Ташкентский Университет Информационных Технологий

Тема: «Арифметические команды»

Выполнил: Ахатов А.
Группа: 282-11 ХКТТр
Принял: Сейтназоров К.К.

Арифметические команды

- Форматы арифметических данных
- Арифметические операции над двоичными числами
- Арифметические операции над десятичными (BCD) числами

Одной из причин, постоянно заставляющих человека совершенствовать средства для выполнения вычислений, — желание эффективно, быстро и без ошибок решать различные счетные задачи. Для начала мечтой людей была автоматизация выполнения простейших арифметических действий. Первая реализованная попытка — начало XVII в., 1623 г. Ученый В. Шикард создает машину, умеющую складывать и вычитать числа. Знаменитый французский ученый и философ Блез Паскаль в 1642 г. изобрел первый *арифмометр,* основным элементом в котором было зубчатое колесо. Изобретение этого колеса уже само по себе было ключевым событием в истории вычислительной техники, подобно лампам и транзисторам в наше время. Правнуки этого колеса еще совсем недавно, каких-нибудь два-три десятка лет назад, использовались в арифмометрах (соответствующая модель была создана в 1842 г.) на столах советских бухгалтеров. Тот, кому довелось поработать на этих арифмометрах, вряд ли вспомнят о высокой эффективности вычислительного процесса — слишком велика была зависимость от человеческого фактора. Снизить эту зависимость удалось лишь в середине прошлого века, когда появились первые ЭВМ на лампах, потом на транзисторах и, наконец, на микросхемах различной интеграции. Таким образом, путь к эффективному автоматизированному решению для проведения расчетов растянулся почти на три столетия. Тем не менее, именно благодаря стремлению разгрузить голову от рутины человек имеет сегодня определенные достижения в области компьютерной техники.

Любой компьютер, от самого примитивного до супермощного, имеет в своей системе команд команды для выполнения арифметических действий. Работая с компьютером при помощи языков высокого уровня, мы воспринимаем возможность проведения расчетных действий как нечто должное, забывая при этом, что компилятор даже очень развитого языка программирования превращает все самые высокоуровневые действия в унылую последовательность машинных команд. Конечно, мало кому придет в голову писать серьезную расчетную задачу на ассемблере. Но даже в системных программах часто требуется проведение небольших вычислений. Поэтому важно разобраться с этой группой команд. К тому же она, на

удивление, очень компактна и не избыточна.

Процессор может выполнять целочисленные операции и операции с плавающей точкой. Для этого в его архитектуре есть два отдельных устройства, каждое из которых имеет свою систему команд. В принципе, целочисленное устройство может взять на себя многие функции устройства с плавающей точкой, но это потребует больших вычислительных затрат. Устройство с плавающей точкой и его система команд будут рассмотрены в главе 17. Для большинства задач, использующих язык ассемблера, достаточно целочисленной арифметики.

Обзор

Целочисленное вычислительное устройство поддерживает чуть больше десятка арифметических команд. На рис. 8.1 приведена классификация команд этой группы.

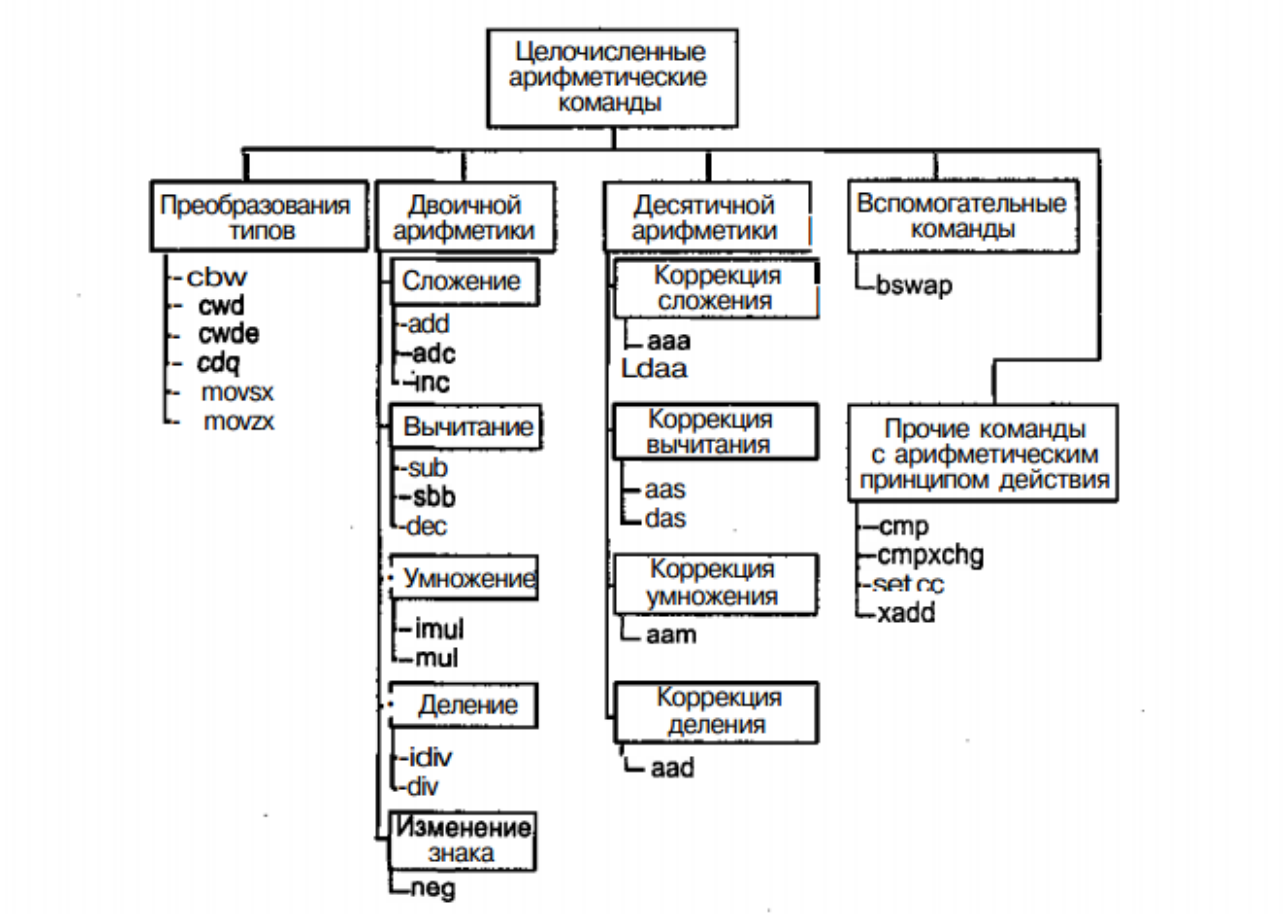

Рис. 8.1. Классификация арифметических команд

Группа арифметических целочисленных команд работает с двумя типами чисел:

* целыми двоичными числами, которые могут иметь или не иметь

знаковый разряд, то есть быть числами со знаком или без знака;

* целыми десятичными числами.

В главе 3 мы обсуждали вопрос о форматах данных, поддерживаемых архитектурой IA-32. Рассмотрим теперь форматы данных, с которыми работают арифметические команды.

Целые двоичные числа

Целое двоичное число — это число, закодированное в двоичной системе счисления. В архитектуре IA-32 размерность целого двоичного числа может составлять 8, 16 или 32 бита. Знак двоичного числа определяется тем, как интерпретируется старший бит в представлении числа. Это 7-й, 15-й или 31-й биты для чисел соответствующей размерности (см. главу 5). При этом интересно то, что среди арифметических команд есть всего две, которые действительно учитывают этот старший разряд как знаковый, — это команды целочисленного умножения **IMUL** и деления IDIV. В остальных случаях ответственность за действия со знаковыми числами и, соответственно, со знаковым разрядом ложится на программиста. К этому вопросу мы вернемся чуть позже. Диапазон значений двоичного числа зависит от его размера и трактовки старшего бита либо как старшего значащего бита числа, либо как бита знака числа (табл. 8.1).

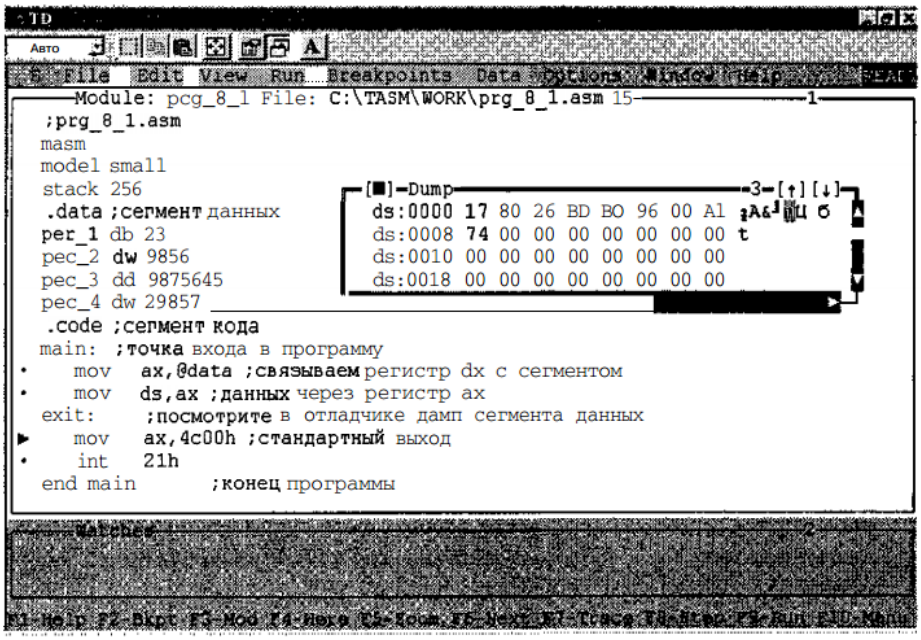

Рис. 8.2. Дамп памяти для сегмента данных листинга 8.1

Таблица 8.1. Диапазон значений двоичных чисел

Размерность	Целое без	Целое со знаком
Байт	0...255	-128...+ 127
Слово	0...65 535	-32 768...+32 767
Двойное слово	0...4 294 967	-2 147 483 648...+2 147

Как описать целые двоичные числа в программе? Это делается с использованием директив описания данных DB, DW и DD. В главе 5 описаны возможные варианты содержимого полей операндов этих директив и диапазоны их значений. К примеру, последовательность описаний двоичных чисел из сегмента данных листинга 8.1 (помните о принципе «младший байт по младшему адресу») будет выглядеть в памяти так, как показано на рис. 8.2.

Листинг 8.1. Числа с фиксированной точкой

```
;prg_8_1.asm
masm
model   small
stack   256
.data                   ;сегмент данных
per_1   db  23
per_2   dw  9856
per_3   dd  9875645
per_4   dw  29857
.code                   ;сегмент кода
main:                   ;точка входа в программу
    mov ax,@data;связываем регистр dx с сегментом
    mov ds,ax           ;данных через регистр ax
exit:                   ;посмотрите в отладчике дамп сегмента данных
    mov ax,4c00h;стандартный выход
    int 21h
end main                ;конец программы
```

Десятичные числа

Десятичные числа — специальный вид представления числовой информации, в основу которого положен принцип кодирования каждой десятичной цифры числа группой из четырех битов. При этом каждый байт числа содержит одну или две десятичные цифры в так называемом *двоично-десятичном коде* (Binary-Coded Decimal, BCD). Процессор хранит BCD-числа в двух форматах (рис. 8.3).

- В *упакованном формате* каждый байт содержит две десятичные цифры. Десятичная цифра представляет собой двоичное значение в диапазоне от 0 до 9 размером четыре бита. При этом код старшей цифры числа занимает старшие четыре бита. Следовательно, диапазон представления десятичного упакованного числа в одном байте составляет от 00 до 99.

- В *неупакованном формате* каждый байт содержит одну десятичную цифру в четырех младших битах. Старшие четыре бита имеют нулевое значение. Это так называемая *зона.* Следовательно, диапазон представления десятичного неупакованного числа в одном байте составляет от 0 до 9.

Как описать двоично-десятичные числа в программе? Для этого можно использовать только две директивы описания и инициализации данных — DB и DT.

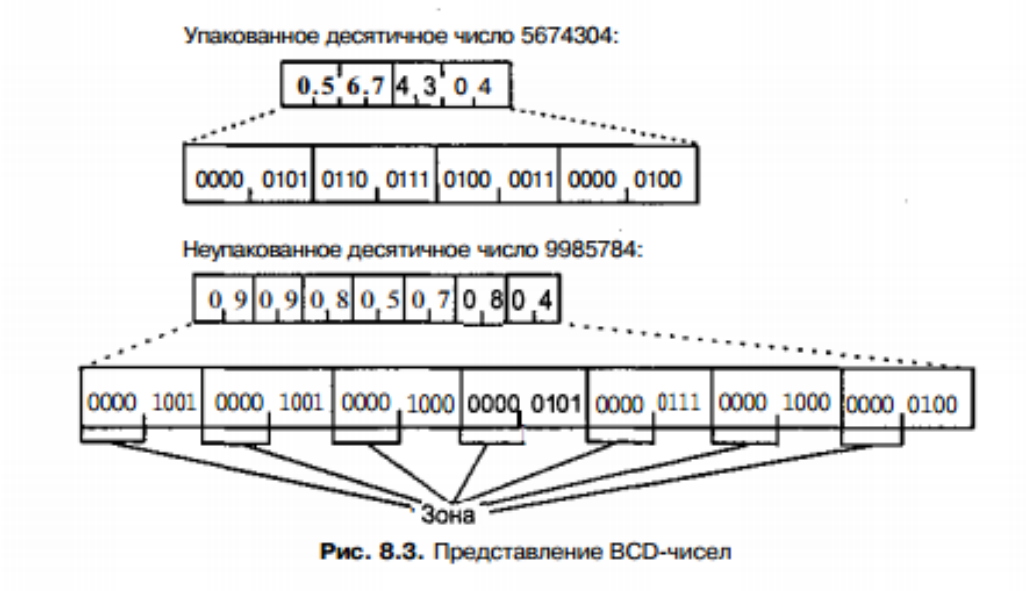

Рис. 8.3. Представление BCD-чисел

Рис. 8.3. Представление BCD-чисел

Возможность применения только этих директив для описания BCD-чисел обусловлена тем, что к таким числам также применим принцип «младший байт по младшему адресу», что, как мы увидим далее, очень удобно для их обработки. И вообще, при использовании такого типа данных, как BCD-числа, порядок описания этих чисел в программе и алгоритм их обработки — это дело вкуса и личных пристрастий программиста, что станет более ясным после того, как мы далее рассмотрим основы работы с BCD-числами. К примеру, приведенная в сегменте данных листинга 8.2 последовательность описаний BCD-чисел будет выглядеть в памяти так, как показано на рис. 8.4.

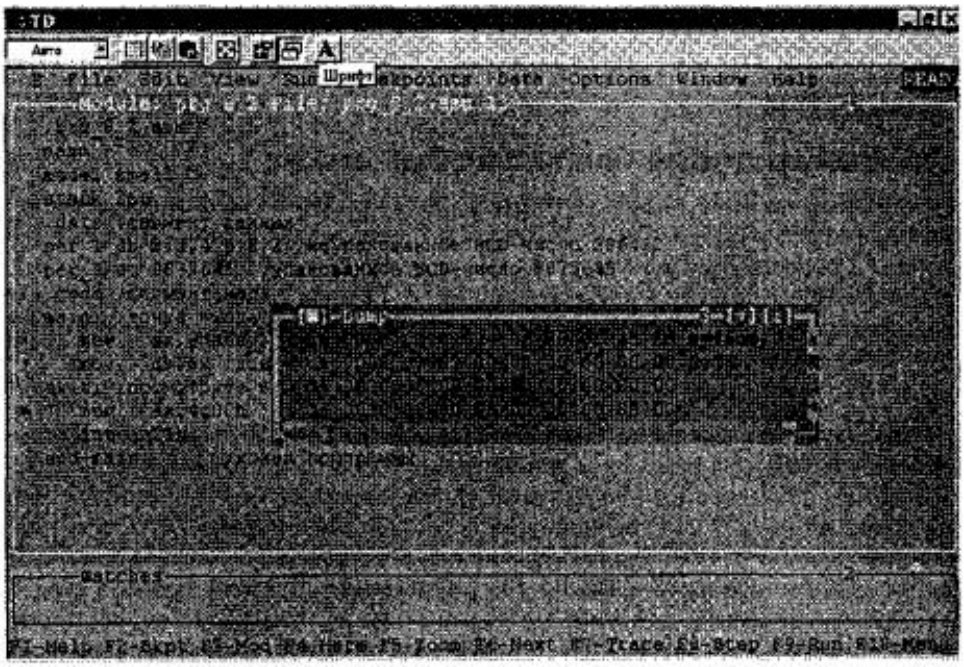

Рис. 8.4. Дамп памяти для сегмента данных листинга 8.2

Листинг 8.2. BCD-числа

```
;prg_8_2.asm
masm
model   small
stack   256
.data                       ;сегмент данных
per_1   db  2,3,4,6,8,2 ;неупакованное BCD-число 286432
per_3   dt  9875645     ;упакованное BCD-число 9875645
.code                       ;сегмент кода
main:                      ;точка входа в программу
    mov ax,@data           ;связываем регистр dx с сегментом
    mov ds,ax              ;данных через регистр ax
exit:                      ;посмотрите в отладчике дамп сегмента данных
    mov ax,4c00h           ;стандартный выход
    int 21h
end main                   ;конец программы
```

После столь подробного обсуждения форматов данных, с которыми работают арифметические операции, можно приступить к рассмотрению средств их обработки на уровне системы команд процессора.

Арифметические операции над целыми двоичными числами

В данном разделе мы рассмотрим особенности каждого из четырех основных арифметических действий для целых двоичных чисел со знаком и без знака. Дополнительные сведения о возможностях ассемблера по обработке целых двоичных чисел можно получить в [8].

Сложение двоичных чисел без знака

Процессор выполняет сложение операндов по правилам сложения двоичных чисел. Проблем не возникает до тех пор, пока значение результата не превышает размерности поля операнда (см. табл. 8.1). Например, при сложении операндов размером в байт результат не должен превышать 255. Если это происходит, то результат оказывается неверен. Рассмотрим, почему. К примеру, выполним сложение: 254 + 5 = 259 в двоичном виде:

11111110 + 0000101 - 1 00000011.

Результат вышел за пределы восьми битов, и правильное его значение укладывается в 9 битов, а в 8-разрядном поле операнда осталось значение 3, что, конечно, неверно. В процессоре подобный исход сложения прогнозируется, и предусмотрены специальные средства для фиксации подобных ситуаций и их обработки. Так, для фиксации ситуации выхода за разрядную сетку результата, как в данном случае, предназначен *флаг переноса* CF. Он располагается в бите 0 регистра флагов **EFLAGS/FLAGS.** Именно установкой этого флага фиксируется факт переноса единицы из старшего разряда операнда. Естественно, что программист должен предусматривать возможность такого исхода операции сложения и средства для корректировки. Это предполагает включение фрагментов кода после операции сложения, в которых анализируется состояние флага CF. Этот анализ можно провести различными способами. Самый простой и доступный — использовать команду условного перехода JC. Эта команда в качестве операнда имеет имя метки в текущем сегменте кода. Переход на метку осуществляется в случае, если в результате работы предыдущей команды флаг CF установился в 1. Команды условных переходов будут рассматриваться в главе 10.

Если теперь посмотреть на рис. 8.1**,** то видно, что в системе команд процессора имеются три команды двоичного сложения:

- команда *инкремента,* то есть увеличения значения операнда на 1:
  inc операнд
- команда сложения (операнд_1 = операнд_1 + операнд_2):
  add операнд_1,операнд_2
- команда сложения с учетом флага переноса CF(операнд_1 = операнд_1+ +операнд_2 + значение_CF):
  adc операнд_1,операнд_2

Обратите внимание на последнюю команду — это команда сложения, учитывающая перенос единицы из старшего разряда. Механизм появления

такой единицы мы уже рассмотрели. Таким образом, команда AD C является средством процессора для сложения длинных двоичных чисел, размерность которых превосходит поддерживаемые процессором размеры стандартных полей.

Рассмотрим пример вычисления суммы чисел (листинг 8.3).

Листинг 8.3. Вычисление суммы чисел

```
<1> ;prg_8_3.asm
<2> masm
<3> model   small
<4> stack    256
<5> .data
<6> a   db  254
<7> .code                ;сегмент кода
<8> main:
<9>     mov ax,@data
<10>        mov ds,ax
<11>    ;...
<12>        xor  ax,ax
<13>        add al,17
<14>        add al,a
<15>        jnc m1       ;если нет переноса, то перейти на m1
<16>        adc ah,0     ;в ax сумма с учетом переноса
<17>   m1: ;...
<18>        exit:
<19>        mov ax,4c00h;стандартный выход
<20>        int 21h
<21>    end main         ;конец программы
```

В строках 13-14 создана ситуация, когда результат сложения выходит за границы операнда. Эта возможность учитывается строкой 15, где команда **JNC** (хотя можно было обойтись и без нее) проверяет состояние флага **CF.** Если он установлен в 1, то результат операции получился большим по размеру, чем операнд, и для его корректировки необходимо выполнить некоторые действия. В данном случае мы просто полагаем, что границы операнда расширяются до размера AX, для чего учитываем перенос в старший разряд командой AD C (строка 16). Напомню, что исследовать работу команд сложения без учета знака вы можете в отладчике. Для этого введите в текстовом редакторе текст листинга 8.3, получите исполняемый модуль, запустите отладчик и откройте в нем окна командами View ► Dump и **View** ► Registers. Далее, в пошаговом режиме отладки можно более наглядно проследить за всеми процессами, происходящими в процессоре во время работы программы.

Сложение двоичных чисел со знаком

Теперь настала пора раскрыть небольшой секрет. Дело в том, что на самом деле процессор не подозревает о различии между числами со знаком и

числами без знака. Вместо этого у него есть средства фиксации возникновения характерных ситуаций, складывающихся в процессе вычислений. Некоторые из них мы рассмотрели ранее при обсуждении команд сложения чисел без знака — это учет флага переноса CF. Установка этого флага в 1 говорит о том, что произошел выход за пределы разрядности операндов. Далее с помощью команды ADC можно учесть возможность такого выхода (переноса из младшего разряда) во время работы программы. Другое средство фиксации характерных ситуаций в процессе арифметических вычислений — регистрация состояния старшего (знакового) разряда операнда, которое осуществляется с помощью флага переполнения OF в регистре EFLAGS **(бит И).**

В главе 4 мы рассматривали, как представляются числа в компьютере. При этом отмечали, что положительные числа представляются в двоичном коде, а отрицательные — в дополнительном. Рассмотрим различные варианты сложения чисел. Примеры призваны показать поведение двух старших битов операндов и правильность результата операции сложения.

Первый вариант сложения чисел:

30566 = 01110111 01100110

+

00687 = 00000010 10101111

-

31253 = 0111101000010101.

Следим за переносами из 14-го и 15-го разрядов и за правильностью результата: переносов нет, результат правильный.

Второй вариант сложения чисел:

30566 = 0111011101100110

+

30566 = 0111011101100110

-

61132 = 11101110 11001100.

Произошел перенос из 14-го разряда; из 15-го разряда переноса нет. Результат неправильный, так как имеется *переполнение* — значение числа получилось больше, чем то, которое может иметь 16-разрядное число со знаком (+32 767).

Третий вариант сложения чисел:

-30566 = 10001000 10011010

+

-04875= 11101100 11110101

-

-35441 = 01110101 10001111.

Произошел перенос из 15-го разряда, из 14-го разряда нет переноса. Результат неправильный, так как вместо отрицательного числа получилось положительное (в старшем бите находится 0).

Четвертый вариант сложения чисел:

-4875 = 11101100 11110101

+

-4875 = 1110110011110101

-

-9750 = 11011001 11101010.

Есть переносы из 14-го и 15-го разрядов. Результат правильный.

Таким образом, мы исследовали все случаи и выяснили, что ситуация ***переполнения*** (установка флага OF в 1) происходит при переносе:

* из 14-го разряда (для положительных чисел со знаком);

в из 15-го разряда (для отрицательных чисел).

И, наоборот, переполнения не происходит (то есть флаг OF сбрасывается в 0), если есть перенос из обоих разрядов или перенос отсутствует в обоих разрядах.

Таким образом, ситуация переполнение регистрируется процессором с помощью флага переполнения **OF,** Дополнительно к флагу OF при переносе из старшего разряда устанавливается в 1 и флаг переноса CF. Так как процессор не знает о существовании чисел со знаком и без знака, то вся ответственность за правильность действий с получившимися числами ложится на программиста. Теперь, наверное, понятно, почему мы столько внимания уделили тонкостям сложения чисел со знаком. Учтя все это, мы сможем организовать правильный процесс сложения чисел — будем анализировать флаги CF и OF и принимать правильное решение! Проанализировать флаги CF и OF можно командами условного перехода **JC\JNC** и **JO\JNO** соответственно (глава 10).

Что же касается команд сложения чисел со знаком, то из изложенного ранее понятно, что в архитектуре IA-32 сами команды сложения чисел со знаком те же, что и для чисел без знака.

Вычитание двоичных чисел без знака

Аналогично анализу операции сложения, порассуждаем над сутью процессов, происходящих при выполнении вычитания.

- Если уменьшаемое больше вычитаемого, то проблем нет, — разность положительна, результат верен.

- Если уменьшаемое меньше вычитаемого, возникает проблема: результат меньше 0, а это уже число со знаком. В этом случае результат необходимо *завернуть*. Что это означает? При обычном вычитании (в столбик) делают заем 1 из старшего разряда. Процессор поступает аналогично, то есть занимает 1 из разряда, следующего за старшим в разрядной сетке операнда. Поясним на примере.

  Первый вариант вычитания чисел:

  05=0000000000000101

  -10 = 0000000000001010.

Для того чтобы произвести вычитание, произведем воображаемый заем из старшего разряда:

100000000 00000101

-

0000000000001010

=

11111111 11111011.

Тем самым, по сути, выполняется действие (65 536 + 5) - 10 = 65 531, 0 здесь как бы эквивалентен числу 65 536. Результат, конечно, неверен, но процессор считает, что все нормально, тем не менее, факт заема единицы он фиксирует, устанавливая флаг переноса CF. Посмотрите еще раз внимательно на результат операции вычитания. Это же число -5 в дополнительном коде! Проведем эксперимент: представим разность в виде суммы 5 + (-10).

Второй вариант вычитания чисел:

5 = 00000000 00000101

+

(-10) = 11111111 11110110

=

11111111 11111011.

То есть мы получили тот же результат, что и в предыдущем примере. Таким образом, после команды вычитания чисел без знака нужно анализировать состояние флага CF. Если он установлен в 1, это говорит о том, что произошел заем из старшего разряда и результат получился в дополнительном коде.

Аналогично командам сложения группа команд вычитания состоит из минимально возможного набора. Эти команды выполняют вычитание по алгоритмам, которые мы сейчас рассматриваем, а учет особых ситуаций должен производиться самим программистом.

* Команда декремента выполняя уменьшения значения операнда на 1:

dec операнд :

* Команда вычитания (операнд_1 = операнд_1 - операнд_2):

sub операнд_1,операнд_2

8 Команда вычитания с учетом заема, то есть флага CF (операнд_1 = операнд_1 - операнд_2 - значение_CP):

sbb операнд_1,операнд_2

Как видите, среди команд вычитания есть команда SBB, учитывающая флаг переноса CF. Эта команда подобна AD C, но теперь уже флаг CF играет роль индикатора заема 1 из старшего разряда при вычитании чисел.

Рассмотрим пример (листинг 8.4) программной обработки ситуации, рассмотренной ранее для второго варианта вычитания чисел.

Листинг 8.4. Проверка при вычитании чисел без знака

```
<1> ;prg_8_4.asm
<2> masm
<3> model   small
<4> stack    256
<5> .data
<6> .code                       ;сегмент кода
<7> main:                       ;точка входа в программу
<8> ;...
<9>     xor ax,ax
<10>        mov al,5
<11>        sub al,10
<12>        jnc m1              ;нет  переноса?
<13>        neg al              ;в al модуль результата
<14>    m1: ;...
<15>        exit:
<16>        mov ax,4c00h;стандартный выход
<17>        int 21h
<18>    end main            ;конец программы
```

В этом примере в строке 11 выполняется вычитание. С указанными для этой команды вычитания исходными данными результат получается в дополнительном коде (отрицательный). Для того чтобы преобразовать результат к нормальному виду (получить его модуль), применяется команда NEG, с помощью которой получается дополнение операнда. В нашем случае мы получили дополнение дополнения, или модуль отрицательного результата. А тот факт, что это на самом деле число отрицательное, отражен в состоянии флага CF. Дальше все зависит от алгоритма обработки. Исследуйте программу в отладчике.

Вычитание двоичных чисел со знаком

Вычитание двоичных чисел со знаком выполнять несколько сложнее. Последний пример показал то, что процессору незачем иметь два устройства

— сложения и вычитания. Достаточно наличия только одного — устройства сложения. Но для вычитания способом сложения чисел со знаком оба операнда (и уменьшаемое, и вычитаемое) необходимо представлять в дополнительном коде. Результат тоже нужно рассматривать как значение в дополнительном коде. Но здесь возникают сложности. Прежде всего, они связаны с тем, что старший бит операнда рассматривается как знаковый. Рассмотрим пример вычитания 45 - (-127).

Первый вариант вычитания чисел со знаком:

45 =00101101

-

-127 = 10000001

=

-44 = 1010 1100.

Судя по знаковому разряду, результат получился отрицательный, что, в свою очередь, говорит о том, что число нужно рассматривать как дополнение, равное -44. Правильный результат должен быть равен 172. Здесь мы, как и в случае знакового сложения, встретились с ***переполнением мантиссы,*** когда значащий разряд числа изменил знаковый разряд операнда. Отследить такую ситуацию можно по содержимому флага переполнения OF. Его установка в 1 говорит о том, что результат вышел за диапазон представления знаковых чисел (то есть изменился старший бит) для операнда данного размера и программист должен предусмотреть действия по корректировке результата.

Следующее вычитание чисел со знаком выполним способом сложения:

- 45 – 45 – 45 **+ (-45)= - 90.**

-415 = 1101 0011

+

-45 =11010011

=

-90 = 1010 0110.

Здесь все нормально, флаг переполнения 0F сброшен в 0, а 1 в знаковом разряде говорит о том, что значение результата — число в дополнительном коде.

Вычитание и сложение операндов большой размерности

Если вы заметили, команды сложения и вычитания работают с операндами фиксированной размерности: 8, 16, 32 бита. А что делать, если

нужно сложить числа большей размерности, например 48 битов, используя 16-разрядные операнды? К примеру, сложим два 48-разрядных числа (рис. 8.5).

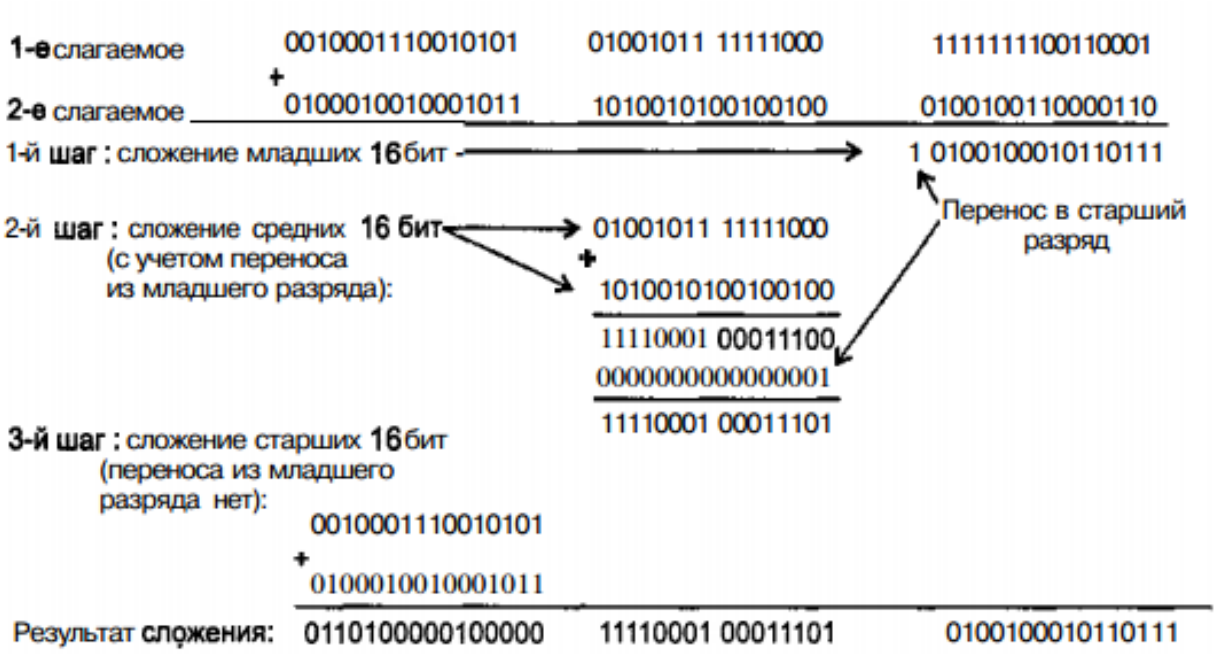

Рис. 8.5. Сложение операндов большой размерности

Рисунок по шагам иллюстрирует технологию сложения длинных чисел. Видно, что процесс сложения многобайтных чисел происходит так же, как и при сложении двух чисел «в столбик», — с осуществлением при необходимости переноса 1 в старший разряд. Если нам удастся запрограммировать этот процесс, то мы значительно расширим диапазон двоичных чисел, над которыми можно выполнять операции сложения и вычитания.

Принцип вычитания чисел с диапазоном представления, превышающим стандартные разрядные сетки операндов, тот же, что и при сложении, то есть используется флаг переноса CF. Нужно только представлять себе процесс вычитания в столбик и правильно комбинировать команды процессора с командой **SB** B. Чтобы написать достаточно интересную программу, моделирующую этот процесс, необходимо привлечь те конструкции языка ассемблера, которые мы еще не обсуждали. Среди прилагаемых к книге файлов в каталоге данной главы находятся исходные тексты подпрограмм, реализующих четыре основных арифметических действия для двоичных операндов произвольной размерности. Не поленитесь внимательно изучить их, так как они являются хорошей иллюстрацией к материалу, изучаемому в этой и последующих главах. К этим примерам можно будет обратиться в - полной мере после того, как будут изучены механизмы процедур и

макрокоманд (главы 14 и 15). Кроме того, в [8] обсуждаются вопросы программирования целочисленных арифметических операций, но в расширенном контексте.

В завершение обсуждения команд сложения и вычитания отметим, что кроме флагов CF и OF в регистре EFLAGS есть еще несколько флагов, которые можно использовать с двоичными арифметическими командами. Речь идет о следующих флагах:

- ZF — флаг нуля, который устанавливается в 1, если результат операции равен O, и в 0, если результат не равен 0;
- SF — флаг знака, значение которого после арифметических операций (и не только) совпадает со значением старшего бита результата, то есть с битом 7,15 или 31 (таким образом, этот флаг можно использовать для операций над числами со знаком).

Умножение двоичных чисел без знака

Для умножения чисел без знака предназначена команда

mul сомножитель_1

Как видите, в команде указан всего лишь один операнд-сомножитель. Второй операнд-сомножитель задан неявно. Его местоположение фиксировано и зависит от размера сомножителей. Так как в общем случае результат умножения больше, чем любой из его сомножителей, то его размер и местоположение должны быть тоже определены однозначно. Варианты размеров сомножителей и мест размещения второго операнда и результата приведены в табл. 8.2.

Таблица 8.2. Расположение операндов и результата при умножении

Первый	Второй	Результат
Байт	AL	16 битов в AX: AL — младшая часть результата; AH — старшая часть результата
Слово	AX	32 бита в паре DX:AX: AX — младшая часть результата; DX — старшая часть результата
Двойное слово	EAX	64 бита в паре EDX:EAX: EAX — младшая часть результата; EDX — старшая часть результата

Из таблицы видно, что произведение состоит из двух частей и в

зависимости от размера операндов размещается в двух местах — на месте второго сомножителя (младшая часть) и в дополнительных регистрах AH, DX, EDX (старшая часть). Как же динамически (то есть во время выполнения программы) узнать, что результат достаточно мал и уместился в одном регистре или что он превысил размерность регистра и старшая часть оказалась в другом регистре? Для этого привлекаются уже известные нам флаги переноса CF и переполнения OF:

- если старшая часть результата нулевая, то после завершения операции CF = O и OF = 0;
- если же флаги CF и OF ненулевые, это означает, что результат вышел за пределы младшей части произведения и состоит из двух частей, что и нужно учитывать при дальнейшей работе.

Рассмотрим следующий пример программы (листинг 8.5).

Листинг 8.5. Умножение

```
<1> ;prg_8_5.asm
<2> masm
<3> model   small
<4> stack    256
<5> .data                    ;сегмент данных
<6> rez label   word
<7> rez_l   db  45
<8> rez_h   db  0
<9> .code                    ;сегмент кода
<10>    main:                ;точка входа в программу
<11>    ;...
<12>        xor ax,ax
<13>        mov al,25
<14>        mul rez_l
<15>        jnc m1           ;если нет переполнения, то на m1
<16>       mov rez_h,ah;старшую часть результата в rez_h
<17>   m1:
<18>        mov rez_l,al
<19>        exit:
<20>        mov ax,4c00h;стандартный выход
<21>        int 21h
<22>    end main             ;конец программы
```

В строке 14 производится умножение значения в rez_l на число в регистре AL. Согласно информации из табл. 8.2, результат умножения будет располагаться в регистре AL (младшая часть) и в регистре AH (старшая часть). Для выяснения размера результата в строке 15 командой условного перехода JNC анализируется состояние флага CF, и если оно не равно 1, то результат остается в рамках регистра AL. Если же CF = 1, то выполняется команда в строке 16, которая формирует в поле rez_h старшее слово результата. Команда в строке 18 формирует младшую часть результата. Теперь обратите внимание на сегмент данных, а именно на строку 6. В этой строке содержится директива label. Мы еще не раз будем сталкиваться с этой директивой. В данном случае она назначает еще одно символическое имя rez

адресу, на который уже указывает другой идентификатор rez_l. Различие заключается в типах этих идентификаторов — rez имеет тип слова, который ему назначается директивой label (имя типа указано в качестве операнда label). Введя эту директиву в программе, мы подготовились к тому, что, возможно, результат операции умножения будет занимать в памяти целое слово. Обратите внимание на то, что мы не нарушили принципа: младший байт по младшему адресу. Далее, используя имя rez, можно обращаться к значению в этой области как к слову.

В заключение можно исследовать в отладчике программу на разных наборах сомножителей.

Умножение двоичных чисел со знаком

Для умножения чисел со знаком предназначена команда

imul операнд_1[,операнд_2,операнд_3]

Эта команда выполняется так же, как и команда **MUL.** Отличительной особенностью команды IMUL является только формирование знака. Если результат мал и умещается в одном регистре (то есть если **CF= OF=** 0), то содержимое другого регистра (старшей части) является расширением знака — все его биты равны старшему биту (знаковому разряду) младшей части результата. В противном случае (если CF = OF = 1) знаком результата является знаковый бит старшей части результата, а знаковый бит младшей части является значащим битом двоичного кода результата. Если вы найдете в приложении команду **IMUL,** то увидите, что у нее имеются более широкие возможности по заданию местоположения операндов. Это сделано для удобства использования.

Деление двоичных чисел без знака

Для деления чисел без знака предназначена команда

div делитель

Делитель может находиться в памяти или в регистре и иметь размер 8, 16 или 32 бита. Местонахождение делимого фиксировано и так же, как в команде умножения, зависит от размера операндов. Результатом команды деления являются значения частного и остатка. Варианты местоположения и размеров операндов операции деления показаны в табл. 8.3.

Таблица 8.3. Расположение операндов и результата при делении

Делимое	Делитель	Частное	Остаток
Слово (16 бит) в регистре AX	Байт в регистре или в ячейке памяти	Байт в регистре AL	Байт в регистре AH
Двойное слово (32 бита), в DX — старшая часть, в AX — младшая часть	Слово (16 бит) в регистре или в ячейке памяти	Слово (16 бит) в регистре AX	Слово (16 бит) в регистре DX
Учетверенное слово (64 бит), в EDX — старшая часть, в EAX — младшая часть	Двойное слово (32 бита) в регистре или в ячейке памяти	Двойное слово (32 бита) в регистре EAX	Двойное слово (32 бита) в регистре EDX

После выполнения команды деления содержимое флагов неопределенно, но возможно возникновение прерывания с номером 0, называемого «деление на ноль». Этот вид прерывания относится к так называемым *исключениям* и возникает внутри процессора из-за некоторых аномалий в вычислительном процессе. К вопросу об исключениях мы еще вернемся. Прерывание 0 (деление на ноль) при выполнении команды DIV может возникнуть по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную под него разрядную сетку, что может случиться:
  - при делении делимого величиной в слово на делитель величиной в байт, причем значение делимого более чем в 256 раз больше значения делителя;
  - при делении делимого величиной в двойное слово на делитель величиной в слово, причем значение делимого более чем в 65 536 раз больше значения делителя; .
  - при делении делимого величиной в учетверенное слово на делитель величиной в двойное слово, причем значение делимого более чем в 4 294 967 296 раз больше значения делителя.

К примеру, выполним деление значения в области del на значение в области ***delt*** (листинг 8.6).

Листинг 8.6. Деление чисел

```
<1> ;prg_8.6.asm
<2> masm
<3> model   small
<4> stack    256
<5> .data
<6> del_b   label   byte
<7> del dw 29876
<8> deltdb  45
<9> .code                   ;сегмент кода
<10>    main:               ;точка входа в программу
<11>    ;...
<12>        xor ax,ax
<13>    ;последующие две команды можно заменить одной mov ax,del
<14>        mov ah,del_b    ;старший байт делимого в ah
<15>        mov al,del_b+1  ;младший байт делимого в al
<16>        div delt        ;в al - частное, в ah - остаток
<17>    ;...
<18>    end main            ;конец программы
```

Выполнение программы в таком виде приведет к ошибке деления на ноль. Причина описана ранее — частное не входит в отведенную под него разрядную сетку. Это происходит в случаях, когда делимое больше делителя на определенную величину. А что же делать, если соотношение делимое и делителя именно такое? Подробно деление, как, впрочем, и умножение, целых чисел произвольной разрядности описано в книге [8]. Чтобы исправить пример из листинга 8.6, необходимо изменить разрядность делимого, исходя из разрядности делителя и требований команды DIV. К примеру, делимое можно сделать равным 298. Пример будет выполнен без ошибок.

Деление двоичных чисел со знаком

Для деления чисел со знаком предназначена команда

idiv делитель

Для этой команды справедливы все рассмотренные ранее рассуждения, касающиеся команд и чисел со знаком. Отметим лишь особенности возникновения исключения 0 (деление на ноль) в случае чисел со знаком. Оно возникает при выполнении команды IDIV по одной из следующих причин:

- делитель равен нулю;
- частное не входит в отведенную для него разрядную сетку, что, в свою очередь, может произойти:
  - при делении делимого величиной в слово со знаком на делитель величиной в байт со знаком, причем значение делимого более чем в 128 раз больше значения делителя (таким образом, частное не должно

находиться вне диапазона от-128до+127);

- при делении делимого величиной в двойное слово со знаком на делитель величиной в слово со знаком, причем значение делимого более чем в 32 768 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от -32 768 до +32 768);
- при делении делимого величиной в учетверенное слово со знаком на делитель величиной в двойное слово со знаком, причем значение делимого более чем в 2 147 483 648 раз больше значения делителя (таким образом, частное не должно находиться вне диапазона от -2 147 483 648 до +2 147 483 647).

Вспомогательные команды для арифметических вычислений

В системе команд процессора есть несколько команд, которые облегчают программирование алгоритмов, производящих арифметические вычисления, а также помогают разрешать проблемы, возникающие при подобных вычислениях.

Команды преобразования типов

Что делать, если размеры операндов, участвующих в арифметических операциях, разные? Например, предположим, что в операции сложения один операнд занимает слово, а другой — двойное слово. Ранее было сказано, что в операции сложения должны участвовать операнды одного формата. Если числа без знака, то выход найти просто. В этом случае можно на базе исходного операнда сформировать новый операнд (формата двойного слова), старшие разряды которого просто заполнить нулями. Сложнее ситуация для чисел со знаком: как динамически, в ходе выполнения программы, учесть знак операнда? Для решения подобных проблем в системе команд процессора есть так называемые *команды преобразования типа.* Эти команды расширяют байты в слова, слова — в двойные слова и двойные слова — в учетверенные слова (64-разрядные значения). Команды преобразования типа особенно полезны при преобразовании целых со знаком, так как они автоматически заполняют старшие биты вновь формируемого операнда значениями знакового бита исходного объекта. Эта операция приводит к целым значениям того же знака и той же величины, что и исходная, но уже в более длинном формате. Подобное преобразование

называется *операцией распространения знака.*

Существуют два вида команд преобразования типа.

- ***Команды без операндов*** **—** эти команды работают с фиксированными регистрами:
  - CBW (Convert Byte to Word) — команда преобразования байта (в регистре AL) в слово (в регистре AX) путем распространения значения старшего бита AL на все биты регистра AH;
  - CWD (Convert Word to Double) — команда преобразования слова (в регистре AX) в двойное слово (в регистрах DX:AX) путем распространения значения старшего бита ax на все биты регистра DX;
  - CWDE (Convert Word to Double) — команда преобразования слова (в регистре AX) в двойное слово (в регистре EAX) путем распространения значения старшего бита AX на все биты старшей половины регистра EAX;
  - CDQ (Convert Double Word to Quarter Word) — команда преобразования двойного слова (в регистре EAX) в учетверенное слово (в регистрах EDX:EAX) путем распространения значения старшего бита EAX на все биты регистра EDX;

it ***Команды обработки строк*** (также глава 12). Эти команды обладают полезным свойством в контексте нашей проблемы:

- movsx операнд_1,операнд_2 — команда пересылки с распространением знака расширяет 8- или 16-разрядное значение операнд_2, которое может быть регистром или операндом в памяти, до 16- или 32-разрядного значения в одном из регистров, используя знаковый бит для заполнения старших позиций значения операнд_1 (данную команду удобно использовать для подготовки операндов со знаками к выполнению арифметических операций);
- movzx операнд_1,операнд_2 — команда пересылки с расширением нулем расширяет 8- или 16-разрядное значение операнд_2 до 16- или 32-разрядного с очисткой (заполнением) нулями старших позиций значения операнд_2 (данную команду удобно использовать для подготовки операндов без знака к выполнению арифметических действий).

К примеру, вычислим значение $y = (a + b)/c$, где a, b, c **—** байтовые знаковые переменные (листинг 8.7).

Листинг 8.7. Вычисление простого выражения

```
<1> ;prg_8_7.asm
<2> masm
<3> model    small
<4> stack    256
<5> .data
<6> a    db  5
<7> b    db  10
<8> c    db  2
<9> y    dw  0
<10>     .code
<11>     main:                ;точка входа в программу
<12>     ;...
<13>         xor ax,ax
<14>         mov al,a
<15>         cbw
     .386
<16>         movsx   bx,b
<17>         add ax,bx

<18>         idivc           ;в al - частное, в ah - остаток
<19>         exit:
<20>       mov ax,4c00h; стандартный выход
<21>         int 21h
<22>     end main         ;конец программы
```

В этой программе делимое для команды IDIV (строка 18) готовится заранее. Так как делитель имеет размер байта, то делимое должно быть словом. С учетом этого сложение осуществляется параллельно с преобразованием размера результата в слово (строки 14-17). Например, расширение операндов со знаком производится двумя разными командами — CBW и MOVSX.

Другие полезные команды

В системе команд процессора есть две команды — XADD и NEG, которые могут быть полезны, в частности, для программирования вычислительных действий:

- xadd **приемник, источник** — обмен местами и сложение. Команда позволяет выполнить последовательно два действия:

1. **обменять значения приемник и источник;**
2. **поместить на место операнда приемник сумму: приемник = приемник + источник.**

В *neg* операнд — отрицание с дополнением до двух. Команда выполняет инвертирование значения операнд. Физически команда выполняет одно действие: операнд = 0 - операнд, то есть вычитает операнд из нуля. Команду NEG можно применять для смены знака и вычитания из константы. Дело в том, что команды SUB и SBB не позволяют вычесть что-либо из константы, так как константа не может служить операндом-приемником в этих

операциях. Поэтому данную операцию можно выполнить с помощью двух команд:

```
neg  ax        ;смена знака (ax)
...
add ax,340  ;фактически вычитание: (ax)=340-(ax)
```

Арифметические операции над двоично-десятичными числами

Определение и формат BCD-чисел были рассмотрены в начале этой главы. У вас справедливо может возникнуть вопрос: а зачем нужны BCD-числа? Ответ может быть следующим: BCD-числа нужны в коммерческих приложениях, то есть там, где числа должны быть большими и точными. Как мы уже убедились, выполнение операций с двоичными числами для языка ассемблера довольно проблематично:

- Значения величин в формате слова и двойного слова имеют ограниченный диапазон. Если программа предназначена для работы в области финансов, то ограничение суммы в рублях величиной 65 536 (для слова) или даже 4 294 967 296 (для двойного слова) существенно сужает сферу ее применения (да еще в наших экономических условиях — тут уж никакая деноминация не поможет).
- Двоичные числа дают ошибки округления. Представляете себе программу, работающую где-нибудь в банке, которая не учитывает величину остатка при действиях с целыми двоичными числами и оперирует при этом миллиардами. Не хотелось бы быть автором такой программы. Применение чисел с плавающей точкой не спасет — там существует та же проблема округления.
- Большой объем результатов в коммерческих программах требуется представлять в символьном виде (ASCII-коде). Коммерческие программы не просто выполняют вычисления; одной из целей их применения является оперативная выдача информации пользователю. Для этого, естественно, информация должна быть представлена в символьном виде. Перевод чисел из двоичного кода в ASCII-код, как мы уже видели, требует определенных вычислительных затрат. Число с плавающей точкой еще труднее перевести в символьный вид. А вот если посмотреть на шестнадцатеричное представление неупакованной десятичной цифры (в начале данной главы) и на соответствующий ей символ в таблице ASCII, то видно, что они различаются на величину 30h. Таким образом, преобразование в символьный вид и обратно получается намного проще и

быстрее.

Эти доводы убеждают в важности овладения хотя бы основами действий с десятичными числами. Далее рассмотрим особенности выполнения основных арифметических операций с такими числами. Для предупреждения возможных вопросов сразу отметим тот факт, что отдельных команд сложения, вычитания, умножения и деления BCD-чисел нет. Сделано это по вполне понятным причинам — размерность таких чисел может быть сколь угодно большой. Складывать и вычитать можно двоично-десятичные числа как в упакованном формате, так и в неупакованном, а вот делить и умножать можно только неупакованные BCD-числа. Почему, будет видно из дальнейшего обсуждения.

Неупакованные BCD-числа

Сложение

Рассмотрим два случая сложения.

Результат сложения не больше 9:

6 = 0000 0110

+

3=0000 0011

=

9 = 0000 1001.

Переноса из младшей тетрады в старшую нет. Результат правильный.

Результат сложения больше 9:

06 = 0000 0110

+

07=00000111

=

13 = 0000 1101.

То есть мы получили уже не BCD-число. Результат неправильный. Правильный результат в неупакованном BCD-формате в двоичном представлении должен быть таким: 0000 0001 0000 0011 (или 13 в десятичном). Проанализировав данную проблему при сложении BCD-чисел (и подобные проблемы при выполнении других арифметических действий), а также возможные пути ее решения, разработчики системы команд процессора решили не вводить специальные команды для работы с BCD-числами, а ввести несколько корректировочных команд. Назначение этих команд — корректировка результата работы обычных арифметических

команд для случаев, когда операнды в них являются BCD-числами. В случае сложения во втором примере видно, что полученный результат нужно корректировать. Для коррекции операции сложения двух однозначных неупакованных BCD-чисел и представления результата сложения в символьном виде в системе команд процессора существует специальная команда AAA (ASCII Adjust for Addition).

Команда AAA не имеет операндов. Она работает неявно только с регистром AL и анализирует значение его младшей тетрады. Если это значение меньше 9, то флаг CF сбрасывается в 0 и осуществляется переход к следующей команде. Если это значение больше 9, то выполняются следующие действия.

1. К содержимому младшей тетрады AL (но не к содержимому всего регистра!) прибавляется *6*, тем самым значение десятичного результата корректируется в правильную сторону.
2. Флаг CF устанавливается в 1, тем самым фиксируется перенос в старший разряд для того, чтобы его можно было учесть в последующих действиях.

Так, во втором примере сложения, предполагая, что значение суммы 0000 1101 находится в AL, после выполнения команды AAA в регистре будет 1101 + 0110 = 0011, то есть двоичное значение 0000 0011 или десятичное значение 3, а флаг CF установится в 1, то есть перенос запоминается в процессоре. Далее программисту нужно использовать команду сложения ADC, которая учтет перенос из предыдущего разряда. Приведем пример программы сложения двух неупакованных BCD-чисел (листинг 8.8).

Листинг 8.8. Сложение неупакованных BCD-чисел

```
<1>     ;prg_8_8.asm
<2>     ;...
<3>     .data
<4>     len equ 2       ;разрядность числа
<5>     b   db  1,7     ;неупакованное число 71
<6>     c   db  4,5     ;неупакованное число 54
<7>     sum db  3 dup (0)
<8>     .code
<9>     main:           ;точка входа в программу
<10>    ;...
<11>        xor bx,bx
<12>        mov cx,len
<13>    m1:
<14>        mov al,b[bx]
<15>        adc al,c[bx]
<16>        aaa
<17>        mov sum[bx],al
<18>        inc bx
<19>        loop m1
<20>        adc sum[bx],0
<21>    ;...
        exit:
```

В листинге 8.8 есть несколько интересных моментов, над которыми стоит поразмыслить. Начнем с описания BCD-чисел. Из строк 5 и 6 видно, что порядок их ввода обратен нормальному, то есть цифры младших разрядов расположены по меньшему адресу. Это вполне логично по нескольким причинам: во-первых, такой порядок удовлетворяет общему принципу представления данных для процессоров Intel, во-вторых, это очень удобно для поразрядной обработки неупакованных BCD-чисел, так как каждое из них занимает один байт. Хотя, повторюсь, программист сам волен выбирать способ описания BCD-чисел в сегменте данных. Строки 14 и 15 содержат команды, которые складывают цифры в очередных разрядах BCD-чисел, при этом учитывается возможный перенос из младшего разряда. Команда AAA в строке 16 корректирует результат сложения, формируя в AL BCD-цифру и при необходимости устанавливая в 1 флаг CF. Строка 20 учитывает возможность переноса при сложении цифр из самых старших разрядов чисел. Результат сложения формируется в поле sum, описанном в строке 7.

Вычитание

Ситуация при вычитании вполне аналогична сложению. Рассмотрим те же случаи.

Результат вычитания не больше 9:

6 = 0000 0110

-

3=00000011

=

3= 00000011.

Как видим, заема из старшей тетрады нет. Результат верный и корректировки не требует.

Результат вычитания больше 9:

6 =00000110

-

7 =00000111

=

-1-11111111.

Вычитание проводится по правилам двоичной арифметики. Поэтому результат не является BCD-числом. Правильный результат в неупакованном BCD-формате должен быть 9 (0000 1001 в двоичной системе счисления). При этом предполагается заем из старшего разряда, как при обычной команде вычитания, то есть в случае с BCD-числами фактически должно быть

выполнено вычитание 16 - 7. Таким образом, как и в случае сложения, результат вычитания нужно корректировать. Для этого существует специальная команда AAS (ASCII Adjust for Substraction), выполняющая коррекцию результата вычитания для представления в символьном виде.

Команда AAS также не имеет операндов и работает с регистром AL, анализируя его младшую тетраду следующим образом: если ее значение меньше 9, то флаг CF сбрасывается в 0 и управление передается следующей команде. Если значение тетрады в AL больше 9, то команда AAS выполняет следующие действия.

1. Из содержимого младшей тетрады регистра AL (заметьте, не из содержимого

всего регистра) вычитается 6.

2. Старшая тетрада регистра AL обнуляется.

3. Флаг CF устанавливается в 1, тем самым фиксируется воображаемый заем из старшего разряда.

Понятно, что команда AAS применяется вместе с основными командами вычитания SUB и SBB. При этом команду SUB есть смысл использовать только один раз при вычитании самых младших цифр операндов, далее должна применяться команда SB B, которая будет учитывать возможный заем из старшего разряда. В листинге 8.9 мы обходимся одной командой SBB, которая в цикле производит поразрядное вычитание двух BCD-чисел.

Листинг 8.9. Вычитание неупакованных BCD-чисел

```
<1> ;prg_8_9.asm
<2> masm
<3> model    small
<4> stack    256
<5> .data                       ;сегмент данных
<6> b    db  1,7                ;неупакованное число 71
<7> c    db  4,5                ;неупакованное число 54
<8> subs db 2 dup (0)
<9> .code
<10>    main:                   ;точка входа в программу
<11>        mov ax,@data        ;связываем регистр dx с сегментом
<12>        mov ds,ax           ;данных через регистр ax
<13>        xor ax,ax           ;очищаем ax
<14>    len equ 2               ;разрядность чисел
<15>        xor bx,bx
<16>        mov cx,len          ;загрузка в cx счетчика цикла
<17>    m1:
<18>        mov al,b[bx]
<19>        sbb al,c[bx]
<20>        aas
<21>        mov subs[bx],al
<22>        inc bx
<23>        loop    m1
<24>        jc  m2              ;анализ флага заема
<25>        jmp exit
<26>    m2: ;...
<27>    exit:
<28>        mov ax,4c00h        ;стандартный выход
<29>        int 21h
<30>    end main                ;конец программы
```

Данная программа не требует особых пояснений в случае, когда уменьшаемое больше вычитаемого. Поэтому обратите внимание на строку 24. С ее помощью мы предусматриваем случай, когда после вычитания старших цифр чисел был зафиксирован факт заема. Это говорит о том, что вычитаемое было больше уменьшаемого, в результате чего разность оказалась неправильной. Эту ситуацию нужно как- то обработать. С этой целью в строке 24 командой JC анализируется флаг CF. По результату этого анализа мы уходим на ветку программы, обозначенную меткой т2, где и выполняются нужные действия. Набор этих действий зависит от конкретного алгоритма обработки, поэтому поясним только их суть. Для этого посмотрите в отладчике, как наша программа выполняет вычитание 50 - 74 (правильный ответ - 24). То, что вы увидите в окне Dump отладчика (в поле, соответствующем адресу subs), окажется далеким от истины. Что делает в этом случае человек? Он просто мысленно меняет местами вычитаемое и уменьшаемое (74 - 50 = 24), а результат рассматривает со знаком «минус». Так и фрагмент программы, обозначенный меткой т2, может, поменяв уменьшаемое и вычитаемое местами и выполнив вычитание, где-то отметить тот факт, что разность, на самом деле, нужно рассматривать как отрицательное число.

Умножение

На примере сложения и вычитания неупакованных чисел мы выяснили, что стандартных алгоритмов для выполнения этих действий над BCD-числами нет и программист должен сам, исходя из требований к своей программе, реализовать эти операции. Реализация двух оставшихся операций — умножения и деления — еще сложнее. В системе команд процессора присутствуют только средства для умножения и деления одноразрядных неупакованных BCD-чисел. Для их умножения необходимо воспроизвести описанную далее процедуру.

1. Поместить один из сомножителей в регистр **AL** (как того требует команда MUL).
2. Поместить второй сомножитель в регистр или память, отведя для него байт.
3. Перемножить сомножители командой **MUL** (результат, как и положено, окажется в регистре AX).
4. Скорректировать результат, который, конечно, будет представлен в двоичном коде.

Для коррекции результата после умножения в целях представления его в символьном виде применяется специальная команда **AAM** (ASCII Adjust for Multiplication). Она не имеет операндов и работает с регистром AX следующим образом.

1. Делит **AL** на 10.
2. Результат деления записывается так: частное — в AL, остаток — в AH.

В результате после выполнения команды MM в регистрах AL и AH находятся правильные двоично-десятичные цифры произведения двух цифр.

В листинге 8.10 приведен пример умножения BCD-числа произвольной размерности на однозначное BCD-число.

Листинг 8.10. Умножение неупакованных BCD-чисел

```
masm
model   small
stack   256
 .data
b   db  6,7             ;неупакованное число 76
c   db  4               ;неупакованное число 4
proizv  db  4 dup  (0)
 .code
main:                   ;точка входа в программу
        mov ax,@data
        mov ds,ax
        xor  ax,ax
    len equ 2           ;размерность сомножителя 1
        xor  bx,bx
        xor si,si
        xor di,di
        movcx,len       ;в cx длина наибольшего сомножителя 1
    m1:
        moval,b[si]

        mul c
        aat              ;коррекция умножения
        adc al,dl        ;учли предыдущий перенос
        aaa              ;скорректировали результат сложения с переносом
        mov dl,ah        ;запомнили перенос
        mov proizv[bx],al
        inc si
        inc bx
        loop m1          ;цикл на метку m1
        mov proizv[bx],dT    ;учли последний перенос
        exit:
        mov ax,4c00h
        int 21h
    end  main
```

Данную программу можно легко модифицировать для умножения BCD-чисел произвольной длины. Для этого достаточно представить алгоритм умножения в «столбик». Листинг 8.10 можно использовать для получения частичных произведений в этом алгоритме. После их сложения со сдвигом получится искомый результат. Попробуйте написать эту программу самостоятельно. Если же данная задача окажется для вас непосильной, ее решение вы можете найти в [8].

Перед окончанием обсуждения команды MM необходимо отметить, что ее можно применять для преобразования двоичного числа в регистре AL в

неупакованное BCD-число, которое окажется в регистре AX: старшая цифра результата — в AH, младшая — в AL. Понятно, что двоичное число должно быть в диапазоне 0...99.

Деление

Процесс деления двух неупакованных BCD-чисел несколько отличается от других рассмотренных ранее операций с ними. Здесь также требуются действия по коррекции, но они должны выполняться до основной операции, выполняющей непосредственно деление одного BCD-числа на другое BCD-число. Предварительно в регистре AX нужно получить две неупакованные BCD-цифры делимого. Это делает программист удобным для него способом. Далее для коррекции результата деления в целях представления его в символьном виде нужно выполнить команду AAD (ASCII Adjust for Division).

Команда AAD не имеет операндов и преобразует двузначное неупакованное BCD- число в регистре AX в двоичное число, которое впоследствии будет играть роль делимого в операции деления. Кроме преобразования, команда AAD помещает полученное двоичное число в регистр AL. Делимое, естественно, является двоичным числом из диапазона 0...99. Алгоритм, по которому команда AAD осуществляет это преобразование, выглядит следующим образом.

1. Умножить на 10 старшую цифру исходного BCD-числа в AX (содержимое AH).
2. Выполнить сложение AH + AL, результат которого (двоичное число) занести в AL.
3. Обнулить содержимое AH.

Далее программисту нужно выполнить обычную команду деления DIV для деления содержимого AX на одну BCD-цифру, находящуюся в байтовом регистре или байтовой ячейке памяти. Деление неупакованных BCD-чисел иллюстрирует листинг 8.11.

Листинге 8.11. Деление неупакованных BCD-чисел

```
<1> ;prg_8_11.asm
<2> ...
<3>  .data              ;сегмент данных
<4> b   db  1,7         ;неупакованное BCD-число 71
<5> c   db  4
<6>  .code              ;сегмент кода
<7> main:              ;точка входа в программу
<8> ...
<9>         mov al,b
<10>         aad        ;коррекция перед делением
<11>         div c      ;в al BCD – частное,  в ah BCD - остаток
<12>     ...
<13>        exit:
```

Аналогично AAM, команде AAD можно найти и другое применение — использовать ее для перевода неупакованных BCD-чисел из диапазона 0...99 в их двоичный эквивалент.

Для деления чисел большей разрядности так же, как и в случае умножения, нужно реализовывать, например, алгоритм деления в «столбик» или найти более оптимальный путь. Любопытный и настойчивый читатель, возможно, самостоятельно разработает эти программы. Но это делать совсем необязательно. Среди файлов, прилагаемых к книге, в каталоге данной главы приведены тексты макрокоманд, которые выполняют четыре основных арифметических действия с BCD- числами любой разрядности. Кроме этого в [8] вы найдете дополнительный материал по этому вопросу.

Упакованные BCD-числа

Как уже отмечалось ранее, упакованные BCD-числа можно только складывать и вычитать. Для выполнения других действий над ними их нужно дополнительно преобразовывать либо в неупакованный формат, либо в двоичное представление. Таким образом, сами по себе упакованные BCD-числа представляют не слишком большой интерес для программиста, поэтому мы их рассмотрим кратко.

Сложение

Вначале разберемся с сутью проблемы и попытаемся сложить два двузначных упакованных BCD-числа:

67 =01100111
+
75 =01110101
=
142= 1101 1100 = 220

Как видим, в двоичном виде результат равен 1 101 1100 (или 220 в десятичном представлении), что неверно. Это происходит по той причине, что процессор не подозревает о существовании BCD-чисел и складывает их по правилам сложения двоичных чисел. На самом деле результат в двоично-десятичном виде должен быть равен 0001 0100 0010 (или 142 в десятичном представлении). Этот пример иллюстрирует необходимость корректировки результатов арифметических операций с упакованными BCD-числами, так же как это было в случае неупакованных BCD-чисел. Для корректировки результата сложения упакованных чисел в целях представления его в десятичном виде процессор предоставляет команду DAA (Decimal Adjust for

Addition).

Команда DAA преобразует содержимое регистра AL в две упакованные десятичные цифры (по алгоритму, приведенному в приложении А, где данная команда описана более подробно). Получившаяся в результате сложения единица (если результат сложения больше 99) запоминается во флаге CF, тем самым учитывается перенос в старший разряд.

Проиллюстрируем сказанное на примере сложения двух двузначных BCD-чисел в упакованном формате (листинг 8.12).

Листинг 8.12. Сложение упакованных BCD-чисел

```
<1> ;prg_8_12.asm
<2> ...
<3> .data                       ;сегмент данных
<4> b   db  17h                 ;упакованное число 17
<5> c   db  45h                 ;упакованное число 45
<6> sum db  2 dup (0)
<7> .code                       ;сегмент кода
<8> main:                       ;точка входа в программу
<9> ...
<10>        mov al,b
<11>        add al,c
<12>        daa
<13>        jnc $+6             ;переход через команду, если результат <= 99
<14>        mov sum+1,ah        ;учет переноса при сложении (результат > 99)
<15>        mov sum,al          ;младшие упакованные цифры результата
<16>    exit:
```

В приведенном примере все достаточно прозрачно; единственное, на что следует обратить внимание, — это описание упакованных BCD-чисел и порядок формирования результата. Результат формируется в соответствии с основным принципом работы процессоров Intel: младший байт по младшему адресу.

Вычитание

Аналогично сложению, при вычитании процессор рассматривает упакованные BCD-числа как двоичные. Выполним вычитание 67 - 75. Так как процессор выполняет вычитание способом сложения, то и мы последуем этому:

67 =01100111

+

-75 =10110101

=

-8 =0001 1100 = 28.

Как видим, результат равен 28 в десятичной системе счисления, что является абсурдом. В двоично-десятичном коде результат должен быть равен 0000 1000 (или 8 в десятичной системе счисления). При программировании

вычитания упакованных BCD-чисел программист, как и при вычитании неупакованных BCD-чисел, должен сам осуществлять контроль за знаком. Это делается с помощью флага CF, который фиксирует заем из старших разрядов. Само вычитание BCD-чисел осуществляется обычной командой вычитания SUB или SBB. Коррекция результата вычитания для его представления в десятичном виде осуществляется командой DAS (Decimal Adjust for Substraction).

В приложении описан алгоритм, по которому команда DAS преобразует содержимое регистра AL в две упакованные десятичные цифры.

Итоги

- Процессор имеет довольно мощные средства для реализации вычислительных операций. Для этого у него есть блок целочисленных операций и блок операций с плавающей точкой. Для большинства задач, использующих язык ассемблера, достаточно целочисленной арифметики.
- Команды целочисленных операций работают с данными двух типов: двоичными и двоично-десятичными числами (BCD-числами).
- Двоичные данные могут либо иметь знак, либо не иметь такового. Процессор, на самом деле, не различает числа со знаком и без. Он лишь помогает отслеживать изменение состояния некоторых битов операндов и состояние отдельных флагов. Операции сложения и вычитания чисел со знаком и без знака проводятся одним устройством и по единым правилам.
- Контроль за правильностью результатов и их надлежащей интерпретацией полностью лежит на программисте. Он должен контролировать состояние флагов CF и OF регистра EFLAGS во время вычислительного процесса.
- Для операций с числами без знака нужно контролировать флаг CF. Установка его в 1 сигнализирует о том, что число вышло за разрядную сетку операндов.
- Для чисел со знаком установка флага OF в 1 говорит о том, что в результате сложения чисел одного знака результат выходит за границу допустимых значений чисел со знаком в данном формате, и сам результат меняет знак (пропадает порядок).
- По результатам выполнения арифметических операций устанавливаются также флаги PF, ZF и SF.
- В отличие от команд сложения и вычитания, команды умножения и деления позволяют учитывать знаки операндов.

- Арифметические команды очень «капризны» к размерности операндов, поэтому в систему команд процессора включены специальные команды, позволяющие отслеживать эту характеристику.
- Хотя диапазон значений двоичных данных довольно велик, для коммерческих приложений его явно недостаточно, поэтому в архитектуру процессора введены средства для работы с так называемыми двоично-десятичными (BCD) данными.
- Двоично-десятичные данные представляются в двух форматах, упакованном и неупакованном. Наиболее универсальным является неупакованный формат.