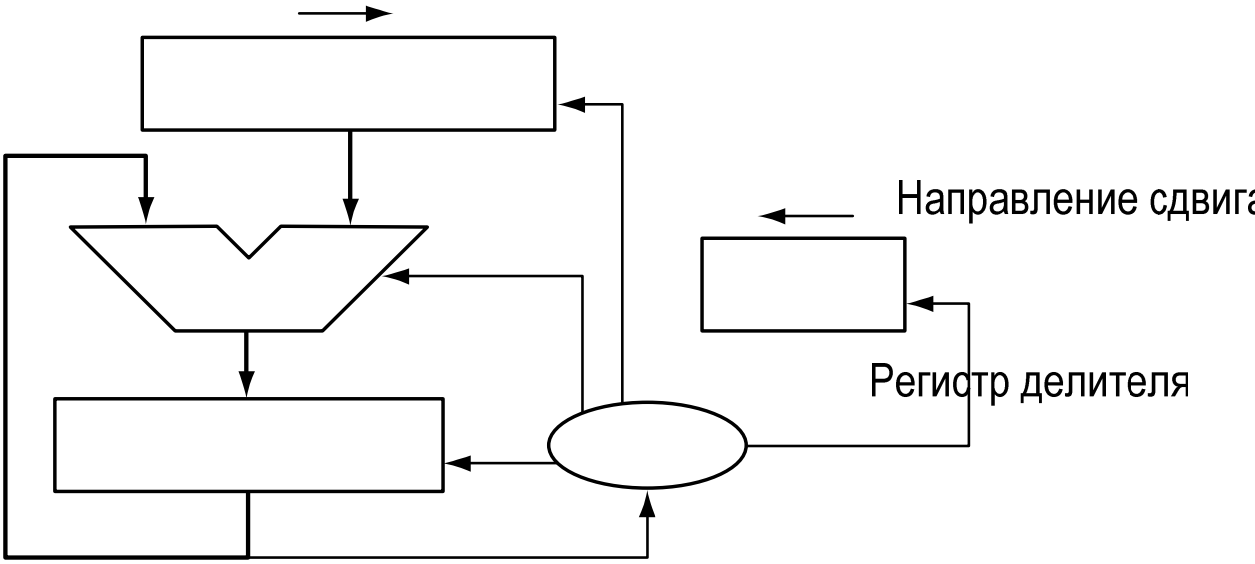
6 Деление двоичных чисел. Двоичные делители.

Похоже, что мы научились складывать и приумножать, что является положительной чертой характера инженера. А также выяснили, что вычитание в двоичной системе является тем же самым сложением, только с обратной величиной, представленной в дополнительном коде. Теперь попробуем научиться правильно делить, для чего придется опять вспомнить начальную школу. И все потому, что деление двоичных чисел не сильно отличается от деления десятичных: на каждом шаге делимое сравнивается с делителем и результат сравнения записывается в частное. Если делимое/остаток больше или равно значению делителя, то в соответствующий разряд частного записывается «1», в противном случае «0». Например:

```
(Делимое)  (Делитель)
 10101001 | 1001
-1001     +------| 1→0→0→1→0 (частное)
  001 положительный остаток (1 в разряд частного)
  0011 заем разряда делимого (0 в разряд частного)
```

```
  00110 заем разряда делимого (0 в разряд частного)
  001100 заем разряда делимого
   -1001 вычитание делителя
      11 положительный остаток (1 в разряд частного)
      111 заем разряда делимого(0 в разряд частного) конец операции, остаток
```

По аналогии с умножителем, представим схему устройства деления или делителя (в данном случае делитель – это устройство для выполнения операции деления, а не число, на которое мы будем делить делимое). Логично предположить, что если результат умножения требует регистра двойной длины, то и при делении нам тоже могут потребоваться регистры двойной длины (Рис. 19а). Начнем с того, что поместим делимое в младшие разряды регистра, в котором будет накапливаться остаток, делитель будет иметь собственный регистр, а регистр частного установим в ноль. На каждом шаге будет вычитаться значение в регистре делителя из регистра остатка. Если разница положительная, то мы сдвигаем регистр частного на один разряд и устанавливаем последний бит в единицу, если разница отрицательная, то мы восстанавливаем значение в регистре остатка, добавляя к нему значение из регистра делителя, регистр частного в этом случае сдвигается влево и значение в новом разряде устанавливается в ноль. Последней операцией на каждом шаге является сдвиг регистра делителя вправо на один разряд. После N шагов в регистре частного получаем результат, а в регистре остатка – остаток. Алгоритм деления в виде блок-диаграммы проиллюстрирован на Рис. 20.



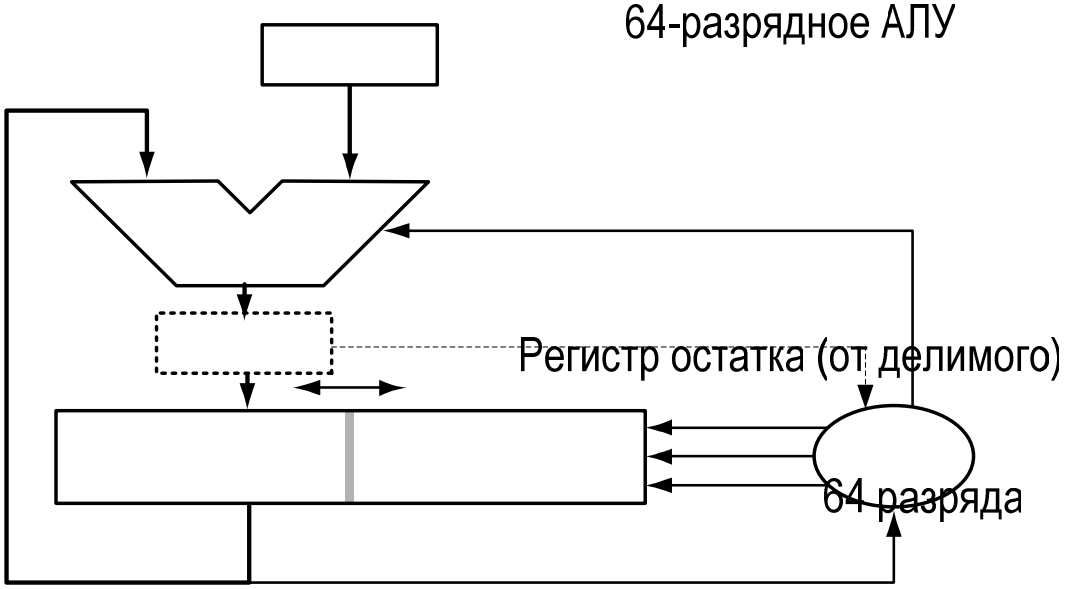


Рис. 19. Схемы простого и оптимизированного делителей

Не сложно заметить, что как и в случае с умножителями, только половина разрядов сумматора используется на каждом шаге. Если вместо сдвига регистра делителя вправо на каждом шаге выполнять сдвиг регистра остатка влево, то можно обойтись сумматором с меньшим числом разрядов.

Кроме того, можно предположить, что на первом шаге мы никогда не можем получить единицу – в этом случае частное будет слишком большим, чтобы поместиться в отведенное ему число разрядов. Исходя из этого, можно изменить порядок операций: вначале выполнять сдвиг, а потом вычитание. За счет этого можно уменьшить количество итераций на одну.

Кроме того, совсем не обязательно выделять отдельный регистр для частного. Если все операнды деления имеют одинаковое число разрядов, то можно использовать тот же трюк, что и при операции умножения, и поместить частное вместе с остатком том же самом регистре: в конце операции старшие разряды регистра остатка будут содержать значение остатка, а младшие – частного (Рис. 19б).

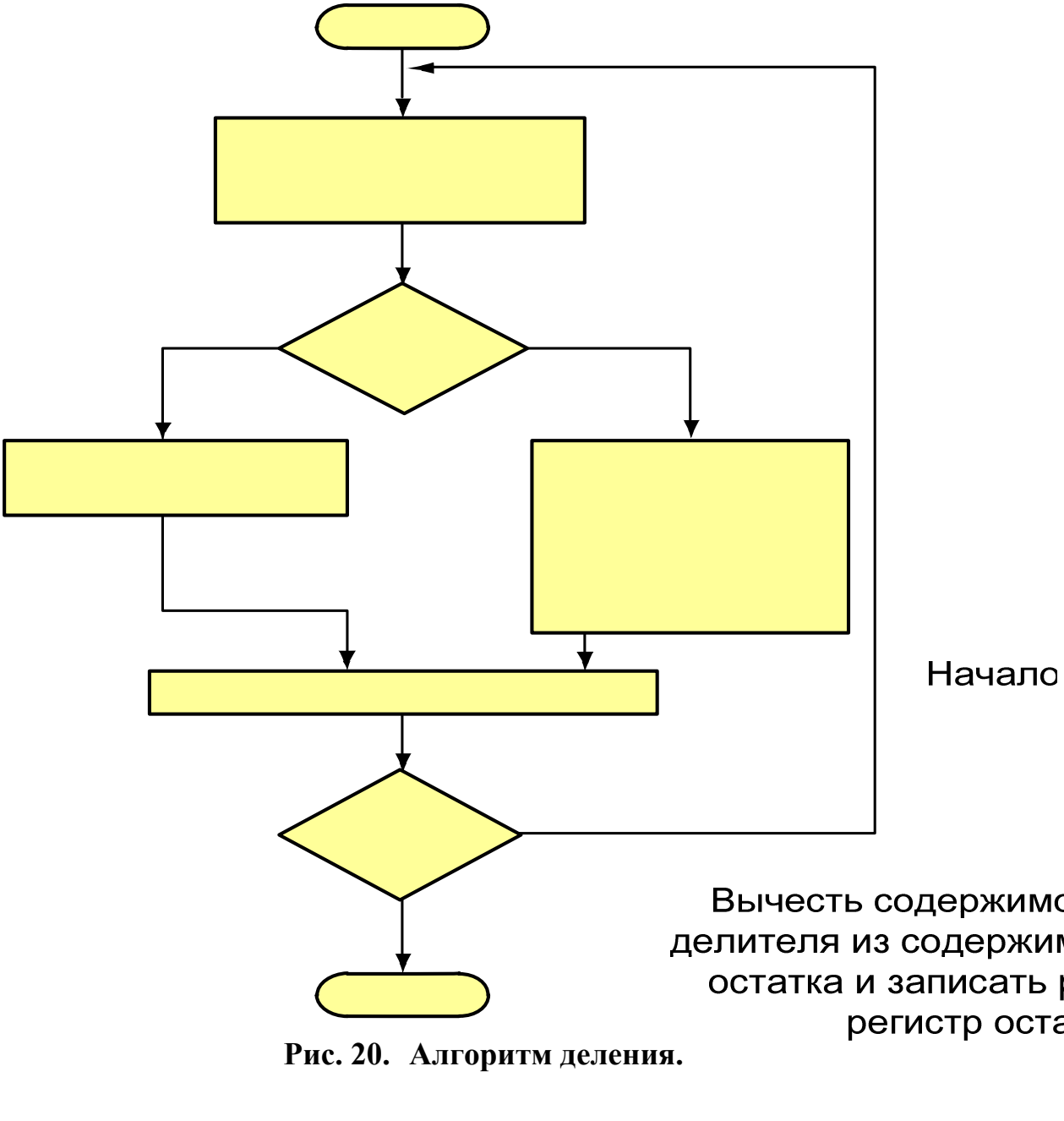


Рис. 20. Алгоритм деления.

Существует модификация алгоритма деления, которая работает без восстановления значения остатка. В этом алгоритме на каждом шаге регистр остатка сдвигается влево, после чего если до сдвига значение в регистре остатка было отрицательное, то значение в регистре делителя складывается со значением в регистре остатка, в противном случае оно вычитается из значения регистра остатка. Если в результате операции значение в регистре остатка отрицательное, то младший бит этого регистра устанавливается в ноль, иначе – в единицу. Если после завершения работы алгоритма в регистре остатка значение по-прежнему отрицательное, то для вычисления истинного остатка нужно произвести восстановление, добавив к нему значение делителя. Следует особенно отметить тот факт, что решение о том, какая операция будет выполняться на каждом шаге, должно быть принято до сдвига регистра остатка, посколько при сдвиге знаковый разряд может быть утерян.

Казалось бы, если существует алгоритм, который не требует восстановления остатка на каждом шаге, зачем использовать алгоритм с восстановлением, который требует дополнительной операции сложения? На самом деле, добавив промежуточный регистр на выходе сумматора, можно заменить восстановление остатка на проверку знака и не копировать результат в регистр остатка, если результат в промежуточном регистре меньше нуля.

Также следует заметить, что перед тем, как начать деление, необходимо проверить, что делитель не равен нулю – алгоритм в этом случае выдаст какой-то результат, но математики могут не согласиться с ним.

Рассмотренные выше алгоритмы позволяют получить частное и остаток при делении двух натуральных чисел. А что произойдет, если надо вычислить частное от деления двух чисел со знаком? Для умножения целых чисел существует специальный алгоритм, который позволяет вычислить произведение двух чисел, записанных в дополнительном коде, и и также автоматически вычисляет знак произведения – для деления не используют специальный алгоритм: операция деления, в общем, реже встречается при расчетах и собственно операция занимает больше времени, поэтому при делении чисел в дополнительном коде делимое и делитель переводят в прямой код, вычисляют частное и остаток, после чего частное может быть обратно переведено в дополнительный код.

Одной из особенностей целочисленного деления является неопределенность результата в случае когда делимое или делитель меньше нуля. Очевидно, что если D – это результат от деления X на Y, а M – остаток от деления, то должно выполняться правило X = D * Y +

M вне зависимости от знака X и Y. Но если X равен –8, а Y – +3, то существует два значения для D и M, удовлетворяющих приведенной выше формуле: D может быть равно –3 или –2, а M = +1 или -2 соответственно. Для избежания неопределенности, обычно считают, что частное всегда будет иметь наименьшее абсолютное значение, а остаток будет вычисляться в соответствии со значением частного, при этом знак остатка всегда будет совпадать со знаком делимого. Такое соглашение, во-первых, означает, что выполняется правило дистрибутивности при умножении, т.е. (-X)/Y = – (X/Y), а во-вторых, частное может быть вычислено по вышеприведенному алгоритму. Недостатком такого подхода является то, что значение остатка может быть как положительным, так и отрицательным числом, т.е. операция деления по модулю не может быть использована для нахождения индекса в таблице, если делимое меньше нуля. Также при пересчете системы координат возникают искажения в районе нуля, связанные с тем, что несколько значений в одной системе отображаются на ноль в другой.

Можно было бы использовать для разрешения конфликта соглашения, принятые в языках программирования высокого уровня, но и там нет единства мнений – в Таблица 7 приведены примеры некоторых языков программирования и результатов деления:

Таблица 7. Деление в разных языках программирования.

Язык программирования	Частное	Остаток
FORTRAN	-8/3 = -2	MOD(-8, 3) = -2
Pascal	-8 div 3 = -2	-8 mod 3 = 1
Ada	-8 / 3 = -2	-8 MOD 3 = 1
C	-8 / 3 = неопределено	-8 % 3 = неопределено
Modula-3	-8 div 3 = -3	-8 mod 3 = 1

Процесс двоичного деления может быть ускорен за счет использования избыточного кодирования остатка и сумматора, с сохранением переноса по аналогии с устройствами умножения, Рис. 21. Сумматор CSA позволяет быстро производить последовательные вычитания или сложения, причем выбор значения бита частного Q-j и выбор операции (нет операции, сложение или вычитание) на следующем шаге производится специальной схемой, которая анализирует значение суммы старших четырех битов из регистров суммы U и переноса V. Эта схема управляет мультиплексорами, через которые подается операнд на один из входов сумматора. На второй вход подаются значения суммы и переноса регистров остатка.

Подразумевается, что значение остатка (как сумма так и перенос) представлено в дополнительном коде.

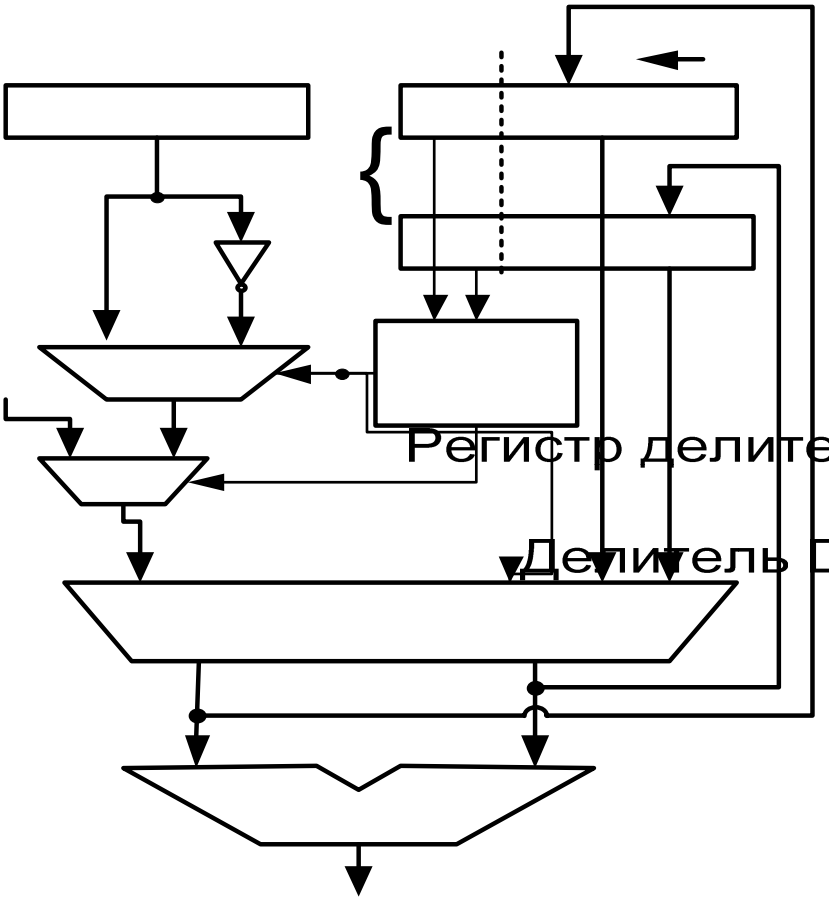


Рис. 21. Структура быстрого делителя

Этот быстрый делитель построен с использованием алгоритма деления SRT (разработчики Sweeney, Robertson, Tocher). Особенностью этого алгоритма является сильное упрощение сравнения значения остатка с делителем для выбора значения бита частного. Вместо сравнения полного значения делителя и остатка, сравниваются значения лишь нескольких старших значащих разрядов частного. Причем сравнение производится с константами ½ и – ½. За счет использования простой логики или таблиц для такого сравнения вместо полномасштабной операции вычитания, алгоритм деления SRT работает значительно быстрее.

Структура быстрого делителя достаточно проста за исключением правил выбора бита частного и следующей операции. Ведь традиционный алгоритм деления на Рис. 20 предусматривает обычное двоичное кодирование остатка и проверку его знака. В структуре быстрого делителя промежуточные значения остатка представлены в форме избыточного кодирования и надо определить действия исходя

из содержания уже двух регистров. Классическим решением является использование значений 4 старших разрядов из каждого регистра U и V, которые представлены в избыточном кодировании в дополнительном коде с диапазоном значений [-2, -1, 0, 1].

Если сумма этих старших четырех разрядов меньше -1/2, то Q_{-j} = -1 (производится восстановление остатка прибавлением делителя D, а в частное записывается 0), в противном случае значение суммы сравнивается с нулем. Если сумма больше или равна нулю, то Q_{-j} = 1 (производится вычитание делителя, в частное записывается 1), в противном случае в частное записывается 0 и операция сложения производится с нулевым значением делителя D.

7 Команды целочисленной арифметики в процессорах MIPS и ARM

7.1 Команды целочисленной арифметики в процессорах MIPS

Процессоры MIPS имеют достаточно простой набор команд целочисленной арифметики, включающий четыре основные арифетические операции: сложение, вычитание, умножение и деление. Все операции выполняются только над 32-разрядными словами, причем константы из кода операции расширяются до 32 бит. Имеются версии команд с учетом знака и беззнаковые команды. Кроме того, имеется набор логических команд, обеспечивающих основные логические функции на поразрядной основе, а также команды сдвига как логического (без учета знака), так и арифметического (с учетом знака). К этой же категории можно отнести команды сравнения двух величин и установки флага по результату сравнения, которые используются в паре с командами условных переходов. В Таблица 8 приведены три базовых формата R, I и J целочисленных команд ядра MIPS, которые содержат поля кода операции (КОП), поля регистров операндов S и T, поле регистра результата, поле числа позиций сдвига Shamt, поле функции (расширения кода операции), поле непосредственной константы и поле адреса (для команд перехода).

Таблица 8. Формат команды MIPS

Тип	**-31-**		**Формат**			**-0-**
R	opcode (6) КОП	Rs (5) Регистр операнда S	Rt (5) Регистр операнда T	Rd (5) Регистр результата	shamt (5) Управление сдвигом	funct (6) Функция расширения кода операции

I	opcode (6) КОП	Rs (5) Операнд S	Rt (5) Операнд T	immediate (16) Непосредственная константа
J	opcode (6) КОП	address (26) Адрес		

Команды целочисленной арифметики используют формат R для регистровых команд, и формат I для команд с непосредственным операндом (константой) в теле команды. В Таблица 8 формат команды указан как тип. Большинство целочисленных арифметических команд имеют версии регистр-регистр и регистр-константа (за исключением операций умножения). В Таблица 9 представлены основные команды целочисленной арифметики.

Таблица 9. Команды целочисленной арифметики MIPS

Команда	**Синтаксис**	**Действие**	**Тип /Код-Оп /Код-функ**			**Комментарий**
Сложение	add Rd,Rs,Rt	Rd = Rs + Rt (со знаком)	R	0	20_{16}	Суммирует содержимое двух регистров
Сложение без знака	addu Rd,Rs,Rt	Rd = Rs + Rt (без знака)	R	0	21_{16}	Суммирует содержимое двух регистров без учета знака
Вычитание	sub Rd,Rs,Rt	Rd = Rs - Rt (со знаком)	R	0	22_{16}	Вычитает содержимое двух регистров
Вычитание без знака	sub Rd,Rs,Rt	Rd = Rs - Rt (без знака)	R	0	23_{16}	Вычитает содержимое двух регистров без учета знака
Сложение с константой	addi Rd,Rs,CONST	Rd = Rs + >>CONST (со знаком)	I	8_{16}	Нет	Сложение регистра с расширенной до 32 бит константой из кода команды, может использоваться для копирования из одно регистра в другой"addi Rd, Rs, 0"
Сложение с константой без знака	addiu Rt,Rs,CONST	Rt = Rs + >>CONST (без знака)	I	9_{16}	Нет	Сложение регистра с расширенной до 32 бит константой из кода команды без учета знака
Умножение	mult Rs, Rt	LO = ((Rs * Rt) <<	R	0_{16}	18_{16}	Умножение содержимого двух регистровс

		32) >> 32; HI = (Rs * Rt) >> 32;				размещением 64-бит результата в двух специальных регистрах LOW and HI. (int HI,int LO) = (64-bit) $1 * $2 .
Умножение без знака	multu Rs,Rt	LO = ((Rs * Rt) << 32) >> 32; HI = (Rs * Rt) >> 32;	R	0_{16}	19_{16}	Умножение содержимого двух регистровс размещением 64-бит результата в двух специальных регистрах LOW and HI. (int HI,int LO) = (64-bit) $1 * $2 .
Деление	div Rs, Rt	LO = Rs / Rt HI = Rs % Rt	R	0_{16}	$1A_{16}$	Делит содержимое двух регистров и размещает частное 32-бит частное в регистр LOW и остаток в HI
Деление без знака	divu Rs, Rt	LO = Rs / Rt HI = Rs % Rt	R	0_{16}	$1B_{16}$	Делит содержимое двух регистров и размещает частное 32-бит частное в регистр LOW и остаток в HI

Логические операции также могут иметь формат R и I в зависимости от типа операнда – регистрового или константы сотвественно. Все логические операции поразрядные и результат определяется индивидуально в каждом разряде. Операции сравнения на самом деле являются операцией вычитания с установкой флага по комбинации знака результата и нулевого флага. Команды сдвига используют поле SHAMT для управления сдвигом. В Таблица 10 приведены основные команды этой группы.

Таблица 10. Команды сдвига, сравнения и логические операции MIPS

Команда	Синтаксис	Действие	Тип	Код-Оп	Код-функ	Комментарий
And	and Rd,Rs,Rt	Rd = Rs & Rt	R	0_{16}	24_{16}	Поразрядная операция И (регистры)
And Immediate	andi Rt,Rs, CONST	Rt = Rs & CONST	I	C_{16}	Нет	Поразрядная операция И (регистр-расширенная до 32-бит константа)
Or	or Rd,Rs,Rt	Rd = Rs | Rt	R	0_{16}	25_{16}	Поразрядная операция ИЛИ (регистры)

Or Immediate	ori Rt,Rs, CONST	Rt = Rs | >>CONST	I	D_{16}	Нет	Поразрядная операция ИЛИ (регистр-расширенная до 32-бит константа)
Exclusive OR	xor Rd,Rs,Rt	Rd = Rs ^ Rt	R	11_{16}	26_{16}	Поразрядная операция Исключающее ИЛИ (регистры)
Nor	nor Rd,Rs,Rt	Rd = ~(Rs | Rt)	R	0_{16}	27_{16}	Поразрядная операция Инверсное ИЛИ (регистры)
Set on Less Than	slt Rd,Rs,Rt	Rd = (Rs < Rt)	R	0_{16}	$2A_{16}$	Сравнение содержимого регистров с установкой флага если меньше
Set on Less Than Immediate	slti Rt,Rs, CONST	Rt = (Rs < CONST)	I	A_{16}	Нет	Сравнение содержимого регистра и константы с установкой флага если меньше
Set on Less Than Immediate Unsigned	sltiu Rt,Rs, CONST	Rt = (Rs < CONST)	I	B_{16}	Нет	Сравнение без знака содержимого регистра и константы с установкой флага если меньше
Set on Less Than Unsigned	sltu Rd,Rs,Rt	Rd = (Rs < Rt)	R	0_{16}	$2B_{16}$	Сравнение содержимого регистров с установкой флага если меньше
Shift Left Logical	sll Rt,Rs, CONST	Rt = Rs << SHAMT	R	0_{16}	00_{16}	Сдвиг на Shamt разрядов влево (эквивалентно умножению на 2^{SHAMT})
Shift Right Logical	srl Rd,Rs, SHAMT	Rd = Rs >> SHAMT	R	0_{16}	02_{16}	Сдвиг на Shamt разрядов вправо, в старшие разряды вдвигаются нули (эквивалентно делению положительных чисел на 2^{SHAMT}). Деление можно выполнять только для положительных чисел в случае операнда в дополнительном коде.
Shift Right Arithmetic	sra Rd,Rs, SHAMT	Rd = Rs >> SHAMT	R	0_{16}	03_{16}	Сдвиг на Shamt разрядов вправо (эквивалентно делению числа в дополнительном коде на 2^{SHAMT}), знаковый разряд вдвигается в освободившиеся биты

Команды загрузки данных из памяти в регистры в процессоре MIPS должны предшествовать командам обработки, команды сохранения данных в памяти должны следовать за командами обработки, чтобы освободить регистры для новых данных. В Таблица 11 приведены основные команды пересылки данных.

Таблица 11. Команды пересылки данных MIPS

Команда	Синтаксис	Действие	Тип	Код-Оп	Код-функ	Комментарий
Load word	lw Rt,CONST(Rs)	Rt = Memory[Rs + CONST]	I	11_{16}	23_{16}	Загружает из памяти в регистр 32-разрядное слово начиная с байта по адресу (Rs+CONST) и последующие три байта
Load halfword	lh Rt,CONST(Rs)	Rt = Memory[Rs + CONST]	I	11_{16}	21_{16}	Загружает из памяти в регистр 16-разрядное слово начиная с байта по адресу (Rs+CONST) и последующий байт
Load byte	lb Rt,CONST(Rs)	Rt = Memory[Rs + CONST]	I	11_{16}	20_{16}	Загружает из памяти в регистр 8-разрядный байт по адресу (Rs+CONST)
Каждая команда загрузки имеет также вариант загрузки без знака (unsigned)- lbu, lhu, lwr						
Store word	sw Rt,CONST(Rs)	Memory[Rs + CONST] = Rt	I	11_{16}	$2B_{16}$	Сохраняет из регистра в память 32-разрядное слово начиная с байта по адресу (Rs+CONST) и последующие три байта
Store half	sh Rt,CONST(Rs)	Memory[Rs + CONST] = Rt	I	11_{16}	29_{16}	Сохраняет из младшей половины регистра в память 16-разрядное слово начиная с байта по адресу (Rs+CONST) и последующий байт
Store byte	sb Rt,CONST(Rs)	Memory[Rs + CONST] = Rt	I	11_{16}	28_{16}	Сохраняет из младшей четверти регистра в память 8-разрядный байт по адресу (Rs+CONST)
Load upper immediate	lui Rt,CONST	Rt = CONST * 2^{16}	I	F_{16}	Нет	Загружает из поля команды в старшую половину регистра 16-

						разрядную константу Максимальное значение константы равно 2^{16}
Move from high	mfhi Rd	Rd = HI	R	11_{16}	10_{16}	Перезагружает регистр значением только старшей половины регистра. Не рекомендуется использовать команды умножения и деления между такими командами.
Move from low	mflo Rd	Rd = LO	R	11_{16}	12_{16}	Перезагружает регистр значением только младшей половины регистра. Не рекомендуется использовать команды умножения и деления между такими командами.

7.2 Команды целочисленной арифметики в процессорах ARM

Процессоры семейства ARM имеют очень компактную и мощную систему команд, и пользуются огромной популярностью во встроенных системах. Все команды являются условными или предикативными, они могут быть выполнены или пропущены как пустая операция NOP в зависимости от значения флага (предиката), указанного в команде. В Таблица 12 приведены основные арифметические команды.

Таблица 12. Арифметические команды в процессорах ARM.

Команда	Синтаксис	Действие	Изменяемые флаги	Комментарий
Add	ADD{cond}{S} Rd, Rn, <Opnd2>	Rd := Rn + Opnd2	N Z C V	Сложение двух операндов
Add with carry	ADC{cond}{S} Rd, Rn, <Opnd2>	Rd := Rn + Opnd2 + Carry	N Z C V	Сложение двух операндов и флага переноса
Subtract	SUB{cond}{S} Rd, Rn, <Opnd2>	Rd := Rn – Opnd2	N Z C V	Вычитание двух операндов
Subtract with	SBC{cond}{S} Rd, Rn,	Rd := Rn –	N Z C	Вычитание двух

carry	<Operand2>	Operand2 – NOT(Carry)	V	операндов и флага переноса
Reverse subtract	RSB{cond}{S} Rd, Rn, <Operand2>	Rd := Operand2 – Rn	N Z C V	Реверсивное вычитание двух операндов
Reverse subtract with carry	RSC{cond}{S} Rd, Rn, <Operand2>	Rd := Operand2 – Rn – NOT(Carry)	N Z C V	Реверсивное вычитание двух операндов и флага переноса
Multiply	MUL{cond}{S} Rd, Rm, Rs	Rd := (Rm * Rs)[31:0]	N Z C*	Умножение двух операндов
Multiply and accumulate	MLA{cond}{S} Rd, Rm, Rs, Rn	Rd := ((Rm * Rs) + Rn)[31:0]	N Z C*	Умножение дввух операндов и сложение с третьим

8 Операции над вещественными числами (плавающая запятая)

Вначале этой главы мы уже отметили, что имея 64-разрядный процессор, можно производить вычисления над числами в диапазоне от 0 до 18 квинтиллионов – казалось бы, этого должно быть достаточно для кого угодно. Кроме того, если предположить, что значения младших разрядов представляют некоторую дробную часть, то можно производить вычисления не только над целыми, но и над вещественными числами в довольно большом диапазоне: выделив 8 двоичных разрядов под дробную часть мы можем представить два десятичных знака после десятичной запятой и при этом максимальное число уменьшиться с 18 квинтиллионов до «каких-то» 70 квадриллионов.

Но и этого оказывается недостаточным: во-первых, большинство процессоров, находящихся в эксплуатации, являются 32-битовыми, так что если мы отведем 8 разрядов под дробную часть, то нам останется только 24 разряда для целой части – какие-то жалкие 16 миллионов, до некоторого времени бюджет приличной итальянской семьи не мог быть подсчитан с использованием такого процессора; во-вторых, два десятичных знака после запятой могут быть достаточны для проведения бухгалтерских расчетов, но вряд-ли удовлетворят математиков и физиков. Поэтому наряду с целыми числами, в двоичной арифметике существует еще один набор чисел, называемый числами с плавающей запятой. Математически правильно такие числа называются вещественными.

Числа с плавающей запятой состоят из двух частей: мантиссы и порядка, называемого также экспонентой. Мантисса используется для представления дробной части вещественного числа, а порядок – для представление степени двойки, таким образом число с плавающей точкой может быть записано как $1.5 \times 2^{-3} = 0.1875$.

Знак числа	Порядок (степень множителя) 2^{-3}	Мантисса (дробная часть числа) 0.1875

Точно также, как при инженерной записи в десятичной форме, если перед запятой находится только одна цифра, отличная от нуля, то такая запись называется нормализованной, например, 3.15576×10^{9} – это нормализованная запись, а 0.315576×10^{10} или 31.5576×10^{8} – нет. В отличии от десятичной системы счисления, в двоичной системе нормализованное число с плавающей запятой всегда будет иметь перед запятой единицу и только ее. Рачительные разработчики аппаратного обеспечения используют этот факт для того, чтобы сэкономить один разряд и не хранить значение единицы до запятой, а подразумевать ее по умолчанию. Эту единицу также часто называют скрытым битом.

Общий подход к представлению вещественных чисел с использованием формата с плавающей запятой довольно давно стал использоваться различными производителями аппаратного обеспечения. Но при этом в каждой из систем имелись свои особенности: разное число бит, отводимых для мантиссы и порядка, разные значения для основания порядка, разные способы округления, способы обработки переполнения и вырождения результата, способы представления денормализованных значений и тому подобные «мелочи». Число таких различий было настолько велико, что стали возникать проблемы при переносе программого обеспечения с одной платформы на другую. Для решения этих проблем IEEE в 1985 году принял стандарт IEEE 754 (IEC 559), который оговаривал правила для представления числе с плаваюшей запятой и работы с ними. В 2008 году принята новая версия стандарта IEEE 754-2008, оговаривающая особенности исполнения некоторых арифметических операций. На Рис. 22 представлены основные форматы представления вещественных чисел в этом формате и представление аномалий, возникающих при вычислениях.

Рис. 22. Форматы представления вещественных чисел и аномалий в стандарте IEEE-754.

Аномальные числа в фо

Основными особенностями этого стандарта являются:

Денормализованое число (denorm)

1. Наличие специальных кодов для представления $-\infty$, $+\infty$ и NaN (Not-a-Number - « НеЧисло»).
2. Использование денормализированных чисел (denorm) для представления результата, меньшего чем $1.0x2^{Emin}$.
3. По умолчанию, округление производится к ближайшему значению, которое может быть представлено в отведенном количестве разрядов для мантиссы.

Положительная или отрицательная бесконечность (infinity)

4. При округлении половинного результата до ближайщего вещественного числа выбирается четное значение: например, если при округлении результата умножения 6.2 на 0.25 = 1.55 до первого десятичного знака после запятой результат будет 1.6, а не 1.5 потому что 6 – четное.
5. Кроме округления в ближайшему значению, есть еще три других способа округления: к нулю, к отрицательной и к положительной бесконечности.

НЕ число (NaN – Not a Number)

6. Использование NaN для обработки особенных ситуаций.

Использование NaN (Not a Number – «не число») для обработки особенных ситуаций является одной из наиболее важных особенностей стандарта IEEE-754, поскольку это позволяет продолжать вычисления при возникновении непредвиденной ситуации, как, например, деления на ноль или попытки извлечь квадратный корень из отрицательного числа – в этом случае результат вычисления принимает специальное значение, которое не может быть использовано для представления обычного числа с плавающей запятой. Результат любой операции, в которой NaN является одним из операндов, тоже будет NaN, таким образом при реализации сложного математического выражения программисту не надо проверять результат на каждом шаге, достаточно проверить результат всего выражения.

NaN может быть результатом многих операций, например $\sqrt{-1}$ или N/0, но сушествует несколько операций, результат которых в стандарте IEEE-754 имеет специальное значение, например $1/0 = \infty$ и, соответсвенно, $1/\infty = 0$. Значения для $-\infty$ и $+\infty$ могут использоваться при вычислении наряду с обычными значениям вещественных чисел.

Для представления чисел, которые меньше, чем 1.0×2^{Emin}, используется специальное представление, которое позволяет избежать вырождения результата при некоторых вычислениях. В системе, которая жестко придерживается правил нормализации при записи, если результат вычислений не имеет достаточно значимых двоичных цифр, чтобы его можно было бы представить как произведение минимальной степени двойки, то такой результат принимает значение нуль. Если это значение является конечным значением некоторых вычислений, то потеря не велика, а вот если такая ситуация возникает в середине цепочки операций, то значение конечного результата становится непредсказуемым – это может быть нуль, одна из бесконечностей или NaN. В системе с жесткой нормализацией необходимо обеспечить оповещение о том, что произошло вырождение результата. В стандарте IEEE-754 вместо этого результат может быть представленен как денормализированное число, что позволит продолжать вычисления до тех пор, пока результат не станет настолько близок к нулю, что он не может быть представлен и в денормализированном виде.

Для представления вещественных числе в стандарте IEEE-754 используется несколько форм записи, обеспечивающих разный уровень точности и диапазон (см. Таблица 13).

Таблица 13. Форматы записи чисел с плавающей точкой

	Одинарная	**Одинарная расширенная**	**Двойная**	**Двойная расширенная**
Число бит в	24	≥ 32	53	≥ 64

мантиссе				
Максимальное значение порядка	127	≥1023	1023	≥ 16383
Минимальное значение порядка	-126	≤-1022	-1022	≤ -16382
Смещение порядка	127	Не стандартизовано	1023	Не стандартизовано

При записи с одинарной точностью (см. Рис. 22) 23 бита отводятся для представления значения мантиссы, 8 бит для порядка и один бит для знака, таким образом общее число бит составляет 32. При записи с двойной точностью число бит для представления мантиссы увеличивается до 52, а число бит для порядка – до 11, общее число бит становиться разным 64. В обоих случаях для записи порядка используется код со смещением, значение смешения равно половине максимального значения, представимого в отведенном числе разрядов (127 в формате с одинарной точностью, 1023 - с двойной). Мантисса записывается с использованием прямого кода.

Как видно из Рис. 22 значение порядка располагается между знаковым битом и мантиссой. Вместе с использованием кода со смещением для представления порядка позволяет сортировать вещественные числа с использованием обычного компаратора для целых чисел.

Как мы уже отметили, использование нормализированного представления для всех, кроме особенных случаев, позволяет избежать выделения отдельного разряда для значимой единицы перед запятой, и таким образом, обеспечивается точность вычислением в 24 бита в формате с одинарной точностью. При проведении самих вычислений значимая единица явно добавляется как старший разряд к операнду – такой формат называется распакованным представлением чисел с плавающей запятой.

Выделив 8 бит для представления порядка, мы можем представить значения в диапазоне от 0 до 255 или, учитывая смещение на 127, в диапазоне от –127 до 128. Реально битовые последовательности из всех нулей и всех единиц в поле порядка используются для кодирование специальных значений – когда в поле порядка находиться значение 255, то нулевое значение мантиссы используется для представления бесконечности, а ненулевое – для представления NaN, т.е. существуют положительная и отрицательная бесконечность и группа значений, представляющих NaN. Значение 0 в

поле порядка используется для представления нуля и денормализованных значений: если мантисса имеет значение 0, то это число 0.0, а если мантисса имеет значение, отличное от нуля, то это денормализованное число, значение которого будет выражаться формулой 0.f x 2^{Emin}. С учетом этих ограничений формат с одинарной точностью может быть использован для представления чисел с плавающей точкой в диапазоне от 2^{-126} до 2^{127}.

В начале этой главы мы упоминали, что при представлении чисел в прямом коде сушествует два представления для числа 0. Чтобы избежать недоразумений, в стандарте IEEE-754 число 0 представляется как битовая последовательность из всех нулей, «отрицательный ноль» не является запрещенной битовой последовательностью, а представлен единицей в знаковом разряде и последовательностью нулей в поле порядка и мантиссы.

9 Операции с плавающей запятой в стандарте IEEE-754

9.1 Сложение чисел с плавающей запятой

Использование отдельных значений для мантиссы и порядка для представления значения чисел с плавающей запятой приводит к необходимости выполнения операции сложения в несколько этапов: поскольку мы не можем складывать числа с разными значениями порядка непосредственно, на первом этапе необходимо произвести операцию выравнивания порядков, для чего число с плавающей точкой переводится в распакованную форму, после чего мантисса числа с меньшим значением порядка сдвигается вправо на столько разрядов, какова разность между порядками двух чисел. После этого производится сама операция сложения над мантиссами. Далее необходима нормализация результата. После проверки на переполнение производится округление и повторная нормализация, если в этом возникла необходимость. Общая структура алгоритма сложения чисел с плавающей запятой представлена на Рис. 23. При использовании дополнительного кода аналогичный алгоритм используется и для вычитания.

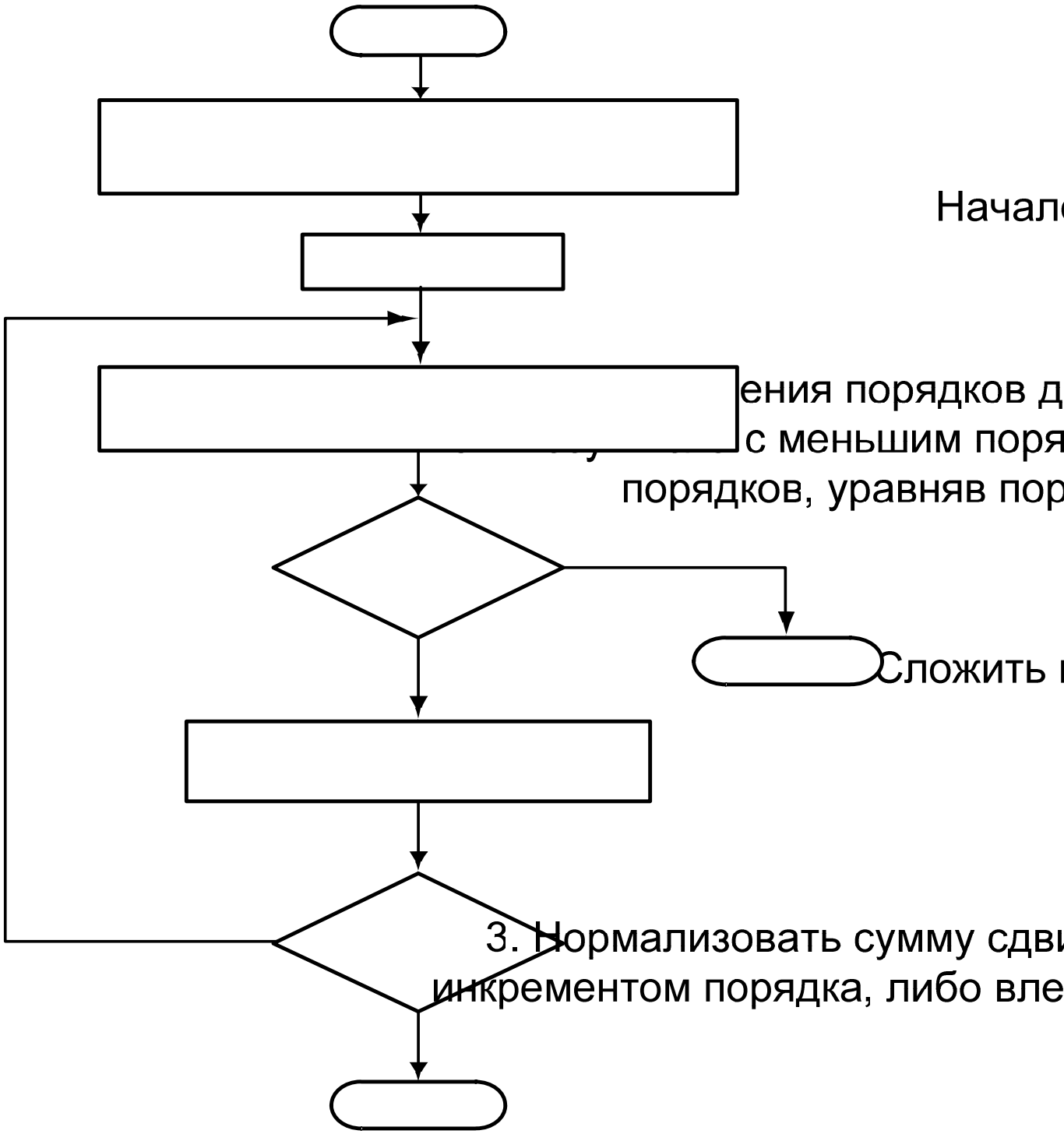


Рис. 23. Алгоритм сложения чисел с плавающей запятой.

Очевидно, что при выполнении выравнивания порядков и сдвига мантиссы одного из операндов вправо, необходимое число разрядов для представления числа с меньшим значением порядка увеличивается. Значит - ли это, что для вычисления правильного значения суммы необходим сумматор большей разрядности? Совсем не обязательно. Например, при вычислении суммы двух чисел с шестиразрядной

мантиссой, имеющих значения 1.10101_2x2^0 и 1.10001_2x2^{-5}, после выравнивания мы получаем 11-разрядный результат:

1.1010100000

0.0000110001

1.1011010001

Поскольку в нашем примере мантисса имеет только шесть разрядов, необходимо произвести операцию округления. В первом отбрасываемом разряде мы имеем единицу, но значения первого обрасываемого разряда не достаточно для правильного выполнения округления – необходимо также знать, есть-ли хотя-бы одна единица в любом из отбрасываемых разрядов, кроме первого. В данном случае такая единица есть в последнем обрасываемом разряде, следовательно в этом случае необходимо производить округление в большую сторону. Но нужно-ли было использовать 11-ти разрядный сумматор в этом случае? Очевидно, что в числе с большим порядком во всех отбрасываемых разрядах будут нули, т.е. переноса из отбрасываемого разряда в последний значащий разряд возникнуть не может, поэтому для вычисления суммы двух N-разрядных мантисс достаточно наличия N-разрядного сумматора, при условии, что мы сохраняем значение первого отбрасываемого разряда и факт наличия хотя бы одной единицы в остальных отбрасываемых разрядах числа с меньшим значением порядка. Поэтому два-три бита расширения для N-разрядного сумматора должно быть достаточно для любых возможных вариантов округления.

В большинстве блоков выполнения операций с плавающей точкой распакованная мантисса представляется в следующем формате:

```
1.XXXXXXXXXXXXXXXXXXXXXXX   0   0   0
|          |                |   |   |
|          |                |   |   - Избыточный бит
|          |                |   |       округления S (sticky bit)
|          |                |   - Первый бит округления R (round bit)
|          |                - Граничный бит G (guard bit)
|          - 23 бита мантиссы из формата числа в стандарте IEEE-754
- Скрытый бит, всегда равен 1
```

Как видно из представленного формата, здесь используется три бита расширения для стандартного распакованного формата мантиссы. Первый граничный бит G (guard) используется для сохранения бита округления в случае постнормализации. Первый бит округления R и

избыточный бит округления S используются для выполнения операции округления, которая заключается в инкременте результата при определенной комбинации битов округления либо просто в отбрасывании младших разрядов.

При возникновении переноса из старшего разряда мантиссы первым отбрасываемым разрядом становится последний значимый бит суммы, при этом значение второго (избыточного) бита для округления (называемого в англоязычной литературе sticky) будет определяться как результат операции «логическое ИЛИ» первого и второго битов округления в числе с меньшим порядком. Например:

1.11101

0.01001 11

1000110 11

Порядок суммы будет равен большему значению порядка двух слагаемых. В случае возникновения переноса из старшего разряда мантиссы, после вычисления суммы необходимо произвести нормализацию, сдвинув мантиссу вправо на один разряд и увеличить значение порядка на единицу.

Сложения чисел с разными знаками, также, как и вычитание чисел с плавающей запятой, производится по таким же правилам, но перед выполнением сложения отрицательное слагаемое или вычитаемое переводится в дополнительный код, а при нормализации используется арифметический сдвиг.

С учетом всего, о чем было сказано выше, алгоритм сложения чисел двух чисел $z_1*m_1*2^{e1}$ и $z_2*m_2*2^{e2}$, где m_i – это значения мантиссы в распакованном виде, e_i – значение порядка, z_i – знак, можно записать в следующем виде:

1 Если e1 < e2, то операнды меняются местами. Это позволяет избежать негативных значений разницы между порядками d = e1 – e2. Предварительно значение порядка суммы устанавливается в e1.

2 Если z1 != z2, то m2 переводится в дополнительный код.

3 m2 сдвигается арифметически вправо на d разрядов, значение первого отбрасываемого бита m2 записывается в g, значение второго отброшенного бита – r, результат операции «ИЛИ» остальных отбрасываемых бит – s.

4 Вычисляется предварительное значение суммы Sum = m1 + m2. Если z1 != z2, старший бит S равен 1 и не было переноса из

старшего разряда, то значение Sum меньше нуля – необходимо перевести Sum в дополнительный код.

5 Если z1 = z2, и был перенос из старшего разряда, то Sum сдвигается вправо на один разряд, при этом старший разряд устанавливается в единицу. Если переноса не было, то Sum сдвигается влево до нормализации. При первом сдвиге влево позиция заполняется значением бита g, при последующий сдвигах влево – нулями. При сдвиге вправо значение порядка суммы увеличивается на единицы, про сдвиге влево – уменьшается.

6 Если на предыдушем шаге Sum была сдвинута вправо, то $s = g \vee r \vee s$, а r будет равно значению отброшенного бита суммы. Если не было сдвига, то $s=r \vee s$, r = g. Если был сдвиг влево на один разряд, то s и r остаются без изменения. Если был сдвиг влево на два и более разрядов, то s и r равны нулю.

7 Округление суммы Sum в зависимости от значений битов r и s и используемого способа округления – см. Таблица 14. Если при округлении возникает перенос из старшего разряда, то Sum сдвигается вправо на один разряд и порядок суммы увеличивается на единицу.

Таблица 14. Правила округления результата.

Способ округления	Результат положительный	Результат отрицательный
К отрицательной -∞	Отбрасывание	+ (r ∨ s)
К положительной +∞	+ (r ∨ s)	Отбрасывание
К нулю 0	Отбрасывание	Отбрасывание
Ближайшее	+ ((r ∧ p0) ∨ (r ∧ s))	+ ((r ∧ p0) ∨ (r ∧ s))

8 Определяется знак результата: если z1 = z2, то знак суммы будет таким-же, в противном случае знак результата вычисляется в зависимости с правилами, представленными в Таблица 15.

Таблица 15. Знак результата сложения.

Слагаемые менялись местами на шаге 1?	Сумма переводилась в дополнительный код?	Знак первого слагаемого	Знак второго слагаемого	Знак результата
Да	**Да**	+	-	-
Да	**Нет**	-	+	+
Нет	**Нет**	+	-	+
Нет	**Нет**	-	+	-
Нет	**Да**	+	-	-

Нет	**Да**	**-**	**+**	**+**

В случае, если одно из слагаемых является денормализованным, то при приведении мантиссы в распакованную форму, смещенное значение порядка денормализованного слагаемого устанавливается в 1 и описанный выще алгоритм работает без изменений.

В случае представлении суммы в виде денормализованного числа, на шаге 5 алгоритма сложения необходимо следить за тем, чтобы смещенное значение порядка не стало меньше 1. Если порядок суммы достигает значения единицы, то нормализация прекращается и производится округление. Если в результате округления возникает перенос из старшего разряда, то результат можно записать в нормализованном виде и значение порядка остается равным единице, в противном случае значение порядка устанавливается в ноль и мантисса остается денормализованной.

Переполнение при сложении может возникнуть, когда при сдвиге суммы вправо, смещенное значение порядка становится равным 255 при вычислении с одинарной точностью (2047 при вычислении с двойной точностью), либо когда тоже самое случается после округления.

Проиллюстрируем рассмотренный алгоритм примере на последовательности операций по сложению двух вещественных чисел:

Исходные операнды

A + $1.110010000111010011110000 * 2^{8}$ = 456.456787109375

B + $1.110110110101011010010000 * 2^{0}$ = 1.8567895889282226

Выравнивание порядков (сдвиг мантиссы B на восемь разрядов вправо)

A + $1.110010000111010011110000|000 * 2^{8}$

B + $0.000000011101101101010011|011 * 2^{8}$

Результат сложения мантисс

A+B + $1.110010100101000001000011|011 * 2^{8}$

Нормализация результата

A+B + 1.11001010010100000100011|01 *2^{8}

Округление к нулю

A+B + 1.1100101001010000100011 *2^{8} =
458.3135681152344

Округление к ближайшему четному

A+B + 1.1100101001010000100011 *2^{8} =
458.3135681152344

Округление к положительной бесконечности

A+B + 1.1100101001010000100100 *2^{8} =
458.3135986328125

Округление к отрицательной бесконечности

A+B + 1.1100101001010000100011 *2^{8} =
458.3135681152344

Стандартным режимом округления по умолчанию для стандарта IEEE-754 является округление к ближайшему четному. Практика использования других режимов округления достаточно ограниченная.

Как выглядит устройство сложения для чисел с плавающей запятой? Практически это прямая реализация описанного выше алгоритма с двумя раздельными каналами, один из которых обрабатывает порядки, а другой канал обрабатывает мантиссы. Оба канала управляются одной схемой управления, которая переключает мультиплексоры и генерирует сигналы управления сдвигом мантиссы и инкремента порядка. Структура устройства представлена на Рис. 24. АЛУ порядков используется для сравнения и вычисления разницы порядков, которая подается на схему управления. Если разность порядков отрицательная, то операнды меняются местами, и мантисса меньшего числа подается на схему сдвига мантиссы для выравнивания. АЛУ мантисс производит операцию сложения или вычитания и результат подвергается нормализации с использованием детектора лидирующих нулей и схемы сдвига. При этом значение порядка корректируется на число нулевых позиций в старших разрядах предшествующих лидирующей единице. Затем нормализованный результат подвергается округлению, при этом может возникнуть перенос из старшего разряда мантисы. Тогда результат должен быть

сдвинут вправо на одну позицию, а значение порядка увеличено на единицу (инкрементировано). Округленная и нормализованная мантисса вместе с скорректированным порядком выдаются на регистр результата.

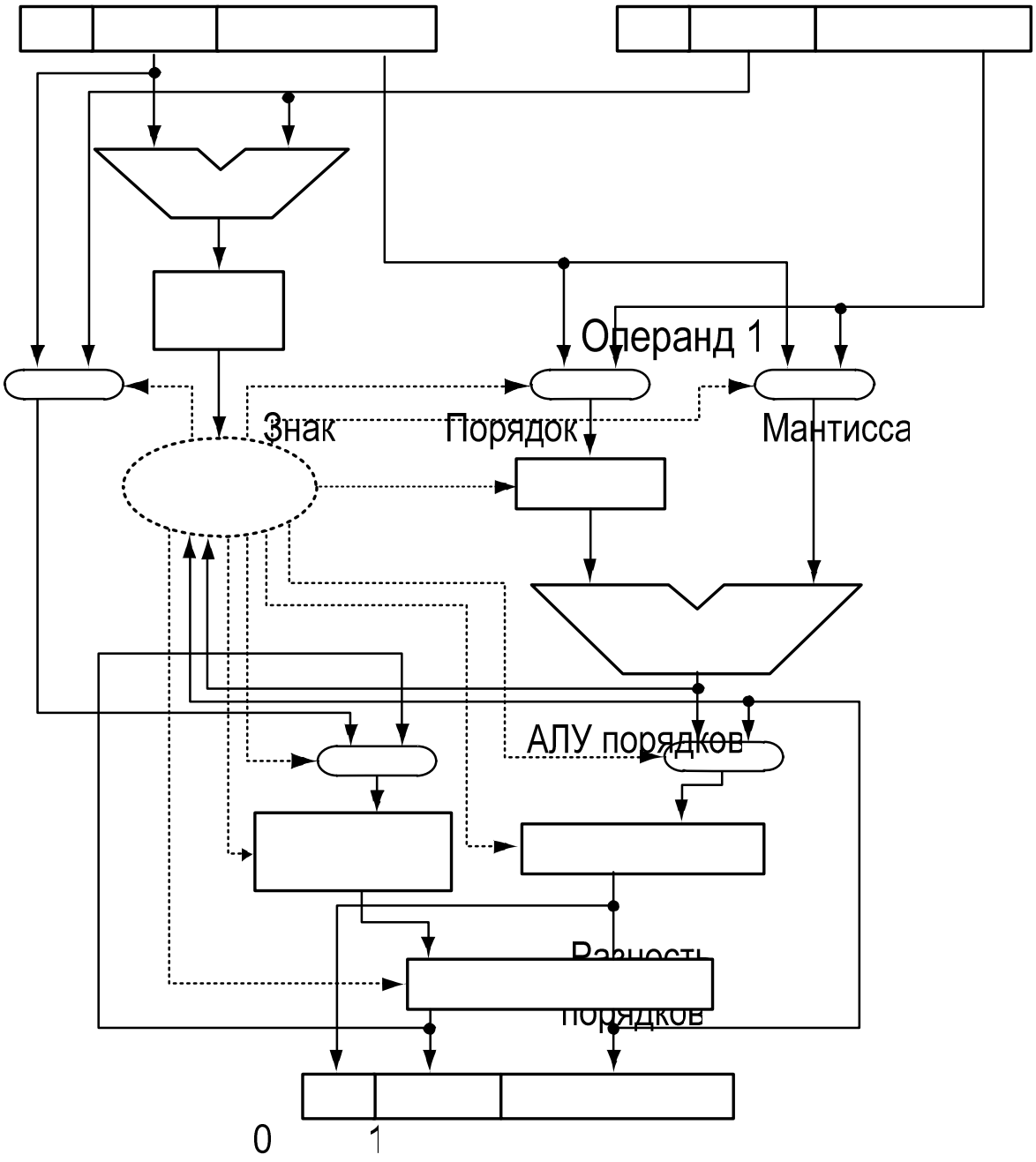


Рис. 24. Устройство сложения чисел с плавающей запятой.

Реальные устройства сложения и вычитания чисел с плавающей запятой могут отличаться от приведенной, так как могут быть скомбинированы с устройствами выполнения умножения для выполнения комбинированной операции умножения-сложения.

9.2 Алгоритм сложения в арифметическом устройстве ПЗ

В качестве практического примера исполнения устройства сложения ПЗ предлагается псевдокод, который может быть использован в качестве основы для создания детальной побитовой модели устройства сложения/вычитания на SystemC или Verilog.

Алгоритм исполнения команды сложения

Входные операнды: SRC0 (SRC0_SIGN, SRC0_EXP, SRC0_MANT) // знак,порядок SRC1 (SRC1_SIGN, SRC1_EXP, SRC1_MANT) // мантисса

Формат операндов: FP32 / FP16

Результат: RES

Формат результата: FP32 / FP16

Операция: RES = SRC0 + SRC1

Точность: 0.5 ULP

Разрядность сложения: SRC_MANT_BIT + 2

Если формат исходного операнда FP32 (1.8.23), то размер поля SRC_MANT_BIT равен 23 битам.

ADDER_WIDTH (ширина сумматора) равна SRC_MANT_BIT + 4, так как распакованная мантисса включает еще один бит X (SRC_MANT_BIT + 1), кроме этого нужны биты для поддержки округления IEEE-754.

```
ADDER_MSK = (1 << ADDER_WIDTH) – 1; // Маска сумматора
ADDER_SIGN_MSK = 1 << (ADDER_WIDTH - 1); // Маска знака сумматора
SUM_SIGN_MSK = 1<< (ADDER_WIDTH + 1); //SUM требует включения R_bit и S_bit для нормализации и округления.
SUM_MSK = (1 << (ADDER_WIDTH + 2)) – 1;
SUM_WIDTH = ADDER_WIDTH + 2;
//Аппаратная реализация не использует SIGN bit and X bit в общем поле операндов, поэтому реальная ширина сумматора должна быть на два бита меньше.
//HW_ADDER_WIDTH = SRC_MANT_BIT + 2
```

Алгоритм:

Считать вещественные операнды в диапазоне 0,1 с входов устройства:

SRCX_SIGN, X = 0, 1

SRCX_MANT_REAL X = 0, 1

```
SRCX_EXP            X = 0, 1
```

SRCX_MANT_REAL = [1 : SRCX_MANT], в режиме FP32 , будет 24 бита (23+X);

1. Особый случай с аномальными операндами:

```
     IF ( (SRC0 == NAN)||(SRC1 == NAN) )
          RES = NAN;
     ELSE IF( SRC0 == INF )
{
     IF ( (SRC1 == INF)&&(SRC0_SIGN ^ SRC1_SIGN)
          RES = NAN;
     ELSE
          RES = (SRC0_SIGN) INF;
}
ELSE IF (SRC1 == INF)
     RES = (SRC1_SIGN) INF;
```

2. Обычные операнды:

В этом случае, SRCX != NAN or INF. X = 0, 1

Шаг A: Сдвиг для выравнивания и сложение.

Преобразовать формат слагаемых из 1.23 в s2.26.

```
ADDER_0 = (SRC0_EXP != 0) ? (SRC0_MANT_REAL << 1) : (MODE ? (SRC0_MANT_REAL << 2) : 0);
ADDER_1 = (SRC1_EXP != 0) ? (SRC1_MANT_REAL << 1) : (MODE ? (SRC1_MANT_REAL << 2) : 0);

IF(SRC0_EXP >= SRC1_EXP)
{
     DST_SIGN = SRC0_SIGN;
     DST_EXP = SRC0_EXP;
     SHFT_R = SRC0_EXP - SRC1_EXP;
IF(SRC1_SIGN ^ SRC0_SIGN)
     ADDER_1 = (~ADDER_1 + 1) & ADDER_MSK;
// Слагаемое ADDER_1 выдвинуто влево в S_bit
     IF(SHFT_R >= ADDER_WIDTH - 1)
```

```
     {
          S_BIT = (ADDER_1 != 0) ? 1 : 0;
          B_BIT = (ADDER_1 < 0) ? 1 : 0;
          ADDER_1 = (ADDER_1 < 0) ? ADDER_MSK : 0;
}
ELSE IF(SHFT_R == 0)
{
     R_BIT = S_BIT = 0;
}
ELSE
{
     R_BIT = ADDER_1[SHFT_R - 1 : SHFT_R - 1]
     S_BIT = ADDER_1[SHFT_R - 2 : 0] ? 1 : 0
     ADDER_1   =   SIGN_EXTEND(ADDER_1[ADDER_WIDTH   -   1   :
SHFT_R]);
}

SUM = (ADDER_0 + ADDER_1) & ADDER_MSK;
SUM = (SUM<<2) | (R_BIT<<1) | S_BIT;

IF(SUM & SUM_SIGN_MSK)
{
     DST_SIGN = SRC1_SIGN;
     SUM = (~SUM + 1) & SUM_MSK;
}
}
ELSE
{
     DST_SIGN = SRC1_SIGN;
     DST_EXP = SRC1_EXP;

     SHFT_R = SRC1_EXP - SRC0_EXP;

IF(SRC1_SIGN ^ SRC0_SIGN)
     ADDER_0 = (~ADDER_0 + 1) & ADDER_MSK;
     IF(SHFT_R >= ADDER_WIDTH - 1)
```

```
     {
          S_BIT = (ADDER_0 != 0) ? 1 : 0;
          B_BIT = (ADDER_0 < 0) ? 1 : 0;
          ADDER_0 = (ADDER_0 < 0) ? ADDER_MSK : 0;
     }
     ELSE
     {
     R_BIT = ADDER_0[SHFT_R - 1 : SHFT_R - 1]
     S_BIT = ADDER_0[SHFT_R - 2 : 0] ? 1 : 0
     ADDER_0   =   SIGN_EXTEND(ADDER_0[ADDER_WIDTH   -   1   :
SHFT_R]);
     }
SUM = (ADDER_0 + ADDER_1) & ADDER_MSK;
SUM = (SUM<<2) | (R_BIT<<1) | S_BIT;
}
```

Шаг B: **Нормализация суммы для получения конечного результата.**

```
IF(SUM == 0)
     RES = (SRC0_SIGN & SRC1_SIGN) 0;
ELSE
{
     //Значение SUM положительно, тогда значение SUM будет
в битах [SUM_WIDTH -2 : 0].
     //  DST_MANT_TMP  имеет  формат  1.(MANT_BIT  +  3).
Дополнительные  3  последних  бита  используются  для
округления.
     //Вычисление позиции лидирующей единицы LOnePos для
значения SUM;
     IF(LOnePos == SUM_WIDTH - 2)
     {
          DST_MANT_TMP = (SUM >> 1) | (SUM & 0x1);
          DST_EXP += 1;
}
ELSE IF( (LOnePos < SUM_WIDTH - 3)&&(DST_EXP != 0) )
{
     IF(DST_EXP > SUM_WIDTH - 3 - LOnePos)
{
```

```
      DST_EXP -= (SUM_WIDTH – 3 - LOnePos);
      DST_MANT_TMP = SUM << (SUM_WIDTH – 3 - LOnePos);
}
ELSE
{
      IF(MODE)
      {
            RES = (DST_SIGN)0;
      }
      // Поддержка обработки денормализованных значений
      ELSE
      {
            DST_MANT_TMP = SUM << (DST_EXP - 1);
            DST_EXP = 0;
      }
}
}
      ELSE IF(DST_EXP == 0)
      {
            IF(MODE)
                  RES = (DST_SIGN) 0;
            // Поддержка обработки денормализованных значений
            ELSE
            {
                  DST_MANT_TMP = (SUM >> 1) | (SUM & 0x1);
            }
      }
      DST_MANT = ROUND_NEAREST_EVEN ( DST_MANT_TMP );
      IF(DST_MANT_TMP = переполнение после округления)
      {
            DST_EXP ++;
IF(DST_EXP = переполнение порядка)
      RES = (DST_SIGN)INF;
      }
      RES = [DST_SIGN : DST_EXP : DST_MANT];
}
```

```
// Конец алгоритма
```

9.3 Умножение и деление чисел с плавающей запятой

В отличии от сложения, умножение чисел с плавающей запятой происходит по значительно более простой схеме. Проиллюстрируем принцип используя те же самые два двоичных числа 1.10101_2x2^0 и 1.10001_2x2^{-5}, как и в алгоритме сложения.

$(1.10101_2x2^0)*(1.10001_2x2^{-5}) = (1.10101_2 * 1.10001_2) * (2^0*2^{-5}) = (1.10101_2 * 1.10001_2) * 2^{0-5}$

Используя обычные правила математики можно заметить, что операция умножения двух чисел в формате с плавающей запятой сводится к перемножению значений мантисс и сложению значений порядков.

Алгоритм умножения реализуется именно таким образом и состоит из пяти основных шагов (Рис. 25).

На первом шаге производится сложение порядков двух чисел. При этом надо помнить, что значения порядков представлены в смещенном коде и для правильного выполнения сложения их надо бы вернуть в исходный прямой код и затем произведя операцию добавить смещение. Для упрощения этой операции вместо двух вычитаний смещения из порядков операндов и прибавления значения смещения к порядку результата производится только вычитание смещения из результата сложения порядков операндов, представленных в смещенном коде.

На втором шаге надо перемножить мантиссы, для этого используются один из способов умножения описанных в секции умножения. Безусловно для повышения скорости работы должен использоваться быстродействующий матричный умножитель по методу Буфа.

На третьем шаге надо нормализовать значение произведения мантисс и произвести соотвествующую коррекцию порядка результата. При умножении, как и при сложении, результат не может быть денормализован влево больше, чем на один разряд так как 2+2 = 2*2.

После проверки переполнения (наличия аномального результата) на четвертом шаге алгоритма мантиссу результата надо округлить до нужного числа разрядов. При округлении возможен перенос из старшего разряда мантиссы и может понадобиться повторная нормализация со сдвигом результата вправо на один разряд и соответственным увеличением порядка на единицу.

На пятом шаге определяется знак произведения на основании знаков операндов, используя обычные правила арифметики. Если знаки одинаковы, то знак произведения положительный. Если разные, то знак произведения отрицательный.

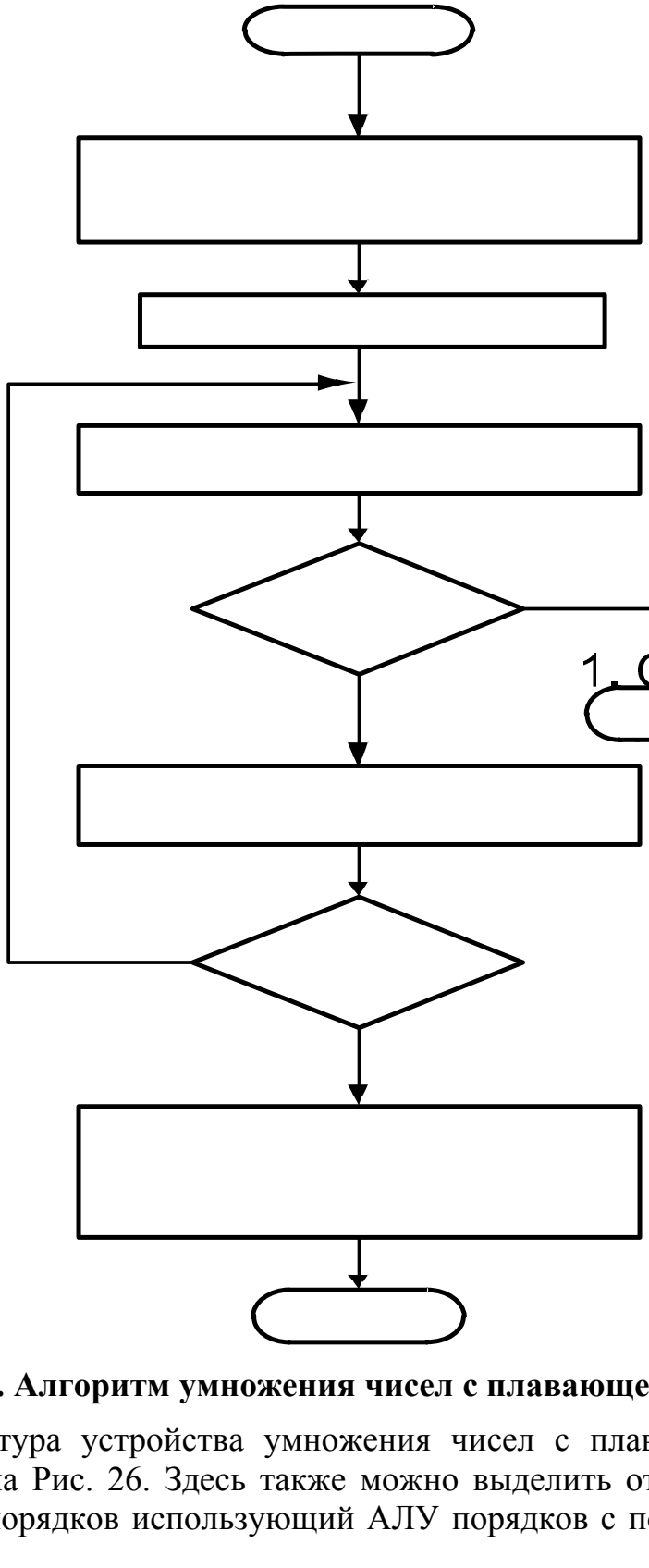


Рис. 25. Алгоритм умножения чисел с плавающей запятой.

Структура устройства умножения чисел с плавающей точкой приведена на Рис. 26. Здесь также можно выделить отдельный канал обработки порядков использующий АЛУ порядков с последующей их

коррекцией после нормализации и округления. Для коррекции может использоваться то же самое АЛУ порядков, либо отдельные сумматоры, в зависимости от требований к пропускной способности. Канал обработки мантисс содержит матричный умножитель с кодированием Буфа и построенный с использованием дерева или матрицы CSA сумматоров с сохранением переноса, который выводит результат в избыточном кодировании на полный сумматор для получения нормализованного двоичного результата.

При этом детектор нулевых старших разрядов определяет число сдвигов, необходимых для нормализации результата и которые также являются величиной, на которую должен быть скорректирован порядок. Затем производится округление и повторная нормализация после округления в случае возникновения переноса из старшего разряда мантиссы.

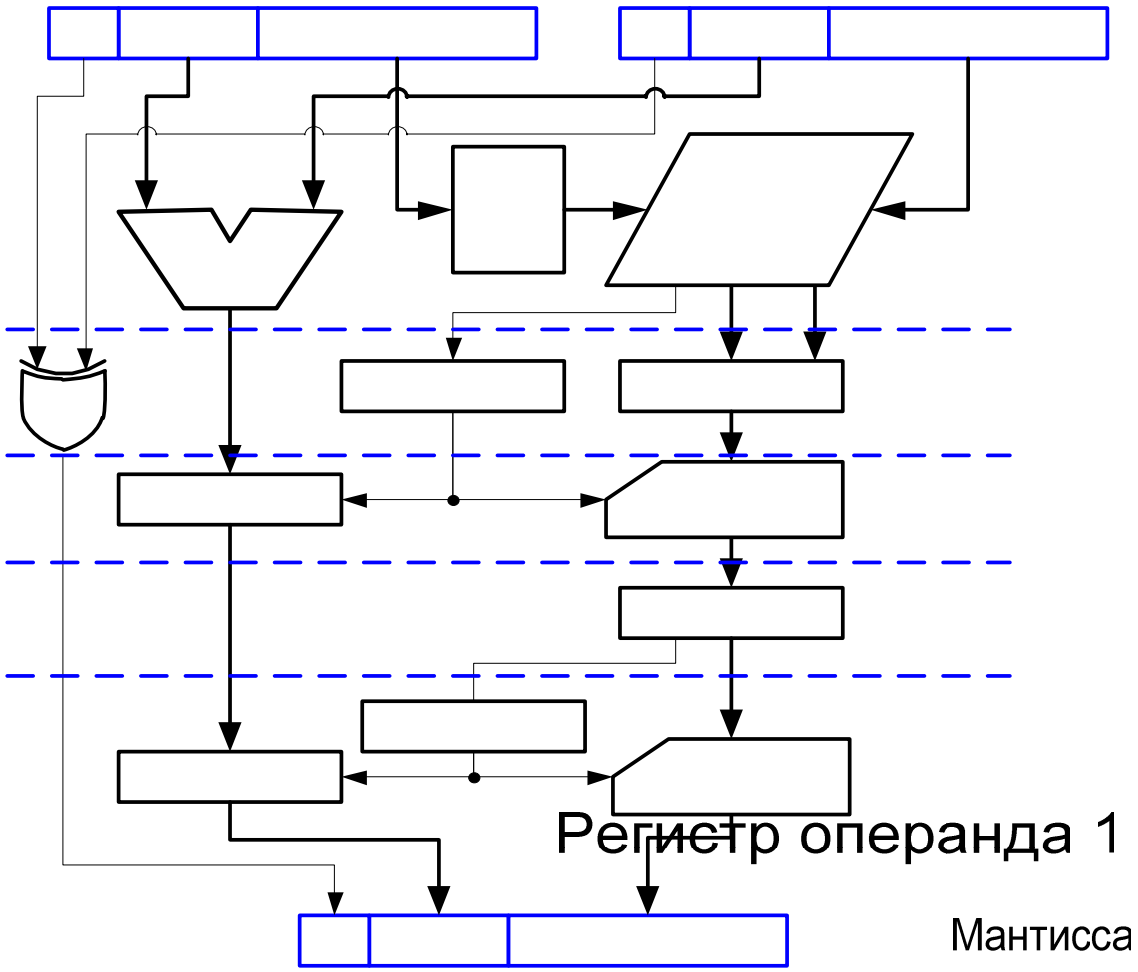


Рис. 26. Устройство умножения чисел с плавающей запятой.

Кодирование в формат Буфа

АЛУ

Следует отметить, что операции с плавающей запятой не могут быть выполнены за один такт, поэтому необходима фиксация промежуточных результатов на выходе логических схем в специальных конвейерных регистрах, которые позволяют выполнить операцию за несколько тактов. Ниже проиллюстрирован пример умножения двух чисел. Рекомендуем обратить внимание на биты расширения мантиссы, необходимые для округления, и как они влияют на конечный результат округления в различных режимах.

Исходные операнды умножения

A + 1.10101111111110111110011 *2^{4} = 26.998998641967773

B + 1.00011111101011100001010 *2^{3} = 8.989999771118164

Результат умножения с битами расширения для округления

A*B + 1.11100101011100010010010|111 *2^{7}

Нормализация результата

A*B + 1.11100101011100010010010|11 *2^{7}

Округление к нулю

A*B + 1.1110010101110001001001 0 *2^{7} = 242.72097778320312

Округление к ближайшему четному

A*B + 1.1110010101110001001001 1 *2^{7} = 242.7209930419922

Округление к положительной бесконечности

A*B + 1.1110010101110001001001 1 *2^{7} = 242.7209930419922

Округление к отрицательной бесконечности

A*B + 1.1110010101110001001001 0 *2^{7} = 242.72097778320312

Устройство деления с плавающей запятой в значительной степени похожи на устройство умножения (Рис. 27). В отличии от умножения при делении порядки чисел надо не складывать, а вычитать, поэтому АЛУ работает по умолчанию в режиме вычитания. При этом значение смещения должно быть прибавлено к разнице порядков, которые вычитались без преобразования в прямой код. Других отличий, кроме специального регистра задержки в канале обработки порядка, нет. В канале обработки мантисс существенным отличием является наличие быстрого делителя мантисс, аналогичного делителю на Рис.21. В отличие от дерева или матрицы сумматоров CSA в умножителе даже быстрый делитель работает значительно медленнее и требуется некоторое количество тактов для выполнения операции деления мантисс. Поэтому разница порядков должна быть сохранена в регистре задержки до момента завершения процедуры деления. Остальная последовательность операций практически такая же как и при умножении. Некоторые отличия имеются в реализации округления, когда мантиссу надо сдвигать влево и уменьшать значение порядка на единицу. В такой момент как раз и пригодится граничный бит G, который позволит сохранить значения обоих битов округления R и S.

Пример деления двух чисел приведен ниже:

```
          Исходные операнды деления
A     + 1.10101110011001100110011 *2^4 = 26.899999618530273
B     + 1.00010001110100000111110 *2^3 = 8.556699752807617
```

```
      Результат деления мантисс
A/B   + 1.10010010011001011110101|111 *2^1
```

```
      Нормализация
A/B   + 1.10010010011001011110101|11 *2^1
```

```
      Округление к нулю
A/B   + 1.10010010011001011110101 *2^1 = 3.14373517036438
```

```
      Округление к ближайшему четному
A/B   + 1.10010010011001011110110 *2^1 = 3.143735408782959
```

```
      Округление к положительной бесконечности
A/B   + 1.10010010011001011110110 *2^1 = 3.143735408782959
```

Округление к отрицательной бесконечности

A/B + $1.10010010011001011110101 * 2^{1} = 3.14373517036438$

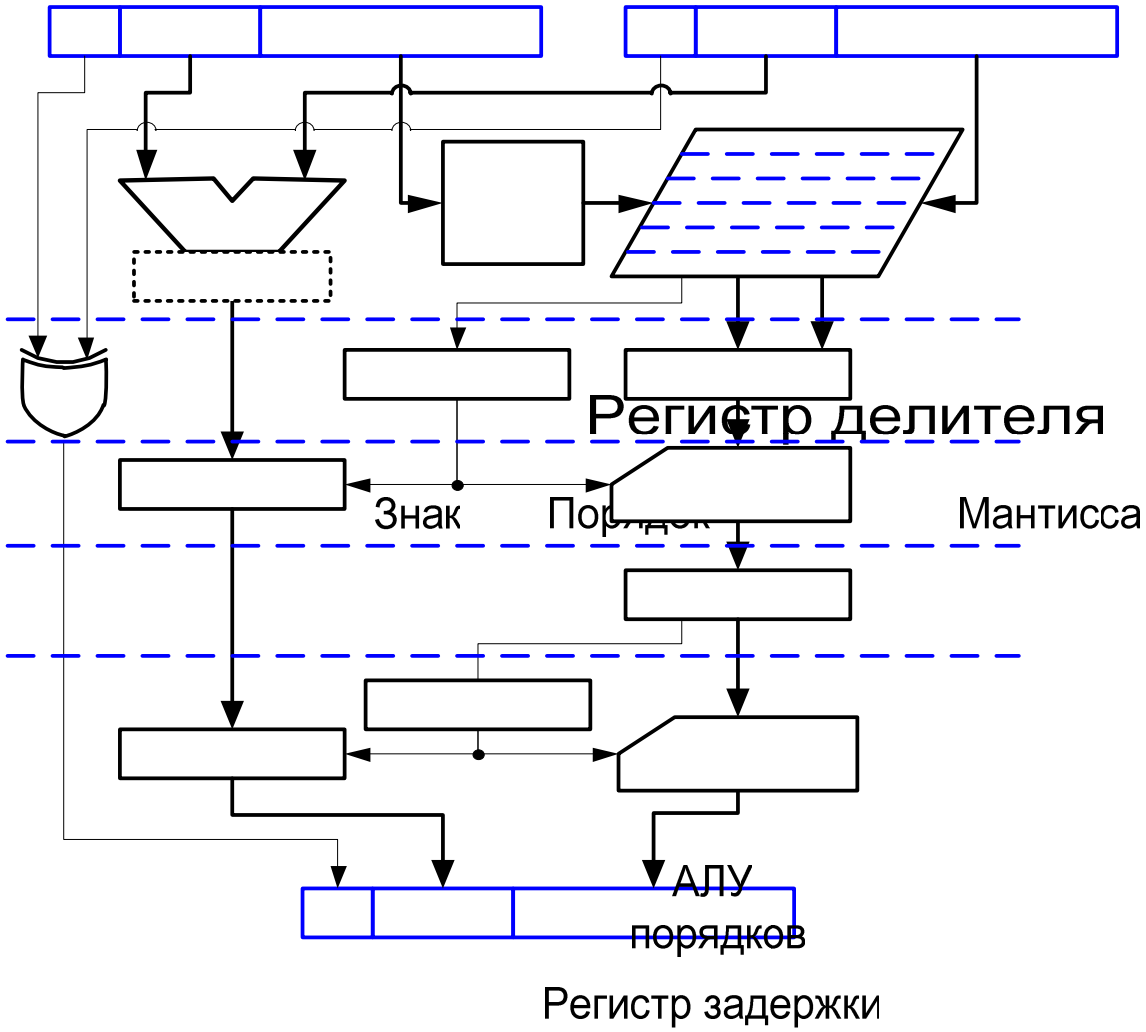


Рис. 27. Устройство деления чисел с плавающей запятой.

Устройства деления, как правило, выполняются как многотактные устройства, в которых результат получается с существенной, по сравнению с сложением и умножением, задержкой. Так как операции деления не очень часто встречаются в прикладных программах, использование итеративных версий алгоритмов деления в комбинированом умножителе-делителе, является одним из оптимальных способов построения таких устройств в современных микропроцессорах. Вариант реализации подобного рассмотрен в секции с описанием арифметического устройства SPARC64.

9.4 Алгоритм умножения в арифметическом устройстве ПЗ

Алгоритм FMUL_FP32

Входные операнды: SRC0 (SRC0_SIGN, SRC0_EXP, SRC0_MANT) // знак,порядок SRC1 (SRC1_SIGN, SRC1_EXP, SRC1_MANT) // мантисса

Формат операндов: FP32

Результат: RES

Формат результата: FP32

Операция: RES = SRC0 x SRC1

Точность: 0.5 ULP

Разрядность сложения: SRC_MANT_BIT + 2

Точность: 0.5 ULP

Умножитель: 24 бит X 24 бит, умножение без знака, выход 48 бит

Алгоритм умножения ПЗ:

Считать вещественные нормализованные значения с входов устройства умножения:

SRCX_SIGN, X = 0, 1

SRCX_MANT_REAL X = 0, 1

SRCX_EXP X = 0, 1

```
SRCX_MANT_REAL = [1 : SRCX_MANT];
RES_SIGN = SRC0_SIGN ^ SRC1_SIGN;
```

1. Особый случай:

```
IF( (SRC0 == NAN)||(SRC1 == NAN) )
     RES = NAN;
ELSE IF( (SRC0 == INF)||(SRC1 == INF) )
{
     IF( (SRC0_EXP == 0)||(SRC1_EXP == 0) )
          RES = NAN;
     ELSE
          RES = (RES_SIGN)INF;
}
ELSE IF( (SRC0_EXP == 0)||(SRC1_EXP == 0) )
     RES = (RES_SIGN) 0;
```

2. Обычные операнды:

В этом случае, SRCX_EXP != 0 or INF, X = 0, 1

```
OUT = SRC0_MANT_REAL * SRC1_MANT_REAL;
// выражение выше использует 24 x 24 умножитель, выход 48 бит.
RES_EXP = SRC0_EXP + SRC1_EXP - 126;
IF (RES_EXP <= 0)
     RES = (RES_SIGN) 0;
ELSE IF(RES_EXP > 0xff)
     RES = (RES_SIGN) INF;
ELSE IF (OUT выходное значение 48 bit)
{
     IF( RES_EXP == 0xff)
          RES = (RES_SIGN) INF;
     ELSE
     {
          //Округлить значение от 48 бит до старших 24 бит RES_MANT;
          RES = [RES_SIGN, RES_EXP, RES_MANT[22 : 0] ];
}
}
ELSE //выход 47 бит
{
     IF ( RES_EXP == 1)
          RES = (RES_SIGN) 0;
     ELSE
{
     RES_EXP = RES_EXP - 1;
// Округлить с 47 до старших 24 bit RES_MANT;
     RES = [RES_SIGN, RES_EXP, RES_MANT[22 : 0] ];
}
}
// Конец алгоритма
```

10 Команды обработки чисел с плавающей запятой в процессорах MIPS и ARM

10.1 Команды обработки вещественных чисел в MIPS

Процессоры семейства MIPS имеют относительно небольшой и хорошо структурированный набор команд арифметики с ПЗ. Все

команды арифметических действий имеют как одинарную, так и двойную точность. Для обработки чисел с ПЗ используется отдельный набор регистров с ПЗ, которые группируются попарно в случае обработки чисел с двойной точностью. Формат команд с ПЗ похож на формат целочисленных команд и имеет два типа FR (регистровый) и FI (регистр-константа), Таблица 16. Поле Fmt формата операндов определяет двойную или одинарную точность операций с ПЗ. Поля Ft, Fs, Fd определяют регистры операндов и результата в наборе регистров ПЗ. Остальные поля аналогичны формату целочисленных команд.

Таблица 16. Формат команд с ПЗ процессора MIPS.

Тип	**-31-**	**Формат**				**-0-**
FR	opcode (6) КОП	Fmt (5) Одинарная или двойная точность	Ft (5) Регистр операнда T	Fs (5) Регистр операнда S	Fd (5) Регистр результата	funct (6) Функция расширения кода операции
FI	opcode (6) КОП	Fmt (5) Одинарная или двойная точность	Ft (5) Регистр операнда T	immediate (16) Непосредственная константа		

Таблица 17. Использование регистров и памяти в операциях с ПЗ в сопроцессоре MIPS

Ресурс	**Пример**	**Комментарии**
32 регистра для чисел с плавающей запятой	$f0, $f1, $f2,..., $f31	Регистры с плавающей запятой в MIPS используются парами в случае исполнения команд двойной точности. Регистры с нечетным номером не могут быть использованы в арифметических опреациях или командах условного перехода, а могут быть использованы только для пересылки «правой половины» регистровой пары, используемой для числа двойной точности.
2^{30} адресуемых слов в памяти	Memory[0], Memory[4],..., Memory[4293967292]	Доступ только в командах пересылки данных. MIPS использует байтовый адрес, поэтому адреса последовательных слов будут различаться на 4. В памяти хранятся структуры данных, содержание регистров при выполнении вызова процедур.

Таблица 18. Команды обработки данных с плавающей запятой в MIPS

Команда	Пример	Тип	Действие	Комментарии
FP add single	add.s Fd, Fs, Ft	FR	Fd = Fs + Ft	Сложение с ПЗ (одинарная точность)
FP subtract single	sub.s Fd, Fs, Ft	FR	Fd = Fs - Ft	Вычитание с ПЗ (одинарная точность)
FP multiply single	mul.s Fd, Fs, Ft	FR	Fd = Fs * Ft	Умножение с ПЗ (одинарная точность)
FP divide single	div.s Fd, Fs, Ft	FR	Fd = Fs / Ft	Деление с ПЗ (одинарная точность)
FP add double	add.d Fd, Fs, Ft	FR	$Fd_{,+1} = Fs_{,+1} + Ft_{, +1}$	Сложение с ПЗ (двойная точность)
FP subtract double	sub.d Fd, Fs, Ft	FR	$Fd_{,+1} = Fs_{,+1} - Ft_{,+1}$	Вычитание с ПЗ (двойная точность)
FP multiply double	mul.d Fd, Fs, Ft	FR	$Fd_{,+1} = Fs_{,+1} * Ft_{,+1}$	Умножение с ПЗ (двойная точность)
FP divide double	div.d Fd, Fs, Ft	FR	$Fd_{,+1} = Fs_{,+1} / Ft_{,+1}$	Деление с ПЗ (двойная точность)

Таблица 19. Команды пересылки, сравнения и условных переходов в сопроцессоре ПЗ MIPS

Команда	Пример	Действие	Комментарии
load word coprocessor 1	lwc1 F(rt), offset(Rs)	F(rt) = Memory[(Rs)+offset]	Пересылка 32-бит данных из памяти в регистр ПЗ
store word coprocessor 1	swc1 $ F(rt), offset(Rs)	Memory[(Rs)+offset] = F(rt)	Пересылка 32-бит данных из регистра ПЗ в память
branch on FP true	bc1t offset	if (cond == 1) go to PC+4+offset	Переход от текущего значения счетчика команд по позитивному результату сравнения чисел с ПЗ
branch on FP false	bc1f offset	if (cond == 0) go to PC+4+offset	Переход от текущего значения счетчика команд по негативному результату сравнения чисел с ПЗ
FP compare single (eq,ne,lt,le,gt,ge)	c.lt.s $f2,$f4	if ($f2 < $f4) cond=1; else cond=0	Сравнение МЕНЬШЕ, РАВНО, НЕРАВНО, МЕНЬШЕ-РАВНО, БОЛЬШЕ, БОЛЬШЕ-РАВНО чисел с ПЗ одинарной точности
FP compare double (eq,ne,lt,le,gt,ge)	c.lt.d $f2,$f4	if ($f2 < $f4) cond=1; else cond=0	Сравнение МЕНЬШЕ, РАВНО, НЕРАВНО, МЕНЬШЕ-РАВНО, БОЛЬШЕ, БОЛЬШЕ-

			РАВНО чисел с ПЗ двойной точности

10.2 Команды обработки вещественных чисел в процессоре ARM

Команды обработки вещественных чисел в ARM являются командами сопроцессора (при этом используются номера 1 и 2). Команды могут исполняться реальным сопроцессором либо эмулироваться на программном уровне с помощью целочисленных команд. Используется восемь регистров F0-F7, все арифметические команды выполняются по схеме «регистр-регистр», а перед этим данные должны быть загружены из памяти командами пересылки. Операции могут выполняться в различных форматах, при этом формат `<prec>` (точность) должна быть специфицирована непосредственно в команде. Поддерживаются следующие форматы:

- S (Single) – одинарная точность
- D (Double) – двойная точность
- E (Extended) – распакованное десятичное число
- P (Packed BCD) – упакованное десятичное число

Также должен быть явно указан режим округления `<round>`, который позволяет выбрать один из режимов, определяемых стандартом IEEE. По умолчанию всегда устанавливается режим округления к ближайшему четному. В ниже следующих примерах обычные регистры ARM определятся как Rx, регистры с ПЗ как Fx. В Таблица 20-24 приведены детали реализации операций с ПЗ в архитектуре ARM.

Таблица 20. Использование регистров, памяти и модификаций команд в операциях с плавающей запятой в архитектуре ARM

Ресурс	Пример	Комментарии
Восемь 80-разрядных регистров в сопроцессоре для чисел с плавающей запятой	F0, F1, F2,..., F7	Регистры с плавающей запятой в MIPS используются парами в случае исполнения команд двойной точности. Регистры с нечетным номером не могут быть использованы в арифметических опреациях или командах условного перехода, а могут быть использованы только для пересылки «правой половины» регистровой пары, используемой для числа двойной точности.
Регистр статуса	FPSR (FP status	Доступный для пользователя регистр

операции с ПЗ	register)	статуса, который определяет вариации операций с ПЗ и служит также для индикации ошибочных результатов, вызывающих прерывание
Регистр управления ПЗ	FPCR (FP Control register)	Системный регистр доступный только для утилит операционной системы, который управляет аппаратным акселератором операций с ПЗ
2^{30} адресуемых слов в памяти	Memory[0], Memory[4],..., Memory[4293967292]	Доступ только в командах пересылки данных. ARM использует байтовый адрес, поэтому адреса последовательных слов будут различаться на 4. В памяти хранятся структуры данных, содержание регистров при выполнении вызова процедур. Поддерживается режим преиндексирования и постиндексирования адреса, также как и возможность копирования текущего адреса в регистр базового адреса
Условное исполнение	Operation{condition}	Как и обычные команды, команды с ПЗ имеют возможность условного исполнения при установке предиката

Синтаксис операций с ПЗ определяется как `op{condition}prec{round} Fd,Fn,Fm`. При этом `Fm` может быть как один из регистров ПЗ, так и одна из констант ПЗ #0, #1, #2, #3, #4, #5, #10 или #0.5. Причем быстрые операции выполняются только с одинарной точностью.

Таблица 21. Двухоперандные команды обработки данных с плавающей запятой в архитектуре ARM

Команда	Полный формат	Действие	Комментарии
ADF add	`ADF{cond}prec{round} Fd,Fn,Fm`	Fd:=Fn+Fm	Сложение с ПЗ
`MUF` multiply	`MUF{cond}prec{round} Fd,Fn,Fm`	Fd:=Fn*Fm	Умножение с ПЗ
`SUF` subtract	`SUF{cond}prec{round} Fd,Fn,Fm`	Fd:=Fn–Fm	Вычитание с ПЗ
`RSF` reverse subtract	`RSF{cond}prec{round} Fd,Fn,Fm`	`Fd:=Fm–Fn`	Реверсивное вычитание с ПЗ
`DVF` divide	DVF{cond}prec{round} Fd,Fn,Fm	`Fd:=Fn/Fm`	Деление с ПЗ
`RDF` reverse	RDF{cond}prec{round} Fd,Fn,Fm	Fd:=Fm/Fn	Реверсивное

divide			деление с ПЗ
POW power	POW{cond}prec{round} Fd,Fn,Fm	Fd:=Fn^ Fm	Возведение в степень с ПЗ
RPW reverse power	RPW{cond}prec{round} Fd,Fn,Fm	Fd:=Fm^ Fn	Реверсивное возведение в степень с ПЗ
RMF remainder	RMW{cond}prec{round} Fd,Fn,Fm	Fd:=остаток от Fn/Fm	Остаток от деления
FML fast multiply	FML{cond}prec{round} Fd,Fn,Fm	Fd:=Fn*Fm	Быстрое умножение с ПЗ (только одинарная точность)
FDV fast divide	FML{cond}prec{round} Fd,Fn,Fm	Fd:=Fn/Fm	Быстрое деление с ПЗ (только одинарная точность)
FRD fast reverse divide	FRD{cond}prec{round} Fd,Fn,Fm	Fd:=Fm/Fn	Быстрое реверсивное деление с ПЗ (только одинарная точность)
POL polar angle	POL{cond}prec{round} Fd,Fn,Fm	Fd:=полярный угол от Fn,Fm (=ATN(Fm/Fn)	Полярный угол при наличии частного

Таблица 22. Однооперандные команды ПЗ

MVF move	MVF{cond}prec{round} Fd,Fm	Fd:=Fm	Копирование из одного регистра ПЗ в другой
MNF move negated	MNF{cond}prec{round} Fd,Fm	Fd:=–Fm	Копирование со сменой знака
ABS absolute value	ABS{cond}prec{round} Fd,Fm	Fd:=ABS(Fm)	Абсолютное значение от содержимого регистра ПЗ
RND round to integral value	RND{cond}prec{round} Fd,Fm	Fd:= integer of Fm	Округление с нормализацией (с использованием текущего режима округления)
URD unnormalised round	URD{cond}prec{round} Fd,Fm	Fd:= integer of Fm (abnorm)	Округление без нормализации
NRM normalise	NRM{cond}prec{round} Fd,Fm	Fd:= norm Fm	Нормализация числа
SQT square root	SQT{cond}prec{round} Fd,Fm	Fd:= sqroot Fm	Квадратный корень из числа

LOG logarithm to base 10	LOG{cond}prec{round} Fd,Fm	Fd:=log Fm	Логарифм от числа
LGN logarithm to base e	LGN{cond}prec{round} Fd,Fm	Fd:=ln Fm	Натуральный логарифм от числа
EXP exponent	EXP{cond}prec{round} Fd,Fm	Fd:=eFm	Экспонента от числа
SIN sine	SIN{cond}prec{round} Fd,Fm	Fd:=sin Fm	Синус от числа
COS cosine	COS{cond}prec{round} Fd,Fm	Fd:=cos Fm	Косинус от числа
TAN tangent	TAN{cond}prec{round} Fd,Fm	Fd:=tang Fm	Тангенс от числа
ASN arc sine	ASN{cond}prec{round} Fd,Fm	Fd:=arcsin Fm	Арксинус от числа
ACS arc cosine	ACS{cond}prec{round} Fd,Fm	Fd:=arccos Fm	Аркосинус от числа
ATN arc tangent	ATN{cond}prec{round} Fd,Fm	Fd:=arctang Fm	Арктангенс от числа

Таблица 23. Команды загрузки и преобразования данных ПЗ

LDF load data	LDF{condition}prec Fd, [Rn,#offset]{!} [Rn]{,#offset}	Fd= Memory[Effect.address]	Загрузка в регистр ПЗ из памяти
STF store data	STF{condition}prec Fd, [Rn,#offset]{!} [Rn]{,#offset}	Memory[Effect.address] = Fd	Пересылка из регистра ПЗ в память
FLT integer to floating point	FLT{condition}prec{round} Fn,Rd Fn,# fp-constant	Fn=Rd	Загрузка регистра ПЗ из целочисленного регистра, или загрузка константы
FIX floating point to integer	FIX{condition}{round} Rd,Fn	Rd:=Fn	Загрузка целочисленного регистра из регистра ПЗ
LFM load floating multiple	LFM{condition} Fd, count, [Rn] [Rn,#offset]{!} [Rn],#offset	Fd++= Memory[Effect.address++]	Блочная загрузка регистров ПЗ из памяти (до четырех регистров 96 бит)
SFM store	SFM{condition} Fd, count, [Rn]	Memory[Effect.address++] = Fd++	Блочная пересылка регистров ПЗ память

floating multiple	[Rn,#offset]{!} [Rn],#offset		(до четырех регистров 96 бит)
LFM load floating multiple	LFM{condition}**SS** Fd, count, [Rn] {!} LFM FD –full stack, descending (postincrement) LFM EA – empty stack, ascending (predecrement)	Fd++= Memory [-- Effect.address ++]	Блочная загрузка регистров ПЗ из стековой памяти (до четырех регистров 96 бит)
SFM store floating multiple	SFM{condition}**SS** Fd, count, [Rn] {!} SFM FD –full stack, descending (postincrement) SFM EA – empty stack, ascending (predecrement)	Memory [--Effect.address++] = Fd++	Блочная пересылка из регистров ПЗ в стековую память (до четырех регистров 96 бит)

Таблица 24. Команды сравнения чисел ПЗ с установкой флагов и прерыванием

CMF Compare floating	CMF{condition} Fn, Fm	Z, V← Fn comp Fm	Сравнение с ПЗ чисел Fn и Fm
CMFE Compare floating with exception	CMFE{condition} Fn, Fm	Z, V, E← Fn comp Fm	Сравнение с ПЗ чисел Fn и Fm с генерацией прерывания
CNF Compare negated floating	CNF{condition} Fn, Fm	Z, V← Fn comp (- Fm)	Сравнение с ПЗ чисел Fn и (- Fm)
CNFE Compare negated floating with exception	CNFE{condition} Fn, Fm	Z, V, E← Fn comp (- Fm)	Сравнение с ПЗ чисел Fn и (-Fm) с генерацией прерывания

11 Арифметическое устройство с плавающей запятой SPARC64

11.1 Общая стуктура устройства

В качестве практического примера построения АЛУ ПЗ рассмотрим реализацию такого устройства в микропроцессоре HAL Sparc64 V (Sparc64-V) предназначенном для построения

многопроцессорных систем класса SMP. Этот микропроцессор поддерживает также расширение системы команд VIS2.0, позволяющее обрабатывать графические, фото- и видеоизображения. Микропроцессор SPARC64-V построен на суперскалярном принципе и позволяет одновременно начинать исполнение 8 команд.

АЛУ ПЗ Sparc64-V имеет два идентичных арифметических блока ПЗ, один блок для исполнения команд VIS и станцию резервирования (буфер) команд ПЗ/VIS для исполнения. Структура арифметического блока представлена на Рис. 28а**.** На каждом такте две команды обработки ПЗ/VIS или одна команда ПЗ и одна команда VIS могут быть загружены в станции резервирования, которые на самом деле являются буфером для декодированных команд обработки данных ПЗ или команд VIS. Каждый блок ПЗ имеет устройство сложения/вычитания FADD и устройство умножения FMUL.

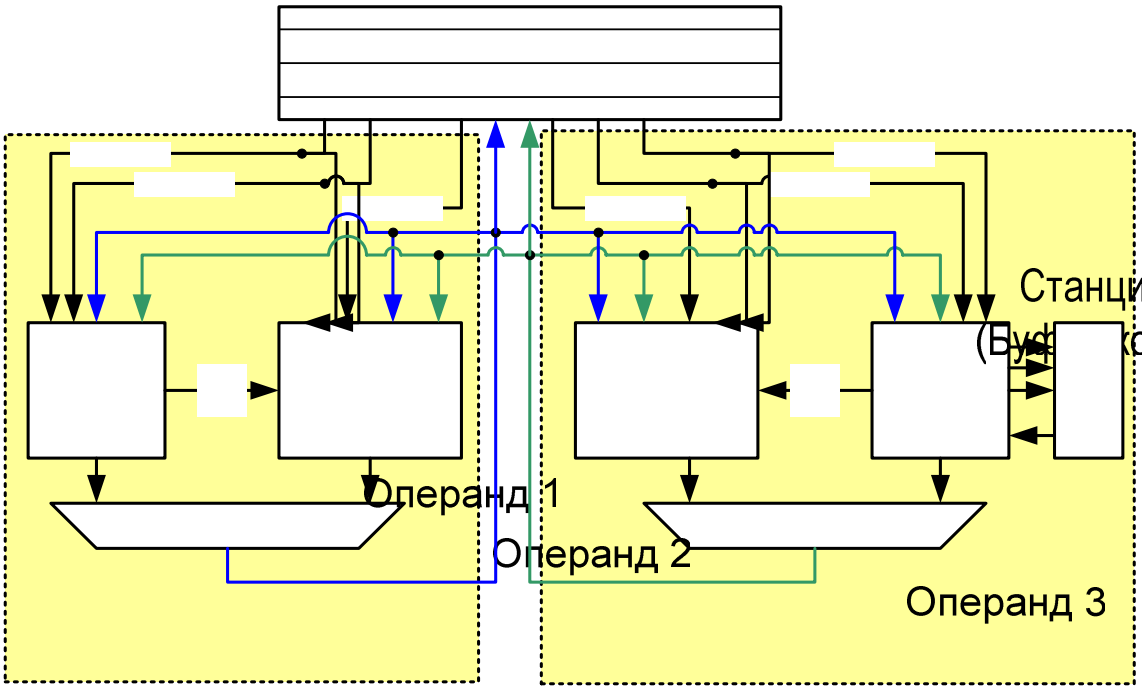
Устройство сложения FADD имеет конвейерную структуру и может исполнять за каждый такт по одной команде сложения, вычитания, преобразования форматов, сравнения или копирования данных между регистрами. Блок умножения FMUL выполняет команды умножения, деления и извлечение квадратного корня. Команда умножения может исполняться в полностью конвейерном режиме (каждый такт может стартовать по одной команде), в то время как деление и извлечение квадратного корня исполняется в многотактном режиме и новая команда не может стартовать до завершения предыдущей. Устройство умножения FMUL посылает сигнал «Завершено» на устройство управления после завершения многотактной команды и только тогда следующая команда может быть загружена в устройство умножения. Блок VIS также имеет конвейерную структуру и может исполнять все команды расширения VIS2.0.

Sparc64-V поддерживает две дополнительные команды, не относящиеся к системе команд SPARC, раздельное (unfused) умножение-сложение uFMADD и раздельное (unfused) умножение-вычитание uFMSUB. В отличие от слитного (fused) умножения-сложения, где округление производится только один раз после операции сложения, в раздельном умножении-сложении округление производится после каждой операции в этой паре (умножение-округление, сложение.вычитание-округление). Округление после операции умножения позволяет достичь совместимости результата с требованием стандарта IEEE-754 по округлению и точно детектировать ситуацию, когда необходимо прерывание по аномалии результата ПЗ.

Если в раздельной команде умножение вызывает прерывание, то последующая операция сложения/вычитания не выполняется и статус

прерывания операции умножения передается в блок обработки прерываний. Для этих команд необходимо три операнда, причем первые «Операнд 1» и «Операнд 2» подаются в блок умножения FMUL, и третий «Операнд 3» подается на блок сложения/вычитания FADD. Результат умножения FMUL пересылается на вход блока сложения/вычитания FADD через внутреннюю шину FMA, как показано на Рис. 28а. Команда сложения/вычитания выпоняется с округленным результатом умножения и «Операндом 3». Последующие команды (сложение, вычитания, конверсия, сравнение и копирование данных) не могут быть запущены в блоке сложения/вычитания на этом такте. Архитектура арифметического устройства поддерживает выдачу и исполнение двух команд раздельного умножения-сложения uFMADD/uFMSUB (соответственно двух умножений FMUL и двух сложений/вычитаний FADD/FSUB) за такт, что обеспечивает пиковую производительность в 4 GFLOPS.

Станции резервирования или буфер декодированных команд FRS имеет 6 портов чтения, по три для арифметических блоков A и B. Три операнда необходимы для команд uFMADD/uFMSUB и PDIST (VIS). Результаты команд FADD/FMUL, исполняемых в арифметическом блоке A подаются на шину результата блока A (FPA), аналогично результаты команд FADD/FMUL/VIS, выполняемых в блоке B подаются на шину результата блока B (FPB). Так как длина проводников шин результата довольно существенна и поддержка непрерывного (без пустых тактов) исполнения команд крайне важна, то половина такта резервируется для пересылки результата, который может быть операндом следующей команды. Например, если длительность исполнения команды сложения ADD – три такта, то сама операция сложения должна быть исполнена за два с половиной такта, оставляя последнюю половину такта на пересылку результата. Это позволяет командам, ожидающим результат в качестве операнда , начинать исполнение на четвертом такте.



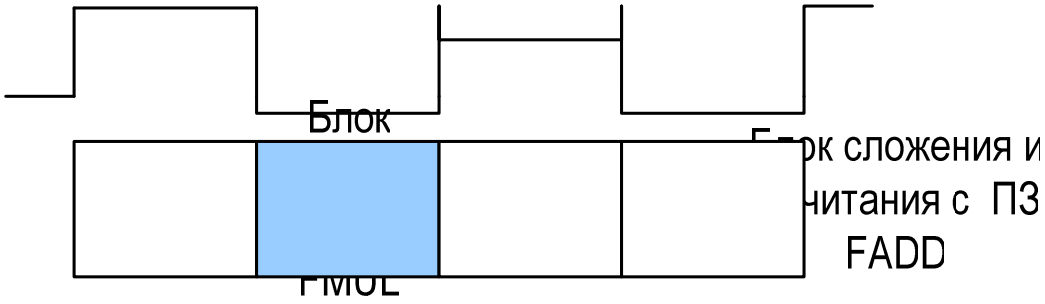


Рис. 28. Арифметическое устройство с плавающей запятой процессора HAL SPARC64.

В Таблица 25 приведены длительности исполнения команд и задержки старта новой команды для команд одинарной и двойной точности стандарта IEEE-754. Из таблицы видно, что команды сложения/вычитания и умножения являются полностью конвейеризованными и новая команда может стартовать на каждом последующем такте, в то время как остальные многотактные команды могут стартовать только после почти полного завершения исполнения предыдущей команды. Длительность исполнения команд одинарной и двойной точности одинакова, за исключением команд деления и извлечения квадратного корня. Так как команды имеют различную длительность исполнения, то устройство управления для избежания столкновений, должно диспетчировать использование шин результата. Разница между длительностью исполнения и задержкой старта для многотактных команд деления и извлечения квадратного корня объясняется тем, что новая команда может стартовать немедленно после прохождения последней итерации деления/квадратного корня

через матрицу умножителя, не дожидаясь полного завершения исполнения команды.

Таблица 25. **Длительность исполнения команд и задержка число тактов между стартом каждой новой команды**

Команды	Длительность исполнения команды	Задержка старта новой команды
FADD/FSUB	**3**	**1**
FMUL	**4**	**1**
FDIVs	**16**	**13**
FDIVd	**19**	**16**
FSQRTs	**22**	**19**
FSQRTd	**27**	**24**
uFMADD/uFMSUB	**7**	**1**

Блок VIS unit поддерживает часть набора команд VIS2.0 в той части, которая использует доступ к регистрам ПЗ. Остальная часть набора команд VIS2.0 поддерживается в целочисленной части микропроцессора. Набор VIS2.0 обеспечивает все стандартные графические команды с фиксированной запятой (логические, выравнивание, арифметические, сравнение и т.д.). Все команды VIS являются командами SIMD-типа с 8-разрядными, 16-разрядными или 32-разрядными операндами. Блок VIS получает операнды от блока FMUL и посылает результат обратно в этот же блок. Это позволяет отделить блок VIS от шин операндов и пересылки результата, исключая его влияние на задержку пересылки результата по шинам FPA and FPB для блоков FADD and FMUL. Пересылка операндов от блока FMUL в блок VIS занимает половину такта и соответственно пересылка результата от блока VIS в блок FMUL также занимает половину такта. Таким образом минимальная длительность исполнения команды в блоке VIS составляет два такта, причем только половина такта предназначена для самой операции, как представлено на Рис. 28б.

Обработка денормализованных операндов и результатов операций часто требует дополнительных аппаратных средств и может приводить к увеличению либо длительности такта, либо длительности исполнения команд. Часть набора операций с денормализованными значениями может быть реализована с использованием значения порядка без обработки мантиссы. Sparc64-V обеспечивает эти операции с минимальными аппаратными затратами. Любая другая операция над денормализованными значениями, требующая обработки мантисс, не поддерживается аппаратными средствами арифметического блока ПЗ, так как это может привести к увеличению задержек и росту длительности исполнения команд. Такие операции

поддержаны через прерывания по незавершенной команде и обрабатываются программными средствами.

11.2 Блок сложения/вычитания

Общая структура блока сложения/вычитания, представленная на Рис. 29, базируется на двух параллельных трактах (конвейерах), в каждом из которых выполняется сдвиг мантиссы только в одном направлении: в тракте 1 только влево и в тракте 2 только вправо. Эта структура значительно отличается от ранне рссмотренного канонического устройства сложения, представленного на Рис. 24.

Тракт 1 используется для эффективного исполнения операций вычитания при разнице порядков равном нулю или единице. Тракт 2 используется для операций сложения/вычитания, когда разница порядков превышает единицу. В тракте 1, тем не менее, может понадобиться шаг округления из-за значения самого младшего граничного (guard) бита. Этого можно избежать, если проанализировать значение старших битов MSB мантиссы. Если поставить условие, что значение мантиссы операнда с большим порядком должно быть меньше 1.5, тогда уменьшаемое в операции вычитание попадает в интервал [1, 1.5), и тогда опреация вычитания с сдвинутым на один разряд вправо мантисой вычитаемого в интервале [0.5, 1) дает результат, попадающий в интервал [0, 1]. Это потребует сдвига влево на один разряд для нормализации результата в интервал [1, 2). Возможный граничный бит будет сдвинут в позицию LSB, ликвидируя необходимость в последующем округлении. Критерии выбора результата из трактов 1 и 2 в зависимости от значений операндов представлены в Таблица 26.

Таблица 26. Выбор тракта для исполнения команд сложения и вычитания

Тракт 1	Тракт 2
Вычитание с разницей порядков «0»	Вычитание с разницей порядков больше чем «1»
Вычитание с разницей порядков «1» и значением мантиссы с большим порядком меньше чем «1.5»	Вычитание с разницей порядков «1» и значением мантиссы с большим порядком больше или равно «1.5»
	Сложение

Операции в тракте 2 обычно требуют сдвига мантиссы вправо и последующего округления. Сложение и вычитание с разностью порядков, больше чем единица, требуют выравнивания порядков с соответственным сдвигом мантиссы меньшего операнда вправо. Вычитание с разницей порядков в единицу требует сдвига на один разряд вправо мантиссы меньшего операнда. Так как вычитание в этом тракте ограничено интервалами значения уменьшаемого [1.5, 2) и вычитаемого [0, 1), то результат всегда будет попадать в интервал (0.5,

2). В этом случае нет необходимости в сдвиге мантиссы результата вправо на один разряд для нормализации и возможный сдвиг влево может быть выполнен на первой ступени выбора результата.

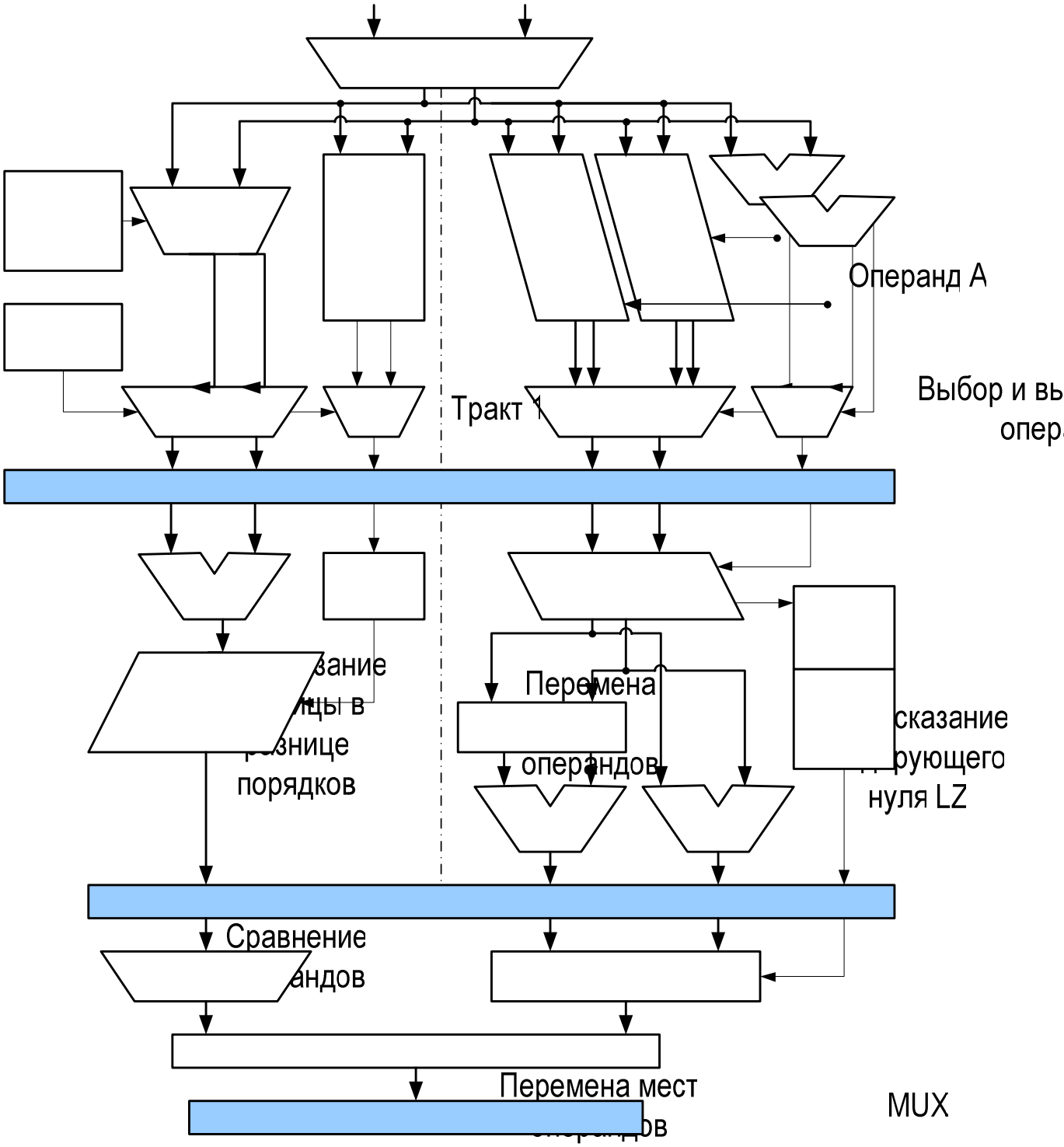


Рис. 29. Устройство сложения чисел с плавающей запятой процессора HAL SPARC64.

11.3 Описание Тракта 1 устройства сложения-вычитания

Значение разности порядков «ноль» или «единица» могут быть предсказаны с использованием двух самых младших разрядов порядка каждого операнда. Это же предсказание используется для определения позиций операндов таким образом, чтобы выравнивание мантисс и предсказание лидирующего нуля были завершены на первой ступени конвейера.

Во избежание получения отрицательного значения мантисы результата, который потребует последующего преобразования в дополнительный код, меньшая мантисса всегда вычитается из большей. Для разности порядков в единицу, предсказание может быть использовано для определения позиций операндов, для чего используются соответствующие мультиплексоры перемены мест операндов. Для нулевой разницы порядков, сравниваются полные значения мантисс и определяется мантиса с большим значением, которая должна встать в позицию уменьшаемого, при этом используется второй уровень мультиплексоров перемены мест операндов.

Операнды с нулевой или единичной разницей порядков могут порождать результирующую мантиссу, содержащую один или более лидирующих нулей, начиная со старших разрядов. Алгоритм, который используется для предсказания вектора лидирующих нулей Z, предполагает, что меньшая мантисса ($B = b_{63}\ldots b_0$) вычитается из большей ($A = a_{63}\ldots a_0$) и выполняет вычитание в дополнительном коде с использованием инвертирования вычитаемого B. Такой порядок операндов гарантируется использованием схем предсказания разницы порядков в единицу и результатом сравнения операндов.

$z_i = !((a_i \oplus !b_i) + (!a_{i-1} \& b_{i-1} i ;));$ i=1 до 63

Этот алгоритм не использует цепочку переносов и гарантирует, что предсказание вектора лидирующих нулей может содержать, по крайней мере один лидирующий нуль. Ошибка предсказания может быть позже скорректирована на третьей ступени конвейера с помощью сдвига мантиссы результата на один разряд влево и соответствующей коррекции порядка.

На второй ступени конвейера подсчитывается и кодируется предполагаемое число лидирующих нулей LZC. Одновременно производится вычитание мантисс. Пост-операционный сдвиг влево выполняется через последовательность мультиплексоров, управляемых декодированным результатом LZC.

Финальной операцией в тракте 1 является коррекция по числу лидирующих нулей, которая выполняется на третьей ступени

конвейера. Так как на выходе блока должны быть только нормализованные результаты, то ошибка предсказания может быть детектирована путем проверки значения скрытого бита MSB. Если он равен нулю, то произошла ошибка в предсказании вектора лидирующих нулей и она должна быть исправлена с помощью сдвига мантиссы результата влево на один разряд для получения конечного нормализованного значения. Этот результат далее посылается на мультиплексоры выбора результатов из Тракта 1 или Тракта 2, и далее на шину результата.

11.4 Описание Тракта 2 устройства сложения-вычитания

В тракте 2 число разрядов для сдвига вправо мантиссы одного из операндов сложения/вычитания определяется из разности порядков операндов. До того, как определены позиции операндов, производится одновременно вычитание порядков операндов (A – B) и (B – A). Младшие биты результата вычитания порядков используются для частичного правого сдвига обоих мантис на 0, 1, 2 или 3 разряда. После того, как вычитание порядков завершено, выбирается частично сдвинутая мантисса меньшего операнда. Во избежание последующего преобразования отрицательного результата в дополнительный код, мантисса с меньшей экспонентой устанавливается в позицию вычитаемого.

На второй ступени конвейера Тракта 2, полный результат вычитания значений порядков используется для завершения сдвига вправо мантиссы меньшего операнда. По мере получения выдвигаемых вправо значений разрядов мантиссы, параллельно вычисляются значения битов, необходимых для округления (LSB+1, LSB, граничный, округления, избыточные-sticky).

Для правильного округления согласно стандарту IEEE-754, нужны значения (A+B), (A+B+1) и (A+B+2). Они могут быть вычислены с использованием только двух сумматоров для получения (A+B) и (A+B+2); число (A+B+1) может быть получено из этих двух результатов. Ступень с сумматором сохранения переноса CSA добавлена перед сумматором (A+B+2) для прибавления единицы в разряд LSB+1 для одинарной точности и разряды 41 и 12 для двойной точности. Такая линейка CSA сумматора используется вместо сумматора с разделительными флагами, который требует специальной конфигурации цепи переноса для вставки бита округления в различные разряды мантиссы. Без таких цепей переноса топология сумматора может быть значительно упрощена и также повышена скорость его работы.

Таблица 27 иллюстрирует выбор результатов сложения, который базируется на комбинации значений битов переполнения (MSB + 1),

признака округления и младшего бита LSB. Бит «признака округления» показывает условный инкремент пренормализованного результата для выполнения округления по стандарту IEEE-754 для всех режимов округления и вычисляется с учетом режима округления, знака, граничного бита, бита округления и избыточных битов.

Таблица 27. Выбор результата сложения для тракта 2

Переполнение MSB+1	Бит признака округления	LSB	Результат с двойной точностью	Результат с одинарной точностью
1	1	-	A + B + 2[64 :12]	A + B + 2[64 :41]
1	0	-	A + B[64 :12]	A + B[64 :41]
0	1	1	A + B +2[63:12], !(A+B[11])	A + B +2[63:41], !(A+B[40])
0	1	0	A + B[63:12], !(A+B[11])	A + B[63:41], !(A+B[40])
0	0	-	A +B [63:11]	A +B [63:40]

Результаты вычитания в тракте 2 находятся в интервале (0.5, 2), который может потребовать сдвига мантиссы результата влево на один разряд для нормализации, если результат меньше единицы. В случае сдвига влево, разряд границы округления изменится с LSB на граничный бит. Два возможных варианта границы округления могут быть обработаны с использованием двух битов заполнения (fill bits) в разрядах LSB+1 и LSB, которые комбинируются с результатами (A+B) и (A+B+2). К примеру, когда разряд LSB и граничный бит оба равны единице, тогда генерируется признак округления, который отражается в выборе значения (A+B+2) для старших разрядов и обнулении младших разрядов в процессе округления. Таблица 28 иллюстрирует в деталях, как биты заполнения (*fill* bits) комбинируются с результатами (A+B) и (A+B+2) для получения конечного результата.

Алгоритм округления, используемый в устройстве сложения, обеспечивает все четыре режима округления стандарта IEEE-754. Выбор округленного результата выполняется на третьей ступени конвейера, за которой следует выбор между результатами трактов 1 или 2. Далее результат транслируется на шину результата.

Таблица 28. Выбор результата вычитания для тракта 2

Бит округления	MSB	LSB	Граничный бит	Результат с двойной точностью	Результат с одинарной точностью
0	0	-	-	A + B[62:10]	A + B[62:39]
1	0	0	0	A + B[62:12], 01	A + B[62:41], 01
1	0	0	1	A + B[62:12], 10	A + B[62:41], 10

1	0	1	0	A + B[62:12], 11	A + B[62:41], 11
1	0	1	1	A + B + 2[62:12], 00	A + B + 2[62:41], 00
0	1	-	-	A + B[63:11]	A + B[63:40]
1	1	0	0	A + B[63:12], 0	A + B[63:41], 0
1	1	0	1	A + B[63:12], 1	A + B[63:41], 1
1	1	1	0	A + B[63:12], 1	A + B[63:41], 1
1	1	1	1	A + B + 2[63:12], 0	A + B + 2[63:41], 0

Устройство сложения используется также выполнения дополнительных команд преобразования чисел из формата ПЗ в целочисленные и наоборот, а также для преобразования из одного формата ПЗ в другой (одинарный в двойную и наоборот). Преобразование из формата ПЗ с одинарной точностью в двойную и преобразование из целочисленного формата в формат с ПЗ производится в тракте 1. Для преобразования чисел ПЗ из одинарной в двойную точность используется сдвиг влево для нормализации, если входной операнд является денормализованным числом. Преобразование целого числа в формат ПЗ может потребовать округления после сдвига влево для нормализации, чтобы учесть младшие разряды преобразуемого числа. Так как стандартное вычитание в тракте 1 не использует округление, то дополнительный такт используется для выполнения округления преобразованного числа ПЗ.

Преобразование из двойной точности в одинарную и из формата ПЗ в целое производится в тракте 2, так как преобразование числа в одинарную точность требует полномасштабного округления. Преобразование из формата ПЗ в целое использует сдвиг вправо на число разрядов, равное значению порядка, что необходимо для выравнивания результата с фиксированной запятой и превращения его в целое число.

11.5 Устройство умножения и деления чисел с ПЗ процессора HAL SPARC64

Устройство умножения FMUL имеет четырехступенчатый конвейер который выполняет операции умножения, деления и извлечения квадратного корня. SIMD операции умножения графических данных VIS2.0 также выполняются в этом устройстве. Умножение ПЗ полностью конвейерное и требует три с половиной такта, последняя половина четвертого такта используется пересылки результата. Операции деления и извлечения квадратного корня не являются конвейерными и используют многотактовую реализация с несколькими последовательными проходами через матричный умножитель. Новые команды не могут диспетчироваться в устройство до окончания исполнения операций деления/извлечения квадратного

корня. Тракт мантисы устройства умножения проиллюстрирован на Рис. 30. Матричный умножитель мантиссы поддерживает формат 60x60, обеспечивая деление и извлечение квадратного корня с округлением по стандарту IEEE-754. Этот же матричный умножитель может поддерживать четыре 8x16 умножения со знаком в режиме SIMD для графических команд VIS.2.0. Если в предыдущей секции мы рассматривали отдельные устройства умножения и деления ПЗ, то здесь реализовано совмещенное устройство, которое использует общий матричный умножитель и ступени округления.

В умножителе используется избыточное кодирование «знак-цифра» Буфа по основанию 4 с диапазоном [-2, 2]. Ниже приведена Таблица 29 для выбора частичного произведения при анализе трех разрадов множителя.

Таблица 29. Выбор частичного произведения в соответствии с кодированием Буфа

Разряды множителя	Выбор частичного произведения	Знак кодирования Буфа в разряде $\mathbf{P_{bs}}$	Представление в «знак-цифра»
000	Прибавить "0"	+	0
001	Прибавить множимое +M	+	+1
010	Прибавить множимое +M	+	+1
011	Прибавить удвоенное множимое +2M	+	+2
100	Отнять удвоенное множимое -2M	-	-2
101	Отнять множимое -M	-	-1
110	Отнять множимое -M	-	-1
111	Отнять "0"	+	0

Первая ступень конвейера умножителя выполняет выравнивание операндов вместе с кодирование множителя в формат Буфа. Для кодирования 60-разрядных операндов в формат Буфа с основанием 4 (radix-4) и основанием 8 (radix-8) требуется 4 уровня сумматоров с

сохранением переноса 4→2 CSA. Основание 4 выбрано чтобы избежать дополнительных задержек при генерации частичных произведений для значений кода Буфа +3 и -3. Модифицированное кодирование Буфа используется для генерации 31-го частичного произведения в дополнительном коде для возможности умножения отрицательных чисел, которые должны быть расширены с учетом знака. Это расширение с учетом знака обеспечивается добавлением двух дополнительных разрядов к вектору частичного произведения, в результате получается 63-разрядный операнд к которому также добавлен знак кодирования Буфа P_{bs}, как показано на последующих уравнениях.

$PP_0 = !P_s\ P_s\ P_{60}\ P_{59}\ P_{58} \ldots\ldots\ldots\ P\ P_0$

P_{bs} знак прибавляемого частичного произведения

Соответственно остальные частичные произведения i от 30 to 1 будут выглядеть следующим образом:

$PP_i = 1\ !P_s\ P_{60}\ P_{59}\ P_{58}\ \ldots\ldots\ldots\ P_1\ P_0$

P_{bs} знак прибавляемого частичного произведения

Знак частичного произведения или значение бита P_{bs}, равно 0 для кодированных значений Буфа 0, 1, или 2, и Pbs равен 1 для кодированных значений -1 and -2. Для умножения без знака, бит P_s равен 0 для кодов Буфа 0, 1, и 2, и бит Ps равен 1 для значений –1 and -2. Для множителя со знаком бит $P_s = P_{59}$ кода Буфа 0, 1, и 2, и бит $P_s = P_{58}$ для значений –1 и –2.

На второй ступени конвейера число частичных произведений уменьшается с использованием дерева сумматоров Уоллеса, которое построено на использовании суматоров с сохранением переноса 4→2 CSA. 32 оставшиеся суммируемых операнда формируются путем комбинирования 31 частичного произведения со слагаемым, которое используется вычисления ошибки в итерации деления и извлечения квадратного корня. Каждый 4→2 уровень CSA уменьшает в два раза число слагаемых. Соответственно, требуются 4 уровня сумматоров CSA для уменьшения числа слагаемых до двух (суммы и переноса) в избыточном формате CSA.

В обычном исполнении дерева сумматоров Уоллеса выходы трех первых уровней сумматоров CSA требуют сдвига для выравнивания частичных сумм на 8, 16 и 32 разряда соответственно. Такая конструкция может иметь чрезмерные задержки в проводниках при передаче сигналов с уровня на уровень. Для уменьшения задержек матрица умножителя перегнута по диагонали, как это показано на Рис. 31.

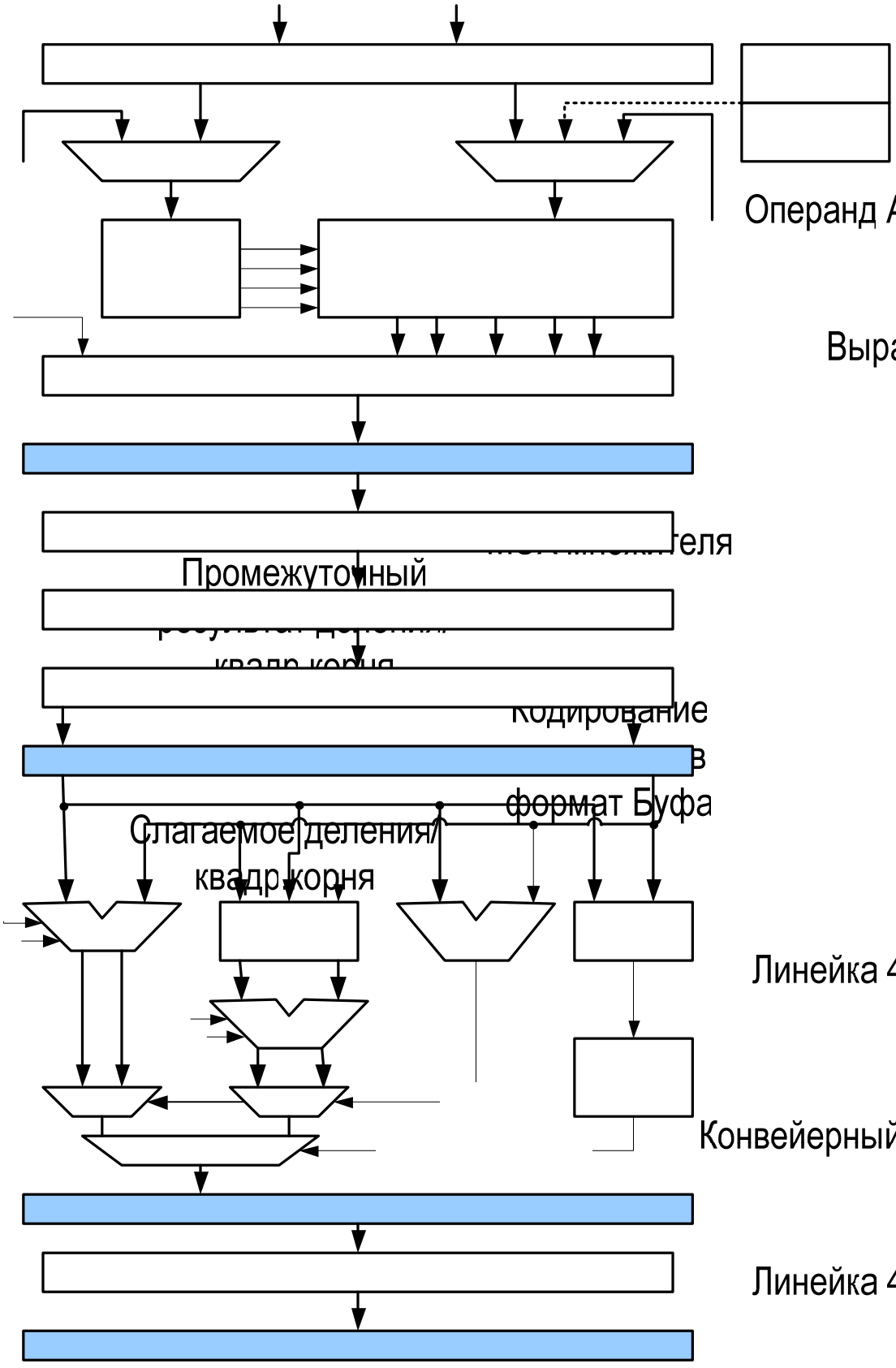

Операнд A
Выравнивани
Промежуточный
Кодирование
формат Буфа
Слагаемое деления/
квадр.корня
Линейка 4 : 2 CSA
Конвейерный регистр
Линейка 4 : 2 CSA
Линейка 4 : 2 CSA
Линейка 4 : 2 CSA

Рис. 30. Устройство умножения и деления чисел с плавающей запятой процессора HAL SPARC64.

Множитель передается по диагонали и частичные произведения уже выравнены таким образом, что не требуется сдвига для значений сумм и переносов на выходе CSA. Так как критический путь в умножителе проходит через генерацию частичных произведений, то такой сдвиг результатов на каждом уровне не влияет на задержку в проводниках. После того, как сгенерированы все частичные произведения, их надо сложить в CSA, если при этом не производится сдвиг, то сильно уменьшается задержка на проводниках. Такой подход является значительным улучшением предшествующих вариантов реализации, когда для выравнивания выполнялся сдвиг и задержка на матрице CSA была значительно большей. Выходы столбца b_{58} используются сумматорами CSA столбца b_{59}. Длинное проводное соединение от столбца b58 к столбцу b59 ликвидируется путем дублирования столбцов b58..b52 в младшей части матрицы, как это показано на рис.3.31. Матрица также структурирована таким образом, чтобы генерировать результат в обычном двоичном формате (не избыточном) таким образом, чтобы группировать вместе верхние старшие биты результата и младшие избыточные биты, что позволяет избежать конечного 64-разрядного сдвига выхода сумматоров CSA.

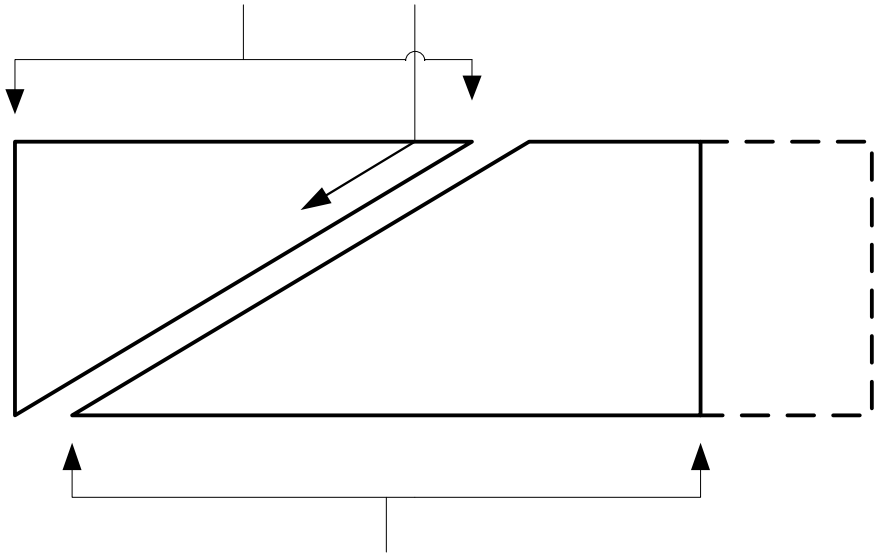


Рис. 31. Перегиб матрицы умножителя для уменьшения задержек

На третьей ступени конвейера двоичные векторы сумм и переносов на выходе последнего уровня CSA должны быть просуммированы с использованием сумматора с предсказанием

переноса CLA, чтобы получить результат в обычном двоичном представлении. Выходной перенос из младших 60 разрядов генерируется специальной схемой, которая обеспечивает значение входного бита переноса, используемого для выбора значения результата в двухканальном сумматоре. Избыточный бит округления генерируется параллельно с выходным переносом, непосредственно используя избыточное представление на выходе CSA, согласно уравнения ниже.

$$p_i = s_i \oplus c_i \;\; ; \;\; k_i = !s_i \cdot \& \; !c_i \;\; ; \;\; z_i = !(p_i \oplus k_{i-1}), \;\; \text{для } i=1 \text{ до } 58$$

В этом выражении S_i и C_i являются разрядами суммы и переноса на выходе последнего уровня CSA. Сигнал распространения P_i, и сигнал уничтожения K_i , генерируемые логической схемой на 60 разрядов, используется для генерации вектора Zi. Избыточный бит округления (sticky bit) является результатом проводного ИЛИ всех битов вектора Zi.

Существуют несколько версий алгоритмов округления для умножителей стандарта IEEE-754. Версия, выполненная в устройстве, может быть описана со ссылкой на Рис. 30. Значения сумм с подразумеваемым переполнением и отсутствием оного генерируются со значением входного переноса и без него. Для этого имеются два канала сдублированных 64-разрядных сумматоров, выходы которых выбираются через соответствующие мультиплексоры. Двухканальная схема сумматора соответствует двухканальному сумматору с выбором результата, представленным ранее на Рис.13.

Затем выполняется округление путем двухуровневой селекции результата, первая селекция базируется на значении бита входного переноса от логики детектора переноса и второй уровень базируется на значении бита переполнения, бита округления, избыточного бита и режима округления. Выбор результата округления очень похож на используемый в устройстве сложения/вычитания (

Таблица 27, Таблица 28). На четвертой ступени конвейера тракты мантисы и порядка выравниваются и объединяются, конечный результат засылается на шину результата.

11.6 Деление и извлечение квадратного корня

Операции деления и извлечения квадратного корня выполняются путем нескольких итераций через матричный умножитель и поэтому не являются конвейерными. Диспетчер не посылает новых команд умножения/деления/извлечения кв.корня и VIS команд на устройство умножения до момента завершения исполнения текущей команды деления/извлечения квадратного корня. Команда деления/извлечения кв.корня посылает сигнал «Завершено» (Done) диспетчеру за 6 тактов

до момента, когда она будет готова выдать результат на шину результата. На шестом такте после выдачи сигнала завершения команда деления/извлечения кв.корня занимает выходную шину и выдает результат.

Команды деления/извлечения кв.корня используют итеративный алгоритм конвергенции Голдшмидта (Goldschmidt). Оба операнда A и B должны быть нормализованными со значениями мантисс в интервале [1,2) и результатом является нормализованное значение A/B в интервале [1,2) для деления. Для команды извлечения квадратного корня результатом является нормализованное значение sqroot B в интервале [1,2).

Для деления стартовым значением для начальной аппроксимации является величина 1/B, которая получается в результате считывания через индексирование старшими 7 битами мантиссы B (за исключением скрытого бита) специальной таблицы, содержащей 128 значений по 10 бит каждое. Допустим, что F_0 является результатом, полученным из таблицы аппроксимации. Последовательность итераций показана на уравнении ниже:

$Q_1 = \text{floor}(A*F0),\ G_1 = \text{ceiling}\ (B*F_0)$

$F_i = \text{floor}\ (2 - G_i),\ Q_{i+1} = \text{floor}\ (Q_i*F_i),\ G_{i+1} = \text{ceiling}\ (G_i*F_i)$ для i = 1, 2, 3.

Для квадратного корня в качестве стартового значения аппроксимации берется 1/B или 1/2B, которое может быть получено путем индексирования одной из двух таблиц, одна для четного значения порядка операнда B и другая для нечетного. Обе таблицы имеют по 128 значений, по 9 бит каждое. Допустим, что F_0 является результатом, полученным из одной из таблиц аппроксимации. Последовательность итераций для извлечения квадратного корня показана на уравнении ниже:

$Q_1 = \text{floor}(B * F_0),\ G_1 = \text{ceiling}\ [(B * F_0) * F_0]$

$F_i = \text{floor}\ [½\cdot (3 - G_i)],\ Q_{i+1} = \text{floor}\ (Q_i * F_i),\ G_{i+1} = \text{ceiling}\ [(G_i * F_i) * F_i],$ для i = 1 ,2, 3

Индексируемые таблицы стартовых значений деления и квадратного корня имеют минимально необходимый размер для обеспечения операций с двойной точностью, доступ к таблицам выполняется в одном такте. Создание отдельных таблиц для различной точности было признано решением, не обеспечивающим такой же скорости доступа, как единая таблица. 4 таблицы с шириной 10 разрядов и 14-разрядный сумматор необходимы для уменьшения числа итераций через умножитель на единицу. Но затраты на

дополнительную аппаратуру превышают выгоды от уменьшения задержки исполнения команды.

Стартовые значения аппроксимации (в таблицах) всегда выбираются меньше, чем реальные значения бесконечной точности, что уменьшает максимальную относительную ошибку в интервале [1,2). В итерациях деления и извлечения кв.корня (см. уравнения выше) каждое умножение выполняется в формате 60x60 бит, где floor и ceiling означают округление к меньшему и округление к большему 120-разрядного результата умножения в формат 60 бит. Округление к большему аппроксимируется путем добавления единицы в младший разряд LSB (бит 60). Каждое из промежуточных произведений округляется к максимальной точности в 60 бит, чтобы минимизировать аккумулируемую в ходе итераций ошибку.

Коэффициенты итеративной сходимости (2 – Gi) и [½* (3 – Gi)] вычисляются путем преобразования в обратный код и сложения с константами (1 для деления и 3/2 для кв.корня) в неиспользуемых уровнях CSA сумматора. Для реализации, описанной выше, Q3 (результат третьей итерации) будет достаточен для операций одинарной точности. Для операций двойной точности необходима четвертая итерация с Q4 на выходе, результат которой будет удовлетворять требованиям этого формата. Итерации через матричный умножитель используют только первые три ступени конвейера, четвертая ступень используется только для округления по стандарту IEEE-754 и пересылки результата на выходную шину.

Округление 120-разрядного результата Q (Q3 для одинарной точности и Q4 для двойной) с выхода умножителя выполняется после финальной итерации алгоритма конвергенции для получения результата, удовлетворяющего требованиям IEEE-754. В алгоритме округления остаток вначале вычисляется как $C = (A - Q_c * B)$ для деления и как $C = (B - Q_c^2)$ для кв.корня, используя один проход через конвейер умножителя. При этом Qc получается путем округления вверх значения Q до ½ ulp (наименьшего шага изменения значения), что составляет 25 бит для одинарной точности и 54 бита для двойной. Правильный результат затем выбирается из группы значений trunc(Q), trunc(Q + ½ ulp), and trunc(Q + 1 ulp), где trunc(x) – округленное вниз значение в 24 бит для одинарной точности и в 53 бит для двойной. Значение $ulp=2^{-23}$ для одинарной точности и $ulp=2^{-52}$ для двойной. Эти три результата на самом деле генерируются для округления в операции умножения, поэтому никаких дополнительных аппаратных средств для округления результата деления/кв.корня не требуется. Выбор результата базируется на значениях LSB и бита округления результата

Q, значении остатка C (три случая C>0, C=0, C<0), знаке ожидаемого результата и заданном режиме округления.

Деление с одинарной и двойной точностью имеет длительности исполнения в 16 и 19 тактов соответственно, операции извлечения кв.корня имеют длительности в 22 такта для одинарной и в 27 тактов для двойной точности. Задержка между стартом двух последовательных команд на три такта короче, чем длительность исполнения, так как последующая операция может быть начата в момент, когда результат последней итерации деления/кв.корня переместился на вторую ступень конвейера. Последовательность тактов выполнения деления с одинарной точностью проиллюстрирована в Таблица 30. Один пустой такт между завершением вычисления Q и началом вычисления остатка C необходим для генерации трех возможных значений результата для округления. Длительность исполнения и задержки между стартом команд для двойной точности будут выглядеть аналогично, за исключением добавления еще одной итерации (три дополнительных такта для Q4).

Таблица 30. Иллюстрация работы конвейера при выполнении деления с одинарной точностью.

Такты	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Начальное значение	F0															
1-я итерация		G1	G1	G1												
			Q1	Q1	Q1											
2-я итерация					G2	G2	G2									
						Q2	Q2	Q2								
3-я итерация									Q3	Q3	Q3					
												ПТ				
Остаток и результат													C	C	C	Рез-т

Правильность реализации алгоритмов деления и извлечения квадратного корня была верифицирована путем нахождения максимального значения возможной ошибки. Возможны два источника

ошибки: ошибка аппроксимации, присущая самому алгоритму и вычислительная ошибка из-за ограниченного размера таблиц и матричного умножителя. Эти ошибки генерируются и аккумулируются в ходе требуемого числа итераций. Начальный размер таблицы и стартовые значения, размерность умножителя и промежуточное округление гарантируют, что финальный 120-разрядный неокругленный результат Q будет удовлетворять следующим условиям.

Операция деления: $(q - ½\ ulp) < Q < q$ для $q \geq 1$, и $(q - ¼\ ulp) < Q < q$ для $q < 1$

Операция кв.корня: $(q - ¼\ ulp + ulp^2) < Q < q$

Здесь $ulp = 2^{-23}$ для одинарной точности и $ulp = 2^{-52}$ для двойной точности, а q является идеальным результатом с неграниченной точностью. Этот интервал значений Q является достаточным для обеспечения правильного округления во всех режимах IEEE-754.

Для специальных случаев, когда один из операндов является нулем, бесконечностью, QNAN или SNAN, в операциях деления и извлечения квадратного корня не выполняются последующие итерации через матричный умножитель. Результаты для таких случаев генерируются специальной логикой на первом же проходе через умножитель. Эта особенность раннего завершения операции позволяет уменьшить длительность исполнения команд деления/кв.корня в случае вырожденных операндов до 10 тактов.

11.7 Обработка денормализованных значений

В общем случае, поддержка обработки денормализованных опреандов требует получения результата также в денормализованном виде. Требования стандарта IEEE-754 требует определения лидирующих нулей, сдвига мантисы с коррекцией порядка и специального режима округления. Безусловно, полная поддержка обработки денормализованных операндов увеличивает аппаратные затраты и длительность исполнения команд. Однако частичную поддержку возможно обеспечить минимальными средствами, которые сравнивают значения порядков. При этом возможна поддержка довольно большого подмножества операций над денормализованными значениями.

Рассмотрим случай умножения, когда оба операнда денормализованы, в этом случае результат гарантировано выпадает из динамического интервала реальных значений в независимости от числа лидирующих нулей в мантиссе. Если один из операндов денормализован, и результирующий порядок (esrc1 + esrc2 – bias)

будет меньше, чем -25 для одинарной точности или -54 для двойной точности, то это гарантирует, что результат также выпадает из динамического диапазона представления, вне зависимости от числа лидирующих нулей в мантиссе денормализованного операнда. Здесь bias – значение смещения, используемое для кодирования порядков. В двух случаях округление выдает наименьшее денормализованное число: в режиме округления к положительной бесконечности для положительного значения и в режиме округления к отрицательной бесконечности для отрицательного значения.

Все другие режимы округления будут выдавать в качестве результата нуль. Такие экстремальные случаи вырождения результата эффективно обрабатываются специальной логикой. Все другие случаи, которые не обрабатываются аппаратными средствами, вызывают прерывание (trap) как незавершенные операции ПЗ. Таблица 31 иллюстрирует обработку денормализованных значений для различных команд, порядок операндов и результата обозначается как esrc1, esrc2, и eres (все в смещённом коде). Команды с денормализованными операндами/результатом вызывают прерывание по незавершенной операции ПЗ. Специальная программная утилита эмулирует незавершенную команду и генерирует правильный результат и соотвествующие флаги.

Таблица 31. Обработка денормализованных значений

Команды	1 денормализованный операнд, 1 нормализованный операнд	2 денормализованных операнда	Денормализованный результат
FADD{s,d}, FSUB{s,d}	Незавершенная операция ПЗ	Незавершенная операция ПЗ	Незавершенная операция ПЗ, если eres < 1
FMUL{s,d}	Незавершенная операция ПЗ, если -25 < (esrc1 + esrc2 - 126) (о.т.) -54 < (esrc1 + esrc2 - 1022) (д.т.)	Обрабатывается в аппаратуре	Незавершенная операция ПЗ, если -25 < eres < 1 (о.т.) -54 < eres < 1 (д.т.)
FDIV{s,d}	Нормал./Денормал.: Незавершенная операция ПЗ, если (esrc1 – esrc2 – 1) < 128 (о.т.) (esrc1 – esrc2 – 1) < 1024 (д.т.) Денормал./Нормал.:	Незавершенная операция ПЗ	Незавершенная операция ПЗ, если -25 < eres < 1 (о.т.) -54 < eres < 1 (д.т.)

	Незавершенная операция ПЗ если -25 < (esrc1 - esrc2 + 126) (о.т.) -54 < (esrc1 - esrc2 + 1022) (д.т.)		
FSQRT{s,d}	Незавершенная операция ПЗ для положительного операнда		

11.8 Особенности и конструктивное исполнение арифметического устройства SPARC64

Описанное АУ с ПЗ представляет собой надежный и высопроизводительный прибор, совместимый с требованиями стандарта IEEE-754 и дополнительно поддерживающий набор графических команд VIS. В устройстве использован инновационный подход, позволяющий избежать округления в тракте 1 блока сложения/вычитания, что позволило уменьшить размеры и повысить тактовую частоту до 1 ГГц. Усовершенствование алгоритма сложения позволило ликвидировать ступень постнормализации в обоих трактах (в отличие от традиционного сумматора). Использование параллельно генерируемых результатов суммы и суммы+2 позволило упростить округление для операций сложения/вычитания и умножения. Использование перегиба матрицы умножителя позволило получить хорошо сбалансированное дерево сумматоров Уоллеса с минимально возможным размером и длиной проводников. Дополнительный тракт пересылки результатов в умножителе для итераций деления и извлечения квадратного корня уменьшает длительность исполнения этих команд на 3 и 5 тактов соотвественно. Частичная поддержка обработки денормализованных значений и раннее завершение операций в специальных случаях для деления и извлечения квадратного корня увеличивает общую производительность АУ с минимальными издержками и без влияния на тактовую частоту.

Таблица 32 описывает конструктивные особенности технологии изготовления и численные параметры кремниевой реализации арифметического устройства.

Таблица 32. Конструктивные характеристики арифметического устройства ПЗ SPARC64.

Способ проектирования	Заказная СБИС

Процесс	0.15u, 6 слоев металлизации CMOS
Число транзисторов	1.9 Million
Плотность транзисторов	87 KT/mm^2
Размер блока на кристалле	9.1 x 2.4 mm^2
Тактирование	1ns @ 1.5V, 85 °C

11.9 Список литературы на детали реализации арифметического устройства SPARC64

Описание устройства было сделано на основе статьи Naini, A.; Dhablania, A.; James, W.; Das Sarma, D. «1 GHz HAL SPARC64R Dual Floating Point Unit with RAS features», Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-01), 2001. Page(s):173 – 183

Эта статья в свою очередь цитирует следующие источники, которые могут быть полезны для углубленного изучения проблемы проектирования арифметических устройств с ПЗ.

1. “The SPARC Architecture Manual Version “, SPARC International, Inc., Menlo Park, CA 1994.
2. “Visual Instruction Set Users Guide”, Sun Microelectronics, Sun Microsystems, 1995.
3. N. T. Quach and M. J. Flynn,”An Improved Algorithm for High-Speed Floating-Point Addition” Technical Report CSL-TR-90-442 Stanford University, August 1990.
4. A. Beaumont-Smith, N. Burgess, S. Lefrere and C.C. Lim, “Reduced Latency IEEE Floating-Point Standard Adder Architectures,” in Proc. 14th Symp. on Computer Arithmetic, pp 35-42, 1999.
5. E. Hokenek, R. K. Montoye, “Leading-zero anticipator (LZA) in the IBM RISC System/6000 floating-point execution unit”, in IBM Journal of Research and Development, pp 71-77, Jan. 1990.
6. Suzuki et. al., “Leading-Zero Anticipatory Logic for High-Speed Floating Point Addition” in Journal of Solid State Circuits, pp 1157-1164, Aug. 1996.
7. “IEEE Standard for Binary Floating-Point Arithmetic”, IEEE Standard 754-1985, IEEE Computer Society, New York 1985.
8. A. D. Booth, “A signed multiplication technique,” in Quarterly J. Mechan. Appl. Math, vol. 4, pt. 2, pp. 236-240, 1951.
9. I. Koren, “Computer Arithmetic Algorithms,” Prentice Hall, 1993.

10. C. S. Wallace, “A suggestion for parallel multipliers,” in IEEE Trans. Electron. Comput., vol. EC-13, pp. 14-17, 1964.
11. R. K. Yu and G. B. Zyner, “167 MHz radix-4 floating point multiplier,” in Proc. 12th Symp. Comput. Arithmetic, pp 149-154, 1995.
12. M. R. Santoro, G. Bewick and M. A. Horowitz, “Rounding Algorithms for IEEE Multipliers,” in Proc. 9th Symp. Computer Arithmetic, pp 176-183, 1989.
13. R.E. Goldschmidt, “Applications of division by convergence,” in M. S. thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, Mass., June 1964.
14. C. V. Ramamoorthy, J. R. Goodman and K. H. Kim, “Some properties of iterative square-rooting methods using high speed multiplication,” in IEEE Trans. Comput., C-21(8), pp 837-847, Aug. 1972.
15. D DasSarma and David W. Matula, “Measuring the accuracy of ROM reciprocal tables,” in Proc. 11th Symp. Computer Arithmetic, pp 95-102, 1993.
16. E. Schwartz, “Rounding for quadratically converging algorithms for division and square root,” in Proc. 29th Asilomar Conf. On Signals, Systems, and Computers, pp. 600-603, Oct. 1995.
17. Stuart Oberman, “Division Algorithms and Implementations” in IEEE Transactions on Computers Vol. 46, No. 8, pp 853-854, Aug. 1997.

12 Список литературы

1. David Patterson, John Hennessy «Computer Organization and Design: The Hardware/Software Interface», 3rd Edition, 2004, 656 Pages, ISBN: 978-1-55860-604-3, ISBN10: 1-55860-604-1
2. Parhami, Behrooz, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, New York, 2000, ISBN 0-19-512583-5,
3. Israel Koren Computer Arithmetic Algorithms, 2nd Edition, A. K. Peters, Natick, MA, 2002 (ISBN 1-56881-160-8)
4. http://www.ecs.umass.edu/ece/koren/arith/
5. Amos R. Omondi Computer Arithmetic Systems: algorithms, architecture and implementation, Prentice Hall International Series In Computer Science , 1994, 520 pages:, ISBN:0-13-334301-4
6. Michael J. Flynn, Stuart F. Oberman «Advanced Computer Arithmetic Design», Wiley-Interscience, 2001, 344 pages, ISBN-10: 0471412090, ISBN-13: 978-0471412090
7. Naini, A.; Dhablania, A.; James, W.; Das Sarma, D. «1 GHz HAL SPARC64R Dual Floating Point Unit with RAS features», Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-01), 2001. Page(s):173 – 183
8. John Hennessy, David Patterson «Computer Architecture: A Quantitative Approach», 4th Edition, Morgan Kaufmann, 2006, 704 pages, ISBN: 978-0-12-370490-0, ISBN10: 0-12-370490-1
9. IEEE 754: Standard for Binary Floating-Point Arithmetic
10. Prof. W. Kahan «Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic», October 1997, Elect. Eng. & Computer Science, University of California, Berkeley CA 94720-1776
11. Timour Paltashev “Lecture Notes on graduate course EE504: Advanced Computer Organization and Design”, College of Engineering, Northwestern Polytechnic University, Fremont, California, year 2009.

13 Практические примеры реализации арифметических устройств и сопроцессоров в студенческих проектах MIPS_CPU и ARM_CPU

Практические примеры реализации арифметических устройств для современных микропроцессоров можно найти на портале кафедры вычислительной техники СПб ГУ ИТМО http://embedded.ifmo.ru/ в разделе «Проекты».

14 Упражнения и задачи

1) Преобразование чисел из одной системы счисления в другую:

12345678_{10}= _____$_8$ = _____$_2$ =_____$_{16}$

2) Адресация байтов и расположение числа в 32-разрядном слове:

a. MSB – big endian

$1234ABCD_{16}$ → Байт 0 = _____$_2$ Байт 1= _____$_2$ Байт 2= _____$_2$ Байт 3= _____$_2$

b. LSB – little endian

$1234ABCD_{16}$ → Байт 0 = _____$_2$ Байт 1= _____$_2$ Байт 2= _____$_2$ Байт 3= _____$_2$

3) Преобразование чисел из прямого в обратный, дополнительный и смещенный код:

Преобразовать число 12345678_{10} в в прямой двоичный код, обратный и дополнительный двоичные коды

Преобразовать число 112_{10} в смещенный двоичный код со смещением 128

4) Сложение и вычитание двоичных чисел

Сложить в двоичном коде два числа 1234_{10} и 5678_{10}, переведя их вначале в прямой код.

Вычесть в двоичном коде число 1234_{10} из 5678_{10}, переведя вычитаемое в дополнительный код.

5) Простейший сумматор

Постройте 4- и 16-разрядный сумматор, используя базовые логические элементы на Рис.7, структуры на Рис.8 и рис.9а. Посчитайте число логических элементов каждого типа и число межсоединений.

6) Простейшее АЛУ

Модифицируйте схему простейшего АЛУ, чтобы оно могло выполнять операцию сдвига на один разряд вправо или влево. Внесите соотвествующи изменения в Табл.5.

7) Сумматор с предсказанием переноса CLA

Постройте схему 16-разрядного сумматора CLA на основе структуры Рис.16.

Постройте схему блока A и блока B с использованием базовых логических элементов на Рис.7.

8) Сравнение различных сумматоров

Сравните с точки зрения аппаратных затрат все рассмотренные типы сумматоров в 16-разрядном исполнении. Какой из них будет иметь наибольшую задержку установления результата с учетом полного распространения переноса? Если взять задержку распространения в одном базовом логическом элементе за 1 пикосекунду, то какими параметрами с точки зрения задержки получения результата будет обладать каждый из рассмотренных типов сумматоров?

9) Эмуляция суммирования и вычитания 64-разрядных слов на 32-разрядной архитектуре

Разработайте кратчайшую последовательность команд для процессора MIPS или ARM, выполняющих сложение и вычитание 64-разрядных слов. В какой архитектуре требуется больше команд? И почему?

10) Устройство одновременного сложения трех операндов

Постройте схему сложения трех операндов A+B+C с использованием обычных сумматоров и сумматоров с сохранением переноса. Какие преимущества дает использование CSA? Как можно модифицировать схему, чтобы она также могла выполнять вычитание третьего операнда?

11) Матричный сдвигатель

Модифицируйте схему матричного сдвигателя на Рис.14 таким образом, чтобы он мог сдвигать в обе стороны (влево и вправо).

12) Эмуляция умножения 16- и 32-разрядных слов на 32-разрядной архитектуре

Разработайте кратчайшую последовательность команд для процессора MIPS или ARM, выполняющих умножение 16- и 32-разрядных слов. При этом используйте только команды сложения и сдвига в соответствии с алгоритмом на Рис. 15. В какой архитектуре требуется больше команд? И почему? Сравните результаты вашей программы с аппаратной командой умножения.

Что можно сделать, чтобы ускорить умножение за счет многразрядного однотактного сдвига? Модифицируйте алгоритм, чтобы использовать матричный сдвигатель для ускорения эмуляции умножения.

13) Эмуляция 64-разрядного умножения на 32-разрядной архитектуре

Разработайте кратчайшую последовательность команд для процессора MIPS или ARM, выполняющих умножение 64-разрядных слов. Помните, что результат будет 128 разрядов и вы должны предусмотреть соотвествующее число регистров для его размещения. Попробуйте уложиться в максимум 35 команд.

14) Матричный умножитель

Нарисуйте структурную схему матричного умножителя 8x8 разрядов, используя в качестве образца Рис.18. Что нужно добавить в схему для получения результата умножения в обычном двоичном коде?
Сколько модулей и уровней CSA вам понадобится для построения такого умножителя.?

Нарисуйте упрощенную структурную схему умножителя 16x16 с использованием жирных точек для обозначения CSA модулей. Сколько уровней и модулей вам понадобится с использованием кодирования Буфа по основанию 2 и без использования кодирования?

15) Эмуляция деления в 32-разрядной архитектуре

Разработайте кратчайшую последовательность команд для процессора MIPS или ARM, выполняющих деление 16- и 32-разрядных слов. При этом используйте только команды вычитания, сложения и сдвига в соответствии с алгоритмом на Рис. 20. В какой архитектуре требуется больше команд? И почему? Сравните результаты вашей программы с аппаратной командой деления.

Попробуйте использовать варианты операндов из Таблицы 7 и сравнить результат.

16) Сложение с плавающей запятой

Сложите два числа $2.64_{10}*10^3$ и $9.78_{10}*10^5$ учитывая, что у вас в распоряжении только четыре цифры в мантиссе. Покажите вариант с граничной цифрой и цифрой округления и без них.

17) Формат IEEE-754

Переведите число 40_{10} в двоичный формат IEEE-754 с одинарной и двойной точностью.
Переведите число -42.5_{10} в двоичный формат IEEE-754 с одинарной и двойной точностью.
Переведите число 0.4_{10} в двоичный формат IEEE-754 с одинарной и двойной точностью.

Переведите дробь $5/8_{10}$ в двоичный формат IEEE-754 с одинарной и двойной точностью.

18) Сравнение форматов IEEE-754 и IA-32

IEEE-754 определяет число с двойной точностью, которое имеет 53-разрядную мантиссу (включая скрытый бит) и 11-разрядный порядок. IA-32 предлагает формат расширенной точности с 64-разрядной мантиссой и 16-разрядным порядком.

А. Предполагая что это кодирование похоже на используемое в одинарной и двойной точности, какое смещение используется для кодирования порядка в IA-32?

Б. Какой интервал чисел может быть представлен форматом расширенной точности?

В. Насколько выше точность представления чисел в IA-32 по сравнению с форматом двойной точности IEEE-754?

19) Представление чисел с ПЗ

Напишите простую программу на языке C, которая вводит вещественные числа с клавиатуры и показывает их битовое представление формата IEEE-754 в шестнадцатеричном коде на экране. Сделайте варианты для одинарной и двойной точности.

20) Манипуляции с числами ПЗ

Представьте, что число с ПЗ в формате одинарной точности записано в ячейке памяти с адресом X. Напишите последовательность команд MIPS или ARM, которые смогли бы умножить это число на 8 и записать его назад по этому же адресу. Пользоваться командами ПЗ запрещается. Не беспокойтесь о возможном переполнении.

21) Эмуляция умножения с ПЗ

Напишите на ассемблере MIPS/ARM программу эмуляции умножения ПЗ в формате с одинарной точностью. В первом варианте не используйте округления, во втором варианте обеспечьте все стандартные режимы округления для IEEE-754.

22) Эмуляция деления с ПЗ

Напишите на ассемблере MIPS/ARM программу эмуляции деления ПЗ в формате с одинарной точностью. В первом варианте не используйте округления, во втором варианте обеспечьте все стандартные режимы округления для IEEE-754.

23) Аппаратура округления

Постройте блок округления результата и управляющий автомат, обеспечивающий округление результата по требованиям IEEE-754 в Таблице 14.

24) Эмуляция деления по Голдшмидту

Напишите программу на языке С для реализации деления по методу Голдшмидта, аналогично реализованному в АУ SPARC64.

25) Эмуляция извлечения квадратного корня

Напишите программу на языке С для извлечения квадратного корня, аналогично реализованному в АУ SPARC64.

15 Контрольные вопросы

1. Принцип сложения и вычитания двоичных чисел.
2. Принцип выполнения операций сдвига двоичных чисел.
3. Как работают двоичные умножители?
4. Как работают двоичные делители?
5. Принцип работы алгоритма сложения в арифметическом устройстве ПЗ.
6. Принцип работы алгоритма умножения в арифметическом устройстве ПЗ.
7. Каковы причины использования в современных компьютерных архитектурах записи в дополнительном коде для представления чисел со знаком?
8. Как называется форма представления чисел, когда в каждой позиции используется не только цифра со значением, но также и знак?
9. Какое преобразование вычитаемого позволяет полностью исключить необходимость в отдельной операции вычитания в АЛУ микропроцессоров?
10. Какая польза от использования сумматоров с сохранением переноса?
11. Над словами какой разрядности выполняются операции в процессорах MIPS?
12. Команды целочисленной арифметики в процессорах MIPS?
13. Команды целочисленной арифметики в процессорах ARM?
14. Команды обработки вещественных чисел в MIPS?
15. Команды обработки вещественных чисел в ARM?
16. В каких системах процессоры семейства ARM пользуются популярностью?
17. Какова особенность принципа работы алгоритма деления SRT?
18. Какую структуру имеет устройство сложения/вычитания FADD в арифметическом блоке ПЗ и сколько команд исполняет за каждый такт?
19. Какие команды выполняет FMUL в блоке ПЗ?
20. Какой алгоритм используют команды деления/извлечения кв.корня?