ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ DSP

DSP приложения обычно программируются на тех же языках, что и другие научные и инженерные задачи: C, BASIC и ассемблер. Мощность и универсальность C делает этот язык программирования основным для специалистов по вычислительной технике и других профессиональных программистов. С другой стороны простота BASIC делает его идеальным для применения учеными и инженерами, которые только что пришли в мир программирования. В независимости от того, какой язык программирования вы используете, большая часть вопросов программного обеспечения DSP лежит в области круговерти нулей и единиц. Это включает в себя такие проблемы: как представляются числа битовой комбинацией, ошибка округления в компьютерной арифметике, вычислительная скорость различных типов процессоров и др. Эта глава посвящена тем вещам, которые вы можете делать на *высоком уровне*, безз необходимости знать, что происходит во время внутренней работы вашего компьютера на *низком уровне*.

Компьютерные числа

Цифровые компьютеры хранят и вызывают числа, к сожалению, эти процессы не обходится без ошибок. Например, вы пробуете запомнить на компьютере число 1.41421356. Лучшее, что может сделать компьютер, это сохранить ближайшее к нему число, которое он *может* представить: 1.41421354. Во многих случаях эта ошибка незначительна, в других она может оказаться катастрофической. Другим примером является классическая ошибка вычисления при сложении двух чисел с несоразмерно различными значениями, например, 1 и 0.00000001. Мы бы хотели получить ответ: 1.00000001, но компьютер ответит 1. Понимание того, как компьютер хранит и обращается с числами поможет предвидеть и устранить эти проблемы *перед* тем, как программа выдаст бессмысленные данные.

Эти проблемы возникают из-за того, что число представляется фиксированным числом бит, обычно 8, 16, 32 или 64. Например, представьте, что для хранения числа используется 8-битная переменная. Имеется 2^8 = 256 возможных битовых комбинаций, поэтому переменная может содержать только 256 различных значений. Это фундаментальное ограничение, и мы ничего не можем сделать с этим. Частично мы можем контролировать, какие значения соответствуют различным битовым комбинациям. В простейшем случае битовые комбинации могут представлять числа от 0 до 255, от 1 до 256, от –127 до +128 и т.п. Более редкой ситуации, 256 битовых комбинаций могут представлять 256 экспоненциально связанных чисел: 1, 10, 100, 1000, …, 10^{254}, 10^{255}. Каждый, кто обращается к данным, должен понимать, что представляют битовые маски. Обычно это обеспечивается алгоритмом или формулой преобразования между представляемым значением и битовой комбинацией, и обратно.

В то время как существует множество схем кодирования, основными являются только два: *фиксированная запятая* (также называемая целыми числами) и *плавающая запятая* (также называемая вещественными числами). В этой книге в программах на языке BASIC для обозначения целых чисел используется символ % в конце имени переменной, например: I%, N%, SUM%. Все другие переменные являются хранят вещественные числа, например: X, Y, MEAN и др. При оценке форматов, представленных ниже, попытайтесь понять их в терминах их *диапазона* (максимального и минимального значений, которые они могут представлять) и *точности* (интервала между числами).

Фиксированная запятая (целые числа)

Представление с фиксированной запятой используется для хранения положительных и отрицательных *целых чисел*: -3, -2, -1, 0, 1, 2, 3. Языки высокого уровня, такие как C и BASIC, обычно выделяют 16 бит для хранения каждого целого числа. В простейшем случае, 2^{16}=65536 возможных битовых комбинаций присваиваются числам от 0 до 65535. Этот формат называется *целое число без знака* (unsigned integer). , и его пример приведен на Рис.4-1 (использовались только четыре бита). Преобразование между битовой комбинацией и числом заключается в изменении основания 2 на основание 10. Недостатком целых чисел без знака является то, что не могут быть представлены отрицательные числа.

Смещенный двоичный код (offset binary) похож на представление целых чисел без знака, за исключением того, что десятичные значения *сдвинуты*, чтобы разрешить представление отрицательных чисел. На примере Рис.4-1, десятичные значения сдвинуты на *семь*, что позволяет представить числа от –7 до 8. Похожим способом 16-битное представление будет использовать для смещения значение 32767, что позволит задать числа от –32767 до 32768. Смещенный двоичный код не является стандартным форматом, в основном он используется в АЦП и ЦАП. Например, диапазон входного напряжения от –5В до +5В может быть отображен при преобразовании числами от 0 до 4095.

Представление *знаком и значением* (sign and magnitude) является другим простым способом задания отрицательных чисел. Крайний левый разряд называется *знаковым разрядом* (sign bit) и устанавливается в *ноль* для положительных чисел и в *единицу* для отрицательных. Другие разряды представляют собой стандартные двоичные представления абсолютного значения числа. В результате такого представления теряется одна битовая комбинация, поскольку для задания нуля используются две комбинации: 0000 и 1000, поэтому результирующий диапазон этого алгоритма от –32767 до +32767.

Эти три метода представления концептуально просты, но сложно реализуются на аппаратном уровне. Когда в компьютерной программе записывается A=B+C, необходимо представлять себе, как представить битовую комбинацию B и сложить ее с битовой комбинацией C, чтобы получить битовую комбинацию A.

ЦЕЛОЕ БЕЗ ЗНАКА		СМЕЩЕННЫЙ КОД		ЗНАК И ЗНАЧЕНИЕ		ДОПОЛНИТЕЛЬНЫЙ ДВОИЧНЫЙ КОД	
Число	Комбинация	Число	Комбинация	Число	Комбинация	Число	Комбинация
15	1111	8	1111	7	0111	7	0111
14	1110	7	1110	6	0110	6	0110
13	1101	6	1101	5	0101	5	0101
12	1100	5	1100	4	0100	4	0100
11	1011	4	1011	3	0011	3	0011
10	1010	3	1010	2	0010	2	0010
9	1001	2	1001	1	0001	1	0001
8	1000	1	1000	0	0000	0	0000
7	0111	0	0111	0	1000	-1	1111
6	0110	-1	0110	-1	1001	-2	1110
5	0101	-2	0101	-2	1010	-3	1101
4	0100	-3	0100	-3	1011	-4	1100
3	0011	-4	0011	-4	1100	-5	1011
2	0010	-5	0010	-5	1101	-6	1010
1	0001	-6	0001	-6	1110	-7	1001
0	0000	-7	0000	-7	1111	-8	1000
от 0 до 65536		**от –32767 до 32768**		**от –32767 до 32767**		**от –32768 до 32767**	

Рис.4-1. Основные форматы представления с фиксированной запятой.

Дополнительный двоичный код (two's complement) является форматом, который используют инженеры по аппаратной части, и обычно используется для представления чисел в компьютерах. Для того, чтобы понять систему кодирования, посмотрите на Рис.4-1 представление нуля, оно *соответствует* двоичному нулю – 0000. При счете наверх, десятичные числа являются просто двоичным эквивалентом (0 = 0000, 1=0001, 2=0002 и т.д.). Теперь вспомните, что эти четыре бита хранятся в регистре, состоящим из четырех триггеров. Если мы снова начнем с 0000 и проведем операцию вычитания, цифровая аппаратная часть автоматически будет считать в дополнительном двоичном коде: 0=0000, -1=1111, -2=1110 и т.д. Аналогично работает одометр в новых автомобилях. При движении вперед он изменяет свое значение как: 00000, 00001, 00002 и далее. При движении назад, одометр отсчитывает: 00000, 99999, 99998 и т.д.

При использовании 16 бит, двоичный дополнительный код может представлять числа от –32768 до 32767. Крайний левый разряд содержит «0», если число положительное или равно нулю, и «1», если число отрицательное. Следовательно, крайний левый разряд называется *знаковым разрядом*, также как и в представлении знаком и значением. Преобразование между десятичным числом и числом в дополнительном двоичном коде является прямым для положительных чисел, т.е. из преобразованием основания 10 в основание 2. Для отрицательных чисел часто используется следующий алгоритм: (1) берется абсолютное значение десятичного числа, (2) преобразовывается в двоичное, (3) дополняются все разряды (единицы становятся нулями, а нули единицами), (4) к двоичному числу добавляют 1. Например: -5 > 5 > 0101 > 1010 > 1011. Дополнительный двоичный код нелегок для понимания людьми, но прост для цифровой электроники.

Плавающая запятая (вещественные числа)

Система кодирования для чисел с плавающей запятой более сложная, чем для чисел с фиксированной запятой. Основная идея этой системы похожа на ту, что используется в экспоненциальной системе обозначения, где *мантисса* умножается на десять, возведенное в некоторую *экспоненту*. Например, 5.4321×10^6, где 5.4321 является *мантиссой* и 6 является *экспонентой*. Экспоненциальная нотация способна представлять как очень большие, так и очень маленькие числа. Например, число атомов на земле 1.2×10^{50}, а 2.6×10^{-23} расстояние, которое проходит черепаха в одну секунду в сравнении с диаметром нашей галактики. Запомните, что числа, представленные в экспоненциальной нотации, *нормализованы*, т.е. слева от запятой используется только одна значащая цифра. Это достигается приведением экспоненты.

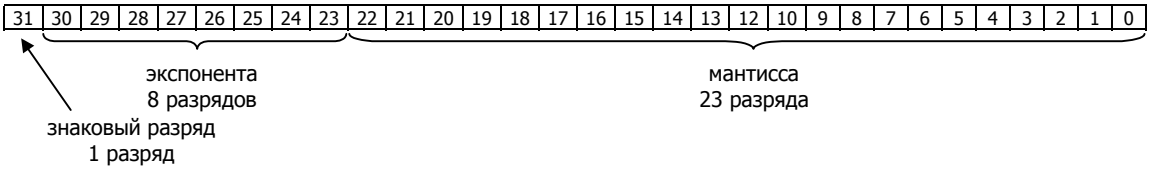
Представление чисел с плавающей запятой похоже на экспоненциальное представление, за исключением того, что оно проводится по основанию два, а не десять. В то время как существует множество похожих форматов, самым распространенным является стандарт ANSI/IEEE 754-1985. Этот стандарт определяет 32-битные числа, называемые числами с *одинарной точностью*, и 64-битные, называемые числами с *двойной точностью*. Как показано на Рис.4-2, 32 бита, используемые при одинарной точности делятся на три группы: биты от 0 до 22 образуют мантиссу, биты от 23 до 30 образуют экспоненту и 31 бит является знаковым разрядом. Эти биты формируют число с плавающей запятой, *v*, с использованием следующего выражения:

$$v = (-1)^S \ x \ M \ x \ 2^{E-127}$$

Уравнение 4-1. Преобразование битовой комбинации в число с плавающей запятой. S – значение знакового разряда, M – значение мантиссы, E – значение экспоненты.

Термин $(-1)^S$ означает знаковый разряд, S, который равен «0», если число положительно, и «1», если число отрицательно. Переменная, E, является числом в диапазоне от 0 до 255, представляющим восемь бит экспоненты. Вычитание 127 из этого числа позволяет получить значение экспоненты от 2^{-127} до 2^{128}. Другими словами значение экспоненты хранится в двоичном смещенном коде со смещением 127.

Мантисса, M, формируется из 23 бит в виде двоичной дроби. Например, десятичное число 2.783 интерпретируется как: 2 + 7/10 + 8/100 + 3.100. Двоичная дробь 1.0101 подразумевает: 1 + 0/2 + 1/4 + 0/8 + 1/16. Числа с плавающей запятой *нормализуются* также как и числа в экспоненциальном представлении, поэтому имеется только одна значащая цифра слева от запятой (называемая *двоичной запятой*). Поскольку единственным значащим числом по основанию два является «1», первой цифрой в мантиссе всегда будет 1, и поэтому нет смысла ее хранить. Устранение этой избыточности позволяет сохранить еще один бит точности. 23 бита образуют мантиссу как:



Пример 1:

0	00000111	11000000000000000000	
+	7	0.75	+1.75 x $2^{(7-127)}$ = +1.316554 x 10^{-36}

Пример 2:

1	10000001	01100000000000000000	
-	129	0.375	-1.375 x $2^{(129-127)}$ = -5.500000

Рис.4-2. Формат хранения числа с плавающей запятой одинарной точности.

$$M = 1.m_{22}m_{21}m_{20}m_{19}...m_2m_1m_0$$

Уравнение 4-2. Алгоритм преобразования битовой маски в мантиссу.

Другими словами, M = 1 + $m_{22}2^{-1}$ + $m_{21}2^{-2}$ + $m_{20}2^{-3}$… Если все разряды битовой комбинации равны нулю, значение мантиссы составляет 1. В обратном случае, если все разряды битовой маски равны единице, значение мантиссы будет чуть меньше двух: 2 – 2^{-23}. Используя эту систему кодирования, наибольшее представляемое число будет составлять ±(2-2^{-23}) x 2^{128} = ±6.8 x 10^{38}. Наименьшее же число будет равно: ±1.0 x 2^{-127} = ±5.9 x 10^{-39}. Стандарт IEEE немного уменьшает этот диапазон, освобождая некоторые битовые комбинации для специальных значений. Поэтому наибольшее и наименьшее значения по этому стандарту составляют: ±3.4 x 10^{38} и ±1.2 x 10^{-38}, соответственно. Освобожденные битовые комбинации позволяют предоставить три специальных класса чисел: (1) ±0, определяет нулевые мантиссу и экспоненту, (2) ±∞, определяет нулевое значение мантиссы при всех битах экспоненты, равных «1». (3) Группа очень маленьких *ненормализованных* значений между ±1.2 x 10^{-38} и ±1.4 x 10^{-45}. Эти числа получаются при устранении условия, что значащая цифра в мантиссе равна единице. Кроме этих трех специальных классов, имеются битовые комбинации, которым не присваиваются значения; они часто называются NAN (Not A Number – не число).

Стандарт IEEE для двойной точности просто добавляет дополнительные разряды для хранения мантиссы и экспоненты. Из 64 бит, формирующих число с плавающей точкой, биты от 0 до 51 отводятся под хранение мантиссы, биты от 52 до 62 для экспоненты и 63 бит является знаковым разрядом. Как и в предыдущем случае, значение мантиссы расположено в диапазоне от единицы и чуть меньше двух, т.е. M = 1 + $m_{51}2^{-1}$ + $m_{50}2^{-2}$ + $m_{49}2^{-3}$… 11 бит экспоненты формируют число от 0 до 2047 со смещением 1023, позволяя получать экспоненты от 2^{-1023} до 2^{1024}.

Наибольшее и наименьшее значения составляют: $\pm 1.8 \times 10^{308}$ и $\pm 2.2 \times 10^{-308}$, соответственно. Это очень большие и маленькие числа, которые часто не используются в приложениях, поэтому трудно найти задачи, которым не подходят числа с одинарной точностью. С другой стороны, сложно представить приложения, в которых числа с двойной точностью могли быть ограничить ваши потребности.

Точность числа

Ошибки, связанные с представлением числа, похожи на ошибки квантования во время аналого-цифрового преобразования. *Хотелось* бы хранить непрерывный диапазон значений, однако *можно* представить только конечный набор квантованных уровней. Каждый раз, когда генерируется число, например, после математических операций, оно должно быть округлено до ближайшего значения, которое может хранить используемый формат.

Например, представьте, что вы отвели для хранения числа 32 бита. Поскольку число возможных битовых комбинаций составляет ровно 2^{32} = 4,294,967,296 битовых комбинаций, можно представить 4,294,967,296 различных чисел. Некоторые языки программирования позволяют объявить переменную, называемую *длинное целое* (long integer), которая использует для представления числа с фиксированной точкой 32 бита в виде дополнительного двоичного кода. Это означает, что имеется 4,294,967,296 возможных битовых комбинаций, задающих целые числа от –2,147,483,648 до 2,147,483,647. С другой стороны, числа с плавающей точкой одинарной точности используют эти 4,294,967,296 битовых комбинаций для гораздо большего диапазона: от -3.4×10^{-38} до 3.4×10^{38}.

При переменных с фиксированной запятой, интервал между соседними числами всегда составляет *ровно единицу*. В представлении с плавающей запятой, интервал между соседними числами изменяется в зависимости представляемого диапазона значений. Если мы случайно возьмем число с плавающей запятой, интервал до следующего числа будет составлять приблизительно *одну десятимиллионную от его значения* (а именно, в 2^{-24} до 2^{-23} меньше его значения). Это является ключевой концепцией представления с плавающей точкой: чем больше числа, тем больше интервал между ними. Рис.4-3 иллюстрирует эту концепцию, показывая последовательные числа с плавающей запятой и интервалы между ними.

0.00001233862713	интервал = 0.00000000000091
0.00001233862804	(1 часть из 13 миллионов)
0.00001233862895	
0.00001233862986	
1.000000000	интервал = 0.00000119
1.000000119	(1 часть из 8 миллионов)
1.000000238	
1.000000358	
1.996093750	интервал = 0.00000119
1.996093869	(1 часть из 17 миллионов)
1.996093988	
1.996094108	
636.0312500	интервал = 0.0000610
636.0313110	(1 часть из 10 миллионов)
636.0313720	
636.0314331	
217063424.0	интервал = 16.0
217063440.0	(1 часть из 14 миллионов)
217063456.0	
217063472.0	

Рис.4-3. Примеры интервалов между числами с плавающей запятой одинарной точности.

Программа в Таблице 4-1 показывает как *ошибка округления* (ошибка квантования при математических расчетах) может вызвать проблемы в DSP. Внутри программного цикла, два случайных числа складываются с числом с плавающей запятой переменной X, и далее вычитаются из него. В идеале, это не должно вызвать *никаких* изменений. В реальности, ошибка округления каждой арифметической операции вызывает постепенный сдвиг X от его начального значения. Этот сдвиг может иметь два типа, в зависимости от того, как ошибки складываются друг с другом. Если ошибки округления случайно положительные или отрицательные, значение переменной будет случайно увеличиваться или уменьшаться. Если ошибки преимущественно одного знака, значение переменной будет сдвигаться гораздо быстрее и равномерней.

```
100 X = 1                  'инициализировать X
110 '
120 FOR I% = 0 TO 2000
130   A = RND              'загрузить случайные числа в A и B
140   B = RND
150 '
160   X = X + A            'добавить A и B к X
170   X = X + B
180   X = X - A            'отменить добавление
190   X = X - B
200 '
210 PRINT X                'в идеале должно быть 1
220 NEXT I%
230 END
```

Таблица 4-1.

Рис.4-4 показывает как в приведенном примере программы переменная X сдвигается в своем значении. Очевидно, что *аддитивная погрешность* гораздо больше, чем *случайная ошибка*. Это происходит из-за того, что случайные ошибки стремятся устранить друг друга, в то время как аддитивные ошибки просто складываются. Аддитивная ошибка приблизительно равна ошибке округления от одной операции, умноженной на общее число операций. В сравнении с ней, случайная ошибка увеличивается пропорционально квадратному корню от числа операций. Как видно из рисунка, для обычных алгоритмов ЦОС аддитивная ошибка может быть в сотни раз больше, чем случайная.

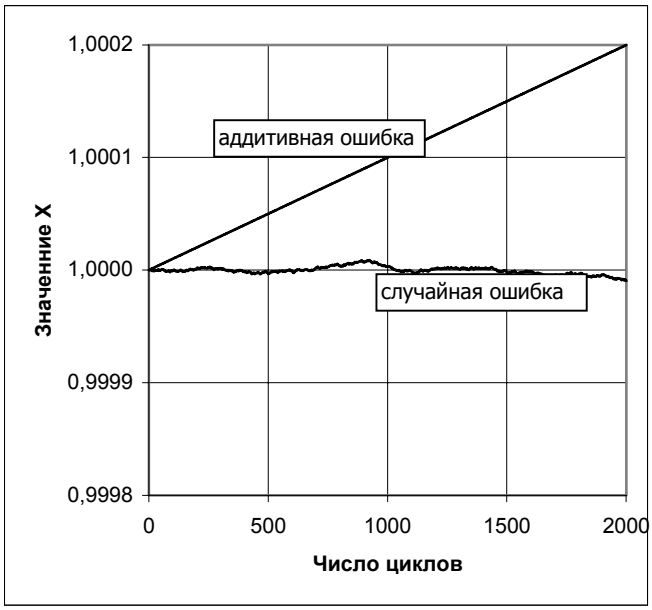


Рис.4-4. Аккумуляция ошибки округления в переменных с плавающей запятой.

К сожалению, почти невозможно управлять или предсказать, какой тип ошибки будет встречен в определенном алгоритме. Например, программа в Таблице 4-1, генерирует аддитивную ошибку. Она может быть изменена для генерации случайной ошибки, если сделать небольшие изменения в числах, которые складываются и вычитаются. Например, кривая случайной ошибки была получена при определении: A=EXP(RND) и B=EXP(RND). Вместо случайно распределенных числе между 0 и 1, значения становятся экспоненциально распределены между 1 и 2.718. Такое небольшое изменение достаточно, чтобы переключить программу в режим генерации случайной ошибки.

Поскольку невозможно предсказать, по какому способу будут аккумулироваться ошибки округления, предполагайте наихудший вариант сценария. Ожидайте, что каждое число одинарной точности будет иметь ошибку округления около *одной сорокамиллионной, умноженной на число операций*. Это основано на допущении аддитивной ошибки и средней ошибки от одной операции, равной одной четверти от уровня квантования. Если проанализировать подобным образом числа с двойной точностью, каждое число будет содержать ошибку около *одной сорокаквадриллионной, умноженной на число операций*.

```
100 'вещественный счетчик цикла
110 FOR X = 0 TO 10 STEP 0.01
120   PRINT X
130 NEXT X
```

Результат работы программы:

```
0.00
0.01
0.02
0.03
...
9.960132
9.970133
9.980133
9.990133
```

```
100 'целый счетчик цикла
110 FOR I% = 0 TO 1000
120   X = I%/100
130   PRINT X
140 NEXT I%
```

Результат работы программы:

```
0.00
0.01
0.02
0.03
...
9.96
9.97
9.98
9.99
10.00
```

Таблица 4-2. Сравнение циклов с вещественным и целым счетчиком цикла.

Таблица 4-2 иллюстрирует особенно досадную проблему ошибки округления. Каждая из приведенных программа выполняет одну задачу: выводит 1001 чисел, равномерно увеличивающихся от 0 до 10. Программа слева использует в качестве счетчиком цикла вещественную переменную, X. При выполнении программы компьютер устанавливает счетчик в начальное значение (0 в этом примере). При завершении каждого цикла значение счетчика увеличивается на длину шага (в нашем случае 0.01). Далее проходит сравнение: нужно продолжать цикл или его завершить? Цикл прекращается, когда компьютер определяет, что значение счетчика *больше чем* значение завершения (в данном примере 10.0). Как показано в результате работы программы ошибка округления вызывает накопление в переменной X аддитивной ошибки во время выполнения цикла. В итоге, накопленная ошибка *предотвращает* выполнение последнего цикла, поскольку перед его выполнением переменная X вместо значения 10.0 будет содержать значение 10.000133. Поскольку X больше значения завершения, компьютер считает, что операция выполнена, и цикл преждевременно заканчивается. Потеря этого последнего значения является распространенной ошибкой во многих программах.

В сравнении с рассмотренной программой, программа справа использует в качестве счетчика цикла целую переменную, I%. Сложение, вычитание или умножение двух целых чисел всегда дает другое целое число. Это говорит о т ом, что представление с фиксированной запятой не имеет ошибки округления при этих операциях. Целые числа идеально подходят для управления циклами, так как гарантируется, что последний цикл всегда выполнится! Если у вас нет особых причин, всегда используйте целые числа в качестве индексов и счетчиков.

Если вы должны использовать в качестве счетчика цикла вещественную переменную, старайтесь использовать дроби, которые являются степенью *двух* (например: 1/2, 1/4, 3/8, 27/16), а не *десяти* (0.1, 0.6, 1.4, 2.3). Например, лучше использовать FOR X=1 TO 10 STEP 0.125, чем FOR X=1 TO 10 STEP 0.1. Это позволяет индексу иметь *точное* бинарное представление. Например, десятичное число 1.125 имеет точное бинарное представление: $1.001000000000000000000000\times2^0$. В сравнении с ним, десятичное число 1.1 попадает *между* двумя числами: 1.0999999046 и 1.1000000238 (в бинарном виде они имеют представление: $1.00011001100110011001100\times2^0$ и $1.00011001100110011001101\times2^0$). Это вызывает ошибки в результатах каждый раз, когда в программе встречается число 1.1.

Следует заметить важную вещь, что числа с плавающей точкой одинарной точности имеют *точные* бинарные представления для всех целых чисел между ±16.8 миллиона (или $\pm2^{24}$). Выше этих значений, интервалы между числами становятся больше единицы, вызывая потерю некоторых целых чисел. Следуя вышесказанному, целые числа в представлении с плавающей точкой, входящие в диапазон ±16.8 миллиона, могут складываться, вычитаться и умножаться без проявления ошибок округления.

Скорость выполнения: языки программирования

DSP программирование может быть разделено приблизительно на три уровня совершенствования: *Ассемблирование*, *Компиляция* и *Определяемое Приложением*. Чтобы понять разницу между ними, необходимо начать объяснение с основ цифровой электроники. Все микропроцессоры основаны на наборе внутренних регистров, которые являются группой триггеров, хранящих последовательности единиц и нулей. Например, микропроцессор 8088, ядро первых IBM PC, содержит *четыре* регистра общего назначения, состоящих из 16 разрядов. Они идентифицируются именами: AX, BX, CX и DX. Также имеется *девять* дополнительных регистров специального назначения, называемые: SI, DI, SP, BP, CS, DS, SS, ES и IP. Например, IP, указатель инструкции, содержит информацию о расположении в памяти следующей инструкции.

Представьте, что вы написали программу сложения двух чисел: 1234 и 4321. При запуске программы IP содержит адрес раздела памяти, который заключает в себя комбинацию нулей и единиц, как показано в Таблице 4-3. Хотя она может и выглядит бессмысленно, эта комбинация нулей и единиц содержит все команды и данные, необходимые для выполнения задачи. Например, когда микропроцессор встречает битовую комбинацию: 00000011 11000011, он интерпретирует ее как команду взять 16 бит, хранящихся в регистре BX, бинарно сложить их с 16 битами, хранящимися в регистре AX и запомнить результат в регистре AX. Этот уровень программирования называется *машинный код*, и находится чуть выше прямой работы с электронными схемами.

```
10111001 00000000
11010010 10100001
00000100 00000000
10001001 00000000
00001110 10001011
00000000 00011110
00000000 00000010
10111001 00000000
11100001 00000011
00010000 11000011
10001001 10100011
00001110 00000100
00000010 00000000
```

Таблица 4-3. Программа в машинном коде для сложения чисел 1234 и 4321.

Поскольку работа в двоичном коде в конечном счете сведет с ума даже самых опытных инженеров, этим комбинациям единиц и нулей присваиваются имена, в соответствии с функциями, которые они выполняют. Этот уровень программирования называется *ассемблирование*, и его пример показан в Таблице 4-4. Хотя программы на ассемблере понять гораздо проще, он в принципе похож на программирование в машинном коде, поскольку существует прямая связь между командами программы и операциями, которые выполняются в микропроцессоре. Например: ADD AX, BX транслируется в 00000011 11000011/ Программа, которая используется для преобразования ассемблерного кода в Таблице 4-4 (называемого *исходный код*) в комбинации единиц и нулей, показанных в Таблице 4-3 (называемые *объектным* или *исполняемым* кодом), называется *ассемблер*. Этот исполняемый код может быть непосредственно запущен на микропроцессоре. Очевидно, что программирование на ассемблере требует исчерпывающего знания внутренней структуры отдельных микропроцессоров, которые будут использоваться.

```
MOV CX,1234              ;запомнить 1234 в регистре CX и перенести его
MOV DS:[0],CX            ;в ячейку памяти DS:[0]

MOV CX,4321              ;запомнить 4321 в регистре CX и перенести его
MOV DS:[2],CX            ;в ячейку памяти DS:[2]

MOV AX,DS:[0]            ;перенести переменные, сохраненные в ячейках памяти
MOV BX,DS:[2]            ;DS:[0] и DS:[2] в регистры AX & BX

ADD AX,BX                ;сложить AX и BX, запомнить сумму в AX

MOV DS:[4],AX            ;перенести значение суммы в ячейку памяти DS:[4]
```

Таблица 4-4. Программа на языке ассемблера для сложения чисел 1234 и 4321.

Программирование на ассемблере связано с непосредственным обращением к цифровой электронике: регистрам, ячейкам памяти, битам состояния и т.п. Следующий уровень программирования манипулирует с *абстрактными* переменными без обращения к аппаратной части. Он называется *языки высокого уровня* или *транслируемые языки*. Существует множество языков высокого уровня, например, C, BASIC, FORTRAN, PASCAL, APL, COBOL, LISP и др. Таблица 4-5 показывает программу на языке BASIC для сложения чисел 1234 и 4321. При этом программист должен знать только о переменных A, B и C, и ничего об аппаратной части.

```
100 A = 1234
110 B = 4321
120 C = A+B
130 END
```

Таблица 4-5. Программа на языке BASIC для сложения чисел 1234 и 4321.

Программа, называемая *компилятором*, используется для преобразования исходного кода высокого уровня непосредственно в машинный код. Это требует от компилятора *назначить* каждой встречаемой абстрактной переменной адрес ячейки памяти. Например, первый раз встретив переменную A в Таблице 4-5 (линия 100), он принимает решение, что программист использует этот символ для хранения переменной числа вещественного типа. Соответственно, компилятор отводит четыре байта памяти, которые будут использоваться для хранения значения этой переменной. После этого, каждый раз, когда в программе будет встречаться переменная A, компьютер по необходимости обновляет ее значение. Компилятор также разбивает сложные математические выражения на более простые, например: $Y=LOG(X^{COS(Z)})$. Микропроцессоры знают только операции сложения, вычитания, умножения и деления. Все более сложное должно быть выполнено как комбинация этих четырех операций.

Языки высокого уровня отделяют программиста от аппаратной части, что делает программирование более легким, и позволяет переносить исходный код между различными типами микропроцессоров. Самое важное, что программист, который использует транслируемые языки, не должен знать ничего о внутренней работе компьютера. За это несет ответственность тот программист, который пишет компилятор.

Большинство компиляторов преобразуют *всю* программу в машинный код *перед* ее выполнением. Исключением являются тип компилятора, который называют *интерпретатором*, и самым распространенным его примером является интерпретатор BASIC. Интерпретатор преобразует в машинный код *одну строку* исходного кода, выполняет этот машинный код и переходит к следующей строке исходного кода. Таким способом достигается интерактивный режим для простейших программ, хотя скорость исполнения при этом значительно снижается (приблизительно в сто раз).

Уровень программирования высокого уровня встречается в *пакетах прикладных программ* для DSP. Часто они обеспечивают поддержку специфичного оборудования. Предположим, что вы купили новый DSP микропроцессор для его применения в вашем проекте. Такие устройства часто обладают множеством встроенных возможностей для ЦОС: аналоговые входы, аналоговые выходы, цифровой ввод/вывод, фильтры устранения эффектов наложения спектров, восстанавливающие фильтры и т.д. Встает вопрос: как программировать? В наихудшем случае, производитель даст вам *ассемблер*, предполагая, что вы самостоятельно изучите внутреннюю архитектуру устройства. Более распространенным сценарием является предоставление *С компилятора*, позволяющего осуществлять программирования без необходимости задумываться, как работает микропроцессор.

В наилучшем варианте, производитель предоставляет пакет вспомогательных программ: библиотеки алгоритмов, предварительно написанные подпрограммы для работы с вводом/выводом, средства отладки и т.п. Вы можете просто соединить пиктограммы для формирования желаемой системы. *Вещами*, которыми вы манипулируете, являются пути сигнала, алгоритмы обработки сигналов, параметры аналогового ввода/вывода и др. Когда вы удовлетворены созданной системой, она преобразуется в машинный код для исполнения на аппаратуре. Другие пакетов прикладных программ используются для обработки изображений, спектрального анализа, измерения и управления, создания цифровых фильтров и т.п.

Различие между этими тремя уровня может быть неочевидным. Например, большинство транслируемых языков позволяют напрямую управлять аппаратной частью. Более того, язык высокого уровня с хорошей библиотекой для DSP не сильно отличается от пакета прикладных программ. Для осмысления этих трех категорий главное понять, с чем вы работаете: (1) с аппаратной частью, (2) с абстрактными переменными или (3) с целыми процедурами и алгоритмами.

Есть также важная концепция внутри этой классификации. При использовании языка высокого уровня, для работы с аппаратной частью вы полагаетесь на программиста, который написал компилятор. Проще говоря, если вы используете пакет прикладных программ, вы надеетесь на программиста, который использовал *лучшие* алгоритмы для работы с DSP. Появляется камень преткновения: эти программисты никогда не сталкивались с конкретными задачами, которые вы решаете. Поэтому они не могут предоставить вам оптимальное решение. При работе на *высоком* уровне следует ожидать, что конечный машинный код будет *менее* эффективным в понятиях использования памяти, производительности и точности.

Какой язык программирования следует использовать? Это зависит от того, *кем* вы являетесь, и *что* вы планируете сделать. Большинство инженеров используют С (или С++). Мощность, гибкость и модульность, все это есть в С. Язык С настолько популярен, что встает вопрос: Почему нужно писать DSP приложения на чем то другом? На это можно дать три ответа. Во-первых, ЦОС развивается настолько быстро, что еще остаются некоторые организации и инженеры, которые

используют другие языки программирования, например, FORTRAN или PASCAL. В основном это военные и правительственные организации. Во-вторых, существуют приложения, которые не требуют предельной производительности, достигаемой при программировании на языке ассемблера. Они попадают в категорию «немного меньше скорости, гораздо больше работы». В-третьих, С не является легким языком, особенно для людей, которые не специализируются в программировании. Это относится к тем ученым и инженерам, которые используют методы ЦОС для реализации своих проектов. Чаще всего они используют BASIC, вследствие его простоты.

Почему BASIC был выбран для примеров этой книги? Эта книга посвящена *алгоритмам*, а не стилю программирования. Вы должны уделить внимание методам ЦОС, а не отвлекаться на особенности отдельных языков программирования. Например, все примеры программ в этой книге включают в себя *номер строки*. Это позволяет более доступно объяснить работу программы: «строка 100 делает то-то и то-то, строка 110 делает это и т.д.». Конечно, вы не будете использовать номера строк в вашей программе. Смысл состоит в том, что *изучение*, имеет другие требования, чем использование. Существует множество других книг, которые описывают оптимальный исходный код для алгоритмов ЦОС.

Бессмысленно сравнивать скорость работы аппаратной и программной части, в не зависимости от того, какой результат будет получен. Программисты, которые предпочитают языки высокого уровня (традиционно, ученые, реализующие некоторые методы) аргументируют свой выбор тем, что ассемблер всего на 50% быстрее оттранслированной программы и при этом в пять раз сложнее. Те, кто любит использовать ассемблер (обычно системные инженеры), говорят обратное: ассемблер в пять раз быстрее и только на 50% сложнее. Как и в большинстве дискуссий, обе стороны приводят выборочные доводы для поддержки своих взглядов. Практика показывает, что подпрограммы, написанная на ассемблере будет в 1.5 – 3 раза быстрее, чем написанная на языке высокого уровня. Единственным способом узнать действительный выигрыш в производительности является написание обеих версий кода и проведение сравнительных тестов. Поскольку персональные компьютеры увеличивают свою производительность приблизительно на 40% в год, написание подпрограммы на ассемблере эквивалентно двухгодичному преимуществу в аппаратной технологии.

Большинство профессиональных программистов чаще всего против использования ассемблера и категорически против языка BASIC. Они аргументируют это тем, что ассемблер и BASIC препятствуют созданию хорошего программного обеспечения. Хороший код должен быть *переносимым* (способным переносится с одного компьютера на другой), модульным (имеющим хорошо организованную структуру подпрограмм) и легким для понимания (достаточно комментированным и имеющим описательные имена переменных). Возможности ассемблера и BASIC не позволяют им достичь этих стандартов. Это сочетается с тем, что чаще всего люди, которые используют ассемблер и BASIC, имеют лишь формальные представления об организации надлежащей структуры программы и подготовке документации.

Любители ассемблера резко отвечают на это следующим доводом. Предположим, что вы написали программу на ассемблере, а ваш конкурент на С. Конечный пользователь в первую очередь обратит внимание на то, что ваша программа в два раза быстрее!

Никто не станет советовать вам писать большие программы целиком на ассемблере, однако, целесообразно использовать его в тех частях программы, которые требуют быстрого выполнения. Например, большинство функций в программном обеспечении DSP написаны на ассемблере, и вызываются из основных программ, написанных на С.

Скорость выполнения: аппаратная часть

Вычислительная мощность увеличивается настолько быстро, что любая *книга*, по этой теме устаревает быстрее, чем она издается! Первый IBM PC был выпущен в 1981 году, и был основан на микропроцессоре 8088 с тактовой частотой 4.77 МГц и 8-разрядной шиной данных. За ним последовали новые поколения персональных компьютеров, с частотой выпуска в 3-4 года: 8088 > 80286 > 80386 > 80486 > 80586 (Pentium) > и т.д. Каждая из этих новых систем увеличивала скорость вычисления приблизительно в *пять* раз, в сравнении со своим предшественником. В 1996 году тактовая частота достигла 200 МГц, а шина данных стала 32-разрядной. При других усовершенствованиях это выразилось в том, что вычислительная мощность выросла за 15 лет приблизительно в *тысячу* раз! Эта тенденция должна сохранится и в *последующие* 15 лет.

Другой путь осознания скорости развития в этой области заключен в предоставляемой производителями информации: реклама, спецификации, прайс-листы и др. Забудьте книги о производительности, о ней уже можно прочитать в газетах и журналах. Следует ожидать увеличение скорости вычисления приблизительно в два раза каждые два года. Изучение текущего состояния мощности компьютеров недостаточно, необходимо представлять себе ее эволюцию.

Держа это в памяти, можно перейти к обзору, как скорость вычисления ограничена аппаратной частью компьютера. Поскольку компьютеры состоят из множества подсистем, время, необходимое для выполнения отдельной задачи зависит от двух факторов: (1) скорость отдельных подсистем и (2) время передачи данных между блоками. Рис.4-5 показывает упрощенную диаграмму основных компонент типичного персонального компьютера, ограничивающих скорость выполнения. *Центральный процессор* (CPU) является сердцем системы. Как было описано ранее, он состоит из множества регистров, каждый из которых состоит из 32 разрядов (на время написания книги – прим. редактора). Также в ЦП включена цифровая электроника, необходимая для элементарных операций, например, для перемещения регистров и целочисленной арифметики.

Более сложная математика управляется передачей данных в специальную микросхему, называемую *математический сопроцессор* (или *арифметико-логическое устройство*, *АЛУ*). Математический сопроцессор может располагаться на одном кристалле с ЦП, или быть отдельным электронным устройством. Например, сложение двух вещественных чисел потребует от ЦП передать 8 байт (4 для каждого числа) в математический сопроцессор, и несколько байт для описания операции над ними. После короткого времени вычисления, математический сопроцессор передаст четыре байта результата, содержащего сумму чисел. Самые дешевые (и устаревшие – прим. редактора) компьютерные системы не включают в себя математический сопроцессор, или его наличие является опциональным. Например, микропроцессор 80486DX имеет внутренний математический сопроцессор, в то время как микропроцессор 80486SX нет. Системы с пониженной производительностью заменяют *аппаратно* реализуемые функции *программно*. Каждая математическая функция разбивается на элементарные бинарные операции, которые могут быть выполнены ЦП. В то время как это обеспечивает тот же результат, время выполнения гораздо больше, приблизительно в 10-20 раз.

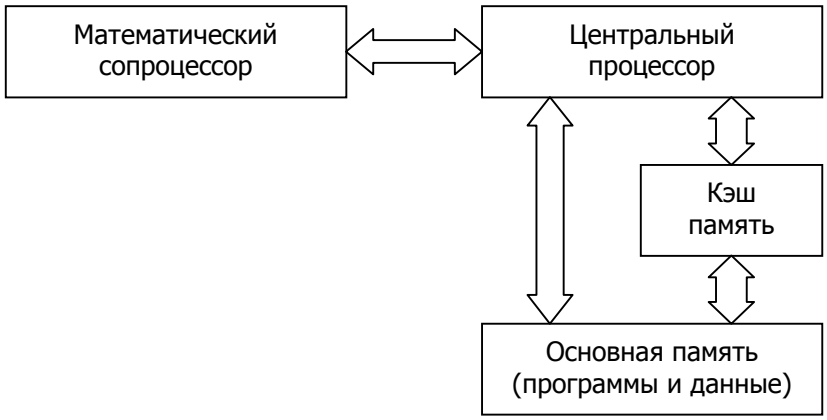


Рис.4-5. Архитектура типичной компьютерной системы.

Большинство программного обеспечения персональных компьютеров может не использовать математический сопроцессор. Это достигается тем, что компьютер генерирует машинный код для работы в обоих случаях, который хранится в исполняемой программе. Если математический сопроцессор присутствует в компьютерной системе, выполняется одна часть кода, если отсутствует, то используется другая. Компьютеру можно указать, чтобы он создавал код только для одной из этих ситуаций. Приложения вроде пословной обработки данных обычно не зависят от математического сопроцессора, так как они реализуют только алгоритмы перемещения данных в памяти, а не расчет математических выражений. Подобно им, расчеты, использующие целочисленные переменные также не требуют математического сопроцессора, поскольку все операции выполняет ЦП. С другой стороны, скорость выполнения при использовании вычислений с плавающей запятой может на порядок отличаться в зависимости от того, используется или нет математический сопроцессор.

ЦП и основная память в большинстве компьютерных систем расположены на отдельных кристаллах. По очевидным причинам, хочется, чтобы основная память была очень большой и очень быстрой. К сожалению, это сделает память очень дорогой. Передача данных между основной памятью и ЦП часто является критическим параметром в отношении скорости. ЦП *запрашивает* бинарную информацию в основной памяти по определенному адресу и должен *ждать* получение ответа. Распространенным способом решения этой проблемы является использование *кэш-памяти* (memory cache). Небольшое количество очень быстрой памяти используется в виде буфера между ЦП и основной памятью. В среднем оно составляет несколько сотен килобайт. Когда ЦП запрашивает бинарную информацию в основной памяти по определенному адресу, высокоскоростная цифровая электроника копирует *часть* основной памяти около этого адреса в кэш. В следующий раз, когда ЦП запросит информацию в основной памяти, существует большая вероятность, что она уже будет находится в кэше, поэтому получение данных произойдет гораздо быстрее. Этот метод основан на таком принципе, что программы чаще всего обращаются к ячейкам памяти, находящимися рядом с теми, которые были запрошены ранее. В типичных приложениях персональных компьютеров добавление кэш-памяти может увеличить скорость выполнения в несколько раз. Кэш-память может располагаться на том же кристалле, что и ЦП, или быть внешним электронным устройством.

Скорость, на которой данные передаются между подсистемами, зависит от количества предоставленных параллельных линий и максимальной скорости, с которой цифровые сигналы могут быть переданы по каждой линии. Цифровые данные обычно передаются на гораздо большей скорости внутри одного кристалла, чем между различными. Подобным образом, путь данных, проходящий через электронные коннекторы других печатных плат (т.е. шинную структуру) будет замедлять скорость передачи. Это является поводом по возможности размещать как можно больше электроники внутри ЦП.

Особенно неприятной проблемой скорости является *обратная совместимость*. Когда компания выпускает новое изделие, например, плату сбора данных или компьютерную программу, она хочет продать его на максимально большом рынке. Это означает, что оно должно быть совместимо с большинством используемых в настоящее время компьютерами, которые могут представлять разные поколения технологии. Это часто вызывает ограничение производительности аппаратного или программного обеспечения. Например, представьте, что вы купили плату ввода/вывода, которая вставляется в ваш персональный компьютер Pentium с тактовой частотой 200 МГц, обеспечивающий одновременную передачу и прием одного байта по восьми линиям. Далее, вы написали программу на ассемблере для быстрой передачи данных между вашим компьютером и некоторым внешним устройством. Максимальная скорость передачи составит только 100,000 байт в секунду, что более чем в *тысячу раз* меньше тактовой частоты микропроцессора! Виновником этого является шина ISA, технология, которая обратно совместима с компьютерами начала 80-х годов.

Таблица 4-6 приводит время выполнения операций для нескольких поколениях компьютеров. Вы должны воспринимать эту информацию как грубую аппроксимацию. Если вы хотите узнать параметры *вашей* системы, проведите измерения на *ней*. Самый простой способ: напишите цикл, который выполняет *миллион* некоторых операций и замерьте время его выполнения с помощью таймера. Первые три системы, 80286, 80486 и Pentium, являются стандартными настольными компьютерными системами, созданными в 1986, 1993 и 1996 годах, соответственно. Четвертая является микропроцессором, специально созданным в 1994 году фирмой Texas Instruments для задач ЦОС, TMS320C40.

	80286 (12 МГц)	**80486 (33 МГц)**	**Pentium (100 МГц)**	**TMS320C40 (40 МГц)**
ЦЕЛЫЕ				
A% = B% + C%	1.6	0.12	0.04	
A% = B% - C%	1.6	0.12	0.04	
A% = B% x C%	2.7	0.59	0.13	
A% = B% / C%	64	9.2	1.5	
ВЕЩЕСТВЕННЫЕ				
A = B+C	33	2.5	0.5	0.1
A = B – C	35	2.5	0.5	0.1
A = B x C	35	2.5	0.5	0.1
A = B / C	49	4.5	0.87	0.8
A = SQR(B)	45	5.3	1.3	0.9
A = LOG(B)	186	19	3.4	1.7
A = EXP(B)	246	25	5.5	1.7
A = B ^ C	311	31	5.3	2.4
A = SIN(B)	262	30	6.6	1.1
A = ARCTAN)B)	168	21	4.4	2.2

Таблица 4-6. Измеренное время выполнения операций для разных компьютерных систем. Время указано в миллисекундах.

Pentium быстрее системы 80286 по четырем причинам: (1) большая тактовая частота, (2) большее число линий на шине, (3) увеличение кэш-памяти, (4) более эффективная внутренняя организация, требующая меньшее число циклов для выполнения инструкции.

Pentium был «Кадиллаком», TMS320C40 – «Феррари»: менее комфортный, но с ошеломляющей скоростью. Кристалл является типичным представителем микропроцессоров, специально спроектированных для уменьшения времени выполнения алгоритмов ЦОС. В этой категории можно назвать также микропроцессоры Intel i860, AT&T DSP3210, Motorola DSP96002 и Analog Devices ADSP-2171. Чаще всего их называют *DSP микропроцессорами*, *цифровыми процессорами сигналов* (ЦПС), *цифровыми сигнальными процессорами* (ЦСП) или *RISC* (Reduced Instruction Set Computer – компьютеры с сокращенным набором команд). Последнее название отражает увеличенную скорость обработки за счет уменьшения числа ассемблерных инструкций, употребляемых программистом для реализации некоторой операции. В сравнении с ним, для обычных микропроцессоров, например, Pentium, вводится название *CISC* (Complex Instruction Set Computer – компьютеры с полным набором команд).

ЦПС часто используются в двух назначениях: как ведомые модули под управлением более традиционных компьютеров, или как встроенные процессоры в определенных приложениях, например, сотовых телефонах. Некоторое модели ЦПС работают только с числами с фиксированной запятой, в то время как другие поддерживают формат с плавающей запятой. Внутренняя архитектура, используемая для увеличения скорости, включает в себя: (1) большое количество очень быстрой кэш-памяти, расположенной на кристалле, (2) раздельные шины для программ и данных, позволяющие осуществлять независимый доступ (так называемая *Гарвардская архитектура*), (3) быстрая аппаратная часть для математических расчетов, располагающаяся непосредственно в микропроцессоре и (4) *конвейерную* организацию.

Конвейерная архитектура (pipeline architecture) разбивает работу *аппаратных средств*, необходимых для определенной задачи на несколько последовательных этапов. Например, сложение двух чисел может быть разбито на три конвейерных этапа. На первом, конвейер только выбирает из памяти числа, которые необходимо сложить. Задачей второго этапа является сложение чисел. На третьем этапе происходит сохранение результата сложения в памяти. Каждый этап может быть выполнен за один командный цикл, вся процедура сложения займет три цикла. Ключевой характеристикой конвейерной архитектуры является то, что каждая задача может быть выполнена перед тем, как закончится предыдущая. В нашем примере, мы можем начать сложение *других* двух чисел как только первый этап перейдет в нерабочий (холостой) режим, по завершению первого тактового цикла. Для операций с множеством чисел, скорость выполнения системы будет равна одному командному циклу на операцию, вместо трех. Конвейерная организация повышает скорость, но вызывает сложность в программировании. Алгоритм должен разрешать начало выполнения нового расчета до того, как будет получен результат предыдущего вычисления (поскольку он еще находится на конвейере).

Главы 28 и 29 обсуждают ЦПС более детально. Эти устройства являются уникальными, поскольку обладают высокой вычислительной мощностью при низкой стоимости, и могут применяться в широком диапазоне научных приложений. ЦПС являются технологией XXI века!

Скорость выполнения: приемы программирования

В то время как аппаратная часть компьютеров и языки программирования являются важной частью оптимизации скорости выполнения, они не являются чем-то, что меняется ежедневно. В сравнение с этим, *навык вашего программирования* может меняться постоянно с приобретением опыта, и оказывать существенное воздействие на улучшение ваших программ. Далее будут приведены три совета, касающихся приемов программирования.

Во-первых, где это возможно, используйте целые числа вместо вещественных. Традиционные микропроцессоры, которые используются в персональных компьютерах, обрабатывают целые числа в 10-20 раз быстрее, чем вещественные. На системах без математического сопроцессора, разница может составлять соотношение 200:1. Исключением является целочисленное деление, которое часто использует преобразование целых чисел в формат с плавающей запятой, что делает операцию ужасно медленной в сравнение с другими целочисленными вычислениями.

Во-вторых, избегайте использования таких функций, как sin(x), log(x), y^X и т.п. Трансцендентные функции вычисляются как последовательность операций сложения, вычитания и умножения. Например, степенные ряды Макларена:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \ldots$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \frac{x^{10}}{10!} + \ldots$$

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \ldots$$

Уравнение 4-3. Степенные ряды Макларена для трех трансцендентных функций.

В то время как эти отношения бесконечны по длине, члены степенного ряда быстро принимают величины, которые можно игнорировать. например:

$$\sin(1) = 1 - 0.166666 + 0.008333 - 0.000198 + 0.000002 - \ldots$$

Такие выражения рассчитываются приблизительно в десять раз дольше, чем обычное сложение или умножение (см. Таблицу 4-6). Существует несколько приемов программирования, которые целесообразно применять для расчетов, например: $x^3 = x \cdot x \cdot x$; $\sin(x) \approx x$ при маленьких значениях x;sin(-x) = sin(x) и т.д. Большинство языков программирования поддерживают трансцендентные функции и позволяют получить другие с их использованием (см. Таблицу 4-7). Не удивительно, что выведенные вычисления гораздо медленнее.

ФУНКЦИЯ	ВЫРАЖЕНИЕ ДЛЯ РАСЧЕТА
Секанс (X)	1/COS(X)
Косеканс (X)	1/SIN(X)
Котангенс (X)	1/TAN(X)
Арксинус (X)	ATN(X/SQR(1-X*X))
Арккосинус (X)	-ATN(X/SQR(1-X*X)) + PI/2
Арксеканс (X)	ATN(SQR(X*X-1) + (SGN(X)-1)) * PI/2
Арккосинус (X)	ATN(1/SQR(X*X-1) + (SGN(X)-1)) * PI/2
Арккотангенс (X)	-ATN(X) + PI/2
Гиперболический синус (X)	(EXP(X) – EXP(-X))/2
Гиперболический косинус (X)	(EXP(X) + EXP(-X))/2
Гиперболический тангенс (X)	(EXP(X) – EXP(-X)) / (EXP(X) + EXP(-X))
Гиперболический секанс (X)	1 / Гиперболический косинус
Гиперболический косеканс (X)	1 / Гиперболический синус
Гиперболический котангенс (X)	1 / Гиперболический тангенс
Гиперболический арксинус (X)	LOG(X + SQR(X*X+1))
Гиперболический арккосинус (X)	LOG(X + SQR(X*X-1))
Гиперболический арктангенс (X)	LOG((1+X) / (1-X)) /2
Гиперболический арксеканс (X)	LOG((SQR(1-X*X)+1)/X)
Гиперболический арккосеканс (X)	LOG(1+SGN(X)*SQR(X*X+1))/X
Гиперболический арккотангенс (X)	LOG((X+1)/(X-1))/2
LOG_{10} (X)	LOG(X) / LOG(10) = 0.4342945 · LOG(X)
PI	4*ATN(1)

Таблица 4-7. Расчет редко используемых функций из более распространенных.

Другим способом является предварительный расчет этих медленных функций и сохранение результатов в *справочной таблице* (look-up table). Например, представьте, что имеется 8-битная система сбора данных, используемая для непрерывного мониторинга *напряжения*, проходящего через резистор. Если интересующим параметром является *мощность*, рассеиваемая на резисторе, измеренное напряжение может быть использовано в вычислениях как: $P = V^2 / R$. Для оптимизации алгоритма расчета, 256 значений напряжения могут быть рассчитаны заранее и результаты сохранены в справочной таблице. При запуске системы, измеренное напряжение, номер между 0 и 255, преобразуется в индекс справочной таблицы, из которой находят соответствующую мощность. Справочные таблицы позволяют увеличить скорость выполнения в сотни раз!

В-третьих, выясните что является быстрым, и что медленным в вашей системе. Это достигается испытаниями и тестами, во время которых можно получить удивляющие результаты. Обратите свое внимание на команды *вывода графики* и операции *ввода/вывода*. Существует несколько способов увеличить скорость их выполнения. Например, оператор BASIC *BLOAD* передает данные из файла непосредственно в память компьютера. Чтение того же файла побайтно может выполняться почти в 100 раз медленнее. Также, помещение оператора *PRINT* внутри управляющего цикла может замедлить выполнение программы почти в *тысячу раз*!